VMPP: A Virtual Machine for Parallel Processing

Edmond C. Loyot, Jr.
Technical Report No. CS-92-30

VMPP: A Proposal for a Virtual Machine for Parallel Processing

Edmond C. Loyot, Jr.
Department of Computer Science
University of Virginia, Charlottesville, VA
ecl2v@virginia.edu
September 18, 1992

Abstract

The field of parallel processing is young and rapidly evolving. Consequently, there is a great diversity of languages and architectures. To make matters worse these languages and architectures often become obsolete at a rapid pace. In this environment, portability becomes an extremely important issue.

Unfortunately, most parallel languages are not portable. This portability problem can be solved using a virtual machine approach. In this approach, front-end translators translate various parallel source languages into code for a virtual machine. Back-end translators translate the virtual machine code into executable codes for a variety of parallel architectures.

The Virtual Machine for Parallel Processing (VMPP) is a proposal for just such a virtual machine. VMPP is designed to provide portability for a variety of high-level parallel programming languages without drastically sacrificing performance. It accomplishes this by defining a graph-based intermediate representation and a data-driven execution model.

Table of Contents

1. Introduction

The field of parallel processing is young and rapidly evolving. Consequently, there is a great diversity of languages and architectures, and no consensus on which are best, even for a given problem domain. To make matters worse, these languages and architectures become obsolete at a rapid pace. This situation is likely to continue for the foreseeable future.

In this environment, portability becomes an important issue. Unfortunately, most parallel programming languages are not portable. These languages lack portability because either their underlying model of computation reflects the architecture for which they were designed, or their compilers are so complex and machine dependent that they are extremely difficult to port to other architectures. Granularity of computation further complicates the situation. In most systems the granularity of computation is either locked in by the underlying programming model, or specified by the programmer. In either case, this makes porting the code to a machine whose processors have a different ratio of communication to computation very difficult.

The portability problem can be solved using a virtual machine approach. In the virtual machine approach, front-end translators are constructed for each parallel language. These front-ends translate the parallel language code into code for a virtual machine. A back-end translator is constructed for each parallel architecture. These back-ends translate the virtual machine code into executable code for their respective architectures. In order to be successful, a virtual machine must have the following characteristics, 1) it must be expressive enough to represent all programming constructs in its input languages 2) it must be implementable on a variety of parallel architectures, and 3) the machine language translations of the virtual machine code must execute efficiently on the architectures under consideration.

The Virtual Machine for Parallel Processing (VMPP) is a proposal for a general purpose virtual machine to be used in parallel processing systems. It is designed to provide portability for a variety of high-level parallel programming languages without drastically sacrificing performance. VMPP accomplishes this by defining a graph-based intermediate representation and a data-driven execution model. This intermediate representation is a suitable translation target for a variety of high-level parallel programming languages and can be translated into efficient executable code for a large class of parallel architectures. VMPP accomplishes all this because of the expressive power and flexibility of its intermediate representation.

This paper is structured as follows. Section 2 contains a detailed discussion of the portability problem. Section 3 outlines the general virtual machine approach. Section 4 describes VMPP in detail, including a design rationale. Section 5 provides a justification of the VMPP approach. Section 6 describes related work and section 7 details our research agenda.

2. The Portability Problem

VMPP addresses the lack of portability in parallel software. This problem is discussed below, after some background on the current state of parallel computing.

2.1. Current State of Parallel Computing

The term parallel architecture encompasses a vast array of machines, all designed for parallel computation of one sort or another. There are literally hundreds of parallel machine designs, some only on paper, some used as research tools, and others available commercially, with new proposals appearing all the time. These designs are very diverse, supporting a variety of computation models.

The primary reason for such diversity is that the field is still young and rapidly evolving. Consequently, there is no consensus on which architecture is best, or even which architecture is

best for a given problem domain. Best, in this case, means fastest. This environment leads to a great deal of competition to produce faster architectures, both for general purpose parallel computing and for problem-domain-specific parallel computing. Researchers in industry and academia are busily producing new and often innovative parallel computer designs, as well as making incremental improvements to old designs, resulting in a proliferation of parallel computer designs. Another reason for the diversity is that old designs tend to stay around, even when better designs are available. This happens because some group has made a substantial investment of time or money in the design, and is therefore reluctant to abandon it. These reasons are unlikely to change in the near future and therefore argue strongly for continuing architectural diversity in the parallel computing field.

The set of languages that can be categorized as parallel programming languages is as large and diverse as the set of parallel architectures. The reasons for language diversity are analogous to those for architectural diversity. Since the field of parallel programming is still in the early stages of development, there is little agreement on which languages are best, or even what metrics should be used. Consequently, each group of parallel computer users has its own metrics for determining which parallel language is best. For example, since parallel machines are infamously difficult to program, the most important metric for one group might be ease-of-use. For another, execution speed of programs may be the only important metric. Other groups may be concerned with reliability, real-time capability, and so on. As a consequence new languages are constantly being developed, resulting in a proliferation of diverse parallel programming languages.

Another reason for the proliferation and diversity of parallel languages is that users are generally unwilling to switch to new (and possibly better) programming languages. Consider, that FORTRAN, one of the very first high-level programming languages, is still quite popular. Since these reasons are likely to persist, so will the proliferation and diversity of parallel programming languages.

The most important points are 1) the set of parallel architectures is very large and diverse, 2) so is the set of parallel programming languages, and 3) this situation is likely to persist for the foreseeable future.

2.2. Portability and Parallel Programming

Portability is the ability to easily move software from one computer to another. When programmers invest the time and effort to develop a piece of software, they want it to be portable. Portable software is available to the widest possible set of users, allowing the maximum number of people to benefit from the initial investment of time and effort. Even if the software will only be used by a single group, portability is still desirable. Suppose the group has several different computers. They will want the ability to run the software on any of their computers, in case one breaks down or is heavily loaded. Even if the group has only one kind of computer, they may someday want to upgrade to a different machine. With portable software, they will be free to choose from a wider range of machines than would otherwise be possible.

Portability is even more important for parallel software. As mentioned above, parallel architectures evolve rapidly. This means that this year's premiere architecture is next year's dinosaur. Because of this proliferation and diversity, an organization is apt to upgrade or change parallel computers more frequently than sequential computers. Another consideration is that writing parallel software is much more difficult than writing sequential software. Parallel programming has all the inherent difficulties of sequential programming, plus the added cognitive burden of managing the parallel aspects of the program. For example, many parallel languages require the programmer to explicitly manage the communication, synchronization and distribution of the pro-

gram's parallel elements. This added complexity makes parallel programming considerably more difficult. As a result, the time and effort invested in software development is greater for a parallel system, making portability more important for parallel software than for sequential software.

The problem is that parallel software is not portable. To see why, we must first categorize parallel programming languages on a scale from low-level to high-level. The most low-level parallel languages require programmers to explicitly manage every aspect of parallelism using unstructured primitive operations. For example, a low-level language using the message passing model of parallelism, requires the programmer to manage all communication, synchronization and distribution using send and receive. A low-level language, based on the shared memory model of parallelism, requires programmers to manage synchronization and communication using semaphores or test-and-set operations. Low-level languages, such as these, are the assembly languages of parallelism. The most high-level parallel languages hide the details of parallelism from the programmer. In a high-level parallel language, the programmer is aware that the program will be executed in parallel, but does not have to explicitly manage all aspects of the program's parallelism. Existing parallel languages run the gamut from very high-level to very low-level, with most falling somewhere in between.

Low-level parallel programming languages are not portable because they closely reflect the architecture for which they were designed. This makes it difficult to build efficient implementations on other parallel architectures. For example, a low-level parallel language, such as C with shared memory primitives, is hard to implement efficiently on a hypercube architecture. Because of the difficulty, low-level languages are usually implemented on only a single architecture. Consequently, when a parallel program written in one of these low-level languages must be ported to a different architecture, it must be rewritten in a language designed for that architecture.

Although high-level parallel programming languages present a more abstract model of computation, they are often just as architecture dependent as their low-level counterparts. High-level languages present two obstacles to portability. First, although the model of computation used by the language is abstract, it may still be inherently difficult to implement on more than a small class of architectures. Accordingly, the language is implemented on only a few machines and is thus not portable. Second, compilers for high-level parallel languages are usually very complex and difficult to implement. This is a serious impediment to portability because of the large number and diversity of parallel architectures. Implementing a complex, architecture-specific compiler for more than a handful of parallel architectures is a daunting task. As a result, most high-level languages run on a single or small group of parallel architectures.

A final portability consideration is granularity. Granularity of computation refers to the amount of computation performed by a parallel program element between communications and synchronizations. All low-level and many high-level parallel programming languages require the programmer to specify the basic granularity of computation in a program. Although two parallel machines may have similar architectures, their communication and computation speeds may be very different. For example, the Intel iPSC/2 and iPSC/860 have very similar architectures, distributed memory using the same communication network, yet the computation speed of the iPSC/860 nodes is much higher. A program written for the iPSC/2 will run on the iPSC/860, but it may be inefficient because of the difference in node computation speed between the two machines. To get the best efficiency, the programmer would have to rewrite the program to change the granularity, or build some scheme into the code to dynamically change the granularity at run time, usually not a trivial task. Thus, parallel programs that require the programmer to specify the granularity of computation are usually not portable even among machines using similar architectures.

In summary, portability is an important quality in parallel software. Current parallel software lacks portability because most parallel programming languages are architecture dependent in one way or another. Granularity considerations further complicate the portability situation.

3. The Shape of a General Solution

A virtual machine approach provides a general solution to the portability problem. In this section we will explain the general concept of a virtual machine, then show how a virtual machine can be used to solve the portability problem. Some general characteristics of a good virtual machine solution are also discussed.

3.1. The Virtual Machine Approach

A virtual machine is an abstract computer. Like an actual computer, it can perform computations by executing instructions in the form of a program. However, because it is an abstraction, it is not associated with any specific instance of a computer, and its implementation is hidden from the user. The virtual machine concept has been used for years in the areas of compilers, operating systems, and programming languages.

A parallel language lacks portability either because its parallel programming model is difficult to implement on more than one class of architectures, or because its compiler is very complex and architecture dependent, or both. Thus, providing portability for *N* parallel languages across *M* parallel architectures requires building *N*M* implementations. The number of implementations can be reduced to *N+M* by designing a virtual machine which models the essential aspects of parallel computing. The virtual machine masks the differences between the various parallel architectures. Using the virtual machine approach, a single front-end translator can be provided for each parallel language, and a single back-end translator can then be constructed for each parallel architecture. Each front-end translates its associated parallel language into instructions for the virtual machine. Each back-end translates virtual machine instructions into executable code for its associated parallel architecture, thus reducing the number of implementations to *N+M*. This concept is illustrated in Figure 1, which shows how portability can be achieved for languages using the data-parallel, shared-memory, message-passing, and dataflow models across architectures using the hybrid (a SIMD/MIMD composite), DMMP (Distributed Memory Multiprocessor), SMMP (Shared Memory Multiprocessor), and dataflow architectures. This type of approach was first outlined by Steel in 1960 [1].

3.2. Characteristics of a Good Solution

In order to provide a good solution to the problems mentioned above, a virtual machine must have the following characteristics: 1) expressibility, 2) implementability, and 3) efficiency.

Expressibility refers to a language's ability to express various programming constructs. The language defined by the virtual machine must have good expressibility in order to represent all programming structures available in the parallel languages under consideration. This ensures that it is a suitable target language for the front-ends mentioned previously. Furthermore, the virtual machine language must also provide the ability to express varying levels of granularity in a single program. This quality is necessary to provide a complete solution to the portability problem.

Implementability indicates whether a language is suitable for implementation on a variety of architectures. It is important because translations from the virtual machine language to the machine language of the architectures under consideration must exist. Implementability ensures that the virtual machine language is an appropriate source language for the back-ends described above.

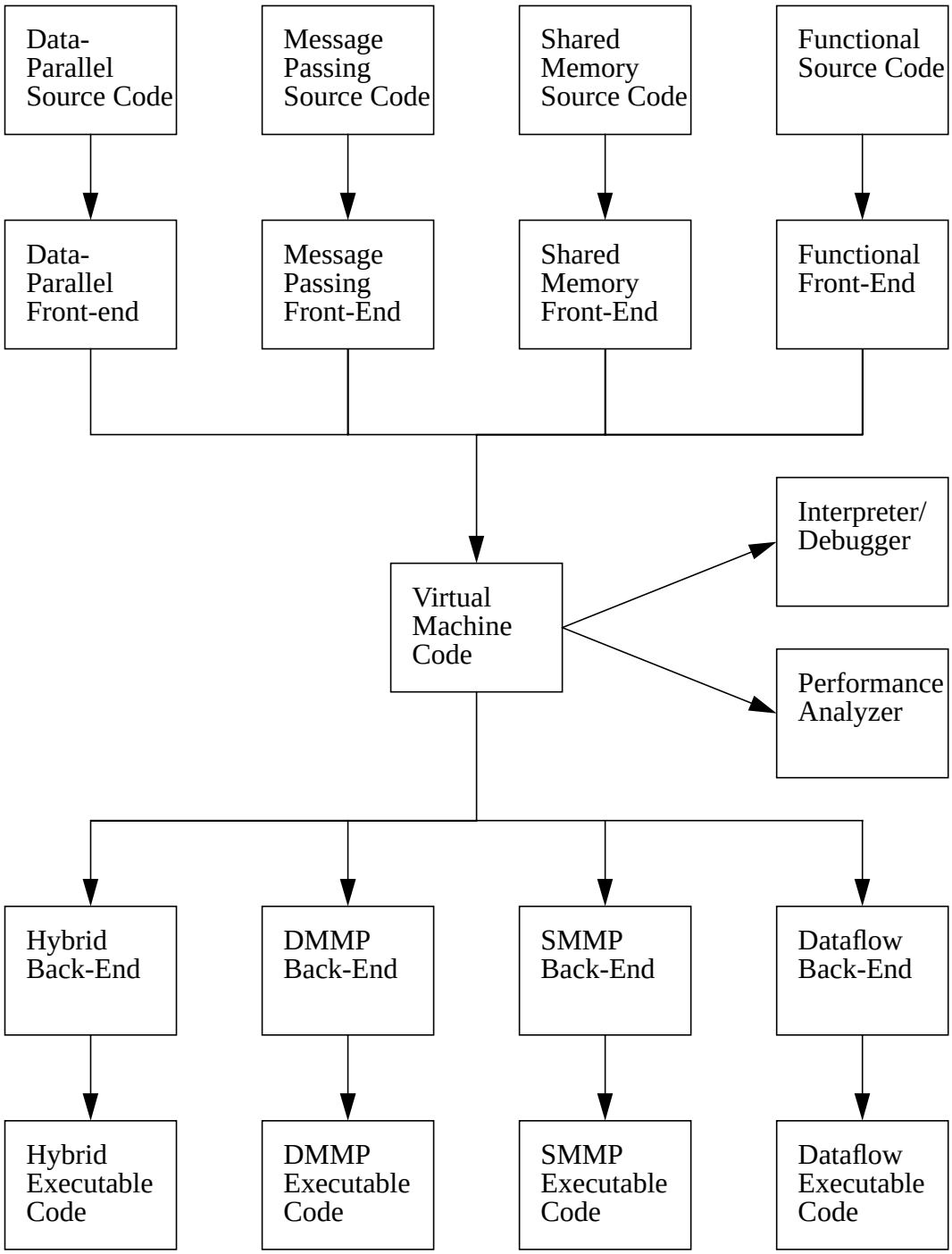


Figure 1. The Virtual Machine Approach.

Efficiency is a measure of how well the machine language translations of the virtual machine code execute on the architectures under consideration. These translations must execute efficiently because if performance is severely degraded, the solution will not be acceptable. There will, of course, be some overhead involved in the virtual machine approach. However, we believe that most users are willing to suffer some performance penalty for the benefits of portable, architecture-independent code, as long as the penalty is not too high. Consider the fact that although hand coded assembly language is often faster than compiled code, relatively few users opt for the difficult job of programming in assembly language. The key to generating efficient code is the ability to preserve information provided by the user at the source-language level. This information can be important when the back-ends generate code. This information must somehow be preserved in the virtual machine language so that it is available to the back-end translators.

4. VMPP: A Virtual Machine for Parallel Processing

In this section we present our proposal for a Virtual Machine for Parallel Processing (VMPP). We first describe the language and execution model and then give a rationale for some of the features incorporated in the design. The VMPP design presented here is the result of preliminary research and is thus incomplete and subject to revision as our research agenda is carried out.

4.1. VMPP Design

VMPP consists of a graph-based virtual machine language and a data-driven execution model. VMPP program graphs consist of nodes, arcs, and tokens. The basic design is motivated by the dataflow model of Dennis [2].

VMPP program graphs contain two kinds of nodes: computation nodes, and memory nodes. Computation nodes represent some sequence of computations. A computation node can be connected to other computation nodes by directed arcs which represent data dependencies. An arc running from computation node A to computation node B, indicates that node B is data dependent on node A.

Memory nodes represent a segment of memory. Computation nodes and memory nodes can also be connected by directed arcs. An arc running from a memory node to a computation node indicates the computation node reads data from the memory node. An arc running from a computation node to a memory node indicates the computation node writes data to the memory node.

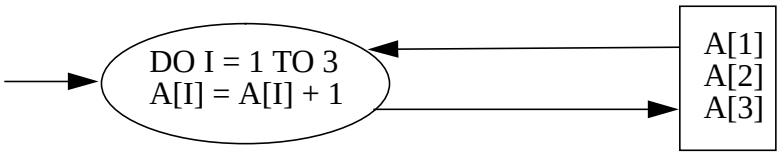
Tokens are used to indicate that a data dependency has been satisfied and may also carry data. A token present on an arc that connects two computation nodes indicates that the data dependency between them has been satisfied. The token may or may not carry the data used to satisfy the dependency. The nodes and arcs form a directed graph. When augmented by tokens, this graph represents a parallel computation.

In the VMPP execution model, any computation node having tokens on each of its input data-dependency arcs can perform its computation. The computation in turn generates new tokens on the node's output arcs, allowing other nodes to execute. The computation may also read and write data in connected memory nodes. A program is executed by placing tokens on the input arcs of the program's root node or nodes. When the root node executes, tokens are created on its output arcs. Since these output arcs are the input arcs of other nodes, some of these other nodes can execute. This process is repeated until no more nodes are able to execute, at which point the program is complete.

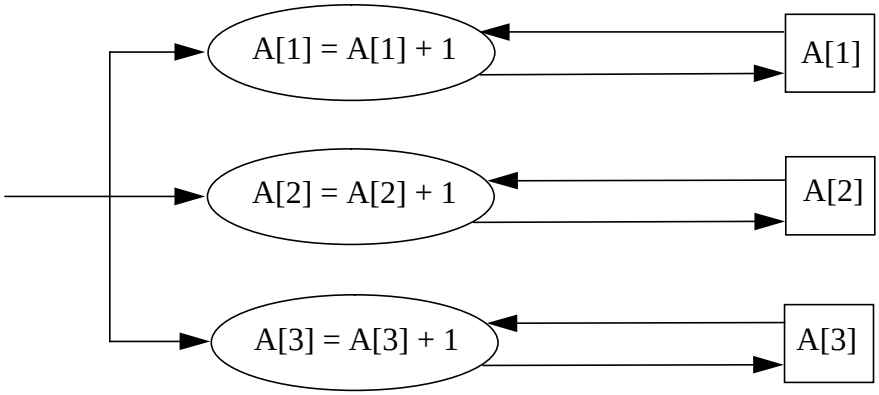
This model is data driven because a node can only execute when all its data dependencies have been met. It exhibits parallelism because all nodes with tokens on each of their input arcs can be executed in parallel. Figure 2(c) shows a VMPP program graph for the FORTRAN program segment shown in Figure 2(a).

```
DO I = 1 TO 3
    A[I] = A[I] + 1
```

(a)



(b)



(c)

Figure 2.  = Computation Node and  = Memory Node.

We extend this basic model by adding the notion of composite and atomic nodes. A composite computation node represents a computation that is made up of several smaller computations. A composite computation node is in fact a subgraph of computation nodes, with each node of the subgraph representing one of the smaller computations. For example, consider the FORTRAN program segment in Figure 2(a). This program segment can be represented by a single composite computation node and a single memory node, as is shown in Figure 2(b). A composite node is used since there are really several separate operations being performed.

An atomic computation node represents a single computational operation. For example, the statement A[1] = A[1] + 1 in the FORTRAN segment of Figure 2(a) can be represented by an atomic computation node as is shown in Figure 2(c).

Memory nodes may also be either composite or atomic. A composite memory node represents a segment of memory that consists of several subsegments. For example, a one dimensional FORTRAN array of reals could be represented by a composite memory node since the array is actually a group of several individual memory elements (see Figures 2(a, b)).

An atomic memory node consists of a single memory segment. For example, a single floating point memory segment would be represented by an atomic memory node as in Figure 2(c).

A composite node in a VMPP program graph can be expanded. Node expansion is defined as the replacement of a composite node by the sub-nodes it represents. Both composite computation nodes and composite memory nodes can be expanded. For example, the composite computation node of Figure 2(b) can be replaced by the computation node subgraph of Figure 2(c). Alternatively, the memory node representing the one dimensional FORTRAN array mentioned above could be expanded into several memory nodes, each representing a single element of the array as in Figures 2 (b, c). Atomic nodes cannot be expanded since they consist of only a single operation or memory segment.

Annotations are another extension to the basic model. Annotations are simply meaningful comments that can be attached to nodes or arcs. For instance, consider a composite computation node that represents a vector operation. This node could be annotated to indicate it represents a vector operation. A back-end for a machine containing vector processors could translate this composite node as a single vector processor instruction. A back-end for a machine without vector nodes could simply ignore the annotation.

4.2. Design Rationale

VMPP is designed to provide solutions to the problems of portability and language heterogeneity. The model presented above incorporates features designed to solve specific aspects of these problems.

Our first concern was to find an architecture-independent means for expressing parallel programs. Our choice of a graph-based intermediate language and a data-driven execution model was motivated by the dataflow model. This model has several advantages: 1) parallelism is easy to detect: any node with all of its input tokens present can be executed in parallel, 2) synchronization is implicit in the firing rules for the nodes, 3) data-dependence analysis and techniques for removing false dependencies are well understood [11], and 4) techniques for translating procedural languages into dataflow graphs have been developed [3]. Because of these reasons, traditional dataflow seemed like a good starting point for our research.

Unfortunately, the traditional dataflow model has several shortcomings as an intermediate language for parallel processing. First, traditional dataflow is a deterministic model. This presents a problem since several of our source languages are non-deterministic. Also, the dataflow model has no concept of a modifiable memory. However, most of our input languages and all or our target architectures use the concept of a modifiable memory. Thus, an intermediate-language model that includes some representation of a modifiable memory would make the translation process easier.

Memory nodes were included in the VMPP model to address these problems. They allow computation nodes to retain state between executions, thus providing a mechanism for non-deterministic program execution. Memory nodes also provide a representation for a modifiable memory which should simplify the translation process.

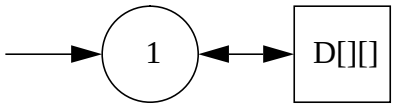
Traditional dataflow has another shortcoming. All nodes in a traditional dataflow graph represent small-grain computations. Granularity of computation refers to the amount of computation performed by a node. Conceptually, granularity can run from small grain, where a node computes a single instruction such as add or compare, to large grain, where a node might compute an entire computationally intensive function like an FFT. When only small-grain computations are

used, the overhead of collecting the data and scheduling the computation may outweigh the benefit of executing the computation in parallel.

Composite nodes address this problem. In order for VMPP to be architecture independent, it must run on machines requiring various computation-grain sizes. Composite nodes can be used to match the granularity of VMPP program graph nodes to the granularity of the parallel machine. When a high-level program is translated into a VMPP program graph, the large-grain computations become composite nodes. Most large-grain computations consist of several medium-grain computations. These medium-grain computations become nodes in the subgraph represented by the composite node. This is repeated until the smallest supported grain size is reached. If the VMPP program is then compiled for a large-grain architecture, the composite nodes can be translated directly into executable code. If the program is compiled for a smaller granularity architecture, the composite nodes can be expanded until the granularity of the program matches the granularity of the architecture.

Consider the FORTRAN code in Figure 3 (b). This is a Gaussian-elimination code for three equations in three variables. The variable PR is the pivot row. The outer loop selects the pivot row, the middle loop iterates over the remaining rows, and the inner loop iterates over the elements in the row specified by the middle loop. The largest-grain computation in this example is the entire code segment, thus this code is represented by a single composite computation node in the VMPP program graph. Analogously, the largest memory segment in this example is the memory used for the program's data structures, thus this memory segment is represented by a single composite memory node. Since this memory node is both read and written by the computation node, the VMPP program graph will contain arcs connecting these nodes. This top-level VMPP program graph is given in Figure 3 (a). The single computation node is labeled as node 1 and corresponds to the code in Figure 3 (b). Node 1 contains a single input data-dependency arc which is used to start the program. Since the program uses only one data structure, the 3x4 array *D*, the memory node contains only array *D*. Node 1 is connected to the memory node by two arcs, one going from the memory node to node 1, indicating that node 1 reads data from the memory node, and one going from node 1 to the memory node, indicating that node 1 writes data to the memory node. These arcs are abbreviated by the single arc with arrows on both ends shown in the diagram. Since node 1 is a composite node, it can be expanded into a subgraph. This subgraph is shown in Figure 3 (c) with the code corresponding to nodes 2 and 3 given in Figure 3 (d). This process can be continued (Figures 3 (e, f)), until a final level of granularity is reached (Figures 4 (a, b) and 5). As the computation nodes read and write smaller and smaller parts of the data structure, the composite memory nodes are also expanded. Note that in Figure 4 (b) the read arcs and memory nodes for row 1 of array *D* have been omitted; this was done in order to make the graph more readable. Also, since the value of variable *M* can be carried in a token along one of the data-dependency arcs, there is no memory node for *M*.

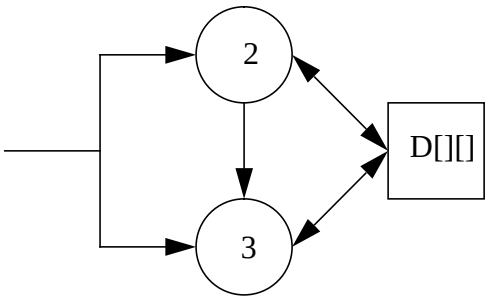
Once a source program has been converted to a VMPP program graph by the front-end, the back-end can match the granularity of the computation nodes to that of the target machine. The back-end knows the details of the target machine. For example, if the target machine is a DMMP, the back-end will know that the machine uses distributed memory, how many processors the machine has, how much memory each processor has, the speed of the processors and the details of the communication links between the processors. Using this information and the VMPP program graph it can select an appropriate granularity as follows. First, the graph is examined at the smallest level of granularity. The back-end calculates the communication-to-computation ratio for each computation node by first measuring the amount of computation done by the node, then



(a)

```
DO PR = 1, 2
    NR = PR + 1
    DO I = NR, 3
        M = D[I][PR] / D[PR][PR]
        DO J = PR, 4
            D[I][J] = D[I][J] - (M * D[PR][J])
```

(b)



(c)

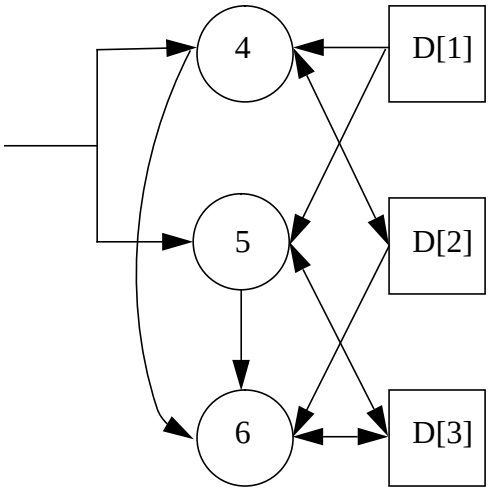
2 =

```
DO I = 2, 3
    M = D[I][1] / D[1][1]
    DO J = 1, 4
        D[I][J] = D[I][J] -
                  (M * D[1][J])
```

3 =

```
DO I = 3, 3
    M = D[I][2] / D[2][2]
    DO J = 1, 4
        D[I][J] = D[I][J]
                  (M * D[2][J])
```

(d)



(e)

4 =

```
M = D[2][1] / D[1][1]
DO J = 1, 4
    D[2][J] = D[2][J] -
              (M *D[1][J])
```

5 =

```
M = D[3][1] / D[1][1]
DO J = 1, 4
    D[3][J] = D[3][J] -
              (M *D[1][J])
```

6 =

```
M = D[3][2] / D[2][2]
DO J = 2, 4
    D[3][J] = D[3][J] -
              (M *D[2][J])
```

(f)

Figure 3. VMPP Program Graphs and Node Definitions.

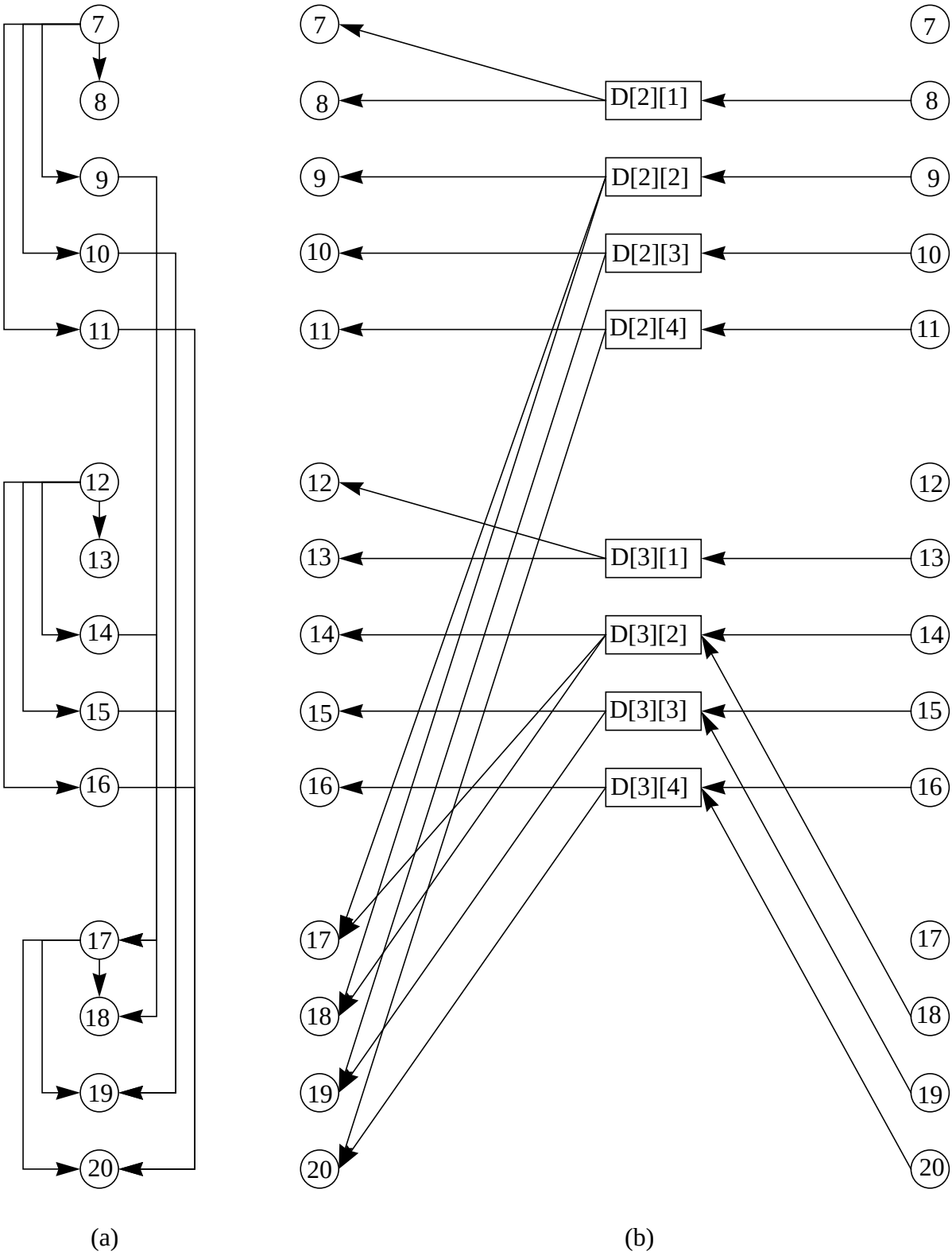


Figure 4. VMPP Program Graph for Code in Figure 3 (b).

(7) = M = D[2][1] / D[1][1]

(8) = D[2][1] = D[2][1] - (M * D[1][1])

(9) = D[2][2] = D[2][2] - (M * D[1][2])

(10) = D[2][3] = D[2][3] - (M * D[1][3])

(11) = D[2][4] = D[2][4] - (M * D[1][4])

(12) = M = D[3][1] / D[1][1]

(13) = D[3][1] = D[3][1] - (M * D[1][4])

(14) = D[3][2] = D[3][2] - (M * D[1][4])

(15) = D[3][3] = D[3][3] - (M * D[1][4])

(16) = D[3][4] = D[3][4] - (M * D[1][4])

(17) = M = D[3][2] / D[2][2]

(18) = D[3][2] = D[3][2] - (M * D[2][2])

(19) = D[3][3] = D[3][3] - (M * D[2][3])

(20) = D[3][4] = D[3][4] - (M * D[2][4])

Figure 5. Node Definitions for the VMPP Program Graph in Figure 4.

measuring the amount of communication that must be done to get the data needed by the node. These numbers form the communication-to-computation ratio. This ratio is then compared to the speed of the processors relative to the speed of the communication links. If the nodes at this level are of too small a grain size, the back-end can look at the nodes at the next level of granularity. This continues until the granularity of the computation matches the granularity of the architecture.

A final shortcoming of the dataflow model is that it cannot represent all the information available in a high-level source language. For example, consider a high-level language that contains vector operations. A vector operation, such as adding two vectors, can be represented by a dataflow graph consisting of the appropriate nodes and arcs. However, the information that this graph actually represents a single high-level operation is not apparent from the dataflow graph, thus this information as been effectively lost.

Node and arc annotations address this problem. These annotations allow all the information available in a high-level source program to be preserved in a VMPP program graph. For example, a composite computation node that models the vector operation mentioned above could be annotated to indicate it contains a vectorizable operation. If the program were then translated to a multiprocessor in which some processing elements were vector processors, the compound node could be executed as a single operation on one of the vector processing elements. If the program were translated to a multiprocessor in which the nodes were standard sequential processors, the composite node could be expanded and each part of the vector operation could be executed on a different processor. This scheme is flexible and powerful. It ensures that information provided in a high-level source program is preserved when the program is translated into a VMPP program graph. This information can be valuable when the VMPP program graph is translated into executable code for a particular architecture. This is particularly true for heterogeneous systems.

5. Why VMPP Will Work

As mentioned in section 3, a good virtual machine solution to the portability problem must have the characteristics of expressibility, implementability and efficiency. In this section we show that VMPP has each of these characteristics.

5.1. Expressibility (Front-End Translations)

To demonstrate that VMPP is sufficiently expressive, we will show how key programming constructs from several high-level parallel programming languages can be translated into VMPP program graphs. In order to make our arguments convincing, the languages we have selected represent a diverse group of parallel programming paradigms. The paradigms represented are the data-parallel model [6], the object-oriented model and the functional model [7]. We have chosen these models because they are dissimilar and each has a following in today's parallel programming community. The languages chosen and the techniques for translating their key constructs are detailed below.

5.1.1. Basic Translation Process

For clarity we will first outline the basic translation process, elements of which are common to all our example languages. Discussions on how any unique features of our example languages affect the basic process are presented in the appropriate sub-section. The process being outlined here is preliminary. We present it only to illustrate that the translations are possible. Determining the steps of the actual translation process, which will differ for each language under consideration, will be part of the research effort.

The translation process is iterative, with each iteration producing a complete VMPP program graph whose nodes represent a different level of granularity from the graphs produced by

previous iterations. The graph produced by any particular iteration is related to previous graphs in a hierarchical manner. Thus, the result of the translation process is a hierarchical series of VMPP program graphs. Figures 3 (a, c, e) and 4 (a) form just such a series. The hierarchy information will be part of the VMPP program graphs but we do not yet have an appropriate notation to represent it in a clean and clear fashion.

The translation process begins by representing the highest granularity computations and memory segments as composite computation and memory nodes, respectively. These nodes are interconnected with the appropriate arcs, representing data dependencies and memory accesses. The result is the top-level VMPP program graph, representing the highest granularity computations. In the next iteration of the translation process, the composite nodes of the program graph are expanded. The appropriate arcs are added, resulting in the next level VMPP program graph. This process is repeated until the lowest-level VMPP program graph, representing the smallest granularity computations, is constructed. This process is illustrated by Figures 3, 4, and 5.

An important part of the translation process not yet discussed is how function and subroutine calls are translated. A function call is translated by using a composite computation node to represent the call. A composite node is used because the function probably performs its computation by executing several sub-operations. This composite node can be expanded by replacing it with the subgraph that represents the function body with suitable adjustments for the parameters and global variables.

This basic process leaves several questions unanswered. First, how do we represent computation in a node? Does a node simply contain a pointer back to the source code? Is the source code first translated to some intermediate format like C? Another question is what to do when loop indices and data structure dimensions unknown. How can we draw the graph when we do not know how many nodes to make when expanding a loop? Answering these questions will be part of the research effort.

5.1.2. Data-Parallel Translations

In the data-parallel model, parallelism is obtained by performing the same set of operations on many data elements simultaneously. A language can support data parallelism either implicitly or explicitly. FORTRAN D [8] is an example of an implicit data-parallel language. In FORTRAN D the programmer specifies how the data, in the form of arrays, is to be distributed. Inner loop iterations that act on the data are executed in parallel. The data parallelism is implicit because the programmer simply writes sequential FORTRAN code and need not be concerned with any special instructions for indicating parallelism.

PC++ [13] is a language that supports explicit data parallelism. In PC++, the programmer specifies which operations can be executed in parallel and which data elements can take part in a parallel operation. Like FORTRAN D the programmer must provide data distribution information to the compiler. Because PC++ is based on C++ [12], it falls under the object-oriented paradigm as well as the data-parallel paradigm. However, since the explicit data-parallel model of PC++ distinguishes it from most other parallel object-oriented languages, we have chosen to present it in the data-parallel section.

FORTRAN D

FORTRAN D is an implicitly data-parallel programming language based on FORTRAN augmented with data alignment and distribution facilities. In FORTRAN D, the programmer specifies how arrays will be aligned with respect to one another and how they will be distributed across the processor set. For example, using the ALIGN and DECOMPOSITION statements, a

programmer could specify that the rows of one array should be aligned with the columns of another. Each row-column of this combined data structure could then be placed on a separate processor using the DISTRIBUTE statement. Parallelism is obtained by executing the inner loops that act on these distributed data structures in parallel.

FORTRAN D actually contains two levels of parallelism, loop-level parallelism and task-level parallelism [9]. Loop-level parallelism is the fine-grained data parallelism discussed above. Task-level parallelism is a coarse-grained functional parallelism based on executing non-loop-related blocks of FORTRAN D code in parallel. This type of parallelism is currently ignored by the proposed FORTRAN D implementation. It is important to note that both types of parallelism are available once a FORTRAN D program has been translated to a VMPP program graph.

The basic translation process covers loops, arrays, function and subroutine calls, accounting for most of the programming structures in FORTRAN D. It does not address how the ALIGN, DECOMPOSITION and DISTRIBUTE statements affect the translation. Array alignments specified by ALIGN and DECOMPOSITION statements provide architecture-independent information about which memory nodes should be placed together when the data is finally distributed. This information is important for the back-end translators and can be passed to them by annotating the appropriate VMPP program graph memory nodes. The DISTRIBUTE statement, on the other hand, provides architecture-dependent information. It is unclear how this architecture-dependent information should affect the architecture-independent VMPP program graph. However, if this information is necessary it can also be passed to the back-ends by appropriate node annotations. All the work done by the designers of FORTRAN D on data-dependence analysis, loop optimizations and removal of false dependencies [10] can be used in a VMPP front-end for FORTRAN D.

PC++

PC++ is an explicitly data-parallel programming language in which the programmer specifies a homogenous “collection” of data elements which are grouped together and can be referenced by a single name. The data elements are C++ classes and therefore have member functions. Data-parallel computation is accomplished by invoking a member function of the data elements via the collection. When this happens, the member function is executed on each data element in parallel. The data element member functions are augmented by special member functions that allow a data element to refer to other data elements in the collection. Using data distribution constructs similar to those in FORTRAN D, the data elements of a collection can be distributed over the processor set in various ways. PC++ also provides support for building hierarchies of distributed data abstractions. The PC++ collection, like a C++ class, is actually a template, and therefore PC++ collections can use inheritance for code sharing and sub-typing.

The main features of PC++ are collections and objects. Since a collection is a group of objects represented by a single entity, a collection can be translated as a single composite VMPP computation node. The objects that make up the collection can be translated as the nodes that make up the composite node representing the collection. Each individual object is translated as discussed in section 5.1.3. Hierarchies of PC++ collections and objects could then be translated as hierarchies of VMPP computation and memory nodes. As in the FORTRAN D case, node annotations can be used to transfer distribution information from the PC++ source through the VMPP program graphs to the back-end translator. Other features of the language, such as arrays and function calls, can be translated using the basic translation process.

5.1.3. Object-Oriented Translations

In the object-oriented model, parallelism is obtained by executing objects in parallel. There are two basic approaches. In the first approach, the memory space for each object is disjoint from that of any other. Objects in this approach communicate by passing messages. This scheme facilitates implementation on distributed memory systems.

In the second approach, all objects share a single address space. In this case, objects communicate via shared memory. This scheme is most natural to shared memory systems. In the following paragraphs we will show how languages using each of these approaches can be translated into VMPP program graphs.

Basic Object-Oriented Translation Process

Objects are a feature common to three of our example languages. For clarity, we will discuss the general translation process for objects here. Translations for features unique to a particular language are discussed in the appropriate sub-section.

An object consists of a group of functions and some associated storage. It is translated as a composite computation node and a composite memory node. The computation node represents the computations performed by the member functions. It is composite because there are usually several member functions, each of which is represented by a node. The composite memory node represents the storage for the object. It is composite because the total storage for an object is usually composed of several sub-units. For example, a matrix object may contain storage for each row. In this case, each row would become a memory node. All the rows together make up the composite memory node for the entire object.

Mentat

Mentat [14] is a parallel object-oriented programming system that uses the distributed memory approach for objects. It is based on the object-oriented sequential programming language C++. The goal of the Mentat system is to provide the programmer with efficient, easy to use parallelism in an object-oriented programming language.

In Mentat, parallelism is achieved by executing certain, programmer specified, objects in parallel. These objects are referred to as Mentat objects. When a program invokes a Mentat object's member function, the function is executed in parallel with the invoking program. The invoking program may not use the result from this member function invocation right away. It may perform some other operations before using the result in a computation. When the invoking program actually uses the result in another computation, two things can happen. If the parallel computation computing the result has completed, the invoking program uses the data without pausing in its execution. If the parallel computation has not completed, the system automatically blocks the invoking program until the result is available. This is invisible to the programmer since the compiler and the run-time system manage all the necessary communication, synchronization and data-dependency analysis. The programmer simply uses the Mentat object like any other C++ object. Because the programmer selects which objects are Mentat objects, the granularity of the parallel computations is specified by the programmer.

A single Mentat member function invocation may actually create many parallel computations. Since Mentat objects may contain other Mentat objects, a Mentat object in the course of computing one of its member functions may call member functions from the contained Mentat objects. These contained Mentat objects execute in parallel with the containing Mentat object. The contained objects may, in turn contain, other Mentat objects and so on. Thus, a single Mentat member function invocation can create many parallel computations. The fact that a Mentat object

has a parallel implementation using other Mentat objects is completely hidden from users of the object. This feature provides encapsulation of parallelism which simplifies the construction of very complex parallel objects.

Another important feature of Mentat is that certain Mentat objects can retain state between executions. Ordinarily, Mentat objects compute pure functions that require no state information, except what is passed directly as arguments. These objects are referred to as regular objects. Alternatively, the programmer can specify that a Mentat object retains state information between member function invocations. Such objects are referred to as persistent objects. Persistent objects allow the Mentat system to perform non-deterministic computations and allow state-retaining objects like files to be modelled by Mentat objects.

To show that Mentat programs can be represented by VMPP program graphs, we need to show how persistent and regular Mentat objects are translated. A persistent object is translated using the basic object-oriented translation scheme discussed above. State is maintained by default, since the memory node representing the object's storage is connected to all invocations of the object's member functions. To translate a regular object, the basic translation outlined above is modified so that a new memory node, representing the object's storage, is produced for each invocation of a member function.

Mentat objects that contain other Mentat objects create a hierarchy of parallel objects. Since VMPP program graphs are inherently hierarchical, they can easily represent nested Mentat objects. The objects are translated as described above with contained objects becoming the subnodes of the composite nodes that represent the containing Mentat objects.

Finally, since the dataflow analysis needed to detect, when a result is actually used, and to block and unblock Mentat objects, is performed by the Mentat compiler, this same kind of analysis can be performed by a VMPP front-end. The resulting dataflow information can easily be expressed in the VMPP program graphs.

PRESTO

PRESTO [15] is another parallel object-oriented programming system. Like Mentat, PRESTO is based on C++, but unlike Mentat, PRESTO uses the shared memory approach for objects. In the PRESTO system, parallelism is achieved through user-managed objects called threads. Threads are PRESTO's basic unit of execution. Conceptually, a thread consists of a program counter and a stack. Threads can be created by the programmer at will. To execute a thread the programmer indicates an object and a member function. The thread then executes the member function in parallel with the thread that created it. The use of threads allows PRESTO objects to have parallel implementations. For example, a member function may create several threads and execute them in parallel to carry out its operation. This parallelism is hidden from the user, thus as in Mentat, PRESTO allows encapsulation of parallelism.

Since all PRESTO objects execute in a single shared address space, some method of synchronization must be provided. PRESTO provides several synchronization classes. The simplest of these classes is the Lock class. A Lock object works like a binary semaphore. Before a critical section of code is entered, the programmer calls the Lock member function of a Lock object. Return from the Lock member function indicates that the thread holds that lock. A lock is guaranteed to be granted to a single thread at a time. The critical code is then executed, after which the programmer calls the Unlock member function. Aside from this basic synchronization mechanism, the system also provides the more structured concept of a monitor.

PRESTO programs can be translated into VMPP program graphs as follows. A PRESTO thread corresponds directly to a VMPP computation node. Just as the thread is PRESTO's basic

unit of execution, the computation node is the basic unit of execution in VMPP. When a PRESTO program specifies that a thread executes a member function, that member function becomes a computation node in the VMPP program graph. If that member function creates and executes other threads, the member functions associated with these other threads also become computation nodes. These computation nodes make up the composite computation node that represents the original thread.

A PRESTO Lock object can also be translated as a VMPP computation node. The Lock node has two input arcs. One arc represents all the calls to the lock member function for that Lock object. Each token arriving on this arc must indicate which call of the Lock member function it represents. The other input arc contains the single lock token. The Lock node has an output arc for each critical section protected by the lock. When the Lock node has a token on each input arc, the Lock node executes producing the lock token on the output arc connected to the critical section computation node for the corresponding Lock member function call. The critical section node now has the lock and can execute. Upon completion, the critical section node returns the lock token on the lock token input arc. This action corresponds to a call of the Unlock member function. This scheme, illustrated in Figure 6, insures that only one critical section node will execute at

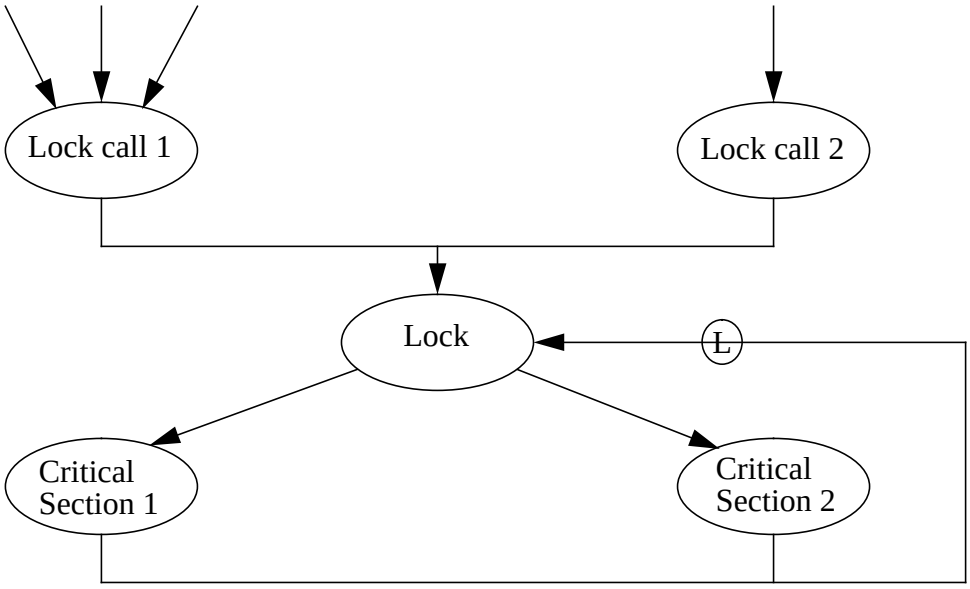


Figure 6. PRESTO Lock Translation.

a time, providing the mutual exclusion semantics of the Lock class.

Since monitors can be implemented using binary semaphores [16], PRESTO's monitors can be translated into VMPP program graphs by representing the monitors using PRESTO Locks and then translating those Locks as discussed above.

5.1.4. Functional Translation

In the programming paradigms we have discussed so far, computation is achieved by updating an implicit state or store, using various language constructs. In the functional programming paradigm there is no implicit state. All computation is achieved by the evaluation of side-effect-free functions or expressions. Any needed state is passed from one function to another explicitly.

In functional languages, parallelism is achieved by evaluating functions and expressions in parallel. Detecting which functions and expressions can be executed in parallel is a fairly easy task. Since there is no implicit state, all functions and expressions can be evaluated in parallel, subject only to the satisfaction of the data dependencies between them.

The most common functional languages used in parallel processing are the dataflow languages. Dataflow languages are essentially a subset of the functional languages [7]. We have chosen SISAL as our example functional language.

SISAL

SISAL is a dataflow/functional language, designed to support single assignment functional programming, particularly on parallel processors. SISAL differs from most other dataflow and functional languages because it supports explicit iteration, in the form of a *for* loop, as well as recursion. Parallelism is achieved by executing expressions, function calls and loop iterations in parallel, subject to the data dependencies of the computation.

SISAL programs can be translated into VMPP program graphs using the basic translation method outlined in section 5.1.1.

5.2. Implementability (Back-End Translations)

To demonstrate that VMPP program graphs are implementable, we will outline how VMPP program graphs can be translated into code suitable for execution on a variety of parallel processing systems. Current parallel processors can be categorized as SPMD (Single Program Multiple Data), DMMP (Distributed Memory Multi-Processor), SMMP (Shared Memory Multi-Processor), Dataflow, and Hybrid (some combination of the above). In this section we show that VMPP program graphs can be translated into code suitable for execution on each of these architectures with the exception of SPMD. We exclude SPMD architectures, because they are not general purpose parallel processors. SPMD machines can only exploit data-parallel code. They cannot in general exploit general purpose parallel code that does not exhibit data parallelism.

5.2.1. General Translation Process

There are certain steps that are common to all the back-end translations discussed below. For clarity, parts of the process that are common to all translations are presented in this section.

The first step is to select the granularity of computation. Each node in the VMPP program graph must be annotated with a measure of the amount of computation performed by the node. Annotations describing the amount of data carried by each arc associated with the node are also added to the graph. These annotations can be determined in the following way. Recall that a complete VMPP program graph contains a hierarchy of computation and memory nodes. The annotation process begins by examining the atomic nodes. These nodes are the innermost nodes of the hierarchy and represent the finest or smallest grain of computation in the graph. Each atomic node is examined and annotated with a measure of the amount of computation performed by the node. This measure can be calculated by counting weighted instructions or perhaps using some heuristic. Next, each arc associated with the node is examined. The node is now given an annotation for each associated arc that is a measure of the amount of data carried by that arc. These annotations can now be used to get a measure of the amount of communication versus the amount of computation performed by the node.

Once the atomic nodes have been annotated, the composite nodes that contain the atomic nodes can be annotated. The amount of computation performed by a composite node is equal to the sum of the computation performed by the nodes it contains. The arcs associated with the composite nodes are also annotated, as in the atomic node case. Once again, a communication-to-com-

putation ratio for the node is calculated. This process is repeated until all the nodes have been annotated. This is a bottom-up process starting with the finest-grain computation nodes and ending with the coarsest-grain computation nodes.

Once the VMPP program graph has been annotated, the back-end can select a granularity of computation that matches its target architecture. This is done by comparing the communication-to-computation ratio of the computation nodes to the communication-to-computation ratio of the target architecture. The back-end begins by looking at the top-level composite computation nodes. Any of these nodes that are too coarse for the target architecture are expanded. This process is repeated until the graph contains only nodes that match, subject to some criteria, the granularity of the target architecture. This process will be influenced by number of nodes available on the target architecture.

There are several issues currently unresolved. One is selecting a granularity for the memory nodes. Since the memory nodes are also hierarchical, there will be some size that is appropriate for the memory nodes. This process will strongly interact with the computation node granularity process, and will be influenced by the amount of memory available on a given node. Also, the issue of scheduling and distribution of the memory nodes has not been examined. These issues are part of the research effort.

5.2.2. DMMP Translation

We have selected the Paragon as our example DMMP architecture. The Paragon has a mesh interconnection network. Processors in the Paragon communicate via message passing. Each processor runs OSF1, a version of Mach which supports processes, tasks, threads, ports etc. [18].

VMPP program graphs are translated into code suitable for execution on the Paragon in the following way. First, an appropriate granularity is selected. Once the VMPP program graph has the appropriate granularity, code is generated for each node in the graph. This code will be C with message passing. The nodes' arcs are translated into appropriate send and receive calls. An incoming arc is translated as a receive, an outgoing arc as a send. Nodes will be numbered or marked in some way so that the sends and receives will have well defined destinations. Run-time system support may be needed to achieve this. Access to memory nodes is achieved in one of several possible ways. If the memory node is used by a single computation node, the memory node simply becomes part of the node's address space. Arcs between the computation node and the memory node then simply become memory accesses. If the memory node is shared by several computation nodes, the computation nodes are translated as threads running in a single task. The memory node they share is part of the task's virtual memory. If the memory node is shared by computation nodes that need to be scheduled on different processors, code is written to make the memory node into a server. The arcs connecting the memory node to the computations nodes then become send and receives and are handled by the server code.

Once code has been generated for the computation nodes, both the computation nodes and the memory nodes need to be scheduled. Computation nodes are scheduled to produce a balanced load with as much parallelism as possible. Memory nodes are scheduled so as to produce the minimum possible communication. Scheduling can be static or dynamic. A scheduling strategy will be developed as part of the research effort.

5.2.3. SMMP Translation

Another popular parallel architecture is the shared memory multiprocessor. We have chosen the Sequent [19] as our example SMMP system. The Sequent is a bus-based multiprocessor that supports shared memory. Processors communicate and synchronize via the shared memory.

The process for translating VMPP programs into code for the Sequent is very similar to the DMMP translation process described above. First, the granularity of the VMPP program graph is selected. Next, code is generated for the computation nodes. Memory nodes are simply blocks of shared memory. The back-end insures that only those computation nodes with arcs connecting them to a particular memory node access that node. Since all memory synchronization is implicit in the arcs connecting the computation nodes, no special mutual exclusion needs to be performed on the memory nodes. The arcs connecting computation nodes represent communication and synchronization. Communication and synchronization are done via the shared memory and can be handled by a simple run-time system that manages token matching, and determines when a node is ready to execute.

5.2.4. Hybrid Translation

Some parallel architectures are really a hybrid of two or more other architectures. Our example hybrid architecture is the CM-5 [22]. The CM-5 is a hybrid of the SPMD and DMMP parallel architectures. The CM-5 consists of a group of processors, each with its own private memory. These processors can communicate via message passing. The CM-5 also contains special hardware and software support for SPMD execution of the processors.

Since the CM-5 supports message passing and independent execution of its nodes, VMPP program graphs can be translated into CM-5 code using the DMMP approach. However, some VMPP computation nodes may represent data-parallel operations. If possible, we would like to use the special hardware and software support for SPMD execution to execute these data-parallel operations. At this time, it is unclear how this can be accomplished. This is a matter for further investigation.

5.2.5. Dataflow Translation

Several dataflow architectures are currently being constructed at various universities [20, 21]. Although considerably less developed than the architectures already discussed, dataflow architectures are of interest to a significant number of researchers in parallel computing. Our example dataflow architecture is the Monsoon [20]. The Monsoon machine consists of a number of processing elements with a small amount of local memory, connected to memory elements by a multistage packet switching network. This system contains some special hardware for supporting dataflow graph execution.

The computation nodes of a VMPP program graphs and the arcs that connect them form a dataflow program. This program can be mapped directly onto the Monsoon once a suitable granularity has been chosen.

The memory nodes of VMPP program graphs can be mapped to the memory elements of the Monsoon architecture. The Monsoon's memory elements are meant to hold the data structures of a Monsoon program. These data structures are in the form of I-structures; write-once, read-many arrays. The write-once semantics of an I-structure are enforced at the compiler level, thus the Monsoon's memory elements can be used as ordinary shared memory. The VMPP memory nodes can therefore be mapped to the Monsoon's memory elements just as in the SMMP translation.

5.3. Efficiency

The success or failure of VMPP hinges on whether the code generated by the VMPP system is efficient. Efficient in this context means fast. We have identified four main reasons for inefficient code in parallel systems. The first reason is failure of the system to detect and exploit available parallelism. A system will run slower than necessary if it fails to detect and exploit enough parallelism to keep all the available processors busy. This is becoming an increasingly important issue as larger and larger parallel machines are designed. Several existing systems ignore certain kinds of parallelism. For example, FORTRAN D exploits only loop-level parallelism, ignoring task-level parallelism. Mentat exploits only parallelism between objects and ignores loop-level parallelism. The VMPP system we are proposing is capable of detecting and exploiting all forms of parallelism available in a source program. In VMPP, potential parallelism is limited only by the semantics of the language in which the program is written.

The second reason for inefficiency in parallel systems is inherently sequential code. Inherently sequential code can be the result of an inherently sequential algorithm, or simply the way an algorithm was expressed when it was coded. There is nothing that can be done about the former but the latter can be addressed. Since VMPP was not designed to execute existing code, we can reserve the right to develop a VMPP programming style. Programs that do not adhere to the style guideline may run slow on certain systems. This approach is used in other parallel programming systems. For example, many compilers for vector supercomputers require that loops be written in a certain way in order to be vectorized. Defining a VMPP programming style would help to ensure that the programmer does not, inadvertently, express a parallel algorithm in an inherently sequential manner. Definition of a programming style will be part of the research effort.

A third reason for inefficient code is mismatched granularities. If the granularity of the parallel computations does not match the granularity of the architecture, the system will probably run slow. For example, if the granularity of the parallel computations is too fine, there will be too much parallelism. The overhead of communication, synchronization and scheduling may begin to dominate the computations, resulting in slower code. If the granularity is too coarse, there will be too little parallelism. In this case, there may be too few computations available for parallel execution. As a result, processors may remain idle, yielding slower code.

The VMPP system we are proposing can match the granularity of the parallel computations to the granularity of the architecture, resulting in efficient code for a wide range of parallel architectures. In order for the VMPP system to do a good job of matching granularities, there must be sufficient structure in the source code. As shown above, the VMPP system constructs hierarchies of graphs, with each level of the hierarchy representing nodes at a different granularity. In order to match granularities effectively for a wide range of architectures, there must be many levels in the hierarchy. Well-structured code provides the opportunity to construct program graphs with many levels. This is because in well structured code, large complicated functions are implemented with smaller simpler functions which, in turn, are implemented with even smaller simpler functions and so on. Large complicated objects are implemented with smaller simpler objects and so on. The VMPP system can exploit this kind of highly-structured code to provide graphs whose hierarchies contain many levels and thus provide good opportunities for granularity matching. In general, this kind of highly structured code is good software engineering and thus can be one of the requirements for the VMPP programming style mentioned above.

The fourth reason for inefficient parallel code is excessive run-time costs. This has been a particular problem for past dataflow systems because at run time, tokens must be matched, nodes must be scheduled, and so on. We believe this will not be a problem for a VMPP system. Our

experience with Mentat has shown that dataflow systems with reasonable run-time costs can be constructed.

There is one more reason for believing that VMPP will produce efficient code. Some systems for portable compiling that use an intermediate-language approach can produce inefficient code. This happens because information obtained by the front-end while parsing the source language is not available when the back-end generates code. VMPP will not suffer from this problem because of the flexibility of the VMPP intermediate language. Almost any kind of information obtained by the front-end can be passed to back-end using node annotations.

Finally, run-time efficiency is not the only type of efficiency that affects the usability of a system. Compile-time efficiency may also play a role. Because of the potentially large graphs that must be manipulated at compile time, the VMPP system may have high compile-time costs. We address this problem in two ways. First, since VMPP is designed for parallel systems, if speed of compilation is a problem, we can potentially compile in parallel. Also, past experience has shown that if a system is deemed useful, further research will often result in faster algorithms and compilation techniques.

6. Related Work and Comparison to VMPP

There have been several attempts to solve the portability problem in parallel computing systems. Although these attempts have succeeded in varying degrees, they are all ad hoc. VMPP has the advantage that it is a general solution and also provides a better solution in most cases.

6.1. Single Language Solutions to the Portability Problem

There are currently several parallel languages designed to provide portability across parallel architectures [8, 14]. The approach taken by these languages is to provide a single parallel programming language along with compilers for a variety of parallel architectures. The language is usually designed to be relatively architecture independent.

The single language approach suffers from two problems. First, porting a parallel compiler to a new architecture is difficult and time consuming. VMPP has a two-fold advantage: building a VMPP back-end should be considerably easier than porting a complex parallel compiler, and the effort to build a VMPP back-end is amortized across several languages because a single back-end can support all languages for which front-ends are available. Thus, the per language cost of porting VMPP is small.

Granularity can also be a problem. In many single language systems, the granularity of the parallel computations is either specified by the user or inherent in the programming language. As discussed above, if the granularity of the parallel computations does not match the granularity of the architecture, the code will be inefficient. When the granularity is specified by the user, the code is not really portable. A granularity that is appropriate for one architecture, may be inappropriate for another. To get efficient code on several architectures, the user must either build some scheme into the code to change the granularity at run time, or must change the code for each architecture. If the granularity is inherent in the programming language, then the code will probably run slow on architectures whose granularity does not match that of the language. VMPP does not suffer from this problem.

6.2. VMMP

Despite the unfortunate similarity in names, VMMP and VMPP are different projects. VMMP is the Virtual Machine for Multi-Processors [24]. It provides a virtual machine suitable for large and medium-grain parallel computation. VMMP runs on shared and distributed memory multiprocessors. The VMMP approach is to provide a coherent set of services for parallel pro-

gramming. The services provide two parallel programming models: tree computations and crowd computations. A tree computation is used to solve a problem by breaking it up into several simpler problems. These problems are then solved recursively. A graph of the computations would form a tree in which data is propagated from the root to the leaves and solutions are returned from the leaves up through the tree to the root. Examples of tree type computations are: divide and conquer type algorithms, and combinatorial algorithms. Communication is permitted only between a computation and its parent or between a computation and its children. In the crowd computation model, a crowd is a set of cooperating processes that work in concert to solve a problem. The processes may communicate in arbitrary ways by passing messages. Each member process in the crowd executes the same code, thus crowd computations are a form of data-parallel computation. In support of crowd computations, special operations for doing reductions, combinations, and gathers are provided. VMMP also provides limited support for shared memory objects.

VMMP can be used as a source language for constructing parallel programs, as well as an intermediate language for parallel compilers and programming tools. To construct a VMMP source program, the programmer makes calls to the various VMMP services from a sequential language like C or FORTRAN. The programmer is responsible for specifying the granularity of the parallel computations, and the distribution of data among the processors of the system. A parallel compiler can use VMMP as an intermediate language. The compiler parses the high-level parallel programming language, then generates C code with embedded calls to the VMMP services.

There are several problems with the VMMP approach. First, the VMMP designers attempted to provide services which are sufficiently high level for programmers to write application programs, yet are sufficiently low level to serve as a basis for an intermediate language for parallel compilers and tools. The resulting VMMP design is deficient on both counts. The tree computation services are high level, but can be used only for a limited set of applications. The crowd computation services are really just message-passing primitives, with some built-in operations for reductions, and combine and gather functions. In our opinion, these are low-level abstractions, as discussed in section 2.2, and thus unsuitable for high-level application programming. On the other hand, the C programming language, augmented with the above services, is not flexible enough to be a good intermediate language for parallel programming. For example, VMMP would not be a suitable intermediate representation for a dataflow language because the tree and crowd models are insufficient for expressing the functional parallelism present in most dataflow languages. VMPP does not suffer from this problem because its intermediate language is not meant to be used directly by the programmer.

Another problem is that the granularity is specified by the programmer. A VMMP program specifies a single granularity of computation. Therefore, that program will execute efficiently only on architectures that match the specified granularity. The code will have to be rewritten to run efficiently on an architecture that supports a drastically different level of granularity. This is not a problem for VMPP, because of its intermediate language's support for multiple granularity.

6.3. PVM

PVM is the Parallel Virtual Machine [25]. The PVM approach is similar to the VMMP approach. PVM provides a virtual machine suitable for large-grain and medium-grain parallel computation. The PVM programmer is provided with a library of services for parallel processing, which can be used from the usual sequential programming languages like C and FORTRAN. These services include primitives for message passing and shared memory. The PVM services are

then implemented on a variety of systems including a network of workstations, vector machines, and multiprocessors. PVM provides full support for architectural heterogeneity, by automatically performing the appropriate data conversions when transferring data from one machine to another. Limited support for language heterogeneity is also provided, because PVM code written in FORTRAN, can call PVM code written in C. As with VMMP, the programmer is responsible for specifying the granularity of computation and the distribution of data.

The PVM system suffers from some of the same problems as the VMMP system. Like VMMP, it is meant to be used as both a source language for constructing parallel programs and as an intermediate language for parallel programming tools. We believe the services provided by PVM are too low-level for application programming. Also, because the programmer specifies the granularity and data distribution, the code is not truly portable.

In general PVM is a better solution than VMMP because it provides support for architectural heterogeneity, and supports a much wider range of architectures. VMPP is a better solution than PVM because it supports a wider range of architectures, and can automatically match the granularity of the application to the granularity of the architecture.

6.4. Multi-model Programming in Psyche

Multi-model programming is the ability to use more than one programming model in a single program or across a group of programs running on a single machine or operating system. The Psyche operating system [26] is designed to support multi-model MIMD programming on shared-memory multiprocessors. Psyche provides a low-level operating system interface that is flexible enough to support a variety of MIMD parallel processing models, particularly the shared-memory and message-passing models.

As long as a parallel programming language is implemented using the Psyche operating system calls, it will be portable to any machine running Psyche. Furthermore, since the Psyche supports several parallel programming paradigms, a variety of parallel languages can be supported.

Psyche has several important limitations. First, it is designed to run only on shared memory multiprocessors, thus portability is severely limited. In effect, this approach simply moves the portability problem from the compiler level to the operating system level. Another problem with the Psyche approach is that it requires all systems to run the Psyche operating system. Convincing users and system administrators to switch to a different operating system, particularly one not supported by the system vendor, may be difficult. Psyche also requires the users of its services to specify the granularity of computation, thus limiting portability. VMPP is a better solution because it supports a wider range of operating systems, does not require users to change their basic system software, and provides automatic granularity matching.

7. Research Agenda

There are five items in our research agenda: 1) complete the design of the intermediate representation (VMPP program graphs), 2) define a front-end translation process for each of the languages mentioned in section 5.1, 3) define a back-end translation process for each of the architectures mentioned in section 5.2, 4) run experiments to determine the efficiency of the VMPP approach, and 5) develop a VMPP programming style if needed. Each of these items is discussed in detail below.

7.1. Complete the Design of the Intermediate Representation

The intermediate representation and execution model presented in section 4 is incomplete. The design must be completed in order to determine the front-end and back-end translation pro-

cesses. To satisfy this item of the agenda, a precise description of the exact syntax and semantics of the intermediate representation and execution model must be produced. Note that this design will not be static. As we research the front-end and back-end translation processes described below, we will discover deficiencies in the design. Each time the design is updated, corresponding changes will have to be made to the translation processes. These changes may, in turn, expose additional deficiencies in design. The result is an iterative process of design and development that uses feedback from later phases of the research to refine and improve the intermediate representation and execution model design.

7.2. Define Front-End Translation Processes

As mentioned in section 5.1, the process used to translate each of the target source languages into VMPP program graphs has not been completely specified. To meet this item of the research agenda, we will develop complete schemes for translating the target languages into VMPP program graphs. We do not plan to build any actual front-end translators, because writing compilers is a complex and time-consuming task, and is thus beyond the scope of this dissertation proposal. Instead we will document a step-by-step procedure for performing the indicated translations which will include references to other related work when appropriate. This information will be sufficiently detailed so that a competent computer scientist, with sufficient time, could construct the actual translators without doing any further original research.

7.3. Define Back-End Translation Processes

The back-end translation procedures, as described in section 5.2, are incomplete. We will meet this item of the research agenda by developing complete schemes for translating the VMPP program graphs into code suitable for execution on the target machines. We will document a step by step procedure for performing each translation, referring to related work when necessary. As in the front-end case, this information will be sufficiently detailed to allow construction of the actual translators.

Unlike the front-end case, we will actually implement two of the back-end translators. The back-end translation process is where two of the most interesting aspects of this work will be carried out: the automatic granularity matching, and the generation of efficient executable code. To demonstrate that these operations can be performed by an automated translation system, we will construct back-ends for the Paragon and CM-5. There is one caveat: as mentioned in section 5.2.4, it is unclear how the CM-5's special SPMD hardware can be used to carry out data-parallel operations on a sub-group of processors. If further research indicates this cannot be done, then the CM-5 back-end will closely resemble the Paragon's back-end. If this turns out to be the case, we will construct a back-end for the Sequent, instead of the CM-5.

7.4. Efficiency Experiments

Efficiency is a key criteria for the success of VMPP. Although it is unclear exactly how efficient VMPP must be in order to be successful, a technique for measuring efficiency is necessary. To measure efficiency a test suite of parallel programs will be constructed. It will consist of a few canonical examples and a small application. These programs will be coded in each of the source languages described in section 5.1. They will be hand translated into VMPP program graphs using the translation processes that result from agenda item 2. The VMPP program graphs will then be translated into executable code and executed on each of the architectures listed in section 5.2. The resulting VMPP execution speeds will be compared with the execution speeds for code written in languages native to the architecture under consideration. These comparisons will provide a measure of the efficiency of the VMPP approach.

More specifically, the following experiments will be run. First, each program in the test suite will be coded in each of the source languages. If there are N test-suite programs, this will result in 5N source-language programs. Each of these 5N programs will be translated into a VMPP program graph. For each architecture, each of the 5N program graphs will be translated into executable code. These programs will then be executed and the resulting execution times recorded. Next, if any of the source languages has a compiler for the architecture under consideration, the test programs written in that source language will be compiled and executed. Times for these programs will be compared with times for the VMPP compiled versions. Finally, each test program will be written in the most common language used on the architecture under consideration. These program will be hand tuned to get the best possible performance, using expert advice when available. These programs will then be executed and the resulting times compared to the VMPP times. This process will be repeated for each architecture.

7.5. Develop a VMPP Programming Style

As mentioned in section 5.3, certain ways of expressing an algorithm, or of using various parallel constructs may be better suited to the VMPP approach than others. This item of the research agenda will be fulfilled by identifying programming structures and practices that produce inefficient executable code in the VMPP system. Alternative ways of expressing these constructs, so that efficient code can be generated, will be explored. This information will then be used to develop a coherent VMPP programming style.

8. Bibliography

[1] T. B. Steel, “UNCOL: The Myth and the Fact”, Annu. Rev. Autom. Program., Vol. 2, 1960.

[2] Jack B. Dennis, “First Version of a Data Flow Procedure Language”, Technical Report TR-673, Massachusetts Institute of Technology, Cambridge Massachusetts, May 1975.

[3] Micah Beck, Richard Johnson, and Keshav Pingali, “From Control Flow to Dataflow”, Journal of Parallel and Distributed Computing, Vol. 12, No. 2, June 1991.

[4] Paul A. Suhler, Jit Biswas, Kim M. Korner, and James C. Browne, “TDFL: A Task-Level Dataflow Language”, Journal of Parallel and Distributed Computing, Vol. 9, No. 2, June 1990.

[5] Arvind and Rishiyur S. Nikhil, “Executing a Program on the MIT Tagged-Token Dataflow Architecture”, IEEE Transactions on Computers, Vol. 39, No. 3, March 1990.

[6] W. Daniel Hillis and Guy L. Steele, Jr., “Data Parallel Algorithms”, Communications of the ACM, Vol. 26, No. 12, pp. 1170 - 1183, December 1986.

[7] Paul Hudak, “Conception, Evolution, and Application of Functional Programming”, ACM Computing Surveys, Vol. 21, No. 3, September 1989.

[8] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu, “FORTRAN D Language Specification”, Technical Report TR90-141, Department of Computer Science, Rice University, Houston Texas, 1990.

[9] Vasanth Balasundaram, Ken Kennedy, “A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations”, Proceedings of the SIGPLAN ‘89 Conference on Programming Language Design and Implementation, Portland OR, June 21-23, 1989, also listed as SIGPLAN Notices Vol. 24, No. 7, July 1989.

[10] David Callahan and Ken Kennedy, “Compiling Programs for Distributed-Memory Computers”, The Journal of Supercomputing, 2, pp. 151-169, 1988.

[11] Constantine D. Polychronopolus, “Parallel Programming and Compilers”, Kluwer Academic Publishers, Boston/Dordrecht/London, 1988.

[12] Bjarne Stroustrup, “The C++ Programming Language”, Second Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[13] Jenq Kuen Lee, and Dennis Gannon, “Object Oriented Parallel Programming: Experiments and Results”, Proceedings of Supercomputing ‘91, Albuquerque, NM, November, 1991.

[14] Andrew S. Grimshaw, “An Introduction to Parallel Object-Oriented Programming with Mentat”, Computer Science Report No. TR-91-07, Department of Computer Science, University of Virginia, Charlottesville, VA, April 4, 1991.

[15] Brian N. Bershad, Edward D Lazowska, and Henry M. Levy, “PRESTO: A System for Object-Oriented Parallel Programming”, Software - Practice and Experience, Vol. 18, No. 8, pp. 713-732, August 1988.

[16] C. A. R. Hoare, “Monitors: An Operating System Concept”, Communications of the ACM, Vol. 17, No. 10, October 1974.

[17] James McGraw, Stephan Skedzielewski, Stephan Allan, Rod Oldenhoeft, John Glauert, Chris Kirkhan, Bill Noyce, and Robert Thomas, “SISAL: Streams and Iterations in a Single Assignment Language, Language Reference Manual, Version 1.2”, Lawrence Livermore National Laboratory Manual M-146 (Rev. 1), Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[18] Robert Baron, Richard F. Rashid, Ellen Siegal, Avadis Tevanian, and Michael Young, “MACH-1: An Operating System Environment for Large-Scale Multiprocessor Applications”, IEEE Software, July 1985.

[19] Anita Osterhaug, editor, “Guide to Parallel Programming on Sequent Computer Systems”, Prentice Hall, Englewood Cliffs, NJ, 1989.

[20] Gregory M. Popadopoulos, “Implementation of a General Purpose Dataflow Multiprocessor”, Technical Report TR432, MIT Lab for Computer Science, Cambridge, MA, September, 1988.

[21] A. P. Wim Bohm and John R. Gurd, “Iterative Instructions in the Manchester Dataflow Computer”, IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990.

[22] Charles E. Leiserson et al, “The Network Architecture of the Connection Machine CM-5”, Symposium on Parallel and Distributed Algorithms ‘92, San Diego CA, June 1992.

[23] Ralph Duncan, “A Survey of Parallel Computer Architectures”, IEEE Computer, February, 1990.

[24] Eran Grabber, “VMMP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors”, IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 3, July 1990.

[25] G. A. Geist and V. S. Sunderman, “Networked Based Concurrent Computing on the PVM System”, Technical Report ORNL/TM-11760, Oak Ridge National Laboratory, Oak Ridge, Tennessee, June, 1991.

[26] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh, “Multi-Model Parallel Programming in Psyche”, Proceedings of the Second ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming (PPOPP), SIGPLAN Notices Vol. 25, No. 3, March 1990.

[27] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein, “The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages