The N-Computer: A Data-Flow Bulk Synchronous Machine for Data Parallel Programs

A dissertation for the degree of Doctor of Philosophy

Muhammad Abdul Qadar
27 April 1995
Many thanks to God who is the source of all the intelligence and Creator of all the laws of universe with the prayer to increase our understanding of the universe and then use that knowledge to the betterment of human beings. For the guidance and assistance, I thank my supervisor, Chris Jesshope. Without his guidance, it would have not been possible to submit a thesis on such an important topic. His assistance which enabled me to focus my wandering thoughts in proper directions, will always be remembered. I also express my gratitude to all my teachers from my first school to the university for their guidance and assistance.

I am grateful to the Ministry of Science & Technology, Government of Pakistan for their financial assistance. I appreciate all the members of the Computer Systems Research Group for their cooperation and friendship. Most importantly, the developers of F-code, Slava Muchnick, A. Shafarenko and A. Bolychevsky, and Carl Sutton and Dima Barsky for their long extending cooperation.

For love, guidance and support from my parents throughout my life, I am beholden and unable to pay thanks. Last but not least, I am indebted to the support and love from my wife, Asma, and my sons Ahsun and Abdullah.
# Contents

0.1 Abstract ................................................................. vi

1 An Intelligent Approach for Parallel Computing and Background Study  1
  1.1 What Should be the approach? .................................. 1
  1.2 Software and Software Model ................................... 4
  1.3 Parallel Machine; Specifications .............................. 9
  1.4 Conventional Approaches to Build Parallel Computers ...... 10
  1.5 Data-Flow Approach to Build Computers ..................... 17
    1.5.1 First Generation Data Flow Machine .................... 17
    1.5.2 Second Generation Data-Flow Machines .................. 19
    1.5.3 Problems with the 2nd Generation Dataflow Machines ... 21
    1.5.4 Third Generation Data-Flow Machines ................... 22
  1.6 How Should the Parallel Machine be Built? .................. 22
  1.7 VBSP model and DFBSM model .................................. 24
  1.8 Organization of the Thesis ..................................... 27

2 Portable Software Platforms: Critique, Comparison and Suggestions  29
  2.1 Introduction ..................................................... 29
  2.2 F-code: A PSP .................................................. 31
  2.3 VCODE: A PSP .................................................. 32
  2.4 Other PSPs ..................................................... 34
  2.5 F-code; A Critique ............................................. 34
    2.5.1 Procedural Support in F-code ............................. 35
    2.5.2 Array Operations and F-code ............................. 37
  2.6 Conclusion and Suggestions ................................... 39

3 The F-code Simulator; F-SIM  42
  3.1 Introduction ..................................................... 42
  3.2 Analysis Phase .................................................. 42
  3.3 Synthesis Phase .................................................. 44
    3.3.1 Evaluation of Non-Geometric Functions .................. 45
    3.3.2 Evaluation of Semi-Geometric Functions ................ 47
    3.3.3 Evaluation of Geometric Functions ....................... 49
  3.4 Summing Up ...................................................... 54

4 The N-Computer and its Processing Phase  55
  4.1 Introduction ..................................................... 55
  4.2 The N-Computer .................................................. 55
  4.3 Node Structure of the N-Computer ............................. 57
C.1.4 Optimized Slice Function for Regular Pattern .................................................. 208
C.1.5 Optimized Slice Function for Irregular Pattern With Mask ................................... 209
C.1.6 The Function Repl ................................................................................................. 210
C.1.7 The Function Comp .............................................................................................. 212
C.1.8 The Function Comp (Length Coercion) ................................................................. 213
C.1.9 The Function Transp ............................................................................................ 214
C.1.10 The Function Diag ............................................................................................. 214
C.1.11 The Function Pack ............................................................................................. 215
C.1.12 The Distribution Function .................................................................................. 216
C.1.13 The Inverse of Distribution Function .................................................................. 217
C.1.14 The Function GET_DISP .................................................................................. 220

D  The N-Computer; OTS Initialization and Optimizations .............................................. 221
  D.1 Structure of an Activity (Non-Geometric) for OTS Initialization ............................... 221
  D.2 Initialization of OTS ............................................................................................... 223
    D.2.1 OTS Initialization Scheme 1 (SPN) .................................................................... 224
    D.2.2 OTS Initialization Scheme 2 (SPC) .................................................................... 227
    D.2.3 OTS Initialization Scheme 3 (SBN) .................................................................... 228
    D.2.4 OTS Initialization Scheme 4 (SBC) .................................................................... 230
    D.2.5 OTS Initialization Scheme 5 (CPN) .................................................................... 232
    D.2.6 OTS Initialization Scheme 6 (CPC) .................................................................... 234
    D.2.7 OTS Initialization Scheme 7 (CBN) .................................................................... 236
    D.2.8 OTS Initialization Scheme 8 (CBC) .................................................................... 237
  D.3 STE Initialization .................................................................................................... 238

E  Some Detailed Examples ............................................................................................... 240
  E.1 Reduce Without The Counter ................................................................................... 240

F  Analysis Phase of F-SIM ............................................................................................. 244

G  Logic for the Simulation of F-code Functions ................................................................ 248

H  Definition of f-code .................................................................................................... 258
  H.1 Objects .................................................................................................................. 258
  H.2 Representation of the f-program ............................................................................. 259
  H.3 Identifiers ............................................................................................................... 259
  H.4 Literals .................................................................................................................... 260
    H.4.1 Constants .......................................................................................................... 260
    H.4.2 Operators ......................................................................................................... 262
    H.4.3 Keywords ......................................................................................................... 265
    H.4.4 Function Labels .............................................................................................. 265
  H.5 Evaluation of the f-program .................................................................................... 265
  H.6 Functions ............................................................................................................... 265
    H.6.1 CONST ............................................................................................................. 267
    H.6.2 HOLD ............................................................................................................. 268
    H.6.3 TEMPLATE ..................................................................................................... 268
    H.6.4 CREATE ......................................................................................................... 269
    H.6.5 VAR ............................................................................................................. 269
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.6.6</td>
<td>SELECT</td>
<td>270</td>
</tr>
<tr>
<td>H.6.7</td>
<td>GLOBAL</td>
<td>271</td>
</tr>
<tr>
<td>H.6.8</td>
<td>DISPOSE</td>
<td>271</td>
</tr>
<tr>
<td>H.6.9</td>
<td>LOCAL</td>
<td>272</td>
</tr>
<tr>
<td>H.6.10</td>
<td>MARK</td>
<td>272</td>
</tr>
<tr>
<td>H.6.11</td>
<td>RAMP</td>
<td>272</td>
</tr>
<tr>
<td>H.6.12</td>
<td>MONADIC</td>
<td>273</td>
</tr>
<tr>
<td>H.6.13</td>
<td>DYADIC</td>
<td>273</td>
</tr>
<tr>
<td>H.6.14</td>
<td>CHOICE</td>
<td>273</td>
</tr>
<tr>
<td>H.6.15</td>
<td>REDUCE</td>
<td>274</td>
</tr>
<tr>
<td>H.6.16</td>
<td>TRANSP</td>
<td>274</td>
</tr>
<tr>
<td>H.6.17</td>
<td>SECT</td>
<td>274</td>
</tr>
<tr>
<td>H.6.18</td>
<td>SLICE</td>
<td>275</td>
</tr>
<tr>
<td>H.6.19</td>
<td>REPL</td>
<td>275</td>
</tr>
<tr>
<td>H.6.20</td>
<td>PACK</td>
<td>276</td>
</tr>
<tr>
<td>H.6.21</td>
<td>GATHER</td>
<td>276</td>
</tr>
<tr>
<td>H.6.22</td>
<td>DIAG</td>
<td>277</td>
</tr>
<tr>
<td>H.6.23</td>
<td>TRANSFORM</td>
<td>277</td>
</tr>
<tr>
<td>H.6.24</td>
<td>PART</td>
<td>277</td>
</tr>
<tr>
<td>H.6.25</td>
<td>COMP</td>
<td>278</td>
</tr>
<tr>
<td>H.6.26</td>
<td>POL</td>
<td>278</td>
</tr>
<tr>
<td>H.6.27</td>
<td>DISPLACE</td>
<td>279</td>
</tr>
<tr>
<td>H.6.28</td>
<td>DISTANCE</td>
<td>279</td>
</tr>
<tr>
<td>H.6.29</td>
<td>TYPE</td>
<td>279</td>
</tr>
<tr>
<td>H.6.30</td>
<td>SHAPE</td>
<td>280</td>
</tr>
<tr>
<td>H.6.31</td>
<td>ASSIGN</td>
<td>280</td>
</tr>
<tr>
<td>H.6.32</td>
<td>CHANNEL</td>
<td>280</td>
</tr>
<tr>
<td>H.6.33</td>
<td>PUT</td>
<td>281</td>
</tr>
<tr>
<td>H.6.34</td>
<td>GET</td>
<td>281</td>
</tr>
<tr>
<td>H.6.35</td>
<td>SEQ</td>
<td>281</td>
</tr>
<tr>
<td>H.6.36</td>
<td>COMMA</td>
<td>281</td>
</tr>
<tr>
<td>H.6.37</td>
<td>PAR</td>
<td>282</td>
</tr>
<tr>
<td>H.6.38</td>
<td>LOOP</td>
<td>282</td>
</tr>
<tr>
<td>H.6.39</td>
<td>SPAWN</td>
<td>282</td>
</tr>
<tr>
<td>H.6.40</td>
<td>IF</td>
<td>283</td>
</tr>
<tr>
<td>H.6.41</td>
<td>CALL</td>
<td>283</td>
</tr>
<tr>
<td>H.6.42</td>
<td>COERCCE</td>
<td>283</td>
</tr>
</tbody>
</table>

H.7 Examples .................................................. 284
0.1 Abstract

There are two fundamental problems which a researcher in the field of general purpose parallel computing is facing. One is to develop a framework for portable and easy to write parallel applications. The other is to design a scalable parallel machine upon which the parallel applications can be executed efficiently and economically.

To solve the first problem, a portable software platform (PSP) for data parallel high level languages like Fortran 90, F-code, has been developed by the members of the Computer Systems Research Group at the University of Surrey. F-code can be considered as a machine independent intermediate level representation of a data parallel program. This thesis addresses the second issue, i.e., the design of a scalable parallel machine, the N-computer, while critically examining and then refining the definition of F-code.

F-code is considered as a problem domain for the N-computer. Two extremes of the design space for a parallel machine, i.e., the conventional von-Neumann and dataflow, are not suitable to implement a design of the machine according to the specifications developed. The conventional approach is unable to address the two major issues of a parallel machine, i.e., synchronization and latency, at the same time. When a solution to solve one problem is found, the other problem becomes worse. Evolution in the design of dataflow machines has solved the major problems facing those kinds of architectures, such as the implementation of a waiting/matching store and scheduling of macroactors instead of single instructions. By scheduling a macroactor which contains code for more than one instruction for which there is no remote communication, the need for explicit synchronization can be eliminated. However, the problem of token recycling and the need for the I-structure storage to access variables, which need a three state synchronizer for each of its operation, are still remained unsolved. Both are quite expensive in the modern dataflow machines. It is observed that the dataflow bulk synchronous machine (DFBSM) model can address these two problems.

The N-computer is designed on top of the DFBSM model for data parallel languages. The implementation not only enjoys the benefits of DFBSM, but is also favoured with the regular struture which is normally available in a data parallel application. The design of the N-computer is described at system level. Each processing node is broken down into six major sub-systems. The logical organization of each sub-system and its functioning and the interaction with other sub-systems are discussed. The addition of certain features in the organization of these sub-systems is argued. Then the ways to execute each F-code function are discussed. In other words, this thesis presents the system level design and functioning of the N-computer for its complete operation cycle, and demonstrates the ways to execute an F-code program on an N-computer.

A plan to build soft prototype of the N-computer system is presented. A major portion
of the plan is implemented in software. A simulation of data distribution strategies is also implemented and presented. Due to different communication patterns in programs, it is not possible to statically fix the distribution strategy for optimal execution of all the programs. Different programs may need to be distributed in different ways. By simulating different distribution strategies for a given program, an optimum way to distribute objects can be selected. The N-computer is capable of dynamically adopting a distribution strategy for each activity in an activity tree depending upon the information available in the activity.

An important phase for the design of the N-computer was a thorough understanding of the problem domain, i.e., F-code. For that purpose, an F-code interpreter, F-SIM, was implemented in C. The implementation of F-SIM is described. F-code is also examined as a PSP for data parallel high level languages and compared with other potential PSPs. A PSP must preserve certain features available in a data parallel application to preserve the efficiency of an implementation. Although, F-code preserves most of them, there is a need to preserve primitive type sizes. To see whether F-code semantics are powerful enough to compile data parallel high level languages, like Fortran 90, the definition of F-code is examined qualitatively. In the light of this study, a modified definition of F-code functions is suggested.
THESIS ORGANIZATION AND READING GUIDE

Figure 0.1: This figure shows the organization of the thesis and a recommended guide to read the thesis. In the start, different questions are raised. Then to get answers to the questions raised, a reading sequence is recommended which is represented by the solid bold lines in the diagram. All the questions should not be raised in the start. A question may be raised after getting the answer of the previously raised question which is represented by the broken line. Although the division of chapter 1 into two parts for reading sequence is done on the basis that the sections in each part are more closely related to a particular topic, it is recommended to read the first chapter as a whole before going any further.
Chapter 1

An Intelligent Approach for Parallel Computing and Background Study

1.1 What Should be the approach?

Parallel computing is widely agreed to be a technology required to develop high performance computer systems which can solve today’s challenging problems at an affordable price. The developments in VLSI technology and the availability of low cost processors have made it feasible to interconnect many processing nodes in a parallel computer. With extensive research in building parallel computers and parallel software, in certain domains, the technology has been developed to build parallel computers for commercial purposes. However, the results of the research has not yet been converged into a unified approach for building parallel machines and solving problems upon them. The parallel computer research community has been aware of this problem and is trying to find the solution to it.

Valiant from the Harvard University writes in his famous paper, *A bridging model for parallel computation* [95],

> Due to the rapidly decreasing cost of processing, memory and communication, it has appeared inevitable for at least two decades that parallel machines will eventually displace sequential ones in computationally intensive domains. This, however, has not yet happened. In order to have a chance of rectifying this situation it is necessary to identify what is missing in our understanding of parallel computation that is present in the sequential case, making possible a huge and diverse industry.

According to Valiant, the missing thing is the central unifying model for parallel computing, which is present for sequential computing. In the same paper, then he claims,

> Our claim is that what is required before general purpose parallel computation
can succeed is the adoption of an analogous unifying bridging model for parallel computation.

If we draw a relation diagram between such a bridging model and the two major phases of a computer system, i.e., hardware and software, for sequential computing, it looks like as in Figure 1.1. Driving force for the development of hardware and software was the same unifying model which became a de facto standard and every one was developing hardware and software according to the unifying model. There is no such de facto standard for parallel computing. The diagnostic seems to be correct, but there is another issue one can identify; that is, the less diverse and slower rate of the development in the machine architecture and VLSI technology in the past. The development within the predominant domain of sequential machines around the unifying model has remained simple and cost effective until now. For parallel computing, developing such a scenario appears to be impossible. Diverse research in parallel computing and the ever growing complexity of VLSI technology are all indications that we may not be able to setup a similar framework to that used in the field of sequential computing. It can, also, be observed that everything developed in the field of computing around the unifying model of sequential computers is not necessarily developed in a way it should have been. For example, the development of high level languages in the sixties and seventies, reflects that sequential model. In fact, there is often a need for a programmer to write a loop for the addition of two matrices, instead s/he could have used a single instruction. If that would have been the case, we would not be re-writing the same applications for parallel computers, today. A significant amount of time and money has been spent in either re-writing the same applications for vector machines or developing the automatic parallelizing tools, e.g., CFT77 a Fortran77 compiler for Cray machines which vectorizes code [57]. Even
then, we did not learn the lesson, and most of the software developed for vector and parallel computers during the last decade was very specific to the particular machine for which it was developed. Therefore, we can say that the sequential model of a computer or a particular vector or parallel machine should not have been the driving force in the development of high level languages. It is a consequence of this approach that we now have non-portable software for our new generation of parallel distributed computers.

The main driving force behind such a unifying model was the technology to build computers, as shown in Figure 1.1. Since, the life time for machine hardware is significantly less than the life time of the software [72, 91], we could not develop software which would be valid for today's parallel/vector machines. To point out the reasons for such a wrong step in the development of high level languages and hence the software, one can say that we could not perceive the future. If someone perceived it [60], then s/he could not convince the industry to follow that approach or the industry did not feel any need to adopt the strategy of developing machine independent software. The approach for computing was not so diverse which could have broken the inertia as it is now. The other reason that the software (language) developers could not perceive the future, is also valid. One can point out that if the reason for not having a portable software was the perception of future, then how can we perceive the future today? We don't know what sort of machines will be available in 20 years time. The answer is to make the software completely machine independent with the main driving force is the nature of the problems. In this approach, the life time for the software will not be related with the life time of the hardware. Hence, for the solution of the same problems, there would be no need to re-write the applications for our future machines.

From this simple discussion, it can be inferred very easily that we need a machine independent model upon which we can develop high-level portable applications which can then be executed on a wide class of parallel machines without any significant loss of efficiency. That model is not necessarily the unifying or bridging model for parallel machines in the same sense as the sequential model has been for sequential machines. To know the congruence of algorithm for a certain class of parallel machines or to develop the parallel machines within certain restrictions implied by the current technology, we may need a machine related model for parallel computers. For such a model, the driving force could be the machine technology as well as the machine independent model for parallel software. Thus, instead of having a single unifying bridging model for the hardware as well as for the software development we need two models, one for software and one for hardware. Therefore, the relation diagram of Figure 1.1 should be transformed into the relation diagram of Figure 1.2 for parallel computing. The machine independent model to achieve portable and long life software, i.e., the life time of software is independent of the life time of machine architecture, for parallel
Figure 1.2: An ideal relation diagram for parallel computing. We need a machine independent model for the parallel software (PSP) and a machine related model for the development of parallel hardware and for an estimate cost of a parallel algorithm in PSP. The arrow lines show the driving force behind the development of each phase.

machines of today and tomorrow is named a PSP, i.e., Portable Software Platform. In the coming sections, the selection or formation of a PSP and a machine related model will be discussed.

1.2 Software and Software Model

The major driving force behind the development of computers is the desire to solve ever emerging problems faster, cheaper and accurately. Most of the problems to be solved on computers can be divided into two major classes, i.e., numerical problems and symbolic problems. Almasi and Gottlieb have differentiated between these two types of problems by the ratio of calculation to data motion among the processing elements [5]. In this thesis, we focus on numerical problems only. Scientific and engineering applications offer very large numerical computation, e.g.,

- global weather prediction,
- quantum chromodynamics, plasmas, particle calculations,
- seismic migration,
• blood flow in the heart,
• molecular dynamics,
• evolution of galaxies,
• automobile and aircraft design etc.

In the past, high-level languages like FortranIV [86], Fortran77 [73], C [66] and Pascal [96] etc., have been strongly influenced by the computer architecture or by the unifying model of sequential computing, which is rather a sequential machine model. Therefore, these languages are not suitable for writing programs for today’s parallel computers. Extending the argument of the previous section, if we develop languages which are strongly influenced by parallel computers available today, then we may again have to develop languages for tomorrow parallel computers and re-write code. Developing new languages and writing code is a very expensive process. Therefore, it may become difficult for the industry to market new innovative machines. Thus to develop software which is machine independent and to reduce the inertia in developing and marketing new parallel machines, we must adopt high level languages which are not strongly influenced by computer architectures and machine related models. The development of high level languages must be influenced by the nature of the problems.

Since, most problems in scientific computation rely at some point on the operation of matrix algebra, finite-difference methods and the evaluation of integrals [78, 5]. Therefore, the numerical methods developed to solve scientific problems inherit data parallelism. Also, the data parallelism is the major source of concurrency which can be exploited by a parallel machine to speed up the execution of a program. Therefore, the high level languages must preserve data parallelism in the solution for a problem. This fact has already been recognized [78, 91, 19] and data parallel languages such as EVAL [78], Fortran90 [74], HPF [46], NESL [20] and C* [83] have been developed.

When compared to EVAL and Fortran90, HPF is more influenced by the current state of the parallel machines, since it provides data distribution and alignment directives which could be different for a program running on different machines. The data distribution and alignment is directly related with the ratio of communication to processing power for a given machine. It is also affected by the interconnect topology of the machine. Thus, HPF restricts the portability of a program. It may also restrict the scalability of the system. Since, by changing the machine size we may have to re-write the program with new directives for data distribution and alignment. Thus it can be predicted that the HPF (or a subset of HPF which contains the data distribution and alignment directives) has less life time and
less portable as compared to Fortran90 or EVAL. Since, majority of scientific programming community has already been fortranized, therefore Fortran90 is probably going to be the major language for writing code for scientific and engineering applications in future.

Although, Fortran90 is emerging as a language for scientific community, there are still going to be many other languages, some of them are mentioned above, for writing programs for scientific and engineering applications. For a compiler writer, it is going to be a lengthy and sometimes a difficult process to provide full specification compilers for every language on a parallel machine. A parallel hardware designer does not want be concerned with all the languages issues when designing hardware. Therefore, we need a well defined unifying model for the high-level, parallel languages. The same thing is represented by PSP in Figure 1.2.

Skillicorn [91] has examined various models of parallel computation and discussed their strengths and weaknesses in adopting them as a unifying model for parallel computation. He has defined three key properties which must be possessed by a useful parallel computation model. I have added two more key properties, since, with these properties the aims mentioned in the previous paragraph, i.e., a well defined model for compiler writer and for a hardware designer can be achieved. The five points are as follows:

1. **Architectural Independence**: The model must be general enough to model several architectures and it must not have features which are specific to some particular type of architecture.

2. **Congruence**: The model should reflect, at the model level, the cost of the underlying execution.

3. **Simplicity**: The model should reduce the overheads of describing and managing massive parallelism.

4. **Preserve Parallelism**: The model must preserve parallelism available in the solution of a problem in a simple and easy to detect way.

5. **Well Defined**: The model must be well defined in its semantics so that there are no ambiguities in implementation.

The importance of the first point has already been discussed in the previous section. For the selection of the level of performance visible in the model for a given algorithm, the first criteria can be used. By plotting boundaries by the first criteria, the congruence of the model can be defined. One criteria can be selected as the product of the number of processors and the execution time [91]. Since, at model level we are interested in the high level or abstract performance not in the exact times, the order notations [67] should be used to abstract from constant factors which are machine dependent. The congruence is defined as [91],
A computation that in the model appears to take time \( T \) using processors \( P \) must be implementable on \( P' \) processors taking time \( T' \), where,

\[
T'.P' = O(T.P)
\]

if the model is to be congruent.

The definition of congruence may vary slightly in terms of providing the construct used for a PSP. The construct must not be too abstract to have no meaning in terms of execution time on a given machine, e.g., \( \lambda \)-calculus \([75, 12]\) is a model with no well defined operational semantics \([91, ?]\), it therefore is too abstract and lack congruence.

The third requirement means, the model should abstract out the explicit decomposition of parallel tasks, the details of communication and synchronization in an algorithm. Otherwise, the programming and the development of algorithm on top of that model is not only going to be very tedious and cumbersome, it will also make the algorithm dependent upon the machines. Although, there is no need to explicitly decompose parallel activities, the model must preserve the parallelism available in the solution of a problem. This can be done by properly defining the semantics of the model. If the parallelism is not preserved in the model then we may not be able to efficiently implement the model on parallel computers. Along with the main object of defining such a model, i.e., to achieve portability between many parallel machines (of today and tomorrow) and many parallel languages, it is also intended to provide a problem domain for the compiler writers and for the hardware designer, therefore, it must be well defined in its semantics.

Many models have been formulated and most of them have been examined by Skillicorn \([91]\) for the first three key properties as mentioned above. Here, to conclude the discussion the models of parallel computation which are used most prominently for studying algorithms and complexity are discussed. The first such model, probably, was the Random Access Machine (RAM) \([89, 3]\) formed for sequential computers. RAM model has been extended to Parallel Random Access Machine (PRAM) by Fortune and Wyllie \([57, 45]\) for modeling parallel computers. Although, the driving force behind the development of RAM and PRAM was to get an estimate of the performance of different algorithms for a class of computers which simulate these models. PRAM models idealized parallel computers with zero synchronization or memory access overheads \([57]\). An \( n \) processors PRAM has a globally addressable memory. The shared memory can be distributed among the processors or centralized in one place. The \( n \) processors operate on a synchronized read-memory, compute, and write-memory cycle. To address the concurrent read and concurrent write of a memory location, there has been four variants of PRAM \([57]\). One is Exclusive Read Exclusive Write (EREW) PRAM which forbids more than one processor from reading and writing the same
memory cell simultaneously [92, 65]. The second variant is the Concurrent Read Exclusive Write (CREW) in which the write conflicts in a PRAM machine are avoided but the concurrent reads to the same memory location are allowed. The third variant which allows Exclusive Read and Concurrent Write (ERCW) is analogous to a computer with multiple instruction streams and single data stream (MISD), according to the Flynn’s taxonomy [44], has generated little interest [5]. The fourth variant, CRCW-PRAM allows either concurrent reads or concurrent writes at the same time [57]. This last variant is also termed as paracomputer by Schwartz [87] and WRAM by Borodin and Hopcroft [27].

Skillicorn [91], notes;

Most parallel models fail by at least some of these criteria. For example the PRAM model is architecture independent, but is neither congruent nor good at hiding descriptive complexity. Programmers must manage the expression of parallelism, communication via shared memory and synchronization explicitly.

Data parallel models which have simple control flow such as sequence [22] or function composition [90] are quite attractive. SIMD-style operation contains substantial parallelism by applying a common sub-operation to many data. Major data types are lists and arrays. A significant amount of work has been done on data-parallel computing mostly originated from the language and compiler development for vector and SIMD machines. A more theoretical work about array theory [76, 77] and mathematics of arrays [91] has been trying to develop equational theories of arrays that can be used for transformation. Bird-Meerten formalisms [17, 18] have also provided a base for related data-parallel models. A Bird-Meerten theory begins with base types and extends them to new data parallel types using type functors.

Skillicorn [91], after studying the the data parallel models concludes;

All of the data parallel approaches are attractive because they do well by all three of the measures.

That paper concludes that the models based upon Bird-Meerten formalism and the theory of categorical data types developed by Malcolm [71] and Spivey [93] is a good starting point to develop a unifying model for parallel software.

In 1992, F-code [24] was developed as a candidate for such a unifying model and in comparison with Bird-Meerten formalism, it was observed that F-code has more potential to preserve the parallelism available in a data parallel program, i.e., the fourth key point as mentioned above. The concatenate operator in Bird-Meerten formalism, which concatenates and combines only adjacent elements is of restrictive nature [94] compared to an approach based on functions which could evaluate all elements concurrently with the least restrictive
trace. A functional approach to array is non-restrictive, since, every elemental operation on an array is a scalar operation supplied with an index. The same approach is adopted in F-code. Also, F-code is very well defined that with all the other features, it can be considered as a ready-made problem domain for a compiler writer and a parallel machine designer. Therefore, it can be stated that the F-code has a potential to be adopted as a unifying model for parallel software.

1.3 Parallel Machine; Specifications

Once F-code was selected as a problem domain, one of the aims of this work was to investigate a design for a scalable distributed memory machine using current hardware/machine technology. In this section, the top level specification of the parallel machine is highlighted.

For a parallel machine, there exists an optimum partitioning and scheduling of a program at which the overall execution time (processing, communication and synchronization time) will be minimum possible. That partitioning of the program is called the optimum grain of parallelism for the given architecture. It has been demonstrated that for a given program whose attributes are known at compile time, an optimum grain of parallelism for a given architecture can be found [85]. Accordingly the program can be compiled and scheduled. The size of the grain is dependent upon:

- The nature of the program (P), e.g., if an application demands more communication, then the grain size will be bigger for a given architecture.

- The characteristics of the hardware (S), e.g., methods of communication and synchronization supported by the distributed memory machine limit the grain size for a given application. There are physical limitations imposed by each component of the system which effect the grain size as well.

If we have a program \( P \) (to solve a problem \( \mathcal{P} \)) with all the attributes known statically, i.e., instruction sequencing and data dependencies can be worked out at compile time, and the set of system characteristics \( S \) of a parallel computer \( M \), then we can find a function \( F \), which will result in an optimal schedule for \( P \). The optimal schedule for the machine \( M \) is defined as the maximum speed-up achieved for the execution of the program \( P \) on the machine \( M \), as compared to execution time for the solution of the problem \( \mathcal{P} \) on a sequential (uniprocessor) machine. The speed-up divided by the cost of the machine (normalized by some costing factor) will give the efficiency factor for the machine.

\( S \) has two major parameters: physical limitations \( S_p \) imposed by the hardware such as the bandwidth of communication network etc.; and the organizational limitations \( S_o \).
imposed by the overall organization of the hardware, e.g., methods of communication and synchronization.

In our investigation, there are three major goals which we want to achieve:

1. To reduce $S$ to its minimum possible by using state of the art components (to reduce $S_p$) and formulate and/or collect brilliant ideas (to reduce $S_{C}$). With the reduction of $S$, we want to achieve a very fine grain optimal schedule or an efficiency factor which is very close to current uniprocessors.

2. In general, $F$ is very complex and when the attributes are not known statically, to evaluate $F$ at runtime is a great overhead [85]. In this design we want to reduce the complexity of $F$ or to determine some alternative mechanism which will keep the efficiency factor constant (or the losses in it should be very small) as compared to the case when the attributes of the program $P$ are known statically.

3. If there is enough parallelism available in the program $P$, then, by adding more hardware resources, the system performance must scale accordingly.

To carry out the design of such a system, which machine related model should we follow? Does any model exist, upon which we can construct the machine of our demands?

In answer to above questions: some (distributed) ideas exist but those are not confined to make a single robust model for our purpose. This will become clear with the discussion in the following sections. This thesis describes the design of such a machine, called the N-computer. In the coming sections, the current approaches to build such a machine will be examined. The conclusion reached is that a hybrid approach has the best potential to be adopted to build the machine of our specifications. A dataflow bulk synchronous model will then be defined as a machine related model for building the hybrid machine.

1.4 Conventional Approaches to Build Parallel Computers

Conventional approaches to build parallel computers are based upon the primitive model of computers, presented by John von-Neumann and his associates about forty years ago. The essential features of this model can be described as follows [5].

1. A processor that performs instructions such as “Add the contents of these two registers and put the result in that register”.
2. A control scheme that fetches one instruction after another from the memory for execution by the processor, and shuttles data between memory and processor one word at a time.

3. A memory that stores both the instructions and data of a program in cells having unique addresses.

The instructions are processed with a single control thread which eliminates the need for explicit synchronization. Forty years ago when the model was formed, processor and memory hardware were both very expensive and bulky that both were treated as precious resources\(^1\).

To examine the strengths and weaknesses of a von-Neumann machine, consider an example. In this example, assume two vectors A and B result from processing of certain expressions. Then the following expression is evaluated;

\[
\text{Reduce}_{\text{add}}(\text{Inverse}(A - B))
\]

Graphically, it is represented in Figure 1.3. A von-Neuman machine would execute one instruction at one time. If the time to execute one thread is \(T_{seq}\), then for \(K\) instructions, the time will be \(K \times T_{seq}\).

\(^1\)von-Neumann's IAS computer had 1 K 40-bits words of primary memory plus 16 K words of secondary (drum) storage [5].
For the execution of the expression, the sequence could be as follows:

1. For all the data elements in A and B;
   - Load $A_i$
   - Load $B_i$
   - Subtract them
   - Take inverse of the result
   - Store intermediate result in a register/memory

2. Load all the intermediate result one by one;
   - Add them

3. Store the final result

The execution time of an instruction ($T_{seq}$), in general, is a sum of the following quantities.

1. Instruction fetch, decode time.
2. Operand access time.
3. Processing time.
4. Result store time.

To reduce the number of clock cycles per instruction execution, complex instruction set (CISC) processor are replaced by reduced instruction set processors (RISC) [51, 47]. With the use of hard-wired control in RISC processors instead of micro-programmed control for CISC processors, the number of clock cycles per instruction has been reduced to one to two [57]. In the superscalar RISC processors, which allows multiple instruction to be issued simultaneously during each cycle, the effective number of clock cycles per instruction is less than that for the scalar RISC processors. To improve the access time, more registers are added in the processor. On chip and off chip caches are added either on the processor chip and on the processor board, respectively, details can be found in many good books about modern computer architecture, two of which are [57] and [34].

To speed up the computation further, parallelism in the machine architecture is introduced. Consider the execution time for the above example which is $T_{seq} \ast K$. Without the reduce operation, if all the threads are executed in parallel, then the execution time can be reduced by a factor of the extent of both vectors.
Vector Computers

In addition to fetching operands, in sequential computers instructions must also be fetched from the memory. Consider the above example of subtracting two vectors, if the vectors have 100 element each, at the machine language level of an ordinary sequential computer, the add instruction is also fetched and decoded 100 times. In a vector processor, this repeated fetching and decoding of the instruction for each pair of the matrix elements is replaced by a single vector instruction. Vector machines exploit the data concurrency to overcome the overheads for the execution of a vector instruction in a vector processor. On a vector machine, the addition of two vectors will still be a sequential process. The most famous vector computers are CRAY 1 [33] and its successors, CDC Cyber 205, NEC SX-X Series machines, Fujitsu VP-200 Series, IBM 3090 [57, 5, 54].

SIMD Computers

Instead of adding both vectors in a sequential way, these additions can be sent off to more than one execution unit and then can be performed in one cycle in parallel. An example of such a machine is ILLIAC IV [13] which was made about 25 years ago. ILLIAC IV has 64 processing units (PU) each consists of a processing element (PE) and a processor memory (PM). The PUs are connected with 2-dimensional mesh topology and are driven by the instructions from a common control unit. This type of computer is named as single instruction stream multiple data streams (SIMD) computer. Although, at the time of ILLIAC IV, SIMD machines were very expensive to build [5], later on SIMD machines proved to be very cost effective to achieve high performance for applications which match the machine structure. The 1993 Gordon Bell prize winner for price/performance was a SIMD machine which achieved 7.5 Gflops per million dollar [64]. Other SIMD machines are IBM GF11 [41], Thinking Machines Corp.'s connection machine CM-1 [52], CM-2 [31], Goodyear MPP [14] and AMT DAP Series [55].

SIMD machine operates in a lock step fashion with a single controller. Each processing element processes local data. If the data elements are not aligned, then the control unit will wait until the data become aligned. If we consider the above example again with the assumption that the two vectors are aligned. Then in one cycle, both vectors can be subtracted, in the next cycle the inverse can be taken. To reduce the result, the data elements will be moved from one processor memory to other processor memory. During the

---

2Instruction prefetching, pipelining and caching can remove this overhead significantly in favorable situations in a sequential machines [5].

3According to Flynn's taxonomy, vector machines are also SIMD, but in most of the literature these machines are treated separately. Sometimes, the vector machines are called pipelined SIMD and the SIMD machines with many processing elements are called parallel SIMD machines [5].

4As for as the processing concerns. However, it will issue an instruction to align the data.
data movement time, the processing element will be idle as for as the processing of the next instruction concerns.

A considerable amount of research is done in mapping and aligning the data to reduce the amount of communication for a given program [97, 43] on SIMD machines. The mapping must be chosen to keep data access conflicts to a minimum. The data mapping and alignment of data is a function of the application as well as the mapping technique, sometimes it may never be possible to completely eliminate conflicts.

In general, the amount of data parallelism in a program does not remain the same for all the instructions, as in the above example. To avoid communication, all the data elements can be mapped to one processor memory at the cost of destroying parallelism. In SIMD machine the restriction of data alignment at compile time is very strong, since, all the processing elements have to wait for the communication time if the alignment is done at run-time. That restriction can be relaxed upto some extent in an MIMD machine.

**MIMD Computers**

In an MIMD machine, each processing element has its own control unit. Therefore, as opposed to SIMD machine, all the processing elements do not operate in a lock step fashion. The program to be computed is broken into processes or tasks which will be distributed to the processors for execution. There have been two main approaches for sharing data among tasks namely *message-passing* and *shared-memory* [5]. The programming of a message passing system is much more difficult than that for the shared memory system. Therefore, from a user's viewpoint, a shared memory system is more attractive. The hardware for a distributed memory system is however, simpler and cheaper. To offer the simpler programming approach with a cheaper hardware, *virtual shared memory* machines have been built. A typical virtual shared memory machine is build by using a physical distributed memory hardware and a combination of hardware and software to provide the shared memory at the programming level.

Examples of distributed memory machines are as follows:

- Cosmic Cube [88] with 64 computing nodes connected in a six-dimensional hypercube topology.
- A commercial version of the Cosmic Cube, Intel iPSC/860 with processing nodes between 32 and 128 [5].
- The NCUBE/ten with upto 1024 processors [42].
- Intel Paragon XP/S with upto 4096 processors connected in a rectangular mesh pattern [5].
Transputer based machines, such as Meiko Computing Surface [56].

CM-5 from Thinking Machine Corporation [32] with 16 K processing nodes system [5]. The interconnection topology is a fat tree. The CM-5 is a synchronized MIMD machine to support both SIMD and MIMD style of processing. Vector units are also available at each node. The machine can be considered as a hybrid vector, SIMD and MIMD approaches. The CM-5 supports a data parallel model of computing using a subset of Fortran90 called CMFORTTRAN.

An approach to build shared memory MIMD computers was to interconnect many processing modules with many memory modules with processors on the one side of the interconnect and the memory on the other side, e.g., NYU ultracomputer [38]. Two major limitations of this approach were the cost of the interconnection network and the scalability. To overcome these limitations, other approaches to build shared memory MIMD computers were adopted. These machines can be considered virtual shared memory computer supported with some special hardware and/or software using physically distributed memory. These machines usually have NUMA characteristics. Some examples are as follows:

- Kendall Square Research KSR1 is a virtual shared memory MIMD computer with private consistent caches, i.e., each processor has its own cache and the system hardware guarantees that the multiple caches are kept in agreement [30, 84] (a COMA machine).

- The Stanford DASH (Distributed Architecture for SHared memory) is a shared memory computer with hardware cache coherence [39]. Unlike the KSR1, DASH was a research machine which has been used to study cache coherence, applications programming, simulation techniques and weak models for memory consistency [5].

- The MIT Alewife machine with distributed shared memory and hardware-coherent caches [37] is also a virtual shared memory research machine.

- Cray T3D is a large scale commercial machine with some special hardware which make it possible to support SIMD, message-passing MIMD and also shared memory MIMD programming models [59].

For all the machines mentioned earlier in this section, the underlying model of computation is the von-Neumann model. The execution of an instruction is still guided by the program counter. The instruction is fetched, decoded and executed after loading the operands. As pointed out by Arvind and Iannucci [9] the latency and synchronization are two fundamental issues which must be addressed by any parallel machine. The MIMD machines try to reduce the latency by providing caches. Due to their von-Neumann style of instruction
execution, there is a problem of synchronization (cache coherence). When the problem of synchronization is tackled, the latency get increased. This is discussed below.

**Cache Coherence Problem**

For small bus based MIMD machine, the addition of caches at each processor has improved the average memory latency and reduced the bus bandwidth consumed by an individual processor [5]. To build high performance scalable shared memory computers by using multi-stage networks, the problem of cache coherence become more difficult to solve with a reasonable cost especially, for the applications which do not have locality. A distributed directory based scheme is used to keep all the caches coherent in large scale shared memory computers. Censier and Feautrier, who proposed a distributed directory scheme for cache coherence define the problem [29] as follows:

*A memory scheme is coherent if the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address.*

---

5 By reasonable cost means that the cache coherence system will not increase the overall latency to access data significantly, i.e., communication traffic to keep caches coherent limit the grain size of the machine for a given application.
Consider a multiprocessor system with distributed caches in which each processor has its own cache to which it has exclusive access and two tasks are running on two processors, one on each processor which communicate through one or more shared memory cells. The system is shown in Figure 1.4. The task at processor $P_j$ needs data and sends a request for data to cache $C_j$. Assume, the data is not available in $C_j$. In this case a read-miss (as shown by broken line in the figure) in $C_j$ results in a request sent to the memory module $M_i$. The memory controller re-transmits the request to the copy of the data in cache $C_i$. This cache writes back its copy to the memory module $M_i$ and the copy of the data can be supplied to the requesting cache at $P_j$.

The task at processor $P_i$ writes a data at cache $C_i$ while executing some store instruction. On the write-hit a command is sent to the corresponding memory controller (as shown by solid line in the figure) which sends invalidations to all caches (cache $C_j$) marked in the presence vector residing in the directory $D_i$. Each memory module maintains a separate directory which records the state and presence information for each memory block. The state information is local, but the presence information indicates which caches have a copy of the block.

Thus, a read-miss in a distributed-directory distributed-cache shared memory computer may cause a long delay. Write updates to all the copies of a single block of data may create $O(N)$ traffic in the communication network with $N$ distributed caches. Therefore, logic to keep distributed directory coherent results in an increased traffic in the communication network and hence will increase the average latency. The similar conclusion is made by Robert Iannucci [58].

1.5 Data-Flow Approach to Build Computers

Data-flow computers are based upon the principle that the execution of an instruction is triggered by the availability of data rather than by a program counter. An abstract data flow machine can be considered as a directed graph of named nodes connected with arcs. The nodes correspond the computation or instructions and arcs represent the data dependencies. The data in the form of a token travel along the arcs. When all the input arcs to a node have a token, the node will fire and generate one or more new tokens. The arcs are assumed to be FIFO queues of unbounded capacity [69].

1.5.1 First Generation Data Flow Machine

In a static dataflow machine, the arcs are kept FIFO queues with the restriction that no arc can have more than one token at a time. This allows at most one instance of a node
(instruction) to be enabled for firing. In other words, to overcome the non-determinism for a reentrant graph (in the case of a loop or recursion) and a possible deadlock which could happen by presenting more than one token on a single arc, nodes in a static dataflow machine fire only when all of the tokens are available on its input arcs and the tokens on its output arcs have been consumed [35]. This means, each node needs extra feedback arcs from its output nodes.

A small static dataflow machine consisting of four micro-coded processors was built at MIT [36].

Reentrant graphs (loops) require strict enforcement of the static firing rule for each iteration of the loop to avoid non-determinate behavior, too. To achieve this the correct result tokens for each iteration of the loop have to be transmitted in proper sequence. Therefore, the static dataflow machine allows only a limited amount of parallelism to be exploited. This is because, the consecutive iterations of a loop can only partially overlap in time; they can never proceed concurrently, even if no loop-carried dependencies are present. Thus, only a pipelining effect may be achieved, where the maximum length of the pipeline is bounded by the critical path through the loop body. The same situation arises for recursion and subprogram invocations [7].

Another drawback is the feedback tokens, which in addition to making the machine more complex, increases the tokens traffic. Communication is an expensive process in any multiprocessor system, therefore, keeping the number of tokens to a minimum is an important requirement.

Static dataflow machines are not general enough to support procedure calls, non-strict op-
erators and general recursion [7], which are essential to any modern programming language. Also, the problem of handling arrays or lists has not been solved satisfactorily. Arrays are handled conceptually in the same way as scalars, i.e., a copy of entire array is always sent to all nodes that need to access it [7].

1.5.2 Second Generation Data-Flow Machines

To allow the execution of loop iterations and subprogram invocations in parallel, each instance of an instruction in a dataflow graph is tagged. With this approach, each arc can be viewed as a bag that may contain an arbitrary number of tokens with different tags. This principle is called tagged-token flow or dynamic dataflow model which was proposed [69] by Arvind at MIT [8] and by Gurd and Watson at the University of Manchester [48]. The enabling rule for tagged-token dataflow machine is as follows:

A node is enabled as soon as tokens with identical tags are present on each of its input arcs.

This eliminates the need for any feedback tokens, therefore, increases parallelism and decreases token traffic.

Based upon this principle, the Manchester Data-Flow Computer [50] was the first actual hardware implementation. The prototype became operational in October 1981 [7]. The overall structure of the Manchester Dataflow Computer is shown in figure 1.5. The waiting/matching unit matches tokens destined for the same instruction. Matched tokens are sent to the instruction fetch unit where they are combined with the appropriate opcode and destination addresses for result tokens. Each complete executable packet is passed on to the processing units, which comprise 20 identically microprogrammed ALUs. The produced result-tokens are passed onto the switch unit, which determines their destination PE and send them to the token queue. The token queue is a FIFO buffer that smooths out uneven rates of token generation and consumption.

The problems noted by Arvind, Bic and Ungerer in the Manchester dataflow computer are as follows[7]:

1. Inadequacy of handling complex structures.
2. Implementation of an efficient waiting/matching unit.
3. waiting/matching unit bottleneck for 20 concurrently working ALUs.

The proposed Manchester Multiring Data-Flow machine [23] can be viewed as an extension of the basic Manchester Data-Flow Computer, attempting to alleviate some of the problems experienced by the later.
Another machine based upon the dynamic dataflow model was the MIT tagged-token data-flow machine (TTDA) [10]. TTDA has been simulated on a multiprocessor emulation facility consisting of 32 TI Explorer Lisp machines interconnected by a high speed network [7]. The basic structure of the MIT TTDA was more or less the same as that for the Manchester dataflow machine, except a special storage, i.e., I-structure storage, provided to handle the problem of large data structures [11]. An I-structure can be considered as a repository for data values, which obey the single assignment principle, i.e., each element of the I-structure may be written into only once. After it has been filled, it may be read any number of times. Thus, the following three instructions are defined to operate on I-structure.

- **allocate**: This instruction reserves a specified number of elements for a new I-structure.
- **I-fetch**: This instruction retrieves the contents of a given I-structure element, if it has already been written.
- **I-store**: This instruction writes a value to a given I-structure element if it is empty.

In MIT TTDA, I-structure resides on a separate memory, called I-structure storage. Each memory location is tagged by two present-bits encoding three possible states, i.e., *present*, *absent* and *waiting*, to perform the operations (allocate, I-fetch, I-store) on each element. I-fetch instruction is implemented as split-phase, i.e., the requests are independent of time from the responses received.

Sigma-1 [101, 53] was a large scale tagged-token dataflow computer built in Japan with 128 processing elements. Figure 1.6 shows the structure of one processing element of Sigma-1.

Figure 1.6: Structure of a processing element of Sigma-1.
1. The input buffer receives data tokens generated locally or by other processing elements. The instruction fetch unit and the waiting/matching unit work simultaneously on the same token transmitted from the input buffer. If no match occurs, the fetched instruction is discarded. On a match, the fetched instruction is dispatched for execution. In parallel with the instruction execution, the destination unit produces the destination addresses for the result tokens which are routed to the local input buffer or to a remote PE.

Sigma-1 also contains I-structure storage for handling large data structures. 170 MFLOPS on a small integration program has been demonstrated on the machine with the ideal peak performance of 425 MFLOPS [53].

With the introduction of parallelism in instruction fetching and waiting/matching and also in the execution and forming destination addresses, it was a more sensible approach to build a parallel computer.

1.5.3 Problems with the 2nd Generation Dataflow Machines

The following major problems can be found in all the tagged-token dataflow machines as mentioned in the previous sub-section.

1. The implementation of a waiting/matching unit. One solution is an associative memory but it is very expensive. Therefore, all the implementations described have been using some form of hashing, sometimes supported by hardware hash tables. Hashing techniques are typically not fast enough to be used effectively as a single stage in the instruction processing pipeline. This often results in the long delay in the pipeline which limit the performance of the machine.

2. The second major problem was the size of the waiting matching store. When the store becomes filled with intermediate results, the machine deadlocks or some swapping or recycling techniques must be employed. In the present tagged-token dataflow machines the swapping or recycling techniques are quite complicated.

3. A third problem is the implementation of different types of stores, e.g., I-structure storage, makes the machines quite expensive.

4. Due to the overheads associated with token matching, communication, instruction scheduling and structure storage access, tagged token dataflow machines perform very poorly on sequential code.
1.5.4 Third Generation Data-Flow Machines

Some of the major problems of 2nd generation dataflow machines have been addressed in the third generation data flow computers. These include the implementation of waiting/matching store and to optimize the execution of sequential code (problem 1 and 4 as mentioned in the previous sub-section).

Explicit Token Store (ETS)

A significant development in the implementation of waiting/matching was the introduction of the explicit token store (ETS) in the Monsoon dataflow multiprocessor [82] at MIT. A similar principle is used in EM-4 [99, 100] as well. ETS has solved the problem of efficiently implementing the waiting/matching store. Papadopoulos has demonstrated the the waiting matching store can be implemented by using conventional storage. The principle is very simple. The tokens destined for the same instruction will explicitly point to a binary rendezvous point in the waiting/matching store. Each location in the waiting/matching store is augmented with an extra bit named as presence bit, to manipulate the state of the location. At the start, all the locations' presence bits, in general, will be in empty state. On the arrival of the first token, the presence bit will be set to full and the data will be stored. At the arrival of the second token, on detecting the presence bit full, a packet will be assembled and propagated to next stages for subsequent processing.

EM-4 and the Macroactor

To overcome the fourth problem of the 2nd generation dataflow computers, as mentioned in section 1.5.3, the concept of strongly connected graphs or macroactors was used in EM-4 [99, 100]. According to this scheme, a dataflow graph is transformed into a graph of macroactors by detecting strongly connected arcs or critical paths in the graph. The nodes within a macroactor are executed sequentially. A macroactor is enabled (fired) when all its input tokens are available. Once a macroactor fires, all the nodes in the block are executed as a block. Thus the execution of the nodes in a macroactor will not carry the overheads of token communication and waiting/matching.

1.6 How Should the Parallel Machine be Built?

To conclude the discussion about conventional von-Neumann style parallel machines, it can be seen that they are not capable of tolerating communication and synchronization cost for large scale parallelism [58]. When it is attempted to solve the problem of synchronization, the communication problem becomes worse. It follows, that due to a limit in the reduction
of the organizational overheads \((S_o)\) of a von-Neumann machine, a very fine grain optimum schedule can not be achieved.

To conclude with the discussion of dataflow parallel machines, it can be seen that it is possible to exploit fine grain parallelism, but it is not always desirable. High cost of exploiting fine grain parallelism makes the efficiency factor very small, in general. To improve the efficiency factor and reduce the cost for the exploitation of fine grain parallelism, problems 1 and 4 as mentioned in section 1.5.3 have been solved, to some extent, with the introduction of ETS and the macroactor. If problems 2 and 3 are solved, it may be possible to achieve a fine grain optimum schedule on such a machine at a reasonable cost. At the moment, these problems can only be solved by combining both approaches, i.e., conventional von-Neumann and dataflow in a balanced way in one machine. Even a great advocate of non-von-Neumann machine, Robert Iannucci writes [58]:

> For situations where instruction sequencing and data dependence constraints can be worked out at compile time, there is still reason to believe that a von-Neumann style sequential (deterministic time order) interpreter provides better control over the machine's behavior than does a dynamic scheduling mechanism and, arguably, better performance/cost.

By combining both approaches and solving the remaining problems in a machine, a program can be executed in such a way that for long sequential threads the conventional approach is adopted and for parallel threads the dataflow approach is adopted. To build such a machine, the dataflow bulk synchronous machine (DFBSM) model has been formed, which is defined as follows:

1. A number of components (nodes), each performing processing and/or memory access.
2. A communication network that can deliver messages from one node to other nodes.
3. Each node has the capability to generate synchronization signals, local synchronization, for scheduling a computational job for processing.
4. A facility to synchronize all or a subset of the processing components at a certain instant of times (global synchronization).

The first three items can be considered as a dataflow machine. The last item is added to eliminate problems 2 and 3 as mentioned in section 1.5.3. By generating a global synchronization, it is possible to recycle all the tags and then re-use the same waiting/matching store. With an efficient way of recycling all the tags, it is also possible to have a relatively smaller
waiting/matching store. Global synchronization can also be used to eliminate the need to implement synchronization requirement for each cell in the I-structure storage. Thus the machine build at top of the DFBSM model has a potential to address all the major problems of dataflow computers.

When DFBSM model was formulated, in its description it appears to be very similar to the Valiant's BSP (VBSP) model [95]. It is also claimed that VBSP is a bridging model, i.e., it can be considered as a reference for both software and hardware developers. Therefore, DFBSM is compared with VBSP in the next section.

1.7 VBSP model and DFBSM model

If the item 3 is eliminated from DFBSM description as given in the previous section, it describes the VBSP model. Although, the VBSP machine is very close to von-Neumann approach and therefore, the same reasoning can be applied for the validity of this approach for the parallel machine of specifications as mentioned in section 1.3, a slightly different approach is adopted to compare these two models.

To compare DFBSM with VBSP, Average Processor Utilization (APU) for a parallel machine is defined as follows:

Suppose the execution time for an application on a $P$ processors PRAM machine (with zero synchronization and communication time) is $T_p$. If the same application takes time $T_r$ on a real implementation of the PRAM machine with $P$ processors, then the relation between $T_p$ and $T_r$ gives the APU, i.e.,

$$APU = \frac{T_p}{T_r}$$
If a machine has a tendency to offer a higher APU, then that machine will be better. To compare the APU of machines based on VBSP and DFBSM, let us see what the factors are in a design which affect the APU. Suppose an application starts execution at time \( T_{r_{\text{start}}} \) and finishes execution at time \( T_{r_{\text{finish}}} \) on a parallel machine. During its execution, the application starts processing at time \( T_{r_{\text{pstart}}} \) and finishes processing at time \( T_{r_{\text{pfinish}}} \). Similarly, communication starts at time \( T_{r_{\text{cstart}}} \) and finishes at time \( T_{r_{\text{cfinish}}} \). The execution starts and finishes time can be calculated as follows:

\[
T_{r_{\text{finish}}} = \begin{cases} 
T_{r_{\text{pfinish}}} & \text{if } T_{r_{\text{pfinish}}} > T_{r_{\text{cfinish}}} \\
T_{r_{\text{cfinish}}} & \text{otherwise}
\end{cases}
\]

\[
T_{r_{\text{start}}} = \begin{cases} 
T_{r_{\text{pstart}}} & \text{if } T_{r_{\text{pstart}}} < T_{r_{\text{cstart}}} \\
T_{r_{\text{cstart}}} & \text{otherwise}
\end{cases}
\]

\[
T_r = T_{r_{\text{finish}}} - T_{r_{\text{start}}}
\]

To increase the APU, there is a need to decrease \( T_r \). From the above equations, to reduce \( T_r \), we need to achieve the following:

- Reduce processing and communication time.
- Overlap communication and processing, i.e., \( T_{r_{\text{cfinish}}} \) should be very close or ideally equal or less than \( T_{r_{\text{pfinish}}} \).

Thus the APU depends upon the capability of a design to offer overlapped computation and communication as well as faster communication and processing. The higher the overlapping, the better the APU. To illustrate the idea, consider an example. Suppose \( N \) instructions (or activities) on a \( P \) processor machine are being executed. For the execution of each instruction there will be some communication time (to bring data to the CPU) and computation time. An execution of \( N \) instructions application versus time is shown in Figure 1.7. The bolder lines show the communication time and the lighter lines show the computation time.

An operation cycle of a VBSP machine is as follows:

1. Each processor starts computation upon data available in its own memory (local memory).
2. It receives requests for data from other processors and generates requests for remote data.
3. When each processor finishes computation on local data, receives data for all remote requests and transmits data for all requests from other processors, all the processors will be synchronized.

4. The next cycle will start from step 1.

In a VBSP machine, all the processors will be synchronized when there is a need to access remote data. This implies that, in general, at the end of the execution of a set of instructions, there will be a communication. In this way one cycle will be completed and the execution of a set of instructions in the second cycle will start. The execution of \( N \) instructions versus time on a VBSP machine is shown in Figure 1.8. Due to the nature of VBSP model, the communication will increase by the end of the cycle and computation will decrease. In the first cycle, all the instructions involved in actual computation, not communication, are executed by the first dotted line. Then there is a time needed to make data local for the next cycle. Due to the reduction in computation work and increase in communication, the reduction in overlapping in computation and communication will result. Hence it will reduce the processor utilization (APU). To increase the \( APU \), Figure 1.8 suggests:

1. Reduce communication in the execution of an application.

2. Evenly distribute all the instructions to all the processors.

The second condition to achieve a higher \( APU \) is a well known problem of load balancing in the context of parallel machines. In general, load balancing demands more communication.

Figure 1.8: Execution of \( N \) instructions on a VBSP machine against time.
Figure 1.9: Average Processor Utilization of a BSP machine. T1 is the time at the start of each cycle in which the average processor utilization will be less due to loading operands from local memory\(^7\). T2 is the time when there will be maximum processor utilization on average, T3 is the time when the overlapping between computation and communication will be very small. T4 is the time for the barrier synchronization.

Therefore to achieve higher APU, VBSP machine needs bigger computation jobs (coarse-grain load balancing) with less communication in each cycle. The problem of communication in a VBSP machine is also addressed in [63] and a time stamp is introduced with remote read requests (based upon the discrete event simulation paradigm). If we plot the average processor utilization against time for a BSP machine, it is as shown in Figure 1.9.

In DFBSM, there is no such restriction that during each cycle the computation can be performed only on the local data. Also there is a local synchronization which can be used to support the execution of instructions needing remote data. Therefore, there will be no need to push all the communication at the end of each processing cycle (super step). Communication for the data needed for one instruction can be overlapped by the execution of another instruction which already has data. That is, most of the communication time can be hidden by the computation time due to overlapping in communication and computation and will not add up the total time needed for the execution of an application on a P processors DFBSM machine and hence will increase the APU.

1.8 Organization of the Thesis

The problems of implementing a parallel machine have been highlighted in this chapter and the approach selected to solve these problems in a parallel machine has been discussed briefly.

\(^7\)Local operands loading time is very small as compared to the remote access time.
In the next chapters the research carried out to implement the N-computer is presented.

The design of the N-computer at a system level only is discussed. To demonstrate the functioning of the system, a potential candidate for a PSP, F-code, is considered as the problem domain.

During this research, F-code being a problem domain for the N-computer was studied and compared with few other PSPs. This study is summarized in chapter 2.

The design of the N-computer is presented in chapters 4 and 5. In chapter 6, some of the possible ways to execute F-code functions on the N-computer are discussed.

One of the initial steps towards this research was to simulate F-code. The work carried out for the implementation of the simulator is discussed in chapter 3.

One aim of my research has been to develop the framework for the simulation of the whole N-computer system and if time allows, to implement the simulator. The work done for this purpose is described in chapter 7.

The N-computer is not a simple system, neither is the F-code, the problem domain. Also, a very new approach is adopted to design the N-computer, therefore, it may be difficult for a reader to grasp every thing in the first reading. Although, every effort has been taken to describe the things in a simple and easy to understand way. To understand the detailed functioning of the system, the reader should understand F-code. Therefore chapters about the F-code and its comparison with other PSPs and about the simulation of F-code are added after the first chapter.

For readers interested in studying the lower level details of the system, some of them are described in appendices at the end of the thesis. The definition of F-code is also attached in an appendix.
Chapter 2

Portable Software Platforms: Critique, Comparison and Suggestions

2.1 Introduction

In this chapter, the major features of known PSPs will be highlighted. Each PSP will be investigated for the attributes which a PSP, within the implied restriction of architecture independence, should have. Architecture independence means that the features like addressing mechanisms, number of registers and the ways to handle them (features normally available in an assembly language) must not be supported in a PSP. Where necessary, a PSP will be compared with others. F-code, being a potential PSP candidate, will be examined in more detail. Its ability to become a target language for high level data parallel programs, will be explored. Important points for which a PSP will be investigated, are:

1. Essential Features Preservation (EFP).
   (a) Data Concurrency (DC).
   (b) Process Concurrency (PC).
   (c) Primitive Type Sizes (PTS).

2. Simplicity

   EFP means to preserve all the features available in a high level program which may improve the efficiency of a particular implementation and, for the PSP compiler, either it is very difficult or it is impossible to infer those attributes again. This means that the time for the execution of a program in a high level language via PSP must not be more than the time for its direct execution (without PSP), in principle. In EFP, Data and Process Concurrency

---

1It is hardly possible to achieve this condition in all the cases, since a compiler may use some optimization in one situation than the other which are difficult to imagine now, and also depends upon the skills of the compiler writer. Therefore, in some cases one implementation may be better than the other. To qualify the statement, it should not be possible to prove that due to some features available in high level languages, like FORTRAN 90, which are not available in a PSP, the direct implementation will always be efficient.
and *Primitive Type Sizes* have been included.

Data concurrency means applying operations on a collection of data at one time. Data concurrency is an important source of parallelism which a parallel implementation exploits to minimize the time for execution of an application. In solutions of most of the scientific and engineering problems, data concurrency is inherited. If a PSP does not preserve the data concurrency either it may be very difficult to extract all the data concurrency or it will be an unnecessarily large overhead for a PSP compiler. Therefore, data concurrency available in an application is considered as an initial feature to be preserved in a PSP.

Process concurrency which includes functional and process parallelism is an important feature which a PSP implementation may utilize to reduce the execution time of an application. Functional parallelism means to evaluate the operands to an operation in parallel, as is available in functional programming languages. Process parallelism is the evaluation of more than one operations in parallel, as is found in process concurrent languages like Occam. Although, process concurrency in an application, written in data parallel high level languages like Fortran 90, will not be as advantageous as the data concurrency for a parallel implementation, it will be an important source to provide excess parallelism to hide communication and synchronization latency. Therefore, a PSP as opposed to a typical assembly language, must represent the process concurrency (PC). Hence, PC is included in the essential features to be preserved in a PSP.

Primitive type sizes (PTS) is the size of different types which can be used in a high level program, for example, 16 bits integer or 32 bits integer. If a PSP has only one type for both integers, it is named *int*. Then the compilation of such a language into PSP will convert both integers into *int*. If a programmer knows that he/she just needs a 16 bits integer and he/she writes codes accordingly, then the execution of such a program may take more time compared with the situation when the language is implemented directly and the program is executed by using the direct path. Especially, for a dataflow execution on a parallel machine, which is inherently more communication demanding, the difference could be much more significant. If a PSP loses information about the primitive type sizes, PTS, then a compiler will never be able to recover it. Therefore, PTS has been included in the essential features.

The other important feature for which a PSP should be examined is the simplicity of the way in which it preserves the essential features (EFP). In the process of preserving the EFP, a PSP code should be as simple as possible. Minimizing the semantics of a PSP may often result in a more complex representation of a top-level program into a PSP. On the other hand trying to preserve every thing in a top-level program as it is, may make the PSP itself much more complicated. Thus simplicity means a balanced PSP which neither make it very

---

2Unless one is willing to interpret a program before the compilation, I am sure no one would like this.
complex nor create a complex representation of a high level program.

2.2 F-code: A PSP

F-code is an intermediate level, an architecture independent language for data parallel high level languages like EVAL [78], Fortran 90 [74, 28] etc. It expresses the data concurrency available in a high level program, by applying functions (instructions) on multi-dimensional data objects as an atomic operation. Basically, a high level program is decoded into F-code functions arranged in a tree like fashion. All the arguments of an F-code function, which do not specify any order for their evaluations, can be evaluated in any order or in parallel.

Functional parallelism available in a high level program can be detected from the structure of the F-code tree. The process parallelism can be preserved by using PAR instruction. Thus, F-code preserves the process concurrency, PC. However, the F-code typing system does not preserve the size of primitive types (PTS).

Every F-code primitive object has three attributes associated with it: type, sort and shape. An object may be of any type, - logical, character, integer, real, complex - sort, - value, name, target - and a shape which is a one dimensional object whose length gives the rank of the object and contents give the size or extent along each dimension. To support computation on multi-dimensional arrays of different shape, F-code has a well defined rank and length coercion mechanism. Length coercion is implicit and by applying functions to objects of different size along their corresponding dimensions automatically creates a result according to the minimum length along each dimension. In general, the rank is coerced to the maximum when objects of two different ranks are computed. For example, the addition of a vector to a matrix will result in a matrix by adding the vector along rows or columns of the matrix. The rank of the objects being computed can be coerced upto the sum of the ranks of all the objects. For example, the addition of two vectors one is along the rows and the other is along the columns, results in a matrix. Rank coercion is guided by binary masks.

F-code has a full set of arithmetic and logical operations on arrays. There are functions which can be used to define constant one dimensional arrays. To reshape an array, to cut a regular piece from an array or to compose arrays to form one array, it has a rich set of geometric operations. The geometric operations can be used to select a part of a multi-dimensional object by using indexing on dimensions (sect and slice). To select sparse elements of an array it has gather and transform functions. The gather uses a logical array and transform uses a multi-index to select the elements of an array. It is also possible to create and reference structured objects. It has quite powerful functions to support memory

\(^3\)An identifier updated in more then one arguments of such functions will be undefined.
management. Primitive or structured objects can be created in the heap. It is also possible to create channels which are FIFO queues of primitive objects. Objects can be pushed on a channel and popped from the channel. F-code has limited support to more than one procedures in a program and it can only support internal procedures (this will be discussed in a separate section).

In short, F-code has a very rich set of semantics, which make it a very complicated virtual machine. The semantics of F-code can support nearly all the features available in a high level language and also some which are found only in some lower level languages, e.g., pointers\(^4\) and channels\(^5\). This gives much more flexibility to the top-level compiler to compile a program in F-code. By preserving nearly all the features in a program, it offers a wide range of optimizations specific to certain implementations. The corollary to this is that it really makes the job very difficult for a hardware designer to come up with a machine which can execute an F-code program in an efficient way.

### 2.3 VCODE: A PSP

VCODE\(^{[22, 21]}\) is an intermediate level language which is claimed to be a candidate for PSP. In \(^{[22]}\), authors write:

"VCODE is a small, simple language that could ultimately serve as a portable intermediate representation for high level data-parallel languages."

VCODE is a single stack, single assignment intermediate level language with relatively small number of instructions (functions) and simpler semantics. Functions operate on arbitrarily long vectors in the stack. There are instructions to copy vectors from deep inside the stack to the top of the stack and also to remove vectors from the stack. The same instructions are used for dynamic memory management (allocation and free). Each entity in the stack is a homogeneous vector of atomic types (boolean, integer, floating and segment descriptor\(^6\)).

The PSP VCODE is based upon a V-RAM machine earlier introduced by G Blelloch \(^{[19]}\). V-RAM is a standard serial RAM with the addition of a vector memory and a vector processor. In V-RAM, each memory location in the vector memory can contain an arbitrarily long vector of atomic value; the vector length is associated with the vector not the memory location. Each instruction of the vector processor operates on a fixed number of vectors from the vector memory.

---

\(^4\)These may be used to access addresses to physical locations where data is stored.  
\(^5\)The F-code channels can be used like Occam channels.  
\(^6\)Segment Descriptor is a vector that specifies a partitioning of one or more data vectors into segments.
Restrictions imposed upon the VCODE due to the sequential nature of V-RAM machine and further restrictions imposed due to its single stack nature are probably the most important point which can limit its scope as a PSP. Compiling a program in a data parallel high level language (high level program) to VCODE will destroy an important source of parallelism, namely process and functional parallelism (process concurrency). Its single stack nature may add extra complexity to the code due to moving the operands at the top of the stack. The process concurrency could be an important source in the excess parallelism, especially when we are evaluating fat trees. The excess parallelism is an important source needed to hide communication and synchronization latency of a parallel implementation. After destroying process concurrency and adding an extra complexity in the intermediate representation of a program, it may be impossible for the down-level compiler (the compiler which compiles VCODE to the actual hardware) to recover that parallelism. In contrast, F-code is a multistack machine where the number of stacks are limited by the number of arguments of a function. Therefore it does not destroy any process concurrency available in a high level program.

VCODE does not support operations on arrays or multi-dimensional data objects. The top-level compiler has to map all those operations onto long vectors, with the addition of the segment descriptor. This was adopted in order to make VCODE simpler and smaller [22]. This is debatable. However having a smaller instruction set does not necessarily mean that the code will be simpler. A PSP instruction set must be carefully chosen to keep the code simple. A reduced instruction set may create code which are rather difficult for the compiler to make sense of. To accept the importance of array vectors, Blelloch et al. say [22]:

"It is possible that we will lose some performance by not carrying the array information into VCODE, since array operations on parallel machines often take advantage of locality. Part of our analysis of VCODE will be to determine how well a compiler can derive this information. It may be necessary to add directives to VCODE to gain full performance."

To conclude, VCODE is much smaller with simpler semantics than F-code. But the extra smallness of VCODE really limits the scope of its being adopted as a PSP. It may act as an intermediate level representation for programs targeted for SIMD style machines. Certainly, it does not offer a suitable intermediate representation of programs for MIMD, dataflow or hybrid parallel architectures. One important point from the VCODE, which should be considered in F-code is the elimination of heterogeneous data objects. VCODE does not support primitive type sizes (PTS).
2.4 Other PSPs

Another intermediate level representation, the Virtual System Architecture (VSA) [61, 62] can be considered as a PSP. It is very similar to the V-code except its functions (methods) update (evaluate) multi-dimensional data objects. It is based upon the principle of *active data* and the application of *methods*. *Active data* consists of the data elements which are going to be updated in a data object. If creation of an active data is named as data activation, then a VSA program first activates data and then the *methods* will be executed upon those activations, synchronously, i.e., there is a need of synchronization before and after the execution of each *method*. In other words, it is a stack machine where each entity of a stack is a multi-dimensional data object. This limits the process concurrency when compared with the F-code.

There has been a similar approach for sequential languages to achieve portability between different architectures. A most prominent example could be *TDF* [1, 2]. It is an intermediate format designed by the United Kingdom's Defense Research Agency for the languages like ANSI C, C++, FORTRAN 77, COBOL and Pascal etc. Since the TDF is intended for sequential languages, therefore it does not preserve the data concurrency. It is a sort of assembly language written in a tree like fashion.

There are some other intermediate level languages mainly designed for functional and dataflow programming languages like DACTL [49] and TAM [40]. TAM is a lower level intermediate level language with explicit synchronization and scheduling of basic code blocks which a compiler will generate. The dataflow language Id [79, 80, 81] has been compiled into TAM. Anyhow, TAM has never been claimed as a PSP. Therefore, the discussion of this is not within the scope of this chapter.

2.5 F-code; A Critique

In F-code, it is possible to create pointers to objects, perform arithmetic on pointers and access the objects by using pointers. A pointer is a scalar integer value and does not have any information about the type and shape of the object. In the current definition of F-code, the functions *create, global* and *local* may create objects of any type (primitive or structured) and return a pointer to the object. The functions *displace* and *distance* are used to manipulate pointers. The function *select* can be used to access structured objects associated with pointers. However, the definition of the select function is needed to be modified in a way that it can be used to access primitive objects associated with pointers. To comply with this condition, *select* can be redefined as follows:
SELECT :::= (select MODE <number> EXPR.p EXPR.e .....)

The MODE will provide the type information for a primitive or structured object. In the case of template, the number will point to the leaf of the template.

2.5.1 Procedural Support in F-code

A program in contemporary high level languages may have more than one independent modules or procedures. It is also common practice to develop these modules independently from one another. These modules may be compiled and tested separately, too. In a main program, these modules can be linked together to produce a single executable object. Parameters can be passed from the caller to the called procedures and a called procedures can returned objects, too. This feature is named as the procedural support of a language. F-code has a limited support for the procedures which is discussed in the following paragraph.

An EVAL [78] program is a set of modules (procedures and functions). Textually, modules can not be written in a nested form. This feature allows the separate compilation of modules as is possible for C functions. Each module is labeled with a tag (name). The execution of a module can be invoked by using the tag of the module. Consider the following program in EVAL:

**Example 1**

```
program LIFE
  MAIN
  NEXT (CURRENT)
  OUTPUT (CURRENT, TIME)
end program
```

The program in Example 1, LIFE, contains the three modules, MAIN, NEXT and OUTPUT. The execution of these modules in the program is invoked by just putting their names along with the arguments if there are any. These modules are not necessarily stored in one file and could be compiled separately.

Let us try to compile this program in F-code. In F-code, the top-level compiler may generate call function to invoke a subroutine by using an integer scalar value being a pointer to that subroutine. The equivalent code for Example 1 in F-code may appear as follows:

**Example 2**

```
......
(seq

call
```

7A single large program is extremely difficult to debug and maintain. Breaking down a large program into module, make it easier to understand the program. Sometimes, even relatively shorter programs are greatly improved when their component parts are declared as independent modules or procedures.
The arguments for modules are considered as global variables which will be updated by the modules (subroutines). How does the compiler know about the pointers to MAIN, NEXT and OUTPUT? There is no way for the compiler to know, if the modules are not in the same file. One possibility might be, if the compiler generates dummy linkage identifiers and some conditions are placed on the linkage editor. Then the compiler may generate meaningful code as shown below:

Example 3

```
.........
(seq
  (call
    (var value LNK_ID_MAIN)
  )
  (call
    (var value LNK_ID_NEXT)
  )
  (call
    (var value LNK_ID_OUTPUT)
  )
)
.........
```

The linkage identifiers (LNK_ID...) are defined as having global scope. While compiling individual modules (subroutines), the compiler may generate the following code:

Example 4

```
.........
(assign
  (var name LNK_ID_MAIN)
  (const
    (main expression
      ...........
    )
  )
)
.........
```
The condition upon the linkage editor will be that it has to interpret all the assignments which carry linkage identifiers. In the current definition of F-code, there is no such condition defined. The parameters can not be passed by name to the called function, too. To pass parameters the objects have to be associated with identifiers. This makes type control between procedures difficult, since the linker would not have information about parameter types.

In Fortran 90, there are two types of procedures named subroutines and functions. Both could be internal or external. Internal procedures are textually nested within a program or procedure with a special keyword CONTAIN. If a program or a procedure contains a procedure then it will be written at the end of the procedure or the program, just before the last statement (which is an end statement). External procedures could be written in some other file or in the same file but outside an existing procedure or program. HPF, being an extension of Fortran 90, keeps the same notion of procedures as in Fortran 90.

In the light of the discussion about EVAL procedures and F-code, F-code supports the concept of internal procedures of Fortran 90. It is not possible to compile a program in F-code which invokes the execution of an external procedure. Nearly all languages support the procedures whether those are internal or external. Therefore, to be able to compile programs in F-code, we need procedural support. This could be done by labeling a subroutine with an identifier.

### 2.5.2 Array Operations and F-code

F-code has a rich set of geometric operations to support the array features of a data parallel high level language. To demonstrate this, let us consider a few examples in Fortran 90 and their possible translation in F-code.

**Example 5: Constant array construction**

\[
\begin{align*}
/(I, I = 2, 16)/ & \implies /2, 3, 4, 5, \ldots, 16/ 
\end{align*}
\]

**Example 6: Possible Compilation in F-code**

\[
\begin{align*}
\text{ramp} \\
\quad (\text{const } 2) \\
\quad (\text{const } 16) \\
\quad (\text{const } 1) \\
\end{align*}
\]

Consider another array constant of Fortran 90:
Example 7

(/
/3,5,7,25,8/
/
)

This array can be constructed in F-code by using comp\textsuperscript{8} function as follows:

Example 8

\begin{verbatim}
(comp 0
 (const 3)
 (comp 0
 (const 5)
 (comp 0
 (const 7)
 (comp 0
 (const 25)
 (const 8)
 )
 )
 )

\end{verbatim}

Array element indexing in a Fortran 90 program does not necessarily start from zero. A programmer can specify a lower bound on each dimension on array. But in F-code, the lower bound is always zero. This should not create a problem while translating indexing from a P lower bound to zero lower bound. The compiler simply needs to find the displacement 0-P, then by adding the displacement to an index in Fortran 90 program the F-code index will be obtained.

Array sections of Fortran 90 which are used to access a regular sub-array of an array can be implemented into F-code by using sect and slice functions. The \texttt{WHERE} construct of Fortran 90, which uses a logical array to update certain elements of an array can be implemented by using \texttt{choice}. Consider the following example of a \texttt{WHERE} construct:

Example 9

\begin{verbatim}
WHERE (ARRAY > 0.0) A = 1.0/ARRAY ! ARRAY is a real array
\end{verbatim}

For a possible translation into F-code, it is supposed that there is a logical array \texttt{LOGICAL\_ARRAY} of the same shape as array but with true elements where the condition

\textsuperscript{8}To compile this by using \texttt{comp} function, the scalars are treated as one dimensional objects of extent 1. F-code does not define many functions, e.g., \texttt{diadic, choice} and \texttt{comp}, on scalar arguments, explicitly. To make the definition clearer, these functions should be defined properly for scalar arguments. To eliminate ambiguity, there is also a need to define the optional constructs in the definition of each function, e.g., masks.
ARRAY > 0.0 is true. Also a unity array of the same shape contains 1.0 elements named as UNIT-ARRAY. The translation will be as follows:

**Example 10**

```
(assign
  (var name A)
  (choice
    (var value LOGICAL-ARRAY)
    (monadic inv
      (choice
        (var value LOGICAL-ARRAY)
        (var value A)
        (var value UNIT-ARRAY)
      )
    )
    (var value A)
  )
)
```

2.6 Conclusion and Suggestions

In the previous sections all known PSPs are examined for Essential Features Preservations (EFP) and for simplicity. F-code preserves two important features of EFP, Data Concurrency (DC) and Process Concurrency (PC). It does not preserve Primitive Type Sizes (PTS). VCODE does not preserve PC and PTS of EFP. It tries to preserve DC, but due to its single stack nature, the intermediate representation of a program becomes in general more complex.

F-code, as a target language for data parallel high level languages is examined. Explicitly stating a distribution strategy for the elements of an array will improve the clarity of the definition. Operations on multi-dimensional data objects in F-code are strong enough to support array operations available in data parallel high level languages like Fortran 90.

In the light of these conclusions, the new definition of F-code is suggested as follows:

**Const:** To support PTS and call for external procedures, the new definition of const function is:

```
CONST ::= (const [<identifier>] | [<type_size>] <constant>)
```

- **identifier ::=** As defined in F-code definition
- **type_size ::=** 1|2|3|4|5|......

39
Anything in brackets [ ] is optional. If there is any identifier given (identifiers always start with letters, not numbers), then it will contain an expression to be declared as a procedure. All the other features of this function are the same as in the standard definition of F-code. If const is used to declare procedure, then the identifier will be associated with the procedure. And the const will return an integer scalar being a pointer to the procedure.

The suggested definition contains the following functions in addition to those previously defined in this section:

\[
\begin{align*}
\text{RAMP} & := (\text{ramp } \text{EXPR}.b \text{EXPR}.e \text{EXPR}.s) \\
\text{MONADIC} & := (\text{monadic } \text{<unary> } \text{EXPR}.a) \\
\text{DIADIC} & := (\text{diadic } \text{<binary> } [\text{<mask>}] \text{EXPR}.l [\text{<mask>}] \text{EXPR}.r) \\
\text{CHOICE} & := (\text{choice } [\text{<mask>}] \text{EXPR}.s [\text{<mask>}] \text{EXPR}.t [\text{<mask>}] \text{EXPR}.f) \\
\text{REDUCE} & := (\text{reduce } [\text{<mask>}] \text{EXPR}.a) \\
\text{TRANSP} & := (\text{transp } \text{<number> } \text{EXPR}.a) \\
\text{SECT} & := (\text{sect } \text{<number> } \text{EXPR}.s \text{EXPR}.i) \\
\text{SLICE} & := (\text{slice } \text{<number> } \text{EXPR}.s \text{EXPR}.i) \\
\text{REPL} & := (\text{repl } [\text{<mask>}] \text{EXPR}.s \text{EXPR}.r.....) \\
\text{PACK} & := (\text{pack } \text{EXPR}.s \text{EXPR}.e..... ) \\
\text{GATHER} & := (\text{gather } [\text{<mask>}] \text{EXPR}.s \text{EXPR}.m) \\
\text{DIAG} & := (\text{diag } [\text{<mask>}] \text{EXPR}.a) \\
\text{TRANSFORM} & := (\text{transform } \text{EXPR}.s \{ [\text{<mask> } \text{EXPR}.t} ..... ) \\
\text{COMP} & := (\text{comp } [\text{<number>}] [\text{<mask>}] \text{EXPR}.l [\text{<mask>}] \text{EXPR}.r)
\end{align*}
\]
TYPE ::= (type EXPR)

SHAPE ::= (shape EXPR)

ASSIGN ::= (assign EXPR.l [<mask>] EXPR.r)

SEQ ::= (seq EXPR ...... )

PAR ::= (par EXPR ...... )

LOOP ::= (loop EXPR)

SPAWN ::= (spawn <identifier> EXPR.a EXPR.n)

IF ::= (if EXPR.c EXPR.t EXPR.f )

COMMA ::= (comma^{14} <direction> EXPR.d EXPR.r)

CALL ::= (call <identifier^{15} >)

COERCED ::= (coerce <identifier> EXPR.a <property>
    <type> <type_size> <number> )

^{14} As proposed by Carl Sutton.
^{15} Must be associated with a subroutine/procedure.
Chapter 3

The F-code Simulator; F-SIM

3.1 Introduction

The first step towards the design of the N-computer was to simulate F-code programs as this is the problem domain for the N-computer. To implement an F-code engine this was the only way to fully understand the problem domain. In this chapter, the structure of the simulator (F-SIM) is described. It is implemented in C [66, 16, 6, 68]. The purpose of implementing F-SIM was to reflect the semantics of F-code in the implementation as closely as possible. Therefore, the implementation is not optimized to provide an efficient interpretation of F-code on a sequential machine. F-SIM simulates an earlier version of F-code [25].

Most of the F-code's functions are strict, i.e., the arguments are evaluated before the evaluation of the body of the function. An F-code program is expressed in a Lisp [15, 98] like fashion. The control thread of the simulator is a depth first left to right tree walk.

The structure of the F-SIM is shown in Figure 3.1. The first two steps can be grouped in the analysis phase of the F-SIM and the third phase is the actual synthesis of the functions. The analysis phase is the same as the front end of compilers and interpreter as given in [4, 26].

3.2 Analysis Phase

First step is the lexical analyzer which partitions the characters of the source program (F-code program) into tokens and deliver them to the syntax analyzer which generates the syntax tree.

Next the syntax analyzer examines the tokens produced by the lexical analyzer and builds the syntax tree. The syntax analyzer also reports the syntax errors to the standard error (stderr), which is useful in the development of F-SIM itself. In actual there is no need for the error reporter because F-code is not intended for manual programming instead F-code programs will be generated by compilers automatically. Therefore, it can be assumed that
the F-code program will be free of syntax errors.

In F-SIM, the lexical analyzer is not independent, i.e., it does not collect all the items of an F-code program and then deliver the tokens to the syntax analyzer. It is not possible to built the lexical analyzer separate from the syntax analyzer because the F-code program is not completely context free. The lexical analyzer is called by the syntax analyzer each time it needs a token. In the F-code program, after the start (left parenthesis), the next item will be a function (expr) from which the syntax analyzer will know what the next token should be. It is illustrated in Figure 3.2.

Each node in the syntax tree is a structure of function name, type (type or class of the function is encoded within the function name), line number (for error report) and the function arguments. If the leaves of a function are also functions then the function arguments will point to these functions.

The analysis phase of the F-SIM is simple and straightforward because the grammar is very simple. Therefore, there is no need to discuss this in detail. The state transition diagrams in Appendix F dictate the function of the syntax analyzer. Also the structure of each node is clear from the state transition of each function.

The syntax analyzer also manages the symbol table which provides the mapping from source program identifiers (names) to the interpret time (or run time) objects or expressions which they denote. The syntax analyzer gets associations for each identifier and keeps the association in the node of the tree for the functions which manipulate the identifier.
3.3 Synthesis Phase

The synthesis phase of the F-SIM is the interpreter which obtains input from the syntax tree and the symbol table. The interpreter is simply a device which walks through the syntax tree and carries out the operations defined by the nodes.

Symbol table is an array of records in which each record contains the pointer to the identifier and the pointer to its association. During execution the interpreter forms and destroys the associations in this table. In F-SIM, the storage space for the name is not fixed but is allocated dynamically.

Each association is also a record which contains all the other attributes of an identifier. The interpreter adds attributes (like write an object of any sort from the stack, or evaluate the 'expr' associated with the identifier if the 'class' is 'expression' when it is needed, etc) to the association and destroys the association.

The interpreter also reports run-time errors to the standard error (stderr). If the interpretation of a function does not terminate properly, then the error will be reported. The interpreter also prints the objects stack to the output file to verify the correctness of the interpreter.

The target for the interpreter is a sequential machine and the interpreter itself is a stack machine. For returning results from a point deep inside of some expression there is another stack which is called 'result-proc' stack. This will be explained later in the description of the interpretation of the respective functions. Each entry of the stack has all the attributes which an F-code data object has; like type, sort, rank, shape and the contents. The interpreter takes objects from one of these two stacks, evaluates the functions and pushes the resulting objects onto the object stack. The interpretation of each function signals to the environment whether it has terminated properly or not. At improper termination the interpreter will exit from the execution state and signal the run time error to the environment. The interpretation of each function is shown in Appendix G.
3.3.1 Evaluation of Non-Geometric Functions

Intp-CONST

Interpretation of \texttt{const} is quite simple. The node contains all information about the attributes of an F-code constant. The interpreter creates an object with these attributes and the value of the constant and pushes that object on to the stack.

Intp-SCOPE

Interpretation of \texttt{scope} associates an identifier according to the class, to the first expression (\texttt{scope} contains two expressions) or its result and then evaluates the second expression using the current association of the identifier. After evaluating the second expression the interpreter destroys the current association of the identifier and signals the proper termination of \texttt{scope}.

Intp-BIND

The function \texttt{bind} is also similar to the \texttt{scope} but instead of getting the association from the first expression, it associates the identifier with the association of the second identifiers (\texttt{bind node contains two identifier and an expression}) and evaluates the expression (\texttt{expr}). After proper termination, the copied association of the identifier is destroyed.

Intp-PROC & RESULT

The functions \texttt{proc} and \texttt{result} help in returning results from inside of an expression. Using a single object stack for the implementation of this was not possible and so two stacks are used. One is main object stack and the other is \texttt{res-proc} stack. The function \texttt{result} pops the object stack after the evaluation of the \texttt{expr} and pushes it on the \texttt{res-proc} stack. The function \texttt{proc} interprets the expression, pops the \texttt{res-proc} stack and, after checking the equality between the descriptor of the popped object and the descriptor stored at the node, it pushes the object onto the object stack as shown in Figure 3.3. In this figure, first the argument of \texttt{result} is interpreted and the result is kept on the object stack. The interpretation of \texttt{result} pops the object and push it onto the res-proc stack. When there will be the function \texttt{proc}, the simulator pops the object from the \texttt{res-proc} stack and push that object to the object stack. The proper termination of \texttt{result} also returns an integer scalar value ‘0’ on the stack. In the implementation the stacks are basically stacks of pointers to the objects. The movement of objects between these two stacks is therefore simply the movement of pointers.
Figure 3.3: Action on both stacks for result and proc functions. The sequence of actions is tagged with numbers. First the function result is interpreted and then the function proc is evaluated.

**Intp-VAR**

The function `var` checks the association types of the identifier and compares the sort in the association with the sort at the var node in the tree. Depending upon the sort of the node and the association type of the identifier the object is pushed onto the object stack. Again this will be a pointer to the object in the heap which was associated with the identifier.

**Intp-CALL**

The function `call` simply finds the current association of an identifier which should be of `class` expression and evaluates the expression associated with that identifier.

**Intp-RAMP**

`Ramp` returns a one dimensional integer value by evaluating three expressions, popping the resulting objects up and making the required object from these three objects.

**Intp-MONADIC**

The 'monadic' node contains a unary operator and an argument expression. The interpreter evaluates the expression. If the evaluation is terminated properly then it pops the object, performs the operation specified by the unary operator and pushes the result onto the object stack.
Figure 3.4: Dyadic function with operator addition. One argument is a vector and the other argument is a matrix. The addition will be performed after rank and length coercion.

**Intp-PART**

The interpretation of 'part' selects the real or imaginary part of the complex object resulting from the interpretation of the expression which the node contains. The sort of the resulting object will be according to the sort of the source object. The type of the resulting object will be real which contains the real or imaginary part of the object according to the 'selector'.

**Simulation of the remaining functions**

The simulation of the functions like \( p, v, \text{seq}, \text{comma}, \text{par}, \text{loop}, \text{if}, \text{create}, \text{static}, \text{type} \) and \( \text{shape} \) are simple and clear from their definitions [25].

### 3.3.2 Evaluation of Semi-Geometric Functions

All the previous functions evaluate whole data objects, therefore this interpretation is simple. In this section, the functions will not necessarily evaluate a whole data object but instead will select and orient the objects which are pushed on the object stack by their arguments. According to the description of masks e.g., a vector can be oriented along the rows or columns of a matrix and then can be evaluated by a particular function. The masks are used to expand the object during rank coercion, e.g., in *dyadic* or in *choice*. In this section the interpretation of functions which evaluate data objects with rank and length coercion is discussed.

**Intp-DYADIC**

The *dyadic* node contains two expressions with masks and a binary operator. For a binary operation, in general, the rank is coerced to the maximum when the rank of the two arguments is not same. Also, the rank of the result can be up to the sum of the rank of
the two arguments. This is determined by the mask with each argument and the shape of the arguments. The length along each dimension is coerced to the minimum of the two arguments. For example, we can perform a dyadic add on a vector and a matrix. The vector can be oriented along the rows or columns of the matrix which will be determined by the mask for the vector. If the vector is oriented along rows then the number of columns in the resulting object will be the minimum of the size of the vector and the number of columns of the matrix. The interpreter evaluates both expressions and after coercing-up to their senior type, pops the objects from the stack. Then, by expanding the shape vectors of both using their respective masks and taking the minimum of both expanded shape vectors, the shape of the resulting object will be obtained. After this it arranges the both arguments' objects according to the resulting object's shape. Now it has two objects with equal type, rank and shape vectors. Then, the interpreter simply performs the operation on two data objects according to the operator as shown in Figure 3.4, and pushes the result onto the object stack.

**Intp-CHOICE**

The *choice* function is a data parallel choice between two objects depending upon the value of the logical object. The rank and length is coerced according to the mask of each expression as in the case of dyadic function. Then, by testing the logical object the contents of resulting object are selected from one of the two objects. Each element of the resulting object is either from *exprt* or from *exprf*, the two of the operands of *choice*.

**Intp-REDUCE**

The *reduce* function reduces the object resulting from interpreting the expression which is an argument of the *reduce*, by applying an associative commutative binary operation to certain elements of it. For example all rows or columns of a matrix can be added (which will result in a vector) by applying the *reduce* function to the matrix. The mask at the *reduce* node indicates along which dimensions the object will be reduced. If an object with shape vectors \( (5, 4) \) is reduced with mask \( (0, 1) \), the resulting object will be of shape \( (5) \), as shown in Figure 3.5. The interpreter evaluates the argument, *expr*, and pops the object. The object is stratified according to the mask and converted into layers. Then, the associative and commutative operation is carried out between all these layers and the resulting object is pushed onto the stack.
Intp-ASSIGN

The assign function is used to achieve side-effect in memory, i.e., changing the contents of values referred to by some name or target. The interpreter for assign, first interprets the arguments, EXPR.l and EXPR.r. Then, if the left argument, EXPR.l, is not a dummy reference, the projection of the right argument, EXPR.r, with the mask is selected and coerced (if the type is junior) to the type of the elements referred to by EXPR.l. Then, the values of EXPR.l are updated by the coerced projection of EXPR.r.

In the simulator, an object of sort name is a collection of pointers arranged with the given shape. Each pointer points to a memory location which holds the value of the type of the object. The result of EXPR.l is a name object, in general (if it is a couple then the name is selected) and the result of EXPR.r is a value object. The memory locations pointed by the result of EXPR.l are updated by the result of EXPR.r.

After the proper termination of assign an integer scalar value 0 is returned to the object stack and the environment is informed about the proper termination of the interpretation of the function.

3.3.3 Evaluation of Geometric Functions

The geometric functions of F-code perform functions such as re-arrange data objects, select a part of data objects or replicate a data object, etc.

Intp-TRANSP

This function transposes its operand according to a number provided with the definition of the function. The function transp simply takes an object of n-dimensions and generates
an object of n-dimensions with the dimension specified by the number moved at the end of the shape vector. For example the transposition of a 2-D matrix with number 0 simply interchanges the rows and columns of the matrix. The transposition of an object with shape vector $(3,4,5)$ and number ‘1’ result an object which has shape vector $(3,5,4)$ with accordingly arranged data elements.

**Intp-SECT**

To select a regular section of an object the function is *sect*. The regular section of an *n*-dimensional object will be of *n* – 1 dimensions with a dimension has been eliminated. The parameter *number* provided with the definition of *sect* function tells which dimension will be eliminated. The layer number along this dimension which will be selected is decoded by the result of another argument of the sect function, which is an integer scalar value. For example, if the *sect* node contains

- exprs, which results an object with shape $= (3,4,5)$,
- expri, which results an integer scalar value $= 2$,
- and the number $= 1$.

Then the resulting object will have shape $(3,5)$ and this will be the third (indices start from 0) layer along second dimension of the source object as shown in Figure 3.6.

The interpretation of *sect* follows the same logic and makes an object with the selected layer of the source object along the eliminated dimension and then pushes that object on the object stack.
Intp-SLICE

To select more than one layer along the eliminated dimension sect function cannot be used, e.g., to select an object \( (3,2,5) \) from the object \( (3,4,5) \), as mentioned in the previous section. The resulting object will have two layers along the eliminated dimension. For this purpose F-code has another function which is called slice. For selection of more than one layer there is a need for more than one integer value, therefore one of the argument of the slice node is a one dimensional integer value instead of scalar as in sect. The interpretation is similar as that of sect but the dimension specified by the number will have size according to the size of the one dimensional object. The interpreter pops the data object from the object stack and selects certain elements of it and pushes the result onto the object stack.

The slice of an n-dimensional data object yields an n-dimensional object with a different (possibly) size in the dimension along which it is sliced.

Intp-REPL

The function repl replicates source object. The definition of the function contains the mask which decodes the dimensions along which the source object will be replicated. The length of the mask will be the rank of the resulting object. For the number of replications along each dimension, the function has an expression as one of its argument which yields an integer scalar value.

The interpreter evaluates source expression and pushes the object onto the object stack. Then it evaluates the expressions which yield the replication factor along each dimension of the replication and pushes the value of each expression to the object stack. After evaluating the arguments, the interpreter pops the objects from the object stack according to the length of the mask at the node. Then, it pops the object which will be replicated and makes a new object and pushes on the object stack. After pushing the resulting object it informs the environment about proper termination of the evaluation of the function. By definition the function repl can replicate an m-dimensional object into an n-dimensional where \( n \geq m \).

Intp-PACK

The function pack repacks the source object of rank m into an object of rank n, where the size of the n-ranked object is equal to the size of the m-ranked object. The size along each dimension of the n-dimensional object (resulting object) will be determined by the n arguments of the function pack which yield integer scalar values, e.g., a source object with shape vector \( (5,6,4) \) can be re-packed into an object of shape \( (12,10) \), etc. The simulator interprets the n arguments and the main expression (EXPR.s), pops the objects from the
Intp-GATHER

With slice and sect functions a regular section of an object can be selected. Irregular sections which may contain some elements of each slice or sect of an object cannot be selected by using these functions. For this purpose the function gather is provided in F-code. The function gather has a binary mask, a main expression (EXPR.a) and a logical expression (EXPR.m) in its definition. The function gather stratifies the result of the main expression (EXPR.a) according to the mask and then, selects those elements of each stratification which has corresponding elements 'true' in the logical object resulting after interpretation of one of its argument, EXPR.m. It means that the rank of the logical object should be equal to the rank of each stratification (or layer) of the source object. In other words the rank of the logical object should be equal to the number of unity bits in the mask, because the source object is stratified along the dimensions which has their corresponding bits zero in the mask. The interpretation of this function follows the same logic. For example we want to gather a source object (6,8,9) with mask (1,0,1) and a logical object (6,9). The source object will be stratified according to the mask and then each layer will be masked according to the minimum size along each dimension of the layers and the logical object as shown in Figure 3.7. The dark points in the logical object shows the 'true' elements. The dotted lines show the minimum of the two lengths.

Intp-DIAG

The interpretation of diag selects the diagonal elements of certain dimensions of the object resulted by the evaluation of its argument. The diag node contains an expression and a mask. The result of the expression is stratified according to the mask as in gather and the diagonal
from each layer will be selected. For a 2-dimensional object the diagonal will contain all the elements which has both indices equal \((i=j)\). Similarly, the diagonal for an \(n\)-dimensional object for \(m\) dimensions \((m \leq n)\) will contain all the elements which has the same indices for the \(m\) dimensions.

**Intp-TRANSFORM**

The function \(\text{transform}\) is a way of indexing a multi-dimensional data object. To index an \(n\)-dimensional object we need \(n\) values one for each dimension. In \(\text{transform}\) these values are decoded in \(n\) value indexing expressions with their posterori type integer. With each indexing expression there is a mask at the node which will map the index for a dimension. \(\text{Transform}\) expands each indexing expression with its mask. The minimum of the \(n\)-expansions yields the shape of the resulting object. Projection of each indexing expression with their mask on the shape of the resulting object yields the indices for selection of the elements from the source object along that particular dimension. By using these indices the elements from the source object are selected. For example, if a \(\text{transform}\) node has followings items in its definition:

- source object with shape vector \((10,12)\),
- indexing expression 1 with shape \((7)\) and mask \((1,0)\),
- indexing expression 2 with shape \((8)\) and mask \((0,1)\),

Then the shape of the resulting object will be \((7,8)\). The contents of expression 1 will yield the first index of the elements selected from the source object. The contents of expression 2 will yield the second index of the elements selected from the source object.

**Intp-POL**

The interpretation of \(\text{pol}\), which evaluates a polynomial of an arbitrary number of (non scalar) variables, is quite similar to the \(\text{transform}\). It evaluates the shape vector of the resulting vector exactly the same way as in \(\text{transform}\) and fill the resulting object from the contents of \(n\) ‘expr.v’ and one ‘expr.c’ according to the definition.

**Intp-COMPOSE**

The function \(\text{compose}\) makes a single object from two objects. The definition of \(\text{compose}\) function has a number which decodes the dimension along which the object are being composed. The mask with each argument of this function simply provides the rank coercion mechanism as in \(\text{dyadic}\) or in \(\text{choice}\). The length is coerced to the minimum except along
the dimension of composition. Along this dimension the length of both object is added and the second object is glued there.

The interpretation of all the geometric and semi-geometric operations is really a careful game of indices. One has to be very much careful for getting and freeing memory during the interpretation of these functions. Of-course without proper understanding of the F-code semantics the interpretation is not possible.

3.4 Summing Up

The main purpose for the implementation of the simulator (F-SIM) was thorough understanding of F-code as a problem domain of the N-computer. By simulating F-code, this purpose was achieved. Therefore, the simulator was not optimized as a sequential interpreter for F-code. The experience obtained from this implementation has been quite useful while designing the N-computer and implementing its soft prototype.
Chapter 4

The N-Computer and its Processing Phase

4.1 Introduction

This chapter can be considered as a logical description of the N-computer. The description addresses all the major system level issues, e.g., local synchronization, global synchronization, output token formation and the tokens formats, etc. The N-computer is a dataflow bulk synchronous (DFBSM) machine with many processing nodes interconnected with a communication network. After describing the top level organization and function of the N-computer system the logical structure of a processing node in the N-computer is explained. A processing node is divided into six major sub-systems. Data parallel high level languages, like Fortran 90 are implemented on the N-computer via F-code [24]. To process an F-code program, the N-computer has two phases of its operation, one is processing phase and the other is initialization phase. The function of the major sub-systems of a processing node of the N-computer during the processing phase is described.

During the processing phase, inter processing nodes and intra-processing nodes tokens will flow. The structure of different tokens is defined. Then, the function of each major sub-system to manipulate the tokens is discussed. In the light of that function, the logical structure of the sub-systems is defined.

4.2 The N-Computer

The N-computer is a design for a machine based on the DFBSM model. The top-level organization of the machine is shown in Figure 4.1.

The processing system processes activities after receiving synchronization signals from the synchronization system. An activity is a unit of computational work to be carried out. There is no limit upon the size of an activity; this could be a single instruction or it could
be a whole program. An activity can be considered as a collection of threads. Therefore, the N-computer can be viewed as a dataflow machine for the execution of a unit of an activity or a unit thread in the collection of threads. After processing a unit activity, the processing system may generate a token for the output token formation system. The output token formation system generates tokens needed for the execution of the parents activities at any processing node of the N-Computer. The output tokens will be routed towards a synchronization system. When all the tokens, needed for the execution of a unit of an activity arrive at a synchronization system, it will schedule the unit for processing. The communication network is responsible for moving tokens from one processing node to the other processing nodes. The synchronization system will also be responsible for generating a global synchronization to synchronize all the processing systems (or a subset of them) at certain times during the execution of a program. After a global synchronization, the next cycle of the N-computer operation will start.

At this stage, any particular compilation strategy for the implementation of F-code on the N-computer is not assumed. This thesis looks at the higher level architectural issues and therefore the lower level issues such as instruction set and compilation strategies are not investigated. Many unresolved problems at the system level need to be studied before such details should be considered. To demonstrate the functioning of the system, our strategy is to extract the information needed at each node from an F-code program in the form of an activity tree; then by using that activity tree show the functioning of the N-computer. This
will demonstrate that the system will be able to execute an F-code program. It will also give us an estimate of the performance of the system. The results can be used to improve the performance and to optimize different parameters of the system. The operational model of the N-computer from that point of view is shown in Figure 4.2.

Data-concurrent high level languages will be implemented on the N-computer via F-code (other types of languages can be implemented with a suitable interface in the front end of this system, e.g., sequential languages using a vectorizer, etc.). The compiler will generate two things from an F-code program: first, code for each activity at each node (shown by the solid line); and second, an activity tree (shown by the broken line). The activity tree is a data structure (organized in a tree like fashion) generated from an F-code program and will be used by the synchronization system and output token formation system to generate the necessary synchronization and communication information for the distributed execution of an F-code program on the N-computer. There will only be one activity tree for a given program and every node of the N-computer will use the same tree for the generation of necessary synchronization and communication information. In case of a large activity tree (for a large program) a subset of the tree might be cached at each node from a single copy of the complete activity tree.

Each activity will consume data objects (which could be vectors, matrices or data objects of any dimensionality in an F-code program) from its children and result in a data object for its parent activity. The processing of each activity may cause side-effects and update local/remote variables. The architecture of an N-computer node for the execution of an activity tree will be explained in the next sections.

4.3 Node Structure of the N-Computer

The logical organization of an N-computer node is shown in Figure 4.3. There are six main building blocks.

1. Main control system.

2. Synchronization system.

3. Processing system.

4. Output token formation system.

5. Starting tokens emitter.


 That is why an activity is called a collection.
There are two phases in the operation of an N-computer node, the initialization phase and the processing phase. The processing phase, as is clear from its name, is needed to process the computation. As mentioned in the previous section, the N-computer is a dataflow machine for the execution of unit activities or threads. An efficient implementation of a dataflow machine demands an efficient way of resource management. Limits on resources demand a way of recycling the resources or throttling. An efficient way of throttling in a dataflow machine could be used for an efficient implementation of loops and recursion. The initialization phase in the N-computer is needed for throttling. The global synchronization as described in the DFBSM in chapter 1 will be used to initiate the throttling process in the N-computer. In both phases, different tokens will flow between these building blocks. Each building block will perform certain operations on incoming tokens and may emit tokens for some other building blocks. In the following sections, the logical structure of each token and the operation for each building block for both the processing and the initialization phase will be explained.

4.4 Processing Phase of the N-Computer

The overall operation of the node in the processing cycle will be as follows:
1. The communication interface receives tokens from other processing nodes of the N-computer and from the output token formation system of the same node and directs them either towards the synchronization system, the starting tokens emitter or the processing system of any node.

2. The synchronization system receives tokens from the communication interface, performs the necessary operation for synchronization and sends synchronized tokens to the processing system. It also sends a token and a signal to the output token formation system and the main control system, respectively.

3. The processing system receives synchronized tokens from the synchronization system and performs the processing represented by the token. It may then emit tokens to the output token formation system, to the communication interface or to the starting tokens emitter.

4. The output token formation system receives tokens from the processing system and from the synchronization system, forms output tokens and transmits them to the communication interface.
5. The starting tokens emitter receives tokens from communication interface or from the processing system and keep them until the start of the next processing phase. In the start of next processing phase it will emit the tokens to the communication interface.

6. The main control system is responsible for controlling the functioning of the main building blocks in both the processing and initialization phase. It has different sub-systems to perform functions other than just controlling the main building blocks, e.g., it has a global synchronization control sub-system. The global synchronization control system generates the global synchronization signal to all the processing nodes of the N-computer and receives the same signal from other processing nodes and performs the global synchronization function. Another sub-system in the main control system is the run-time control system. The run-time control system is responsible for keeping the run-time variables in the activity tree updated. The updating of run-time variables can be done in the processing phase or in the initialization phase.

### 4.5 Tokens

Tokens are light weight descriptors which contain data and some other information necessary to process the token. The logical structure of each token is explained in the following paragraphs.
Tokens of type token1 contain a tag (TAG1), information about the token types (TYPE1) and the contents of the token (CONTENTS1). The structure of the token is shown in Figure 4.4-a. The token's tag contains the number of the destination processing node. The token's type is needed to properly guide the token towards the synchronization system, the starting tokens emitter or the processing system at its destination.

Tokens of type token2 contain a tag (TAG2), a direction bit (d) and an operand (OP). The structure of the token is shown in Figure 4.4-b. The tag is an address of a space in memory which will be used as a rendezvous point for the synchronization of an activity. The direction bit tells whether the token is a left or right arc of a binary activity. The operand is the data which will be consumed by the activity.

Tokens of type token3 contain a descriptor for computation or activity (Act), left operand (OP1), right operand (OPr) and a tag (TAG3). The tag is the same as in token4 which points to a memory location in the output token formation system. In token3, the tag and the operand fields are optional. If an activity does not output a token, then there will be no tag with token3. If an activity is monadic, then token3 will have only one operand. The structure of token3 is shown in Figure 4.4-c.

Tokens of type token4 contain a tag (TAG3) and a data packet (OT). The structure of the token is shown in Figure 4.4-d. The tag points to a memory location in the output token formation system which will be used to temporarily store the data packet in the token (OT).

Tokens of type token5 contain a tag (TAG3) and a data packet (DATA). The tag is the same as in token3 and token4. The data is the result of some computation done by the processing system. The structure of token5 is shown in Figure 4.4-e. TAG3 is needed to support the execution of more than one unit of activity or more than one activity at same time by the processing system. The tag will guide the result of each processed unit to get a tag for the output token from the output token storage (OTS) in the output token formation system. The basic assumption in providing TAG3 in token3 and 4 is that the size of OTS is much bigger than the number of activities which can run in parallel at a processing node. Otherwise, the OT field in token4 can be used as a tag in token3, as well. It is also possible to route the OT field of token4 via the processing system by adding it with token3. It appears that this is not an efficient way of implementing the system. The OTS can be implemented as a two stage memory, one being large and slow and the other being small and fast with the necessary mapping logic from the slower to the faster memory. The faster memory can be considered as an on chip cache. All the output tokens are stored in the slower memory and while the processing system processes activities, the output tokens can be loaded from the slower memory to the faster memory. In this case, the access time of output tokens from the slower memory to the faster memory can be overlapped with the processing time.
if the token4 and token3 are transmitted separately. Thus adding the OT field with token3 and not transmitting token4 will be not be an efficient design, since, it will eliminate the overlapping in processing and output token formation. Therefore, there is a need for token4 and the tag (TAG3) with the input tokens to the output token formation system and the input tokens to the processing system.

Token6 contains a type field (TYPE2) and a contents field (CONTENTS2) as shown in Figure 4.4-f. The type field will be used to determine the communication type of a token. Then accordingly the contents will be transmitted towards its destination. The contents of token6 will be a token1. All the possible types of token (token6) are shown in Figure 4.5. Token7 is a subset of token1 which contains the type field (TYPE1) and a data packet (CONTENTS1) as shown in Figure 4.4-g, just like token1.

4.6 Communication Interface

Although the communication interface and its functioning is very much related to the communication network used in the implementation of the N-computer, there is an important part of the communication interface which is related with the logical functioning of the N-computer node. The functioning of that part of the communication interface is as follows:

- Direct incoming tokens to the appropriate systems by interpreting TYPE1 field of token1 or L field of token6.
- Decompose the compound tokens into simple tokens.
• Form broadcast or point-to-point tokens for the communication network.

When receiving a token of type $token_1$ from the communication network it strips off $TAG_1$ and then depending upon the value of field $TYPE_1$, the following actions will be taken.

• If the token is of $Proc$ type (for processing system), it simply directs the contents ($CONTENT_1$) of the token to the processing system.

• If the token is of $STE$ type (for starting tokens emitter), it strips off the type and directs the contents to the starting tokens emitter.

• If the token is of $ETS$ type (for synchronization system), then the contents will have two sub-fields. One is the decomposition field ($decomp\_type$) and the other is data field ($DATA_1$), as shown in Figure 4.6. The data field of an ETS token is further divided into sub-fields, depending upon the type of the decomposition field. If the decomposition field is $SIMPLE$, the data field will have two sub-fields, the tag ($t$) field and the data field ($DATA_2$). If the decomposition field is $COMPOUND$, the data field ($DATA_1$) is further divided into four sub-fields, i.e., the tag field ($t$), the increment field ($I$), the number field ($N$) and the data field ($DATA_2$).

If the decomposition field is $SIMPLE$, the data field of the token ($DATA_1$) will be transmitted to the synchronization system. If it is $COMPOUND$, by using tag ($t$), increment ($I$) and the number ($N$), the communication interface makes $N$ tokens with
Figure 4.7: Token6 and its contents (CONTENTS2) for broadcast and point-to-point communication type (TYPE2).

The communication interface receives tokens from the output token formation system (Token6) and depending upon the type (TYPE2), the following actions will be taken. The token format is shown in Figure 4.7.

- If the type is broadcast, then the contents (CONTENTS2) will be broadcasted to all the processing nodes of the N-computer.

- If the type is point-to-point, then the contents (CONTENTS2) will have two sub-fields, named local/remote field (L) and data field (DATA3). If the local/remote field is set to local, then the data field will have two sub-fields named type field and contents field. These two fields will be similar to those in token1. For these tokens, the communication interface performs the same functioning as for token1 (described above).

If the local/remote field is remote, then the data will contain token1, as shown in Figure 4.7 and will be transmitted to the communication network.

Token7 is a special form of token1 obtained after stripping of the type (TYPE1). When the communication interface receives tokens from the starting tokens emitter, the token will
be processed exactly the same way as token1 when its type is ETS, as described above. The STE will always output ETS type tokens for local synchronization system. The type in token7, as shown in Figure 4.4-g, is a decomposition field.

The communication interface may also receive tokens from the processing system as shown by the dotted line in Figure 4.3. Those tokens will be the same as token6 and the functioning of the interface for token6 has already been described. The system level structure of the communication interface and the system functioning in pseudo OCCAM is described in Appendix A.2.1.

4.7 Synchronization System

The synchronization system is responsible for the synchronization and scheduling of all the local units of activities in an activity tree. This system also sends information to the output token formation system (token4) about the output tokens of a synchronized unit of an activity. The synchronization of units of activities is of a dataflow type, i.e., when all the operands for a unit activity are available it gets synchronized and scheduled for processing.

To achieve a dataflow synchronization for the execution of a unit of an activity, each unit amongst a set of units requires a unique tag. Each unit activity will generate some data which flow towards its parents activities, guided by the tag. Let us call a tag with data a token. Hence the tokens will flow from a source unit activity towards a destination unit activity, and when all the tokens for a destination unit activity arrive, the unit will be
synchronized and will be scheduled for processing. The waiting/matching function to achieve the dataflow synchronization will be implemented by using the ETS concept as introduced in [82]. The tag will explicitly point to a memory location which will act as a rendezvous point for synchronization. The structure of the synchronization system is shown in Figure 4.8.

The ETS is a linearly addressed storage space, of which each location is augmented with an extra bit to manipulate the state of the location. Augmented with necessary control circuitry, it can be used as a dataflow synchronizer. The logical definition of the ETS in our system is as follows:

**Definition:** The storage for an ETS is an array of locations $M$, such that the $i^{th}$ location, $M[i]$, contains $p.m$, where $m$ is a fixed size multiple element field and $p$ is the presence state of the location $M[i]$.

The presence state, $p$, may be manipulated independently of the location's other fields, $m$. The manipulation is an atomic read-modify-write cycle where 'modify' is a state transition function applied to $p$. There are two operations on $m$: read and write. Each field of $m$ may be manipulated independently of the others.

**Definition:** The ETS multiple element field, $m$, consists of the following elements (sub fields):

1. An activity field, $Act$, is a fixed-size storage space associated with the token's tag field and will be used to keep information about an activity which will be executed after synchronization.

2. An operand field, $OP$, is a fixed-size storage space associated with the token's tag field. This will be used to store the operand of a semi-synchronized activity\(^2\) in the ETS\(^3\).

3. A global synchronization count down field ($GSCD_{field}$), which encodes whether the generation of the synchronized token will count down the global synchronization counter register, $GSCR1$, in the global synchronization generator. This will be used in the implementation of the choice function of the $f$-code.

4. An output token field, $OT$, is a fixed-size storage space, associated with the same token, and used to keep the pointer to the output tokens for the activity, stored in the output token storage.

\(^2\)The semi-synchronized activity is a binary activity for which one token has arrived: i.e., an activity associated with the ETS location for which the presence field is full.

\(^3\)The OP field could be of a variable size but to keep the ETS control and organization algorithm simple, this is kept fixed. The size of $OP$ will be according to the maximum size of the $OP$ field in the token\(^2\).
It is possible to implement the above logical description of the ETS in more than one way. A straightforward way is to have the memory organized in multiple fields as shown in Figure 4.9. In this case the tag \((t)\) will simply point to a memory location with more than one field and the function of the synchronization system after receiving incoming token (token2) in the light of Figure 4.8 and 4.9, will be as follows:

- If the presence field \((p)\) of the ETS location pointed by the tag \((t)\) is reset, then store the OP field of the token in the OP field of the ETS location.

- If presence field \((p)\) of the ETS location pointed by the tag is set, then:
  - access the OP field, make a token \((\text{TOKEN3})\) and output the token to the processing system;
  - if the activity is going to output a token, then access the OT field, make a token \((\text{TOKEN4})\) and output the token to the output token formation system;
  - reset the presence field;
  - count down the global synchronization counter register which contains the number of tokens to be matched before the generation of global synchronization.

- If the global synchronization register is zero, send the global synchronization signal to the main control system of the same processing node.

There could be a situation where more than one data element of the same activity will be executed at the same processor. In this situation it may be desirable to keep the activity (act), global synchronization field (GSCD) and the output token field (OT) for all the data
elements in one place and make it possible that every synchronized token will get these fields properly. An organization for such implementation is shown in Figure 4.10-a. The OTS locations containing output tokens for all the data elements are considered as a frame and the OT field in storage2 is the starting address of that frame. The parameter OT_INC in stage1 storage is the displacement of the output tokens for the current data element from the start of the frame. This implementation may save $O(\log_2(\text{number_of_elements}))$ extra bits for each ETS location as compared with the first implementation at the cost of an extra circuitry for the loading of activity, global synchronization count down and output token field from stage2 storage. It appears that the two stage implementation as shown in Figure 4.10-a is going to be more efficient when compared to the one stage implementation as shown in Figure 4.9. However, there are some problems with this implementation. First of all the problem of fixing the space for $t_2$ and OT INC. The size of OT INC may approach the size of OT in stage2 storage, since OT is already going to be a displacement from the start of the output token storage frame. Since, the system is going to be implemented in hardware, dynamically fixing the space for these fields will add an extra complexity in the system. If the number of elements at one processing node is not much, then, the savings in the storage space with the addition of two stage storage will not be significant. Therefore, it is difficult to say that the two stage ETS is going to be more efficient without the lower level simulation study. Therefore, to keep the description of the system simple, the one stage ETS implementation is kept as a reference when describing the system in details.

It is also possible to add the tag for second stage memory ($t_2$) in the incoming token, i.e., combining the tag of stage1 storage ($t$) and stage2 storage ($t_2$) in the tag of token2 as shown in Figure 4.10-b. An important design rule for the N-computer is that communication is the most expensive parameter of the system. Therefore if it is possible to get the required information locally then never attempt to get it from a remote node. In the light of this principle, the two stage implementation of the ETS with a two stage tag, as shown in Figure 4.10-b is not desirable.

A sub-system in the synchronization system is the global synch generator. It consists of a register named a global synchronization count register (GSCR1) and an associated control circuitry. The size of the register will depend on the size of the ETS. The register can be counted up or down depending upon the global synchronization count up signal (GSCUSg1) or the global synchronization count down signal (GSCDSg1).

The working of the synchronization system in pseudo OCCAM is described in Appendix A.2.2.
4.8 The Processing System

The processing system will receive synchronized tokens from the synchronization system and process the computation represented by the token. The processing system could be a von-Neumann style processor and memory unit or a combination of many functional units with a simpler control. The compiler will generate the activity tree and code according to the processing system. In either case, the activity will represent a certain amount of computation to be done. In the first case, the computation will be coded in the memory which the CPU will execute. In the second case, the computation will be a pointer to a particular functional unit, which will perform operations on the data in the token. Therefore, logically, the operation of the processing system will be the same in both cases. It can be defined as follows:

Definition: The processing system contains appropriate circuitry to perform the computation represented by the Act in the incoming token, ETS.PROC.TOKEN, on the data in the token (OP$_{left}$, OP$_{right}$). For the time the computation is done, the ACT$_{tag}$ of the ETS.PROC.TOKEN will be kept at the processor. When it finishes the computation, it will output the result in the form of output token, PROC.OTF.TOKEN.

To demonstrate the functioning of the N-computer, it is assumed that the processing system contains $M$ different functional units; each one is capable of performing computation represented by one activity out of all possible activities. The structure of the processing
Figure 4.11: Processing system of the N-computer

The functioning of the $i$th functional unit of a processing node, after getting synchronized token from the dispatcher, will be as follows. The system description in pseudo OCCAM is given in Appendix A.2.3.

- Start the processing on the operands.
- Store the tag (TAG3) in an output tag buffer, if there is any.
- After finishing the processing, output the result and the tag to the output token formation system. It may be possible that the processing of an activity does not output any result. In this case, there will be no tag with the incoming token and there will be no output token.

4.9 Output Token formation System

In the processing phase, the output token formation system receives results from the processing system (token5) and tokens from the synchronization system (token4) and generates output tokens (token6) as shown in Figure 4.3. It will also send signals to the global synchronization control system for the generation of global synchronization.

An important component of the output token generation system is the output token's storage, OTS, and its organization. The storage for output tokens is divided into two sections. The output token storage 1, OTS1, and the output tokens storage 2, OTS2.
Why is there a need for two sections (OTS1 and OTS2) in OTS?

All the units of an activity may not generate the same number of output tokens due to geometric operations and length coercion of an F-code program. These types of units can be termed non-uniform units of an activity. Each activity may have more than one parent in the activity tree. For one parent, all the data elements may be uniform and for other it may not. When a result of a unit is computed, output tokens for all its parents will be generated.

In the initialization phase, the OTS and ETS are initialized independently. ETS could be much faster than the OTS and the amount of OTS needed could be much larger than the ETS at a node to achieve a reasonable performance, since a unit activity (which uses one ETS location for its synchronization) may have to generate more than one output token. Also for output tokens, there is a need to store more information, e.g., about the number of parents, types of the output tokens for each parent, communication type of the token for each parent, decomposition type of each token, decomposition parameters, number of tokens, tags for each token and in case of choice, tags for true and false parents etc.

In the ETS initialization there is a need to initialize the OT field of each ETS location. To initialize the OT field of ETS, the system must know the address of output token storage for the output tokens for each unit of an activity. From the address of the start of output token storage for the output tokens of the first unit of an activity (pointer to the start of output token storage) it is not straightforward to find the OT field for every unit. For that one of the following strategy may be adopted.

1. Keep the number of output tokens for each unit of an activity with the activity.

2. Evaluate the exact number of output tokens for every unit activity at ETS initialization time.

3. Keep the maximum number of output tokens for a data element of an activity with the activity.

The first option needs a lot of storage space to store the activity tree since every data element of an object requires a number. Therefore, for each activity there will be a need of storage space equal to the extent of the result to initialize the OT field of ETS. Since, each processing node has a copy of the activity tree, this will be multiplied by the number of processing nodes for the overall storage space for an activity. Therefore, this is not a practical option.

The second option will increase the complexity of the ETS initializer and make the initializer slower. The third option will waste the OTS space for non-uniform units. Since, OTS is an important resource in the N-computer, wasting it could be quite expensive. Thus
none of the above options looks attractive. An activity with non-uniform units and the OTS is shown in Figure 4.12.

By dividing the output token storage into two sections, OTS1 and OTS2, the need for evaluating the exact number of output tokens for every data element at ETS initialization time can be eliminated. Thus the drawbacks of any one of the above option can be eliminated. The two sections model of OTS is shown in Figure 4.13. Since, the ETS initializer knows that for OT field of each element of the activity, it is just going to be incremented by a fixed amount which can be determined by the number of parents, to get the OTS2 for the next data element of the activity.

Addition of OTS3

The ETS model of synchronization is ideal for binary events. If, there is an associative/commutative operation applied to more than two data elements, then, to evaluate the same operation for more than one time, more than one ETS location are required. If the number of data elements is significantly large, then we may run-out of ETS space. It is possible to execute the same binary operation (associative and commutative) for more than one time by using a single ETS location, if a counter is associated with the location. The counter is not necessarily implemented with the ETS. The best place for such a counter is

---

4 This situation may arise if an F-code program is going to reduce a large object.
the output token formation system. Because, until the operation is performed on all the data elements, the result will be fed to the same activity and at the end, the result will be guided towards some other activity. Therefore, a counter storage (OTS3) is added in the OTS as shown in Figure 4.14. The addition of OTS3 demands an extra piece of information in OTS1 for each activity. That piece of information will determine whether there is a counter in the OTS3 for the generation of output tokens for current activity or not. This can be implemented either as a single bit field associated with the first location of OTS1 for each activity or as a separate location which will store the pointer to the counter register in OTS3. To keep the description of the system simple, it is assumed as the pointer to the counter register (counter_pt) is added in OTS1 for each activity. To describe the other option (single bit with the first location), the pointer to OTS3 can be divided into two fields, first is the single bit field and if the first field is set then the second field will be the pointer to OTS3. Otherwise there will be no pointer to OTS3 (no second field). In this way the second option can be simulated within the first option but with some extra detail. Also, the initializer for ETS, to initialize OT field will need a check operation on the activity and if the activity requires second field in OTS3 (the OTS3 pointer) then accordingly the next OT field can be updated.

**Definition:** The storage for OTS1 is an array of m+2 locations, where m is the number of parents of the activity. The first location, named count_pt of the array, if it is more than zero, contains a pointer to a location in OTS3. The second location of the array contains m,
Figure 4.14: Output token storage model with three storage sections. One is for all the parents and for the counter pointer (OTS1), the other is for the counter registers and local tags (OTS3) and the last one is for all the tokens of a result of an activity (OTS2) when the counter is zero.

while the other m locations contain the pointers to the OTS2.

Definition: The storage for OTS2 is an array of locations, M+1, to store M output tokens, such that the first location, M[0], contains the following fields:

1. A number field, n, is a fixed size storage space, associated with the OTS1's pointer. It registers how many tags are associated with the pointer.

2. A choice field, c, is also a fixed size storage location associated with the output token's pointer. The c field will provide the facility to make a choice between the output tags from a given set of tags, to generate output tokens.

The result of computation will be combined with the n tags to form n tokens for a single parent, which will be transmitted by the output token formation system.
3. A global synchronization count down field, \( \text{GSCDfield}_2 \), which encodes whether the generation of output tokens will count down the \( \text{GSCR2} \) in the global synchronization generator. This will contribute to the implementation of the choice function of F-code.

4. A direction field, \( d \), needed to decode whether the token will be transmitted as a left or a right child of a binary activity.

5. A template field which contains three subfields. Field 1 to decode communication type of the output token - \( \text{com.type} \), 2- decomposition type \( \text{decomp.type} \) and 3- type. All the parameters used to classify a token are shown in Figure 4.5.

6. Two fields for decomposition parameters if the decomposition type is compound. These are increment \( (I) \) and the number \( (N) \).

The remaining \( M \) fields of the array are known as tag fields. Every location of the tag field is a storage space used to store a tag, \( t \). It contains \( n \) two-elements fields needed to store a complete network tag. If the communication type is point-to-point, then the processor field \( (p) \) and the tag field \( (t) \), otherwise, only the tag \( (t) \) field.

**Definition:** Each word of OTS3, pointed by the field count.pt in OTS1, will be an array of two locations. The first location will be a counter register, which will be used by the output token formation system to output a token. The next location will be a local tag which will be used for the output token until the counter register is more than zero.

The output token storage space in OTS2 is organized in such a way that if the number of tokens to be transmitted, \( n \), is more than one, then the \( n \) spaces can be concatenated to store \( n \) tags. In the first field of the output token storage, if the choice field, \( c \) is set, then each location of the tag field will contain two tags, the first one named as true tag and the second one named as false tag. Depending upon the result of the computation, either true or false tags will be transmitted as tokens. It also has a global synchronization generator with a similar organization as in the synchronization system.

The structure of the output token formation system is shown in Figure 4.15. OTS is the storage organized into three sections, OTS1, OTS2 and OTS3, as explained above. The OT is a small amount of explicit token storage without presence bits. This storage will be used to temporarily store the OT field of the token from the synchronization system (token4) until the token from the processing system (token5) arrives. The tag in token4 (TAG3) points to a location in the OT store. The synchronization system maintains a pool of pointers to the OT store. The tag for the token from synchronization system to the processing system (token3) and token4 is collected from the same pool. It is assumed that the token4 will always arrive before the token5. Therefore, there is no need to provide a presence bit for
each location of OT storage. The making of the output tokens can be initiated with the
arrival of the token5. When token5 arrives from the processing system, the address of the
OT is sent back to the pool in the synchronization system. The OT_address link between the
output token formation system and the synchronization system is not shown in Figure 4.3,
just to keep the description of the system simple. In the processing phase the
output token formation system will work as follows:

- Input tokens from the synchronization system and the processing system in parallel.
- When both tokens arrive, check the counter pointer field of OTS1. If it is more than
  zero;
  - Make a token by accessing the local tag field of OTS3 and output the token.
  - Decrement the counter register in the OTS3.
- If the pointer field is zero, then;
  - Access the number of parents and the address for the output tokens for each
    parent from the OTS1.
  - According to the number of parents (n) and from the pointer for the tokens of a
    parent activity, make tokens (say, m tokens) and output them as follows.
  * Get the first field of m+1 array of memory locations of OTS2. The first field
    consists of multiple elements as explained earlier.
  * Then for remaining m locations;
    - If the choice field, c, is set, then access the true or false tag according to
      the result of computation. Otherwise, only access the tag.
    - Depending upon the communication type, access the processor field in
      the tag.
    - If the token is compound, access the decomposition parameters (I and
      N).
    - Output the token.
    - Depending upon the GSCD_field2, send signal to the global synch generator.

Since, the loading of output tokens does not necessarily need the arrival of token from
the processing system, it can be done in parallel to the processing of a unit of an activity. At
the arrival of the result from the processing system all the loaded tokens can be transmitted
to the communication interface. To implement this the same memory in the output token formation system, which is used to temporarily store the OT field of the token from the synchronization system (token4) with a slightly modification in its organization, can be used. The new organization of OT store and the output token formation system is shown in Figure 4.16. It is a small explicit token storage with the presence bit for synchronization for each location. The logic for the functioning of the system without waiting for the arrival of a token from the processing system will be as follows:

- At the arrival of the token from the synchronization system:
  
  - If the counter pointer of OTS1 is more than zero and then counter register in OTS3 is more than zero;
    
    * Load the local tag from the OTS3 in an output buffer.
    * Decrement the counter register in OTS3.
  
  - Otherwise,
    
    * Load all the tags from the output tokens storage (OTS1 and OTS2, as explained above) in an output buffer;
Figure 4.16: Structure of the output token formation system with a modified OT store.

* If the presence bit of the OT store is reset, store the buffer's address or tag in the location's `buffer_tag/Data` field and set the presence bit;

* If the presence bit is set, take the data from the location's `buffer_tag/Data` field and assemble with the output tokens in a buffer and output all the tokens. Reset the presence bit and send back the address of that location of the OT to the synchronization system.

• At the arrival of the token from the processing system (token5):
  
  – If the presence bit is reset, store the result in the location’s `buffer_tag/Data` field and set the presence bit;

  – If the presence bit is set, load the tag of the buffer which contains all the output tokens, from the location's `buffer_tag/Data` field and assemble the tokens and output them. Reset the presence bit and send back the address of that location of the OT to the synchronization system.

The loading of output tokens in parallel to the processing of an activity will make it possible to implement the output token storage in a slower memory. The detailed functioning of the system in pseudo OCCAM is given in Appendix A.2.4.
4.10 Starting Tokens Emitter

In the N-computer, there is a need to emit tokens at the start of each processing cycle for the execution of the activity tree in the current global synchronization. Tokens which will be emitted at the start of each processing cycle, without waiting for any processing to be done, are named starting tokens or leaves tokens of an activity tree. The following two ways could support this:

- Define a new starting token storage with its control process to support the storage and transmission of the starting tokens. There will be some activities in the activity tree (constants) which will be used to initialize the starting tokens storage. The starting tokens (transition tokens) are those which travel between a global synchronization normally by the end of current global synchronization and the start of the next global synchronization cycle.

- Modify the output token formation system and the OTS to store the leaves tokens and access them at the proper time.

Since, this research addresses the logical specifications of the system and some higher level implementation issues, therefore in their logical description both possibilities look the same. To describe the system, the first option was chosen. It keeps the overall system
simpler. The starting tokens emitter is shown in Figure 4.17. The STE storage is simply
a memory organized in such a way that each location stores an OUTTOKEN. In the start
of each processing cycle, the STE storage is read by the control system and output to the
communication interface. The STE counter keeps track of how many memory locations will
be read and output. During the processing cycle, the tokens arrive from the communication
interface and, then, they will be written in the STE storage. Every write operation is guided
by the counter register and the control system, increments it after every write to the STE
storage. The system in pseudo OCCAM is described in Appendix A.2.5.

4.11 Global Synchronization Control System (main-
control system)

This sub-system in the main control system will synchronize all the processors (or a sub-
set of them) in the network at certain times during the execution of a program. There
will be two occasions when global synchronization is required. One is when there is not
enough system resources available and there is a need to recycle all the tags. This type of
global synchronization is termed Physical Global Synchronization (PGS). The other type
of global synchronization is needed due to the nature of the program, e.g., when there is
a need to synchronize all the processors before accessing a global variable or to re-use the
resources for loops or recursion, etc. This type of synchronization is termed Logical Global

Figure 4.18: Global synchronization control system: logical structure
Synchronization (LGS).

When a processing node finishes its work for the current cycle, it will broadcast a PGS or LGS to all the processors (or a subset of them). There are two conditions for a processing node to finish its work which are as follows:

1. When all the activities in the synchronization system get synchronized.

2. When all the output tokens in the output token formation system are generated.

Since the N-computer is not a pure dataflow machine, both conditions are necessary to guarantee the correct access of variables. At each node, the global synchronization control system (GSCS) receives all the broadcasted signals from all the nodes, and generates a global synchronized signal. The GSCS has a link from all the processors in the network. The ith processor in a P processors N-computer will have the following logical links:

\[ G_{Simj} ; \quad 0 \leq j < P, \quad j \neq i \]

From all processors (except from processor i) to processor i of the N-computer

\[ G_{Souti} ; \quad \text{From processor i to all other processors of the N-computer} \]

It has links to the synchronization system (LS_{control}) and the output token formation system (OT_{control}). The logical structure of the system is shown in Figure 4.18. The above GSin and GSout are logical links and can be implemented on one physical link. The system functioning in pseudo OCCAM is described in Appendix A.2.6.

### 4.12 Run-time Control System (main-control system)

Since an activity is a collection of threads, it contains information about the extent of that activity together with other parameters, e.g., a pointer to the start of ETS (ETS_{pt_{start}}) and a pointer to the start of OTS (OTS_{pt_{start}}). The extent could be a run-time parameter which would be set by another activity tree.

The run-time control system will initialize the run-time parameters in the activity tree after getting run-time extents. The initiation of the run-time control system can be fully supported by the dataflow synchronization by using ETS. In this case, the run-time control system will act as an activity which will consume the extent and initialize the run-time parameters in the ETS.

It is also possible to use a global synchronization signal to initiate the run-time control system and the initialization of the run-time parameters. In some cases, this may be desirable. Suppose there is a need for a global synchronization to recycle the ETS and OTS.
(physical global synchronization) for the evaluation of some branches of an activity tree. An activity before the global synchronization also evaluates the run-time extent needed for some other branch in future. In this case, there is no need to use ETS and OTS resources to evaluate the run-time control system. The global synchronization may initiate the process. In this case, some memory locations can be fixed for run-time extents and the run-time control system will load the extents from those memory locations.

A compiler may choose an appropriate way to initiate the initialization of run-time parameters in an activity tree. Either way, the logic of the process after getting run-time extents will be the same, which is as follows:

- Coerce the extent of the activity tree if necessary. Once the extent of an activity is known, there may be a need to propagate the geometric activities (in terms of an F-code tree, the default length reduction for an expression as well) down towards the leaves of the tree. The extent coercion can be considered as the conversion of an eager activity tree into a lazy activity tree, i.e., a tree will evaluate only the required data elements.

- Initialize the run-time parameters of activities in the activity tree which are as follows:

  1. Pointer to the start of ETS.
  2. Pointer to the start of OTS.
  3. Output tokens' type, i.e., STE or ETS tokens.
  4. Output tokens' communication type, i.e., point to point or broadcast.
  5. Output tokens' decomposition type, i.e., compound or simple. A compound token will be converted into more than one simple token at its destination.
  6. In the case of compound tokens, the break down parameters for the compound tokens will be initialized. These are the increment \( I \) and the number \( N \). These parameters will be used to make \( N \) tokens from one compound token.
  7. The same number of output tokens for each result of an activity for a parent activity parameter \( N \) will be initialized. In an activity tree, it is possible that each result of an activity is not going to generate the same number of output tokens for a parent. If the information, that each result is going to generate the same number of tokens for a parent, is kept with the activity then the OTS initialization can be optimized. This will be explained in a section about OTS initialization in the next chapter.
The structure of the system is shown in Figure 4.19. If the initialization of the run-time parameters is done by an ETS activity, then there will be a need to generate an output token when it finishes the initialization, to synchronize the process after the run-time control activity. Since the processes, needed to be synchronized after the run-time control system, are the initialization of ETS and OTS. In order to do this, the output tokens count down the global synchronization counter register (GSCR1) which will contribute in the generation of global synchronization. The global synchronization will then initiate the ETS and OTS initialization process.

It is also possible to provide a logical flag to be checked by the ETS and OTS initializer before starting the initialization. The flag can be set by the run-time control system. The same flag can be used for synchronization in case the run-time control system is initiated by the global synchronization. In its implementation in hardware, the flag can be implemented on the control signal (RT\textsubscript{control}, needed to control the functioning of the sub-system) from the main control system to the run-time control sub-system and there will be no need to provide an extra bit of memory for this. The logical description of the run-time control system in pseudo OCCAM is described in Appendix A.2.7.

4.13 Summing Up

The organization of a processing node in the N-computer has been described in this chapter. The processing node consists of six sub-systems, i.e., main control system, synchronization system, processing system, output token formation system, starting tokens emitter and com-
munication interface. There are two phases of operation in the N-computer, i.e., processing phase and the initializing phase. In the processing phase, different tokens will flow between these sub-systems of the same processing node or remote processing nodes. The logical structure of each token has been defined. To process these tokens, the logical organization of each sub-system in the processing phase has been described. Then, the functioning of each sub-system has been explained.

Each location of the explicit token storage (ETS) in the synchronization system has three piece of information other than the presence bit. These are the computation information (act), operands (OP) and a pointer (OT) to the output token storage (OTS).

Each unit of an activity may not generate the same number of output tokens, and, therefore, to keep the ETS initialization simpler in order to initialize OT field for each unit of an activity, output token storage (OTS) has been divided into two stages, i.e., OTS1 and OTS2.

An ETS location is a binary synchronizer. Therefore, to process a computation which is even commutative and associative on more than two data elements, more than one ETS location are required. To address this problem of a binary synchronizer, a counter storage (OTS3) is added in the output token formation system. With the addition of OTS3, it is possible to use one ETS location for synchronization and scheduling of a commutative and associative operation for any number of data elements. The same feature can also be used to schedule a macroactor (macroactor is described in section 1.5.4 in chapter 1) by appropriately arranging the incoming tokens for the microactor.

To conclude with, the logical organization of a processing node of the N-computer, which consists of six major sub-systems, and the functioning of each sub-system have been described in this chapter. In the next chapter, the second phase of the operation cycle of the N-computer, i.e., initialization, will be discussed.
Chapter 5

Initialization Phase of the N-Computer

5.1 Introduction

As mentioned in the previous chapter, the N-computer is an implementation of a bulk synchronous dataflow model. The implementation supports the model by providing lower level (in hardware) data flow synchronization at each processing node and a global synchronization to synchronize all the processing nodes for the execution of an F-code program. The design of the N-computer and the dataflow processing of the activities have been explained in the previous chapter. To achieve the dataflow execution of the activities at each processing node of the N-computer, there is a need to initialize resources, such as ETS, OTS and STE. In the first section of this chapter, the question of why there is a need to initialize the resources, is answered.

After developing the need for the initialization of resources, an operation cycle of the N-computer is explained without going into lower level details. Delays in the operation cycle are highlighted. The time to initialize ETS and OTS appears to cause major delay. To keep these delays to their minimum possible values, guidelines for the design of the system are highlighted.

For every activity in the activity tree, there is a need either to initialize ETS or STE. The process for initializing ETS for an activity tree is described. It is noted that the initialization of ETS is a simple process and can be implemented in hardware. To create leaves tokens for the dataflow execution of an activity tree, the initialization of the starting token emitter (STE) is described. Due to dynamic shape of the objects of an F-code program, there may be a need to initialize OTS to support dataflow execution of activities. Therefore, the logic for the process to initialize OTS is described. It will be observed that the OTS initialization is relatively a complex process and every effort must be taken to optimize it.
5.2 Need to Initialize

A dataflow machine implemented in hardware demands a limit on resources, e.g., token space. If the token space is limited to a reasonable amount in such a way that the waiting/matching function can be implemented on a chip by using fast memory/logic then the performance of the machine can be improved significantly. This approach is adopted in the design of the N-computer. Since, the waiting/matching space is limited we need to re-use it. The global synchronization in the N-computer is used to recycle all the resources including the waiting/matching store.

5.2.1 Need to Initialize ETS

To execute another part of the activity tree after global synchronization there is a need to initialize the waiting/matching store (ETS) of the synchronization system of the N-computer for the following reasons:

1. To reduce the load on a communication network, all the information needed for the execution of an activity should not be kept in an incoming token. Therefore, a token will carry only the information which cannot be generated locally from the activity tree. All the other information, such as activity (the computation which will be processed), the global synchronization field and information about the output tokens will be stored in the ETS.

   Therefore, during the recycling, in general, there will be a need to initialize the waiting/matching store with this information. Sometimes, it might not be required to initialize the ETS after every global synchronization, which will be highlighted next.

2. In general, there will not be the same activity tree to be executed after each global synchronization.

   For a data parallel program, the data concurrency could be thought of as a loop and each scalar tree can be evaluated by using the same resources with the same initialization values and the need for initialization after each global synchronization can be eliminated. This approach tends to limit the exploitation of data parallelism and appears to rely more on the process parallelism to act as an excess parallelism to hide the communication and synchronization overheads in the dataflow execution of the program. Since, the process parallelism is very small compared to the data parallelism in an SPMD style program (which is the problem domain of the N-computer), this approach does not look attractive. However, in some cases, an F-code tree may be optimized in the form of a loop of a data parallel tree and for each iteration the need
to initialize the resources could be eliminated. In those situations, it is possible to bypass the initialization of ETS either by keeping a piece of information with the activity or by defining a loop activity which will be interpreted by the initializer. That option will be provided in the N-computer. Even in this situation there will be a need to initialize ETS for the first iteration of the loop.

Thus, in either case, there is a need to initialize ETS. It may not be required after each global synchronization.

5.2.2 Need to Initialize OTS

If each processing system in the N-computer is of a von-Neumann style, then due to the dynamic extent of an activity tree it would not always be possible to generate code for each data element of an activity in such a way that the compiled code will generate output tokens. In some cases, it is possible to compile the process for output token formation if the tree is static. Then by sending information about the output tokens for each data element of an activity from the ETS to the processing system, the processing of the element will generate the output tokens directly. For this type of compiled code, the execution of a program on different sizes of machines demands the modification of the output token process in the compiled code. This can be done by parameterizing the compiled code with machine size.

It is not always possible to compile output tokens and may not even be desirable, since the logic for the generation of output tokens will be the same and it might be desirable to implement it in hardware, separately. For this implementation, there is a need to know the complexity of the OTS initialization. After that it can be decided at compile time, where the output tokens generation for an activity tree should be performed. To support the execution of a dynamic extent tree, the system has to provide with OTS initialization at run-time, and the best time for the OTS initialization is at the recycle or global synchronization time in the operation cycle of the N-computer, because, the global synchronization will tell the system that the resources needed to be initialized are available now, i.e., the OTS is free. Therefore, the generation of a data base of output tokens from the activity tree during the initialization phase is studied. If it is desirable to compile output tokens, then a compiler may implement the same logic for the compilation of output tokens.

In the initialization phase, the ETS in the synchronization system, the OTS1 and OTS2 in the output tokens formation system and the global synchronization counter register in the synchronization and output token formation systems may be initialized from the activity tree at each processing node of the N-computer. During the initialization, no processing node

---

1In the N-computer, there is an option that if a compiler wants to combine the computation and the output tokens in the compiled code of an activity then it can do so.
Figure 5.1: A part of a typical activity tree. The shape of each activity is shown along with the activity. M is the mapping rule which will map a child object of extent 1 into its parent’s object of extent 2.

will execute activities in the activity tree. When the initialization finishes, the execution of activities will start.

5.3 Operation Cycle of the N-Computer

Let us assume an activity tree with multi-dimensional objects (multi-dimensional collections) to represent each data element, as shown in Figure 5.1. Just like an F-code tree it uses the same principles to map each element of a child activity into the elements of its parent activity. The strategy for the execution of the activity tree on multi-processing nodes N-computer is as follows:

- Distribute the units\(^2\) of each activity to all the processing nodes of the N-computer with a distribution function.
- Initialize the synchronization and output tokens data base (ETS and OTS) for the data elements of each activity for all the activities in a part of the activity tree at a processing node.
- Start the processing phase in which each data element of an activity will be scheduled for execution with dataflow synchronization.

\(^2\)A unit may contain more than one data element of the object.
In an activity, each element is represented by a multi-index (ME_index) according to the shape of the activity. In the initialization process, the distribution function will map certain units of an activity to a processing node. For an N processing node N-computer, the distribution function, $F$, can be stated, in general, as:

$$F(ME\_index, \text{Shape}, N) \implies P_i$$

$P_i$ is the processing node number where a data element defined by $ME\_index$ will be executed.

The inverse of $F$ must also exist, which can be defined as follows:

$$F^{-1}(P_i, \text{shape}, N) \implies \text{All the } ME\_indices, \text{ which belong to the collection having extent, shape, mapped to the processor } P_i \text{ in a network of } N \text{ processors}$$

There is a need for another function which is as follows:

$$G(P_i, \text{shape}, N) \implies \text{No\_of\_data\_elements}$$

The function $G$ returns the number of data elements at $P_i$, it does not return the indices of all the elements like $F^{-1}$. $G$ is defined separately, since, let $m$ be the number of elements at a processing node, depending upon the distribution function ($F$), sometimes there is no need to perform $O(m)$ operations to know the number of elements at a processor. To initialize ETS, there is no need to know the indices of all the elements.

There is another function which will be required at each node of the N-computer, called the child activity to parent activity mapping function, $R$. This function results in all the multiple elements indices of parent’s elements which will consume the result of the child activity’s element. $R$ can be stated, in general, as:

$$R(ME\_index_{child}, \text{Act}_{child}, \text{Act}_{parent}) \implies ME\_indices_{parent}$$

Since it is assumed that the activities in the activity tree obey the same principles for mapping elements from child to parent activity as in F-code, the function $R$ can be considered as an interpreter of F-code’s geometric functions and rank/length coercion rules. Given the functions as defined above, and the activity tree, the initialization at a macro level is described as follows:
ETS Initialization

1. For an activity \( \text{ACTIVITY}_j \), get the number of elements at \( P_i \) (using function \( G \)).

2. For each element at \( P_i \)
   
   (a) Find out the current empty location in the ETS, \( \text{ETS.EMPTY} \) for the current element of the activity.
   
   (b) Initialize the \( \text{Act} \) and \( \text{GSCD}_{\text{field1}} \) of the \( \text{ETS.EMPTY} \) with the information in the \( \text{ACTIVITY}_j \).
   
   (c) Get the address of the output token storage which will be used for the output tokens of the current element of the activity. Initialize the OT field of the ETS with this address.

OTS Initialization

- For an activity, \( \text{ACTIVITY}_j \), first get the indices of all the elements at \( P_i \) (using function \( F^{-1} \)).

- For each element determine the elements of the parent activity for \( \text{ACTIVITY}_j \) (using function \( R \)).

- Initialize the \( n \) and \( \text{GSCD}_{\text{field2}} \) of the \( \text{OTS.EMPTY} \).

- For each element of the parent activity:
   
   1. Find the processor where the element will be executed (by using function \( F \)).
   
   2. Find out the ETS location which will be used for the synchronization and scheduling of the parent's element at the processor found in the previous step.
   
   3. Store that location and the processor number in the output token storage, \( \text{OTS.EMPTY} \).

After initialization, the processing phase will start. If an operation cycle for the execution of the activities in the N-computer is examined, it is as in Figure 5.2. Each processor executes activities in a set of activities with a dataflow synchronization for each element of an activity. At some point, as described in section 5.2, all the processors need to be synchronized globally.

Before the synchronization of all the processors, some processor may finish the execution of all the earlier activities and then sit idle, waiting for global synchronization signals from the other processors. In Figure 5.2, this delay is averaged and shown as \( D_1 \). \( D_1 \) depends upon the distribution function \( (F) \) which distributes all the activities to each processor of the N-computer. The more balanced the distribution, the less \( D_1 \) will be. The time between the
Figure 5.2: Processing in the N-Computer. Every node processes activities with a dataflow local synchronization. All the nodes will then be synchronized and again the dataflow execution at each node will start.

last processor finishing its job (activities between two global synchronizations) and starting the initialization of the ETS is called D2 in Figure 5.2 (end-up-delay). In the light of the implementation of the global synchronization control, which will be implemented in hardware with a physical link between all the processing nodes, D2 will be very short.

After D2, every processing node of the N-computer will start the initializer for the initialization of the ETS and OTS. During this time, known as D3, no processor will be processing activities in the activity tree. After initialization, again the dataflow execution of the activities starts. D4 shows an average delay for the starting tokens to reach the ETS and generate the synchronized tokens for processing. Let us call it a start-up-delay. The start-up and end-up delays depend upon the distribution function ($F$).

In the light of the above discussion, in the implementation of the system as described in this chapter, a major aim was to minimize the initialization time in an N-computer operation cycle. To achieve this, the following guidelines were used.

1. Implement the initialization in hardware. For that there is a need to bound the space complexity of the process and keep the number of operations to its minimum possible.

2. When possible, make the information needed for the initialization at each node, local before the initialization phase (in processing phase).
Figure 5.3: A typical activity tree. Each child activity may have more than one parent. To transform indices from a child to a parent there could be a geometric activity. The geometric activity will map each data element of a child element to its parent non-geometric activity. The non-geometric parent will have information about the computation and the ETS to consume a data element of its child activity.

3. Minimize D1 with the selection of an optimum distribution function, in general.

4. Study the possibility of overlapping the initialization with the processing and the ways to increase the $T_{proc}$.

To see the possibility of hardware implementation of the initializer, ETS and OTS initialization will be discussed at micro level in the next sections.

5.4 Initialization Strategy

To study the initialization process, the activities in an activity tree are logically divided into two groups; geometric activities and non-geometric activities. The geometric activities are the functions which transform each multi-index of a child object into multi-indices of its parent’s object. The geometric activities are equivalent to the F-code’s geometric functions and the rank/length coercion principles. The non-geometric activities in an activity tree represent the computation to be done on data elements generated by its child activities. A non-geometric activity is a collection of threads in such a way that each thread is performing some computation represented by the same computation tag or a program counter (PC). In addition to that, a non-geometric activity will have information about the ETS location and other necessary parameters needed for the initialization. The logical structure of a typical activity tree is shown in Figure 5.3. A child activity in the activity tree could have more
than one non-geometric parent\(^3\), and for every non-geometric parent there may be a different geometric parent.

Activities in the activity tree can be evaluated in an eager or a lazy way if no restriction on their evaluation is imposed by the program. In eager evaluation all the arguments of an activity are evaluated before the evaluation of the activity itself (body of the function). This evaluation is also called a data flow evaluation. In the lazy evaluation, first the body of a function (activity) is evaluated and then the result will be passed to the unevaluated arguments. This evaluation is termed as demand driven evaluation. In general, the data parallel functions (which process a collection instead of a single data element) are strict except the function IF\(^4\). By nature, strict functions implies eager or data flow evaluation\(^5\). Sometimes, to save redundant processing, the evaluation of these functions are preferred to be done in a lazy way. This lazy evaluation is termed spatial lazy evaluation.

The ETS and OTS model of the N-computer fully supports the eager evaluation of an activity tree if the order for the evaluation of their arguments is not imposed by the program. It also supports the lazy evaluations of an activity tree with certain restrictions needed to generate the global synchronization. A compiler may generate a code for lazy evaluation of a program and equivalent activity tree for the initialization (to support IF function of F-code).

As far as the ETS and OTS initialization from an activity tree is concerned, in principle, it can be initialized lazily or eagerly for activities which do not impose any order on the evaluation of the child activities, e.g., the choice and some geometric functions which select part of an object instead of replicating the object\(^6\) may demand lazy ETS initialization. The lazy ETS initialization implies the initialization of ETS and OTS for those elements of an activity which are needed by its parent activity. The eager ETS initialization means initializing the ETS and OTS for all the data elements belong to an activity in the activity tree by assuming that every element will be consumed by its parent activity.

In the lazy ETS initialization, a parent (root) activity will propagate down, towards the leaves, directs. A direct is the result of the root function which is needed at the leaves of the tree for the initialization of each data element as shown in Figure 5.4. According to these directs, ETS and OTS will be initialized for an activity at each processor. Initialization of ETS in a demand driven way in the N-computer is a very expensive process, in general, since there is a need to evaluate the geometric operations in reverse order and this also demands

---

\(^3\)Dataflow optimization of an assignment and more than one reference to the held object in F-code program may create this sort of tree. This will be discussed in section 6.4.2 and 6.4.6.

\(^4\)Some other functions of F-code like COMMA and HOLD are also restrict the order of evaluation of their arguments, but these functions are not very similar to the non-strict function IF.

\(^5\)In the literature, the terms eager and strict evaluations are used with the same meanings.

\(^6\)lazy evaluation for replication is not desirable, since the replication at the root of the tree is much efficient then at the leaves of the tree.
communication between the processing nodes during the initialization phase. To eliminate the remote communication, the system has to perform a lot of redundant processing. During the initialization, a dataflow graph will be traversed from the leaves to the root of the activity tree for each data element and the information will be obtained from the root and then ETS can be initialized for the element, i.e., walk from the leaves to the root and get the directs for ETS and OTS initialization. For the lazy ETS initialization, every processing node must have a copy of the root object used as a direct, e.g., the logical object in case of choice function, otherwise it would not be possible to eliminate remote communication during the initialization.

In most cases, it will be possible to optimize the activity tree by propagating down the geometric and/or choice function before the initialization and then initialize the ETS and OTS eagerly. In this way, there may be a wastage of some resources for the spatial non-strict activities in the activity tree, but that can be recycled. The total processing time of an activity tree (including execution and eager initialization) could be significantly less than that for the lazy ETS initialization. With the provision of the global synchronization count down field (GSCD\text{field}) in ETS and OTS and the choice flag in OTS, it is possible to support the lazy evaluation of activities even with the eager ETS initialization. Therefore the lazy ETS and OTS initialization in the N-computer is not supported.

In the following sections the initialization of ETS, STE and OTS from an activity tree will be discussed.

5.5 ETS Initialization

For the execution of an activity tree on the N-computer, there is always a need to initialize ETS for each activity if the activity is not a leaf. The initialization of STE and OTS is not always necessary. If an F-code program has static nature (i.e., the shape of the objects is known at compile time) then the information for STE and OTS can be generated by the

Figure 5.4: Flow of directs for lazy initialization for function choice and slice from the root to the leaves of the tree.
compiler. Let us assume that the program is of static nature and see the complexity of the ETS initialization process.

In the ETS initialization process, the \textit{Act}, \textit{GSCDfield}, and \textit{OT} fields of ETS will be initialized for all the data elements of an activity. Sometimes, for monadic activities, the \textit{OP} field of ETS will be initialized during the ETS initialization and the presence field, \(p\), of ETS will be set. To initialize these fields for all the elements of the activity at a processing node, the following pieces of information are required.

1. \textit{act.comp}: It is a light weight descriptor of a computation to be executed.

2. \textit{act.type}: It contains information about the type of activity such as unary, binary or reduce.

3. \textit{act.ets.ETS.ptstart}: It contains a pointer to the first location of ETS which will be used for the synchronization of elements of current activity at each node.

4. \textit{act.ets.GSCDfield}: It contains information as to whether the synchronization of the elements of an activity will contribute towards the generation of global synchronization.

5. \textit{act.ot.token.OTS.ptstart}: It contains a pointer to the first location of OTS1 which will be used to store the information about the output tokens for current activity.

6. \textit{act.parents}: It contains the number of parent activities for the current activity.

7. \textit{act.dist.N}: This parameter represents the number of units which will be distributed along the processing nodes.

8. \textit{act.dist.L}: This parameter represents the number of elements in each unit.

9. \textit{act.dist.P}: This parameter represents the first processing node from where the distribution of units will start.

These piece of information and the storage required for them is discussed in Appendix B.1.

It has not been finalized whether the initialization process and the ETS are going to be implemented on the same chip. The lower level details about the implementation of the ETS are yet needed to be studied. If the initialization process and ETS are implemented on different chips, then sending a packet to initialize all the ETS locations for all the data elements of the current activity would be desirable, because, the communication between two chips is, in general, slower than the communication within a chip. The initialization information for all the elements of an activity are going to be the same except the \textit{OT} field. The division of OTS into two sections, OTS1 and OTS2, has made it possible to construct
the information for the $OT$ field of the ETS for each element from the pointer to the start of the OTS1 ($act.ot_token.OTS.pt_{start}$), therefore a packet can be sent from the initializer to the ETS for an activity and the ETS system will form the required information for each location to be initialized.

If it is feasible to implement a memory whose $n$ locations ($n$ is more than one) can be written with the same word in one cycle. Then the scheme sending a packet for $n$ ETS locations even the initializer and the ETS is implemented on the same chip, would be closer to its implementation. The logic for the ETS initialization at processor $P_i$ will be as follows:

- Get number of data elements at $P_i$ by using function $G$ if the activity is not reduce with local/central reduction. If the activity is reduce with local/central reduction, then the number will be equivalent to the total number of data elements of the activity.

- Construct a packet from $act.ets.ETS_{pt_{start}}$, $act.comp$, $act.parents$, $act.ets.GSCD_{field}$, $act.ot_token.OTS_{pt_{start}}$ and the number of elements obtained in the first step. If the activity is monadic ($act.type$), then the packet will contain information about the setting of the presence field and the operand for the $OP$ if there is any.

- Send the packet to the ETS for the initialization of all the locations for current activity.

The detailed process for initializing ETS is described in pseudo OCCAM in Appendix B.2. Suppose there are $N_{processor}$ processing nodes in the $N$-computer. The logic for the evaluation of the function $G$ is as follows:

1. Divide $act.dist.N$ by $N_{processor}$. Suppose, $R$ is the remainder and $Q$ is the quotient of this division.

2. If $R$ is zero, then $Q$ will be the number of data elements at $P_i$.

3. If $R$ is more than zero, then;
   - if $P_i$ is within $R$ distance from $act.dist.P$, then the number of data elements will be $Q+1$;
   - otherwise the number of elements will be $Q$.

The description of the process to implement function $G$ is given in Appendix B.2.1. If the starting processing node for the distribution is the first processing node (node 1) then there are only four operations which will be evaluated in function $G$, i.e., one division which yields quotient and remainder, two conditions and one addition. If the starting processing node is not the first processing node then the evaluation of function $G$ will need few more operations.
Once the function G is evaluated, to initialize all the ETS locations for an activity, the system just need to increment ETS.pt\textsubscript{start} and evaluate the OT field from OTS.pt\textsubscript{start} for each ETS location\(^7\).

It is conceivable from the above discussion that ETS initialization for an activity tree is very simple, and therefore, can be easily implemented on a chip.

### 5.6 STE Initialization

The leaves of an activity tree do not need any ETS for the synchronization of computation which will generate these leaves. The synchronization to start the propagation of data on the leaves up to the tree will be satisfied by global synchronization in the N-computer. Therefore, the initializer for such leaves will not initialize ETS or send a token to the ETS for initialization. It will rather send the tokens to the STE. In this case the output tokens will be formulated with the data available at the leaves and stored in the STE. At the start of the next processing cycle, the STE will emit all the tokens. The emitted tokens from the STE are of the ETS or proc (for processing system) type. The flow of tokens related to the STE is shown in Figure 5.5. The logic for the initialization of STE for leaves activities will be as follows:

- Generate an output token.
- Send the token to the STE.

The process of implementing the above logic in pseudo OCCAM is described in Appendix D.3.

### 5.7 OTS Initialization

As mentioned earlier, the OTS initialization is not required for each activity in the activity tree during the initialization phase. The process described here can be implemented at compile time to initialize the output token storage for the activity. If the program has dynamic shape then there may be a need to implement the OTS initialization during the initialization phase. The general process to initialize the OTS as described in section 5.3 and shown in Figure 5.6 will be discussed first. As is clear from Figure 5.6 that there are 7 stages and three loops in the OTS initialization. The functioning of stage 2 and stage 3 is clear from

\(^7\)Here comes an important question that, if there are more than one data element of an activity are going to be processed at one processing node, then just store the OT for the first data element at one place. In the processing phase, to get OT for the ith data element just add the weighted i, if there is any. This issue is addressed in section 4.7.
Figure 5.5: Flow of tokens from the initializer to STE in the initialization cycle and from the STE to the ETS and the processing system in the processing cycle.

their names. The function of stages 1, 3, 4, 5 and 6 will be discussed in the following sections. It will be observed that the OTS initialization process is more complex when compared to the ETS and STE initialization. However, there are many situations when the OTS initialization can be optimized and the three major loops, i.e., loops on N1, N2 and N3 in Figure 5.6 can be eliminated. Those situations will be described in section 5.11. To support the logic as described in the OTS initialization process, the processes for the following main functions are required.

1. To distribute data a distribution function \( F \) and its inverse \( F^{-1} \), (stage 1 and 5).
2. To evaluate geometric activities to get parent multi-indices from a child multi-index, \( \text{INTP}_GEM \) (stage 4).
3. To evaluate the ETS location for a parent multi-index, \( \text{GET}_\text{DISP}_{\text{ETS}_{\text{parent}}} \), (stage 6).
4. To evaluate the OTS location to store output tokens, \( \text{GET}_\text{DISP}_{\text{OTS}} \), (stage 3).

The processes for implementing the above main functions are described in the following sections.

5.8 The Distribution Function, \( F \)

An important criteria for selecting a distribution function is to achieve the optimum execution time for a program on a parallel machine. This means balancing communication and processing time. A program may have a different pattern of communication. It is not necessary that all the programs will have same communication to computation ratio. Thus, to achieve an optimum execution of different programs on the N-computer, there is a
need of dynamic distribution function, i.e., which can distribute the computation depending upon the communication pattern of a program in such a way that the execution time of the program is very close to the optimum.

In the context of the N-computer, the following is further criteria for the selection of the distribution function.

1. The logic to know a processing node where a given data element of an object is mapped, should be very simple, i.e., it can be implemented in hardware efficiently.

2. The implementation of the inverse of the distribution function should also be very simple.

3. To know the number of data elements of an object at a processing node, ideally, the process should be very simple and the complexity should not be a function of the number of data elements. This is an important feature, since the number of data elements are required at the ETS initialization time and ETS initialization is always performed at run-time (during the initialization phase).

To fulfill the above criteria, a regular distribution strategy is chosen which can distribute the units of an object in a cyclic way, a similar approach as provided in HPF [46]. A unit could be a single element or a block containing all the elements along one or more dimensions of a multi-dimensional object. The cyclic distribution can start from any processing node in the network.

It is assumed, in general, that the computation is distributed according to the shape of the activity, i.e., according to the distribution of its resulting object. The distribution
function will be able to dynamically select the starting processing node and the size of the unit from the information given in the activity. According to the nature of the program this information can be provided by the compiler or by the run-time control system in the activity tree. The starting processing node parameter has already been given in section 5.5, which is act.dist.P. To inform the distribution function about the formation of a unit, the process need a binary mask along each activity. For every true value of the mask, the corresponding dimension will be distributed along the network starting from the given processing node (act.dist.P). Therefore, the following parameter is added to the activity.

- act.dist.m: A binary mask for distribution.

For a multi-index (K) of object with shape E of rank r, the distribution function with the mask act.dist.m can be written by using the same notations as used to define F-code functions, as follows.

\[
\text{INDEX} = P^m(K)_0 + \left( \sum_{i=1}^{n} \left( P^m(K)_i \times \prod_{j=0}^{i-1} P^m(E)_j \right) \right) ; \quad n \leq r
\]

Where, \( P^m(K)_i \) represents the \( i \)th element of the projection\(^8\) of vector \( K \) on mask \( m \). From the result of above function, the processing node for the multi-index will be evaluated, by adding the INDEX to the first processing node for mapping modulo the number of the last processing node in the network, as follows:

\[
P_{\text{current}} = (1 + \text{act.dist.P} + \text{INDEX})_{\text{mod NProcessor}}
\]

Since, the multi-index is numbered from zero and the processing nodes are numbered\(^9\) from 1, addition of 1 in the above expression is to bring the index into the same domain as the processing nodes are. To implement the above function the logic is described in pseudo Occam in Appendix C.1.12.

5.9 The Inverse of Distribution Function, F_INVERSE

This function generates all the multi-indices of an object one by one and the number of each multi-index out of all the multi-indices which are mapped at a processing node with the distribution function as described in the previous section.

To write the inverse distribution function in a formal way, let us define a shadow of a vector on a mask, which is as follows:

\(^8\)This is the same projection as defined in F-code.

\(^9\)The numbering of processing nodes from 1 instead of 0 eliminates one addition in the evaluation of function G, which is evaluated more often than function F.
so that

\[ b_k = \begin{cases} 
0 & \text{if } m_k = 0 \\
a_{X(k,m)} & \text{otherwise}
\end{cases} \]

where \( X(k, m) \) is the number of mask bits \( m_j = 1 \) with \( j < k \). In a simple form, the shadow can be written as follows:

\[ S^m(k) \rightarrow b \]

Let us define the function count on a vector, \( V \), of length \( r \), which will count the vector for \( n \) times. The function is as follows:

\[ C^n(V)_{(m_0, i_0), (m_1, i_1)} \rightarrow \begin{cases} 
null & \text{If } V \geq m_0 \text{ or } m_1 \text{ for } j = 0 \\
V'_{j} & \text{otherwise; } \forall \ j \leq 0 < n
\end{cases} \]

The count function, \( C \), counts each element of the vector \( V \) with the respective modulo and the increment as given in the modulo vector \( mV \) and the increment integer \( i \), for \( n \) time and output the result \( V' \) after each increment. The counting will proceed if each component of the vector \( V \) is less than its respective element of the modulo vector \( (mV) \). If this condition is not true, then the counter function will not proceed and return a null result. The elements of the modulo vector \( mV_0 \), for which the vector \( V \) will be counted with the other modulo \( (mV_1) \), will be zero and the vice versa. The second modulo component is optional. If there is only one modulo component for the counter function, then the vector will be counted for the non-zero element of the modulo vector with the given increment. The count function with one modulo component is as follows:

\[ C^n(V)_{(m_0, i_0)} \rightarrow \begin{cases} 
null & \text{If } V \geq m_0 \text{ for } j = 0 \\
V'_{j} & \text{otherwise; } \forall \ j \leq 0 < n
\end{cases} \]

Assume, there are \( n \) elements at the processing node \( (P_{\text{current}}) \) where the inverse distribution function for a multi-dimensional object of shape \( K \) will be evaluated. The inverse distribution function will form the multi-index of the first data element which will be the distance between the first processing node for the distribution and the current processing node (modulo the number of processing node). Assume, the first multi-index is \( V \). Then the inverse distribution function will be as follows:

\[ C^n(V)_{(S^m(K), N_{\text{processor}}), (S^m(K), 1)} \]

The algorithm to implement the above function is described in Appendix C.1.13.
5.10 The Function INTP\_GEM

This function is supposed to generate parent's indices from a given child index for F-code geometric functions and rank/length coercion principles to initialize OTS. In the next sections, the implementation of the rank/length coercion will first be considered and then the evaluation of the geometric operations will be described. The structure of each logical activity will be defined accordingly. The function INTP\_GEM will evaluate any one of the following processes depending upon the information available in the geometric activity.

5.10.1 Rank/Length Coercion

The implementation of the rank and length coercion can be considered as the counter function as defined in section 5.9. To evaluate rank/length coercion, the following three piece of information along with the child multi-index are required.

1. \texttt{gem\_act.rl.E}: This is the shape of the parent activity.
2. \texttt{gem\_act.rl.m}: A binary mask for rank coercion.
3. \texttt{gem\_act.rl.n}: An integer which represents the number of parent data elements which will consume a child data element. This parameter can be termed a rank coercion factor.

Assume the vector V is the child multi-index. The rank/length coercion function will be as follows:

\[
C^n(V) |_{(S(E), 1)}
\]

The algorithm for the implementation of this function is described in pseudo Occam in Appendix C.1.1.

5.10.2 The Function Sect

This function selects the $L$th layer along the $d$th dimension of a multi-dimensional object. For a given child index, two parameters ($L$, $d$) are required to know whether that element is within the layer which is going to be selected. If it is not within the layer, then the function \texttt{INTP\_GEM} returns a null token. Otherwise, it will return the child index without the component along the $d$th dimension. To evaluate this function the sect activity will have the following piece of information.

1. \texttt{gem\_act.sect.L}: The number of the layer to be selected by the function sect.
2. \textit{gem\_act\_sect\_d}: The number of the dimension along which the layer will be selected.

3. \textit{gem\_act\_sect\_r}: The rank of the object from which the layer will be selected.

If $K$ is the child multi-index, then the evaluation of \texttt{sect} function can be written as follows:

$$PICK1(K, d, L) \rightarrow \begin{cases} V_j := K_i & \forall 0 \leq i < r \\ \forall 0 \leq j < r - 1 \\ \text{if } K[i] = L \text{ for } i = d \\ \text{null otherwise} \end{cases}$$

The function \texttt{pick1} selects all the components from a vector ($K$) other than the $d$th component, if the $d$th component is equal to the value given in $L$ and output the resulting vector ($V$). Otherwise, it will return the \texttt{null} multi-index. The process for implementing this function is described in Appendix C.1.2.

5.10.3 The Function Slice

This function selects more than one layer along the $d$th dimension of a multi-dimensional object. The evaluation of this function can be done in three different ways which will be discussed soon. To tell the initializer a particular type of evaluation, the following parameter is added with the geometric activity.

\textit{gem\_act\_slice\_type}: Two bits field to decode one out of three types for the evaluation of \texttt{slice}.

In the first type of evaluation, for a given child index, three parameters are required to know whether that element is within the layers which are going to be selected. These parameters are as follows:
1. `gem_act.slice.type1.V`: A vector of integers whose elements are equal to the number of layers to be selected in the slice. Assume the length of the $V$ is $V_{length}$.

2. `gem_act.slice.type1.d`: The number of dimension for the selection of slice.

3. `gem_act.slice.type1.V_length`: An integer scalar, represents the length of the vector $V$.

If the child multi-index is not within the layer, then the function $INTP.GEM$ returns a null token. Otherwise, it will return the child index. The evaluation of this function for a child multi-index $K$ of length $r$, is written as follows:

\[
PICK2(K, d, V) \rightarrow \begin{cases} 
    V'_i = K_i & \text{if } i \neq d \\
    0 \leq i < r \\
    V'_i = j & \text{if } i = d \text{ and } K_i = V_j; \text{ for any } j \text{ when } 0 \leq j < V_{length} \\
    \text{null} & \text{otherwise}
\end{cases}
\]

The parameter required to select the layers is an array whose size is limited by the extent along a dimension. The process for implementing this function is described in Appendix C.1.3. The logic for this process is very similar to that for the $sect$ function except instead of checking for one component along the dimension of sect, it checks more than one component, e.g., if a matrix is sliced along the columns, then for a given child index the process will perform the following logic for every column;

1. if the second index of the child multi-index is less than the current column number out of all the columns to be selected, then the current child index is not a part of the slice. Therefore, output a null index and exit the process. Otherwise do as in the following items;

2. if the second index of the child is equal to the current column number, then convert that element into parent's index element;

3. if the second index of a child is more than the current column number then check for the next column number (go to item 1).

To implement this, a vector ($V$), which has the index for each column to be selected in the slice of the matrix, is required. The length of the vector can go up-to the total number of columns minus one. In general, the length along a dimension could be a large number, say 500. To select, say 300 layers in the slice, a vector of 300 values for the implementation of the logic as described above, is required. There is another drawback of this implementation, if a slice function is going to slice, say, 10 columns, then to know that whether a child index belong to the 10th column or not the process has to check the index for the first nine columns.
In some situations, it is possible to implement the slice without having a vector of columns as in the above example (or a vector of layer numbers, in general) and without the need to check for each column to select the last column. Suppose, in a program a slice is made from the number of columns which make a regular pattern, e.g., the slice made from every second column, every third or so on. In this case, only one scalar instead of a vector is required. The situation is shown in Figure 5.7. To know whether a child index is going to be selected in the slice for the example of Figure 5.7, the process just need a division for the second element of the multi-index of the child element. If the remainder is zero, then the element is going to be selected in the slice. A program may need a slice with some columns which are not divisible by 2 but still make a regular pattern. A slice could be selected with a regular pattern within a certain domain starting from a point and ending at a point along the dimension of the slice. Therefore, by having three parameters, start, increment and end the slice function can be optimized for a wide variety of regular patterns. The parameters are defined as follows:

1. \textit{gem\_act\_slice\_type2\_S}: The starting point from where the selection of layers for a slice will start, (S).

2. \textit{gem\_act\_slice\_type2\_I}: The distance between any two contiguous layers to be selected in the slice, (I).

3. \textit{gem\_act\_slice\_type2\_L}: The last layer to be selected for the slice (L).

4. \textit{gem\_act\_slice\_type2\_d}: The number of dimension for the selection of slice.

The optimized implementation of the function to support slice operation of F-code as given below, is described in Appendix C.1.4.

\begin{equation}
\text{PICK3}(K, d, S, I, L) \rightarrow \begin{cases} 
V'_i = K_i & \text{if } i \neq d \\
V'_i = Q & \text{if } i = d \text{ and } S \leq K_i \leq L \text{ and if } R = 0 \\
null & \text{otherwise}
\end{cases} \quad 0 \leq i < r
\end{equation}

In the cases when the pattern for the selection of layers for the slice from an object is not regular, the implementation of the process slice can be simplified by converting the vector which contains the components to be selected along a given dimension, into a binary mask before the initialization as shown in Figure 5.8. In the implementation of the slice function, the process will map the given child index for the dimension of slice along the mask and accordingly select or reject the child index for the parent index. If the child index is selected
Figure 5.8: A Slice from a matrix along columns with an irregular pattern for the selection of columns with a mask.

for the parent index, then it can be converted into the parent index. The evaluation of the third type of slice requires the following parameters.

1. \texttt{gem.act.slice.type3.M}: A binary mask whose length is equal to the extent of the object along the dimension of the slice.

2. \texttt{gem.act.slice.type3.d}: An integer scalar, represents the dimension of the slice.

The function to evaluate the slice with mask is given below and the process for implementing this function is given in Appendix C.1.5.

\[
V'_i = \begin{cases} 
K_i & \text{if } i \neq d \\
K_i & \text{if } i = d \text{ and } M_{K_i} = \text{TRUE} \\
\text{null} & \text{otherwise}
\end{cases} 
\]

\[0 \leq i < r\]

\[\text{PICK4}(K, d, M) \rightarrow \]

The run-time control system can optimize the slice function before the start of the initialization in one of the following ways:

1. Regular pattern encoded in three parameters.

2. With a binary mask for irregular pattern when the number of layers in a slice is significantly high.

3. With a vector, if the number of layers to be selected are significantly less than the extent of the object along the dimension of the slice. If the system uses 6 bits integer for the extent along each dimension, i.e., extent could be 64. Then, to have a binary
mask for the dimension of extent 64, we need 64 bits and if a program is going to select less than 11 layers from 64 for a slice, to have the vector of less than 11 values of 6 bits each instead of the mask will save the amount of space needed.

To support an irregular pattern, the amount of space either for the mask or for the vector can be fixed, say, up-to six elements in the vector. The same space in the geometric activity can be used as a mask for the slice. If a slice function in a program does not fit in the space provided in the fast memory, then the slower memory can be used.

5.10.4 The Function Replicate

This function replicates a multi-dimensional object along one or more than one dimension according to a replication factor for each dimension. In other words, a child multi-index is going to be counted to generate \( N \) parent multi-indices for \( N \) times. There may be a different increment for counting each dimension modulo the size of the dimension. Assume, \( V \) is the first parent multi-index for a child multi-index which is going to be replicated \( n \) times. \( E \) is the shape of the parent object and \( D \) is the vector of increment. The length of \( V \), \( E \) and \( D \) is the same. Then the evaluation of replication with a count function is as follows:

\[
C_{\text{general}}^{n}(V)(E, D) \rightarrow V_i; \quad 0 \leq i < n
\]

This counter function is a more general form of the counter function defined earlier in section 5.9. The general counter function counts the vector \( V \) for \( n \) times. Each element of the vector is counted in modulo the respective value of vector \( E \) with the respective increment given in vector \( D \). If the vector \( V \) is not replicated for some elements, then the corresponding element of the vector \( D \) will have 0 value. To implement the above function, the following pieces of information are added with the repl activity.

1. gem.act.repl.D: A displacement vector, each element of this vector represents the replication displacement (or increment) along that dimension for the counter function.
2. gem.act.repl.E: A modulo vector. This will be the shape of the resulting object.
3. gem.act.repl.m: A binary mask which will be used to make the first parent multi-index from the child multi-index. The true bits in the mask represent the dimensions which will overlap between the child and parent object.
4. gem.act.repl.n: An integer value represents the number of replications for a child multi-index.
5. gem.act.repl.r: An integer value represents the rank of the resulting object.

The process for the replication is described in pseudo Occam in Appendix C.1.6.
Figure 5.9: Structure of the activity tree for the comp function when the objects are going to be composed along the last dimension. The indices of the left child will be the same for the parent object except those for which the length coercion is involved.

5.10.5 The Function Comp

This function composes two multi-dimensional objects with the same rank, along certain dimensions to form a single object. To generate a parent multi-index from a child multi-index, the child multi-index will be displaced with some displacement parameter for the dimension of composition. The displacement parameter for a child activity can be formed from the shape of the parent activity and the binary mask. Assume, there is a vector \( V \) (child multi-index), displacement vector \( D \) and the parent shape \( E \) of length \( r \) then the transform function (\( T \)) to evaluate comp function is as follows:

\[
T(V)_{(D,E)} \rightarrow \begin{cases} 
V_i + D_i & \text{if } V_i < E_i; \quad \forall \ 0 \leq i < r \\
\text{null} & \text{otherwise}
\end{cases}
\]

The process for implementing the \( T \) function is described in Appendix C.1.7 and the following parameters other than the child multi-index for this implementation are required.

1. \( \text{gem}\_\text{act}\_\text{comp}\_E \): A vector to represent the shape of the parent activity.

2. \( \text{gem}\_\text{act}\_\text{comp}\_D \): A vector to represent the displacement along each dimension.

If the two objects are going to be composed along the last dimension, then the multi-index of the left child will not be displaced to form the parent multi-index. That situation can be detected by the compiler (static property) and accordingly the activity tree can be formed. If the shape is dynamic and it is not possible to propagate the length coercion down
to the leaves of the activity tree, then there will be a need to perform the length coercion. The activity tree for this situation is shown in Figure 5.9. The length coercion is evaluated as follows:

\[ L(V)_E \rightarrow \begin{cases} V_i & \text{if } V_i < E_i; \forall 0 \leq i < r \\ null & \text{otherwise} \end{cases} \]

The process for implementing the \( L \) function is described in Appendix C.1.8. The runtime control system can detect whether there is a length coercion involved in the optimization of the function \( \text{comp} \). If there is no length coercion involved, then in the optimization of the function \( \text{comp} \) when the two objects are going to be composed along the last dimension, the first object does not need any geometric activity.

5.10.6 The Function \( \text{Transp} \)

This function transposes its object, i.e., it moves a given dimension of its argument to be the last dimension of its resulting object. The logic for its implementation is very simple. Assume \( V \) is the child multi-index of object of rank \( r \) and \( n \) is the number of the dimension going to be at the end. Then the evaluation of \( \text{transp} \) function, \( T_2 \) to generate a vector of parent multi-index \( V' \) is as follows:

\[ T_2(V)_n \rightarrow \begin{cases} V'_i = V_i & 0 \leq i < n \\ V'_i = V_{i+1} & n < i < r - 1 \\ V'_i = V_n & i = r - 1 \end{cases} \]

The process for implementing the function \( T_2 \) is described in Appendix C.1.9. The evaluation of this function requires two parameters which are as follows:

1. \texttt{gem.act.transp.n}: An integer scalar value to represent the dimension to be transposed.
2. \texttt{gem.act.transp.r}: An integer value represents the rank of the object to be transposed.
5.10.7 The Function Diag

This function selects the main diagonal of a multi-dimensional object or a diagonal of a layer for all other layers. A simple logic to implement this function is to compare the elements of the child index for the dimensions of the diagonal and if they are equal, then transform the child index to the parent index. Assume a vector \( V \) of child multi-index of length \( r_c \) and a mask \( m \) of the same length. The true elements of the mask represent the dimensions for the selection of the diagonal. The length of the parent multi-index is \( r_p \). The function, which checks whether the child multi-index belongs to the diagonal and if it belongs to the diagonal then generate the parent multi-index, is as follows:

\[
D(V)(m, r_c) \rightarrow \begin{cases} 
V'_j = V_i & \text{if } m_i = \text{FALSE}; \\
V'_i = V_i & \text{if } m_i = \text{TRUE} \text{ and } a \leq 0 \text{ OR } a = V_i; \\
null & \text{if } m_i = \text{TRUE} \text{ and } a > 0 \text{ and } a \neq V_i; \\
\forall 0 \leq i < r_c - 1; 0 \leq j < r_p - 1 \\
V'_j = a & \text{if } j = r_p - 1 \text{ and } i = r_c - 1
\end{cases}
\]

Where \( a \) is an integer with initial value 0. The process to implement the function \( D \) is described in pseudo Occam in Appendix C.1.10.

The evaluation of this function is very simple and requires following parameters other than the child multi-index.

- \( \text{gem.act.diag.m} \): A binary mask to decode the layers for the diagonal.
- \( \text{gem.act.diag.rc} \): A integer represents the rank of the object.

5.10.8 The Function Pack

This function repacks the object into a different shape array first, by flattening the multi-dimensional array into a one-dimensional array and then converting it into a multi-dimensional array. Both objects must have the same size when flatten. To implement the process, first form the single element index with a weighted sum for each element of the multi-index and then count the single index modulo the shape of the parent object. The process is described in Appendix C.1.11. The evaluation of this function needs two parameters other than the child multi-index, which are as follows:

1. \( \text{gem.act.pack.Eparent} \): Shape of the parent object.
2. \( \text{gem.act.pack.Echild} \): Shape of the child object.
<table>
<thead>
<tr>
<th>decomp_type</th>
<th>com_type</th>
<th>diff</th>
<th>scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMPLE</td>
<td>Point-to-point</td>
<td>CONSTANT</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON-CONSTANT</td>
<td>2</td>
</tr>
<tr>
<td>SIMPLE</td>
<td>BROADCAST</td>
<td>CONSTANT</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON-CONSTANT</td>
<td>4</td>
</tr>
<tr>
<td>COMPOUND</td>
<td>Point-to-point</td>
<td>CONSTANT</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON-CONSTANT</td>
<td>6</td>
</tr>
<tr>
<td>COMPOUND</td>
<td>BROADCAST</td>
<td>CONSTANT</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NON-CONSTANT</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 5.11: Possible template combinations for output tokens.

5.10.9 The Functions GET DISP OTS and GET DISP ETS_{parent}

The function GET DISP OTS, stage 3 in Figure 5.6, is straightforward. If an activity has \( m \) data elements at a processing node, then for the \( i \)th data element the OTS location will be the addition of weighted \( i \) to the OTS ptstart. The weight will be according to the number of the current parent out of \( N_2 \) parents. The process to support this function is described in detail in Appendix C.1.14.

The function to know the ETS location for the parent multi-index, which is not necessarily going to be at the same processing node, is not so straightforward. The logic for the evaluation, in general, is shown in Figure 5.10. The process will evaluate the displacement of the current multi-index from the first multi-index at \( P_i \) and then add that displacement to the ETS ptstart of the parent activity.

5.11 OTS Initialization; The General Process

In general, the OTS initialization is a complex process. Consider, the macro initializer for the OTS as shown in Figure 5.6. The initializer will have a three loops complexity. The process to implement this macro level logic is described in Appendix D.2.1. The first loop
is according to the number of data elements of an activity at processor $P_i$. The second loop within the first loop has its extent according to the number of parents. Assume, there are $N_2$ number of parents. The third and inner most loop, may have a different extent for each instance of the second loop. The extent of the third loop is according to the number of data elements of a parent activity which will consume the result of the current data element of the child activity.

If $N_1$ is the number of data elements of the child activity at $P_i$, $N_2$ is the number of parents and $N_3_i$ represents the number of data elements of the $i$th parents which will consume the result of current child data element. And, $OP_{N_1}$ is the number of operations in outer loop only\textsuperscript{10}, i.e., to evaluate the inverse distribution function, $OP_{N_2}$ is the number of operations in the second loop only and $OP_{N_3}$ is the number of operations for the third loop. Most of the operations of the process are in the third loop. It is possible that the number of operations in the third loop for every parent is not the same. One parent could be a choice while the other parent might not. For a general case, the overall time complexity (number of operations) of the initializer can be written as follows:

$$T_{1_{\text{general}}} = N_1 \times (OP_{N_1} + (N_2 \times OP_{N_2}) + \sum_{i=1}^{N_2} (N_{3_i} \times OP_{N_{3_i}}))$$ (5.1)

\textsuperscript{10}Without taking into account the number of operations in the sub-loops of the outer loop.
It is clear from the above complexity function that to reduce the time for the OTS initialization there is a need to reduce the number of operations in each loop. If the number of operations in each loop is unity, then the above complexity will approach the minimum complexity of the process, which is as follows:

$$T_{i_{\text{min}}} = N_1 \sum_{i=1}^{N_2} N_3$$  \hspace{1cm} (5.2)

Let $M$ be the total data elements of all the parents activities which will consume a child data element. Then the equation 5.2 means to generate output tokens for a data element which will be destined to $M$ data elements, the OTS initializer will perform $O(M)$ operations. If the index of the third loop is the same for each instance of the second loop, which is $N_3$, then the equation 5.2 will become as follows:

$$T_2 = N_1 \times N_2 \times N_3$$  \hspace{1cm} (5.3)

Equation 5.2 and 5.3 represent an ideal situation for the implementation of the above process. In the same framework, it is not possible to reduce the complexity further. To reduce the complexity even further, we may have to adopt a different framework. Since a data parallel program has regular pattern, it is possible that the output tokens for a data element may have regular patterns. If the system detects those regular pattern, it is possible to have an OTS initializer to initialize the output tokens for $M$ data elements with a complexity even less than $O(M)$. The regular patterns and the initialization of OTS for those patterns are discussed in the next sections.

### 5.11.1 Optimizations for OTS Initialization

Depending upon the regular patterns which the ETS locations for an activity may have, the OTS initialization can be optimized. The following three types of activity can be marked by detecting the regular patterns.

1. **Communication type**: An activity may generate an output token which is going to be propagated to all the processing nodes. The communication type decodes this information. The communication type of activity for the output token can be a *broadcast* or *point to point*.

2. **Decomposition type**: An activity may generate more than one output token with the same data, which will be propagated to the same processing node. In some situations, these tokens can be integrated into one *compound* token. Then the compound token can be propagated to the destination processing node. Here, it can be broken down
into more than one simple token. Therefore, the decomposition type of activity for output tokens can be compound or simple.

3. **Difference Type**: An activity may have constant or non-constant difference type output tokens. For a constant difference activity, output tokens are going to be consumed by the ETS location which is a constant distance away from the ETS location which will generate the output token.

By combining these three parameters, where, each one has two values, there could be eight different ways to initialize the OTS. If these three types are called template for output tokens, then the possible template combinations are shown in Figure 5.11. To tell the initializer about the optimization, the following parameters are added in the activity.

1. **act.ot.token.type**: An activity is of constant or non-constant difference type.

2. **act.ot.token.com.type**: An activity is of point to point or broadcast type.

3. **act.ot.token.decomp.type**: An activity is of simple or compound type.

4. **act.ot.token.I**: In the case of a compound type, this is the increment parameter.
5. act.ot_token.N: In the case of a compound type, this represents a number of simple tokens in a compound token.

Optimization for Constant Difference Activity (SPC)

The ETS for all the data elements of an activity at a processing node is going to have contiguous locations starting from the same location at each processing node. Assume, there is a diadic operation on the objects which are distributed to all the processing nodes evenly, i.e., if \( N_{\text{processor}} \) processing nodes and the objects have sizes, \( S_1 \) and \( S_2 \), then each processing node will have \( S_1/N_{\text{processor}} \) and \( S_2/N_{\text{processor}} \) elements of the respective arguments. The pattern of ETS for such a situation is shown in Figure 5.12. If the activities have such a pattern in their ETS, then these are called constant difference activities. If the activities do not have such a pattern in their ETS, then these are called non-constant difference activities. For the OTS initialization of the constant-difference activity, there will be no need to evaluate the process GET_DISP_ETS_{parent} as shown in Figure 5.6 and 5.10. The micro-level description of this process is given in Appendix D.2.2. The number of operation for this type of OTS initialization is as follows:

\[
T_{\text{constant}} = N_1 \times (O_{P_{N_1}} + (N_2 \times O_{P_{N_2}}) + \sum_{i=1}^{N_2} (N_{3_i} \times O_{P_{N_{3_i}}}))
\]  

Where the \( O_{P_{N_{3_i}}} \) is less than the \( O_{P_{N_{3_i}}} \) in equation 5.1.

Optimization for Broadcast Activity (SBN)

The other optimization can be done if the output tokens for an activity are going to be broadcasted to all the processing nodes, e.g., if there is a diadic operation on two matrices which results in a box. If the dimension of the box along which the matrices are going to be replicated is distributed evenly to all the processing nodes, then each processing node is going to receive the token for the diadic activity from its child activities. The child activities can broadcast each result, if it is going to be consumed by the same ETS location at each processing node. If the activity's parent ETS pattern is such that the result of the activity can be broadcasted to all the processing nodes, then the activity is of broadcast type. Otherwise, it will be of point to point type. For the OTS initialization of broadcast type activities, there is no need to evaluate the function \( F \) (stage 5 in Figure 5.6) to know the destination processing node for the output token. Also, there is no need to know all the parent's multi-indices for a given child multi-index, the system just need to know the first parent multi-index. Therefore, the evaluation of the function INTP_GEM can be stopped as soon as the first parent multi-index is available. The pattern of ETS for broadcast type activity is shown in Figure 5.13. If the complexity of the initialization process for broadcast
type tokens is compared with the complexity of the general process, the index for the third loop is 1 for broadcast type tokens. The complexity of equation 5.1 for this process will be as follows:

\[ T_{\text{broadcast}} = N_1 \times (O_{P_{N_1}} + (N_2 \times (O_{P_{N_2}} + O_{P_{N_3}}))) \] (5.5)

Suppose, the number of operations for each data element in the third loop for equation 5.1 and equation 5.5 has following relations:

\[ O_{P_{N_3}} \approx A \times O_{P'_{N_3}} \]

Where \( A \geq 1 \) is a constant. In this case the complexity of equation 5.5 with the parameters of equation 5.1 can be written as follows:

\[ T_{\text{broadcast}} = N_1 \times (O_{P_{N_1}} + (N_2 \times (O_{P_{N_2}} + O_{P_{N_3}}/A))) \] (5.6)

The complexity of the initialization for broadcast type activity is \( O(N_1 \times N_2) \), which shows that to generate all the output tokens for a data element which will be consumed by one parent activity, we just need one iteration of the process in the third loop of Figure 5.6 without stage 5. Therefore, detection of the broadcast pattern for the output tokens and the addition of communication type with the activity is a significant step in reducing the
Figure 5.15: ETS pattern for compound, point to point output tokens for a non-constant-difference activity. A resulting element of a child activity is going to be consumed by more than one ETS location with a regular pattern at the destination processing node.

initialization time and improving the performance of the N-computer. The description of the initializer for this type of tokens in pseudo Occam is described in Appendix D.2.3.

**Optimization for Compound Activity (CPN)**

The diadic operation on a vector and a matrix which results in a matrix as shown in Figure 5.14, is considered as an example, here. Assume, the resulting matrix is distributed to the processing nodes along its rows, i.e., each row will be at one processing node. To perform the diadic operation, each element of the vector is going to be replicated for the number of columns (c) of the matrix. For this situation, instead of generating c output tokens for each element of the vector, a compound token can be generated. The single compound token will travel from the producer processing node to the destination processing node. At the destination node, the compound token can be broken down into c simple tokens for the diadic activity. The activity for which it is possible to generate compound output tokens, is called a *compound* activity. All other activities are called *simple* activities. The process for the initialization of OTS for the compound activity is very simple compared to the initialization for the simple activity.

Consider the macro level initializer of Figure 5.6. The third loop will not be executed for all the parent multi-indices for a child multi-index. Instead, for general implementation
BTS pattern at ETS pattern at BTS pattern at ETS pattern at ^ ETS pattern et ^

Figure 5.16: ETS pattern for simple, broadcast output tokens for a constant-difference activity. The difference between the child ETS locations and corresponding parents ETS locations is the same for all the data elements. All the processors will have the same pattern with no more than one output token per child data element per processor.

of the OTS initialization, the third loop will be executed according to the number of the processing nodes in the N-computer. To understand the point, consider the replication of the vector into a box in the example of Figure 5.14 and only the first dimension of the box is distributed to the processing nodes. Then, there may be more than one processing nodes which will consume the compound tokens of each element of the vector. ETS pattern for such an activity is shown in Figure 5.15. If the number of processing node in the N-computer are $Q$, then the complexity of the initialization process for compound activity will be as follows:

$$T_{compound} = N_1 \cdot (OP_{N_1} + (N_2 \cdot OP_{N_2}) + \sum_{i=1}^{N_2} (Q \cdot OP''_{Q_i}))$$

The parameter $OP$ denotes the number of operations in each loop designated by its subscript. All the other parameters are interpreted in the same way as for the previous schemes but in the context of the above process. To compare the complexity of the above process with the complexity of equation 5.1, the number of operations in the first and the second loops are going to be the same. The improvement, if there is any, depends upon the savings in the number of operations in the third loop. For a compound token, there must be at least two parent’s data elements consuming a child data element at a processing node. Assume there are $C$ parent’s data elements for a child data element. This means that in
the third loop, the initialization of OTS2 is going to be for \( C/2 \) tokens to its maximum as opposed to the general initialization discussed in section 5.7, in which the initialization of OTS2 is going to be for \( C \) tokens. It means the complexity of the third loop is reduced at least by 50%. At least for half of the time, there will be no need to evaluate the function \( \text{GET\_DISP\_ETS} \) and to send the result to the OTS2. However, there is a need to know for every processing node of the N-computer whether a child data element will have a parent data element to be processed at the processing node. The micro level process for the OTS initialization of the compound activity is given in Appendix D.2.5.

### 5.11.2 Combination of the Three Template Parameters and OTS Initialization

By detecting different regular patterns in the ETS for activities in the activity tree, the OTS initialization can be optimized as mentioned in the previous section. From the OTS initialization point of view, an activity can be simple or compound, point to point or broadcast and constant or non-constant difference. These three types, which can effect the output token system initialization, each parameter can take two values, give a total of eight possible ways to initialize OTS. These are illustrated in Figure 5.11.
Figure 5.18: ETS pattern for compound, broadcast output tokens for a non-constant-
difference activity. A result of child activity is going to be consumed by more than one ETS
location at each processing node with a regular pattern. However, the distance between the
child ETS and the parent ETS is not the same for all the data elements.

**Simple, Broadcast and Constant Activity (SBC)**

An activity can be of broadcast and constant difference type. The OTS initialization for this
type of activity will enjoy the advantages of both broadcast and constant difference types as
mentioned above. For constant difference activity, there is no need to evaluate the function
GET_DISP_ETSparent as given in Figure 5.6 and 5.10. For a broadcast type activity, there is
no need to run the third loop for all the parent multi-indices. The ETS pattern for this type
of activity is shown in Figure 5.16. The process to initialize OTS for this type of activity
is given in Appendix D.2.4. If $OP_{N_3}^n$ are the number of operations for each instance of the
third loop (which will be one for this type of output tokens), and $B$ is a constant with the
following condition:

$$OP_{N_3} \approx B \cdot OP_{N_3}^n$$

Then the complexity of equation 5.6 will become as follows:

$$T_{SBC} = N_1 \cdot (OP_{N_1} + OP_{N_3}/B)$$  \hspace{1cm} (5.8)

Where, the constant $B$ is significantly more than the constant $A$ as in equation 5.6.
Figure 5.19: ETS pattern for compound, broadcast output tokens for a constant-difference activity. A result of child activity is going to be consumed by more than one ETS location at each processing node with a regular pattern. Also the distance between the child ETS and the parent ETS (d) is the same for all the data elements.

**Compound, Point to Point and Constant Activity (CPC)**

The activity will be of type *compound, point to point* and *constant* difference, if the ETS has pattern as shown in Figure 5.17. If a processor is going to consume the result of a data element of a child activity for more than one data element of a parent activity with regular pattern of the ETS such that the parent’s ETS for each data element can be evaluated from the corresponding child’s data element ETS, then the initialization can become more simpler when compared to the initialization for *compound, point to point* and *non-constant* difference activity, as mentioned in section 5.11.1. In this case, there will be no need to evaluate the function \textit{GET\_DISP\_ETS}. The process to initialize OTS for this type of activity is described in Appendix D.2.6.

**Compound, Broadcast and Non-constant Activity (CBN)**

An activity could be of type *compound, broadcast* and *non-constant difference*. The pattern of ETS for this type of token is shown in Figure 5.18. Since the token are of broadcast type as in section 5.11.1, there is no need to run the third loop for all the processing nodes of the N-computer. The process just needs to know the first parent’s ETS location for a child data element at a processing node, and then a single OTS initialization will be sufficient for the
$Q \times C$ output tokens ($Q$ and $C$ are defined earlier). The process to initialize OTS for this type of activity is discussed in Appendix D.2.7.

**Compound, Broadcast and Constant Activity (CBC)**

An activity can have the ETS pattern as shown in Figure 5.19. This type of activity is of *compound*, *broadcast* and *constant* difference. This is the most friendly pattern for the OTS initialization. For these types of tokens, there is no need to know the parent’s indices for a child index and then no need to run the $GET\_DISP\_ETS$ to know the ETS location for the parent indices. The third loop does not exist in the initialization process for this activity. The process for the OTS initialization is discussed at micro level (in pseudo OCCAM) in Appendix D.2.8.

### 5.12 Summing Up

In this chapter, the initialization phase of the N-computer operation cycle has been described. The reasons for the initialization of ETS and OTS were discussed.

In order to implement the data flow synchronization system in hardware, the number of tokens which can be propagated in the network at one time, is needed to be limited to a reasonable limit such that it can be implemented economically. Communication is considered to be one of the most expensive step for the execution of a program on the N-computer. Therefore, all the information which can be generated locally from an activity tree are not communicated remotely. The information required for scheduling a computation and generating output tokens is kept in the synchronization and output token formation system. Due to the limit on the token space, the tokens will be recycled when an activity tree requires more tokens as provided in the system. If the parts of the activity tree before and after recycling are not exactly the same, there will be a need for initializing the synchronization system.

Due to run-time extents of an F-code program, it is not always possible to know the ETS pattern for its execution at compile time and then compile the output tokens. It may be desirable to implement the output token formation system (or a part of it) in hardware, since, the logic for the generation of output tokens is the same and a separate faster implementation will increase the overlapping in the processing time and the output token formation time and hence will reduce the effective pipeline time i.e., the synchronization, processing and output token formation time, for the execution of an activity. Thus, to support the execution of a dynamic extent activity tree, there will be a need to initialize the data base (OTS) in the output token formation system when extents become known. In case of hardware (faster) implementation of the output tokens storage in the output token formation system, recycling
required to initialize the faster storage with the output tokens data base for a new activity tree.

In the operation cycle of the N-computer, the initialization time is a major delay. To minimize this delay, ETS and OTS initializing processes are designed to work independent of each other. The process for initializing ETS is very simple. To initialize ETS for \( n \) units of an activity at a processing node, the initializer will perform about \( 4/n \) operations (one addition, two conditions and one division) and \( n \) memory writes. A more complex way of distributing objects may require few more operations. However, the overall number of operations for ETS initialization are going to be very small, and therefore, an hardware implementation of the process is completely feasible.

The process for initializing OTS is, in general, much complex as compared to the ETS initialization. Therefore, if the program is of static nature, the major data base in the output tokens formation system (OTS2) must be initialized at compile time. By detecting regular patterns in the ETS for one activity, the OTS initialization can be optimized. To decode these optimizations, certain information is provided with an activity. The description of the macro OTS initializer follows that by parametrizing the activity with the template and then detecting regular pattern in the ETS locations of an activity tree, the complexity of the OTS initialization can be reduced significantly.

The distribution function \( (F) \) which can dynamically distribute the objects was selected and the logic for the implementation of \( F \) and its inverse was described. If there are \( n \) data elements of an object at a processing nodes, then to know the number of data elements the process \( (G) \) does not have to perform \( O(n) \) operations. The OTS for more than one output token was initialized in a pipeline. Accordingly, the logic for the evaluation of F-code geometric functions was described.

To store leaves and transition tokens\(^{11}\), the initialization of the starting token emitter (STE) has been described.

In the previous and this chapter, the N-computer has been explained. In the previous chapter the organization of the N-computer's processing node and its operation in the processing phase was highlighted. This chapter discussed the initialization phase of the N-computer. In the next chapter, the execution of F-code's functions on the N-computer will be discussed.

\(^{11}\)These are the tokens which flow across the global synchronizations, i.e., before and after recycling.
Chapter 6

Execution of Each Activity on the N-Computer

6.1 Introduction

In this chapter the ways to execute each activity on the N-computer are discussed. An F-code function is considered as an activity. To demonstrate the execution of each activity on the N-computer, there is a need to show the ways to initialize the resources, e.g., ETS, OTS and STE, for the activity.

All the F-code functions are divided into two types of activities, one is a geometric activity and the other is a non-geometric activity. Some of F-code's functions have a dual nature, i.e., they represent computation (non-geometric) as well as re-arranging their resulting data elements into a different object (geometric) according to their relation with the parent function. If a function has a geometric and non-geometric nature, then that function will be broken down into two activities in the activity tree, i.e., the geometric and non-geometric activities.

The non-geometric activities compute data elements and generate a resulting data element. These activities need ETS for their evaluation. The non-geometric activities transform child multi-index into parent’s multi-indices and therefore do not need any ETS. The evaluation of the non-geometric and the geometric activities are separated from each other, since, the geometric activities will be evaluated by the initializer. The initialization for the non-geometric activities is discussed in the following section. The evaluation of the geometric activities by the initializer is discussed in section 5.7.

6.2 Initialization for Non-Geometric Activities

For the initialization, all the activities (non-geometric) in an activity tree are divided into four classes. The initializer needs to know which class an activity belongs to. The classes
are as follows:

1. Unary activity, which consumes a single token to initiate its processing.

2. Binary activity, which consumes two tokens to initiate its processing.

3. Reduce activity, which reduces its argument with an associative and commutative operator.

4. Leaf activity, which does not consume any token to initiate its processing. These are constants and will be propagated in the network at appropriate time determined by the global synchronization.

All F-code's non-geometric functions in a program can be classified into one of these four activities. The resources (ETS, OTS and STE) are initialized for a part of the activity tree and then the activities are scheduled for processing with a dataflow synchronization by using the ETS model for synchronization of binary events. After the global synchronization another part of the activity tree will be processed. The continuation of tokens between the global synchronization is achieved by following ways:

1. By using STE.

2. By using global memory. Therefore, the activities read and write are introduced in the activity tree.

The read and write activities are defined as follows:

```
READ ::= ( reader EXPr^ adress )
```

The argument of read activity could be known at compile time or may be a run-time object. The shape of the read activity will be the object to be read. The control flow to execute the read activity is not given in the syntax of the activity, explicitly. It is mentioned in the subscript of the read activity. After the generation of the \( i \)th global synchronization, the read activity may be evaluated. It is assumed that before the generation of the \( i \)th global synchronization the \( EXPR_{address} \) has been evaluated. With this assumption, the read activity can be classified in the leaf activity for an activity tree. Therefore, in general, there is no need to use ETS for the read activity. STE will be initialized with the tokens for the processing system (Proc tokens). The tokens will be sent to the processing system after getting \( i \)th global synchronization signal. The tokens will be a read request for the data and the tag which will guide the data to its parent activity. In some cases, the tags and the data to be read can not be compiled. Assume the conventional von-Neumann style processor in
data/control flow ==> f
control flow ==>

Figure 6.1: A typical breakdown of an activity tree across the global synchronization with read and write activities.

the processing system. If the compiled code block has not been compiled with the output tokens due to the dynamic shape of the activity which was not available at compile time, then with the current definition of the system, the output tokens cannot be generated until the activity is synchronized in the ETS. In that case ETS is required for the evaluation of the read activity. If the processing node has multiple functional units to process activities, then again we will need ETS for the evaluation of the read activity. Therefore, the read activity can be considered as a unary activity and ETS can be used to evaluate it.

\[
\text{WRITE} ::= (\text{write EXPR_{address}} \text{EXPR.a})
\]

The write activity has two arguments, one is an expression which provides address to the global memory (EXPR_{address}) where the result of the second argument (EXPR.a) will be written. To support the execution of write activity in the N-computer\(^1\) it can be considered a binary or unary activity depending upon the nature of the address expression. If the address expression is known before the initialization of the ETS and OTS, then this can be considered a unary activity which needs just one token to initiate the execution. In either

\(^1\)Since, we have not selected a particular type of processor in the processing node, to establish links between global synchronization the execution of read and write activities are demonstrated.
way the ETS can be used to process the write activities. An activity tree with read and write activities across the global synchronization is shown in Figure 6.1.

6.3 Representation of the Activity Tree

An F-code program is written in the form of an activity tree to demonstrate the initialization. Each activity will be one of the four classes as mentioned in section 6.2, and there will be link between the child and parent activity. Each activity will have the other parameters as defined in chapter 5. To define the notations used to write an activity tree, consider the following piece of an F-code program.

```
......
(EXPR\_{parent}
    ......
        (diadic
            (EXPR.l)
            (EXPR.r)
        )
    ......
)
......
```

The activity tree for the above code is as follows:

```
......
Activity_{ch..., p1} - EXPR\_{parent}
......
Activity_{ch,p1, p1} - diadic function
......
Activity_{ch,p1, p1} - EXPR.l
......
Activity_{ch,p1, pk} - EXPR.r
......
```

Instead of using the parenthesis to represent the child and parent functions in F-code program, in the activity tree, the subscripts are used to represents these links. The first subscript represents the links for parent activities and the second subscript is a tag of the current activity, which may be used by a child activity to represent a link, e.g., the subscript \text{ch},p1, p1 will be interpreted as a child activity of parent with tag p1 and a parent activity with tag p1. This way of representation will eliminate the need to explicitly write the control functions which are not needed for the initializer when a child has more than one parent activity, since, the links between a parent and a child activity will satisfy the control
6.4 Initialization for Each Activity

Depending upon the processor in the processing node and the nature of the program the unary activities may not always need the ETS for their evaluation. If it is required to evaluate a unary activity (read, write, monadic) by using ETS, then the model of ETS can support it and ETS can be initialized. The Initialization for the diadic function of F-code is quite straightforward. The constant activity being a leaf activity, will initialize the STE. Monadic is of the unary type, i.e., there is only need for a single token to initiate the processing. The presence field of ETS location (p) can be set to full at initialization time for monadic activity and when a token is received by the synchronization system, the computation represented by the activity will be scheduled for processing. The initializer for diadic activity will work as follows:

1. Initialize ETS and OTS by using function INT, as defined in Appendix B, for left child of the diadic activity.
2. Initialize ETS and OTS by using function INT for right child of the diadic activity.
3. In parallel, ETS and OTS will be initialized as follows:
   (a) Initialize ETS for the activity by using function ETS_INT as defined in Appendix B.2.
Figure 6.3: ETS and OTS allocation for the two inputs b_choices needed to transform three input choice. Expr.s output tokens will trigger either b_choice depending upon the result of the function. The b.choices will also receive output tokens from expr.t and expr.f and send tokens to the parent expression. The broken and solid lines are used when two lines are crossing each other, just to differentiate between them.

(b) Initialize OTS for the activity by using function OTS.INT as defined in Appendix D.2.

6.4.1 The Choice Activity

As opposed to the diadic activity, the initialization of ETS and OTS for the choice is not so straightforward. The choice activity, depending upon the result of a logical activity, either returns the result of true or false activity. The choice consumes three input tokens but a single ETS location can only be used to synchronize activities with at most two inputs, i.e., binary activities. One method to support the execution of the choice function by using ETS is to convert it into two b_choice functions. The idea is demonstrated with the following example of a piece of an F-code program.

Example 11

.....

(EXPR_parent

129
Since this chapter discusses the initialization without taking into account the geometric semantics of F-code functions, the geometric functions are not included in Example 11. With the addition of geometric functions, the idea of breaking the choice into two b.choice will be the same. To write the F-code program with two b.choice, a new function named single, is introduced which is defined as follows:

\[
\text{SINGLE ::= (single EXPR.i ...)}
\]

The single function has more than one argument. All the arguments can be evaluated in any order, but they have the same parent function. The result of all the arguments will be one object consumed by the parent of the single function. Example 11 can be transformed into two b.choice functions activity tree as follows:

**Example 12**

\[
\begin{align*}
\text{(EXPR\_parent} \\
\quad \text{(comma right \_} \\
\quad \quad \text{(single \_} \\
\quad \quad \quad \text{(b\_choice \_} \\
\quad \quad \quad \quad \text{(EXPR\_t)} \\
\quad \quad \quad \quad \text{)} \\
\quad \quad \quad \text{(b\_choice \_} \\
\quad \quad \quad \quad \text{(EXPR\_f)} \\
\quad \quad \quad \quad \text{)} \\
\quad \quad \quad \text{)} \\
\quad \quad \text{(EXPR\_s)} \\
\quad \text{)} \\
\text{....}
\end{align*}
\]

The function comma, as defined in F-code, is the control activity needed to write the program with correct sequence control. In the processing cycle, the control flow will be
satisfied by the flow of tokens and there will be no evaluation of comma and single activities. In the activity tree of Example 12, expr.s has two parents, both b.choice functions. The ETS and OTS will be initialized for both parents (in the activity tree the choice flag, act.ot.token.c, of the expr.s will be set). Depending upon the result of expr.s, at runtime, only one token will be transmitted. The output token will initiate only one b.choice function. Both b.choice functions have same parent activity, EXPRparent. Therefore, the initiation of either b.choice will propagate the correct value (expr.t or expr.f) to the same function. The conversion of three inputs choice into two inputs b.choice is shown in Figure 6.2. Using two b.choices in this way not every token in the ETS will get matched. Therefore, to generate a global synchronization either the global synchronization count down field for ETS (act.ets.GSCDfield) and OTS (act.ot.token.GSCDfield) for these activities will be reset or the global synchronization counter registers GSCR1 and GSCR2 will be properly initialized. The allocation of ETS for two b.choices is shown in Figure 6.3. Out of all the ETS locations used for both b.choices, there will be only half of these locations which get matched. Similarly, all the output tokens initialized for the b.choices will not be transmitted. To comply with the functioning of the global synchronization generator, either act.ets.GSCDfield and act.ot.token.GSCDfield for both b.choices can be reset or the GSCR1 and GSCR2 can be initialized by half of the ETS and OTS locations for b.choice activity, respectively. The first option, reset both GSCDfields, demands that the body of expr.t and expr.f, both b.choices and the body of exprparent must fit in ETS\(^2\). Now the choice function can be initialized by using the binary ETS model and the initialization process. For the initializer the F-code tree of example 12, in the form of activity tree will be as follows:

Example 13

```
Activitych..., p1    - EXPRparent in example 12
......
Activitych,p1, p2    - first b.choice in example 12
Activitych,p1, p3    - second b.choice in example 12
Activitych,p2, p4    - expr.t in example 12
Activitych,p3, p5    - expr.f in example 12
Activitych,p2,p3, p6 - expr.s in example 12
......
```

The logic for the initialization process when we consider the activity tree as a list, will be as follows:

- Start initialization from the bottom of the list and walk towards the top of the list.

For example, when at activity Activitych,p2,p3, p6,

\(^2\)Physically, all the tokens must come within one ETS and OTS.
Main Expression
activity tree

Figure 6.4: A held expression may be recalled more than one times in the main expression of the hold function.

1. Initialize ETS and OTS for Activity_{ch\_p2\_p3, p6}.
2. Initialize ETS and OTS for Activity_{ch\_p3, p5}.
3. Initialize ETS and OTS for Activity_{ch\_p2, p4}.
4. Initialize ETS and OTS for Activity_{ch\_p1, p3}.
5. Initialize ETS and OTS for Activity_{ch\_p1, p2}.
6. Initialize ETS and OTS for Activity_{ch\_\ldots, p1}.

The initialization of expr.s and other activities in the activity tree will be exactly the same as in the diadic activity with the required information along each activity. For each b.choice function there will be no computation involved. It is just a synchronized redirection of a data element (value). Another way to evaluate choice is to generate global synchronization after the evaluation of expr.s, evaluate expr.t and expr.f. Also it could be supported by the compiled code.

6.4.2 The Hold Activity

A held activity may have more than one parent activity and in some cases it may be desirable to optimize it and use the dataflow synchronization for its execution, e.g., if a held object is recalled in the main expression (EXPR.a) for more than one time and the resources requirement of the activity tree is not much (enough ETS and OTS available) then the
tree can be executed without global synchronization. This is explained in section 6.4.6. Therefore, the OTS initialization supports more than one parent of an activity. A possible structure of an activity tree with a hold function is shown in Figure 6.4. The initialization for hold activity can be supported by appropriately writing the activity tree, i.e., the held activity will have links with its parent activities. Consider the following piece of F-code program with the hold function.

**Example 14**

```
(EXPRparent
  (hold A
    (EXPR.a  %% main expression of hold function
      (EXPRheld_parent1
        (var value A )
        
      )
      (EXPRheld_parent2
        (var value A )
        
      )
      (EXPRheld_parent3
        (var value A )
        
      )
    )
  )

  (EXPR.i  %% held expression of hold function
    
  )

)
```

The activity tree for the F-code program of Example 14 can be written as follows:

**Example 15**

```
Activity_ch..., p1  - EXPR_parent

Activity_ch..., p2  - EXPR.a

Activity_ch..., hp1  - first parent of held expression
```
In the above activity tree, each activity has a link to its parent activity, therefore by using the initializer as described in Appendix B, ETS and OTS can be initialized. The initializer will walk the list and initializes ETS and OTS in the same way as described in the previous section.

The execution of hold function on the N-computer can also be supported by writing the result to the global memory and then after global synchronization, reading the global memory as needed in the main expression. Example 14 for such sort of execution will become as follows:

**Example 16**

```lisp
(EXPR\_parent
  (EXPR\_a
    (EXPR\_held\_parent1
      (read <address>)
    )
    (EXPR\_held\_parent2
      (read <address>)
    )
    (EXPR\_held\_parent2
      (read <address>)
    )
  )
  GSYNCH
```

134
The activity tree for Example 17 is as follows:

**Example 17**

- **Activity** $ch_{i.., p1}$ – $EXPR_{parent}$
- **Activity** $ch_{.., p1}$, $p2$ – $EXPR.a$
- **Activity** $ch_{.., hp1}$ – first parent of held expression
- **Activity** $ch_{.., hp2}$ – second parent of the held expression
- **Activity** $ch_{.., hp3}$ – third parent of the held expression
- **Activity** $ch_{hp1..hp2hp3}$ leaf – read activity
- **GSYNCH**
- **Activity** $null_{.., p'1}$ – write activity
- **Activity** $ch_{.., p'1}$, $p'2$ – held expression ($EXPR.i$'s root activity)

The result of the held expression (expr.i) will be written to the global memory. After the global synchronization (GSYNCH) the held object will be read. The read and write activities have already been explained.

### 6.4.3 The Reduce Activity

We consider in this section, how to implement the reduce operation on the N-computer. It is highlighted that the binary synchronization model of the ETS is not a suitable model to achieve an efficient implementation of the reduce activity. The ways to achieve an efficient implementation of the reduce activity are discussed.

There are at least three ways to perform the reduction on a distributed memory machine. The first way is to reduce all the data elements at the processing nodes where the result of the reduce is mapped. This is called central reduction and is shown in Figure 6.5. In general, this is not an efficient way, since, if M elements are going to be reduced and we
Figure 6.5: All the elements which are going to be reduced to generate single element are propagated to the destination processing node and reduced there.

assume that they are elements distributed to the N processing nodes, then the amount of remote communication for the reduction is going to be $O(M)$. Since, ETS is a synchronizer for binary events, the number of ETS locations required at the destination to reduce M elements is going to be $M - 1$. The result of a reduction is not necessarily a scalar, it could be any data object of rank less than the source. If the resulting object is not evenly distributed to all processing nodes, then we may waste considerable ETS store\(^3\). Thus, there are two problems with the central reduction, one is the large amount of remote communication and the other is the possible wastage of the large number of ETS locations.

The second way is called *local/central* reduction. In the local/central reduction, first all the local elements are reduced at a processing node (local reduction) and then the result of each local reduction is reduced at the destination node (central reduction) as shown in Figure 6.6. In this scheme the amount of remote communication to reduce M elements, in the worst case, will be $O(N)$, where N is the number of processing nodes. If the data elements are distributed to P processing nodes ($P \leq N$), then the remote communication is $O(P)$. From the communication point of view, local/central reduction is better than central reduction.

The amount of ETS required to reduce M elements at a processing node is $O(M/P)$, that is if M elements are distributed evenly, the ETS at each processing node other than the destination node will be $M/P - 1$ and at the destination node will be $(M/P - 1) + (P - 1)$. Therefore, at most $P - 1$ locations may get wasted at a processing node. Thus from the ETS requirement and wastage point of view, this strategy is more efficient when compared to the central reduction. The process to support the initialization of ETS for the reduce activity, the initialization of the OTS for intermediate reductions and the initialization of

\(^3\)Since, to keep the initialization simple it is assumed that an activity is going to consume the same amount of ETS location at each processing node.
OTS for the child activity is more complex when compared to that for the central reduction. During ETS initialization, the process has to find the ETS location which will be used for the current data element and it has to initialize the OT field (if the output tokens are not compiled along with the code for the computation of each activity). For other activities the evaluation of the ETS and OTS location from the parameters ETS\_pt\_start and OTS\_pt\_start is quite straightforward. If \( m \) elements of an activity are mapped at a processing node, then to find out the ETS and OTS locations for the \( i \)th data element we just have to add \( i \) to the parameters ETS\_pt\_start and OTS\_pt\_start. For the reduce activity there will be a need to find out that how many local reductions for each element out of the \( i \) elements will be carried out at that processing node. The need to evaluate this information will complicate the initialization process. This issue will further be discussed in section 6.4.4.

The third way to support the reduce operation is called local/distributed reduction. In this scheme, all the local elements are reduced just as in the local/central reduction. Then the results of local reductions for adjacent (or nearest neighbours) nodes are reduced on one of the processing nodes which is nearest to the destination node, as shown in Figure 6.7. If the object which is being reduced, is distributed along \( P \) processors, then the communication will be \( O(P) \). But the number of hops for \( O(P) \) communications will be \( O(\log P) \). For \( M \) elements, the amount of ETS required at each node will be \( M/P \), i.e., one location extra as required in the local/central reduction. Therefore, at most one ETS location is going to be wasted at the first and the last processing node, i.e., \( P_1 \) and \( P_n \) in Figure 6.7. Thus the wastage of ETS is significantly less than in the local/central reduction which is \( P - 1 \). However, the initialization process is going to be more complex than for the previous two schemes. In order to know the ETS and OTS locations for the \( i \)th data element at a processing node from ETS\_pt\_start and OTS\_pt\_start, and how many local reductions for the first \( i-1 \) elements are going to be performed, there is a need to know how many distributed reductions will be
done at each processing node.

In the above three ways, to reduce $m$ data elements at one processing node, we need $m - 1$ ETS locations, since, each ETS location is a binary synchronizer. Thus the binary ETS model forces us to break the reduce operation into a binary tree and then execute that binary tree just like a diadic function as shown in Figure 6.8. However, there are some serious problems for the evaluation of reduce function with the binary ETS model, which are as follows:

1. The function reduce does not impose any order for the evaluation of the reduction operator. The operator can be applied as soon as any two elements are available. However, the above strategy imposes an order, e.g., if one token is available on the input arc of node 1 and 2 in Figure 6.8, both will be waiting for the other input for each activity and the operation will not be executed.

2. To support the initialization, we need to construct a tree as shown in Figure 6.8. The structure of this tree demands a significant amount of resources.

3. To reduce seven data elements as in Figure 6.8, we need six ETS locations. Each ETS location will store the same activity (computation to be done, act). Therefore, the redundant information will be stored in the ETS.

To overcome the above problems a new feature is added in the system. This feature is a counter associated with an ETS location. This is named a counter in the output token formation system. To make it possible the `count.pt` field is added in OTS1, and OTS3 is provided in the output token storage, as defined in section 4.9. With the counter associated with an ETS location, to reduce $m$ data elements, we just need one ETS location. In the light of the counter, the initialization for the local/central reduction is discussed in the following section.

6.4.4 Local/Central Reduction

The initialization for the reduction of all the local elements locally before the reduction of their results is a complex process. To initialize the ETS at a processing node, the system needs to know the local reduction coefficients for that processing node. If the final reduction is not at the same node, then local reduction coefficients for the final reduction processing node will be required at the local reduction processing node to generate the output token for the intermediate result of the local reduction. The initialization of OTS for the root activity of EXPR.a also needs the local reduction coefficients for the destination processing node, too. It is not necessary for all the processing nodes to have the same reduction coefficients.
Figure 6.7: All the local elements are reduced first and then the result of each local reduction is reduced in a distributed way.

Therefore, there is a need to know the local reduction coefficients for all the resulting elements of the reduce activity for the destination processing nodes (the processing nodes where the multi-indices will be reduced) at the producer processing node. This is an expensive step in the initialization process, because;

1. Either, it demands the access of the reduction coefficients for remote processing node during the initialization phase, i.e., there will be a remote communication during the initialization phase.

2. Or, it demands a substantial amount of storage for the local reduction coefficients for all the nodes at each node.

3. Or, it demands a process to evaluate the remote reduction coefficients during the initialization phase. This involves a considerable amount of processing.

If it is assumed that at least one local reduction will be performed for each resulting data element at each processing node in the network, then, by fixing one ETS for each resulting data element at each processing node, the amount of processing in the ETS initializer for reduce function can be minimized. In this case the amount of ETS required for the reduce activity at each processing node will be according to the shape of the resulting object. If the child data elements which are going to be reduced to generate a single parent data element are not well distributed, then an amount of ETS is going to be wasted at some processing nodes. For example, consider the example of Figure 6.8. In this example, seven data elements of the child object are going to be reduced to create a resulting data element. The resulting object will have thirty data elements. To initialize ETS, according to our assumption, we need thirty ETS locations at each processing nodes, one for each resulting data element. If the number of processing nodes is small as compared to the number of elements being reduced for each result and if these data elements are distributed in such a way that each
processing node has at-least one local reduction, then there will be no wastage of ETS, i.e., an allocation of ETS without being used.

The initialization process with this assumption will have more or less the same complexity as for the initialization for binary or unary activity. For the OTS initialization of EXPR.a, the process will have the following logic:

1. Find the index of the parent data element which will be the result of the reduction of current data element. This can be evaluated by performing a simple geometric operation (projection) on the current multi-index.

2. Find the ETS location which will be used for the parent multi-index from $ETS_{pt_{start}}$. Initialize the tag space in the OTS with this value.

3. If the local reduction coefficient for the current result is more than one, then make the tag local by appropriately initializing the processor field of the tag. Otherwise, initialize the processor field by the number of the processing node where the resulting data element is mapped.
Since, the ETS locations are allocated at each processing node according to the result of the reduce activity, there is no need to know the local reduction coefficients for the destination node at the producer node.

In the N-computer, the initialization of ETS and OTS is supported for the local/central reduction with the assumption that each processing node will have one local reduction for each resulting multi-index of the reduce activity. If the child object of the reduce activity is mapped in such a way that all the multi-indices which are going to be reduced to generate a reduced multi-index are at the same processing node, then there will be no need to perform initialization with the assumption. In this situation, the local/central reduction will be like the central reduction. For example, if a box which is mapped along the third dimension, then there is no need to adopt the local/central reduction if it is going to be reduced along the third dimension. This special case of the local/central is supported as the central reduction. The compiler or the run-time control system may choose one out of these two ways to execute the reduce activity. The reduce function in the following piece of F-code program can be written in the activity tree form as in Example 19.

**Example 18**

```
.....
(EXPRparent
.....
(Reduce
 (EXPR.a)
  )
    .....)
.....)
```

The activity tree is straightforward, which is as follows:

**Example 19**

```
.....
Activity_i, p1 = EXPRparent
.....
Activity_i, p2 = reduce activity
Activity_i, p3 = root activity of EXPR.a
.....
```

### 6.4.5 The Ramp Activity

To determine the value of each element of the result of the ramp activity, the process consumes two scalers and an index to the resulting element. A way to initialize ETS and OTS is
to broadcast $EXPR.b$ and $EXPR.s$ to all the processing nodes as a compound token. Every node, by just consuming these two tokens will generate all the data elements locally. The logic for the evaluation of each data element is shown in Figure 6.9. It is assumed that the extent of the result of the ramp is known. Consider the following ramp tree:

**Example 20**

```
.....
.....
(ramp
  (EXPR.b)
  (EXPR.e)
  (EXPR.s)
)
.....
.....
```

Since, the result of each argument of ramp function will determine the extent of the result of the ramp, for the initialization process there is a need to know the result of each argument. If the arguments are constant expression at compile time, then it can be evaluated at compile time. But if these arguments are run-time, then the ramp has to be evaluated at run-time. In that case, the logic for evaluation on N-computer could be as follows:
Example 21

```
.....
.....
(diadic add
   (read addressEXPR.b)
   (monadic index mul
      (read addressEXPR.s)
   )
)

GSYNCH
.....
.....
(write addressEXPR.b
   (EXPR.b)
)

(write addressEXPR.e
   (EXPR.e)
)

(write addressEXPR.s
   (EXPR.s)
)

.....
```

The output token for `EXPR.b`, `EXPR.e` and `EXPR.s` will have broadcast type. It is possible to compile the write activity with the code for `EXPR.s`, `EXPR.b` and `EXPR.e`. In this case the system has to make sure that the result of these activities has been written to all the processing nodes before the loading of these results after global synchronization. Otherwise, the system might get synchronized globally, even the write request are in the network. In this case, after the global synchronization the initialization process will read wrong values to form the extent of the tree (diadic in this case). The write activities in the tree of Example 21 will have extent according to the number of processing nodes in the N-computer. The result of their arguments (scalar) will be broadcasted to all the processing nodes. After processing the write activities, each activity at each processing node will generate a null output token just to satisfy the condition to generate the global synchronization. Therefore, in this case the write operation is supported as a unary operation in ETS, i.e., it will receive a single token containing the value to be written and the ETS will have address of the location where the data will be written. After the global synchronization, the run-time control system will evaluate the extent of the tree. Then the initializer will start the initialization process. The `monadic index` is a new activity which will, in this case, multiply each data element of its arguments with the index of that element. The tree of Example 21 can be written in the activity tree format as follows:

4The output tokens for these arguments will have broadcast communication type.
Example 22

\[
\begin{align*}
\text{Activity}_{ch., p1} & \quad \text{diadic activity} \\
\text{Activity}_{ch., p1, leaf} & \quad \text{read activity} \\
\text{Activity}_{ch., p1, p3} & \quad \text{monadic activity} \\
\text{Activity}_{ch., p3, leaf} & \quad \text{read activity} \\
\end{align*}
\]

\[
\begin{align*}
\text{GSYNCH} & \\
\text{null, p1} & \quad \text{write activity} \\
\text{ch., p1', p2} & \quad \text{EXPR.b activity} \\
\end{align*}
\]

\[
\begin{align*}
\text{null, p3} & \quad \text{write activity} \\
\text{ch., p3', p4} & \quad \text{EXPR.e activity} \\
\end{align*}
\]

\[
\begin{align*}
\text{null, p5} & \quad \text{write activity} \\
\text{ch., p5', p6} & \quad \text{EXPR.e activity} \\
\end{align*}
\]

The initialization for the above activity tree is quite clear in the light of the initialization process explained in Appendix B.

To evaluate ramp in this way, the monadic \text{index} activity, which is not a unary activity as defined earlier, is required. This is basically a binary activity for which one operand can be generated at initialization time from the shape of the activity (index of a unit will be one operand). Therefore, two types of unary activities, one is simple unary activity and the other is operated on index unary activity, are required.

6.4.6 The Assign Activity

In principle, the execution of the parents of assign activity needs global synchronization, since the assignment in F-code is intended to achieve side effects and the flow of data and control of the assign function is not the same. When the assign statement is being executed, the system may not know the flow of data (the destination could be defined at run-time). In many cases (if the compiler knows the destination), it may be possible to optimize the assign activity in such a way that the object being assigned will flow directly to the activity which is going to consume the object. This is termed dataflow optimization of the assign function. In general, it is an efficient way to support an assignment in the N-computer, since, it eliminates the need to first store the object in the memory and then retrieve it.

In imperative style programs, in general, a value is assigned to a variable (in data parallel program, this could be an object) and then recalled more than one time within the life time of the variable. In this way we don't have to evaluate the object every time we need it. An important question to consider is whether to allow the dataflow optimization for such
situations in the N-computer? If it is not allowed, then, when there will be more than
one parent of an assign function, the global synchronization has to be generated. In some
situations, this may not be a desirable condition. Suppose, there are enough resources (ETS
and OTS) for the dataflow execution of the assign and its parent functions (which are more
than one). Then the strategy in which the system first has to write the object to the
memory and then after the global synchronization retrieve it, is not an attractive feature of
the system. Therefore, in the N-computer, the option is provided to execute the dataflow
optimized assign function even with more than one parent.

In general, the execution of assignment in the N-computer is supported by the global
synchronization. The resulting object is written to the global memory after the execution
of the arguments of the assign. Then, after global synchronization the execution of parent
activity of assign will start. Consider the following piece of F-code for the assign function.

Example 23

```f-code
(assgin
  (EXPR.I)  %% this is a name object associated with the identifier A
  (EXPR.r)
)
```

A tree for the assign expression to show the logic for its evaluation is as follows:

Example 24

```f-code
(read address_assign)
```
The expression, expr.l, in the assign function will provide the address of the memory location where the data generated by the right child will be stored. The activity tree for Example 23 is as follows:

**Example 25**

```
....
Activity_{ch.., p1} - EXPR_{parent1}
....
Activity_{ch.., p2} - EXPR_{parent2}
....
Activity_{ch..p1.p2, leaf} - read activity
....
GSYNCH
....
Activity_{null, r'1} - write activity
Activity_{ch..r'1, p'2} - EXPR.r activity
....
```

### 6.4.7 The Gather Activity

In the initialization phase of the N-computer the geometric activities, which manipulate the multi-index of a child object to form a parent object's multi-indices, will be evaluated. Therefore, these activities are not discussed in this chapter. These activities, in general, are functions which manipulate regular patterns (dimensions) of an object, e.g., to coerce a matrix into a box, there is no need for the coercion function to have a set of input and
output indices in order to generate the result. Similarly, to have a slice from an object there is no need to have a set of input and output indices and a relation between them. By having a shape of the object and a simple rule, these geometric operations can be performed. As opposed to them the gather function of F-code does not transform a child object to its parent object by just having the shape vector and a simple principle. In general, it is a mapping of an object on to a logical object and then accordingly the selection or rejection of certain elements of the object to form a parent object. If a child object has $M$ data elements, then to form a parent object, in general, we need $M$ logical values. The shape of the parent object depends upon those logical values. To initialize ETS and OTS for the parent activity the system need to know the shape. The evaluation of the parent's shape by evaluating the gather function at initialization time by the initializer is going to be very expensive. Therefore, the evaluation of the gather function in the initializer is not supported.

It is suggested that the gather function is evaluated in the processing phase. A possible way is to compile the gather in an activity which will evaluate the function along with its arguments and write the resulting object to global memory. The extent of the resulting object will be broadcasted to all the processing nodes, too. Then, after global synchronization the parents activity will be executed.

6.4.8 Other Control Functions

The control functions of F-code, e.g., IF, LOOP, PAR etc., will be executed on the N-computer by using global synchronization. In some cases, it may be possible to rewrite the functions to generate an activity tree which does not need global synchronization, e.g., depending upon the nature of the program, optimization of IF function in the form of the choice function. This thesis does not discuss these optimizations.

The function IF is implemented in the run-time control system which will choose an activity tree out of two (true and false) for the initialization after global synchronization. The function LOOP will be evaluated by the run-time control system and the initializer will not initialize ETS and OTS for the execution of each iteration of the same activity tree. Assume, the argument of LOOP consumes a considerable amount of ETS. Then the initializer will interpret the loop and will not initialize the ETS and OTS after every global synchronization. Each iteration of the loop will generate the global synchronization. If the loop argument is not large enough, the loop activity can be executed by using ETS. At the end of the execution of the loop's argument, the root will trigger the loop activity in the ETS of each processing node, which in turn updates the global synchronization counter registers in the synchronization system and in the output token formation system and emits the leaves tokens for the argument. The function PAR can be evaluated using global synchronization.

147
After the evaluation of all its arguments, global synchronization can be generated. The function SPAWN can be evaluated just like the loop or PAR function.

6.5 Summing Up

It has been demonstrated that the N-computer is powerful enough to execute each non-geometric function of F-code by using dataflow and global synchronization. The conversion of an F-code program into the activity tree of four types of activities implies that the program will get executed on the N-computer. The flow of tokens between global synchronization is supported by read and write activities in the activity tree.

An F-code program with choice function can be converted into an activity tree with two b-choice functions. The hold function can be converted into the activity tree with or without the global synchronization. To reduce \( m \) elements, one location of ETS will be enough for synchronization. If the \( m \) elements are distributed to many processing nodes, then the elements at one processing node can be reduced locally and the result of each local reduction can be propagated to the destination node for the final reduction.

For the evaluation of the ramp by using ETS, the scalar values of its arguments are broadcasted to all the processing nodes and then by using two ETS locations, each resulting element of the ramp can be generated. The assign function can be converted into an activity tree with global synchronization. Depending upon the nature of the program and the availability of the resources, it can also be converted into the activity tree without the global synchronization. The evaluation of the function gather is not supported during the initialization phase and therefore it must be compiled as an activity by the compiler. The control functions, e.g., IF, LOOP, PAR and SPAWN, can be executed by using global synchronization or by compiling as an activity and then using ETS for the initiation of the execution of the activity. These can also be evaluated by the run-time control system, e.g., LOOP function.
Chapter 7
Towards the Prototype of the N-Computer

7.1 Introduction

This chapter describes the work carried out in the soft implementation of the N-computer system and for the simulation of the data distribution strategies. The purpose was to verify the functioning of the system and to gather some preliminary performance data.

First, the overall structure of the implementation is discussed with a brief introduction to each stage in the implementation. The implementation is divided into ten major stages. Then, in the subsequent sections, the implementation of each stage is explained. Some detail is necessary, however, to keep the discussion simple, every effort is taken to reduce the lower level details. The functioning of the system at each stage is demonstrated with some examples. The examples should be considered as a part of the explanation. Therefore, if it is difficult to grasp every point in the explanation, then a careful study of examples should make it clear.

The implementation of each feature in the soft prototype is not necessarily an efficient implementation on the target machine. Since, this is the first implementation of its type to simulate the N-computer, the emphasis was not on the efficiency of the implementation but rather on the demonstration that the logic works.

7.2 Organization of the Implementation

The organization adopted to implement the prototype system is shown in Figure 7.1. In this plan, an F-code program is converted into a syntax tree. The syntax tree is a machine representation of an F-code program. To convert an ascii representation to the machine representation, the logic has already been explained in chapter 3. Each node in the syntax tree has the same structure as the F-code function for which it represents. In the second
stage the syntax tree is converted into an activity tree. The structure of a node in the activity tree is different to the structure of a node in the syntax tree. One major difference is that the activity tree is a doubly linked tree in which it is possible to travel in either direction, i.e., from root to leaves or from leaves to root. At this stage, each activity in the activity tree will not have all of the parameters it requires, these will be inferred in subsequent stages, after creating associations for identifiers. This process walks the tree from the leaves to the root to obtain parameters, such as the shape of an activity. The coerced activity tree can then be used to simulate data distribution strategies and the operation of the N-computer.

For a given distribution function, the communication pattern of a program on a multiprocessor can be observed. All the remote references can be accumulated with appropriate weights such as distance traveled etc. The ratio between remote and local references can be used as one criteria to select a better distribution function.

For the simulation of the operation of the N-computer, the inferred activity tree is marked with global synchronizations. In this implementation, only the logical global synchronizations are marked and for the simulation of the operation of the N-computer, it is assumed that the system has sufficient resources for an activity within one logical global synchronization. In the future, this can be extended for the N-computer with fixed resources and accordingly the physical global synchronization can be marked in the activity tree. To verify the correctness, the activity tree with the global synchronization is printed to an output file. Due to the time constraints, the next stage, to simulate the operation of the N-computer, has not, been implemented. It can be implemented with the logic as already been explained in chapter 4 and chapter 5. Then each part of the activity tree (marked with the global synchronization) can be initialized with the remaining parameters and broadcasted to all the processing nodes. After getting first part of the activity tree, the operation cycle of the N-computer, as explained in chapter 4, will start.

The stages 1, 2, 3, 4 and 5, as shown in Figure 7.1, are implemented in software (in C). The stages 6 and 7 are partially implemented. The implementation of the syntax tree (stage 1) has already been discussed in chapter 3. The same logic is adopted for the current version of F-code. Therefore, the implementation of the process to convert an F-code program into a syntax tree is not discussed here. The implementations of the other stages are discussed in the following sections.

7.3 Stage 2: Activity Tree After First Pass

At this stage the activity tree is a doubly linked syntax tree in which it is possible to walk in both directions, i.e., from root to the leaves and from leaves to the root. This structure makes it possible to infer the attributes from the leaves of the syntax tree for an F-code
program. An activity with all major fields and the activity tree to which it belongs is shown in Figure 7.2. The first field (A1) is the name of the activity. The second field (A2) is the operator field which will be used to store normal operators as defined in F-code and a flag for the constant activity. If A1 is constant then the field A2 shows that whether the constant activity is used to define a subroutine (SUBRT) or it is a simple constant (CONST). If the constant activity is a simple constant, then the fourth field (A4) will have the value as given in the constant function. If the second field for a constant activity is SUBRT, then the child activity of the constant activity will be the subroutine expression.

If an activity has an identifier in its definition then the third field (A3) will have a pointer to the name of the identifier. The fifth field (A5) is a pointer to the parent activity and the sixth field (A6) will have pointers to all the child activities. The fifth field (A5) is organized in such a way that it can have pointers to more than one parent activity. There are some other fields (A6, A7) which will be initialized in subsequent passes and will be explained in their respective sections.

Other than the major fields shown in Figure 7.2, there are some other fields which are required to store the information defining the F-code function. Therefore, the structure with the fields as given, should not be considered as a sufficient structure to convert a complete program into the activity tree.

\[A \text{ logical field could be structure of more than one parameters.}\]
Figure 7.2: An activity with major fields. In the first pass, the fields A1 to A6 will be initialized. In the second pass (inference) the field A7 f will be initialized and so on.

After creating the activity tree from the syntax tree, it is printed to an output file (pretty print) to verify the correctness of the activity tree generator.

7.4 Stage 3: Identifier Associations; Second Pass

After the first pass, the activity tree does not have a symbol table to keep the associations of the identifiers. Each node explicitly has the name of the identifier if there is any. In the second pass the associations for the identifiers are created.

If an activity has an identifier, then after the second pass it will have a pointer to a list of associations for identifiers (ID_ASSOC). The list is used to keep the attributes of an identifier, too. There are two types of list, one is for channels and the other is for normal identifiers. The scope of common identifiers is a last in first out (LIFO) list while the channels are FIFO list. Therefore, we need a different logic to walk the list to make the current association unvalid when its scope expires. The structure of the list is shown in Figure 7.3.

Each entry in the list (ID_ASSOC) has three major fields. One is the pointer to the next
Figure 7.3: A list of identifiers associations with the structure of each association with major fields. Each field is explained in section 7.4.

entry (next_name) in the list. This will be used by the list walker program to get the proper entry to access a given identifier. The second field is the name of the identifier for which the current entry is. The third field is the horizontal extension (H.EXTN) of an entity in the list. If there is more than one association for the same identifier, i.e., the same name is used for more than one time to represent more than one identifier, then, more than one horizontal extension (H.EXTN) field can be added to the association.

There are five sub-fields in the horizontal extension field of an entity in the list. The first (1) and the fifth (5) fields make the horizontal extension a doubly linked list, i.e., it is possible to walk in either direction of the list to access the current association of the identifier, but the list is ordered and has a first and last entry. The third field is the scope flag which is used to represent the valid association for the current identifier. The valid association will be the last horizontal extension for the entity in ID_ASSOC list with scope flag set. The association will not be destroyed when its scope expires, since it will be used for further stages of the simulator. Since, the same structure is used for the list for channels with a different logic to destroy the scope of the current association, the scope flag is also used for the channels.

One important use of identifiers is in the use of channels in an F-code program. After creating a channel, a program may have a get function before the put function. For example, within the scope of a channel get and put functions can be used in the left and right arcs of a diadic function, respectively. Both arcs of the diadic function can be evaluated in any sequence. In the second pass, the simulator will create an entity in the list for channel
whenever it encounters a channel function. For a channel, the first function detected by the system which manipulates the channel (put and get) will create a horizontal extension (H.EXTN) in the channel list. To know that whether the channel has already been extended or not, a flag (ch.empty flag) is added in the horizontal extension.

All the fields of the horizontal extension (H.EXTN) except the field 2 (attrb.pt), are required in the current pass of the simulator and are used to access the proper association of the identifier. After the second pass, only field 2 is required for inference and coercion. This field is a pointer to a structure used to keep all the attributes of an identifiers (ID.ATTRB). Like other F-code functions an identifier may have dynamic (run-time) or static (compile time) attributes. This information is encoded in the first field of the attribute structure (ID.ATTRB). An identifier could be primitive, structured or a channel and we may need this information for the further stages of the simulator. Second field (ID.TYPE) of the attribute structure (ID.ATTRB) is used for this information. Depending upon the value of this field, fields 3, 4 and 5 are initialized. If the ID.TYPE is a channel, then the third field is used for proper sequencing in the next stages of the simulator. Remember, H.EXTN will not be used in the next stages of the simulator. For primitive and channel identifiers, the number of leaves (field 4 in ID.ATTRB) will be one and the field 5 (leaves pointers) will point to the structure LEAF.ATTRB. LEAF.ATTRB will have the attributes of the object which is associated with the identifier. In case the object associated with the identifier has run-time attributes, then the third field of the structure LEAF.ATTRB will have a pointer to the activity which will generate the primitive object associated with the identifier.

The template in F-code is a definition of a structured object with type and shape for each field. Identifiers are used to represents a template as well. Therefore, an identifier can be associated with the template rather than the object. The information associated with each field is called a label. The template name (identifier) can be used to create an object of given type and shape. In this case, the labels will be used to find the attributes of the object being created. The labels can be static or dynamic. For a static label,

the first field of the attribute pointer (LEAF.ATTRB) will have the shape and type of the label. If type is not primitive, then the field 5 (ATTRB_PT) will have a pointer to the type of the label. Non-primitive types of the label can be run-time parameters. The shape of the label can be a run-time parameter as well. For a run-time shape of the label, field 2 will have pointers to the shape expressions. The expression, object or template associated with identifier will have a destination, i.e., an activity which will consume the identifier. The field 4 (DEST) is used to keep pointers to the destination activities.

For the simulation of the N-computer and the data distribution strategies, if the nature of the program allows, then we may get rid of all the identifiers and appropriate links between
child and parent activities, i.e., just to keep the fields 3 and 4 of the attribute structure (LEAF_ATTRB) and integrate these fields into one activity. For the simulation of the N-computer and the data distribution schemes, there is no need to keep the identifiers.

By the end of second pass, if an activity has an identifier, then the activity will have a pointer to the attribute structure (ID_ATTRB). To demonstrate the functioning of pass two, the activity tree is printed to an output file with the pointers to ID_ATTRB for an activity if there are any. The following are the examples of the activity tree before and after the second pass. The pointers to ID_ATTRB are printed with the activity, as integers within the left and right angles (< and >). This also demonstrate the proper scoping of the identifiers.

Example 26 (hold/var)

Before Pass 2

After Pass 2

```
Before Pass 2

(hold a
  (diadic add
    (var value a)
  )
(hold a
  (diadic mul
    (var value a)
  )
  (diadic add
    (var value a)
    (ramp
      (const 1)
      (const 200)
      (const 3)
    )
  )
)
)

After Pass 2

(hold a <359296>
  (diadic add
    (var value a <359296>)
  )
(hold a <359360>
  (diadic mul
    (var value a <359360>)
  )
  (diadic add
    (var value a <359424>)
    (ramp
      (const 1)
      (const 20)
      (const 2)
    )
  )
)
)

In the above example, we can quite easily get rid of the hold function in the activity tree
```
by properly creating the links from a child activity to the parent activities. At this stage, the structure of the activity tree is very close to the actual F-code tree. The following example demonstrates the functioning of the system for the template and create functions.

**Example 27 (template/create/var)**

<table>
<thead>
<tr>
<th>Before Pass 2</th>
<th>After Pass 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(template z</td>
<td>(template z</td>
</tr>
<tr>
<td>(diadic add</td>
<td>&lt;359632&gt;</td>
</tr>
<tr>
<td>(create a</td>
<td>(diadic add</td>
</tr>
<tr>
<td>(diadic add</td>
<td>&lt;359768&gt;</td>
</tr>
<tr>
<td>(assign</td>
<td>(diadic add</td>
</tr>
<tr>
<td>(var name a)</td>
<td>&lt;359768&gt;</td>
</tr>
<tr>
<td>(diadic mul</td>
<td>(diadic mul</td>
</tr>
<tr>
<td>(10(const &quot;hfkdjk&quot;)</td>
<td>10(const &quot;hfkdjk&quot;)</td>
</tr>
<tr>
<td>(01(const &quot;uhjnlm&quot;)</td>
<td>01(const &quot;uhjnlm&quot;)</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>(const 40)</td>
<td>(const 40)</td>
</tr>
<tr>
<td>(const 2)</td>
<td>(const 2)</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>(const 5)</td>
<td>(const 5)</td>
</tr>
<tr>
<td>(const 6)</td>
<td>(const 6)</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>01(const &quot;nhipwj&quot;)</td>
<td>01(const &quot;nhipwj&quot;)</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>(const 10)</td>
<td>(const 10)</td>
</tr>
<tr>
<td>(const 20)</td>
<td>(const 20)</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
</tbody>
</table>

In the following example, a channel is created and three objects are pushed onto it with a defined sequence. Then these objects are popped from the channel. With the function `channel` only the scope of the channel is defined with some attributes. This definition creates an entity in the list of channel associations (ID_ASSOC as in Figure 7.3). Then every `put` function will extend this list horizontally (H_EXTN). Also a pointer to the second field of H_EXTN (ID.ATTRB) is kept with the put activity in the activity tree. LEAF.ATTRB of H_EXTN (fifth field) will also have a pointer to the activity whose result will be pushed onto the channel. For the `get` function, the pointer to the leaf attributes (LEAF.ATTRB) is copied to the activity. Also the pointer to the `get` activity is copied in the destination field (DEST) of the leaf attribute. This will make it possible to infer the attributes across the
identifiers and eliminate the channel, put and get functions (in general, if possible, all the functions which use identifiers can be eliminated in this way) and optimize the program as a data flow tree of activities which will be used for the simulation of both data distribution function and the N-computer. The logic is described diagramatically in Figure 7.4.

Example 28 (channel/put/get)

Before Pass 2

(channel a
(comma right
(comma left
 (get a)
 (comma left
 (get a)
 (comma left
 (get a)
 (get a)
 )
 )
 )
 )
 )

(channel a
(comma right
(comma left
 (get a)
 (comma left
 (get a)
 (comma left
 (get a)
 (get a)
 )
 )
 )
 )

After Pass 2

...... (channel a <0>
...... (comma right
...... (comma left
 (get a <358792>)
 (comma left
 (get a <358856>)
 (comma left
 (get a <358920>)
 (get a <358984>)
...... )
...... )
...... )
...... )

...... (comma right
...... (put a <358984>
...... (const 1)
...... )
...... )
...... )
...... )

...... (comma right
...... (put a <358980>
...... (const 20)
...... )
...... )
...... )
...... )
The following is an example for select function in the activity tree and the action on the activity tree in pass 2.

**Example 29** (select/template)

**Before Pass 2**

```
(template a
  (select value a 3
    (template b
      (diadic add
        (const "hello")
        (const "guys")
      )
      a
      (const 5)
    )
    integer
    (const 10)
    (const 20)
    real
    (const 20)
    (const 10)
    complex
    (const 20)
    (const 20)
    (const 10)
    (const 5)
  )
)
```

**After Pass 2**

```
(template a <360856>
  (select value a 3 <360856>
    (template b <361056>
      (diadic add <358856>
        (const "hello")
        (const "guys")
      )
      a <360856>
      (const 5)
    )
    integer</358792>
    (const 10)
    (const 20)
    real
    (const 20)
    (const 10)
    complex
    (const 20)
    (const 20)
    (const 10)
    (const 5)
  )
)
```

Similarly, the other activities which involve identifiers are processed in this pass. All the above examples are the output file (pretty print) after the second pass of the simulator.
7.5 Stage 4: Inference Level 1; Third Pass

All activities in an activity tree after the 2nd pass have no information about type, sort and shape. Since this information is not kept explicitly with each function of an F-code program\(^2\). Rather, there are well defined rules to evaluate this information for an expression from its argument expressions. In general, this information is made explicit at the leaves of the tree. To simulate the operation of the N-computer and the distribution function we need an activity tree with known shape if the shape is not a run-time expression. Although run-time shape expressions can be supported in the simulator, to keep the description simple at this level, it is assumed that the shape of an activity tree is known at compile time. The simulation of the N-computer with run-time shape expressions will be discussed while marking the global synchronizations in section 7.9.

To make the shape explicit for every activity, it is inferred. Since, an activity tree is not meant for code generation for the processing system, there is no need for the type inference. However, for data distribution simulation, we may need to know the length of the data which will be communicated between the processing nodes of the N-computer, therefore, the type of an activity tree is also inferred. We need sort inference, since the sort of the arguments of an expression affects the shape of the expression (choice function of F-code). Therefore, the sort of an activity tree is also inferred. After the third pass, each activity in the activity tree has four more parameters, i.e., type, sort, shape and rank as shown in Figure 7.2 (A8).

The simulation of the N-computer with run-time shape expressions will be discussed while marking the global synchronizations in section 7.9.

In the following examples the activity trees before and after the third pass are given. The inferred attributes are printed along the activity within braces \((\{\}\)) . The first element in the braces is the type, the second element is the sort, the third element is the rank and the fourth element is the shape vector.

**Example 30**

*Before Pass 3*

<table>
<thead>
<tr>
<th>Activity</th>
<th>Type</th>
<th>Sort</th>
<th>Rank</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>(diadic add)</td>
<td>..........</td>
<td>..........</td>
<td>..........</td>
<td>..........</td>
</tr>
<tr>
<td>10(const &quot;hello.pass3&quot;)</td>
<td>..........</td>
<td>..........</td>
<td>..........</td>
<td>..........</td>
</tr>
<tr>
<td>01(const &quot;test.pass3&quot;)</td>
<td>..........</td>
<td>..........</td>
<td>..........</td>
<td>..........</td>
</tr>
</tbody>
</table>

*After Pass 3*

<table>
<thead>
<tr>
<th>Activity</th>
<th>Type</th>
<th>Sort</th>
<th>Rank</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>(diadic add)</td>
<td>{i,v,2,&lt;11,10&gt;}</td>
<td>(diadic add)</td>
<td>{i,v,2,&lt;11,10&gt;}</td>
<td></td>
</tr>
<tr>
<td>10(const &quot;hello.pass3&quot;)</td>
<td>10 {c,v,1,&lt;11&gt;}</td>
<td>(const &quot;hello.pass3&quot;)</td>
<td>10 {c,v,1,&lt;11&gt;}</td>
<td></td>
</tr>
<tr>
<td>01(const &quot;test.pass3&quot;)</td>
<td>01 {c,v,1,&lt;10&gt;}</td>
<td>(const &quot;test.pass3&quot;)</td>
<td>01 {c,v,1,&lt;10&gt;}</td>
<td></td>
</tr>
</tbody>
</table>

In the third pass, the links between the parent and child activities which involve identifiers are also updated. If an activity has an identifier, then it will have a pointer to the memory.

---

\(^2\)This makes the representation of a program in F-code concise.
location for the attributes of the identifier (ID_ATTRB). The structure ID_ATTRB has a
pointer to the destination and the source activity for the object associated with the identifier.
In the third pass, by using those fields of ID_ATTRB, the links between the child and
parent activities are established. To verify the number of parents for each source activity for
identifiers the number of parents will be printed in the output file just before the activity
(P = ..). As is clear from the following example.

Example 31

Before Pass 3

\[
\text{(hold a } <355848> \\
\text{(diadic add } \\
\text{(diadic add } \\
\text{(var value a } <355848>)} \\
\text{(const "jjjhffj") \\
\text{(diadic add } \\
\text{(const "kdffkdfkdfdj") \\
\text{(var value a } <355848>)} \\
\text{(const "kfdjfdkkdfdj") \\
\text{) \\
\text{) \\
\text{)}} \\
\text{)}} \\
\text{)} \\
\text{)}
\]

After Pass 3

\[
\{i,v,l,<7>} \text{(hold a } <355848> \\
\{i,v,l,<7>} \text{(diadic add } \\
\{c,v,l,<12>} \text{(var value a } <355848>)} \\
\{c,v,l,<7>} \text{(const "jjjhffj") \\
\{i,v,l,<7>} \text{(diadic add } \\
\{c,v,l,<11>} \text{(const "kdffkdfkdfdj") \\
\{c,v,l,<7>} \text{(var value a } <355848>)} \\
\{c,v,l,<7>} \text{(const "kfdjfdkkdfdj") \\
P=2, \{c,v,l,<12>} \text{(const "kfdjfdkkdfdj") \\
\text{)}} \\
\text{)}} \\
\text{)}} \\
\text{)}}
\]

7.6 Stage 5: Inference Level 2; Fourth and Fifth Pass

There are some situations when it will become more complex to infer the attributes in
one pass of the activity tree and the list for identifier attributes as used in the simulator.
Therefore, those situations are addressed in a separate pass of the activity tree. For example,
to know the label of a field of a template, we need to evaluate the integer scalar expressions
associated with it. Consider Example 32, after defining the template a as a structure object
which has three fields, i.e., integer, real and complex, an object of type a is created and
associated with the identifier z. Then, elsewhere in the program, the third field of the
structured object of type a is selected. In the third pass, all the field expressions are treated
as run-time. Therefore, the select and create functions do not know the extent of the object
being selected and created. For such a program, the output after the third pass is printed
with run-time shape, as shown on the right hand side of Example 32.

160
Example 32

Figure 7.5: Structure of the implementation of the simulation of the data distribution strategy 1.

<table>
<thead>
<tr>
<th>After Pass 2</th>
<th>After Pass 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(template a &lt;360296&gt;</td>
<td>(template a &lt;360296&gt;</td>
</tr>
<tr>
<td>(create z</td>
<td>(create z &lt;360496&gt;</td>
</tr>
<tr>
<td>(diadic add</td>
<td>(diadic add</td>
</tr>
<tr>
<td>(const &quot;hello&quot;)</td>
<td>(const &quot;hello&quot;)</td>
</tr>
<tr>
<td>(select value a 3</td>
<td>(select value a 3 &lt;360296&gt;</td>
</tr>
<tr>
<td>(const &quot;test&quot;)</td>
<td>(const &quot;test&quot;)</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>a</td>
<td>a &lt;360296&gt;</td>
</tr>
<tr>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>(const 10)</td>
<td>(const 10)</td>
</tr>
<tr>
<td>(const 10)</td>
<td>(const 10)</td>
</tr>
<tr>
<td>(const 20)</td>
<td>(const 20)</td>
</tr>
<tr>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>(const 20)</td>
<td>(const 20)</td>
</tr>
<tr>
<td>(const 10)</td>
<td>(const 10)</td>
</tr>
<tr>
<td>complex</td>
<td>complex</td>
</tr>
<tr>
<td>(const 20)</td>
<td>(const 20)</td>
</tr>
<tr>
<td>(const 20)</td>
<td>(const 20)</td>
</tr>
<tr>
<td>(const 10)</td>
<td>(const 10)</td>
</tr>
<tr>
<td>(const 5)</td>
<td>(const 5)</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
</tbody>
</table>

In the fourth pass, the shape expressions are evaluated. In fact this pass is an interpreter for the shape expressions. The result of the shape expression is stored in the shape vector. In the fifth pass, the shape of the activities is inferred again and the activity tree is printed.
to an output file. The output for Example 32 after fifth pass is as follows:

**Example 33**

**After Pass 5**

\[
\{i,v,l,<4>\} (\text{template } a \ <360296> \\
\{i,v,l,<4>\} (\text{create } z \ <360496> \\
\{i,v,l,<4>\} (\text{diadic add} \\
\{c,v,l,<5>\} (\text{const } "hello") \\
\{\text{cmp},v,<4,20,20,10,5>\} (\text{select value } a \ 3 \ <360296> \\
\{c,v,l,<4>\} (\text{const } "test") \\
) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \\
\) \<

**7.7 Stage 6: Coercion; Sixth Pass**

In the third and fifth pass, the attributes in an activity tree are propagated from leaves to the root of the tree to get the attributes for each activity in the activity tree with the rules as defined in F-code. In this stage, the shape of a parent activity is propagated down to the shape of the child activities with the same rules but in reverse order. This will make it possible to simulate the system for those elements of the child activity which are going to be consumed by the parent activity. After the sixth pass, the coerced activity tree is printed again to an output file, as is clear from the following example.

**Example 34**

**Before Pass 6**

\[
\{i,v,l,<8>\} (\text{diadic mul} \\
\{i,v,l,<8>\} (\text{diadic add} \\
\)
\]

**After Pass 6**

\[
\{i,v,l,<8>\} (\text{diadic mul} \\
\{i,v,l,<8>\} (\text{diadic add} \\
\)
\]

162
Figure 7.6: An activity tree before and after LGS markings. Each LGS node has a part of the activity tree as its child. The N-computer simulator will process the leaf LGS by executing its child activity and then process its parent LGS and so on.

\[
\begin{align*}
\{c,v,l,<9>\} & \text{(const "lenghtis9")} & \{c,v,l,<8>\} & \text{(const "A_string")} \\
\{i,v,l,<10>\} & \text{(diadic add)} & \{i,v,l,<8>\} & \text{(diadic add)} \\
\{c,v,l,<10>\} & \text{(const "hello_fred")} & \{c,v,l,<8>\} & \text{(const "hello_fred")} \\
\{c,v,l,<13>\} & \text{(const "test_this_one")} & \{c,v,l,<8>\} & \text{(const "test_this_one")} \\
\end{align*}
\]

The following example shows the coercion for activities which involve the identifiers.

**Example 35**

**Before Pass 6**

\[
\begin{align*}
\{r,v,l,<5>\} & \text{(hold a <356320>} & \{r,v,l,<5>\} & \text{(diadic mul} \\
\{r,v,l,<5>\} & \text{(diadic add} \\
\{i,v,l,<5>\} & \text{(diadic add} \\
\{i,v,l,<8>\} & \text{(var value a <356320>)} \\
\{c,v,l,<5>\} & \text{(const "hello")} \\
\{r,v,l,<8>\} & \text{(diadic div} \\
\{c,v,l,<12>\} & \text{(const "see_the_test"}) \\
\{i,v,l,<8>\} & \text{(var value a <356320>)} \\
\end{align*}
\]

**After Pass 6**

\[
\begin{align*}
\{r,v,l,<5>\} & \text{(hold a <356320>} & \{r,v,l,<5>\} & \text{(diadic mul} \\
\{r,v,l,<5>\} & \text{(diadic add} \\
\{i,v,l,<5>\} & \text{(diadic add} \\
\{i,v,l,<5>\} & \text{(var value a <356320>)} \\
\{c,v,l,<5>\} & \text{(const "hello")} \\
\{r,v,l,<5>\} & \text{(diadic div} \\
\{c,v,l,<5>\} & \text{(const "see_the_test"}) \\
\{i,v,l,<5>\} & \text{(var value a <356320>)} \\
\end{align*}
\]
After having a coerced activity tree, we can simulate the function of the N-computer as well as different distribution strategies. In the next section, the implementation of the data distribution simulation will be discussed. The results of the simulation of some distribution strategies will be presented. Due to the time limitations, all the possible distribution strategies are not simulated.

### 7.8 Stage 7: Data Distribution Simulation

The operation of the N-computer also involves the implementation of the different variants of a distribution function, therefore, the simulation of distribution strategies has been implemented.

Each activity in the coerced activity tree has some information required for the simulation...
of data distribution strategies. This structure could also be extended if necessary to match the requirement for some given distribution strategy. In the current implementation, for a given distribution strategy, the simulator determines all local and remote references for each activity in the activity tree. Then for the given number of processing nodes, the ratio between local and remote references is evaluated. This ratio is plotted against the number of processing nodes. In the following sections, the implementation of the simulation of different distribution strategies are discussed.

7.8.1 Distribution Strategy 1

In this strategy, for each activity in an activity tree the objects are distributed in a linear cyclic way starting from processing node 1. The structure of the implementation is shown in Figure 7.5. Different programs and the results of the simulations of this distribution strategy are presented in the following section.

Some Examples

For the simulation of the examples in this section, each object is distributed to all the processing nodes of the N-Computer element wise. The distribution starts from the first processing node. The simulator evaluates local and remote references for each activity in the activity tree. Then, the local and remote references for all the activities are summed up and the ratio between the local and remote references is evaluated. This ratio is plotted against the number of processing nodes in the network.

Example 36

F-code Tree

```
(diadic mul

110 (diadic add

01(ramp

(const 0)

(const 500)

(const 2)

)

10(ramp

(const 0)

(const 100)

(const 2)

)

011 (diadic add

01(ramp

(const 0)

)

```

Coerced Tree (after pass 6)

```
\{i,v,3,<51,51,167>\}\{diadic mul

\{i,v,2,<51,51}\}\{diadic add

\{i,v,1,<51}\}\{ramp

\{i,v,0,<>\}\{const 0 \}

\{i,v,0,<>\}\{const 500 \}

\{i,v,0,<>\}\{const 2 \}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}

\}
The result of the simulation for the activity tree in the above example is shown in Figure 7.7. For the program in Example 36 of if we have 3, 17 or 51 processing nodes for the execution of the program, then, for the given distribution strategy the ratio between local and remote references will be maximum. The compiler can fix the number of processing nodes for a given program by detecting the peaks in the local/remote references ratio graph. To achieve higher local to remote references ratio for the given distribution scheme the objects might have to be distributed to $m$ processing nodes in a network of $n$ processing nodes, where $m$ is less than or equal to $n$. If $m$ is less than $n$, then this will be at the cost of reducing the exploitation of parallelism available in a program, which is an obvious compromise to reduce the communication.
Depending upon the nature of the program, there may not be many peaks in the local to remote reference ratio. Instead, the ratio may go on decreasing as the number of processing nodes increases. This is clear from the simulation of the following example.

Example 37

F-code Tree                      Coerced Tree (after pass 6)

(diadic mul) 101 (diadic add) 01(ramp)
      (const 0)          (const 100)          (const 3)
      (const 500)       (const 100)       (const 2)
      (const 2)         (const 2)         ......  

101 (i.v,3,<51,51,167>) (i,v,8,<51,167>) (i,v,1,<167>)

The result of the simulation for the activity tree in the above example is shown in Figure 7.8. For the given program and the distribution scheme the local/remote references ratio falls as the number of processing nodes increases. As opposed to the ratio in Figure 7.7, there is no significant peak.

In this distribution scheme, first the junior most dimension is distributed along the processing nodes and then the next senior dimension and so on. Therefore, if a child object of dimensions $i$ is not coerced for the first $i$ dimensions of the parent object and the extent of each dimension for the first $i$ dimensions of the parent object is divisible by the number of processors in the network, then, all the replications of child object are going to be consumed
This situation for the given distribution scheme is called fully alignment of the child and parent objects. There could be a situation when objects are partially aligned. In Example 36 nearly all the objects are either aligned or partially aligned when the number of processing nodes are 3, 17 or 51. However, in Example 37, this situation does not come for any processing nodes in the network (from 2 to 100). Therefore, the number of remote communication is higher than the number of local communication. Thus, for the given program this distribution scheme is not recommended.

### 7.8.2 Distribution Strategy 2

Instead of element cyclic distribution, the object can be distributed with a block cyclic distribution scheme. Blocks can be formed by detecting the communication pattern of the program.

The dimension of a parent object for which the child object is replicated (rank coercion) can be detected and the blocks for the distribution of parent object can be formed along that dimension. In this situation, for all the element in a block the child activity will just send one compound token (compound tokens are discussed earlier in section 5.11.1). And the child element is replicated at the destination processing node. This distribution strategy is adopted for the program in the following section.

**An Example**

An object is divided into blocks and the blocks are distributed to the processing nodes in a cyclic way. Depending upon the nature of the program, there could be many ways to form a block. Consider Example 37 again.

If we make blocks of a parent object along one of the dimension for which the child object is replicated (rank coerced) and distribute these blocks to the processing nodes. Then, we may improve the local to remote references ratio significantly. To see the improvement, let us make blocks of each parent object along the dimension for which the left child is being replicated and distribute these blocks to the processing nodes in a cyclic way. It means the left child will send a compound token for its parent activity but the right child will send simple tokens. The result of the simulation of this distribution strategy for Example 37 is shown by the local to remote references ratio plot of Figure 7.9. There is a significant improvement in L/R (local to remote references) ratio as compared to the plot of Figure 7.8. This simulation only distribute blocks if the number of blocks are more than or equal to the number of processing nodes in the network. If the number of blocks are less than the processing nodes then the object will be distributed element wise. Therefore, for processing

\(^3\)It is assumed that there is no length coercion involved at run-time.
nodes more than 51 the curve in Figure 7.9 is the same as the curve in Figure 7.8. Since, the maximum number of blocks are 51. By comparing results of Figure 7.9 with those in Figure 7.8, one can see the improvement in the local to remote references ratio.

Due to the time constraints, not all distribution schemes have been simulated. In future, the simulation can be extended for other distribution strategies. Also in order to get more concrete results more information must be processed, e.g., the weight of each remote and local reference and the processing time for each activity in the activity tree. These parameters can be approximated after studying the lower level details of the N-computer which will be the second phase of this research.

7.9 Stage 8: LGS Marked Activity Tree

For the simulation of the N-computer, after having a coerced activity tree, the next stage was to mark the logical global synchronization in the activity tree. Partitioning of a single activity tree into more than one activity tree with global synchronizations is shown in Figure 7.6. In this stage, the tree walker walks the activity tree and detects activities which need global synchronization and accordingly forms the LGS tree. The LGS tree generator also establishes links between different parts of the activity tree separated by the global synchronizations. These links are made by using global memory write and read before and
Figure 7.10: The formation of the LGS tree in two stages from an activity tree with the function ramp. In the first stage all the LGSs are marked and in the second stage, all parallel LGSs are converted into one LGS.

after the synchronization.

The conversion of an activity tree into the LGS tree as shown in Figure 7.6 is done in two stages. To understand the issue, consider the activity tree with ramp function as shown in Figure 7.10. Assume, the three children of the ramp are non-constant expressions. The results of the arguments of ramp are required to evaluate the shape of the resulting object, i.e., shape of the ramp. As discussed in section 6.4.5, one way to support the execution of the ramp function on the N-computer is with the global synchronization. The global synchronization is generated after the evaluation of its arguments and then the ramp will be processed. At the top of the activity tree, the LGS marker generates an LGS node and the top activity will become the non-LGS child of the LGS activity. When the tree walker knows that the ramp has non-constant expressions, it creates an LGS child node in the LGS tree. The ramp function is terminated with a read request. The argument of the ramp function is made a non-LGS child of the newly created LGS node in the LGS tree with a write request as the root of the tree. In the second stage, all the parallel LGSs in the LGS tree are detected and combined into one LGS to form the final LGS tree. In the current implementation only the first stage is implemented.

To verify the functioning of the system, the LGS tree is printed to an output file. For an example, the following F-code program is input.
Example 38

An F-code tree

\[
\begin{align*}
&\text{hold a} \\
&\text{hold b} \\
&ramp \\
&\text{const} \\
&\text{diadic mul} \\
&\text{var pointer a} \\
&\text{const 2} \\
&\text{var pointer a} \\
&\text{diadic add} \\
&\text{const 1} \\
&\text{var pointer b} \\
&\text{const "hello_world"} \\
&\text{const "test_string"}
\end{align*}
\]

In the above example, the arguments of the ramp function are treated as the run-time expressions. The first argument is a pointer to a subroutine, the second argument is a pointer to an identifier and the third argument is again a run-time expression.

To support the simulation of the program which has pointers, a pointers list is maintained. The name of that list is identifier pointers list (ID_PT_LIST). Whenever, there is a need to generate a pointer in a program for either a subroutine or an object, a new association in ID_PT_LIST for the subroutine or the object is created and the index to that list, as an integer value, is returned into the program. Further in the program, if a pointer is used to call a subroutine or to access an object, the integer value which refers to the pointer is used as the index to the list, ID_PT_LIST, and the subroutine or the object is accessed accordingly.

The activity tree after stage 8 for the above example is given below. Each LGS may have two types of child. One is non-LGS child and the other is LGS child. In the printout, the non-LGS child is printed in curly braces ({ }) and the LGS child is printed in parenthesis.

Example 39

LGS tree for example 38 after Stage 8

The heap pointer for the attributes of identifier a is <333424>
The heap pointer for the attributes of identifier b is <333528>
The identifier pointers list, ID_PT_LIST, with index and the association of each index for Example 38 is as follows:

\[
\begin{align*}
\text{ID_PT_LIST index} & = \langle 0 \rangle \text{ start} \\
& \quad (\text{diadic mul} \\
& \quad \quad (\text{var pointer a } \langle 383528 \rangle) \\
& \quad \quad (\text{const } 2)) \\
\text{ID_PT_LIST index} & = \langle 0 \rangle \text{ end} \\
\text{ID_PT_LIST index} & = \langle 1 \rangle \text{ start} \\
& \quad \{c,v,1,\langle11\rangle\} (\text{const } \text{"hello-world"}) \\
\text{ID_PT_LIST index} & = \langle 1 \rangle \text{ end} \\
\text{ID_PT_LIST index} & = \langle 2 \rangle \text{ start}
\end{align*}
\]
Yet, the attributes for the subroutines are not inferred, as is clear from the first entity in the identifier list above.

7.10 Summing Up

A complete plan to implement a soft prototype of the N-computer system, i.e., the operation of the N-computer and its problem domain, has been presented in this chapter. The implementation of the soft prototype is not very simple. The whole plan was divided into 10 stages. Although, the complete plan could not be implemented due to the time constraints, a major portion of it has been implemented.

The operation of the N-computer also involves the implementation of different variants of a distribution function. Therefore, the simulation of distribution strategies has been included in the plan. To find an optimum distribution for a program, two distribution schemes have been simulated. The results show that a scheme which optimally distributes a program will not necessarily optimally distribute another program. Therefore, to achieve an efficient execution of a program on a distributed memory machine, the system must support different ways to distribute the objects, which can be selected depending upon the nature of the program. To support this in the N-computer, the distribution parameters have been included in the structure of each activity. During the initialization phase, the initializer evaluates different variants of the distribution function according to the distribution parameters with the activity.

To demonstrate the functioning of the system, an F-code tree has been inferred, coerced and then converted into a global synchronization marked activity tree. Each partition of the activity tree within the global synchronization will be used to initialize the resources (ETS and OTS) of the soft implementation of the N-computer. The activities will then be processed in the processing phase. Due to time constraints, soft prototype of the operation of the N-computer could not be completed.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

A universally accepted issue in parallel computing, is the lack of generality. It has two aspects:

1. The lack of a suitable and simple framework to write, debug, and efficiently execute portable applications.

2. The lack of any scalable parallel computer which can execute these applications fast, cheaply and accurately.

Despite much effort to achieve general purpose parallel computing, these problems have not yet been solved. This thesis has addressed both issues in general while focusing on the second one in particular.

From the start of this work, it has been understood that solutions would only be achieved by carefully studying both the software and hardware issues concerning this problem. The core of the work was the use of a data parallel description of the problem domain and the use of a data driven architecture to accommodate the high-latency and non-deterministic accesses to the distributed shared memory. The major achievement of this work has been the understanding of the marriage of these two issues, which has enabled advances to be made in dataflow architectures exploiting the collective behavior of data parallel programs. For example, special matching rules can be used to minimize the synchronization overheads when implementing symmetric operations such as reduction. Some more tangible results of this work are the high-level design and partial simulation of an implementation of such an architecture. This work details the algorithm for the execution of each of the data parallel operations in F-code. In the following paragraphs more specific conclusions made in this thesis are summarized.
While addressing the first issue, the suitability of F-code as a PSP was studied and it was concluded that F-code has a great potential to become a PSP for data-parallel high-level languages since:

- it is architecture independent,
- it is sufficiently abstract to be general enough without limiting congruency and,
- it has well-defined and simple semantics which preserve parallelism.

However, it was observed that there is a need for some refinement in the definition of F-code, which can be summarized as follows:

1. It has been demonstrated [19, 20, 21, 22] that heterogeneous objects can be simulated with homogenous objects at a higher level. Therefore, the elimination of structured objects (template) would add simplicity in the semantics of F-code.

2. The usage of pointers is not very well defined, e.g., pointers to primitive objects can be created, but the way to manipulate the objects by using the pointers is not fully defined. There is also a need to define a virtual distribution scheme for F-code objects, which make the arithmetic on pointers properly defined. Otherwise, it will not always be possible to prove correctness of an F-code program without assuming certain implementation details.

3. To preserve the essential features available in a high level program in order to prove that the implementation of a program via PSP will not be less efficient, there is a need for the addition of primitive type sizes in F-code. For example, a program in Fortran 90 may use 32 bits as well as 64 bits floating point, and its direct implementation may optimize the execution of a program according to the type sizes, which will be impossible in F-code.

4. F-code does not support the external procedure call of a Fortran 90 program, therefore, there is a need to define the way to declare subroutines and then call them in a program.

Under the light of above conclusions, a revised definition of F-code has been suggested in chapter 2.

To address the second issue for general purpose parallel computing, a machine related model, dataflow bulk synchronous machine (DFBSM) model, was formulated. This thesis describes the system-level design of a dataflow bulk synchronous machine, the N-computer.

There have been two major problems with the design of modern dataflow machines, such as EM-4 [100] and Monsoon [82], which are the use of the I-structure to introduce state in a
dataflow machine and token recycling. Each operation on a location in the I-structure storage demands a three states synchronizer. The dataflow bulk synchronous machine does not need I-structure storage for reading and writing data to memory. The global synchronization (bulk synchronization) in DFBSM, which synchronize all or a subset of processing nodes at certain times during the execution of a program, provides an alternate way of maintaining memory consistency on assignment. The same signal (global synchronization) is also used for recycling the tokens space.

The N-computer is the first known proposal for a dataflow bulk synchronous machine. Even the idea of dataflow bulk synchronous model originated in the Computer Systems Research Group under the direction of Chris Jesshope. I could not find any reference of similar work elsewhere. Due to the regular structure of a data parallel applications, the run-time support to exploit fine grain parallelism is relatively simple. In short, the implementation of dataflow bulk synchronous model for data parallel applications essentially increases the scope of the N-computer as a machine of our specifications as described in section 1.3 of chapter 1.

An activity tree, which is generated by the compiler from an F-code program in addition to the compile code, will act as the source for generating all the necessary information at each processing node required to support distributed execution of the code with dataflow synchronization and scheduling. For that purpose, the N-computer has two phases in its operation, i.e. a processing phase and an initialization phase. In the initialization phase, the resources at each processing node, e.g., synchronization and output token formation are initialized from the activity tree. In the processing phase, each processing node of the N-computer can be considered as a three stage pipeline, i.e, synchronization, processing and output token formation (each stage may be further pipelined, which is an implementation decision). Tokens will flow between these stages. The structure of an N-computer node with different sub-systems and the interaction between these sub-systems, has been described in some details. However, it should be noticed that although the output token formation stage is the most complex, it can proceed in parallel with the processing stage. Therefore, depending upon the processor, the output token storage (OTS) in the output token formation sub-system can be implemented on a slower memory.\footnote{To be more specific, only OTS2 in the output token formation system should be implemented on a slower memory.}

When programs are of static nature, it will not only be possible but also recommended that the output tokens be generated at compilation time for each activity. When this is done there will be no need to initialize the OTS at run-time, and only the ETS will need to be initialized. Therefore, it was desirable to initialize ETS and OTS independently of each other. To achieve this and at the same time to keep the initialization processes simple,
the output token storage has been organized into two sections (OTS1 and OTS2). The initialization of OTS can be optimized by detecting regular patterns of communication in a program. It is also not always necessary to initialize ETS after every global synchronization, e.g., for different iterations of the same loop.

The functioning of each sub-system in both phases of operation has been described with logically consistent algorithms which were then written in a pseudo Occam like language. In most cases, by adding some lower level implementation details, these descriptions will have a one to one correspondence with their implementation in software.

It has also been demonstrated that the N-computer can execute F-code functions, by describing ways to initialize ETS and OTS for non-geometric functions of F-code in the processing phase in chapter 6 and for geometric functions in the initialization phase in chapter 5. The addition of a choice bit (c bit) in OTS and the global synchronization count down field (GSCD) in ETS and OTS has made it possible to execute choice function without the generation of a global synchronization signal and at the same time to guarantee the generation of global synchronization. Another important issue was the execution of the function reduce and the support of the macroactor. To achieve this, a counter storage (OTS3) is added in the output token formation. Due to this feature, a counter can be associated with the output token generation for an activity. In fact, this feature offers the use of the same ETS location for non-binary (more than two inputs) activities without imposing any restriction on the order of arrival of the input tokens. The execution of a macroactor or strongly connected arcs activity can be supported by generating null tokens for the arrival of first $n-1$ tokens and generating the final trigger token for the execution of the activity when all the tokens have arrived.

Finally, the work done for the implementation of the soft prototype of the N-computer system gives us a plan to build the prototype system. Although, the partial implementation of the simulation of data distribution schemes has not converged to a final result, it gives the indications that to achieve an efficient execution of a program on a distributed memory machine, the system must support different ways to distribute the objects which can be selected depending upon the nature of the program.

To summarize, this thesis has addressed both important issues of general purpose parallel computing. The major achievement has been the understanding of the marriage of the two issues of general purpose parallel computing, which has enabled advances to be made in dataflow architectures exploiting the collective behavior of data parallel programs.
8.2 Future Work

This thesis presents the first phase of the research done for the design of a data flow bulk synchronous machine (DFBSM) for the execution of data parallel programs via F-code. With the demonstration that we can implement a machine on top of the DFBSM model and presenting the design of such a machine, this thesis opens many ways to proceed with further research. The building of efficient parallel computers should not only be considered as an academic research exercise, it should become a base for a huge industry of future parallel computers. To make this happen, the following research needs to be completed:

1. The soft prototype of the N-computer by implementing stages 7, 8 and 9.

2. Extend the data distribution simulation and devise a system which can generate data distribution directives for an optimum execution of an F-code program on a distributed memory machine.

3. Select a processor for the processing sub-system of the N-computer. This processor must be efficient both in executing macroactors as well as single instructions. Then compile the F-code for the N-computer.

4. Depending upon the results of the previous three steps, refine the design of the N-computer and implement different sub-systems in hardware, e.g., synchronization and output token formation systems.

There is another very important phase of this research which must be done as soon as possible. The base of this research is F-code, and it has been assumed that the F-code is a representation of data parallel high level languages. Although, qualitatively this assumption seems valid, there is a need to demonstrate this. For this, data parallel high level languages especially Fortran 90 must be compiled in F-code.
Part I
Appendices
Appendix A

The N-Computer: Lower Level Description of the Processing Phase

In this appendix, a lower level organization of the N-computer will be described. The functioning of each sub-system will be described in pseudo Occam[70]. To access a field of a multiple fields buffer, the C like syntax is used, e.g., m_field.buffer.field2 will access field2 of the multiple fields buffer (structure). The syntax for Occam input and output is replaced with \(\Rightarrow\) and \(\Leftarrow\) respectively. To improve readability, an END statement is added for constructs used to combine processes, e.g., SEQ, PAR etc. The processes (instructions) written in a sequence in a process without any ordering direct like \(PAR\) of \(SEQ\) will be considered as sequential processes. This description is not a complete system specification with each and every signal and state. It is a higher level system specification described the logic with defined signals. The simplified representation instead of detailed specifications is chosen to demonstrate that the logic will work by just looking onto the specifications. The complete system is not implemented yet.

The structure of an N-computer node as given in Figure A.1 will be considered to describe the working of the system. The main-control system, in addition to the functioning of the sub-systems it has, is responsible to control the functioning of each system of the N-computer's node. First the functioning of the main control system and its signals to the other systems of the node will be described. Then the functioning of each system will be described first for the processing and then for the initialization phase.

A.1 The Main Control System

The main control system will generate the data base for synchronization and output token formation from the activity tree and control the functioning of each system of the N-computer node. For this purpose, it will have six flags, a state machine and three subsystems as shown in figure A.2. The flags are the control links between the main control system and the other systems of an N-computer node. This system has following flags:

1. \(\text{Int}_{\text{control}}\): To control the initialization sub-system.
2. \(\text{RT}_{\text{control}}\): To control the run-time control system.
3. \(\text{GS}_{\text{control}}\): To control the global synchronization sub-system.
4. \( \text{LS}_{\text{control}} \): To control the synchronization system.

5. \( \text{OT}_{\text{control}} \): To control the output token formation system.

6. \( \text{STE}_{\text{control}} \): To control the starting token emitter.

The flag to control the communication interface is not defined, since it is supposed that the communication interface will be working all the time without getting any control signal form the main control system. The same is true with the processing system. The function of the main control system other than the functioning of its sub-systems is as follows:

\[
\text{PROC MCONTROL}
\]

\[
\text{SEQ}
\]

\[
\text{PAR}
\]

\[
\begin{align*}
\text{Int}_{\text{control}} & := \text{SET} & \quad \text{start initialization} \\
\text{RT}_{\text{control}} & := \text{RESET} \\
\text{GS}_{\text{control}} & := \text{RESET} \\
\text{LS}_{\text{control}} & := \text{RESET} \\
\text{OT}_{\text{control}} & := \text{RESET} \\
\text{STE}_{\text{control}} & := \text{SET}
\end{align*}
\]

\[
\text{END PAR}
\]

\[
\text{WHILE START}
\]

\[
\text{SEQ}
\]

\[
\text{WHILE Int}_{\text{control}} = \text{SET} \quad \text{initialization is going on}
\]

181
Figure A.2: Structure of the main control system of the N-computer's node

```plaintext
-END WHILE
PAR
  LS\_control := SET
  OT\_control := SET
  RT\_control := SET
  GS\_control := SET
END PAR
PAR
  WHILE GS\_control = SET  \(-\) processing is going on
    \(-\) wait
END WHILE
  WHILE RT\_control = SET
    \(-\) wait
END WHILE
END PAR
Int\_control := SET  \(-\) start initialization
END SEQ
END WHILE
END SEQ
```
A.2 The Processing Phase

In this section the functioning of each system of the N-computer in the processing phase will be described. The higher level logical description of these system is given in chapter 4.

A.2.1 Communication Interface

The functioning of the communication interface related with the N-computer is described in section 4.6. A lower level description of the system, in the light of Figure A.3, is described here. INBUFFER1 has the same structure as that for TOKEN1, except the processing node descriptor in the tag.

```plaintext
PROC COMM_INTERFACE

WHILE START
  ALT
    TOKEN1 ==> INBUFFER1  -- from network
    IF
      INBUFFER1.TYPE1 = ETS  -- to ETS
      IF
        INBUFFER1.decomp_type = SIMPLE
        TOKEN2 <= INBUFFER1.TOKEN2
      TRUE  -- compound token
      SEQ k=0 FOR INBUFFER1.N
        INBUFFER1.TOKEN2.t := (k * INBUFFER1.I) + INBUFFER1.I
        TOKEN2 <= INBUFFER1.TOKEN2
      END SEQ
```

Figure A.3: Logical structure of the communication interface.
END IF
INBUFFER1.TYPE1 = Proc  -- to processor
IF
INBUFFER1.decomp.type = SIMPLE
COM_Proc <= INBUFFER1.Proc_TOKEN
TRUE  -- compound token
SEQ k=0 FOR INBUFFER1.N
INBUFFER1.ETS_TOKEN.t := (k * INBUFFER1.I) + INBUFFER1.I
COM_Proc <= INBUFFER1.Proc_TOKEN
END SEQ
END IF
INBUFFER1.TYPE = STE  -- could be compound or simple
TOKEN7 <= INBUFFER1.TOKEN7
END IF
TOKEN7 ==> INBUFFER2  -- from STE
IF
INBUFFER2.TYPE1 = ETS
IF
INBUFFER2.decomp.type = SIMPLE
TOKEN2 <= INBUFFER2.TOKEN2
TRUE  -- compound token
SEQ k=0 FOR INBUFFER2.N
INBUFFER2.TOKEN2.t := (k * INBUFFER2.I) + INBUFFER2.I
TOKEN2 <= INBUFFER2.TOKEN2
END SEQ
END IF
INBUFFER2.TYPE1 = Proc
IF
INBUFFER2.decomp.type = SIMPLE
COM_Proc <= INBUFFER2.Proc_TOKEN
TRUE  -- compound token
SEQ k=0 FOR INBUFFER2.N
INBUFFER2.ETS_TOKEN.t := (k * INBUFFER2.I) + INBUFFER2.I
COM_Proc <= INBUFFER2.Proc_TOKEN
END SEQ
END IF
TOKEN6 ==> INBUFFER4  -- from OTF
IF
INBUFFER4.TYPE2 = BROADCAST
TOKEN1 <= INBUFFER4.TOKEN1
IF
INBUFFER4.TYPE1 = ETS
IF
INBUFFER4.decomp.type = SIMPLE
TOKEN2 <= INBUFFER4.TOKEN2
TRUE - compound token
SEQ k=0 FOR INBUFFER4.N
   INBUFFER4.TOKEN2.t := (k * INBUFFER4.I) + INBUFFER4.I
   TOKEN2 <= = INBUFFER4.TOKEN2
END SEQ
END IF
INBUFFER4.TYPE1 = STE
TOKEN7 <= = INBUFFER4.TOKEN7
END IF
TRUE - not broadcast
IF
INBUFFER4.L = LOCAL
IF
INBUFFER4.TYPE1 = ETS
IF
   INBUFFER4.decomp_type = SIMPLE
   TOKEN2 <= = INBUFFER4.TOKEN2
   TRUE - compound token
   SEQ k=0 FOR INBUFFER4.N
      INBUFFER4.TOKEN2.t := (k * INBUFFER4.I) + INBUFFER4.I
      TOKEN2 <= = INBUFFER4.TOKEN2
   END SEQ
   END IF
END IF
INBUFFER4.TYPE1 = STE
TOKEN7 <= = INBUFFER4.TOKEN7
END IF
TRUE - remote token
TOKEN1 <= = INBUFFER4.TOKEN1
END IF
END IF
Proc.COM => INBUFFER3 - from Processing system
IF
INBUFFER3.TYPE2 = BROADCAST
TOKEN1 <= = INBUFFER3.TOKEN1
IF
INBUFFER3.TYPE1 = ETS
IF
   INBUFFER3.decomp_type = SIMPLE
   TOKEN2 <= = INBUFFER3.TOKEN2
   TRUE - compound token
   SEQ k=0 FOR INBUFFER3.N
      INBUFFER3.TOKEN2.t := (k * INBUFFER3.I) + INBUFFER3.I
      TOKEN2 <= = INBUFFER3.TOKEN2
   END SEQ
   END IF
END IF
A.2.2 Synchronization System

The higher level logic for the synchronization system in the processing phase is described in section 4.7. In the light of Figure A.4, the lower level functioning of the system is as follows:

PROC SYNCH
WHILE START
IF
\texttt{LS\_control = SET}  \quad \text{- processing cycle}
SEQ
\texttt{TOKEN2 =\rightarrow INBUFFER}
IF
\texttt{INBUFFER.t.P = EMPTY}
PAR
\texttt{INBUFFER.t.OP := INBUFFER.v}
\texttt{INBUFFER.t.P := FULL}
END PAR
END IF
END ALT
END WHILE
INBUFFER.t.P = FULL
SEQ
PAR
SEQ  – proc token
  IF
    INBUFFER.d = LEFT
    PAR
      OUTBUFFER1.OP\text{\_left} := INBUFFER.v
      OUTBUFFER1.OP\text{\_right} := INBUFFER.t.OP
    END PAR
    INBUFFER.d = RIGHT
    PAR
      OUTBUFFER1.OP\text{\_left} := INBUFFER.t.OP
      OUTBUFFER1.OP\text{\_right} := INBUFFER.v
    END PAR
  END IF
PAR
  OUTBUFFER1.Act := INBUFFER.t.Act
  OUTBUFFER1.Act\text{tag} := Act\text{tag}
END PAR
TOKEN3 <= OUTBUFFER1
END SEQ
IF  – global synch signal
  INBUFFER.t.GSCD\text{\_field1} = SET
  GSCDSg1 <= SIGNAL
  INBUFFER.t.GSCD\text{\_field1} = RESET

Figure A.4: The synchronization system model for the N-computer.
SEQ - OTF token

PAR
  OUTBUFFER2.OT := INBUFFER.t.OT
  OUTBUFFER2.Acttag := Acttag
END PAR
TOKEN4 <= OUTBUFFER2
END SEQ

IF
  INBUFFER.t.Act = BINARY
  INBUFFER.t.p := EMPTY
END IF
END PAR
UPDATE Acttag
END SEQ
END IF
END SEQ

LScontrol = RESET - initialization cycle, for packet type initialization
INBUFFER <= Int_Synch.data_path
PAR i=0 FOR INBUFFER.n
  ETS[i+INBUFFER.S]).Act := INBUFFER.Act
  IF
    INBUFFER.GSCD_field1 = SET
    ETS[i+INBUFFER.S].GSCD_field1 := INBUFFER.GSCD_field1
    GSCUSg1 <= SIGNAL
    TRUE
    ETS[i+INBUFFER.S].GSCD_field1 := INBUFFER.GSCD_field1
  END IF
  ETS[i+OUTBUFFER.S].OT := (INBUFFER.OT) + (i * INBUFFER.Nparents) + 1
END PAR
END IF
END WHILE

In the processing phase, for every GSCDG1 signal the global synchronization generator will count down the GSCR1 register. When the register is zero, it will signal the main control system. During initialization, it will count up the GSCR1 register for every GSCUSg1 signal. The algorithm for the system will be as follows:

PROC GSG1 - Global Synchronization Generator 1

---

1In cases when equal numbers of ETS tokens are going to be matched at each node, it will be possible to know the contents of the GSCR1 at compile time or before initialization starts. In this case, whether it will be useful to add the GSCR1 as a parameter with each activity in the activity tree, is implementation dependent. If the initialization process is implemented in hardware in such a way that it works in parallel to the initialization of ETS, then by adding GSCR1 as a parameter the activity would not bring any fruit.
WHILE START
  IF
    LS_{control} = SET
    PAR
      SEQ
        GSCDSg1 <= SIGNAL – if there is a GSCDSg1 signal
        DECREMENT GSCR1
      END SEQ
      IF
        GSCR1 = 0
        LS_{control} := RESET
        TRUE
        SKIP
      END IF
    END PAR
    LS_{control} = RESET
  SEQ
    GSCUSg1 <= SIGNAL – if there is a GSCUSg1 signal
    INCREMENT GSCR1
  END SEQ
END IF
END WHILE

The control signal $L_S^{control}$ is set by the main control system and reset by the global synchronization generator in the synchronization control system. In this way one physical link will be enough to control the functioning of the synchronization system as well as to inform the global synchronization control system about the completion of the work for the current processing phase by the synchronization system at the node.

A.2.3 Processing System

In the light of Figure 4.11 in section 4.8, the processing system of the N-computer will work as follows:

PROC PROCESSING

WHILE START
  PAR
    SEQ
      TOKEN3 ==> INBUFFER
      PAR
        DISPATCH\_OUT[INBUFFER.ACT].LEFT <= INBUFFER.OP_{left}
        DISPATCH\_OUT[INBUFFER.ACT].RIGHT <= INBUFFER.OP_{right}
        DISPATCH\_OUT[INBUFFER.ACT].Act_{tag} <= INBUFFER.Act_{tag}
    END PAR
  Process_{Act} – trigger the functional unit for its function

189
A.2.4 Output Token formation System

It is assumed that the second section of the output token storage (OTS2) is implemented on a slower memory. While the processing system process activities, the output token formation system load the required tokens from the slower OTS2 to the faster buffers. The organization of the output token formation system is shown in Figure 4.16 and the model of OTS is shown in Figure 4.13. For the following algorithm, two buffers are added in the Figure 4.16, first is for TOKEN4 (INBUFFER1) and second is for TOKEN5 (INBUFFER2). The lower level description of the system functioning is as follows:

PROC OTFS
WHILE START
IF
OT_conrol = SET  - Processing cycle
ALT
TOKEN4 ==> INBUFFER1  - token from the synchronization system
  tag := INBUFFER1.Acttag
  TRUE
  IF -for reduce activity
  OTS1[INBUFFER1.O] > 0 AND OTS3[OTS1[INBUFFER1.O]] > 0
  DECREMENT OTS3[OTS1[INBUFFER1.O]]
  Set token_template to NULL, i.e., token is simple, local and point to point
  TRUE
  buffer.Tag.parents := OTS1[INBUFFER1.O+1]
  SEQ i = 0 FOR OTS1[INBUFFER1.O+1]  - for each parent
  OTS2_PT := OTS1[INBUFFER1.O + 1 + i]
  buffer.Tag.com_type[i] := OTS2[OTS2_PT].com_type
  END SEQ
  SEQ i = 1 FOR M
  Processor[i].OUTBUFFER.Acttag <= Acttag
  Processor[i].OUTBUFFER.DATA :]= Processor[i].result
  MUX.IN[i] <= Processor[i].OUTBUFFER
END PAR
ALT i=0 for M
  MUX.IN[i] <= Processor[i]. OUTBUFFER
  TOKEN5 <= MUX.IN[i]
END ALT
COM_Proc ==> BUFFER1
  Process token according to the information in the token.
  The implementation of direct tokens to the processing system
  is not studied in this thesis.
END PAR
END WHILE
buffer\_tag\_decomp\_type[i] := OTS2[OTS2\_PT].decomp\_type
buffer\_tag\_type[i] := OTS2[OTS2\_PT].type
buffer\_tag\_d[i] := OTS2[OTS2\_PT].d
buffer\_tag\_GSCD\_field[i] := OTS2[OTS2\_PT].GSCD\_field
SEQ j=0 FOR OTS2[OTS2\_PT].n -- for each output token
IF
OTS2[OTS2\_PT].c = RESET
IF
OTS2[OTS2\_PT].decomp\_type[i] = COMPOUND
buffer\_tag\_N[j]:= OTS2[OTS2\_PT].N[j]
buffer\_tag\_I[j]:= OTS2[OTS2\_PT].I[j]
END IF
buffer\_tag\_p[j] := OTS2[OTS2\_PT].p[j]
buffer\_tag\_t[j] := OTS2[OTS2\_PT].t[j]
TRUE
buffer\_tag\_c[i] := SET
IF
OTS2[OTS2\_PT].true\_decomp\_type[i] = COMPOUND
buffer\_tag\_true\_N[j]:= OTS2[OTS2\_PT].true\_N[j]
buffer\_tag\_true\_I[j]:= OTS2[OTS2\_PT].true\_I[j]
END IF
IF
OTS2[OTS2\_PT].false\_decomp\_type[i] = COMPOUND
buffer\_tag\_false\_N[j]:= OTS2[OTS2\_PT].false\_N[j]
buffer\_tag\_false\_I[j]:= OTS2[OTS2\_PT].false\_I[j]
END IF
buffer\_tag\_true\_p[j] := OTS2[OTS2\_PT].true\_p[j]
buffer\_tag\_true\_t[j] := OTS2[OTS2\_PT].true\_t[j]
buffer\_tag\_false\_t[j] := OTS2[OTS2\_PT].false\_t[j]
END IF
END SEQ
END SEQ
END IF
IF
OT[\_tag].p = EMPTY -- result from the processing system has not arrived
OT[\_tag].buffer\_tag\_Data := buffer\_tag
OT[\_tag].p := FULL
TRUE
IF
OTS1[INBUFFER1\_OT] > 0 OR OTS3[OTS1[INBUFFER1\_OT]] > 1
TOKEN6 <= buffer\_tag->.tokens[0]
TRUE
SEQ i = 0 FOR OTS1[INBUFFER1\_OT]
IF

191
buffer\textsubscript{tag}.c[i] = SET

IF

\begin{align*}
&\text{OT}[\text{tag}].\text{buffer}_\text{tag}/\text{Data} = \text{FALSE} \\
&TOKEN6 <= = buffer\textsubscript{tag}.\text{false}.\text{tokens}[i] \\
&TOKEN6 <= = buffer\textsubscript{tag}.\text{true}.\text{tokens}[i] \\
&\text{END IF}
\end{align*}

TRUE

\begin{align*}
&TOKEN6 <= = buffer\textsubscript{tag}->.\text{tokens}[i] \\
&\text{END IF}
\end{align*}

IF

\begin{align*}
&\text{buffer}_\text{tag}.\text{GSCD}_\text{field}[i] = \text{SET} \\
&\text{GSCDSg2} <= =\text{SIGNAL}
\end{align*}

END IF

END SEQ

\begin{align*}
\text{OT}[\text{tag}].p := \text{EMPTY} \\
\text{OT}[\text{address}] <= = \text{tag}
\end{align*}

END IF

END IF

\begin{align*}
&TOKEN5 => = \text{INBUFFER2} \quad \text{-- token from the processing system} \\
\text{tag} := \text{INBUFFER2.}\text{Act}_\text{tag}
\end{align*}

IF

\begin{align*}
&\text{OT}[\text{tag}].p = \text{EMPTY} \\
&\text{OT}[\text{tag}].\text{buffer}_\text{tag}/\text{Data} := \text{Data} \\
&\text{OT}[\text{tag}].p := \text{FULL}
\end{align*}

TRUE

IF

\begin{align*}
&\text{Token}_\text{template} = \text{NULL} \\
&TOKEN6 <= = \text{OT}[\text{tag}].\text{buffer}_\text{tag}->.\text{tokens}[0]
\end{align*}

TRUE

SEQ \text{i} = 0 \text{ FOR } \text{OT}[\text{tag}].\text{buffer}_\text{tag}.\text{parents}

IF

\begin{align*}
&\text{OT}[\text{tag}].\text{buffer}_\text{tag}.c[i] = \text{SET} \\
&\text{IF}
\end{align*}

\begin{align*}
&TOKEN5.\text{Data} = \text{FALSE} \\
&TOKEN6 <= = \text{OT}[\text{tag}].\text{buffer}_\text{tag}.\text{false}.\text{tokens}[i] \\
&TOKEN6 <= = \text{OT}[\text{tag}].\text{buffer}_\text{tag}.\text{true}.\text{tokens}[i] \\
&\text{END IF}
\end{align*}

TRUE

\begin{align*}
&TOKEN6 <= = \text{OT}[\text{tag}].\text{buffer}_\text{tag}->.\text{tokens}[i] \\
&\text{END IF}
\end{align*}

IF

\begin{align*}
&\text{OT}[\text{tag}].\text{buffer}_\text{tag}.\text{GSCD}_\text{field}[i] = \text{SET} \\
&\text{GSCDSg2} <= = \text{SIGNAL}
\end{align*}

192
The global synchronization generator subsystem receives signals, \( GSCUSg2 \) or \( GSCDSg2 \) and accordingly counts up or down the \( GSCR2 \) register. When the register is zero, it will inform the global synchronization control system that it has finished its work. The algorithm for the global synchronization generator sub-system will be as:

**PROC GSG2 — Global Synchronization Generator 2**

**WHILE START**

**IF**

\( OT_{\text{control}} = \text{SET} \)

**PAR**

**SEQ**

\( GSCDSg2 \) \( \leq \) \( SIGNAL \) \quad — if there is a \( GSCDSg2 \) signal

DECREMENT \( GSCR2 \)

**END SEQ**

**IF**

\( GSCR2 = 0 \)

\( OT_{\text{control}} := \text{RESET} \)

\( \text{TRUE} \)

STIFING

**END IF**

**END PAR**

\( OT_{\text{control}} = \text{RESET} \)

**SEQ**

\( GSCUSg2 \) \( \leq \) \( SIGNAL \) \quad — if there is a \( GSCUSg2 \) signal

INCREMENT \( GSCR2 \)

**END SEQ**

**END IF**

**END WHILE**

### A.2.5 Starting Tokens Emitter

The \( STE \) system in the light of Figure 4.17, will work as follows:
PROC STE

WHILE START
  IF
    $STE_{control} = \text{SET}$  
    - STE write cycle
  ALT
    $\text{TOKEN7} \Rightarrow \text{INTOKEN}$
    SEQ
      $\text{STE}[\text{STE.COUNTER}] := \text{INTOKEN}$
      $\text{STE.COUNTER} := \text{STE.COUNTER} + 1$
    END SEQ
    $\text{Int.STE}_{data \_path} \Rightarrow \text{INTOKEN}$
    SEQ
      $\text{STE}[\text{STE.COUNTER}] := \text{INTOKEN}$
      $\text{STE.COUNTER} := \text{STE.COUNTER} + 1$
    END SEQ
  END ALT
  $STE_{control} = \text{RESET}$  
    - STE read cycle
  SEQ
    WHILE $\text{STE.COUNTER} > 0$
      SEQ
        $\text{TOKEN7} \Leftarrow \text{STE}[\text{STE.COUNTER}]$
        $\text{STE.COUNTER} := \text{STE.COUNTER} - 1$
      END SEQ
    END WHILE
  $STE_{control} := \text{SET}$
  END SEQ
  END IF
END WHILE

A.2.6 Global Synchronization Control System

This system in the light of Figure 4.18 will work as follows:

PROC GSC - AT PROCESSOR j

WHILE START
  SEQ
    WHILE $\text{GS}_{control} = \text{RESET}$
    - WAIT
    END WHILE
  PAR
  SEQ
    PAR
      WHILE $\text{LS}_{control} = \text{SET}$
  END PAR
A.2.7 Run-Time Control System

PROC RTCS

WHILE START
SEQ
WHILE RT\_control = RESET  \hspace{1em} \text{– initialization going on}
\hspace{1em} \text{– wait}
END WHILE
READ\_RT\_EXT
COERCE\_ACTIVITY\_TREE
INT\_RT\_PARAMETERS
RT\_control := RESET \hspace{1em} \text{– initialization may start}
END SEQ
END WHILE

The functions \textit{READ\_RT\_EXT} is to load run-time extents for an activity tree. Then the activity tree will be coerced by the function \textit{COERCE\_ACTIVITY\_TREE}. Finally, the run-time parameters for a dynamic shape activity tree will be initialized by the function \textit{INT\_RT\_PARAMETERS}. This function can be evaluated in the processing phase or in the initialization phase. To evaluate this function, the control (RT\_control) will be set at appropriate time either by an activity or by the global synchronization.
Appendix B

The N-Computer: Initializer and ETS Initialization

In this appendix, to initialize ETS and OTS, micro level functioning of the system will be described. The initializer supports the initialization of ETS and OTS separately, independent of each other. The initializer will work as follows:

```
PROC INT (activity_tree)
  WHILE START
    IF
      Int\_{control} := SET
      PAR
        ETS\_INT(activity\_tree)
        OTS\_INT(activity\_tree)
      END PAR
      Int\_{control} := RESET
    END IF
  END WHILE
END PROC
```

To discuss the ETS, STE and OTS initialization, we need to define the structure of an activity in the activity tree. In the following section, the structure of an activity is defined. Then the initialization will be discussed in details.

**B.1 Structure of an Activity (Non-Geometric) for ETS Initialization**

A brief description of an activity tree, i.e., a data structure organized in a tree like fashion with each node representing a collection of threads, has already been given in section 5.4. Here, the logical structure of each activity (non-geometric) for the ETS initialization will be defined. The amount of memory required to store each field in the structure will be highlighted. It is not necessary that each activity in an activity tree will need all that space to store these fields. There are some parameters which could be common for more than
one activities in an activity tree. Also some are simple binary information which needs only one bit for the storage\textsuperscript{1}. An activity, \textit{act}, will have following parameters for the ETS initialization:

1. \textit{act.comp}: It is a light weight descriptor of a computation to be executed at a processing node and the space required for this depends upon the way the computation is represented. If there are \textit{n} different activities in an activity tree, then the activity store would demand \(O(\log_2 n)\) space to store the computation tag. In this case, more than one instance of the same activity in an activity tree will represent the same computation. This way of representing a computation job will be more likely if the system has different functional units.

2. If we have a von-Neumann style conventional processor at each processing node, then each activity is going to be compiled into a code block. If there are \textit{m} activities in an activity tree and there are \textit{m} different code blocks then the computation tag demands \(O(\log_2 m)\) space in the activity store.

In either case, \textit{n} and \textit{m} are not going to be very large. For a processing system of different functional units, the number of units will be known and the space can be fixed accordingly. The parameter \textit{m} can be limited by the machine size for the worst case. It is most unlikely that an activity tree is going to have more than 1000 activities for a data parallel program, since, a single activity in a data parallel program will be for an object. Therefore, we can limit the space in the activity store for a computation tag to 10 bits.

2. \textit{act.type}: It contains the information about the type of an activity such as unary, binary or reduce. The unary activities are of two types, one is simple and the other is operated on index. Similarly, the reduce activity is of two types, one is centralized and the other is distributed. The difference between these is discussed in section 6.4.5 and 6.4.3. To store five piece of information, we need three bits in the activity store.

3. \textit{act.ets.ETS_ptstart}: It contains a pointer to the first location of ETS which will be used for the synchronization of units of current activity at each node. The space required to store this parameter is according to the size of ETS. If \(S_{ets}\) is the number of locations in ETS, then the pointer to an ETS location needs \(\log_2(S_{ets})\) bits.

The pointer to the start of ETS for a parent activity is needed to initialize OTS for its child activity as well, therefore, by keeping the pointer in an area in the activity store and a pointer to that location in the activity tree for both parent and child activities will reduce the space required for this parameter in the activity store. Therefore we can limit the space along the activity for the pointer to the pointer of start of ETS location up to 10 bits\textsuperscript{2}.

4. \textit{act.ets.GSCD_field}: It contains the information as to whether the synchronization of each unit of an activity will contribute towards the generation of global synchronization. Therefore, this parameter is a one bit field in the activity store.

\textsuperscript{1} Although this thesis does not discuss the lower level details for the implementation of the activity store but an idea about the overall requirement of the storage space and most likely organization of it can easily be deduced from this discussion.

\textsuperscript{2} Remember maximum 1000 activities in an activity tree for a single processing phase.
5. *act.ets.type:* There could be more than one way to initialize the ETS. This field contains the information about the way to initialize the ETS for the activity. In the initialization process, there are two ways to initialize ETS\(^3\). Therefore, one bit will be enough for this parameter in the activity store.

6. *act.parents:* It contains the number of parents activities for the current activity. In the light of F-code program, the number of parents are not going to be very large, in general. The number of activities for a single processing phase will be limited, too. Therefore, the number of parents of an activity is going to be very small. A two digit limit on the number of parents looks reasonably good. For this parameter, six bits will be enough to accompany 64 parents. If an activity has more than 64 parents (very unlikely) then that can be evaluated by using global synchronization and global memory. If the output tokens are compiled with the computation of each unit of an activity, then this parameter will have no parent (NULL).

7. *act.dist.N:* This parameter represents the number of units or blocks which will be distributed by the distribution function. The size of this parameter can be fixed according to the machine size. A pointer to a memory location reserved for this parameter could be kept with the activity instead of keeping the parameter itself. The two stage storage of \(N\) parameter for distribution will reduce the amount of space required for an activity at the cost of a delay to access the parameter. If every activity in the activity tree has a different \(N\) parameter, then the space required to store the pointer to these parameters will be 10 bits (In the light of the limit of 1000 activities in an activity tree).

8. *act.dist.L:* This parameter represents the number of elements in each unit in case the distribution unit is a block. The amount of space required for this parameter will be less than that for the first parameter. Like \(N\) parameter in the previous item, the size for this can be limited to 10 bits.

9. *act.dist.P:* This parameter represents the first processing node for the distribution of objects. This parameter is according to the number of processing nodes in the network. The processing nodes could be fixed up-to 1000 for the prototype of the machine, which is a significantly high limit. In this way we need 10 bits to store the first processor parameter.

The space needed to store the above information of an activity is about 64 bits only. To store about 100 activities, we need only 100 words of 64 bits each. This space can easily be provided on chip with the ETS initialization process which will be described in the following section.

### B.2 Initialization of ETS

As described in section 5.5, each location of ETS either can be initialized by the initializer or the initializer send a packet to the synchronization system for more than one ETS location

\(^3\)Which will be discussed in a section about ETS initialization in this chapter.
and the synchronization system will initialize each location independent of the initializer. The initializer also need to know the type of the activity, i.e., binary, unary or reduce. The synchronization system as described in appendix A.2.2 has the required functioning to initialize more than one locations of ETS from a single packet from the initializer. The ETS initializer will work as follows:

ETV_INT(activity_tree) — general initialization

ETV_INT(child_activities)
IF
act.type = REDUCE.DISTRIBUTED
numb := act.dist.N
TRUE
numb := G (Pi, act.ext, N)
END IF
IF
act.ets.type = PACKET
SEQ — ETS initialization
Int.token1.n := numb
Int.token1.act := act.comp
Int.token1.S := act.ETS_ptstart
Int.token1.GSCD_field := act.GSCD_field1
Int.token1.OT := act.OTS_ptstart
Int.token1.N parens := act.No.of.parents
IF
act.type = monadic
Int.token1.OP := act.OP
Int.token1.OP.P := FULL
END IF
Int.synchdata_path <= Int.token
END SEQ
TRUE
SEQ i=0 FOR numb
ETS_CURRENT := act.ETS_ptstart + i
ETS[ETS_CURRENT].act := act.comp
ETS[ETS_CURRENT].GSCD_field := act.GSCD_field1
IF
act.type := MONADIC
ETS[ETS_CURRENT].OP := act.OP
ETS[ETS_CURRENT].P := FULL
END IF
ETS[ETS_CURRENT].OT := act.OTS_ptstart + (i * (act.No.of.parents + 1))
END SEQ
END IF

The function G is described in the next section.
B.2.1 The Function G

To know number of units at a processing node of the N-computer for an activity (act) in an activity tree, for a cyclic regular distribution the following process can be implemented. Suppose, \( N_{\text{processor}} \) is the number of processing nodes in the N-computer and the function is being evaluated at processing node \( P_i \). The distribution parameters of an activity and the logic for the following process is described in section 5.5. The first processing node is numbered as one (1) and the last processing node is numbered as \( N_{\text{processor}} \). A unit could be a single element or it could be a collection of elements. In general, we need a number of elements for ETS initialization. In some situations, we just need a number of units. To know the number of elements, the process is named \( G_{\text{elements}} \), which is as follows:

```plaintext
WHILE START  - at processor \( P_i \)
    INPUT_CHANNEL\( G_{\text{elements}} \) ===> act.dist  - input the distribution parameters
    \( Q, R := \text{act.dist.N} \mod N_{\text{processor}} \)
    IF
        \( R = 0 \)
        OUTPUT_CHANNEL\( G_{\text{elements}} \) <=\( = Q \ast \text{act.dist.L} \)
        TRUE
        IF
            \( \text{act.dist.P} \leq P_i \)
            IF
                \( P_i < \text{act.dist.P} + R \)
                OUTPUT_CHANNEL\( G_{\text{elements}} \) <=\( = (Q + 1) \ast \text{act.dist.L} \)
                TRUE
                OUTPUT_CHANNEL\( G_{\text{elements}} \) <=\( = Q \ast \text{act.dist.L} \)
            END IF
            TRUE
            IF
                \( N_{\text{processor}} < \text{act.dist.P} + R \)
                IF
                    \( P_i (\text{act.dist.P} + R) - N_{\text{processor}} \)
                    OUTPUT_CHANNEL\( G_{\text{elements}} \) <=\( = (Q + 1) \ast \text{act.dist.L} \)
                    TRUE
                    OUTPUT_CHANNEL\( G_{\text{elements}} \) <=\( = Q \ast \text{act.dist.L} \)
                END IF
                TRUE
            END IF
            OUTPUT_CHANNEL\( G_{\text{elements}} \) <=\( = Q \ast \text{act.dist.L} \)
        END IF
    END IF
END WHILE
```

The black box structure of the process is shown in Figure B.1. Sometimes, there is no need to know the number of elements and the system just want to know the number of
Figure B.1: A black box system to support the function \( G \) with its inputs and output.

units\(^4\). For example, the inverse distribution function required the number of units as given in Appendix C.1.13. In this case, there will be no need to perform multiplication between the number of units and the size of each unit. The function is named \( G_{\text{units}} \) and its description is as follows:

\[
\text{WHILE START \, - \, at processor } P_i \\
\text{INPUT_CHANNEL}_{G_{\text{units}}} \implies \text{act.dist} \, - \, \text{input the distribution parameters} \\
Q, R := \text{act.dist.N \%\% } N_{\text{processor}} \\
\text{IF} \\
R = 0 \\
\text{OUTPUT_CHANNEL}_{G_{\text{units}}} \leq Q \\
\text{TRUE} \\
\text{IF} \\
\text{act.dist.P} \leq P_i \\
\text{IF} \\
P_i < \text{act.dist.P + R} \\
\text{OUTPUT_CHANNEL}_{G_{\text{units}}} \leq (Q + 1) \\
\text{TRUE} \\
\text{OUTPUT_CHANNEL}_{G_{\text{units}}} \leq Q \\
\text{END IF} \\
\text{TRUE} \\
\text{IF} \\
N_{\text{processor}} < \text{act.dist.P + R} \\
\text{IF} \\
P_i \leq (\text{act.dist.P + R}) - N_{\text{processor}} \\
\text{OUTPUT_CHANNEL}_{G_{\text{units}}} \leq (Q + 1) \\
\text{TRUE} \\
\text{OUTPUT_CHANNEL}_{G_{\text{units}}} \leq Q \\
\text{END IF} \\
\text{TRUE} \\
\text{OUTPUT_CHANNEL}_{G_{\text{units}}} \leq Q \\
\text{END IF} \\
\text{END IF} \\
\]

\(^4\)Remember, a unit may contain more than one elements.
END IF
END WHILE
Appendix C

The N-Computer; OTS Initialization (General Implementation)

In the following appendices, the processes to implement the functions required to support OTS initialization, e.g., distribution function, inverse distribution function, the function INTP_GEM and GET_DISP OTS and GET_DISP ETS_{parent}, will be described.

C.1 The Function INTP_GEM

This function is the interpretation of the rank/length coercion and geometric activities of F-code. All the information required for this function are kept in the geometric activities in the activity tree. There will be following different geometric activities.

1. \textit{gem.act.rl}: Rank/length coercion.
2. \textit{gem.act.sect}: The sect function.
3. \textit{gem.act.slice}: The slice function.
4. \textit{gem.act.repl}: The repl function.
5. \textit{gem.act.comp}: The comp function.
7. \textit{gem.act.transp}: The transp function.

To interpret the above activities to generate parent multi-indices from a child multi-index, each activity requires few more parameters. These parameters and the interpretation of each activity is discussed in the following sections.
C.1.1 Rank/Length Coercion

The macro level logic for the evaluation of rank/length coercion is defined in section 5.10.1. In this appendix, the implementation of the logic will be given. For the hardware implementation, it is assumed that the object are of six dimensional or less and accordingly the processes are described. If a program has object with more than six dimension, then either that object will be broken down to two or more than two objects or the slower memory can be used to support the initialization. In the light of the limit of six dimensional implementation on hardware, the storage requirement for the three parameters required for the rank/length coercion is as follows:

1. \textit{gem.act.rl.E}: This is the shape of the parent activity. The parent activity already has this parameter which is a vector of up-to six elements. Therefore, instead of copying the same information, this field will have a pointer to the shape of the parent activity.

2. \textit{gem.act.rl.m}: A binary mask of up-to six elements.

3. \textit{gem.act.rl.n}: An integer which represents the number of parent data elements which will consume a child data element. This parameter can be termed as a rank coercion factor.

With these three parameters and the child multi-index, the process to generate all the parent multi-indices can be considered as the generation of a virtual matrix from the child multi-index (vector), as shown in Figure C.1. The following implementation generate the matrix row wise by counting the elements upon which the child multi-index is not mapped, i.e., dimension 2, 3 and 6 in Figure C.1. The process black box is shown in Figure C.2. It

![Diagram](https://via.placeholder.com/150)
Figure C.2: A black box system to support rank/length coercion with its inputs and output.

is assumed that CHILD_INDEX, PARENT SHAPE and MASK are arrays of six elements (r=6). If the object is not six-dimensional, then the elements of CHILD_INDEX which correspond to the non-existing dimensions will be 0. The elements of PARENT SHAPE for the non-existing dimensions will be 1. The elements of MASK for the non-existing dimensions will be 0. The logic for the following process is described in section 5.10.1.

WHILE START
  INPUT CHANNEL ri = > CHILD_INDEX
  k := 0
  COUNT DONE FLAG := NO
  SEQ i=0 FOR gem act rl n
  j := 0
  WHILE gem act rl E[j] > 1
       IF
           i = 0
           IF
               gem act rl m[j] = TRUE
               IF
                   CHILD INDEX[k] < gem act rl E[j]
                   PARENT_INDEX[j] := CHILD_INDEX[k]
                   INCREMENT k
                   TRUE
                   OUTPUT CHANNEL ri <= = NULL
                   EXIT
               END IF
               TRUE
               PARENT_INDEX[j] := 0
           END IF
           TRUE
           IF
               gem act rl m[j] = FALSE AND COUNT DONE FLAG = NO
               PARENT_INDEX[j] := PARENT_INDEX[j] + 1
           END IF
Figure C.3: A black box structure of the process to support sect function with its input and output.

\[
\text{IF} \quad \text{PARENT_INDEX}[j] > \text{gem.act.rl.E}[j] \\
\quad \text{PARENT_INDEX}[j] := \text{PARENT_INDEX}[j] - \text{gem.act.rl.E}[j] \\
\quad \text{TRUE} \\
\quad \text{COUNT_DONE_FLAG} = \text{YES} \\
\quad \text{END IF} \\
\quad \text{TRUE} \\
\quad \text{SKIP} \\
\quad \text{END IF} \\
\quad \text{END IF} \\
\quad \text{INCREMENT} j \\
\quad \text{END WHILE} \\
\quad \text{OUTPUT_CHANNEL}_{rl} <= \text{PARENT_INDEX} \\
\quad \text{COUNT_DONE_FLAG} := \text{NO} \\
\quad \text{END SEQ} \\
\quad \text{END WHILE}
\]

C.1.2 The Function Sect

It is assumed that the system knows the sect parameters \((d, L, r)\) as defined in section 5.10.2. These parameters are integer scalars. The black box structure of the process is shown in Figure C.3. The function \(PICK1\) for the implementation of the sect as defined in section 5.10.2 is as follows:

\[
\text{WHILE START} \\
\quad \text{INPUT_CHANNEL}_{sect} \rightarrow \text{CHILD_INDEX} \quad - \text{input} \\
\quad j := 0 \\
\quad \text{IF} \\
\quad \quad \text{CHILD_INDEX}[d] = L \\
\quad \quad \text{SEQ} \ i = 0 \text{ FOR} \ r
\]

206
Figure C.4: A black box structure of the process to support slice function with its input and output.

IF
i = d
SKIP
TRUE
PARENT INDEX[j] := CHILD INDEX[i]
INCREMENT j
END IF
END SEQ
OUTPUT CHANNEL slice <= PARENT INDEX
TRUE
OUTPUT CHANNEL slice <= NULL
END IF
END WHILE

C.1.3 The Function SLICE

There are three different ways to perform the slice function on a child multi-index as discussed in section 5.10.2. For the implementation of the function PICK2 to evaluate the slice of type 1 the process is as follows:

WHILE START
   INPUT CHANNEL slice1 => CHILD INDEX  - input
   i := 0
   FLAG1 := SET
   WHILE i < r AND FLAG1 = SET
      IF
         CHILD INDEX[d] < V[i]
         OUTPUT CHANNEL slice1 <= NULL
         EXIT
      TRUE
      IF

207
The above implementation is the most general one and it is possible to optimize the implementation of the slice function by detecting regular pattern as discussed in section 5.10.3. The process to support the optimized slice function is described in the next section.

### C.1.4 Optimized Slice Function for Regular Pattern

In the optimized evaluation of slice function, the components being selected along the dimension of the selection make a regular pattern, starting from a position $S$ and ending at a position $L$. The distance between any two contiguous selected components will be $I$. It is assumed that the system knows these three parameters. The black box structure of the process is shown in Figure C.5. As compare to the process of Figure C.4, it does not need a vector whose length can grow according to the size of the dimension of the slice, rather
then it needs three scalars. The following process will implement the \textit{PICK3} function for the evaluation of the slice of type 2 as described in section 5.10.3.

\begin{verbatim}
WHILE START  
  INPUT\_CHANNEL\_slice2 ===> CHILD\_INDEX  - input  
    IF
      CHILD\_INDEX[d] \geq S  
        IF
          CHILD\_INDEX[d] \leq L  
            Q, R := (CHILD\_INDEX[d] - S) \%\% I  
            IF
              R = 0  
                i := 0  
                WHILE i < r  
                  IF
                    i = d  
                      PARENT\_INDEX[i] := Q  
                      TRUE
                    PARENT\_INDEX[i] := CHILD\_INDEX[i]  
                    END IF
                END WHILE  
            END IF  
        END IF  
    END IF  
    OUTPUT\_CHANNEL\_slice2 <== PARENT\_INDEX  
    TRUE
    OUTPUT\_CHANNEL\_slice2 <== NULL  
    END IF  
    TRUE
    OUTPUT\_CHANNEL\_slice2 <== NULL  
    END IF  
    TRUE
    OUTPUT\_CHANNEL\_slice2 <== NULL  
    END IF
END WHILE
\end{verbatim}

\subsection{C.1.5 Optimized Slice Function for Irregular Pattern With Mask}

This process supports the second optimization for the implementation of the slice function as described in section 5.10.3. It is assumed that the system knows about the mask (M) and the dimension (d) for the function \textit{PICK4}. The black box structure of the system is shown in Figure C.6.

\begin{verbatim}
WHILE START  
  INPUT\_CHANNEL\_slice3 ===> CHILD\_INDEX  - input  
    i := 0  
    IF
      M[CHILD\_INDEX[d]] = 1
\end{verbatim}

209
C.1.6 The Function Repl

The logic for the evaluation of the function repl is as follows:

1. If the parent object has the same number of dimensions as of the child object, then the first multi-index of the parent object will be the same as the child multi-index. Otherwise, for the dimensions not in the child object, the first parent multi-index will
have zero (0) values. Make the first parent index according to the dimensions of the parent object and output.

2. After generating the first parent multi-index, find the next multi-index by adding the replication factor for each dimension in the previous multi-index. Similarly, find all the multi-indices of the parent object.

The first item of the above logic is to make the first parent multi-index and then the second item will perform the count function as described in section 5.10.4. The black box structure of the process is shown in Figure C.7 and its description in pseudo Occam is as follows:

```
WHILE START
  INPUT_CHANNEL_{repl} ==> CHILD_INDEX
  k := 0
  COUNT_DONE_FLAG := NO
  SEQ i=0 FOR gem_act.repl.n
    j := 0
    WHILE gem_act.repl.E[j] > 1
      IF
        i = 0
        IF
          gem_act.repl.m[j] = TRUE
          PARENT_INDEX[j] := CHILD_INDEX[k]
          INCREMENT k
          TRUE
          PARENT_INDEX[j] := 0
        END IF
        TRUE
        IF
          gem_act.repl.D[j] != 0 AND COUNT_DONE_FLAG = NO
        END IF
        PARENT_INDEX[j] >= gem_act.repl.E[j]
```
C.1.7 The Function Comp

The black box structure of the process is shown in Figure C.8. The parameter $E$ (gem_act.com.E) is needed for the length coercion. If the child index is going to be displaced along $i$th dimension, the parameter $D[i]$ (gem_act.comp.DISP) will have the displacement otherwise it will be 0. For all non-existing dimensions the value of the corresponding elements will be 0. The implementation of the function $T$ as described in section 5.10.5 is as follows:

```plaintext
WHILE START
    INPUT_CHANNEL_{comp} ==> CHILD_INDEX
    i := 0
    WHILE gem_act.comp.E[i] > 0
        IF CHILD_INDEX[i] < gem_act.comp.E[i]
            PARENT_INDEX[i] := CHILD_INDEX[i] + gem_act.comp.D[i]
            TRUE
            OUTPUT_CHANNEL_{comp} <= NULL
        END IF
    END WHILE
END SEQ
```
C.1.8 The Function Comp (Length Coercion)

If the objects are composed along the last dimension, then for the left child of the function comp there will be no need to perform the above process. In this case, we just need the function $L$ as described in section 5.10.5. In the black box configuration, this process does not need the parameter $DISP$ (gem_act.comp.DISP) as shown in Figure C.8. In that case, the function INTP_GEM will evaluate the following process.

```plaintext
WHILE START
    INPUT_CHANNEL_i ==> CHILD_INDEX
    i := 0
    WHILE gem_act.l.E[i] > 0
        IF
            CHILD_INDEX[i] >= gem_act.l.E[i]
            OUTPUT_CHANNEL_i <= NULL
            EXIT
        END IF
        INCREMENT i
    END WHILE
    OUTPUT_CHANNEL_i <= CHILD_INDEX
END WHILE
```

Figure C.9: A black box system to support the function transp of P-code.
C.1.9 The Function Transp

The black box structure of the process is shown in Figure C.9. It is assumed that the system knows the dimension of the transpose (n). The implementation of the function $T_2$ as described in section 5.10.6 is as follows:

\[
\text{WHILE START}
\]
\[
\text{INPUT\_CHANNEL}_{\text{transp}} \rightarrow \text{CHILD\_INDEX}
\]
\[
j := 0
\]
\[
\text{SEQ} \ i = 0 \ \text{FOR} \ \text{gem\_act\_transp}\_r
\]
\[
\text{IF}
\]
\[
i = \text{gem\_act\_transp}\_n
\]
\[
\text{PARENT\_INDEX}[\text{gem\_act\_transp}\_r-1] := \text{CHILD\_INDEX}[i]
\]
\[
\text{TRUE}
\]
\[
\text{PARENT\_INDEX}[j] := \text{CHILD\_INDEX}[i]
\]
\[
\text{INCREMENT} \ j
\]
\[
\text{END IF}
\]
\[
\text{END SEQ}
\]
\[
\text{OUTPUT\_CHANNEL}_{\text{transp}} \leftarrow \text{PARENT\_INDEX}
\]
\[
\text{END WHILE}
\]

C.1.10 The Function Diag

The black box structure of the process is shown in Figure C.10. The implementation of the function $D$ as described in section 5.10.7 is as follows:

\[
\text{WHILE START}
\]
\[
\text{INPUT\_CHANNEL}_{\text{diag}} \rightarrow \text{CHILD\_INDEX}
\]
\[
\text{DIAG\_FLAG} := \text{YES}
\]
\[
j := 0
\]
\[
\text{SEQ} \ i = 0 \ \text{FOR} \ \text{gem\_act\_diag}\_r
\]
\[
\text{IF}
\]
\[
\text{gem\_act\_diag}\_m[i] = \text{TRUE}
\]
\[
\text{gem\_act\_diag}\_m[i] := \text{TRUE}
\]

Figure C.10: A black box system to support the function diag of F-code.
IF
DIAG_FLAG = YES
LAST := CHILD_INDEX[i]
DIAG_FLAG := NO
TRUE
IF
LAST = CHILD_INDEX[i]
SKIP
TRUE
OUTPUT_CHANNEL dia <= NULL
EXIT
END IF
TRUE
PARENT_INDEX[j] := CHILD_INDEX[i]
INCREMENT j
END IF
END SEQ
PARENT_INDEX[j] := LAST
OUTPUT_CHANNEL dia <= PARENT_INDEX
END WHILE

C.1.11 The Function Pack

The black box structure of the process is shown in Figure C.11. The implementation of the logic as described in section 5.10.8 is as follows:

WHILE START
INPUT_CHANNEL pack ==> CHILD_INDEX
index := 0
ALL_PREV_EXT := 1
i := 0
WHILE gem_act.pack.E_child[i] > 0
index := index + CHILD_INDEX[i] * ALL_PREV_EXT
ALL_PREV_EXT := ALL_PREV_EXT * gem_act.pack.E_child[i]
INCREMENT i
END WHILE
i := 0
WHILE gem_act.pack.E_parent[i] > 0
Q, R := index % gem_act.pack.E_parent[i]
PARENT_INDEX[i] := R
index := Q
INCREMENT i
END WHILE
OUTPUT_CHANNEL pack <= PARENT_INDEX
END WHILE
Figure C.11: A black box system to support the function pack of F-code.

C.1.12 The Distribution Function

The distribution function will return a processing node for a given child multi-index. It is assumed that the process knows, the shape of the object (act.ext), the starting processing node (act.dist.P) and the mask (act.dist.MASK). The system is shown in Figure C.12. The logic for the following process is described in section 5.8.

```
WHILE START
  INPUT_CHANNEL => multi_index
  i := 0
  factor := 1
  FLAG1 := RESET
  WHILE act.ext[i] > 0
    IF act.dist.MASK[i] = 1
      IF FLAG1 = RESET
        single_index := multi_index[i]
        FLAG1 := SET
      TRUE
        single_index := single_index + (factor * multi_index[i])
      END IF
      factor := factor * act.ext[i]
    END IF
    INCREMENT i
  END WHILE
  single_index := single_index + 1  --multi-index starts from 0
R := single_index % N_{processor}  -- yields the remainder
IF
  R = 0
```
C.1.13 The Inverse of Distribution Function

This function returns all the multi-index of an object of shape (act.ext) at a processing node (P_i). The object is distributed with the distribution function as described in the previous section. To find out all the elements at a given processing node, the following logic for the implementation of the function as described in section 5.9, is given. The black box structure of the system is shown in Figure C.13.

1. Starting from a given value of the first dimension which is mapped, count the multi-index modulo the size of each mapped dimension.

```
IF
   act.dist.P = 1
   P := N_{processor}
   TRUE
   P := act.dist.P - 1
END IF
TRUE
IF
   act.dist.P = 1
   P := R
   TRUE
   P := R + act.dist.P
   IF
      P > N_{processor}
      P := P - N_{processor}
   END IF
END IF
END WHILE
```
2. For each value of the multi-index for the mapped dimensions as obtained in the first step, generate all the values of the non-mapped dimensions.

WHILE START
    INPUT_CHANNEL\_F\_INV \(\Rightarrow\) act.ext
    IF
        \(P_i > \text{act.dist.P}\)
        START := \(P_i - \text{act.dist.P}\)
        TRUE
        START := \(P_i + (N_{\text{processor}} - \text{act.dist.P})\)
    END IF
    FIRST\_MAPPED\_DIM := -1
    OVERFLOW := NO
    i := 0
    INPUT_CHANNEL\_G\_units <= act.dist
    OUTPUT_CHANNEL\_G\_units \(\Rightarrow\) LOCAL\_UNITS
    WHILE i < LOCAL\_UNITS -- loop for all the units
    j := 0
    WHILE act.ext[j] > 0 -- loop for mapped dimensions
        IF
            \(\text{act.dist.MASK}[j] = \text{TRUE}\)
            IF
                i = 0
                IF
                    FIRST\_MAPPED\_DIM < 0
                    FIRST\_MAPPED\_DIM := j
                    DATA[j] := START
                    TRUE
                    DATA[j] := 0
                END IF
            TRUE
            IF
        END IF
    END WHILE

Figure C.13: A black box structure of the inverse distribution function.
FIRST_MAPPED_DIM = j
DATA[j] := DATA[j] + N_{processor}
IF
DATA[j] < act.ext[j]
OVERFLOW := YES
END IF
TRUE
IF
OVERFLOW = YES
INCREMENT DATA[j]
IF
DATA[j] < act.ext[j]
OVERFLOW = NO
TRUE
OVERFLOW := YES
END IF
END IF
END IF
END IF
END WHILE
FIRST_LOCAL_DIM := -1
SEQ j = 0 FOR act.dist.L  loop for all the local elements within the unit
k := 0
WHILE act.ext[k] > 0  loop for local dimensions
IF
act.dist.MASK[k] = FALSE
IF
j = 0
IF
FIRST_LOCAL_DIM < 0
FIRST_LOCAL_DIM := k
END IF
DATA[k] := 0
TRUE
IF
k = FIRST_LOCAL_DIM
DATA[k] := DATA[k] + 1
IF
DATA[k] ≥ act.ext[k]
DATA[k] := DATA[k] - act.ext[k]
CARRY_FLAG := SET
END IF
TRUE
IF
CARRY_FLAG = SET
DATA[k] := DATA[k] + 1
IF
DATA[k] ≥ act.ext[k]
DATA[k] := DATA[k] - act.ext[k]
CARRY_FLAG := SET
TRUE
CARRY_FLAG := RESET
END IF
TRUE
SKIP
END IF
TRUE
SKIP
END IF
END WHILE
OUTPUT_CHANNEL_pJW <== DATA - output
OUTPUT_CHANNEL_pINV i <= j + (i * act.dist.L) - output
END SEQ
j := 0
INCREMENT i
END WHILE
END WHILE

The two WHILE loops, one for the mapped dimension and the other for the local dimensions, can be integrated into one WHILE loop. But to understand the logic easily, these are written as two separate loops.

C.1.14 The Function GET_DISP_OTS

For the ith element of an activity, this process just add weighted i to OTS_pt_start. The weight is calculated according to the number of parents of the activity. Since, in the OTS initialization process this function is just after the inverse distribution function (F_INVERSE), a simple addition in the inverse distribution function can return the number of the current multi-index. In this way we can eliminate the need to implement the similar logic again in this function. To evaluate i for the GET_DISP_OTS function with the inverse distribution function we just need to add a counter which will be incremented each time the the multi-index is generated.
Appendix D

The N-Computer; OTS Initialization and Optimizations

D.1 Structure of an Activity (Non-Geometric) for OTS Initialization

To initialize the OTS, the following information are required in the activity.

1. `act.ot.token.class`: It contains information whether the output tokens have been compiled with the code for the computation for each activity. If the class of output tokens is run-time, then the activity will contain the information from item 2 up to the item 14 to support run-time initialization of OTS. A one bit field is required for this parameter in the activity store.

2. `act.ot.token.GSCDfield`: It contains information whether the generation of output tokens for current parent activity will contribute towards the generation of global synchronization. A one bit field will be enough for this parameter in the activity store.

3. `act.ot.token.c`: It contains the information whether the generation of output tokens is going to be a choice between more than one token (true or false) at run-time. The space required to store this field is one bit, too.

4. `act.ot.token.OTS.ptstart`: It contains a pointer to the first location of OTS1 which will be used to store the information about the output tokens for current activity. The space needed for this parameter is according to the size of OTS1. If \(S_{ots1}\) is the number of locations in OTS1, then the pointer to the ETS location needs \(\log_2(S_{ots1})\) bits. The pointer to the OTS1 can also be kept in an area of activity store. This parameter may require field of upto 10 bits to keep the pointer to that location, just like for the pointer to the ETS location.

5. `act.ot.token.T`: If every unit of an activity is going to generate the same number of output tokens, then this parameter will be set. Otherwise it will be reset. This parameter required one bit field in the activity store.

6. `act.ot.token.type`: It contains the information whether the output tokens are of type ETS or STE and one bit will be enough to store this parameter.
7. **act.ot.token.com.type**: It contains information about the communication type of the output tokens, i.e., whether point-to-point or broadcast. We need one bit for this, as well.

8. **act.ot.token.decomp.type**: It contains information about the decomposition type of the output tokens, i.e., whether the token is the simple or compound. One bit will be enough to store this field in the activity store.

9. **act.ot.token.I**: In case of a compound output token, this is an increment parameter. Depending upon the distribution function and the nature of the program, the increment parameter could be more than one. In general, the output tokens are going to be consumed by the parent activity. All the local units of a parent activity occupy the consecutive locations in ETS, therefore, if a compound token is going to be for a single parent activity, then the increment is not going to be large. If a compound token is for more than one parent activities then the increment could be in the order of the size of the local units of a parent activity. If the number of local units of an activity is more than the limit of the increment parameter, we can always generate the simple tokens from its child activity for two parents (instead of a compound tokens for a virtual one parent for the output tokens).

For a compound token, there must be at least two parent units at one processing node which are going to consume the token. It is, probably, never going to be that an activity has a result for two units at a processing node which are going to be 1K or more apart from each other. It seems as the 10 bits limits on I will be more than enough for most of the applications.

10. **act.ot.token.N**: In case of a compound output token, this is a number parameter. The size of this parameter is limited according to the size of ETS. It is not allowed that a compound token will be broken down into simple tokens which exceed the size of ETS in one processing cycle. If the size of ETS is 16K, then the limit on E would be 14 bits.

11. **act.ot.token.diff**: It contains information that whether the output tokens are of constant-difference type or not. If this parameter is more than 0, then it will contain the difference for the output tokens. The difference (diff) parameter can be divided into two sub-fields, one is logical one bit and the other is a number. The size of the number could be limited according to the size of ETS but the discussion about the size of the increment parameter in item 9 is valid for this as well. Thus we can limit the space to 10 bits for this parameter in the activity storage.

12. **act.ot.token.d**: It contains information about the left or right arc of the parent activity for which the tokens will be propagated. One bit will be enough to store this field in the activity store.

13. **act.par.act**: It contains links to the parent activities of the current activity which will consume its result. This field is needed to access the parameters of parents activities while initializing the output tokens for a child activity. The parent's parameters needed are the shape and the pointer to the start of ETS location. Therefore, this could be a pointer to the field in the activity store which stores these parameters.
14. *act.gem_par*: It contains link to the geometric activity which will be used to convert the index of each unit of an activity to its corresponding parent's units for each parent activity. This is a pointer to a space which contains information about the geometric operations involved. The space for this field can be limited according to the limitations of the first field (act.comp), which is 10 bits.

15. *act.others*: There are some other information needed along an activity which will be discussed in future sections.

If an activity has more than one parent activity, then the information for the output tokens (starting from number 2 and ending at number 14) for every parent activity may not be the same. In this case, the activity store will store the information which are not same for all the parent. Here, to explain the logical structure of the activity, we assume that the output tokens information (starting from number 2 and ending at number 14) will be according to the number of parent activities of an activity. Almost all the parameters are self-explanatory. The need for each parameter will become clear in the discussion of the initialization process in the next sections.

### D.2 Initialization of OTS

As discussed in section 5.7, there are eight different template combinations for an output token and accordingly the process for OTS initialization is different for each combination. The overall process to initialize OTS in the light of the functions defined in section 5.7 is as follows:

```
OTS_INT(activity_tree)

OTS_INT(child_activities)
INDICES := F_INVERSE (Pi, act.ext, N)
OTS_CURRENT := act.OTS.pt_start
OTS1[OTS_CURRENT].N_parent := act.parents
SEQ i=0 FOR INDICES.numb
SEQ k=0 FOR act.parents   - for all the parents
   IF
      act.ot_token.diff[k] = non-constant
     IF
        act.ot_token.com_type[k] = Pt_T_Pt
       IF
          act.ot_token.decomp_type[k] = SIMPLE
          OTS_INT.SPNI()
         TRUE   - compound
         OTS_INT.CPN()
       END IF
      TRUE   - broadcast
     IF
        act.ot_token.decomp_type[k] = SIMPLE
```

223
OTS_INT_SBN()
TRUE - compound
OTS_INT_CBN()
END IF
END IF
TRUE - constant
IF
act.ot_token.com_type[k] = Pt.T.Pt
IF
act.ot_token.decomp_type[k] = SIMPLE
OTS_INT_SPC()
TRUE - compound
OTS_INT_CPC()
END IF
TRUE - broadcast
IF
act.ot_token.decomp_type[k] = SIMPLE
OTS_INT_SBC()
TRUE - compound
OTS_INT_CBC()
END IF
END IF
END IF
END SEQ
END SEQ

The functions OTS_INT... will be defined in the next sections. The function F_INVERSE, INTP_GEM, F, GET_DISP_ETS and other as used in the OTS initialization are discussed in Appendix C.

D.2.1 OTS Initialization Scheme 1 (SPN)

This scheme is for the simple, point-to-point and non-constant type output tokens. In this scheme, it is assumed that the output tokens for each unit of an activity does not have a regular pattern and each token will not be broadcast to the network. This is the most general form of OTS initialization. For this type of output tokens, the macro-level OTS initialization is given below.

1. For an activity, find all the units which will be processed at the same processor where OTS is being initialized (the function F^{-1}). This processor is named as local processor, P_{l}.

2. Find the starting location of OTS1 from OTS_ptstart and initialize with the number of parents (act.parents).

3. For each parent activity do as follows:
4. Get the pointer to the current empty location in OTS2, initialize the current location in OTS1 with the pointer. Initialize the template for the output tokens for current parent’s activity in OTS2.

5. If the flag c (act.ot_token.c) of the activity is set, then for each unit found in step 1;
   
   (a) Find all the units of current true parent’s activity (using function INTP.GEM) one by one.
   
   (b) For each unit of current parent found in the previous step;
      i. Find processor (using function F) and the tag (using function GET_DISP.ETS).
      ii. Initialize the processor and the tag field in the OTS2.
   
   (c) Find all the units of current false parent’s activity (the function INTP.GEM) one by one.
   
   (d) For each unit of current parent found in the previous step;
      i. Find processor (using function F) and the tag (using function GET_DISP.ETS).
      ii. Initialize the processor and the tag field in the OTS2.

6. If the flag c (act.ot_token.c) of the activity is reset, then for each unit found in step 1;

   (a) Find all the units of current parent’s activity (the function INTP.GEM) one by one.
   
   (b) For each unit of current parent found in the previous step;
      i. Find processor (by using function F) and the tag (using function GET.DISP.ETS).
      ii. Initialize the processor and the tag field in the OTS2.

The lower level functioning of the system with defined parameters and the functions will be as follows:

- OTS_INT1() — Simple, Pt-T-Pt, non-constant-difference

INDICES := F(Pi, act.ext, N)
OTS_CURRENT := act.OTS_ptstart
OTS1[OTS_CURRENT].N_parent := act.No_of_parents
SEQ i=0 FOR INDICES.numb
SEQ k=0 FOR act.No.of_parents — for all the parents

OTS_INT_SPN {

    PARENT := INTP.GEM (act.gem_parent[k], INDICES.ARRAY[i])
    IF
      PARENT.numb > 0
      OTS2_PT.n := PARENT.numb
      OTS2_PT.c := act.c[k]
      OTS2_PT.GSCD_field := act.GSCD_field[k]
OTS2_PT.type := act.ot_token.type[k]
OTS2_PT.com_type := act.ot_token.com_type[k]
OTS2_PT.decomp_type := act.ot_token.decomp_type[k]
OTS2_PT.d := act.ot_token.d
SEQ j=0 FOR PARENT.numb
IF
act.c[k] = SET  - The choice flag is set
parent_processor := F(PARENT.ARRAY[j], act.parent_true[k].ext, N)
OTS2_PT.true.p[j] := parent_processor
OTS2_PT.true.t[j] := act.parent_true[k].ETS_ptstart +
    GET_DISP_ETS(act.parent_true[k], PARENT.ARRAY[j], parent_processor)
parent_processor := F(PARENT.ARRAY[j], act.parent_false[k].ext, N)
OTS2_PT.false.p[j] := parent_processor
OTS2_PT.false.t[j] := act.parent_false[k].ETS_ptstart +
    GET_DISP_ETS(act.parent_false[k], PARENT.ARRAY[j], parent_processor)
act.c[k] = RESET  - no choice, single parent
parent_processor := F(PARENT.ARRAY[j], act.parent[k].ext, N)
OTS2_PT.p[j] := parent_processor
OTS2_PT.t[j] := act.parent[k].ETS_ptstart +
    GET_DISP_ETS(act.parent[k], PARENT.ARRAY[j], parent_processor)
END IF
END SEQ
OTS1[OTS_CURRENT].parent[k] := OTS2_PT
UPDATE (OTS2_PT)
TRUE  - No output token
OTS1[OTS_CURRENT].parent[k] := NULL
END IF

}
END SEQ
END SEQ

Within the OTS.INT1(), another function named OTS.INT.SPAN() is defined which perform the functioning of the code it enclosed. The parameters outside this function are global. This function is called in the complete initializer in Appendix D.2.

The PARENT is a structure which contains the number of data elements (PARENT.numb) and an array of indices of the parent’s elements which will consume the current data element (INDICES.ARRAY[i]). The argument INDICES is a structure which contains the number of data elements at processor Pi (INDICES.numb) and the indices for all the data elements (INDICES.ARRAY[i]). The function UPDATE() will update the OTS2_PT in such a way that it will point to the next empty location in the OTS2. This process can be implemented along with the initialization of each field of the OTS. In this case the OTS_PT will be incremented after each initialization.
D.2.2 OTS Initialization Scheme 2 (SPC)

For the OTS initialization of the constant-difference activity, as described in section 5.11.1, there will be no need to evaluate the tag by using GET_DISP_ETS function as in Scheme 1 described in the previous section. For this type of output tokens, the initializer for step 5(b)i of scheme 1 will be as follows:

5(b)i. Find the processor (using function F) and evaluate the tag by adding
act.ot.token.diff for true parent to the ETS location of the current unit of the child activity.

The step 5(d)i of scheme 1 will be as follows:

5(d)i. Find the processor (using function F) and evaluate the tag by adding
act.ot.token.diff for false parent to the ETS location of the current unit of the child activity.

The step 6(a) of scheme 1 will be as follows:

5(b)i. Find the processor (using function F) and evaluate the tag by adding
act.ot.token.diff for the parent to the ETS location of the current unit of the child activity.

The micro level description of the process to support the OTS initialization is as follows:

OTS.INT2() - Simple, Pt-T-Pt, constant-difference

INDICES := F.INVERSE (Pi, act.ext, N)
OTS.CURRENT := act.OTS.pt_start
OTS[OTS.CURRENT].N_parent := act.No_of_parents
SEQ i=0 FOR INDICES.numb
SEQ k=0 FOR act.No_of_parents  for all the parents

OTS.INT.SPC{
   PARENT := INTP.GEM (act.gem.parent[k], INDICES.ARRAY[i])
   IF
      PARENT.numb > 0
      OTS2_PT.n := PARENT.numb
      OTS2_PT.c := act.c[k]
      OTS2_PT.GSCD_field := act.GSCD_field[k]
      OTS2_PT.type := act.ot_token.type[k]
      OTS2_PT.com_type := act.ot_token.com_type[k]
      OTS2_PT.decomp_type := act.ot_token.decomp_type[k]
      OTS2_PT.d := act.ot_token.d
      SEQ j=0 PARENT.numb
      IF
         act.c[k] = SET

227
OTS2_PT.true.p[j] := F(PARENT.ARRAY[j], act.parent_true[k], N)
OTS2_PT.true.t[j] := act.ETS_ptstart + i + act.ot_token.true.disp[k]
OTS2_PT.false.p[j] := F(PARENT.ARRAY[j], act.parent_false[k], N)
OTS2_PT.false.t[j] := act.ETS_ptstart + i + act.ot_token.false.disp[k]
act.c[k] = RESET
OTS2_PT.p[j] := F(PARENT.ARRAY[j], act.parent[k], N)
OTS2_PT.t[j] := act.ETS_ptstart + i + act.ot_token.disp[k]
END IF
END SEQ
OTS1[OTS_CURRENT].parent[k] := OTS2_PT
UPDATE (OTS2_PT)
TRUE
OTS1[OTS_CURRENT].parent[k] := NULL
END IF

}
END SEQ
END SEQ

D.2.3 OTS Initialization Scheme 3 (SBN)

The optimization of the OTS initialization process for the broadcast type activity is described in section 5.11.1. From step 5(a) of scheme 1, the initialization for the broadcast type output tokens will be as follows:

5(a). Find all the units of current true parent’s activity at local processing node (the function INTP.GEM.P) one by one.

5(b). For each local unit of current parent found in the previous step;
   i. Find the tag by using GET_DISP.ETS for local processing node.
   ii. Initialize the tag field of OTS2.

5(c). Find all the units of current false parent’s activity at local processing node (the function INTP.GEM.P) one by one.

5(d). For each local unit of current parent found in the previous step;
   i. Find the tag by using GET_DISP.ETS for local processing node.
   ii. Initialize the tag field of OTS2.

The initialization of scheme 1 from step 6(a) will be as follows:

6(a). Find all the units of current parent’s activity at local processing node (the function INTP.GEM.P) one by one.
6(b). For each local unit of current parent found in the previous step;
   i. Find the tag by using \textit{GET\_DISP\_ETS} for local processing node.
   ii. Initialize the tag field of OTS2.

The description of the above logic in pseudo Occam is as follows:

\begin{verbatim}
OTS\_INT3()  -- Simple, broadcast, non-constant-difference

INDICES := F\_INVERSE (Pi, act.ext, N)
OTS\_CURRENT := act.OTS\_pt\_start
OTS1[OTS\_CURRENT].N\_parent := act.No\_of\_parents
SEQ i:=0 FOR INDICES.numb
    SEQ k=0 FOR act.No\_of\_parents  -- for all the parents

OTS\_INT\_SBN{
    PARENT := INTP\_GEM\_P (act.gem\_parent[k], INDICES.ARRAY[i], P\_local)
    IF
        PARENT.numb > 0
            OTS2\_PT.n := PARENT.numb
            OTS2\_PT.c := act.c[k]
            OTS2\_PT.GSCD\_field := act.GSCD\_field[k]
            OTS2\_PT.type := act.ot\_token.type[k]
            OTS2\_PT.com\_type := act.ot\_token.com\_type[k]
            OTS2\_PT.decomp\_type := act.ot\_token.decomp\_type[k]
            OTS2\_PT.d := act.ot\_token.d
            SEQ j=0 FOR PARENT.numb
                IF
                    act.c[k] = SET  -- The choice flag is set
                        OTS2\_PT.true\_t[j] := act.parent\_true[k].ETS\_pt\_start +
                        GET\_DISP\_ETS(act.parent\_true[k], PARENT.ARRAY[j], P\_local)
                    OTS2\_PT.false\_t[j] := act.parent\_false[k].ETS\_pt\_start +
                        GET\_DISP\_ETS(act.parent\_false[k], PARENT.ARRAY[j], P\_local)
                ELSE
                    act.c[k] = RESET  -- no choice, single parent
                        OTS2\_PT.t[j] := act.parent[k].ETS\_pt\_start +
                        GET\_DISP\_ETS(act.parent[k], PARENT.ARRAY[j], P\_local)

    END IF
END SEQ
OTS1[OTS\_CURRENT].parent[k] := OTS2\_PT
UPDATE(OTS2\_PT)
TRUE
OTS1[OTS\_CURRENT].parent[k] := NULL
END IF

}
\end{verbatim}
D.2.4 OTS Initialization Scheme 4 (SBC)

This scheme is for the simple, broadcast, and the constant-difference output tokens. Since it is of the broadcast and constant-difference type, each processor is going to consume no more than one token generated by a unit child activity. To initialize the OTS for this type of tokens:

- just like scheme 2, there is no need to evaluate the function GET_DISP.ETS;
- there is no need to run the function INTP.GEM.P as in scheme 3.

But there may be a situation when a unit does not generate any output token for its current parent\(^1\). Therefore, there is a need to know that whether a unit is going to be consumed by its parent activity. If all the units of an activity are going to be consumed by a parent activity, then this information can be kept with the activity and the need to know the consumption of a unit by a parent can be eliminated. For this purpose, the parameter act.ot.token.N is added. If act.ot.token.T is set, then each child data element will generate the output tokens. Otherwise, the OTS initializer will check for every child's unit that whether it is going to be consumed by its parent activity. Therefore, the function IS_PAR_Cons.DATA is defined.

The pattern of output tokens is shown in Figure 5.16. The initializer for the broadcast, simple and constant-difference output tokens for step 5(a) as in scheme 3 of the previous section, will work as follows:

5(a). If all the units of the current activity are not going to be consumed by the current parent activity,

5(a)i. If current unit is consumed by the parent activity.
   5(a)i-1. Evaluate the tag by adding act.ot.token.diff for true parent to the ETS location of the current unit of the child activity.
   5(a)i-2. Evaluate the tag by adding act.ot.token.diff for false parent to the ETS location of the current unit of the child activity.

5(a)ii. If current unit is not consumed by the parent activity.
   5(a)ii-1. Skip and do not make any output token.

5(b). If all the units of the current activity are going to be consumed by the parent activity.

5(b)i. Evaluate the tag by adding act.ot.token.diff for true parent to the ETS location of the current unit of the child activity.

\(^1\)Length coercion and geometric operations which select a part of object in F-code.
5(b)i-2. Evaluate the tag by adding `act.ot_token.diff` for false parent to the ETS location of the current unit of the child activity.

The initialization of scheme 1 from step 6(a) will become as follows:

6(a). If all the units of the current activity are not going to be consumed by the current parent activity,

   6(a)i. If current unit is consumed by the parent activity.

      6(a)i-1. Evaluate the tag by adding `act.ot_token.diff` for parent to the ETS location of the current unit of the child activity.

   6(a)ii. If current unit is not consumed by the parent activity.

      6(a)ii-1. Skip and do not make any output token.

6(b). If all the units of the current activity are going to be consumed by the parent activity.

   6(a)i-1. Evaluate the tag by adding the `act.ot_token.diff` for parent to the ETS location of the current unit of the child activity.

The micro level description of the process in pseudo Occam is as follows:

```
OTS_INT4() -> Simple, broadcast, constant-difference

INDICES := F_INVERSE (Pi, act.ext, N)
OTS_CURRENT := act.OTS_ptstart
OTS1[OTS_CURRENT].N_parent := act.No_of_parents
SEQ i=0 FOR INDICES.numb
    SEQ k=0 FOR act.No_of_parents, for all the parents
    OTS_INT_SBC{
        IF
            act.ot_token.N[k] = RESET
            PARENT_FLAG := IS_PAR_CON_DATA (act, INDICES.ARRAY[i])
            TRUE
            PARENT_FLAG := YES
        END IF
        IF
            PARENT_FLAG = YES
            OTS2_PT.n := 1
            OTS2_PT.c := act.c[k]
            OTS2_PT.GSCD_field := act.GSCD_field[k]
            OTS2_PT.type := act.ot_token.type[k]
            OTS2_PT.com_type := act.ot_token.com_type[k]
            OTS2_PT.decomp_type := act.ot_token.decomp_type[k]
            OTS2_PT.d := act.ot_token.d
        IF
```
act.c[k] = SET
    OTS2_PT.true.t := act.ETS_ptstart + i + act.ot_token.true.disp[k]
    OTS2_PT.false.t := act.ETS_ptstart + i + act.ot_token.false.disp[k]
act.c[k] = RESET
    OTS2_PT.t := act.ETS_ptstart + i + act.ot_token.disp[k]
END IF
OTS1[OTS_CURRENT].parent[k] := OTS2_PT
UPDATE(OTS2_PT)
TRUE
OTS1[OTS_CURRENT].parent[k] := NULL
END IF

END SEQ
END SEQ

D.2.5 OTS Initialization Scheme 5 (CPN)

This scheme is for the compound, Pt_T_Pt, and non-constant-difference tokens. If a processor is going to consume the result of a unit of a child activity for more than one unit of a parent activity with regular pattern of the ETS, then this situation can be decoded into a compound token. The logic for the initialization of output tokens for this type of activities will be as follows.

1. For an activity, find all the units which will be processed at the same processor where the OTS is being initialized (the function F⁻¹). This processor is named as local processor, P_i.

2. Find the starting location of the OTS1 from the OTS.ptstart and initialize with the number of parents (act.parents).

3. For each unit found in step 1, and for each parent of the current activity, do as follows:

   (a) If the child unit is going to be consumed by the current parent, get the pointer to the current empty location in OTS2, initialize the current location in OTS1 with the pointer. Initialize the template for the output tokens for current parent’s activity in OTS2.

   (b) For each processing node of the N-computer;

      i. The initializer finds the first unit of the parent activity which will consume the child unit (using the function F_PAR_AT_P).

      ii. If the flag c (act.ot_token.c) of the activity is set;

          A. It finds the tag (using function GET_DISP_ETS) for true parent.
          B. It initializes the processor and the tag field in the OTS2 for true parent.
          C. It finds the tag (using function GET_DISP_ETS) for false parent.
D. It initializes the processor and the tag field in the OTS2 for false parent.
   iii. If the flag c (act.ot_token.c) of the activity is reset;
   A. It finds the tag (using function GET_DISP.ETS) for the parent.
   B. It initializes the processor and the tag field in the OTS2 for the parent.

The micro level description of the process is as follows:

OTS_INT5() - Compound, Pt.T.Pt, non-constant-difference

INDICES := F.INVERSE (Pi, act.ext, N)
OTS_CURRENT := act.OTS.ptstart
OTS[OTS_CURRENT].N_parent := act.No_of_parents
SEQ i=0 FOR INDICES.numb
SEQ k=0 FOR act.No_of_parents  - for all the parents

OTS_INT_CPNI

FIRST_PARENT := YES
SEQ j=0 FOR N  - for all processors
PARENT := F.PAR_AT_P (act, INDICES.ARRAY[i], j)
IF
  PARENT.numb > 0
  IF
    FIRST_PARENT = YES
      OTS2_PT.n := PARENT.numb
      OTS2_PT.c := act.c[k]
      OTS2_PT.GSCD_field := act.GSCD_field2[k]
      OTS2_PT.type := act.ot_token.type[k]
      OTS2_PT.com_type := act.ot_token.com_type[k]
      OTS2_PT.decomp_type := act.ot_token.decomp_type[k]
      OTS2_PT.d := act.ot_token.d
      FIRST_PARENT := NO
      m := 0
      TRUE
      OTS2_PT.n := OTS2_PT.n + 1
  END IF
  IF
    act.c[k] = SET  - The choice flag is set
      OTS2_PT.true.p[m] := j
      OTS2_PT.true.t[m] := act.parent.true[k].ETS_ptstart +
               GET_DISP.ETS(act.parent.true[k], PARENT.ARRAY[0], j)
      OTS2_PT.true.I[m] := act.ot_token.true.I[k]
      OTS2_PT.true.N[m] := act.ot_token.true.N[k]
      OTS2_PT.false.p[m] := j
      OTS2_PT.false.t[m] := act.parent.false[k].ETS_ptstart +
               GET_DISP.ETS(act.parent.false[k], PARENT.ARRAY[0], j)

233
OTS2_PT.false.I[m] := act.ot.token.false.I[k]
OTS2_PT.false.N[m] := act.ot.token.false.N[k]
act.c[k] = RESET — no choice, single parent
OTS2_PT.p[m] := j
OTS2_PT.t[m] := act.parent[k].ETS_ptstart +
     GET_DISP_ETS(act.parent[k], PARENT.ARRAY[0], j)
OTS2_PT.I[m] := act.ot.token.I[k]
OTS2_PT.N[m] := act.ot_token.N[k]
END IF
UPDATE(m)
TRUE — no output token at j processor
SKIP
END IF
END SEQ
IF
FIRST.PARENT = YES
OTS[OTS_CURRENT].parent[k] := NULL
TRUE
OTS[OTS_CURRENT].parent[k] := OTS2_PT
UPDATE(OTS2_PT)
END IF

END SEQS
END SEQ

D.2.6 OTS Initialization Scheme 6 (CPC)

In the initialization process for this type of tokens, the step 3(b)i of Scheme 5 will be as follows:

- The initializer finds whether the parent activity will consume the child unit at the processor (using function IS_PAR_CONS_DATA_AT_P).

The step 3(b)ii-A, 3(b)ii-C and 3(b)iii-A of Scheme 5 will be as follows:

- Evaluate the tag by adding act.ot.token.diff for the parent to the ETS location of the current unit of the child activity.

Like the process of scheme 2, this will have less complexity in the third loop as compared to the process of the scheme 5, described in the previous section. The micro level description of the process to initialize the OTS for this type of activity is as follows:

OTS.INT6Q — Compound, Pt.T_Pt, constant-difference
INDICES := F.INVERSE (Pi, act.ext, N)
OTS.CURRENT := act.OTS_ptstart
OTS1[OTS.CURRENT].N_parent := act.No_of_parents
SEQ i=0 FOR INDICES.numb
SEQ k=0 FOR act.No_of_parents  - for all the parents

OTS.INT_CPC{

FIRST_PARENT := YES
SEQ j=0 FOR N  - for all processors
PARENT_FLAG := IS_PAR_AT_P (act, INDICES.ARRAY[i], j)
IF
PARENT_FLAG = YES
IF
FIRST_PARENT = YES
OTS2_PT.n := 1
OTS2_PT.c := act.c[k]
OTS2_PT.GSCD_field := act.GSCD_field2[k]
OTS2_PT.type := act.ot_token.type[k]
OTS2_PT.com_type := act.ot_token.com_type[k]
OTS2_PT.decomp_type := act.ot_token.decomp_type[k]
OTS2_PT.d := act.ot_token.d
FIRST_PARENT := NO
m := 0
TRUE
OTS2_PT.n := OTS2_PT.n + 1
END IF
IF
act.c[k] = SET
OTS2_PT.true.p[m] := j
OTS2_PT.true.t[m] := act.ETS_ptstart + i + act.ot_token.true_disp[k]
OTS2_PT.true.I[m] := act.ot_token.true.I[k]
OTS2_PT.true.N[m] := act.ot_token.true.N[k]
OTS2_PT.false.p[m] := j
OTS2_PT.false.t[m] := act.ETS_ptstart + i + act.ot_token.false_disp[k]
OTS2_PT.false.I[m] := act.ot_token.false.I[k]
OTS2_PT.false.N[m] := act.ot_token.false.N[k]
act.c[k] = RESET
OTS2_PT.p[m] := j
OTS2_PT.t[m] := act.ETS_ptstart + i + act.ot_token.disp[k]
OTS2_PT.I[m] := act.ot_token.I[k]
OTS2_PT.N[m] := act.ot_token.N[k]
END IF
UPDATE (m)
TRUE  - no output token for this processor

235
D.2.7 OTS Initialization Scheme 7 (CBN)

The micro level description of the process as described in section 5.11.2 will be as follows:

OTS_INT8() — Compound, broadcast, non-constant-difference

INDICES := F_INVERSE (Pi, act.ext, N)
OTS_CURRENT := act.OTS_ptstor£
OTS1[OTS_CURRENT].Npareni := act.No_of_parents
SEQ i=0 FOR INDICES.numb
SEQ k=0 FOR act.No_of_parents — for all the parents

OTS_INT_CBN{

PARENT := F_PAR_AT_P (act, INDICES.ARRAY[i], P_local)
IF
PARENT.numb > 0
OTS2_PT.n := 1
OTS2_PT.c := act.c[k]
OTS2_PT.GSCD_field := act.GSCD_field2[k]
OTS2_PT.type := act.ot_token.type[k]
OTS2_PT.com_type := act.ot_token.com_type[k]
OTS2_PT.decomp_type := act.ot_token.decomp_type[k]
OTS2_PT.d := act.ot_token.d
IF
act.c[k] = SET — The choice flag is set
OTS2_PT.true.t := act.parent_true[k].ETS_ptstart +
GET_DISP_ETS(act.parent_true[k], PARENT.ARRAY[0], P_local)
OTS2_PT.true.I := act.ot_token.true.I[k]

236
D.2.8 OTS Initialization Scheme 8 (CBC)

The micro level description of the process as described in section 5.11.2 will be as follows:

```
OTS_INT8()  - Compound, broadcast, constant-difference

INDICES := F_INVERSE (Pi, act.ext, N)
OTS_CURRENT := act.OTS_ptstart
OTS[OTS_CURRENT].N_parent := act.No_of_parents
SEQ i=0 FOR INDICES.numb
  SEQ k=0 FOR act.No_of_parents  - for all the parents
    Obf TS_INT_CBC{
      IF
        act.ot_token.N[k] = RESET
        PARENT_FLAG := IS_PAR_CONS_DATA (act, INDICES.ARRAY[i])
        TRUE
        PARENT_FLAG := YES
        END IF
      IF
        PARENT_FLAG = YES
        OTS2_PT.n := 1
      END IF
    }
END SEQ

END SEQ
```

237
OTS2_PT.c := act.c[k]
OTS2_PT.GSCD_field := act.GSCD_field2[k]
OTS2_PT.type := act.ot_token.type[k]
OTS2_PT.com_type := act.ot_token.com_type[k]
OTS2_PT.decomp_type := act.ot_token.decomp_type[k]
OTS2_PT.d := act.ot_token.d
IF
  act.c[k] = SET
    OTS2_PT.true.t := act.ETS_ptstart + i + act.ot_token.true.disp[k]
    OTS2_PT.true.I := act.ot_token.true.I[k]
    OTS2_PT.true.N := act.ot_token.true.N[k]
    OTS2_PT.false.t := act.ETS_ptstart + i + act.ot_token.false.disp[k]
    OTS2_PT.false.I := act.ot_token.false.I[k]
    OTS2_PT.false.N := act.ot_token.false.N[k]
  act.c[k] = RESET
    OTS2_PT.t := act.ETS_ptstart + i + act.ot_token.disp[k]
    OTS2_PT.I := act.ot_token.I[k]
    OTS2_PT.N := act.ot_token.N[k]
END IF
OTS1[OTS_CURRENT].parent[k] := OTS2_PT
UPDATE(OTS2_PT)
TRUE
OTS1[OTS_CURRENT].parent[k] := NULL
END IF

END SEQ
END SEQ

D.3 STE Initialization

The starting token emitter as described in section 5.6 in pseudo Occam is as follows:

STE_INIT()

INDICES := F_INVERSE (Pi, act.ext, N)
SEQ i=0 FOR INDICES.numb
  SEQ k=0 FOR act.No_of_parents
    token.type := act.ot_token.type
    token.decomp_type := act.ot_token.decomp_type
    token.v := act.OP[i]
    IF
      act.ot_token.decomp_type = COMPOUND
    END IF
PARENT := F_PAR_AT_P(act.gem_parent[k], INDICES.ARRAY[i], P_{local})
token.I := act.ot.token.I
token.N := act.ot.token.N
TRUE
PARENT := INTP_GEM_AT_P(act.gem_parent[k], INDICES.ARRAY[i], P_{local})
END IF
SEQ j=0 FOR PARENT.numb
  token.t := act.parent[k].ETS_ptstart +
  GET_DISP_ETS(act.parent[k], INDICES.ARRAY[i], P_{local})
  Int_STEData_path <= token
END SEQ
END SEQ
END SEQ
Appendix E

Some Detailed Examples

E.1 Reduce Without The Counter

In this appendix, a way to support the reduce function of F-code without taking into account the counter in the output token formation system will be discussed with the help of an example. This example is given to support the conclusion made in section 6.4.3.

A straightforward way to support the execution of reduce function by using ETS model of synchronization for binary events without taking into account the counter in the output token formation system, is to break the object and convert the function into a binary tree. Then execute that binary tree just like a diadic function. To break an object into more than one object, we have to apply geometric operations. An object can be broken down into more than one object by applying geometric operations. To illustrate the idea, consider the following example:

Example 40

\[
\begin{align*}
&\text{(EXPR}_\text{parent} \\
&\quad (\text{EXPR}_\text{add} 001 \\
&\quad \quad (\text{EXPR}_\text{a}) \\
&\quad ) \\
&\quad ) \\
&\end{align*}
\]

An equivalent F-code program for the above code in the binary functions, as shown in Figure E.1, will be as follows:

Example 41

\[
\begin{align*}
&\text{(EXPR}_\text{parent} \\
&\quad \%\% p1 \text{ activity} \\
&\quad (\text{hold A} \\
&\quad \quad (\text{diaidc add} \quad \%\% p2 \text{ activity}
\end{align*}
\]
(diadic add % p3 activity
  (geometric_function
    (var value A)
  )
  (diadic add % p5 activity
    (geometric_function
      (var value A)
    )
    (geometric_function
      (var value A)
    )
  )
)
(diadic add % p4 activity
  (diadic add % p6 activity
    (geometric_function
      (var value A)
    )
    (geometric_function
      (var value A)
    )
  )
)
(diadic add % p7 activity
  (geometric_function
    (var value A)
  )
  (geometric_function
    (var value A)
  )
)
)
)
(EXPR.a) % p8 activity
)

The F-code tree of Example 41 can be transformed into activity tree which is as follows:

Example 42

......
Activitych..., p1 ← EXPRparent
......

241
Figure E.1: The structure of a binary tree to support the evaluation of reduce function by using ETS without the counter.
The initialization process as described in Appendix B will initialize ETS and OTS from the above activity tree. The above description shows that the function reduce can be supported by using binary ETS model. But there are some serious problems in adopting this strategy for the evaluation of reduce function which are given in section 6.4.3.
Appendix F

Analysis Phase of F-SIM

In this appendix, the state transition diagrams for the analysis phase of F-SIM are described. State0 is the start of the analysis phase. State1 inputs the expression from the standard input of an input file. State2 decides which expression has been input. Then the analyzer enters into the state3 in which it performs different functioning according to the expression.

Figure A.1- First three states of the syntax analyzer of F-SIM
From state 2 (Termination)

State transition for 'const'  

State transition for 'result'

State transition for 'scope'

State transition for 'bind'

State transition for 'var', 'call' and 'assign'

State transition for 'monadic'

State transition for 'choice'

Figure A.2- State transition diagrams of the syntax analyzer for individual functions
Figure A.3 - State transition of the syntax analyzer of the individual functions
Figure A.4- State transitions of the syntax analyzer for individual functions
Appendix G

Logic for the Simulation of F-code Functions

In this appendix the logic implemented in F-SIM for the simulation of each F-code function is described in flow charts.

Figure G.1- Interpretation of the individual functions
Figure B.2 - Interpretation of the individual functions
Figure B.3- Interpretation of individual functions
Interpreter of the 'ramp'

Interpreter of the 'dyadic'

Figure B.4- Interpretation of individual functions
Interpretation of the 'choice'

Interpretation of the 'transp'

Figure B.5- Interpretation of the individual functions
Interpretation of the 'meat'

Interpretation of the 'slice'

Figure B.6- Interpretation of the individual functions
**Interpretation of the 'reduce'**

1. **Intep exprs**
2. **Interpretation of the 'reduce'**
   - If exprs = null
     - Increment count
     - Walk down and get next expr
   - If count < length
     - No
   - Increment count
   - Intep expr
3. Switch on type
   - If int
     - Coerce expr in the integer
   - If int
     - Walk up to the previous expr
4. Fill the res-object with the contents of the object <exprs> according to the definition
5. Pop the object <exprs> and make the final descriptor of the res-object
6. Pop each object according to the mask length and put the value into the shape vector of the res-object
7. Push the resulting object on the stack

**Interpretation of the 'repl'**

*Figure B.7- Interpretation of the individual functions*
Figure 5.8 - Interpretation of the individual functions
Interpretation of the 'assign'

Interpretation of the 'comma'

Interpretation of the 'loop'

Figure B.9- Interpretation of individual functions
Interpretation of the 'create'

Interpretation of the 'if'

Figure B.10- Interpretation of individual functions
Appendix H

Definition of f-code

This appendix is a part of the report written by A. B. Bolychevsky, V. Muchnick and A. Shafarenko, which explains the concept of F-code and gives its definition. In this appendix, the definition and the example are given. All the report can be consulted as in [24].

H.1 Objects

The f-functions receive as operands, and return as results, primitive objects, which are homogeneous, scalar or data-concurrent nonscalar, aggregate of scalar elements having the following characteristics:

1. Type $t \in \{\text{logical, character, integer, real, complex}\}$. The types form an ascending hierarchy in the same order as they are listed above. For example, a character type is junior to the real type and senior to the logical type.

2. Sort $s \in \{\text{value, name, target}\}$. If an object is of sort value, it contains data of type $t$; if it is a name, every element of it contains a reference to a value element of type $t$; if it is a target, every element of it contains either a reference to a value element of type $t'$ or a special dummy reference. Targets are used for elementwise movement of values in the course of assignment. A name can be used the same way as a target, but also admits data-parallel dereferencing, yielding a value of type $t$.

3. Shape $d = \{d_i\}$ It is a vector of the extents of each of the object’s dimensions $d_i$, each of which is a nonnegative integer number. The length of $d$ is equal to the object’s rank. In particular, if the object is scalar, vector $d$ has zero length.

4. Contents $c$, which form an array of shape $d$.

The f-program can manipulate heterogeneous aggregates of data as well, such as Pascal records or C structures. Characteristics of such structured objects are as follows:

1. Template (metatype) $T$, which is a tree whose leaves are types; all the nodes of this tree except the root one are labelled with shape vectors. Thus, at each level of hierarchy

---

Footnote:

1In this document we assume that an index of any kind starts with zero
a template determines a sequence of fields. The number of the fields is equal to the
number of successors of the node, whilst the types (or templates) and shape vectors of
those fields are defined by the corresponding leaves (or subtrees) and their labels.

2. Shape \( d \).

3. Contents \( c \), which form an array of shape \( d \) built up from structures of template \( T \).

Structured objects can only consist of values. However, while accessing their homogeneous
fields, primitive objects of any sort may arise. Such an access is effected through pointers,
which are primitive objects of integer type. Pointers are also used for references to
subroutines.

### H.2 Representation of the f-program

The f-program is a single expression to be evaluated. It is represented in a LISP-like list
form. Two kinds of atoms are used:

**Literals.** A literal is a self-defined entity having a regular syntax. Its semantics is unambiguously and statically determined by the syntax derivation. For example, \( 3.5 \) is a
literal representing real number \( 3.5 \) and the literal \( \text{mul} \) denotes multiplication.

**Identifiers.** The identifier is the only semantically variable entity in the f-language. They
are associated with primitive objects, pairs, templates or channels.

While defining syntax of the language, we will use the standard BNF notation with the
only extension: ellipsis \( ... \) stands for any number of repetitions of the preceding item (or
the braced group of items).

### H.3 Identifiers

The identifier is composed of reasonably large set of characters. It need not start with an
alphabetic symbol.

\[
\text{<identifier> ::= <symbol>...}
\]

\[
\text{<symbol> ::= <h-digit> |}
\]

\[
G | H | I | J | K | L | M | N | O | P |
\]

\[
Q | R | S | T | U | V | W | X | Y | Z |
\]

\[
g | h | i | j | k | l | m | n | o | p |
\]

\[
q | r | s | t | u | v | w | x | y | z |
\]

\[
$ | _ | - | .
\]
H.4 Literals

H.4.1 Constants

Apart from traditional constants, the set of f-constants includes machine specific characteristics and pointers to subroutines.

<constant> ::= <logical> | <character> | <string> |
<integer> | <real> | <complex> | <special> |
<subroutine>

<logical> ::= true | false

<character> ::= '<ASCII>'

<string> ::= "<ASCII>..."

<integer> ::= <decimal> | <octal> | <hexadecimal> | <bit>

<decimal> ::= <number> | <sign><number>

<sign> ::= + | -

<number> ::= <significant> | <significant><d-digit>...

<octal> ::= 0<o-digit>...

<hexadecimal> ::= 0X<h-digit>... | Ox<h-digit>...

<bit> ::= 0B<mask> | 0b<mask>

<mask> ::= <b-digit>...

<b-digit> ::= 0 | 1

<o-digit> ::= <b-digit> | 2 | 3 | 4 | 5 | 6 | 7

<d-digit> ::= <o-digit> | 8 | 9

<significant> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<h-digit> ::= <d-digit> |
A | B | C | D | E | F | a | b | c | d | e | f

<real> ::= <unsigned> | <sign><unsigned>
\[
\begin{align*}
\text{<unsigned>} & ::= \text{<number>}.\text{<number>}\text{<exponent>} \mid \\
& \quad \text{<number>}.\text{<number>} \mid \\
& \quad \text{<number>}\text{<exponent>}
\end{align*}
\]
\[
\begin{align*}
\text{<exponent>} & ::= E\text{<decimal>} \mid e\text{<decimal>}
\end{align*}
\]
\[
\begin{align*}
\text{<part>} & ::= \text{<number>} \mid \text{<unsigned>}
\end{align*}
\]
\[
\begin{align*}
\text{<complex>} & ::= \text{<part>}\text{<sign>}\text{<part>}i \mid \text{<sign>}\text{<part>}\text{<sign>}\text{<part>}i
\end{align*}
\]
\[
\begin{align*}
\text{<special>} & ::= \text{top} \mid \text{epsilon} \mid \text{max-int} \mid \text{max-char} \mid \text{hole}
\end{align*}
\]
\[
\begin{align*}
\text{<subroutine>} & ::= \text{<EXPR>}
\end{align*}
\]

Notes:

1. As there is no point in defining the standard ASCII character set, we simply included a non-terminal token \texttt{<ASCII>} denoting an arbitrary ASCII character. Quote " should be doubled if it occurs as an \texttt{<ASCII>} in a string.

2. Non-terminal \texttt{<EXPR>} is defined in section H.6 and generates f-expressions.

These are examples of legal constants and like entities that we shall use further:

1. \texttt{<logical>}: false
2. \texttt{<character>}: 'a' '+' '3'
3. \texttt{<string>}: "12abc@#/" "This is string ""X"""
4. \texttt{<number>}: 12 25
5. \texttt{<decimal>}: 12 -25
6. \texttt{<octal>}: 013 07777
7. \texttt{<hexadecimal>}: 0X12A3F 0x12a3F 0x12a3f
8. \texttt{<mask>}: 1011 0001 000
9. \texttt{<bit>}: 0b100111 0B1111011
10. \texttt{<real>}: 3.56 1e10 -12.2E-3 -12.2e-3 +25.9
11. \texttt{<complex>}: 1.5+2.7i -3e7+0.1i 2-3i
12. \texttt{<special>}: epsilon
13. \texttt{<subroutine>}: (dyadic add (var value X) (const 1))

261
H.4.2 Operators

These literals denote all the element-wise operators that are supported by the f-language:

\[
\begin{align*}
\text{<unary> &:} & \text{ neg | inv | frac | sqrt | arg | conj | modulus | sin | cos | tan | arcsin | arccos | arctan | sinh | cosh | tanh | exp | ln | even | odd | not | bit-not |} \\
\text{<total> &:} & \text{ add | mul | max | min | and | or | xor | bit-and | bit-or | bit-xor} \\
\text{<binary>} & \text{ <total> | sub | div | idiv | modulo | pow | ipow | log | shr | shl | ror | rol | gt | ge | lt | le | eq | ne}
\end{align*}
\]

To avoid any ambiguity, the following table defines the semantics of operators. For the sake of convenience let us denote the operand of a unary operation as \( z \) and the operands of the binary operation as \( x \) and \( y \).

<table>
<thead>
<tr>
<th>Literal</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>neg</td>
<td>(-z)</td>
</tr>
<tr>
<td>inv</td>
<td>(\frac{1}{z})</td>
</tr>
<tr>
<td>frac</td>
<td>(z - \lfloor z \rfloor)</td>
</tr>
<tr>
<td>sqrt</td>
<td>(\sqrt{z})</td>
</tr>
<tr>
<td>arg</td>
<td>(\text{arg } z)</td>
</tr>
<tr>
<td>conj</td>
<td>(z^*)</td>
</tr>
<tr>
<td>modulus</td>
<td>(</td>
</tr>
<tr>
<td>sin</td>
<td>(\sin z)</td>
</tr>
<tr>
<td>cos</td>
<td>(\cos z)</td>
</tr>
<tr>
<td>tan</td>
<td>(\tan z)</td>
</tr>
<tr>
<td>arcsin</td>
<td>(\text{arcsin } z)</td>
</tr>
<tr>
<td>arccos</td>
<td>(\text{arccos } z)</td>
</tr>
<tr>
<td>arctan</td>
<td>(\text{arctan } z)</td>
</tr>
<tr>
<td>sinh</td>
<td>(\text{sinh } z)</td>
</tr>
<tr>
<td>cosh</td>
<td>(\text{cosh } z)</td>
</tr>
<tr>
<td>tanh</td>
<td>(\text{tanh } z)</td>
</tr>
<tr>
<td>exp</td>
<td>(e^z)</td>
</tr>
<tr>
<td>ln</td>
<td>(\ln z)</td>
</tr>
<tr>
<td>even</td>
<td>true only if ( z ) is even</td>
</tr>
<tr>
<td>odd</td>
<td>true only if ( z ) is odd</td>
</tr>
<tr>
<td>not</td>
<td>(\bar{z})</td>
</tr>
<tr>
<td>bit-not</td>
<td>bitwise (\bar{z})</td>
</tr>
<tr>
<td>round</td>
<td>\text{round } z</td>
</tr>
<tr>
<td>trunc</td>
<td>\text{truncate } z</td>
</tr>
<tr>
<td>font</td>
<td>\text{character with code } z</td>
</tr>
</tbody>
</table>
Literal  Definition

  add    \( x + y \)
  mul    \( xy \)
  max    max\((x, y)\)
  min    min\((x, y)\)
  and    \( \& \)\)
  or     \( | \)\)
  xor    \( \oplus \)\)
  bit-and bitwise \( \& \)\)
  bit-or bitwise \( | \)\)
  bit-xor bitwise \( \oplus \)\)
  sub    \( x - y \)
  div    \( x/y \)
  idiv   \( x/y, \text{error if } x \quad (\text{mod} \ y) \neq 0 \)
  modulo \( x \quad (\text{mod} \ y) \)
  pow    \( x^y, x \text{ and } y \text{ must be real} \)
  log    \( \log_x y, x \text{ and } y \text{ must be real} \)
  ipow   \( x^y, x \text{ is arbitrary, } y \text{ must be integer} \)
  shr    shift \( x \ \text{y times right} \)
  shl    shift \( x \ \text{y times left} \)
  ror    rotate \( x \ \text{y times right} \)
  rol    rotate \( x \ \text{y times left} \)
  gt     \( x > y \)
  ge     \( x \geq y \)
  lt     \( x < y \)
  le     \( x \leq y \)
  eq     \( x = y \)
  ne     \( x \neq y \)

The following tables define admissible operand types and the result type for the above operators. Every item in the first column specifies a group of operators with the same rule of determining the result type. Letters \( L, H, I, R, C \) and \( U \) stand for an operand of logical, character, integer, real, complex, or “unclear” type, respectively. The corresponding item in the second column indicates the result type versus the unclear operand type in the increasing order. For example “. . \( I \ R \ R \)” would mean that any operation belonging to the group is undefined for \( U \) being logical or character, and will yield an integer result if the unclear-type operand happens to be integer, and a real result if it happens to be real or complex.
Arithmetic

| U add U | U sub U | neg U | U mul U | ... | I R C |
| U div U | inv U | ... | R C |
| I idiv I | I mod I | integer |
| modulus U | ... | I R R |
| frac R | sqrt R | real |
| R pow R | real |
| U ipow I | ... | I R C |
| conj C | complex |
| arg C | real |
| U max U | U min U | ... | H I R . |
| I shr I | I shl I | integer |
| I ror I | I rol I | integer |

Basic Functions

| sin R | cos R | tan R | real |
| arcsin R | arccos R | arctan R | real |
| sinh R | cosh R | tanh R | real |
| ln R | R log R | real |
| exp U | ... | R C |

Comparisons

| U gt U | U ge U | U lt U | U le U | ... | L L L |
| U eq U | U ne U | L L L L L |
| even I | odd I | logical |

Logic

| not L | L and L | L or L | L xor L | logical |
| bit-not I | I bit-and I | I bit-or I | I bit-xor I | integer |

Descents

| re C | im C | real |
| round R | trunc R | integer |
| font I | character |

Note, that the set of admissible types for an operand of an unclear type is always a subrange so that it can be characterised with the junior $t'$ and senior $t''$ admissible types.
### H.4.3 Keywords

The following literals are used by f-functions as keywords:

\[ \text{<type>} ::= \text{"logical | "character | "integer | "real | "complex} \]

\[ \text{<sort>} ::= \text{value | target | name} \]

\[ \text{<access>} ::= \text{<sort> | pointer} \]

\[ \text{<property>} ::= \text{<sort> | couple} \]

\[ \text{<selector>} ::= \text{re | im} \]

### H.4.4 Function Labels

All the function labels are literals as well.

### H.5 Evaluation of the f-program

The evaluation rules are as follows:

1. The first item of the function list is a literal determining the function to be applied to the other items of the list.

2. Subsequent items are literals and identifiers the function needs to specify the activity it is to provide, or the arguments of the function, which are expressions that the f-system evaluates prior to the function invocation, unless otherwise stated in the function definition.

3. Any f-function that assumes its arguments to be evaluated before its invocation does not require any specific order of that evaluation. It ends up by forming operand objects, which we call \textit{a priori} operands. The f-function deals with \textit{a posteriori} operands, which are obtained from the \textit{a priori} ones as a result of optional \textit{type coercion}. A coercion changes the operand type; a value going up the hierarchy generalising the contents elementwise, a target going down with the contents preserved. A name can not be coerced.

While defining the function semantics, we will specify the \textit{a posteriori} types of all of the operands. It will be assumed that if the \textit{a priori} type of an operand happens to be different, then the coercion will normally take place. If the coercion is impossible, e.g. if it should attempt to lower the type of a value or change the type of name, this will result in an error.

### H.6 Functions

In this section we present a list of function definitions. Construct \texttt{EXPR} enumerates all the functions and is introduced to recursively close the f-code syntax.
We chose to define the semantics by providing mathematical formulas which evaluate the characteristics of the result object from those of operands. Here are some auxiliary formulas we are going to use for that purpose. They are to do with some transformations of, and relations between, coordinate or shape vectors we use to define the semantics of the f-language, rather than with any nonscalar objects the f-language manipulates.

1. The contents array of a primitive object is indexed with a multi-index, which is a vector of nonnegative integers. Let us denote the length of a as \( r(a) \) and introduce a partial precedence of such vectors. We will say that a vector \( a \) precedes a vector \( b \) if \( r(a) = r(b) = r \) and \( a_k < b_k \) for all \( 0 < k < r \). We should denote this as \( a \prec b \). All the other operations with vectors that we will use below should be interpreted elementwise.

2. For a vector \( a \) of some length and and a mask \( m \) of the same length we introduce a projector as follows:

\[
P : (a, m) \rightarrow b,
\]

where \( b_k = a_x(k, m) \), and \( x(k, m) \) is the number of the \( k \)-th unity bit in the mask \( m \). The length of the result vector is equal to the number of unity bits in the mask.

3. Let us introduce an expander:

\[
E : (a, m) \rightarrow b,
\]

so that

\[
b_k = \begin{cases} 
\infty & \text{if } m_k = 0 \\
ax(k, m) & \text{otherwise} 
\end{cases},
\]

where \( X(k, m) \) is the number of mask bits \( m_j = 1 \) with \( j < k \). Expander \( E(a, m) \) is determined only if \( m \) has the number of unity bits equal to the length of \( a \). The length of \( b \) is equal to the number of bits in mask \( m \). Thus the following identity takes place:

\[
P(E(a, m), m) = a.
\]

To facilitate references to different function arguments and operands obtained in their evaluation, we will use a special suffix .x with some character x. Letters in formulas may use the same character as a superscript. Also the following shorthand notations will use the same character:

\[
p^x(k) = P(k, m^x),
\]
\[ p^\sharp(k) = P(k, m^\sharp), \]
\[ e^\sharp = E(d^\sharp, m^\sharp), \]

where \( m^\sharp \) is a mask preceding the argument \( \text{EXPR.x} \) and \( \overline{m^\sharp} \) is the result of inversion of \( m^\sharp \):

\[
\overline{m_i^\sharp} = \begin{cases} 
0, & m_i^\sharp = 1 \\
1, & m_i^\sharp = 0
\end{cases} .
\]

Let us denote as \( s^\sharp(k) \) an array containing all the elements of some layer of operand \( \text{EXPR.x} \) contents. Its shape is \( d = p^\sharp(d^\sharp) \) and its elements are determined by the following formula:

\[ s^\sharp(k)_l = e^\sharp_{\min(E(k,m^\sharp),E(l,m^\sharp))}, \ 1 < d. \]

### H.6.1 CONST

This function returns an object, whose type, sort, shape and contents are determined by a literal.

\[ \text{CONST} ::= ( \text{const} \ <\text{constant}> ) \]

Depending on the \(<\text{constant}>\) the result is as follows:

1. **<logical>:** A logical scalar value with the contents specified.

2. **<character>:** A character scalar value containing the character specified between apostrophes.

3. **<string>:** A one-dimensional character value with the contents and shape determined by the sequence of characters between quotes. All the doubled quotes in this sequence are replaced by single ones.

4. **<integer>:** An integer scalar value containing the number specified.

5. **<real>:** A real scalar value containing the number specified.

6. **<complex>:** A complex scalar value containing the number specified.

7. **top:** A real scalar value that contains machine-specific constant \( \tau \), such that \( \tau \) and \( \frac{1}{\tau} \) approximate the upper and lower boundaries of the floating-point range.

8. **epsilon:** A real scalar value that contains machine-specific constant \( \epsilon \), being the minimal floating point number for which \( 1 + \epsilon > 1 \) and \( 1 - \epsilon < 1 \).

9. **max-int:** An integer scalar value containing the maximal integer number supported by implementation.

10. **max-char:** A character scalar value containing the maximum code character belonging to the character set of the implementation.

11. **hole:** A complex scalar target containing the dummy reference. Its a priori complex type enables coercion to any a posteriori type.

12. **<subroutine>:** An integer scalar value containing a pointer to the subroutine defined by the \( f \)-expression specified.
**H.6.2 HOLD**

This function arranges a scope for a primitive object. It does not assume its first argument to be precomputed.

\[
\text{HOLD} ::= ( \text{hold} \ <\text{identifier}> \ \text{EXPR}.a \ \text{EXPR}.i )
\]

The algorithm of the function evaluation is this:

1. The existing association for the identifier (if any) is saved.
2. The identifier is associated with the \text{EXPR}.i operand.
3. The argument \text{EXPR}.a is computed using the new association of the identifier. Its a posteriori type coincides with the a priori one.
4. The object associated with the identifier is destroyed.
5. The current association for the identifier is destroyed and the saved one (if any) is restored.
6. Operand \text{EXPR}.a is returned.

**H.6.3 TEMPLATE**

This function arranges a scope for a template. It does not assume its first argument to be precomputed.

\[
\text{TEMPLATE} ::= ( \text{template} \ <\text{identifier}> \ \text{EXPR}.a \ \text{FIELD} \ldots )
\]

\[
\text{FIELD} ::= \text{MODE} \ \text{EXPR}.e \ldots
\]

\[
\text{MODE} ::= <\text{type}> | <\text{identifier}>
\]

The algorithm of the function evaluation is this:

1. The existing association for the identifier (if any) is saved.
2. The identifier specified after the function label is associated with a template that is built in the following way. The number of successors of the template root is determined by the number of the fields specified. If the type alternative is chosen as mode, then the node in question is a leaf of that type. If the identifier alternative is chosen, then this identifier must be associated with another template; in that case the node is the root of a subtree, which is a copy of that template. In both cases all the operands \text{EXPR}.e_i, 0 \leq i < n, must be scalar values of a posteriori integer type, and the label of the node is determined as follows:

\[
d_i = c^{e_i}, \ 0 \leq i < n.
\]

3. The argument \text{EXPR}.a is computed using the new association of the identifier. Its a posteriori type coincides with the a priori one.
4. The current association of the identifier is destroyed and the saved one (if any) is restored.
5. Operand \text{EXPR}.a is returned.
H.6.4 CREATE

This function creates a primitive or a structured variable and arranges a scope for it. It does not assume its first argument to be precomputed.

CREATE ::= ( create <identifier> EXPR.a MODE EXPR.e ... )
MODE ::= <type> | <identifier>

Operands $EXPR.e_i$, $0 \leq i < n$, must be scalar values of a posteriori integer type.

The algorithm of the function evaluation is this:

1. The existing association for the identifier (if any) is saved.

2. Depending on the alternative chosen as a mode, one of the following is done:

   (a) $<\text{type}>$: The primitive value of this type having arbitrary contents is created; its shape is determined as follows:

   $$d_i = c^{e_i}, \ 0 \leq i < n,$$

   Then this value is enreferenced by creating a name with the same type and shape that refers to the value elementwise. The identifier specified after the function label is associated with a couple that includes the original value and the result of its enreferencing.

   (b) $<\text{identifier}>$: This identifier must be associated with a template; the structured object of that template having arbitrary contents is created; its shape is determined in exactly the same way as in type case. Then the integer scalar value is created being a pointer to this structured object. The identifier specified after the function label is associated with this pointer.

3. The argument $EXPR.a$ is computed using the new association for the identifier. Its a posteriori type coincides with the a priori one.

4. The objects created during execution of the step 2 are destroyed.

5. The current association for the identifier specified after the function label is destroyed and the saved one (if any) is restored.

6. Operand $EXPR.a$ is returned.

H.6.5 VAR

This function returns an object using some identifier.

VAR ::= ( var <access> <identifier> )

Depending on the keyword the result is computed as follows:

1. The keyword is value

   (a) If the identifier is associated with a value, this value is returned.
(b) If the identifier is associated with a name, the result of dereferencing the name is returned. This result is a value with the same type and shape as the original name. Each element of the newly created value has the same contents as the one referred to by the corresponding element of the original name.

(c) If the identifier is associated with a couple, the value belonging to this couple is returned.

(d) Otherwise this results in an error.

2. The keyword is name

(a) If the identifier is associated with a name, this name is returned.

(b) If the identifier is associated with a couple, the name belonging to this couple is returned.

(c) Otherwise this results in an error.

3. The keyword is target

(a) If the identifier is associated with a target, this target is returned.

(b) If the identifier is associated with a name, the target having the same type, shape and contents is returned.

(c) If the identifier is associated with a couple, the target having the same type, shape and contents as the name belonging to this couple is returned.

(d) Otherwise this results in an error.

4. The keyword is pointer

(a) If the identifier is associated with a couple, the integer scalar value being a pointer to the value belonging to this couple is returned.

(b) Otherwise this results in an error.

H.6.6 SELECT

This function performs access to the field of structured object.

SELECT ::= ( SELECT <access> <identifier> <number> EXPR.p )

The operand must be a posteriori integer value; its elements are treated as pointers to structured objects having template $T$ associated with the identifier specified. The literal <number> defines the ordinal number $N$ of the field to be selected; The root of $T$ must have at least $N$ successors.

The elementary selection with respect to an element of the operand is defined as follows:

1. If the $N$-th successor of the root of $T$ is in its turn a root of some subtree, i.e. the corresponding field is structured, then the integer scalar value is obtained being a pointer to that field (in this case pointer must be specified as an access).
2. If the $N$-th successor is a leaf node, i.e. the corresponding field is primitive, then either a copy of this field is obtained (if the access value is specified) or the target/name referring to this field elementwise (if the access specified is target/name, respectively), or the pointer to this field (if the access is pointer).

Let $d_F$ be the common shape vector of all elementary selections: the empty vector if a structured field is selected or the label of the corresponding node of $T$ in the case of a primitive field. The function result is computed as follows:

$$d = \min \left( E(d^p, m), E(d^F, m) \right),$$

$$c_k = F(c^p_{T(k,m)})_{p(k,m)}, \ k < d,$$

where

$$m_i = \begin{cases} 1, & 0 \leq i < r(d^p) \\ 0, & r(d^p) \leq i < r(d^p) + r(d^F) \end{cases},$$

and $F(x)$ is the result of element selection using the pointer $x$.

### H.6.7 GLOBAL

This function creates a primitive or a structured object in the global heap memory.

GLOBAL ::= ( global MODE EXPR.e ... )

MODE ::= <type> | <identifier>

If an identifier is specified as a mode then it must be associated with a template. Operands $EXPR.e_i, 0 \leq i < n$, must be scalar values of a posteriori integer type.

The primitive value of the type specified (if type alternative is chosen as a mode) or the structured object of the template specified (if identifier alternative is chosen) having arbitrary contents is created in the global heap memory. The shape of this object is determined as follows:

$$d_i = c^e_i, \ 0 \leq i < n,$$

Then the integer scalar value is created being a pointer to newly created object. This pointer is returned as the function result.

### H.6.8 DISPOSE

This function disposes the object previously created by GLOBAL function. It always returns integer scalar value 0.

DISPOSE ::= ( dispose EXPR )

The operand must be an a posteriori integer scalar value being a pointer to the object allocated in the global heap memory by GLOBAL function.
H.6.9 LOCAL

This function creates a primitive or a structured object in the local heap memory.

\[
\text{LOCAL} ::= (\text{local MODE EXP}r.e ... )
\]

\[
\text{MODE} ::= \text{<type>} | \text{<identifier>}
\]

If an identifier is specified as a mode then it must be associated with a template. Operands \(E\text{XPR}r.e_i\), \(0 \leq i < n\), must be scalar values of a posteriori integer type.

The primitive or the structured object having arbitrary contents is created according to exactly the same rules as in \text{GLOBAL} function. The only difference is that this object is created in the local heap memory.

Then the integer scalar value is created being a pointer to newly created object. This pointer is returned as the function result.

H.6.10 MARK

This function marks the levels of locality of the local heap memory. It does not assume its argument to be precomputed.

\[
\text{MARK} ::= (\text{mark EXP}r.a)
\]

The function is executed as follows:

1. The state of the local heap memory is stored.
2. The argument \(E\text{XPR}r.a\) is computed. Its a posteriori type coincides with the a priori one.
3. All the objects created in the local heap memory while computing the argument are disposed.
4. The operand \(E\text{XPR}r.a\) is returned.

H.6.11 RAMP

This function returns a one-dimensional value containing arithmetic progression, whose parameters are determined by the operands.

\[
\text{RAMP} ::= (\text{ramp EXP}r.b EXP}r.e EXP}r.s)
\]

All the operands must be scalar values. Their a posteriori types are determined as follows:

\[
\hat{t}^b = \hat{t}^e = \hat{t}^s = \min(\text{real}, \max(t^b, t^e, t^s, \text{integer})).
\]

The shape and contents are as follows:

\[
d_0 = \left[\frac{c^e - c^b}{c^s}\right] + 1,
\]
\[
c_k = c^b + kc^s, \quad 0 \leq k < d_0.
\]
H.6.12 MONADIC

This is a function performing a data parallel monadic operation.

MONADIC ::= ( monadic <unary> EXPR.a )

The operand must be a value. Its a posteriori type is this:

$$\hat{t}^a = \min(t^S, \max(s, t^J)),$$

where $t^S$ and $t^J$ are the senior and junior admissible types of the operator $\langle$unary$,\rangle$, respectively.

The result is computed as follows:

$$d = d^a,$$

$$c_k = \odot a_k, \; k < d,$$

where $\odot$ is the operation determined by $\langle$unary$\rangle$.

H.6.13 DYADIC

This is a function performing a data parallel dyadic operation.

DYADIC ::= ( dyadic <binary> <mask> EXPR.l <mask> EXPR.r )

Both operands must be values. Their a posteriori types are as follows:

$$\hat{t} = \hat{t} = \min(t^S, \max(t^I, t^J)),$$

where $t^S$ and $t^J$ are the senior and junior admissible types of the operator $\langle$binary$\rangle$, respectively.

The result is computed as follows:

$$d = \min(e^l, e^r),$$

$$c_k = c^l_{p'(k)} \odot c^r_{p'(k)}, \; k < d,$$

where $\odot$ is the operation determined by $\langle$binary$\rangle$.

H.6.14 CHOICE

This function performs the data-parallel choice and is applicable both to the alternatives of any sort.

CHOICE ::= ( choice <mask> EXPR.s <mask> EXPR.t <mask> EXPR.f )

Operand $EXPR.$s must be a logical value, and $EXPR.$t and $EXPR.$f must be of the same sort. Their a posteriori types are determined as follows:

$$\hat{t} = \hat{t} = \begin{cases} \max(t^I, t^J) & \text{for values} \\ \min(t^I, t^J) & \text{for names and targets} \end{cases}$$

The result sort coincides with that of $EXPR.$t and $EXPR.$f.

$$d = \min(e^t, e^t, e^t),$$

$$c_k = \begin{cases} c^l_{p'(k)} & \text{if } c^l_{p'(k)} = true \\ c^l_{p'(k)} & \text{otherwise} \end{cases}, \; k < d.$$
H.6.15 REDUCE

This function reduces its operand by applying an associative commutative binary operation.

\[
\text{REDUCE} ::= ( \text{reduce} \ <\text{total}> \ <\text{mask}> \ \text{EXPR}.a )
\]

The operand must be a nonscalar value. Its a posteriori type is this:
\[
\hat{t}^a = \min \left( t^S, \max \left( t^a, t^t \right) \right),
\]

where \( t^S \) and \( t^t \) are the senior and junior admissible types of the operator \(<\text{total}>\), respectively.

The result is computed as follows:
\[
d = \hat{p}^a(d^a),
\]
\[
c_k = \bigoplus_{i < p^a(d^a)} s^a(k), \ k < d.
\]

where \( \bigoplus \) is the reduction generated by \(<\text{total}>\).

H.6.16 TRANSP

This function transposes its operand.

\[
\text{TRANSP} ::= ( \text{transp} \ <\text{number}> \ \text{EXPR}.a )
\]

Operand \( \text{EXPR}.a \) is an arbitrary object having the rank greater than unity; its a posteriori type coincides with the a priori one. The result sort is the same as that of \( \text{EXPR}.a \). Let us define a mapping
\[
\mathbf{T} : (a, N) \rightarrow b
\]
such that
\[
b_i = \begin{cases} a_i, & 0 \leq i < N \\ a_{i+1}, & N \leq i < r(a) - 1 \\ a_N, & i = r(a) - 1 \end{cases}
\]

Then the shape and contents of the result are determined as follows (\( N \) is the number expressed by the literal):
\[
d = \mathbf{T}(d^a, N),
\]
\[
c_k = \circ_{\mathbf{T}(k, N)}, \ k < d.
\]

H.6.17 SECT

This function returns a layer of its first operand computed by fixing one of the components of the multi-index.

\[
\text{SECT} ::= ( \text{sect} \ <\text{number}> \ \text{EXPR}.s \ \text{EXPR}.i )
\]
Operand $EXPR.s$ is a nonscalar of an arbitrary sort. Its a posteriori type coincides with its a priori one. Operand $EXPR.i$ is an a posteriori integer scalar value.

The result sort are the same as that of $EXPR.s$. This function first turns the number expressed by the literal (which we shall denote as $N$) into an auxiliary mask $m$:

$$m_j = \begin{cases} 1, & \text{if } j \neq N \\ 0, & \text{if } j = N \end{cases}, \quad 0 \leq j < r(d^s).$$

Then the shape and contents of the result are computed as follows:

$$d = P(d^s, m),$$

$$c_k = c_{l(k)}^s, \quad k < d,$$

where

$$l_j(k) = \begin{cases} E_j(k, m), & \text{if } j \neq N \\ c_j^s, & \text{if } j = N \end{cases}, \quad 0 \leq j < r(d^s).$$

### H.6.18 SLICE

This function slices a regular fragment from its first operand.

\[ \text{SLICE} ::= (\text{slice } \langle \text{number} \rangle \text{ EXPR.s EXPR.i}) \]

Operand $EXPR.s$ is a nonscalar of an arbitrary sort; its a posteriori type coincides with its a priori one.

Operand $EXPR.i$ is an a posteriori integer one-dimensional value.

The result sort is the same as that of $EXPR.s$. Its shape and contents are determined as follows ($N$ is the number expressed by $\langle \text{number} \rangle$):

$$d_j = \begin{cases} d_j^s, & \text{if } j \neq N \\ d_0^s, & \text{if } j = N \end{cases}, \quad 0 \leq j < r(d^s),$$

$$c_k = c_{l(k)}^s, \quad k < d,$$

where

$$l_j(k) = \begin{cases} k_j, & \text{if } j \neq N \\ c_j^s, & \text{if } j = N \end{cases}, \quad 0 \leq j < r(d^s).$$

### H.6.19 REPL

This function replicates one of its operands.

\[ \text{REPL} ::= (\text{repl } \langle \text{mask} \rangle \text{ EXPR.s EXPR.r } \ldots) \]

Operand $EXPR.s$ can be of any sort, type and rank. Operands $EXPR.r_i$, $0 \leq i < n$, are scalar values of a posteriori integer type, whose number $n$ must coincide with the number of bits in the mask. The result sort coincides with the sort of $EXPR.s$.

$$d_i = \begin{cases} c_i^t, & \text{if } c_i^t = \infty \\ c_i^t \cdot c_i^t, \quad \text{otherwise} \end{cases}, \quad 0 \leq i < n,$$

$$c_k = c_{p^t(k)}^s \pmod{d^s}, \quad k < d.$$
H.6.20 PACK

This function repacks its first operand into a different shape array.

PACK ::= ( pack EXPR.s EXPR.e ... )

The first operand is an arbitrary nonscalar. The function returns an object of the same sort as that of EXPR.s. Operands $EXPR.e_i$, $0 \leq i < n$, are scalar a posteriori integer values.

The result shape and contents are determined as follows:

$$d_i = c^e_i, \ 0 \leq i < n,$$

$$c_k = c^s_{l(k)}, \ k \prec d,$$

where $l(k)$ is the multi-index that provides the same sequential number (in terms of lexicographical ordering of indices while flattening an array) in the operand elements as multi-index $k$ in the result.

H.6.21 GATHER

This function gathers the first operand elements using the logical object provided by the second operand.

GATHER ::= ( gather <mask> EXPR.s EXPR.m )

Operand EXPR.s is an arbitrary nonscalar object. Operand EXPR.m is a logical value.

Let us introduce the following denotations: $A(v, x)$ is vector $v$ with the component $x$ appended; $R(v)$ is vector $v$ with the last component removed; and $L(v)$ is the last component of vector $v$: $A(R(v), L(v)) \equiv v$.

The result sort coincides with that of EXPR.s. The result shape and contents are determined as follows:

$$d = A\left(\hat{p}^s(d^s), \sum_{k \prec d^m} I(c^m_k)\right),$$

where

$$I(x) = \begin{cases} 1, & \text{if } x = true \\ 0, & \text{if } x = false \end{cases},$$

$$c_k = G (s^s(R(k)), c^m_{L(k)}), \ k \prec d,$$

where $G(A, B)$ is a vector computed by compression of array $A$ using logical array $B$ of the same rank. This vector is composed by elements

$$\{A_k \mid k \prec \min (d^A, d^B) : B_k = true\},$$

ordered according to the lexicographical order of their coordinates.
H.6.22 DIAG

This function slices a hyperdiagonal from its nonscalar operand.

\[
\text{DIAG} := ( \text{diag} \ <\text{mask}> \ \text{EXPR}.a )
\]

The only operand EXPR.a is an arbitrary nonscalar object; its a posteriori type coincides with the a priori one.

The result sort is the same as that of EXPR.a. Its shape and contents are as follows:

\[
d = A \left( \text{p}(d^a), \min_{0 \leq i < n} \text{p}^e(i(d^a)) \right),
\]

where \( n = r(\text{p}(d^a)) \) is the number of unity bits in the mask;

\[
c_k = s(\text{R}(k))q(k), \ k \prec d,
\]

where

\[
g_i(k) = L(k), \ 0 \leq i < n.
\]

H.6.23 TRANSFORM

This is a nonscalar analogue of indexing.

\[
\text{TRANSFORM} := ( \text{transform} \ \text{EXPR}.s \ \{ <\text{mask}> \ \text{EXPR}.t \} \ldots )
\]

Operand EXPR.s is an arbitrary nonscalar. All of EXPR.t_i, 0 \leq i < n, are a posteriori integer values. An equality \( n = r(d^a) \) must be satisfied.

The result sort is same as the one of EXPR.s.

\[
d = \min_{0 \leq i < n} e^h,
\]

\[
c_k = c_{i(k)}, \ k \prec d,
\]

where

\[
l_i(k) = c_{p+1(k)}, \ 0 \leq i < n.
\]

H.6.24 PART

This function extracts the real or imaginary part from its operand.

\[
\text{PART} := ( \text{part} <\text{selector}> \ \text{EXPR} )
\]

The operand may have any sort and is of the a posteriori complex type. The result type is real. Its sort and shape are the same as those of the operand. The contents of the result are determined as follows:

1. If the operand is a value then the result is computed by elementwise extraction of the real or imaginary part of the operand depending on the selector.

2. If the operand is a name or a target then each element of the result will refer to the real or imaginary part of the elementary value referred to by the corresponding (in terms of multi-index) element of the operand. If a target operand element contains the dummy reference, then the corresponding element of the result will contain the dummy reference, too.
H.6.25 COMP

This function composes its operands to form a single object.

\[
\text{COMP ::= ( comp <number> <mask> EXPR.1 <mask> EXPR.r )}
\]

The operands must be of the same sort. An equality \( r(e^l) = r(e^r) \) must be satisfied. The a posteriori types of the operands are as follows:

\[
\hat{p}^l = \hat{p}^r = \begin{cases} \max(t^l, t^r) & \text{for values} \\ \min(t^l, t^r) & \text{for names and targets} \end{cases}
\]

The result sort coincides with that of EXPR.1 and EXPR.r.

Let \( N \) be the number expressed by <number> and

\[
g^x = \begin{cases} 1, & \text{if } e^x_N = \infty \\ e^x_N, & \text{otherwise} \end{cases}
\]

Now the shape and contents of the result are as follows

\[
d_j = \begin{cases} \min(e^l_j, e^r_j), & \text{if } j \neq N \\ g^l + g^r, & \text{if } j = N, \ 0 \leq j < r(e^l) \end{cases},
\]

\[
c_k = \begin{cases} c^l_{p^l(k)}, & k_N < g^l \\ c^r_{p^r(q(k))}, & \text{otherwise} \end{cases}, \ k < d,
\]

where

\[
q_i(k) = \begin{cases} k_i, & i \neq N \\ k_i - g^l, & i = N, \ 0 \leq i < r(d). \end{cases}
\]

H.6.26 POL

This is evaluation of a polynomial of an arbitrary number of (nonscalar) variables.

\[
\text{POL ::= ( pol EXPR.c \{<mask> EXPR.v}\ldots )}
\]

Operand EXPR.c is a nonscalar value. All of \( EXPR.v_i, 0 \leq i < n, \) are values. An equality \( n = r(d^c) \) must be satisfied. The a posteriori types of the operands are as follows:

\[
\hat{c}^c = \hat{c}^{v_0} = \ldots = \hat{c}^{v_{n-1}} = \max(\hat{c}^c, \hat{c}^{v_0}, \ldots, \hat{c}^{v_{n-1}}, \text{integer}).
\]

The result is computed as follows:

\[
d = \min_{0 \leq i < n} e^{v_i},
\]

\[
c_k = \sum_{1 < d^c} c_i^c \prod_{0 \leq i < n} (c^v_{p^v(q(k))})^{l_i}, \ k < d.
\]
H.6.27 DISPLACE

This function displaces a pointer by some number of steps.

\[ \text{DISPLACE} ::= ( \text{displace} \ \text{MODE} \ <\text{mask}> \ \text{EXPR}.p <\text{mask}> \ \text{EXPR}.s ) \]

\[ \text{MODE} ::= \text{<type>} | \text{<identifier>} \]

Both operands must be values of a posteriori integer type. If an identifier is specified as a mode then it must be associated with a template.

The result is computed as follows:

\[ d = \min(e^p, e^s) , \]

\[ c_k = c_{p^r(k)}^p + M c_{p^r(k)}^s , \ k < d , \]

where \( M \) is the size in conventional units of memory of an object of the type specified (if type alternative is chosen as a mode) or of the template specified (if identifier alternative is chosen).

H.6.28 DISTANCE

This function evaluates a distance between two pointers in steps.

\[ \text{DISTANCE} ::= ( \text{distance} \ \text{MODE} \ <\text{mask}> \ \text{EXPR}.l <\text{mask}> \ \text{EXPR}.r ) \]

\[ \text{MODE} ::= \text{<type>} | \text{<identifier>} \]

Both operands must be values of a posteriori integer type. If an identifier is specified as a mode then it must be associated with a template.

The result is computed as follows:

\[ d = \min(e^l, e^r) , \]

\[ c_k = (c_{p^r(k)}^l - c_{p^r(k)}^r)/M , \ k < d , \]

where \( M \) is defined in the same way as in function DISPLACE.

H.6.29 TYPE

This function analyzes the type of its operand.

\[ \text{TYPE} ::= ( \text{type} \ \text{EXPR} ) \]

The operand is an arbitrary object.

The result of this function is an integer scalar value determined by the operand type as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>1</td>
</tr>
<tr>
<td>character</td>
<td>2</td>
</tr>
<tr>
<td>integer</td>
<td>3</td>
</tr>
<tr>
<td>real</td>
<td>4</td>
</tr>
<tr>
<td>complex</td>
<td>5</td>
</tr>
</tbody>
</table>
H.6.30 SHAPE
This function returns the shape of its operand.

SHAPE ::= ( shape EXPR.a )

The operand is an arbitrary object.

\[ d_0 = r(d^a), \]
\[ c_i = d_i^a, \quad 0 \leq i < d_0. \]

H.6.31 ASSIGN
This function always returns an integer scalar value 0. It is used to achieve a side effect, which is changing the contents of the values referred to by some name or target.

ASSIGN ::= ( assign EXPR.l <mask> EXPR.r )

Operand EXPR.l must be of sort name or target, and EXPR.r of sort value. Their a posteriori types coincide with the a priori ones. The action undertaken by the function is:

\[ c'_k \leftarrow c_{p^r(k)}, \quad k < d^t : p^r(k) < d^r, \]

where \( a \leftarrow b \) means the following. If \( a \) is a dummy reference then there is nothing to do, otherwise the contents of the scalar value referred to by \( a \) is replaced by \( b \) coerced to the type of that value.

H.6.32 CHANNEL
This function creates a channel and arranges a scope for it. It does not assume its argument to be precomputed.

CHANNEL ::= ( channel <identifier> EXPR.a <sort> <type> <number> )

The algorithm of the function evaluation is this:

1. The existing association for the identifier (if any) is saved.
2. The identifier is associated with a channel being a FIFO queue of primitive objects. Type, sort and rank specified are ascribed to the channel (the latter characteristic being defined by <number>). On creation, the channel is empty.
3. The argument EXPR.a is computed using the new association of the identifier. Its a posteriori type coincides with the a priori one.
4. Any objects that may remain in the channel are disposed.
5. The current association for the identifier is destroyed and the saved one (if any) is restored.
6. Operand EXPR.a is returned.
H.6.33  PUT
This function puts an object into a channel.

PUT ::= ( put <identifier> EXPR )

The identifier specified must be associated with a channel. The type ascribed to the channel determines the a posteriori type of the operand. Sort and rank of the operand must coincide with those ascribed to the channel. This function yields its operand as a result.

H.6.34  GET
This function gets an object from a channel and returns it as the result.

GET ::= ( get <identifier> )

The identifier specified must be associated with a channel. If the channel is empty then the function evaluation is suspended until an object appears in the channel.

H.6.35  SEQ
This function sequentially computes its arguments.

SEQ ::= ( seq EXPR ... )

The arguments are evaluated sequentially from left to right and must return a posteriori integer scalar values. If current operand is 0, the next one (if any) is selected. If the current operand is greater than 0, the function terminates returning the current operand value minus one. If all the operands have happened to be zeroes it returns zero as well. This provides for correct completion: any argument can terminate this function by returning 1, or even terminate any enveloping SEQ or LOOP (see Section H.6.38) by returning a proper positive value. Finally, if a negative value is returned by any one of the arguments, SEQ will terminate the f-program.

H.6.36  COMMA
This function computes its first argument, discards the result, then computes its second argument and returns the result.

COMMA ::= ( comma EXPR.d EXPR.r )

This function can be used to sequentialise two activities which yield some data object to be processed further, and which for that reason can not be placed in a SEQ list. When sequentialisation is not needed, function CHOICE can be used.
H.6.37  PAR

This function serves to parallelise a set of activities. As most of the functions, it assumes its arguments to be precomputed in parallel.

\[ \text{PAR} ::= (\text{par} \ \text{EXPR} \ldots) \]

The operands must be scalar values of a posteriori integer type. This function is introduced to support parallel evaluation of side-effecting functions. If all the operands are nonnegative, the function returns the maximum of these, otherwise it returns the minimum operand. This is performed for correct interpretation of completions.

H.6.38  LOOP

This function repeatedly computes its argument.

\[ \text{LOOP} ::= (\text{loop} \ \text{EXPR}) \]

The result returned by the argument every time it is being computed must be an a posteriori integer scalar value. If it is equal to zero, the loop continues. If it is positive, the loop terminates returning this value minus one. A negative integer returned by the argument will cause termination of the f-program.

H.6.39  SPAWN

This function executes a number of identical activities in parallel. It does not assume its last argument to be precomputed.

\[ \text{SPAWN} ::= (\text{spawn} \ <\text{identifier}> \ \text{EXPR}.a \ \text{EXPR}.n) \]

The operand \( \text{EXPR}.n \) must be a positive, a posteriori integer, scalar value (let us denote it as \( N \)). The algorithm of function evaluation is this:

1. The existing association for the identifier is saved.

2. \( N \) integer scalar values \( 1, \ldots, N \) are created.

3. The argument \( \text{EXPR}.a \) is computed \( N \) times in parallel, the identifier being associated with one (unique) of the values created at the previous step prior to each evaluation of the argument. All the results yielded must be a posteriori integer scalar values.

4. The saved associations for the identifier (if any) is restored.

5. If all the results yielded at the step 3 are nonnegative, the function returns the maximum of these, otherwise it returns the minimum. This is for correct interpretation of completions.
H.6.40  IF
This is a usual if-then-else construct. It does not assume the last two arguments to be precomputed.

IF ::= ( if EXPR.c EXPR.t EXPR.f )

Operand EXPR.c must be a logical scalar value. If it is true then EXPR.t is computed, otherwise EXPR.f. In either case the operand computed must be an integer scalar value, which is then returned.

H.6.41  CALL
This function invokes the subroutine pointed to by its operand (which must be a posteriori integer scalar value) and returns its result.

CALL ::= ( call EXPR )

H.6.42  COERCE
This function ensures correct inheritance of external data whenever its required attributes are statically known.

COERCE ::= ( coerce <identifier> EXPR.a <property> <type> <number> )

Let $t$ and $N$ be the type and the number expressed by <type> and <number>, respectively. The algorithm of the function evaluation is this:

1. The association for the identifier (which must exist) is saved.
2. Depending on the property specified, one of the following is done:

(a) The keyword is value
   i. If the identifier is associated with a value, this value is selected.
   ii. If the identifier is associated with a name, the result of dereferencing this name is selected.
   iii. If the identifier is associated with a couple, the value belonging to this couple is selected.
   iv. Otherwise this results in an error.

Then the identifier is associated with a copy of the value selected which is coerced to the type $t$. The rank of this value must be equal to $N$.

(b) The keyword is name
   i. If the identifier is associated with a name, this name is selected.
   ii. If the identifier is associated with a couple, the name belonging to this couple is selected.
   iii. Otherwise this results in an error.
Then the identifier is associated with the name selected. The rank of this name must be equal to $N$ and its type must coincide with $t$.

(c) The keyword is target
   i. If the identifier is associated with a target, this target is selected.
   ii. If the identifier is associated with a name, the target having the same type, shape and contents is selected.
   iii. If the identifier is associated with a couple, the target having the same type, shape and contents as the name belonging to this couple is selected.
   iv. Otherwise this results in an error.

Then the identifier is associated with the target selected which is coerced to the type $t$. The rank of this target must be equal to $N$.

(d) The keyword is couple
   i. If the identifier is associated with a couple then it becomes associated with the same couple once again. The rank of the objects forming this couple must be equal to $N$ and their type must coincide with $t$.
   ii. Otherwise this results in an error.

3. The argument EXPR.a is computed using the new association for the identifier. Its a posteriori type coincides with the a priori one.

4. The objects created during execution of the step 2 are destroyed.

5. The current association for the identifier is destroyed and the saved one is restored.

6. Operand EXPR.a is returned.

H.7 Examples

In order to help the reader to understand the above f-code definition we conclude this report by providing an f-expression that implements a parallel bubble sorting algorithm. To define the algorithm, here is the appropriate EVAL code:

```f-code
procedure BUBBLE use integer vector <>S
begin
  let ORD be same[0] < same[1] c ? same ! same[<,>];
  loop limit trunc((extent S + 1) / 2) repeat
  end_loop;
end_procedure
```

Now let us present the respective f-expression, which includes checking the attributes of the global variable S inherent in the EVAL semantics.
(scope expression $size
  (sect 0
   (shape
     (var value $1)
   )
   (const 0)
  )
  (seq
   (par
    (monadic neg
     (dyadic ne
      (type
       (var name S)
      )
      (const 3)
     )
    )
    (monadic neg
     (dyadic ne
      (scope link $1
       (shape
        (var value S)
      )
      (call $size)
     )
     )
     (const 1)
    )
   )
  )
)(scope expression ORD
  (choice
   10 (dyadic lt
    (sect 0
     (var value $LHP)
     (const 0)
    )
    (sect 0
     (var value $LHP)
     (const 1)
    )
   )
   11 (var value $LHP)
   11 (slice 0
    (var value $LHP)
(ramp
  (bind $1 $LHP
    (call $size)
  )
  (const 0)
  (const -1)
)

(scope enreference $loopcnt
  (monadic trunc
    (dyadic div
      (dyadic add
        (bind $1 S
          (call $size)
        )
        (const 1)
      )
      (const 2)
    )
  )
)

(loop
  (seq
    (scope dereference $LHP
      (pack
        (var name S)
        (monadic trunc
          (dyadic div
            (bind $1 S
              (call $size)
            )
            (const 2)
          )
          (const 2)
        )
      )
      (const 2)
    )
    (assign
      (var name $LHP)
      11 (call ORD)
    )
  )

  (scope dereference $LHP
    (pack
      (slice 0
        (var name S)
      )
  )
(ramp
  (const 1)
  (bind $1 S
      (call $size)
    )
  (const 1)
  )
)
(monadic trunc
(dyadic div
  (dyadic minus
    (bind $1 S
      (call $size)
    )
    (const 1)
    )
    (const 2)
  )
  (const 2)
  )
)
(assign
  (var name $LHP)
  11 (call ORD)
)
)
(assign
  (var name $loopcnt)
  (dyadic minus
    (var value $loopcnt)
    (const 1)
  )
)
(choice
  (dyadic eq
    (var value $loopcnt)
    (const 0)
  )
  (const 2)
  (const 0)
)
)
)
Bibliography


