HIGH PERFORMANCE VIRTUAL ARCHITECTURE PARALLEL LIBRARIES WITH DATA REDISTRIBUTION FOR MULTICOMPUTERS

Hassan Hosseini

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

January 1998

0-612-40310-6

Abstract

High Performance Virtual Architecture Parallel Libraries With Data Redistribution For Multicomputers

Hassan Hosseini, Ph.D.
Concordia University, 1998

Sequential programs which use library calls to perform their intensive numerical computations may not deliver satisfactory performance for large problem instances in uniprocessor systems. Replacing the library system with one that performs the computations on a multicomputer can provide significant improvement in the execution time of these programs. These parallel libraries also encourage programmers who have no knowledge of multicomputer programming to use multicomputers to run their newly developed compute-intensive applications.

Multicomputer programs perform computations on distributed data. Transfer of data between processors is carried out using communication operations which are normally costly. Introduction of parallel library systems gives rise to four important issues. The first one is the design of the library routine without knowing the ***problem instance size*** and the ***physical system size***, as this is the case with many partitionable and reconfigurable systems. *Performance* of the library routine which is sensitive to the granularity of the computation and the mapping of the computation onto the physical system is the second issue. Maintaining a *call interface* which resembles those of sequential libraries is the third one. Finally, once ported to a new platform, parallel system *speedup* becomes a major concern.

Data distribution at each parallel library call is performed sequentially which, consequently, degrades the performance of the library routine. Since distributed data used or produced by one library call is often used in the subsequent calls to the same routine or other library routines, it is beneficial to redistribute the data from the former library call to prepare for the latter. The redistribution operation is a parallel operation and reduces the overall execution time of a parallel library call.

This thesis presents the design of a parallel library system which possesses several unique properties. The design supports dynamic grain adjustment and delayed mapping of the virtual to physical processors in order to reduce the communication overhead of the library calls. It also supports transparent distributed data management that results in a call interface similar to those of sequential libraries. Furthermore, the design supports transparent data redistribution across parallel library calls. Once ported to a new system, the library can be easily adjusted with the target system parameters to deliver the best performance based on the new parameters.

Feasibility, performance, and overhead of our design have been experimented using a source to source transformer, a compiler, library design of several virtual architecture parallel algorithms, a mapping module, a virtual communication library, a redistribution library, and a multicomputer simulator. The implementation of the library system on an actual multicomputer has been thoroughly discussed in the thesis.

Acknowledgments

Parents and teachers play a big role in shaping a person's life, but hardly ever appreciated enough. This is an opportunity to express my gratitude towards my supervisor, Dr. Lixin Tao, in assisting me in getting this work to this stage. I would also like to dedicate this work to the memories of my father and to my mother.

My family will certainly remember these years in the future. My wife, Parvaneh, gave me the support and encouragement I needed to pursue this work. My children Behnam, and Behnaz, the highlights of my life, have made this a much more challenging endeavor.

Thanks to the members of our research team, Dr. Lixin Tao, Hoang Uyen Trang Nguyen, and Thien Bui who each contributed to the implementation of our programming environment. Dr. Lixin Tao led the team, Trang developed the Wormhole routing simulator, and Trang and Thien contributed to the code execution module.

My good friends and partners are difficult not to mention. Trang and Thien were always there for advice and assistance on technical and non-technical issues. I am thankful to Ken Hahn, George Felfli, and Osama Abu-Shihab for giving me encouragement and, in some cases, technical advice. My relatives and friends who were concerned and, therefore, will be remembered were: My brother-in-law Hassan, Ben, Helen, Jahan, and Firoozeh.

Many thanks to the UNIX system administrators at Concordia who have provided superior system support and services. I would especially like to thank Stan for his technical support. A token of appreciation also goes to the secretarial staff at Concordia, especially Halina Monkiewicz, who were always friendly and prompt in providing assistance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reusability, modularity, and portability are aspects of software engineering that allow fast development of reliable, easily maintainable, and portable programs [63]. Libraries are a mechanism to support these aspects. Moreover, ease of use, efficiency, and correctness of a library system encourage the programmers to take advantage of the existing routines as opposed to developing new ones from scratch. There are many existing sequential programs that rely on numerical library routines, such as LINPACK [21], to perform their intensive numerical computations such as matrix and vector operations, Gaussian elimination, Fast Fourier Transform, and so on. These libraries have been designed to facilitate programming and provide good performance. Some hardware characteristics such as the hierarchy of memory, the number of registers and the memory latency have been considered in the design of these libraries to improve the performance. However, when large problem instance sizes need to be solved using these library routines, regardless of the various optimizations, the performance may still be unsatisfactory due to the sequential nature of the library routines. Even if the libraries are based on the multitasking or multithreading technique, the performance will be limited by the speed of the uniprocessor system.

With the advent of parallel computing, programmers have considered running their existing or newly developed programs on distributed-memory systems to reduce the execution time of their programs beyond the limits of the uniprocessor systems. Since, for large problem instance sizes, the majority of the execution time of the program is spent in the sequential library routines, these library routines have been the first target for improving the overall performance of the programs. Programming on multicomputers is in general difficult because many fine details must be managed by the programmer. Performance gain is also challenging on these systems since communication operations are expensive. The execution time is usually sensitive to the granularity of the computation and the mapping of the computation to the physical processors. Partitionable and reconfigurable systems make programming on these systems even more difficult because the programmer is not aware of the system size and topology at programming time. Parallel libraries can greatly reduce the programmer's task in using multicomputers. A desirable parallel library design preserves the sequential call interface, offers good performance, and can be easily ported. Parallel libraries which possess these characteristics allow the existing sequential programs with library calls to be easily assimilated into a multicomputer environment. The library system can provide the programmer with the tool to convert the sequential program, that interfaces with the parallel library, to an SPMD[1] form. The library system can then provide the necessary components to be linked with the SPMD form to obtain an executable image for the specific multicomputer. These components are either the library routine and its constituents or other runtime libraries used in the parallel library routine implementation, such as a threading library, mapping library, communication library, or redistribution library provided by a one-time compilation using the resident compiler of the target system. In any case, no parallel programming knowledge will be required from the programmer. Knowledge of how to load and run an executable image on the target system can be easily encapsulated. Portability can be easily accommodated in such design by using standard calls, such as calls to standard communication and threading libraries. This will require absolutely no changes to the user program to run on the new system. However, a scheme is required to ensure the best speedup on the target system after porting. With advances in the hardware technology, the latency and the startup cost of communication will be reduced in the future systems. Even in the currently existing multicomputers, the communication latency varies noticeably between different multicomputers. Consequently, library routines must adjust the degree of concurrency and the granularity based on the physical system attributes, such as the communication latency and the processor speed.

Once a design with the above characteristics is implemented, the best algorithms for the scientific and numerical applications can be selected and placed in a parallel library to be used by the programmers. These library routines can also be tuned for better performance prior to compilation. Providing such a system for the programmer requires a rather elaborate design of the library system. There are several design issues which affect either the performance, the user interface, or the portability.

In the following sections, we unravel the problems in the design of parallel libraries, provide a critical view of the current approaches to solve some of these problems, and present a comparative view of the current approaches with the ones proposed in this thesis.

1.1 Parallel Library Issues

The difficulty of parallel library design stems from the lack of a clear model similar to the sequential stack model. In sequential library calls, input and its size are passed as parameters on the stack to the library routine. The library routine also returns the results to the caller through the parameters on the stack. This provides a natural and easy interface for the callers. The performance of the library routine, for a given problem instance, is then solely determined by the underlying algorithm and the optimizations performed by the compiler. Sequential library routines are easy to write and optimize. The associated cost of calling these routines is also well defined. These libraries are also easy to port bearing in mind the interoperability issues. In a parallel library routine, on the other hand, the library writer must deal with the distributed data and its impact on the user interface,

[1]Single Program Multiple Data model is referred to the programming model where computation is described by the same code operating on different data items.

the communication overhead of the library routine, and the complexity in the design of the library routine itself. Unknown physical system size, topology, and physical system characteristics at the library routine design time make this problem even harder to deal with. Mapping of processes to the processors, grain size of computation and communication, and scalability of the parallel algorithm have serious impact on the communication overhead of the program. In addition, the communication overhead of the library routine upon invocation, due to data distribution, may be avoided by data redistribution. Here we elaborate on the factors which concern the design of parallel libraries and their impact on the library writer, the user of the library, and the compiler.

Performance - Degree of parallelism and the granularity of the computation used by a library routine must be selected to minimize the execution time of the routine. Different systems require different values for the degree of parallelism and the granularity to deliver their best performance. A parallel library must harness the underlying system based on its hardware characteristics such as the communication latency and the processor speed. When the library is ported to another system, a centralized modification to the library may be used to ensure that the library routine selects the granularity and the mapping based on the characteristics of the new system. An example would be to relay to the library system the startup cost of communication and the link bandwidth.

Problem instance size - The problem instance size is usually input at runtime. The parallel library routine based on the logical architecture must be described without this knowledge. This also delays the grain adjustment and mapping of the library routine to runtime when the problem instance size is known.

System size and topology - We use the term *partitionable* for systems which are based on a fixed topology from which a partition is granted to the user upon request. Size of this partition depends on the system availability at the time of the request. The term *reconfigurable* is used for systems in which the topology may also change statically or dynamically. In these systems a user's request for a partition is submitted after an executable image of the parallel program is built. The granted partition may differ between different runs of the same program depending on the requested size, system load, the maximum system size, and the scheduling policy. Partitionable and reconfigurable systems give rise to unknown attributes at the time of library routine design. The library routine writer is unaware of the physical system size at implementation time. Hence, the library routine must minimize the execution time of the library routine given the size and topology of the granted processors.

Mapping - Communication among the processors is much more expensive than an instruction execution. Assignment of the logical processors to the physical processors impacts the communication requirements of the parallel program which, in turn, affects the execution time of the program. Since the physical system size and topology are not known until runtime in both partitionable and reconfigurable systems, mapping of the computation to the physical processors must be delayed to runtime. The design of the library system for partitionable systems is simpler since only a limited number of mappings from the logical processors to the physical processors needs to be considered. The mapping library can be simplified on these systems since the target topology is known. For reconfigurable systems, the physical topology may also change, so the library designer must consider all possible physical topologies. The mapping library must be extended to accommodate any target topology. These systems, however, may provide better performance as there is a higher possibility of matching the logical topology with the physical one. Mismatch between the logical topology and the physical topology may introduce additional communication overhead. Since the topology of a logical architecture algorithm is generally representative of the communication pattern of that algorithm, it is desirable to map the neighboring nodes of the logical topology as close as possible in the target architecture. Mapping functions that minimize this distance tend to reduce the communication overhead through avoidance of hot spots in the network and reduction of the communication latency between the nodes. If the communication pattern of the parallel algorithm is based on a unique pattern or topology not supported by the mapping library, or it requires a mapping function not commonly used or supported, the mapping function can be built into the library routine. The most commonly used mapping functions can, however, be placed in a common library to be used by all the library routines.

If the size of the logical architecture and the physical system are the same, one-to-one mapping can be used to execute the parallel library routine. Even with the flexibility of dynamic grain adjustment the logical architecture size may not be reducible to the physical system size. In such case, many-to-one mapping will be required to execute the parallel library routine. In other words, more than one logical processor will be emulated at least by one physical processor. Logical processors must be emulated using a model that allows execution of multiple logical processors in a physical processor without intolerable overhead. In addition, communication between logical processors will require resolution of message destination at each physical processor.

Call parameters and Distributed data - Prior to calling a parallel library routine, the actual parameters must be decomposed and distributed among the logical processors. A library writer must determine the level of the caller intervention on the initial data distribution, the final data accumulation, and the corresponding impact on the user interface to the library routines. Each parallel algorithm requires its data in a specific distribution upon starting the parallel computation and leaves the distributed data in a specific distribution in the memory of the physical processors. Although the data distribution can be hard-coded in the library routine, the information can be used for data redistribution in the later phases.

Granularity - The amount of computation by a process between two communication phases is known as the *Granularity.* The granularity of a library routine must not be fixed at library routine implementation time since, depending on the problem instance size and the physical system size, the fixed grain size may have adverse effect on the performance of the program. Therefore, the granularity must also be determined by the library routine once the problem instance size and the physical system size are known. Runtime grain size adjustment has been shown to provide larger range of scalability for many algorithms. Higher scalability is desirable for parallel libraries since the routines are more likely to deliver desirable performance when the problem instance size and system size are determined at runtime. The majority of parallel algorithms behave poorly for small problem instance sizes. It is, therefore, necessary for the parallel library routine to identify such cases and possibly execute these using fewer processors or even sequentially. Considering the cost of communication, this seems to be a reasonable solution for small problem instance sizes. Our interest is in extending the range of scalability on large problem instance sizes.

Data redistribution - In a program with two or more parallel library calls, it is beneficial to keep the distributed data of one phase in the processors' local memories to be reshuffled for a subsequent phase. Data redistribution across the parallel phases of a program may enhance the performance of the program by avoiding unnecessary accumulation and distribution of data. Each library routine considered in isolation must distribute the initial data and accumulate the final results. With data redistribution in mind, the library routine implementor must set up such distribution and accumulation routines as part of the library routine. The distribution and accumulation routines are inserted by the compiler based on the requirement of the caller program. If the runtime system recognizes potential redistribution, it executes necessary code to perform reshuffling of the data between two parallel library calls. The library routine writer must design the routines bearing in mind the potential redistribution between parallel phases.

Portability - Effortless porting of the parallel library and the user programs from one system to another is a desirable property. If the library system relies on commonly used runtime libraries and storage schemes, this process is straightforward. However, using special programming constructs, runtime libraries, or communication libraries not commonly available will make the porting problematic.

Coherency - The parallel library calls may be nested within various programming constructs or may be called in other functions. The conditions and loop bounds of these programming constructs may not be known statically. Some of the values may be input by the user. The SPMD code images loaded onto the nodes of the physical system must stay coherent so that all the physical processors can potentially participate in every parallel library call even though some nodes may not be mapped any logical processors to emulate. Input and output statements must be executed by the processor which has I/O capability. Some of these values are required by the other nodes so that the nodes follow the same execution path as the host.

The design of parallel libraries becomes even more complex since the factors mentioned above are intertwined. These factors create several new dimensions which are not pertinent to the sequential libraries. Therefore, parallel libraries require a more elaborate design. Providing a level of encapsulation equivalent to the sequential libraries leaves many of the design issues to be considered by the library routine designer.

1.2 Current Status

Two commonly known parallel libraries are ScaLAPACK [12] and CMSSL [42]. The routines in both these libraries have been designed to be called either from message passing programs or data parallel programs such as the ones written in HPF (High Performance Fortran) [27]. Existing compilers for HPF convert the user's HPF program to the message passing form [6]. In this section we will highlight the distinct features of these libraries. We consider these libraries in their handling of the parallel library design issues discussed in the previous section. Since the aforementioned parallel libraries rely on the user's HPF program to specify the logical system size and the data distribution prior to the call, we evaluate these libraries in the context of HPF.

Performance is largely determined by the granularity of the computation, the mapping of the data to the logical processors, and the mapping of the computation onto the physical processors. ScaLAPACK and CMSSL libraries perform the parallel computation using the granularity and the data mapping specified by the user's HPF program. Data distribution is explicit in the program and obtaining good performance would require several trial and error runs which vary the granularity and the mapping of the data onto the logical processors.

The mapping of the logical processors to the physical processors is considered a compiler dependent issue in HPF and the current parallel libraries. The programmer does not have the freedom to specify a mapping which suits the parallel program. The HPF language does not support directives which relay the communication pattern of the program to the compiler so that proper mapping of the computation to the physical processors can be performed. User defined mapping of the computation to the physical processors is recommended as an extension in the HPF language specification. Current implementations of the HPF language such as [2] do not support this feature.

Unknown problem size, physical system size, and physical system topology have high impacts on the programmer in HPF. The programmer must declare a logical mesh that matches the physical system in size. If an algorithm is based on a logical architecture that is not a mesh, the programmer's task in describing the algorithm in HPF is very difficult. Although the programmer is not aware of the physical system size and topology at programming time, the intrinsic functions can be used to describe the logical architecture size equal to the physical architecture size. Using this feature of HPF and the data distribution directives, the grain size of the computation can be set so that the ScaLAPACK or CMSSL routines use only part of the physical system for the computation. There are two shortcomings in this scheme. The first shortcoming is that determining what fraction of the granted system delivers the best performance is not a trivial task for the programmer. To only use a fraction of the physical system, the granularity of the computation must be statically defined in the specification part of the program. As it was previously described, when the grain size is selected statically, the program may not deliver its best performance when ported to a different platform. The second shortcoming has to do with dynamically allocated data used in the distribution directives. In such case, the actual distribution does not take place until the data is allocated. However, the user is required to specify its distribution and granularity at programming time. If the problem size is not known statically, the programmer has no way of deciding on a suitable granularity and distribution at programming time. Even when the problem instance size is known, partitionable and reconfigurable systems pose a similar problem since the best grain size of the computation depends on the size of the system at load time.

Although ScaLAPACK and CMSSL have interfaces that resemble the sequential library routines, the user's HPF program must manage the distributed data. Therefore, the sequential call interface with transparent data distribution is not supported in ScaLAPACK and CMSSL. The HPF compiler translates the distribution directives into message passing code that sets up and maintains the information about the distributed data in designated structures. Internally, ScaLAPACK and CMSSL are designed to accept these structures as parameters. Obviously, ScaLAPACK and CMSSL routines cannot be called from sequential programs.

Redistribution of data is supported in HPF. The redistribution is completely transparent to ScaLAPACK and CMSSL routines. The programmer specifies the redistribution of data across the logical architectures using compiler directives. The fact that the HPF compiler must allow any logical architecture declaration that matches in size with the physical architecture, limits the redistribution support in HPF. HPF requires the two parallel phases to have equal virtual architecture sizes. This greatly simplifies the redistribution task. In extended compilers that allow the two phases to have different size logical architectures, the redistribution becomes more complex.

The issue of coherency in implementations of HPF and parallel libraries is handled using proper translation of data-parallel global name space programs to message passing programs. Access to the distributed data is translated into communication operations. If conditional and iterative programming constructs use the distributed data for decision making, the compiler must produce appropriate code so that all logical processors use the most current value of the variable. The owner-compute rule, used in HPF, causes the owner of an element to perform the write to that element of the distributed data. A read operation of the distributed data must be broadcast, by the owner of the data, to all the logical or physical processors (Note that these are the same size in an HPF program).

In the current implementations of the HPF compiler, portability is not a major concern. The compiler translates the data parallel programs to message passing programs based on well known communication libraries. User programs compile and run successfully on different platforms. However, the delivered performance may not be the best on the target system. This is because HPF, and obviously ScaLAPACK and CMSSL, do not perform any adjustment of granularity and mapping across different systems to account for the hardware characteristics of the target system.

1.3 Our Approach

We have based our library routines on the *virtual architecture* programming model [53, 69, 38, 78]. In this model the program is described on a virtual architecture of desired size and topology, to which is then associated an SPMD code. We have based our parallel libraries on virtual architecture algorithms the following reasons. 1) The virtual topology of these algorithms is generally representative of the communication pattern of these algorithms. This information can assist the compiler or the run time system to perform mapping of the computations to the physical processors. Suitable mapping of the regular communication can reduce the communication overhead of the program.

2) Virtual architecture parallel algorithms are easy to implement since the programmer can describe the algorithms on the virtual architecture matching that of the algorithm and let the compile time or the runtime system perform the necessary translation to account for the mapping of the computation and handle the communication among the virtual processors. 3) The size of the virtual topology of an algorithm is related to the problem instance size. This property can be used to deduce the size of the virtual architecture at run time once the problem instance size is known. 4) These algorithms can be easily described in block form by reducing the size of the virtual architecture and recoding the computation performed by each virtual processor. This property can be used to implement the algorithms in coarse grains and reduce the communication overhead.

We have extended our application domain by several design decisions. A large number of virtual topologies have been accounted for in our library design, and extensions to handle new topologies can be added with very minimal effort through changes in the mapping library. The communication library provides collective communication operations. Consideration of both partitionable and reconfigurable systems in our design has further extended our application domain.

Like other programming models, there are compile time and runtime concerns to deal with in the virtual architecture programming model. The mismatch between the virtual architecture size and the physical system size (which is usually not known at compile time) dictates the use of *delayed many-to-one mapping*. The quality of the mapping, on the other hand, affects the communication cost of the program. A collection of rich mapping functions [58, 57, 5] across regular topologies ensures that the communicating nodes are placed as close as possible in the physical system. This strategic mapping reduces communication cost of the program in many currently used packet switching systems [79, 15, 16, 26, 50] by reducing the distance between the communicating nodes, hence lower latency and link contention. On other systems that use *wormhole routing*, poor mapping degrades the performance due to contention [71, 17, 59, 18, 3, 76, 61].

Following the design and the implementation of the virtual architecture programming paradigm, we decided to implement these parallel routines in a precompiled form, parameterized by the virtual architecture size. Although this task may not seem difficult at first, assuring efficiency of these routines and low overhead are rather challenging problems. A large problem instance size described at fine grain may run poorly on a small physical system due to the abundance of small messages. This granularity problem is then overcome by our *contraction* technique which ensures efficient execution of these routines by *runtime granularity adjustment.*

Our goal in this research is to offer a new design of precompiled virtual architecture routines and *parallel libraries* of scientific numerical applications for multicomputers. The novel features of this design are performance improvement through dynamic grain adjustment and delayed mapping of the virtual processors, automatic data distribution, and the initial and final data layout maintenance for redistribution. The representation of the parallel library routines, in this thesis, is appropriate for parallel numerical and scientific applications that can be described in block matrix form. Sequential programs may call these parallel library routines in different language constructs and benefit from efficient execution and automatic data redistribution between phases. Contrary to the currently used techniques, our parallel library routines, which are in parameterized contracted form, support coarse grain communication as well as coarse grain data redistribution between two consecutive phases. The coarse grains reduce not only the initial data distribution communication cost but also the communication overhead of the parallel library routine in the course of its execution.

Our design meets the ease of use criteria by providing a sequential call interface to the virtual architecture parallel library routines. This is achieved by encapsulating the virtual architecture library components in the library and completely transparent to the caller. From the viewpoint of the library routine implementor, our library design offers a set of guidelines to follow in setting up a new virtual architecture parallel library routine. Virtual architecture parallel routines are independent of physical architecture and are easy to implement. Our centralized mapping library and our virtual communication library further facilitate implementation of these library routines in the message passing model. Our design meets the performance criteria through several design choices. The first one of these is the selection of virtual architecture parallel algorithms as the basis for these libraries. The communication pattern of these algorithms is often regular and matches their virtual topologies. This information can be used by the runtime system to perform systematic mapping of the computation to the physical processors. The centralized mapping library of optimal mapping functions is used to map the virtual processors to the physical processors. The mapping is delayed to runtime since the virtual architecture size and the topology of a parallel library routine are not known at library routine design time. Runtime grain adjustment is another scheme offered in our design which reduces the communication overhead of the library routines. Our design allows runtime grain adjustment based on the problem instance size, physical system size, and physical system attributes. This is suitable for partitionable and reconfigurable systems where the allocated partition is not known before hand. Once a partition is allocated, for each parallel library call, the objective is to minimize its execution time even by limiting parallelism and using only a subset of the physical processors. We have met the portability criteria in our design by not requiring any special language features. The user program and the SPMD form are both based on the conventional programming language constructs. Our library routines require the use of a threading library and a communication library. The implementation can be easily altered to use any threading or communication library. Our virtual collective communication library is based on our virtual point-to-point communication routines which, in turn, use a low level communication library such as PICL. Finally, our library design utilizes a physical system based on its physical characteristics. Once the library system is ported to a new architecture readjustment of the library parameters with those of the new system allows all routines in the library to perform grain adjustment based on the new system parameters.

Chapter 2 of this thesis gives a motivation of parallel libraries and a detailed presentation of the related work. Chapter 3 describes our parallel library routine design and representation. Chapter 4 discusses data redistribution in multiphase programs. In Chapter 5, we discuss our integrated programming environment implementation details. Finally, we conclude this research in Chapter 6.

Chapter 2

Background and Related Work

Matrix and vector algorithms form the core part of many numerical computations. Broad use of these algorithms motivated programmers to develop these routines in the form of precompiled routines or libraries. These library routines increased the programmers' efficiency as well as their confidence in the correctness of the programs. With the introduction of parallel systems and implementation of many programming languages on these systems, the need for new libraries recurred. These new libraries facilitate the programming of parallel systems by encapsulating many of the details in the library routines. Henceforth, numerous researchers have been studying the design of parallel libraries for multicomputers. These libraries have also been customized for integration into programming environments [6, 22] based on languages such as HPF [52, 67], Fortran 90 [1, 20, 68, 63, 81], Fortran D [38], and Vienna Fortran [8, 7].

ScaLAPACK [12, 23, 35, 13, 24, 12, 9, 32] is one of the most commonly known parallel libraries on multicomputers. This library is a distributed-memory version of the LAPACK [23] software for dense, banded, and sparse matrix computations. The design goals of ScaLAPACK are scalability, portability, flexibility, and ease-of-use. In sequential and shared memory versions of this library, performance is enhanced by efficient use of the hierarchical memory through data reuse by avoiding frequent cache reloads. This is achieved by describing the computations using block oriented matrix-matrix operations known as the Level 3 BLAS [11]. In the distributed-memory version of ScaLAPACK, block forms of the algorithms are used to reduce the frequency of communication.

Another distinguished parallel library is the CMSSL [40, 41, 44, 43, 42] of Thinking Machines Corporation. The design goals of CMSSL are performance, scalability, robustness, and portability. Routines to handle data distribution provide data distribution independent functionality in CMSSL. Careful scheduling, data motion, and automatic algorithm selection at runtime are the techniques employed to achieve performance. CMSSL routines can be invoked from languages that support array syntax.

In this chapter we unravel the parallel library specific design issues and the measures taken in the current systems to resolve them. Example programs and performance data are presented, where necessary, to form a basis for comparison with our design in this thesis. The examples are based on the integrated environment of HPF and ScaLAPACK or CMSSL.

2.1 Parallel Libraries

In design of parallel library routines, the *transparency* on the side of the programmer and a *suitable call interface* does not come for free. The library designer must encapsulate many details of the parallel computation to achieve this task. The interoperability of the compiled code and the library routine is a critical area in the design of the system. The analogy with sequential programming is the stack model used in the sequential library calls. The caller and the callee must abide by certain conventions to ensure a proper library call and return. In high level programming languages, the compiler must ensure the correct call sequence in the code generation phase.

Performance of a parallel library routine depends on factors such as the problem size, system size and characteristics, and mapping. In order for the library routine to maintain the desirable performance for larger problem sizes, its algorithm and the supporting design must be *scalable*.

The library routine must be easily *portable* across different platforms. Porting requires recompilation of the library routine using the native compiler and fine tuning certain parameters in the library routine to take advantage of the system characteristics of the target architecture.

2.1.1 ScaLAPACK and CMSSL Parallel Libraries

ScaLAPACK and CMSSL are the most commonly known parallel libraries in the parallel computing community. In this section we will highlight the main features of these two libraries.

The performance goal has been achieved in ScaLAPACK through elimination of overhead due to load imbalance, data movement, and algorithm restructuring for Cholesky, LU, and QR factorization. The authors propose block matrix versions of these algorithms based on meshes of processors. In the analysis of their algorithms they assume a mesh of physical processors (matching that of the logical architecture) is available. The mapping of the logical processors to the physical processors is not addressed in ScaLAPACK and is considered a machine-dependent optimization issue.

The authors claim that range of use of a library package may be identified by how stable the algorithms are over a wide range of input problems and the range of data structures the library supports. For example, dense, packed, and banded matrices each require a different internal data structure. Ease of use is one of the goals of the ScaLAPACK parallel library. The authors state that ease of use is concerned with factors such as portability and the user interface to the library. In ScaLAPACK portability is claimed using the fact that the library is implemented using a standard language such as Fortran. It is assumed that the system to which the ScaLAPACK library is being ported to has the Level 3 BLAS and the BLACS library. The Level 3 BLAS is the sequential version of the matrix numerical routines and BLACS is the communication library. The BLACS library provides point-to-point communication routines as well as communication routines over rows and columns of the logical meshes. The authors further claim that the user interface of ScaLAPACK can be enhanced if many of the implementation details are hidden from the programmer. The interface to ScaLAPACK has been developed for HPF. When calling ScaLAPACK routines from HPF programs, the programmer must specify the data distribution and redistribution.

CMSSL is another parallel library which consists of a collection of numerical routines. The objectives of CMSSL are scalability across systems and problem sizes, consistency with languages with an array syntax, high performance, robustness, and portability.

CMSSL supports global shared address space as well as node level programming. In the global mode, CMSSL accepts distributed data structures. Internally, the library consists of a set of routines executing on each node and a set of communication functions. CMSSL extracts the data distribution information, selects the best algorithm for both the global and the local computation, and carries out the computation using the local routines and the communication routines. The information about the shape of the arrays and their distribution is passed to the CMSSL routines in array descriptors. Similar to the ScaLAPACK library routines, CMSSL routines perform the computation using the distribution and the granularity defined by the programmer. Mapping of the processes to the processors is not addressed in CMSSL.

Both ScaLAPACK and CMSSL support limited form of virtual architecture programming. These libraries assume the logical processors are arranged in form of meshes. They further assume that these meshes of logical processors are selected to match the physical system. HPF allows definition of logical meshes which match the physical system size using intrinsic functions. However, it does not allow definition of other topologies. The virtual architecture parallel library design allows description of parallel algorithms based on various virtual topologies. Gaining efficiency through granularity adjustment is the job of the programmer. Improving efficiency through mapping cannot be supported by libraries such as ScaLAPACK and CMSSL, since the communication pattern of the algorithm is not relayed to the compiler, where the authors claim mapping should be done.

Both ScaLAPACK and CMSSL are suitable for languages such as HPF where data distribution is defined by the programmer. Our libraries encapsulate data distribution in the library routine and require absolutely no intervention from the programmer. The programmer may use the sequential programming model with calls to parallel numerical library routines based on the design in this thesis.

As it will be described in the thesis, in many cases a parallel library call may deliver better performance if it is run on a part of the available partition. In these cases, the virtual architecture size is smaller than the physical system size. As the future technology reduces communication latency, these calls can take advantage of more physical processors to reduce the execution time of the library routine. Even if the logical architecture definition of HPF is extended to allow definition of any logical mesh of processors, it will still not provide the runtime grain adjustment. Providing such functionality requires the programmer's intervention and it is not a trivial task.

2.1.2 Performance Metrics

In view of the fact that the main purpose in parallel computing is performance, an acceptable design of libraries must adhere to this criteria. Performance is usually measured using *speedup*.

The *sequential execution time* of a routine is the time elapsed between the beginning and the end of the execution of the routine on a single processor system. On the other hand, *parallel execution time* is the time elapsed from the start of the execution of the routine on a multiple processor system until the last processor is done. Speedup of a parallel algorithm is defined as the ratio of the parallel execution time of a program to the execution time of the best existing sequential program. There may be several algorithms to solve the same problem sequentially. It is, however, reasonable to consider only the fastest sequential algorithm in the computation of speedup. Assuming $T_{Seq}(n)$ is the runtime for the best existing sequential algorithm for a problem of size n, and $T_{Par}(n,p)$ is the parallel execution time of the problem of size n on p processors, speedup, S, can be defined as:

$$S(n,p) = \frac{T_{Seq}(n)}{T_{Par}(n,p)}$$

Speedup S gives us an insight as to how well we have utilized the p processors in the computation. Theoretically, speedup greater than the number of processors is unattainable. Gaining such speedup is a contradiction, since it implies that one could emulate the parallel processors to obtain a better sequential execution than the one used for the computation of speedup. In practice, however, speedup greater than the number of processors is sometimes observed. This may occur due to the nature of the sequential algorithm or hardware characteristics. For example, searching a tree for an element may result in superlinear speedup depending on the location of the element in the tree, or the memory limitations of the program, such as the cache size, may significantly degrade the sequential execution time of the program. In the latter example, the parallel version may meet the memory requirements better since each processor holds a smaller data partition. In general, one desires linear speedup regardless of the number of processors used.

The ratio of speedup and the number of processors can be used to determine the fraction of time that the processors are effectively employed to solve the problem. This ratio is known as the *efficiency, E.* Therefore, efficiency is defined as:

$$E(n,p) = \frac{1}{p}\frac{T_{Seq}(n)}{T_{Par}(n,p)} \tag{1}$$

where n and p are as previously defined. The value of efficiency varies between 0 and 1. Speedup of p results in an efficiency of 1.

In scientific computations the sequential execution time is usually proportional to the floating point operation count. Therefore, the performance, G in operations per second, can be defined as:

$$G(n,p) = \frac{p}{t_{calc}}E(n,p) \tag{2}$$

where t_{calc} is the time for a floating point operation. Scalability is measured using the rate at which the problem size must grow in order to maintain a constant efficiency when the system size grows. An algorithm is *highly scalable* if the efficiency depends on the problem size and number of processors only through their ratio. In other words, in such algorithms if the problem size and the system size are increased at the same rate, efficiency remains constant. Other definitions of scalability will be shortly discussed.

Speedup and efficiency are two measures that give us an insight on how well we have utilized the parallel system. Based on this definition, on a single processor both speedup and efficiency are one, however as the number of processors increases, a diminishing return is observed. In other words, the speedup will be lower than the number of processors, and therefore the efficiency drops below one. To increase the speedup and efficiency, the problem size must be increased.

The term *scalability* has been widely used to express the system behavior when changes to the problem size or system size occur. Scalability has significant importance in parallel computing, because it provides information as to what algorithm is the most suitable for a specific architecture and allows one to predict the behavior of algorithms on specific machines when very large problems are used. This information is of high importance in design of parallel libraries where the problem instance size and the physical system size are only known at runtime. The information can be used by the library routine to make decisions on what algorithm would provide the fastest execution time[1].

In [31] scalability is defined using *isoefficiency* metric. Using this metric, an algorithm is analyzed for the relationship between the system size and the problem size. Many algorithmic and architectural factors that affect the execution time of the algorithm can be captured in a single expression. As it was previously described, in most parallel programs, as the number of processors increases, efficiency drops and as the problem size is increased, efficiency increases. Using isoefficiency function, one can formulate the rate at which the problem size must increase with respect to the system size in order to maintain a constant efficiency. Algorithms that require the problem size[2] to increase at the same rate as the system size to maintain constant efficiency, are said to be highly scalable. On the other hand, algorithms that require the problem size to grow at a faster rate than the system size are said to poorly scale. These algorithms will require memory beyond the capacity of the underlying architecture to deliver constant efficiency.

A parallel algorithm incurs overhead due to communication, blocking receives, link contention, and load balancing. The accumulation of these times in the execution of a parallel program is called the *overhead*. The overhead, T_{OH}, is a function of the problem size, system size and topology, given a specific mapping of the virtual processors to the physical processors. Considering the overhead, the speedup and efficiency can be restated as:

$$S(n,p) = \frac{T_{Seq}(n)}{\frac{T_{Seq}(n)+T_{OH}(n,p)}{p}}$$

$$E(n,p) = \frac{1}{1+\frac{T_{OH}(n,p)}{T_{Seq}(n)}} \tag{3}$$

Assuming the number of sequential operations and the time for a single operations are $W(n)$ and t_{comp} respectively, equation 3 can be restated as:

$$W(n) = \frac{1}{t_{comp}}\left(\frac{E(n,p)}{1-E(n,p)}\right)T_{OH}(n,p) \tag{4}$$

For a given algorithm and architecture, one can obtain $T_{OH}(n,p)$ which, in turn, shows how W must grow with respect to p to maintain a constant efficiency. For scalable algorithms, the lower the rate of increase of W, the more scalable the algorithm is. The ideal case is when W must grow linearly with p. Let's consider the example of matrix multiplication on a torus. For simplicity, assume a physical torus of p processors is used to perform an $n \times n$ matrix multiplication. The algorithm

[1]Routines based on selection of algorithms are known as polyalgorithms.

[2]problem size here refers to the number of operations in the sequential program. The author defines the problem size in this form to have a uniform meaning of problem size.

is described in chapter 3. The number of operations required by the best sequential algorithm [3] is n^3. In the parallel algorithm, during each iteration $\left(\frac{n}{\sqrt{p}}\right)^3$ operations are performed by each processor to compute the new values for the resulting submatrix, followed by two communication operations to send the operand submatrices to the neighbors. Therefore, during each iteration, the total number of operations performed by all the p processors is $\left(\left(\frac{n}{\sqrt{p}}\right)^3 + 2\right)p$. Over $\sqrt{p}$ iterations, a total of $n^3 + 2p^{3/2}$ operations are performed, out of which n^3 operations are useful work. The overhead is, therefore, $\theta(p^{3/2})$. The number of operations by the sequential algorithm can then be stated as $W = 2Kp^{3/2}$, where K is $\frac{1}{t_{comp}}\left(\frac{E(n,p)}{1-E(n,p)}\right)$ in equation 4. If the number of processors is increased from p to p', where both $\sqrt{p}$ and $\sqrt{p'}$ divide n, then W must increase by a factor of $(p'/p)^{3/2}$ to maintain the same efficiency. Another way to view this is that increasing the number of processors by a factor of p'/p requires the number of operations, W, to be increased by a factor of $(p'/p)^{3/2}$ to increase the speedup by a factor of p'/p. An increase of $(p'/p)^{3/2}$ in W implies an increase of $(p'/p)^{1/2}$ in n. For instance, increasing the number of processors from 16 to 64 mandates doubling n to maintain a constant efficiency. This results in an increase of a factor of 8 in the number of operations, W. Algorithms that require an increase of W less than the rate of change in p to maintain a constant efficiency, will not sustain the efficiency as p increases beyond certain limits, because eventually the number of processors supersedes the number of operations. Therefore, some processors will be idle, and drop in efficiency is not escapable. Therefore, $\theta(p)$ is the lower bound on the isoefficiency function. This lower bound is imposed by the degree of parallelism of the underlying algorithm. For example, in the matrix multiplication algorithm, the total amount of computation is $O(n^3)$, however, each iteration uses values from the previous iteration of the same processor. In other words, the n intermediate values of each processor have to be computed one after another. Thus, at most $\theta(n^2)$ processors can be busy at a time. For this problem the degree of parallelism is $\theta(W^{2/3})$. So, given p processors, W must be at least $\theta(p^{3/2})$ in order to use all the processors. This is the lower bound on the isoefficiency function. The optimal value for isoefficiency function due to degree of parallelism is $\theta(p)$, and it occurs when the algorithm's degree of parallelism is $\theta(W)$. If the algorithm's degree of parallelism is less than $\theta(W)$, then the isoefficiency function due to parallelism is greater than $\theta(p)$. In this case, the accumulative isoefficiency function will be the maximum of isoefficiency due to parallelism or communication. For our matrix multiplication example, the isoefficiency function due to degree of parallelism and communication is $\theta(p^{3/2})$. Algorithms that have an isoefficiency of $\theta(p)$ are ideally scalable.

In [75] and [73], Sun and Rover propose the *isospeed* metric of scalability. Their scalability analysis has been applied to several problems, and further supported by experimental results on the nCube 2 and the MasPar MP-1. They claim that isospeed scalability metric provides a quantitative measure of performance that other metrics fail to provide. The authors further claim that the *fixed-size speedup*, *fixed-time speedup*, and *memory-bound speedup* [74] are all based on the conventional definition of speedup which is deficient. The isoefficiency metric [31] is more advanced than these models, however, it is still implicitly tied to speedup.

[3] Although there are other algorithms for matrix multiplication with lower time complexities, these algorithms tend to have very large constants. Therefore, we use the conventional sequential matrix multiplication for the computation of speedup.

Sun and Rover address the issue of how the system size affects performance and what the costs are to maintain the performance. They believe that the metric should be quantitative and reflect the actual goals of parallel computing. Scalability must be a function of the algorithm and the machine. Both algorithm and the machine introduce parallel overhead. Fast execution time or speed is the major objective in parallel computing. Speed is defined as the amount of work divided by time. The work is usually quantified by the number of floating point operations. *Average speed* is the achieved speed of a system divided by the number of processors. Scalability is defined for an algorithm-machine combination as its ability to maintain a constant average speed when system size increases. The cost of maintaining this average speed is an increase in the problem size. This increase is a quantitative measure of scalability. If W is the amount of work for an algorithm when p processors are used, and W' is the amount of work for $p' > p$ processors to maintain a constant average speed. The scalability from p to p' is defined as:

$$\psi(p, p') = \frac{p'W}{pW'} \tag{5}$$

In the ideal case, $W' = \frac{p'W}{p}$, when $\psi(p, p') = 1$. This occurs when the algorithm does not require any communication and the work is evenly divided among the processor. Generally, $W' > \frac{p'W}{p}$ and $\psi(p, p') < 1$. In order to define a unique scalability for an algorithm and machine pair, the authors define the initial speed as the maximum speed reached by the single processor execution of the algorithm on the machine as the problem size increases toward infinity. The scalability is then defined as:

$$\psi(1, p') = \frac{T_1}{T_{p'}} = \frac{\text{Sequential execution time with problem size } W}{\text{Parallel execution time with problem size } W' \text{ and } p' \text{ processors}} \tag{6}$$

This formula is based on constant speed metric of scalability. The primary difference with the conventional scalability based on speedup is that this metric is not based on the fixed problem size model.

Based on the commonly known definition of scalability, an algorithm is considered highly scalable if it can maintain constant efficiency as the system size and problem size linearly increase. This definition of scalability, however, does not capture the execution time of the parallel algorithm. Based on this definition, if an algorithm maintains a low efficiency when the system size and the problem size vary as stated above, the algorithm is considered highly scalable. Obviously, such algorithms are not desirable for library implementation. If a single algorithm is to be used for library implementation, it is important that the algorithm maintains a constant high efficiency over a wide range of problem sizes. Once the algorithm is selected, the major objective in its design will be to minimize its parallel execution time even for range of problem sizes that do not deliver desirable efficiency. On the other hand, if multiple algorithms (also known as polyalgorithms) are to be used for a library routine implementation, range of scalability (at high efficiency) of the algorithms can be examined. Suitable algorithms can then be selected to cover a wider range of system and problem sizes. Our parallel library design provides support for the use of algorithms based on runtime grain adjustment. ScaLAPACK and CMSSL have shown that algorithms based on block partitioned matrices have a wider range of scalability and can maintain a higher efficiency than the ones based on fine grain.

Operation	Mflop/s per node	Efficiency
Local		
l_2-norm	126	98
Matrix-vector	115	90
Matrix-matrix	115	90
Global		
l_2-norm	126	98
Matrix-vector	80	63
Matrix-matrix	83	65
LU-factorization	61	48
Unstructured grid	37	29

Table 1: Local and global performance per node for some library routines in CMSSL

Although we agree with the claims and proven results in ScaLAPACK and CMSSL, we claim that the range of scalability of the algorithms based on dynamic grain adjustment is wider and they can often maintain better efficiency. A detailed description of the effects of granularity and the techniques to reduce the communication overhead of a parallel program will be covered in this and the upcoming sections.

Maintaining desirable efficiency has been the primary goal in parallel libraries such as ScaLAPACK. Authors in [12] have discussed the design of ScaLAPACK. They have designed the numerical library routines using the block form which results in scalable execution on multicomputers. The authors define a more general form of the distribution function using a block factor. Their major objective is to develop a scalable solution to the linear algebra problems on multicomputers. They redefine some of the linear algebra functions using block matrix distribution and ensure locality of reference at a node after each communication step. The authors fail to address the issue of redistribution across parallel library routines. ScaLAPACK was designed to interoperate with languages such as HPF. In HPF, grain size, distribution of data, and redistribution is defined by the programmer, leaving very little flexibility for the library routine to optimize for performance. The mapping of virtual to physical processors, which is transparent to the HPF programmer and library routine user, is not addressed in ScaLAPACK.

Johnsson [41] describes the design of the CMSSL library which consists of 250 overloaded numerical routines, nearly equivalent to 1000 routines in the conventional libraries. Robustness with respect to performance and numerical stability is the objective in the design of CMSSL. In CMSSL, the author claims that a change in the problem size or the system size still delivers desirable efficiency. This is achieved through the use of block matrix algorithms as well as poly-algorithms for the implementation of a library routine. A change in the system size does not require recompilation of the program since many systems are changing such that the requests for partitions may change from time to time and also during the course of execution of a program.

Table 1 shows the performance of some global and local CMSSL routines on the Thinking Machines CM-5. The parallel version of the routines are referred to as global, whereas the sequential node version of the routine is called local. One important note here is the use of the unstructured grid application where the efficiency has significantly dropped. The drop in speed, in this application, is due to the sequential portions of the program.

In ScaLAPACK, scalability has been studied under constant granularity, varying problem size, and varying system size. The primary reason for this definition of scalability in ScaLAPACK is the fact that the library routine does not have control over varying the grain size. However, this is not the case with the library design presented in this thesis. In our library routines, grain size is determined at routine invocation based on the system size and the problem instance size. Therefore, it is easier to maintain desirable efficiency and speed. Furthermore, our libraries increase performance by selecting optimal runtime mapping of the virtual processors to the physical processors.

The performance results from ScaLAPACK have been presented in [12] for the distributed-memory LU factorization on the Intel Touchstone Delta system. The Delta system is a distributed-memory i860-based MIMD computer, in which the nodes are connected using a two dimensional mesh. The authors have conducted some experiments over a wide range of problem sizes and system sizes and concluded that, on average, a block size of 5 is close to optimal. Therefore, in all their analysis, they have been using this value for the block size. Their initial investigation was on how performance depends on the relative dimension sizes of the logical architecture (always a mesh) for the same number of processors. Figure 1 shows three graphs that exhibit the performance metrics of the ScaLAPACK LU decomposition library routine. Figure 1(a) shows that for different processor templates there is very little difference in performance for different problem sizes. Figure 1(b) shows the performance for different system sizes. It is clearly shown in the graph that for a fixed block size the system behaves poorly when the problem size is large and the system size is small. This graph shows precisely where the ScaLAPACK routines fail to deliver desirable performance, that is, on small systems with small grain size and large problem size. Finally, 1(c) shows the performance of the library routine for different grain sizes. It is again shown that for small grain sizes scalable performance is not maintained as the number or processors increases. This degradation in scalability is because of increasing overhead of communication. On the other hand, for large grain sizes the performance linearly increases with the changes in the system size.

In the remainder of this chapter we discuss predominant concepts relating to parallel library design. In addition, we identify intricacies in the parallel library design and the current research activities to overcome them.

2.1.3 Conformance Model

When a program invokes a procedure, both the caller and the callee must agree on how to pass the parameters into and out of the procedure. This agreement between the caller and the callee is known as the *procedure call model*. This agreement is realized in implementation with an *entry sequence*, which runs before the procedure body is executed, and a *return sequence*, which runs after the procedure body is executed. When a data parallel program calls a procedure, there is an additional complication. This complication originates from the fact that upon a call, the procedure parameters may be distributed across the nodes of the system. In data parallel programming, the

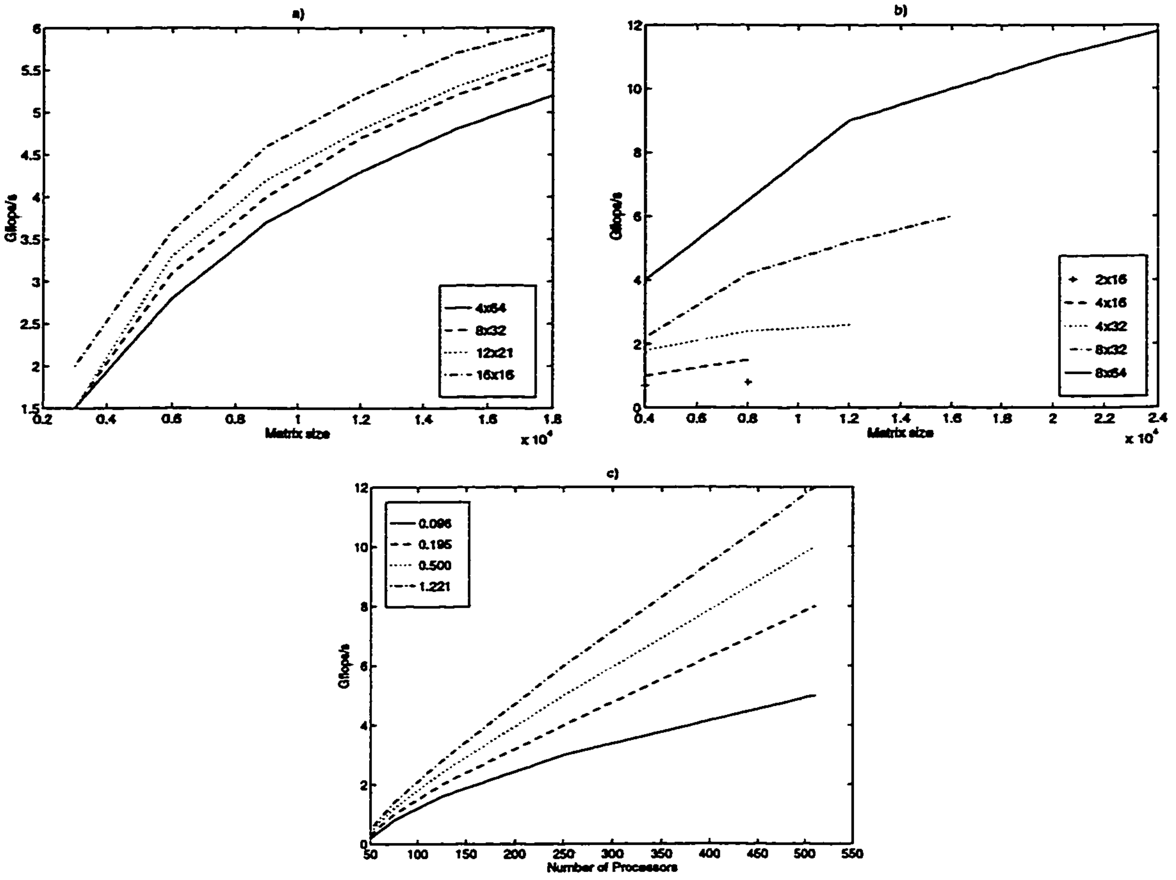


Figure 1: Performance metrics of ScaLAPACK LU decomposition library routine. a) Performance for different processor templates. b) Performance for different number of processors. c) Isogranularity curves where labels are reduced by a factor of a million.

procedure call model must be enhanced to specify the agreement on the distributed parameters. The agreement may be that a library routine requires a specific distribution upon entry or may accept any distribution. On the other hand, the routine may restore the distributed parameters back to their initial distribution upon return, or it may not guarantee any particular distribution.

Yang and O'Hallaron [82] discuss different parallel call models in their paper. These models relate to the distribution of initial data on entry to and exit from parallel procedures. Depending on whether the data conforms to some distribution or inherits a distribution, different call models are described and their impact on the quality of the code produced by the compiler is discussed. The authors go on discussing the impact of compile time distribution information on the redistribution cost. Table 2 represents the space of the procedure call models for data parallel programs. Each model has two components in the procedure call overhead, namely quality of the compiled code and unnecessary redistributions.

In conform-on-entry the callee requires that the distribution of data conforms to a specific distribution or a set of distributions upon entering the procedure. This model can either be enforced by the programmer or the compiler. This process will typically require redistribution of data from

	Conform models	Inherit models
Entry models	Conform-on-entry	Inherit-on-entry
Return models	Conform-on-return	Inherit-on-return

Table 2: Space of the call models

some source distribution (prior to the entry sequence) to a target distribution (specified by the conformation rule). The conformation rule information can be utilized by the caller's compiler to optimize the overall communication cost as well as producing better redistribution code. On the callee's side, the distribution information may affect the quality of the generated code.

In inherit-on-entry model, there is no conformation rule between the caller and the callee. The callee can handle any distribution upon entry. Hence, the produced code will not be efficient on the callee's side. An advantage of this model on the caller's side is that the caller does not have to worry about the distribution of parameter prior to the call. This will simplify the compiler or the programmer's task. On the callee's side, the compiler or the programmer's task will, however, be more difficult since the distribution of data is unknown and any operation that requires distribution knowledge will be inefficient.

Conform-on-return ensures that the distribution of data abides by certain rules prior to return to the caller. Like conform-on-entry, this model allows useful information to be relayed to the compiler which assists in efficient code generation.

In the last model, inherit-on-return, the caller inherits the distribution of data from the callee. The advantage of this model is that unnecessary redistributions may be avoided, since the data may no longer be needed after the caller's resumption. On the other hand, if the data is used after the call, the distribution information is not available at compile time and the generated code may be inefficient.

We have used the parallel procedure call models *conform-on-entry* and *conform-on-return* described in [82]. Although the library routine writer may hard-code the data distribution and accumulation in the library, the information on initial and final data layout is stored in the global memory of the physical processors for later data redistribution. We will explain this in Chapters 3 and 4.

2.1.4 Delayed Granularity Adjustment and Delayed Mapping

When a regular parallel computation is to be described on a virtual architecture, it is easier to describe the algorithm at the finest level of granularity and leave the degree of parallelism as high as possible. Once the parallel program is mapped onto a multicomputer, low grain sizes result in emulation of many virtual processors in a physical processor. Although many virtual processors are collapsed onto the same physical processor, local communication still goes through the router and back to the same node[4]. Hence, the abundance of small messages results in poor performance. Another issue is the overhead of emulating the virtual processors. One approach often used to deal

[4] A local mailbox is a more efficient solution for intraprocessor communication.

with this overhead is threading [72, 49]. In this approach, variables of the virtual processors are expanded and placed in the common memory shared by all the threads. The use of threads gives the compiler more flexibility in producing code that results in efficient scheduling of the virtual processors. Threading can be used for both SIMD and SPMD parallel computation models. In SIMD model the threads must be synchronized at specific points to maintain the semantics of the computation and take advantage of the MIMD execution. Communication in the virtual topology and deadlock free scheduling are issues that must be carefully considered in implementing this technique. Moreover, threading is an attractive technique only when the number of virtual processors per physical processor is small. When the number of virtual processors increases, communication among local virtual processors induces significant overhead in the program execution time due to the fine granularity of the parallel program.

Scaling down virtual architecture programs may be done either at algorithm design time, at compile time, or at execution time. At algorithm design time, it is more effective in performance gain but would require programmer's intervention. If the programmer does not have knowledge of the target system size and the problem size, this technique may not work, considering performance is the primary goal. Scaling down at compile time, on the other hand, is not as effective in performance, however the programmer's task is greatly reduced. The scaled down form still suffers from the same problems as the previous technique. Scaling down the virtual architecture size at execution time is the most flexible among the three techniques. The algorithm is described in a parameterized form, which is then scaled down at runtime taking into consideration the physical system and the problem instance attributes. The parameterized scaled down virtual architecture will then allow for determining a balance between the granularity and the parallelism of the program. The same argument holds for mapping of the virtual processors to the physical processors. Mapping must be performed once the virtual architecture program is scaled down. Lack of knowledge from the physical system attributes prior to the program load time necessitates delaying this mapping to the load or execution time.

ScaLAPACK routines perform the computation at the level of granularity described by the caller. As a matter of fact, the granularity of computation is a parameter to the library routines. This parameter is passed implicitly using the components of the matrix that describe the attributes of the distribution. Therefore determining the granularity of computation is not an integrated part of ScaLAPACK.

2.1.5 Data Distribution

The layout of data in the hierarchical memories of a multicomputer is critical to the performance of the program. In many numerical libraries, algorithms are described in a form to minimize data movement between layers of the memory. The algorithms are described in block form to maximize the ratio of floating point operations to memory references.

ScaLAPACK [12, 6] has used block algorithms to improve the performance of the parallel library routines. Although there is a resemblance between the technique used in ScaLAPACK and the one presented in this thesis, our technique is more flexible in obtaining desirable performance. In

m	0	1	2	3	4	5	6	7
k	0	0	0	1	1	1	2	2
i	0	1	2	0	1	2	0	1

a) Block

m	0	1	2	3	4	5	6	7
k	0	1	2	0	1	2	0	1
i	0	0	0	1	1	1	2	2

b) Cyclic

Figure 2: Block and cyclic decomposition example for 8 elements over 3 processors

ScaLAPACK the application programmer is responsible for distributing data over the processors. A vector of length M is, for instance, distributed over a set of p processors by assigning the vector entry at index m to the kth processor, where it is stored as the ith entry in a local array. Therefore, the distribution can be regarded as mapping of the global index, m, to an index pair, (k, i). Note that the value of m lies in the range 0 to $M - 1$, and k lies in the range 0 to $p - 1$.

Similarly for matrices, the p processors are considered to form a grid of P rows and Q columns. The distribution of the $M \times N$ elements can be described as the tensor product of two decompositions, μ and ν. Therefore, if $\mu(m) = (k, i)$ and $\nu(n) = (l, j)$, then the entry (m, n) is assigned to the processor (k, l) at index (i, j) in the local array.

The authors in [12, 6] describe two common decompositions called *block* and *cyclic* decompositions. The block decomposition, λ, assigns contiguous entries in the global vector to the same processor, whereas scattered decomposition, δ, assigns consecutive entries in the global array to different processors. Where $L = \lceil M/P \rceil$, distribution functions λ, and δ can be defined as:

$$\lambda(m) = (\lfloor m/L \rfloor, m \bmod L) \tag{7}$$

$$\delta(m) = (m \bmod P, \lfloor m/L \rfloor) \tag{8}$$

Figure 2 shows two examples of block and cyclic mapping in one dimension.

The decomposition strategy described in ScaLAPACK scatters blocks instead of single elements. This is a more general form of distribution, since if block size of one is used, all other block and scattered decompositions can be reproduced. In block distribution, the global index, m, maps to an element using a triplet $\mu(m) = (k, t, i)$, where k is the processor position, t is the block number, and i is the local index within the block. This can be written as,

$$\zeta_r(m) = \left(\left\lfloor \frac{m \bmod T}{r} \right\rfloor, \left\lfloor \frac{m}{T} \right\rfloor, (m \bmod T) \bmod r \right)$$

where $T = rP$. With $r = 1$, this reverts backs to the original scattered decomposition. A block decomposition is obtained if $r = L$. The block scattered decomposition of a matrix is the tensor product of two block scattered decompositions, μ_r and ν_s which results in scattered blocks of $r \times s$. Figure 3 shows an example block cyclic mapping in one dimension.

In ScaLAPACK a matrix is an object composed of three components, *data part*, *decomposition part*, and *storage part*. The library routine receives these structures as parameters and performs the computation. These components describe the processor grid, matrix size, parameters of the block scattered decomposition, and the actual user data buffer. The application program, or the compiler

m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
k	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2
t	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3
i	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

k	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
t	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3
i	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
m	0	1	6	7	12	13	18	19	2	3	8	9	14	15	20	21	4	5	10	11	16	17	22

Figure 3: Block cyclic decomposition example for 23 elements over 3 processors for a block size of 2 (top) and the corresponding inverse mapping from the triplets to the global indices (bottom).

is responsible for identifying the contents of these structures based on the data distribution at the time of the call to the ScaLAPACK routine. Figure 4 illustrates the structure of a matrix object in ScaLAPACK.

There has also been some work in the area of an interface to ScaLAPACK. In [6], the authors describe a compilation and runtime system for HPF to provide an interface from HPF to ScaLAPACK. In this system, called the ADAPTOR, the application programmer describes the data distribution using HPF compiler directives. These directives are compiled into some runtime routines that distribute the data onto the processor template. A call to a ScaLAPACK routine is transformed such that the parameters to the library routine are the matrix structures rather than the matrices themselves. The information within these structures is obtained using HPF runtime library routines. ADAPTOR has two major components, the first of which compiles an HPF program into a message passing program and the second one is a runtime library, called DALIB, that is linked in with the message passing program. An important distinction between our design of virtual architecture parallel libraries and those of ScaLAPACK and CMSSL is that our library routines deduce the size of the virtual architecture from the problem instance size. In other libraries the size of the logical architecture is defined by the programmer and the library routine does not deduce the size. Our parallel libraries perform reduction in the size of the virtual architecture transparently. In other library systems this reduction in size is done by the programmer prior to calling the library routine. Finally, our library system is designed to support virtual architecture algorithms based on any topology and benefit from the topology information, whereas other libraries rely on the programmer to consider a logical mesh for the implementation of the virtual architecture parallel algorithm. The library routine does not consider the communication pattern of the algorithm and the effect of its mapping onto the physical architecture.

Various researchers have worked in the area of automatic distribution in HPF. Even though this may appear related to the automatic distribution described in this thesis, this work is centered around identifying the optimal cyclic factors in an HPF program. Originally in HPF, the application programmer must have defined the value of x in cyclic(x) distribution. Recent work such as [10]

```
Matrix{
  DATA_PART_PTR{
    pointer to matrix element values
    total number of rows in matrix
    total number of columns in the matrix
    total number of rows of r by r blocks in the matrix
    total number of columns of r by r blocks in the matrix
    number of rows of r by r blocks in each processor
    the template row containing the first matrix block
    the template column containing the first matrix block
    pointer to the user supplied buffer
  }
  DECOMPOSITION_PART_PTR{
    block size
    the number of rows of processors in the template
    the number of columns of processors in the template
    the ID number of the processor to the left in the template
    the ID number of the processor to the right in the template
    the ID number of the processor below in the template
    the ID number of the processor above in the template
  }
  STORAGE_PART_PTR{
    column or row major storage of blocks
    column or row major storage of elements within the block
    offset between successive elements in the same row
    offset between successive elements in the same column
    offset between starts of successive blocks in the same row
    offset between starts of successive blocks in the same column
  }
}
```

Figure 4: A matrix object in ScaLAPACK

formalizes the automatic data distribution using graph theory. Given an HPF program with distribution and redistribution primitives, the authors construct a directed edge-weighted graph called the alignment-distribution graph(ADG). Nodes of the ADG describe program operations, whereas the edges connect the definition of array objects to their use. Alignment and distribution are attributes of array objects. A solution to the automatic distribution problem is the labeling of the edges of the ADG with distribution parameters.

2.2 Application Domain

The hierarchical memory of computer systems advocates algorithms that have locality of reference. In the implementation of these algorithms, once the data is close to a processor it is used as much as possible before it is sent out to another processor. In distributed-memory multicomputers non-local memory is added on top of the conventional memory hierarchy. Accessing non local memory is several hundred times more expensive than a local memory access [25]. This renders use of many algorithms based on block partitions more appealing [29, 30]. In these algorithms a processor requests a block of data from another processor, performs its local computation using the block, and sends the block to other processors for further computation. The block algorithms also have the flexibility that the block size may be fixed dynamically based on the system size and the communication parameters prior to commencement of the parallel phase. Larger block sizes not only reduce the communication overhead of the parallel program, but also allow a higher degree of overlapping between the computation and the communication [25].

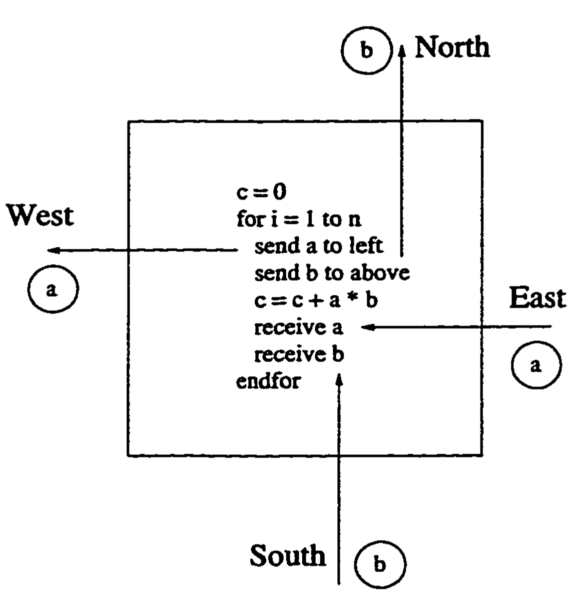
Many scientific computations can be described using the block matrix form. Among these are vector-matrix operations, matrix-matrix operations, Jacobi Relaxation, and LU Decomposition. The level 3 BLAS [24] routines are based on such algorithms. These routines form the core part of the ScaLAPACK [12] parallel library.

Consider multiplication of two $n \times n$ matrices A and B on a torus of size $n \times n$. Initially each processor is assigned an element of A and an element of B, and a local variable is set to 0 in all the processors. Then the algorithm goes through n iterations. In each iteration, the local values of A and B are multiplied and added to the local variable c. The local value of A is passed to the left neighbor and the local value of B is passed to the neighbor above. In return, new value of A and B are received from the right neighbor and the neighbor below, respectively. At the end of the nth iteration every processor holds a value of the result matrix C.

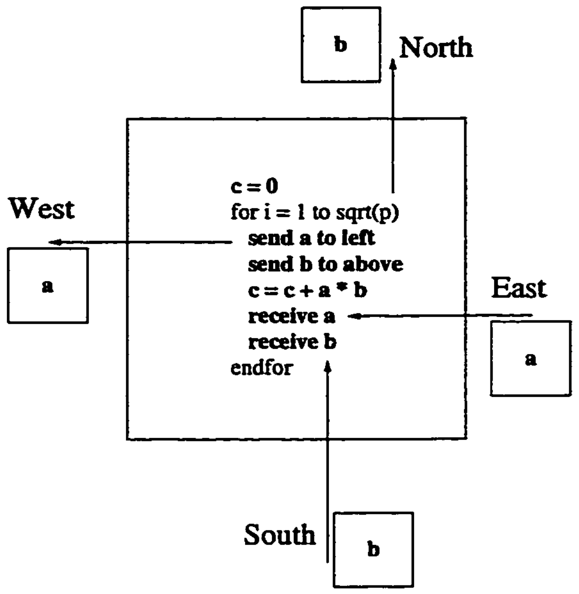
Each processor in the torus performs $\theta(2n)$ floating point operations, $\theta(n)$ **sends**, and $\theta(n)$ **receives**. Finally every processor returns its value of c to the host or I/O processor.

In block matrix multiply, each processor holds a partition of A and B, and has a local partition for C. The algorithm behaves similar to the fine grain one with exception that the computation at each node consists of a matrix multiplication and addition followed by a matrix assignment. Assuming the size of the partition is s, the number of floating point operations by each processor is $\theta(s^3 + s^2)$. The number of **sends** and **receives** are both $\theta(n/s)$.

Figure 5 demonstrates two types of matrix multiplication. Figure 5a demonstrates the multipli-



(a) Fine Grain Matrix Multiply



(b) Block Matrix Multiply

Figure 5: Fine grain and block matrix multiply

cation of two $n \times n$ matrices on a torus of $n \times n$ virtual processors. The overall computation consists of n iterations. During each iteration, one scalar value is communicated to the left neighbor and one to the upper neighbor. After performing the scalar multiplication and addition, scalars for a and b are received from the right and the lower neighbor, respectively. Figure 5b demonstrates the same computation using $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ processors. In this case each processors will hold three partitions of sizes $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$. The overall computation consists of $\sqrt{p}$ iterations. During each iteration the partitions of **a** and **b** are communicated to the upper and left neighbors, respectively. Local computation which is the matrix multiplication of the partitions is performed following by receipt of the **a** and **b** partitions from the lower and right neighbors, respectively. If Figure 5b, statements in bold indicate that the operation involves blocks of data as opposed to scalars. Consider these two algorithms as far as the amount of overlap between computation and communication is concerned. In the fine grain program, by the time the control comes back from the **sends** to the program, the messages are already at the destinations, so overlapping of computation and communication is meaningless. However, in the block matrix algorithm, when the **sends** return to the program and continue with the computation, the messages will still be in transit (if the messages are large enough). Therefore, the program benefits from overlapping of computation and communication.

2.3 Programming Paradigms

Different programming paradigms trade off between ease of programming and efficiency of programs depending on the target architecture. Parallel programming languages are designed to support the programming paradigms. An algorithm may be easy to describe in one language but more difficult in another. On the other hand, the efficiency of the algorithm depends on how well the programming paradigm matches that of the underlying architecture. A closer match results in more efficiency, at the expense of more programming effort. A mismatch of the programming paradigm, and therefore the supporting language, and the architecture allows the programmer to describe the algorithm at a more abstract level but may result in a more inefficient execution.

A characterization of parallel programming is based on whether the parallelism is *explicit* or *implicit* in the program. In explicit parallel programming, the programmer describes exactly how the processes communicate to solve a given problem. Therefore, the compilation process is very simple, but the programmer's task is more complex. *Implicit* parallel programming, on the other hand, allows the programmer to describe the computation using the conventional sequential programming paradigm and performs compile-time analysis on the code to detect potential parallelism. The compiler's task is very complex and in some cases may fail to provide desirable performance. We have used the former approach of explicit programming to describe our parallel libraries for the purpose of efficiency and the latter approach for the user interface to facilitate use of these libraries.

Virtual Shared Memory programming paradigm is one in which the programmer views the computation as a set of processes accessing a pool of shared variables. Although this programming paradigm is more suitable for multiprocessors, it can be implemented on multicomputers. Implementation of this programming paradigm on multicomputers requires analysis by the compiler as to the location of data so that appropriate **send** and **receive** can be inserted in the program for communication. In *Message Passing* paradigm, the programmer views the program as a collection of processes with private local variables. The communication among these processes is done using primitives such as send and receive. Message passing programs are more difficult to write, however they can be optimized and can run more efficiently on multicomputer. This programming paradigm is the most suitable for implementation of parallel libraries for multicomputers.

Data parallelism is the type of parallelism where many data items are subject to the same processing. Many data parallel languages have been developed to support this kind of parallelism. SIMD architectures are very suitable for data parallel programs since the synchronous execution of the instructions is supported by the hardware. MIMD architectures can also be used for data parallel programming but the global synchronization cost may be intolerable. A more relaxed form of data parallel programming is one that does not require synchronization at each instruction. This form is called *Single Program Multiple Data* (SPMD) programming paradigm. In SPMD programming paradigm, synchronization is done only at communication points which are implicit in the algorithm. SPMD programs are suitable for multicomputers since the programming paradigm closely matches the underlying architecture. In contrast to data parallelism, *control parallelism* refers to simultaneous execution of different operations. Pipelining is an example of control parallelism.

Our system is based on implicit parallelism from the user's view as the library call interface resembles that of sequential programming. The library routines are based on explicit message passing SPMD programs. Below, we look at some of the existing languages and systems based on a similar programming paradigm.

Quinn and Hatcher [60, 64, 37, 65] have studied the compilation of C* virtual architecture programs onto the distributed-memory architectures. In compilation of a SIMD C* program to C, they identify the necessary synchronization requirements of the original program. The communication primitives inside loops and the conditionals are pulled out by performing transformations on the program. These transformations are done because of the limitation of the nCube 3200 architecture. In this architecture communication requires participation of all the processors. After these transformations, emulation loops are placed around the sections of code delimited by communication. In addition, Quinn and Hatcher emulate sections of code delimited by communication statements and perform full synchronization at communication points. Quinn and Hatcher describe data level and processor level parallelism in [36]. A parallel SIMD algorithm may be described in finest granularity and then transformed automatically to the contracted form. The contracted form described in [36] is quite different than the one described in this thesis. In [36] the authors transform statements so that all local communication is converted to assignment. At the boundaries, the communication statements remain as they are and some optimization may be applied to combine small messages into large messages. The authors rely on the SIMD programming model of C* on CM5 to define these transformations. A more efficient method is for the programmer to describe the algorithm using larger grain sizes. In most of the virtual architecture algorithms, for a given problem instance size, the size of the virtual architecture can be chosen from a range of possible values. After fixing the size of the virtual architecture, the computation and the communication operations, by each virtual processor, must be adjusted to reflect this size. Our library routines select the size of the virtual architecture at runtime. To support this, the algorithms are described using parameterized grain size and the best grain size is selected at runtime. In such case, the programmer can use the best existing sequential algorithm to perform the computation at each node. Describing a parallel program in a form such that the grain size is not fixed at program design time is useful in design of parallel libraries. A parallel library routine can determine the grain size upon invocation using the problem size and the physical system size. Although Quinn and Hatcher discuss processor level parallelism in [36], the use of this technique in the design of libraries is overlooked in their paper.

Rosing [69, 70] has also considered virtual architecture programming in his thesis. DINO was primarily developed to support virtual architecture programming for data parallel applications. The primary objective in DINO is to create a collection of data mapping libraries that are subsequently used by programmers to distribute the initial data across the virtual architecture. DINO is mainly recognized for its rich mapping library. The contraction of SIMD procedures is done using a similar technique as that of Quinn and Hatcher with some minor added optimizations.

Kali [53] is another language, compiler, and runtime environment that supports shared name space programming. The compiler translates shared name space programs to message passing programs that produce assignment for local communication and message passing instructions for nonlocal data transfer.

2.4 Communication Model

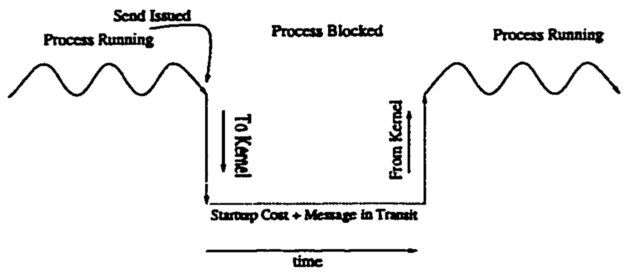
In a distributed-memory multicomputer, processors must exchange data using communication. This is in general available using communication libraries on the processors. The most common form of communication is point-to-point communication where two processors exchange data by executing **send** and **receive** operations. The sending processor makes a system call to the kernel of the node operating system. The kernel prepares the message and delivers it to the network layer. The network layer then transfers the data using the physical layer. The message, then, travels through the network from one node to the next. Once at the destination, the message ripples up from the physical level to the user level into the user buffer.

One common form of message passing primitive used is *blocking* or *synchronous*. When a process calls **send**, it specifies a destination and a buffer to send to that processor. While the message is being sent, the sending processor is blocked. The instruction after the **send** is not executed until the message is completely transferred. Similarly, **receive** must wait until the message is completely received and placed in the message buffer. If the processor is waiting for a message from a specific processor, it must wait until a message from that processor arrives.

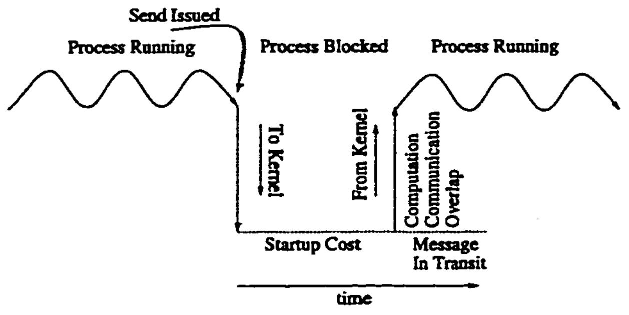
A different form of communication is called nonblocking or asynchronous primitive. A nonblocking **send** implies that the sending processor returns control to the caller immediately before the message is sent. The sending process can then continue computing in parallel with the message transmission. The kernel must ensure that the user buffer is released before returning to the caller, otherwise the user data may be clobbered prior to transmission. Releasing the user buffer may be done by copying it immediately after the call to **send**. The message can then be prepared for transmission while control goes back to the caller and computation continues. Major problem with this scheme is the additional copying of the user buffer to the kernel buffer which may reduce the overall system performance. Usually an additional copying from the user buffer into the hardware transmission buffer is required as well. One other solution is to interrupt the sender when the message has been sent to acknowledge that the buffer can be reused. The cost here is more program complexity but no additional copying is required.

Among the different techniques described above, applications that have potential of overlapping computation and communication can take advantage of nonblocking send with copying. Figure 6(a) demonstrates how a process is blocked during a blocking send, whereas Figure 6(b) demonstrates how the processor is released after the message is ready for transmission. The area of computation and communication overlap is shown in the figure. Since the startup cost is significant, this model favors programs based on large message.

The *startup cost* or *startup latency* is the amount of time from initiating a send by the user program until the message buffer is free for use by the caller. The startup cost is usually in the order of thousand of instructions, which introduces significant overhead in the execution of parallel programs. Table 3 shows the metrics from some of the existing parallel machines. Although the value of startup latency has improved over the past decade, this value is still intolerable. The high value of startup makes large number of small messages unattractive. When a small message is sent from one processor to another, by the time control comes back to the caller, the message has most



(a) Blocking send



(b) Non-blocking send

Figure 6: Blocking and non-blocking send

probably reached the destination. Therefore, no overlap of computation and communication can be done. On the other hand, if large messages are used, when control comes back to the caller, the message is most probably still in transit and the computation following the **send** can be overlapped with the ongoing communication. This is under the assumption that a **send** operation issued by the user program does not return until the user buffer has been copied to the router buffer. If the user buffer is copied immediately after calling **send**, the user program can resume right away. The later approach requires extra buffering, however it significantly reduces the startup cost.

We will examine the effect of granularity in Chapter 3. Large messages support overlapping as described above, but limit the parallelism. Limitation in parallelism may increase the total execution time in two aspects. One is obviously the usage of less processors to solve the problem. The other is the load imbalance that may be introduced at execution time.

2.5 Automatic Data Redistribution

Data redistribution [12, 14] consists of rearrangement of data between two parallel phases of a program. The primary reason to support data redistribution is the fact that different parallel phases of a program possibly require different distribution for good performance. A data distribution suited for one parallel phase may behave poorly for another phase. The two parallel phases may have different topologies and sizes. Initial data distribution comprises a significant portion of a

MPP Models	IBM SP2	Cray T3D	Cray T3E	Intel Paragon	Intel ASCI TeraFLOPS
Large Configuration	400-node 100 Gflop/s	512-node 153 Gflop/s	512-node 1.2 Tflop/s	400-node 40 Gflop/s	4536-node 1.8 Tflop/s
CPU Type	67 MHz 267 Mflop/s POWER2	150 MHz 150 Mflop/s Alpha 21064	300 MHz 600 Mflop/s Alpha 21164	50 MHz 100 Mflop/s Intel 860	200 MHz 200 Mflop/s Pentium Pro
Node Architecture	1 processor 64MB-2GB Memory 1-4.5GB Local Disk	2 processors 64MB Memory 50GB Shared Disk	4-8 processors 256MB-16GB DSM Shared Disk	1-2 processors 16-128MB Memory 48GB shared disk	2 processors 32-256MB memory Shared disk
Interconnect Memory	Omega Network, NORMA	3-D Torus DSM	3-D Torus DSM	2-D Mesh	Split 2-D Mesh
OS Nodes	Complete AIX (IBM Unix)	Microkernel	Microkernel	Microkernel	Light-Weighted Kernel(LWK)
Native Programming Model	Message Passing (MPL)	Shared Variable and Message Passing	Shared Variable and Message Passing	Message passing (Nx)	Message Passing (MPI)
Other Models	MPI, PVM, Linda	MPI, HPF	MPI, HPF	SUMMOS, MPI, PVM	Nx, PVM
Startup Time and Bandwidth	40 μs 35 MB/s	2 μs 150 MB/s	2 μs 480 MB/s	30 μs 175 MB/s	10 μs 380 MB/s

Table 3: Characteristics of the most commonly known parallel systems

parallel program's execution time. Since initial data distribution is done sequentially, it degrades the performance of the program drastically. If data is reused between the phases, it is beneficial to reshuffle the data among the processors to prepare for a subsequent phase. The reshuffling is a parallel operation and eliminates a large fraction of sequential communication operations.

Most of the research work in the area of redistribution is based on HPF. In HPF, data redistribution is explicit. HPF supports block and cyclic mapping previously described. The data redistribution for these primitives is simple and has been looked at in [19], [46], [66], and [77]. Another form of distribution previously described, which is also supported in HPF, is the cyclic(x) mapping. In support of ScaLAPACK library calls from an HPF program, the block-cyclic with an arbitrary block size renders the redistribution more difficult. An HPF program may distribute the data using a block factor prior to a call to a ScaLAPACK routine. The user program may redistribute data to a different block factor prior to calling a second ScaLAPACK routine. The authors in [80], [48], and [77] have looked at this type of redistribution.

The author in [19] discusses the merits of data redistribution on load balancing and phase parallelism. He formalizes data redistribution and examines a special case where this is achieved by local permutation. The author goes on considering orthogonal redistribution which is a special case when every node exchanges a constant number of data items with every other node. This special case produces the largest number of communications in a redistribution phase. Our formalization is a variation of [19], with the difference that the author in this paper does not address the issue of redistribution in the context of parallel libraries.

The work by Kalns and Ni described in [45, 46] is also in the area of data redistribution in HPF programs. In these papers, the authors use virtual processor mapping technique to minimize the number of data exchanges for block to cyclic redistribution. The high level language used is HPF which has directives to perform explicit data redistribution between parallel phases. Their technique is strictly applicable to explicit data redistribution directives, and requires that the source and target virtual processors have the same size and dimensionality. The processor mapping technique used in their papers also has the drawback that it does not preserve local communication. This may result in degradation in performance due to contention in systems based on packet switching. On these systems, the communication cost is proportional to the distance that the message must travel. The difference between our work and [45, 46] is that our data distribution and redistribution is implicit. The distribution information of a parallel routine is maintained as part of the library routine. For virtual algorithms that have a regular communication pattern and the virtual processors communicate with their nearest neighbors, the contracted form tends to be very efficient because of the locality of reference after each communication. A large number of parallel programs fall in this class. Such programs, once contracted, run using coarse grains and data redistribution is also done in coarse grain form. The grain size, which is determined at runtime, is a function of the problem size and the physical system size. Consequently, the size of the virtual architecture can be reduced to be smaller or equal to the physical architecture size. Each physical processor will then be assigned at most one virtual processor. Another difference between our work and Ni's is that since our target domain is the class of regular applications that possess neighboring communication, our contracted form preserves the locality of communication whereas in Ni's work the number of redistribution messages is reduced at the cost of increasing the distance between communicating nodes.

The authors in [77] present algorithms to perform redistribution between different cyclic(x) distributions, as defined in High Performance Fortran. They initially propose optimized algorithms for cyclic(x) to cyclic(y) redistribution where x is a multiple of y or y is a multiple of x. They then propose two algorithms, called the GCD and LCM method, for general cyclic(x) to cyclic(y) redistribution when there is no enforced relation between x and y.

In [47] and [48] the authors also present a scheme to determine the processor sets and the elements involved in the redistribution in special cases where the source and destination block sizes are factors of one another. The general case of redistribution can be expressed in terms of these two special cases. The authors develop a closed form for determining the processors and the elements of redistribution from which they evolve a cost model for array redistribution. It is then shown that a multiphase array redistribution has a lower cost in communication overhead than the single phase approach. This work of the authors is based on the original work in [34] where they utilize tensor product theory on matrices [39] to describe algebraic semantics of regular distributions.

The distribution across library calls in our design requires a scheme that works similar to the cyclic redistribution, however the data redistribution is not explicit in our programs. The potential redistribution is recognized by the compiler and appropriate code is inserted to efficiently redistribute data for the second parallel phase. Redistribution in our library routines is done across parallel phases using information embedded in the library routine. A library routine is passed information about the current distribution of all global data. Using this information and the requirements of the parallel phase, appropriate redistribution is performed prior to launching the parallel phase.

2.6 Examples of Library Systems

In HPF, the programmer can declare several logical architectures of the same size in a single program. For each parallel phase, the programmer can then distribute the data onto a logical architecture using the **DISTRIBUTE** compiler directive. After completing a phase of the parallel program, the **REDISTRIBUTE** compiler directive may be used to reshuffle the data to the appropriate locations to be used in the next parallel phase of the program.

In HPF, logical processor templates are declared using the **PROCESSOR** directive. This directive may only appear in the specification part of the program. The **NUMBER_OF_PROCESSORS** and **PROCESSORS_SHAPE** may be used to inquire about the total number and the shape of the actual physical processors used to execute the program. Following are some legal processor declarations in HPF:

```
!HPF$ PROCESSOR P(10)
!HPF$ PROCESSOR Q(NUMBER_OF_PROCESSORS())
!HPF$ PROCESSOR R(8,NUMBER_OF_PROCESSORS()/8)
!HPF$ PROCESSOR S(1972:1997,-20:17)
!HPF$ PROCESSOR SCALAPROC.
```

Distribution of data in HPF is described using the cyclic(k) which specifies the assignment using blocks of size k in cyclic form. The cyclic(k) form of distribution covers a large class of distributions, two special cases of which are block and cyclic. The former is actually cyclic($\lceil \frac{N}{P} \rceil$), where N is the data size and P is the number of processors, whereas the latter is cyclic(1). For example,

```
      REAL XVECT(10000)
!HPF$ DISTRIBUTE XVECT(BLOCK)
```

specifies that the array XVECT should be distributed across the set of currently defined logical processors by slicing it uniformly into blocks of contiguous elements. If there are 50 processors, the directive implies that the array should be divided into groups of $\lceil 10000/50 \rceil = 200$ elements, with

XVECT(1:200) mapped to the first processor, XVECT(201:400) mapped to the second processor, and so on. If there is only one processor, the entire array is mapped to that processor as a single block of 10000 elements. The block size may be specified explicitly in a **DISTRIBUTE** directive:

```
      REAL YVECT(10000)
!HPF$ DISTRIBUTE YVECT(BLOCK(256))
```

This specifies that groups of exactly 256 elements should be mapped to successive logical processors. The number of logical processors must be at least 40 for this directive to be satisfied.

The following example uses a cyclic distribution format:

```
      REAL ZVECT(52)
!HPF$ DISTRIBUTE ZVECT(CYCLIC)
```

If there are 4 logical processors, the first processor will contain ZVECT(1:49:4), the second processor will contain ZVECT(2:50:4), the third processor will contain ZVECT(3:51:4), and the fourth processor will contain ZVECT(4:52:4). Therefore, successive array elements are assigned to successive processors in a round robin form.

For a multi-dimensional array, the distribution is specified independently for each dimension of the array:

```
      REAL XXVECT(8,8), YYVECT(19,19)
!HPF$ DISTRIBUTE XXVECT(BLOCK,BLOCK)
!HPF$ DISTRIBUTE YYVECT(CYCLIC,*)
```

The XXVECT array will be partitioned into contiguous rectangular patches, which will be distributed onto a two-dimensional arrangement of the logical processors. The YYVECT array will have its row distributed cyclically over one-dimensional arrangement of the processors. The "*" specifies that YYVECT is not to be distributed along its second axis. Therefore, an entire row is to be distributed as one object.

The **REDISTRIBUTE** directive is used to change the arrangement of the distributed data on the same or another logical processor template. For example:

```
!HPF$ DIMENSION(8,8),DYNAMIC :: XXVECT
!HPF$ PROCESSOR P(NUMBER_OF_PROCESSORS())
!HPF$ PROCESSOR Q(8,NUMBER_OF_PROCESSORS()/8)
!HPF$ DISTRIBUTE XXVECT(BLOCK,BLOCK) ONTO P
      .
      .
      .
!HPF$ REDISTRIBUTE YYVECT(CYCLIC,*) ONTO Q
```

The array XXVECT is declared DYNAMIC so that redistribution can be applied to it. This is a requirement in the HPF language. In this example the redistribution takes place across two different logical architectures of the same size. Note that the **REDISTRIBUTE** directive is considered an executable statement and may appear among the executable statements. However, the **DISTRIBUTE** directive can be only used in the specification part of the program.

Following is a complete example of an HPF program calling a ScaLAPACK routine. Statements S1 through S6 declare the arrays and constants in the program. S7 declares the processor template as a 1×3 array. S8 and S9 will then distribute A, B, and X onto the processor template in the form of (cyclic(64), cyclic(64)). Statement S21 is the call to the ScaLAPACK routine. To run this program, the programmer must first convert the program to a message passing form using an HPF compiler. In the message passing program, the matrix objects are represented using structures which contain information about the distribution as specified in declaration part of the program (see Figure 4 on Page 24). Once translated to the message passing form, the local versions of the ScaLAPACK routine and the communication library are linked in to generate an executable.

```
S1:        program simplegesv
S2:        use HPF_LAPACK
S3:        integer, parameter :: N=500, NRHS=20, NB=64, NBRHS=64, P=1, Q=3
S4:        integer, parameter :: DP=kind(0.0D0)
S5:        integer :: IPIV(N)
S6:        real(DP) :: A(N, N), X(N, NRHS), B(N, NRHS)
S7:  !HPF$ PROCESSORS PROC(P,Q)
S8:  !HPF$ DISTRIBUTE A(cyclic(NB), cyclic(NB)) ONTO PROC
S9:  !HPF$ DISTRIBUTE (cyclic(NB), cyclic(NBRHS)) ONTO PROC :: B, X
S10:
S11: !
S12: !     Randomly generate the coefficient matrix A and the solution
S13: !     matrix X.  Set the right hand side matrix B such that B = A * X.
S14: !
S15:       call random_number(A)
S16:       call random_number(X)
S17:       B = matmul(A, X)
S18: !
S19: !     Solve the linear system; the computed solution overwrites B
S20: !
S21:       call la_gesv(A, B, IPIV)
S22: !
S23: !     As a simple test, print the largest difference (in absolute value)
S24: !     between the computed solution (B) and the generated solution (X).
S25: !
S26:       print*,'MAX( ABS(X^ - X) ) = ',maxval( abs(B - X) )
S27: !
S28: !     Shutdown the ScaLAPACK system, I'm done
```

```
S29: !
S30:         call SLhpf_exit()
S31:
S32:         stop
S33:         end
```

The following is an example of a call to a ScaLAPACK routine from a message passing program. The full program is much lengthier and many sections are omitted for brevity. Statements on lines S7 and S8 call the routines to initialize the ScaLAPACK system and the processor grid. The matrix descriptors are setup on lines S13 to S16 using calls to DESCINIT. This routine assembles all the distribution information about an array or matrix in a descriptor to be passed to the ScaLAPACK routine on line S26. Statement on line S20 calls MATINIT to distribute the matrices onto the processor grid. A call to the ScaLAPACK routine is then issued on line S26 followed by calls to BLACS routines to free structures and exit the program one lines S34 and S39.

```
S1:          PROGRAM EXAMPLE1
S2:   *
S3:   *      Example Program solving Ax=b via ScaLAPACK routine PDGESV
S4:   *
S5:   *      INITIALIZE THE PROCESS GRID
S6:   *
S7:   *      CALL SL_INIT( ICTXT, NPROW, NPCOL )
S8:          CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
S9:   *
S10:  *      DISTRIBUTE THE MATRIX ON THE PROCESS GRID
S11:  *      Initialize the array descriptors for the matrices A and B
S12:  *
S13:         CALL DESCINIT( DESCA, M, N, MB, NB, RSRC, CSRC, ICTXT, MXLLDA,
S14:        $               INFO )
S15:         CALL DESCINIT( DESCB, N, NRHS, NB, NBRHS, RSRC, CSRC, ICTXT,
S16:        $               MXLLDB, INFO )
S17:  *
S18:  *      Generate matrices A and B and distribute to the process grid
S19:  *
S20:         CALL MATINIT( A, DESCA, B, DESCB )
S21:
S22:  *
S23:  *      CALL THE SCALAPACK ROUTINE
S24:  *      Solve the linear system A * X = B
S25:  *
S26:         CALL PDGESV( N, NRHS, A, IA, JA, DESCA, IPIV, B, IB, JB, DESCB,
S27:        $             INFO )
```

```
S28: *
S29:
S30: *
S31: *      RELEASE THE PROCESS GRID
S32: *      Free the BLACS context
S33: *
S34:        CALL BLACS_GRIDEXIT( ICTXT )
S36: *
S37: *      Exit the BLACS
S38: *
S39:        CALL BLACS_EXIT( 0 )
S40:        STOP
S41:        END
```

Now we will look at an example library call in CMSSL to multiply a matrix by a vector:

```
CMF$LAYOUT A(:SERIAL,:SERIAL),x(:SERIAL),y(:SERIAL)
DIMENSION A(81,81), x(81), y(81)
   .
   .
   .
CALL GEN_MATRIX_VECT_MULT(y,A,x,IER)
```

The first line is analogous to the **DISTRIBUTE** directive of HPF. The **:SERIAL** distribution in the Connection Machine Fortran has the same semantics as the **BLOCK** distribution in HPF. CMSSL uses a scheme similar to ScaLAPACK to pass the information about the distribution of the arrays to the library routines. The translation of the array reference to actual structures that contain various information about the array (including its distribution) is completely transparent to the programmer.

Since the data is assumed not to move in HPF, the compiler can, through analysis, determine the data redistribution required to map the data from the source logical architecture to the destination logical architecture. This is basically because in HPF data is static with respect to the logical processors. In programming languages such as our virtual architecture language this is not the case. Following the data distribution, the data is allowed to move around. By the time the program reaches the end of a parallel phase, the distribution of the data is not that of the beginning. Since the final distribution of data is determined by the algorithm, this information cannot be obtained by the compiler and a technique similar to that of Kalns and Ni [46] fails and a different library set up must be considered. Another shortcoming of the current work in the design of data redistribution libraries is that the dimensionality and the sizes of the source and target logical processors are assumed to be the same.

Most compilers for data parallel languages perform many-to-one mapping to support portability of parallel programs to different size systems. The code produced by these compilers runs at a granularity equal to that defined in the original program [46]. Therefore, if the virtual architecture is very large and the distributed data is fine grain, the communication overhead of the program may become intolerable in small size systems. Likewise, this argument holds for data redistribution between the phases when many small messages must be exchanged between the processors.

Chapter 3

Parallel Library Design

A sequential program may consist of several calls to computation intensive numerical routines, possibly nested within various programming constructs such as conditionals or iterative constructs. The problem instance size of these calls is seldom known statically. Often data used or computed by one library call is also used as parameter in a subsequent call. Many sequential libraries exist which are utilized to perform these intensive numerical computations. These libraries have a well defined caller interface and have been optimized for good performance. The objective of these library routines is to encourage reusability of software and robustness of user programs. As application programs require larger problem instances to be solved by these library routines, the delivered performance becomes unsatisfactory. Using a standard language and obeying the interoperability issues, a sequential library can be easily ported to another system by a one-time optimized compilation. To better utilize the target system, the library routines may have to be tuned to take advantage of the new system (i.e. architecture, number of registers, cache size, *etc.*).

With the advent of multicomputers, the need to perform these intensive numerical computations in parallel was deemed necessary. Multicomputers operate on distributed data using multiple processors which is unlike the clear uniprocessor stack model of sequential libraries. If the sequential call interface is to be maintained for parallel libraries, there is a gap between the call parameters and the distributed data which must be filled appropriately by the compile time and the run time systems. Data from one processor is only available to other processors through communication which is normally a costly operation. Expensive communication operations make performance of the library routines sensitive to the location of the computation and the data on the multicomputer.

Portability of parallel libraries is also quite different from sequential libraries. A parallel library is considered portable if it can run on the same system using different partitions and configurations, or on another physical system. In other words, it must be able to run on a system with different sizes and topologies without the need for recompilation. Porting to another system must only require a one-time parameter tuning and compilation of the library. Another requirement is that the library must deliver its best performance on different size systems. If ported to another system, it must utilize the physical system based on its hardware characteristics, such as the communication latency and the processor speed.

Existing parallel libraries require the programmer to specify the computation on the distributed data. Management of the distributed data and redistribution between two parallel library calls are the jobs of the programmer. Therefore, the call interface of these libraries is not similar to the sequential libraries. The performance of the library routines is also dependent on the data distribution done by the programmer. Achieving good performance requires several trial and error runs. Furthermore, If sequential programs are to be assimilated into these programming environments, they must be completely rewritten. However, this is not the case with parallel libraries with sequential call interface. With parallel libraries which possess call interface similar to sequential libraries, programmers who have only knowledge of sequential programming and would like to run their intensive computation on a parallel system may develop their application with solely knowledge of the call interface of the parallel library routine as if they are calling a sequential library routine. The complexity of management of the distributed data, efficient execution on the parallel system, and the data redistribution between calls are handled by the compile time and the run time systems.

Virtual architecture algorithms have several unique properties which offer solutions to currently existing problems in parallel library design. The size of the virtual architecture is generally a function of the problem instance size. This property can help the parallel library routine to derive the size of the virtual architecture at runtime when the problem instance size is known. Virtual architecture algorithms can be described in parameterized block form which supports runtime grain size adjustment. The virtual topology of the algorithm is generally representative of the communication pattern of the algorithm. This information can assist the runtime system to perform mapping of the virtual processors to the physical processors and reduce the communication overhead of the program.

The objective of this chapter is to offer a design of the parallel libraries based on the virtual architecture programming model which adheres to several unique properties. The call interface of these library routines resembles those of sequential libraries. These library routines deliver higher performance than the existing parallel libraries, manage distributed data transparently, and provide automatic data redistribution across parallel phases. Furthermore, the objective of these parallel libraries is to run efficiently on partitionable and reconfigurable systems without recompilation. If ported to another system, upon providing the physical system attributes to the library system and a one-time compilation, the library routines must fully harness the target system to provide the lowest possible execution time to the user programs.

The features of the parallel library design proposed in this thesis consist of the following:

- Sequential call interface.
- Support for virtual architecture parallel algorithms.
- Delayed runtime grain size adjustment to improve performance.
- Delayed runtime mapping to improve performance.
- Centralized mapping library.
- Virtual point-to-point and collective communication library.

- Data distribution encapsulation.
- Transparent data redistribution.
- Easily portable to systems with a low level communication library and a threading library.
- Easily tunable to harness a parallel system based on its communication network characteristics.

These features not only make use of the parallel libraries very simple, they also facilitate creation of new library routines based on virtual architecture algorithms. The library system can be easily extended to handle new topologies and mapping functions.

A library routine writer must take the following steps in the design of a parallel library routine:

- Select highly scalable virtual architecture parallel algorithms for library implementation.
- Identify the distributed data requirements of these algorithms.
- Develop the execution time functions for the runtime grain size adjustment.
- Examine the behavior of these algorithms under different processor mapping functions and select the most viable one for each algorithm.
- Implement the required components of the library routine to perform the grain size adjustment, delayed mapping, automatic data distribution, automatic data accumulation, and the core computation.

To support the above, we have proposed an internal representation of the virtual architecture parallel algorithms. Furthermore, through proper design of the library routines we support *ease of use*, *reusability*, and *portability* of parallel programs which generally classify as important objectives of software engineering. The outcomes of this approach are parallel library routines which can be used by sequential programmers to run their intensive numerical computations in parallel. The components of the design are a transformer which converts sequential programs to an SPMD form, an internal representation of virtual architecture parallel libraries, a mapping library, a virtual communication library, and a redistribution module. Several library routines have been implemented and performance results from selected routines have been presented.

In the rest of this chapter, we describe the infrastructure and the framework for parallel library design of virtual architecture programs. The description of the design is followed by the experimental results from our integrated programming environment. These experimental results confirm our analysis and illustrate that our design outperforms currently existing parallel libraries. It is further shown that the design does not only adhere to our major goal of performance, but also to the reusability, portability, and ease of use criteria.

The library design in this thesis bears several unique properties not observed in other parallel libraries. These are:

- Runtime granularity adjustment to minimize the execution time of the virtual architecture routine. In other libraries, the algorithms perform the computation at the granularity specified by the programmer.

- Delayed mapping to reduce communication overhead of the program. Other libraries do not address the issue of mapping. Mapping is claimed to be a compiler dependent issue.
- Encapsulated data distribution information. In other libraries data distribution must be defined by the programmer.
- Support for automatic data redistribution. In other systems, data redistribution must be explicitly specified by the programmer.
- Easy to use. Our proposed library design is very easy to use since many of the cumbersome details are hidden from the programmer.
- Portable across different platforms. Although other libraries also claim portability by using standard language constructs and communication libraries, our provisioning for performance tuning in portability is unique and has not been used in other parallel libraries.

Figure 7 depicts the series of steps in setting up a parallel library routine. The first step is to identify a suitable virtual architecture algorithm which can be described in block form. The scalability of the algorithm must be examined based on the parameterized grain size of the computation and the mapping of the virtual processors onto the physical system. One can develop an overhead function for the parallel algorithm and investigate its ability to maintain constant high efficiency using a metric of scalability such as the isoefficiency metric. The next step is to analyze the effects of different mapping functions and their impacts on the execution time of the routine. Once a suitable algorithm and mapping function are determined, an execution time function is developed for the algorithm. The execution time is a function of the problem size, grain size, and the physical system size and attributes such as the communication latency components. We first assign parameters to the granularity of the computation and the problem size. Then, we approximate the execution time by considering the bulk part of the computation at each virtual processor (normally a function of the problem size and the grain size) and the communication operations. For the computation part, the number of operations multiplied by the time to perform one floating point operation (a parameter of the underlying machine) results in the total time spent by the virtual processor on the computation. The communication time can be approximated by a startup cost (another parameter of the underlying system) for each **send** operation. The **receive** operations are difficult to model in the execution time as their blocking will affect the total execution time. We assume that the messages are available when a receive is executed. The execution time function contains parameters for the time to perform a floating point operation and the startup cost of communication. These parameters are globally defined in our library. The use of these parameters in symbolic form by all library routines for grain size adjustment reduces the portability effort. These parameters can be changed before compilation of the parallel library on the new system. This is clearly shown at the bottom of Figure 7 right before the compilation phase.

The execution time function mainly captures the effects of granularity on the execution time of the library routine. Mapping has not been considered in this model. This factor has been considered separately and its inclusion in the grain size adjustment phase will most likely not affect the decision

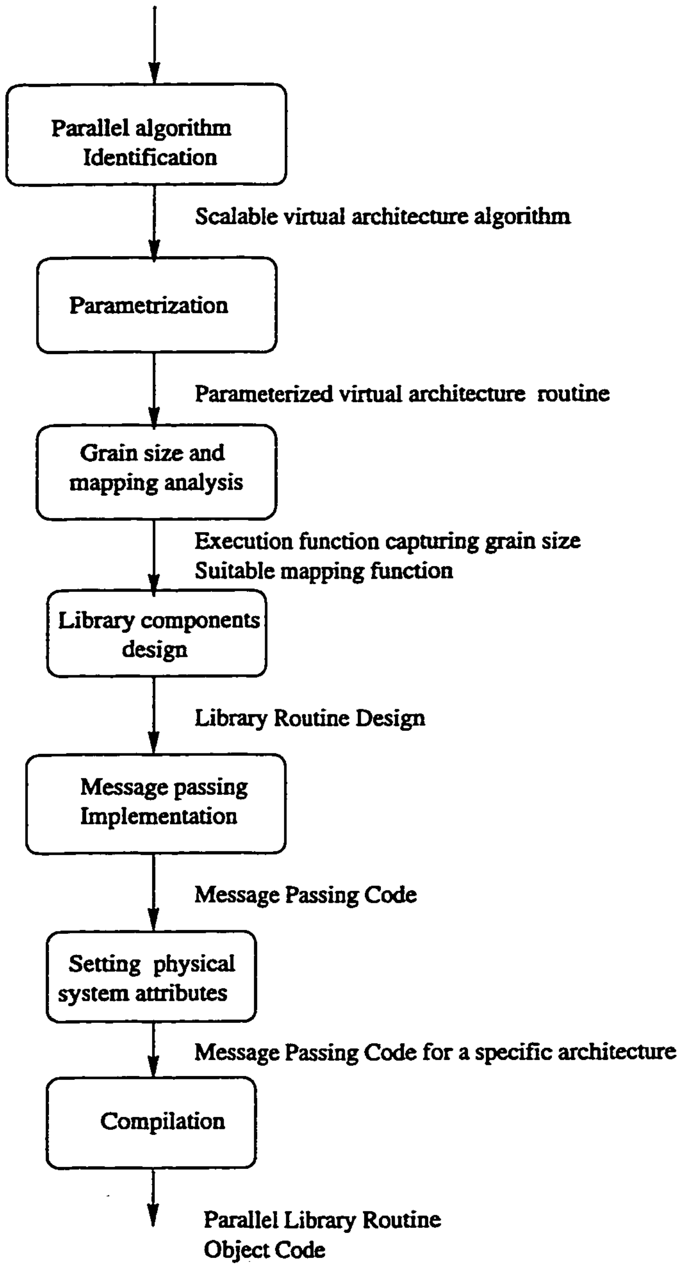

Parallel algorithm Identification
Scalable virtual architecture algorithm
Parametrization
Parameterized virtual architecture routine
Grain size and mapping analysis
Execution function capturing grain size
Suitable mapping function
Library components design
Library Routine Design
Message passing Implementation
Message Passing Code
Setting physical system attributes
Message Passing Code for a specific architecture
Compilation
Parallel Library Routine
Object Code

Figure 7: Steps of library routine creation

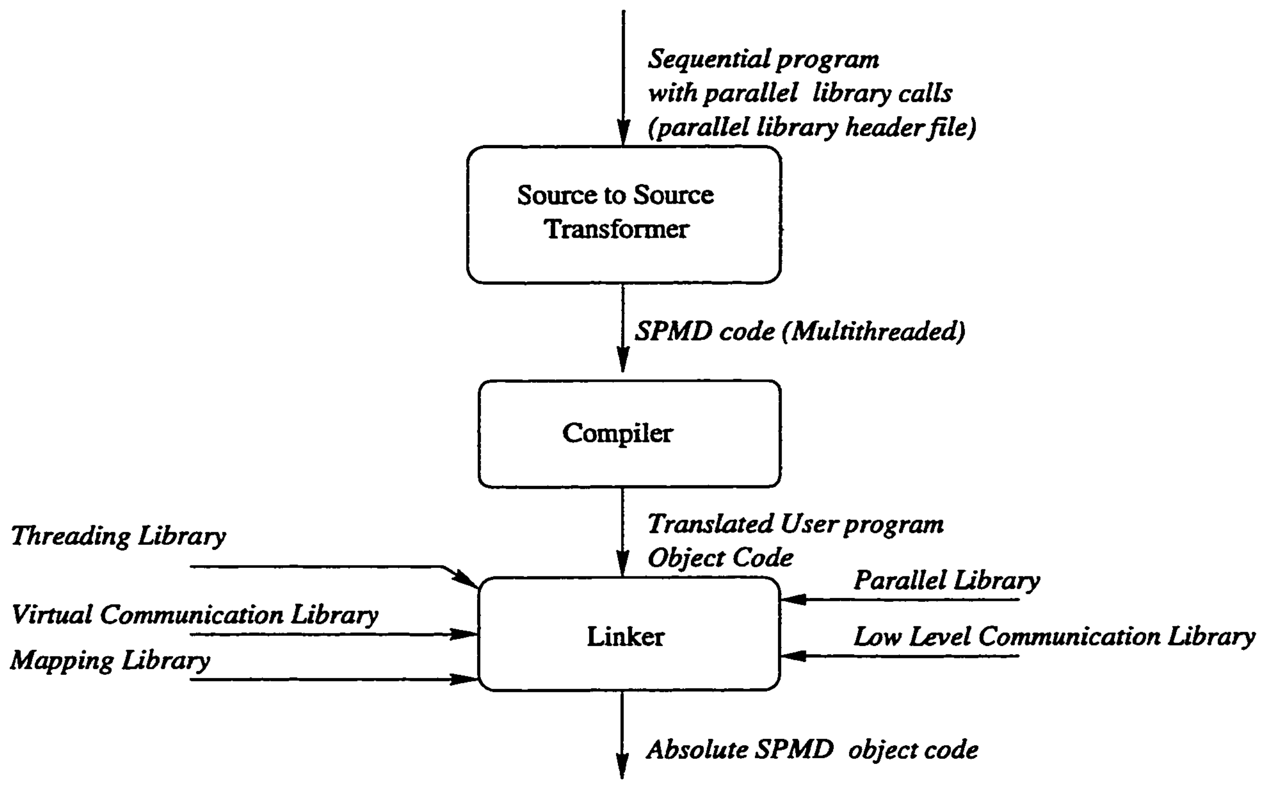

Sequential program
with parallel library calls
(parallel library header file)
Source to Source Transformer
SPMD code (Multithreaded)
Compiler
Translated User program
Object Code
Threading Library
Virtual Communication Library
Mapping Library
Linker
Parallel Library
Low Level Communication Library
Absolute SPMD object code

Figure 8: Compilation process of a sequential program with library calls

making. The impact of granularity on the execution time is much more severe than the impact of mapping. The message passing implementation of the library routine consists of the development of a set of defined routines to initialize the virtual architecture algorithm information in a structure called the *library handle*, perform data distribution, data accumulation, grain size adjustment, mapping, and parameterized computation.

Figure 8 shows the steps in using a library routine. Users' sequential programs are translated to an SPMD form by a source to source transformer. The SPMD form contains calls to the components of the library as well as calls to the threading library routines. The components of the library routine use the virtual and physical communication routines and the mapping library routines. These references are resolved in the link step and an executable is produced. When the absolute SPMD object code is loaded onto the nodes of a multicomputer, the library calls will execute in parallel on the target system. The other portions of the user program will be either executed only on the host physical processor (such as I/O statements) or by all the processors (such as iterative and conditional constructs). In addition to the communication requirements of the virtual architecture parallel library routines, the main processes of the physical processors may need to obtain updated values from the host processor. These requirements will be discussed in Section 3.2.

In the following subsections the internal representation of the distributed data and the virtual architecture programs are presented in detail.

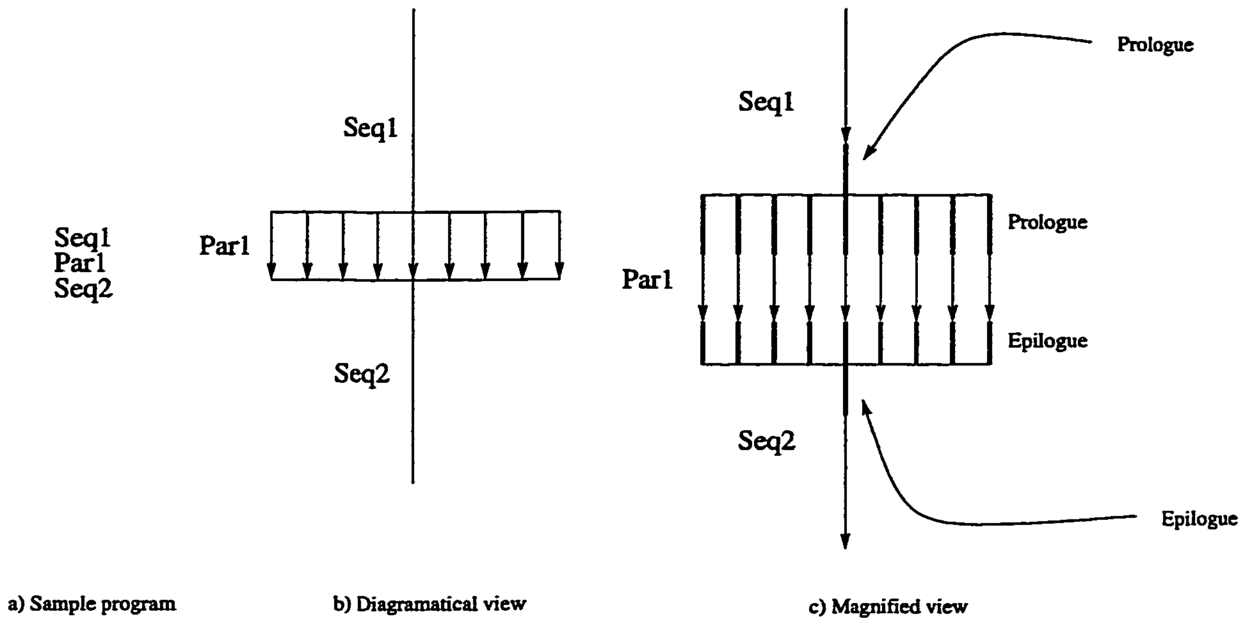


Figure 9: Prologue and epilogue in a parallel call

3.1 Integrated Components of the Library

A virtual architecture algorithm consists of a data distribution function, core computation and data accumulation. For a parameterized parallel library implementation of the virtual architecture algorithms, these components must be defined in the library. The mapping of the data as well as the processors must be encapsulated in the library.

In this section, we consider a call from a sequential program to a parallel routine in isolation. On a parallel system, the sequential phases can only take advantage of a single processor, but the parallel phases may harness the whole system. Consider a program that consists of sequential phases *seq1* and *seq2* and a parallel phase *par1*, executed in the order specified by the program in Figure 9(a) and the parallel execution demonstrated in Figure 9(b). The parallel phase is shown in more detail in Figure 9(c).

In most systems the sequential portion of the program is executed by a single processor which is connected to the I/O subsystem and the file system. Consider the program at the end of the sequential phase *seq1*. Upon termination of this sequential phase, the caller must prepare to make transition to the parallel computation. This part of the program is known as the *call sequence* or *prologue*. When the parallel computation finishes, the processors must prepare to make transition to the sequential computation. This part of the code is known as the *return sequence* or *epilogue*. Prologue and epilogue each has an associated component in the sequential phase and the parallel phase. We now examine each of these components.

Sequential component This part of the call/return sequence from the sequential code to the parallel code, and vice versa, deals mainly with the global data distribution and accumulation.

- *Prologue* - Prior to transition to a parallel phase, the sequential code must determine the contraction factor, identify the mapping of logical to physical processors, decompose the data, and send the data to the virtual processors. Once the contraction factor is determined, the storage of data must be changed to match the block row major form for the decided granularity. This phase is denoted by **Data Decomposition** in Figure 10(a). The data distribution phase, denoted by **Data Distribution** in the figure, sends the partitions to the virtual processors. Data distribution is done by a number of send operations from the host virtual processor to the nodes. This may be optimized by sending larger partitions to two nodes which in turn break up the partitions and send to other nodes (in form of a tree). This will introduce additional complexity in the data distribution section and will require more space at the nodes, but the time to distribute the data will be logarithmic instead of the naive linear approach. The implementation of the **Initial Data Receipt** in Figure 10(b) will be impacted by this decision as this is the module that must receive the intermediate partitions.

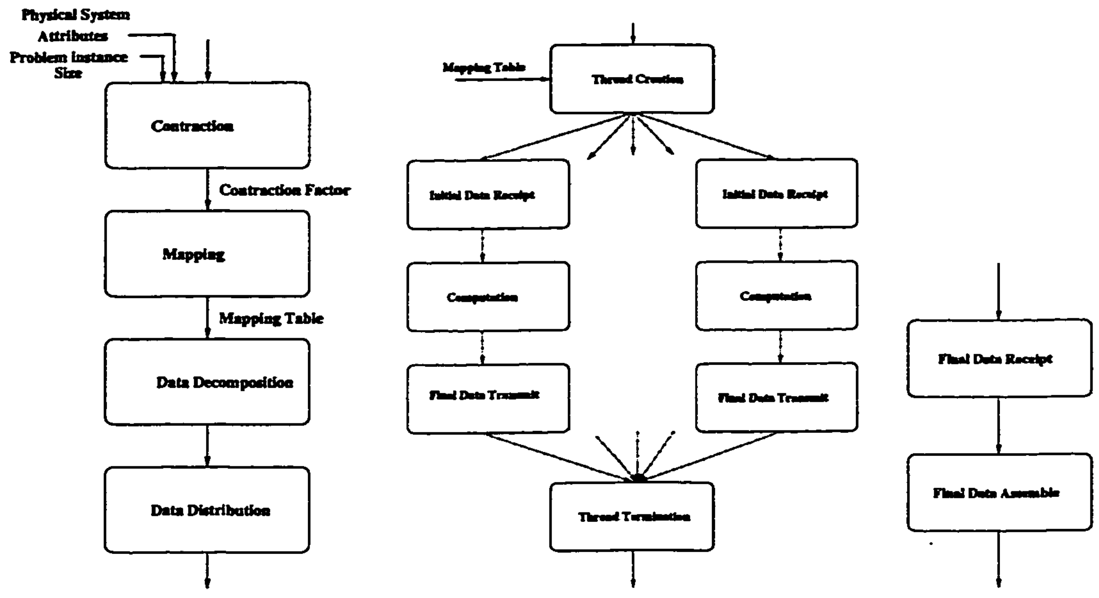
- *Epilogue* - Upon transition from a parallel phase to the sequential phase, the sequential code must receive the resulting partitions of data and assemble them in the global data. A **send** operation by the virtual processor holding the data partition and a matching **receive** executed by the host is the approach used in our libraries. This may create contention at the host virtual processor (and therefore the host physical processor). Other possibilities are to gather the data in form of a tree similar to the data distribution but in reverse. The trade-off is between the required space at each virtual processor, the number of startup costs incurred, and the delay due to contention a the host processor. The **Final Data Transmit** modules of Figure 10(b) will be affected by this design decision as these modules must receive intermediate partitions and pass them to other virtual processors. Once the data is received, it is stored in block row major form. This storage must be changed to single element row major form to adhere to the semantics of the user program.

Parallel component This part of transition from the sequential code to the parallel code and vice versa deals mainly with virtual processor spawning, initial local receipt and final local data send.

- *Prologue* - Upon transition from the sequential code, all physical processors must compute the mapping table which identifies the virtual processors of the current phase assigned to them. All physical processors must spawn one thread for each one of these virtual processors. The threads will then receive their initial data.
- *Epilogue* - Prior to transition to the sequential code, all virtual processors must send the final data to the processor designated as the host[1]. All threads will then terminate.

Our libraries consist of the four high level components described above and the parallel computation. These are shown in Figure 10. Two important steps in the call/return sequence that are not pertinent

[1] We refer to *host* as the processor which has I/O capability. The term node is used for all other processors. We use these terms similarly for the virtual architecture. This will not cause any confusion since host virtual processor is always mapped to the host physical process of the partition.



(a) Sequential prologue (b) Parallel prologue and epilogue (c) Sequential epilogue

Figure 10: The components of a parallel library call

to sequential libraries and play a big role in the performance of the library routine are the contraction and the mapping. The contraction step determines the granularity at which the parallel library routine runs, whereas the mapping step determines the communication pattern of the reduced virtual architecture in the physical system. The parallel computation along with the data distribution and accumulation are done once the contraction factor and the mapping are fixed.

Using our design, a user program is converted to an SPMD form. The SPMD form is then compiled and linked with the necessary components to obtain an executable image. The executable image is then downloaded onto each physical processor of the allocated partition. Each physical processor will run one process which is basically the same executable image. This can be thought of as SPMD execution at the physical processor level. Initially, the main starts execution on each physical processor. When a parallel library call is entered, the host processor, which holds the global data and the size of the problem instance, broadcasts the size to all the physical processors. The physical processors will then simultaneously derive the contraction factor, the virtual processor architecture size, and the mapping of the virtual processors onto the physical processors. Until this stage, all physical processors run a single flow of execution and prepare for the parallel execution. When the mapping table is determined, every physical processor creates one thread for each virtual processor assigned to it for the upcoming phase. The threads, collectively over all the physical processors, constitute the parallel library call. All threads execute the code for the virtual architecture parallel algorithm. This is, in a sense, another level of SPMD execution used in our system. The host virtual processor sends the initial data to all other virtual processors. The virtual processors cooperatively

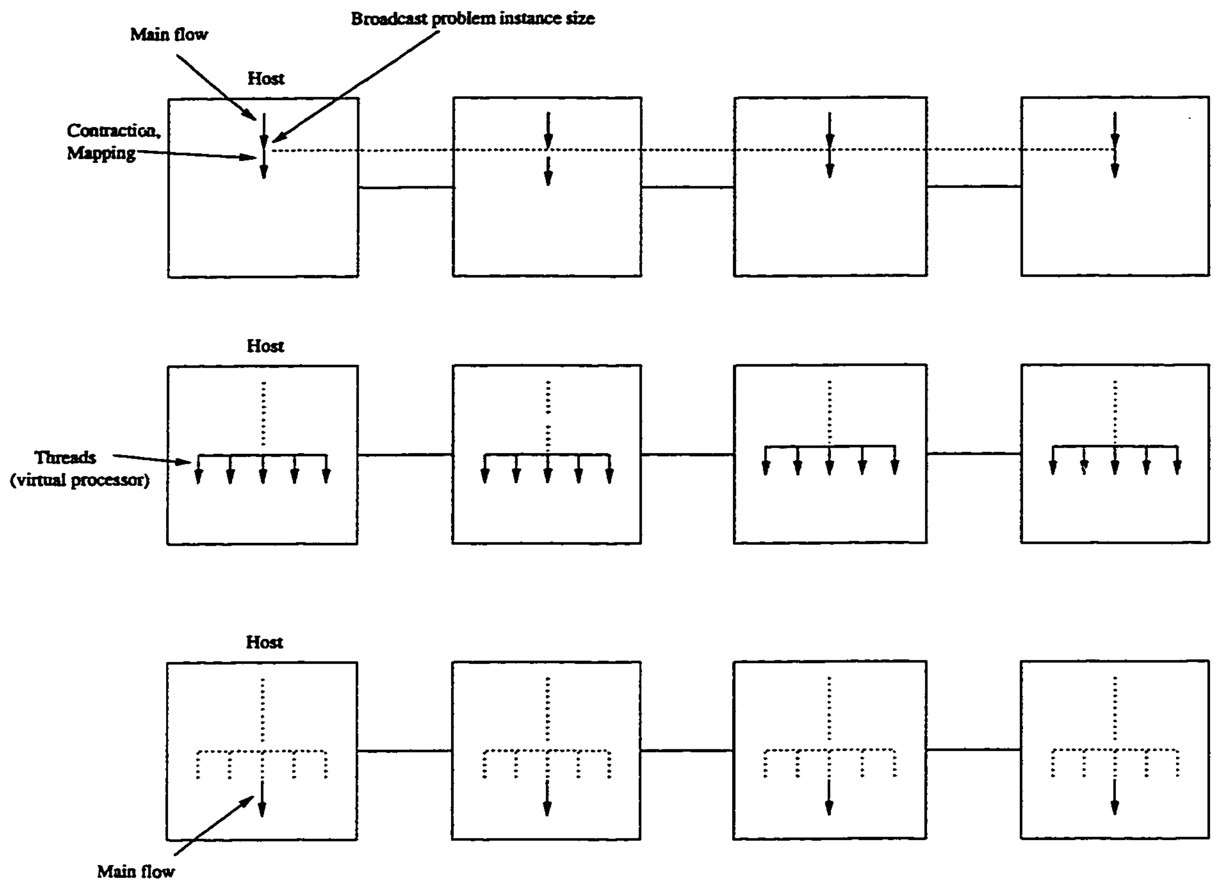


Figure 11: Flow of control in a parallel library call

complete the parallel computation. Within each physical processor, the threads join and execute the main thread in the SPMD program. Each physical processor performs local synchronization of its threads. There is no global synchronization among the threads at the end of the parallel phase. Any physical processor which completes its execution of the assigned parallel computation may resume the main flow of execution. Figure 11 shows four processors in form of a line in three stages of the execution. The top one shows the state of each processor upon entering a parallel library call. The host physical processor broadcasts the problem instance size to all other physical processors. All physical processors will then compute the contraction factor and the mapping table simultaneously. The middle stage shows the state of the processors when one thread for each virtual processor is created. Each physical processor creates the threads assigned to it by the mapping. The physical processors each hold a complete copy of the mapping table. The threads communicate with one another using virtual communication. When the threads of a physical processor are done, they are merged into a single flow of execution shown in the bottom of Figure 11. The figure is showing a single parallel library call. However, the main process running on each physical processor may survive several parallel phases.

3.1.1 Virtual Processor Implementation

If the size of the virtual architecture is larger than the physical system size, at least one physical processor will be assigned two or more virtual processors. This normally happens when the grain size of the computation cannot be adjusted to exactly match the physical system in size. In such a case, there are two choices of grain sizes. One of these values will cause some processors to have no work assigned to them and the other assigns two or more virtual processors to at least one physical processor. This problem is known as load imbalance. The library routines can be enhanced to compare the impacts of the load imbalance for the two possible grain sizes. If the smaller grain size results in lower execution time, more than one virtual processor will be mapped to a physical processor. To keep the overhead of context switching low in each physical processor we decided to use multi-threading within the physical processors. In other words, virtual processors are emulated using *threads*. Threads can be scheduled at the user level with much lower cost than processes. The scheduling of threads can also be implemented in a manner suitable to the application. For our parallel libraries, context switching takes place only when a virtual processor blocks on a **receive**. If more than one thread can run within a physical processor, any one of the two are equally a good candidate for running.

The library designer may rely on calls to a threading library to create, join, and destroy threads, or obtain the thread identifier. The thread identifier can be stored in the mapping table upon creation which subsequently provides the one-to-one correspondence between the threads and the virtual processors. It is likely that two calls to the **thr_create** routine, on two processors, return the same thread identifier. Since in our design of the library each physical processor needs to know the correspondence of its own threads and the virtual processors, no conflict will arise. Our use of threads has been motivated by the low overhead in creation and scheduling offered by threads as opposed to processes. Thread creation is far less expensive than process creation. User level threads can be created by basically a function call which sets up the structures and the stack for the thread. These threads are managed at the user level and are completely transparent to the kernel. The lower overhead in user level threads is at the cost of thread scheduling which must be done by the programmer. User level threads can easily deadlock if the scheduling mechanism used by the programmer has potential for deadlock or is not implemented correctly. Each thread is allocated a stack space from the heap. The code is shared by all threads created within a processor. Therefore, the thread structure must contain fields for its identifier, the program counter, and the stack pointer. Furthermore, threads have access to their creator's address space. This feature suits our need of shared access to the handles within a physical processor by all threads. The type of threads we use for our design are unbound threads[2] which basically do not do scheduling. The scheduling is done by the programmer. The consequence of this is that unbound threads cannot keep runtime statistics. In this case, runtime statistics are maintained in the kernel at the process level. We are considering only one process on each physical processor, but in cases where a physical processor of the parallel system may be used by several programs simultaneously, several independent processes may be created on one physical processor. Each process may, in turn, create threads for a library

[2]This terminology is used in some literature, but it does not have a standard definition.

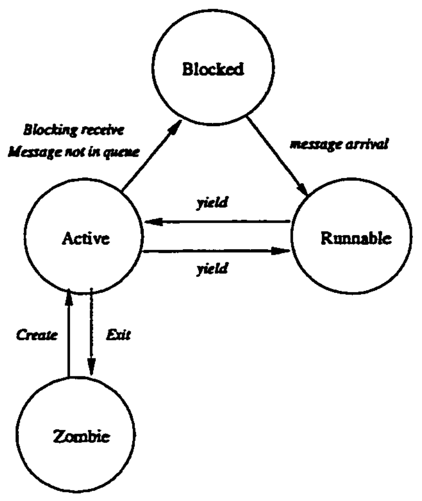


Figure 12: Thread state transition diagram

call by the user program[3]. If a thread is given control, it will continue virtually forever until it relinquishes the processor. In a threading library, the programmer has the flexibility of using thread suspension, preemption, or yielding. Suspension refers to the case where the thread stops itself, preemption refers to when a thread takes an action that makes a higher priority thread run, and yielding basically means a thread voluntarily gives up the processor to another thread. In these cases, the former thread will be put in a blocked state waiting for another thread to give it a chance by executing another yield or other possible calls that allow it to continue. A simplified view of the thread state transition used in our design is shown in Figure 12. A thread is *active* when it is initially created.

Message passing programs can cause deadlock if not programmed properly. Although this is a general problem in message passing programs, the underlying design may introduce deadlock. Assuming the message passing program is deadlock-free, we examine our design and its potential for deadlock. When one-to-one mapping is used, the blocking **receives** yield if the message is not in the queue. In this case control comes back to the same thread until the message arrives. When many-to-one mapping is used, the thread yields control to another thread if its message is not in the queue. Therefore, all threads within a physical processor get a chance to execute.

When multiple threads call a function that manipulates a common object (this may well happen in our design when we have many-to-one mapping), problems could arise. It is important to make sure that the threads do not have adverse effect on each other. This can be done through use of locks or use of reentrant code. Locks are inefficient and have been outruled due to the fact that we do not expect the threads to write to common area in the global memory. Each thread has its own data partitions. When the handles are being initialized, there is only the main thread (or process) running and others have not been created. If the function is reentrant, it means that it can be called by several threads and deterministic behavior will be observed. Functions can be made reentrant

[3]This is a special form of timesharing on parallel systems, where a physical processor can be simultaneously used by different user programs. We are not aware of such commercial systems, but they may exist or future systems may well be timesharing in this form.

```
int thr_create(void *stack_base, size_t stack_size, void *(*start_routine) (void *),
               void *arg, long flags, thread_t *new_thread);
```
Add a new thread of execution to the current process

```
void thr_exit(void *status);
```
Terminate a thread

```
void thr_join(thread_t wait_for, thread_t *departed, void * * status);
```
Wait for a thread to terminate

```
thread_t thr_self(void);
```
Return the thread ID for the calling thread

```
void thr_yield(void);
```
Yield the current thread

Figure 13: Thread library calls

by ensuring that, once called by multiple threads, they do not produce inconsistent results because of race conditions. We have made our functions reentrant by allowing each thread to have its own data partition in the handle to work with. A thread must not, and neither does it need to, access the data partitions of other threads. Violation of this rule simply introduces a bug in the parallel library routine. Our other option would have been to make the functions reentrant by passing these partitions as parameter to the threads, but this scheme would not support our redistribution module which will be presented in the next chapter.

There are a number of thread libraries available [55]. We are only concerned with the interface of the thread libraries. The set of important thread library calls in our design are listed in Figure 13. The syntax is borrowed from Solaris threads library [55], however our design does not rely on specific implementation of a threads library. The first call interface is that of **thr_create** which creates a thread using a function and its arguments and returns the thread identifier to the user program. The stack space of a thread is allocated from the dynamic memory. A stack with default size is allocated by the **thr_create** routine if **stack_base** and **stack_size** parameters are NULL and 0, respectively. If the default stack size does not suit the user program, the user must allocate a stack of desired size and pass its pointer and size using these parameters. The thread's stack space is allocated from the heap. The function that is the starting point of the thread is specified in the argument. The parameters of the function must be placed in a composite structure and passed to the starting routine. These parameters are not in the scope of the thread and may not be global (in the user program). Therefore, they must be passed to the thread function. The flag argument determines whether the thread is joinable by other threads or processes. This is to ensure that upon exit, the thread's status, resources, and exit status are only discarded if they are not needed by another thread. Two allowable flags in Sun Solaris are PTHREAD_CREATE_JOINABLE and

PTHREAD_CREATE_DETACHED. We use the former since the created threads must be collected by the main process. The last argument in **thr_create** is the actual thread identifier. **Thr_exit** terminates a thread and if the thread is joinable the status is kept until it is joined by another thread. **Thr_join** waits for a thread to terminate and returns its identifier and status. **Thr_self** returns the identifier of the calling thread, and **thr_yield** yields control to another thread in the current process.

The source to source transformer translates library calls of the sequential code to SPMD form which performs contraction, determines the mapping, scans the mapping table, and creates threads on each physical processor. The thread identifiers along with the local identifier (the order of creation) are stored in the mapping table for local data reference. Each physical processor uses an incrementing counter (starting from zero) to assign a unique local identifier to each thread. The local identifier and the contraction factor are used by each thread to access its own data in the contiguous memory which is allocated for all the threads in a physical processor. Each physical processor has information about the mapping of all the virtual processors to the physical processors. However, the thread identifiers of each physical processor are not available to any other physical processors. This restriction does not impose any limitations in our design since a physical processor does not need to have access to the thread identifiers of threads on other physical processors.

3.1.2 Virtual Communication Library

In our library design, virtual processors are mapped to the physical processors. Virtual processors communicate with one another using their virtual processor identifiers. For a message to be sent to the right destination and be received from the right source, the mapping table must be consulted. This table provides the physical location of each virtual processor. This translation is handled by our virtual communication library.

In the description of the design of library components, we assume that the system provides a communication library such as PICL [28]. We have used PICL in experimenting the design of our libraries. Although MPI is becoming the standard for message passing multicomputers, we believe that its abstraction of the communication library prevents many compiler and runtime systems to perform optimizations such as mapping of the computation onto the processors. Our design can be easily described in the context of MPI with some penalty in the execution time due to abstraction of virtual topologies. Without loss of generality, we have decided to use necessary routines from the PICL communication library in describing our design. Library designers may use conditional compilation directives to describe the library routines using all well known communication libraries. This will diversify the use of the libraries since for a target system the library can be easily compiled for the resident communication library using macro definitions. Figure 14 shows the routines from PICL which are used in our design. **Who** is frequently used by the SPMD program to obtain its processor number. In our design, this is often used to distinguish the host processor from the node processors. The calls to **send** and **recv** are used for interprocessor communication. The **send** is nonblocking but the **recv** is a blocking primitive. These routines have been encapsulated in the virtual communication library. The library routine makes calls to the virtual communication routines

void who (int *numProc, int *me, int *host);
Return the number of processors, the node ID number and the physical host ID number

void send (char * msg, int size, int tag, int dest);
Send a message to a physical processor (non-blocking)

void receive (char * msg, int size, int tag);
Receive a message from a processor (blocking)

void recvinfo (int *size, int *tag, int *source);
Return information about the most recently received message

int probe (int tag);
Check whether a message with tag has arrived (non-blocking)

void bcast (char * msg, int size, int tag, int root);
Broadcast a message from a root physical processor to all other physical processors

Figure 14: Communication library calls

which, in turn, call the low level communication routines. **Probe** is used by the virtual **receive** to check whether a message with a given tag is in the queue. If not, the virtual communication routine yields the physical processor to another virtual processor. **Recvinfo** is used to obtain the information on the recently received message. The virtual receive routine obtains information on the received message using this call. **Bcast** is used by the library system to broadcast the problem instance size to all the physical processors. This is the first piece of information that is communicated between the host and the node programs upon calling a parallel library routine. Using the problem instance size, all physical processors can compute fields of the handle simultaneously.

Tags are used in the communication routines to distinguish a message type from another at the destination. In our design of the libraries, a unique tag is assigned to each parameter of the library call. These tags are defined using macro definitions in the library modules. Use of different tags causes the resulting program to be more robust and easier to trace since chances of mixing messages will be lowered. We use the tag field to attach additional information to the message, such as the source of the message (virtual source). PICL library also supports **receive** with a special message tag called **ANY**. A **recv** with **ANY** message tag will pick up the first message in the queue. This call will only block if the message queue is empty. This is usually used in conjunction with **recvinfo** to drop the synchronization restriction of receiving a number of messages in a specific order. The call to **recvinfo** provides the information to take further action on the message. We do not allow the use of **ANY** message tag in the virtual communication routines. The ANY message tag is a PICL specific implementation and is normally not supported in other communication libraries. Use of this tag type results in programs that are hard to debug.

In some virtual architecture algorithms some decision making may be based on the virtual processor identifiers. The virtual version of PICL's **who** can be used in the implementation of these algorithms in the library. The **vwho** routine returns the identifier of the currently executing virtual processor, the total number of virtual processors, and the host virtual processor identifier. **Vwho** obtains the identifier of the currently executing thread, searches the mapping table for that thread, and returns the virtual processor identifier (which is the index of the mapping table). The total number of virtual processors can be obtained from the current handle, provided as parameter to the virtual communication routines. An additional value returned by **vwho** is the *rank* of the virtual processor. When more than one virtual processor are mapped to a physical processor, each virtual processor is assigned a unique integer starting from zero called the rank of the virtual processor. This information is maintained in the mapping table and is required by the components of the library when the distributed data of a virtual processor is accessed in a contiguous space allocated for all the local virtual processors. Algorithm 1 describes the step by step actions of **vwho**.

Algorithm 1 Virtual who

```
Require: Addresses of: numvp, myvp, myrank, hostvp
 1: Get my thread identifier (thr_self)
 2: Search and find my entry in the mapping table
 3: Retrieve the virtual processor rank and identifier
 4: Assign values to myrank and myvp, respectively
 5: Get number of virtual processors from the handle and assign to numvp
 6: Assign value 0 to hostvp
Ensure: Parameter information is obtained
```

The virtual send operation must use the mapping table to find the physical location of the destination virtual processor. It must then attach the source and destination virtual processor identifiers to the message before launching the message. Once a message arrives at a processor, the virtual source and destination are stripped from the message and it is delivered to the right virtual processor. In reality, we attach the source and destination virtual processor identifiers to the virtual tag[4] to build a physical tag. The virtual receive also uses the same convention in acquiring a message. Algorithms 2 and 3 show the processing performed by each of the virtual

Algorithm 2 Virtual send

```
Require: User data buffer, size, destination virtual processor, and virtual tag
 1: Lookup the entry for destination virtual processor id in the mapping table
 2: Build a physical tag using the virtual destination, virtual source, and virtual tag
 3: Send the message to the destination physical processor using physical tag
Ensure: Message is sent to the destination
```

communication library routines used in our design. Virtual send looks up the physical location of the virtual processor and sends the data. In order to resolve the destination of the message and provide the virtual source of the message on the receiving end, the virtual send must attach to the tag the virtual processor identifier of the destination and the source. In other words, the

[4]By virtual tag we mean the tag used by the virtual communication call. When the message is launched the tag is a composite or physical tag.

Algorithm 3 Virtual receive

```
Require: Arguments: virtual tag, user data buffer, message size
 1: Query my virtual processor identifier
 2: Construct the physical tag using virtual tag, virtual destination, and virtual source
 3: arrived ← FALSE
 4: repeat
 5:    Probe for message with physical tag
 6:    if message not arrived then
 7:       Yield to other threads (thr_yield)
 8:    else
 9:       arrived ← TRUE
10:    end if
11: until arrived == TRUE
12: Receive the message with virtual tag (Low level receive)
13: Assign user data buffer
Ensure: Proper message is received
```

void vwho(int *numvp, int *me, int *host, LibHandlePtr handle);

Return the number of virtual processors, the virtual node ID number and the virtual host ID number

void vsend(char * msg, int size, int tag, int destvp, LibHandlePtr handle);

Send a message to a virtual processor (non-blocking)

void vreceive(char * msg, int srcvp, int size, int tag, LibHandlePtr handle);

Receive a message from a virtual processor (blocking)

Figure 15: Virtual **who** and the virtual point-to-point communication prototypes

tag for the transmitted data will actually be $(vsrc, vdest, vtag)$, where $vsrc$, $vdest$, and $vtag$ are the virtual source, destination, and tag, respectively. If the number of virtual processors in the current phase is C, and the maximum number of tags (parameters of the library call) is M, one can build a unique value from this combination using the (C, C, M) varying base number. The decimal base number will be $vsrc * C * M + vdest * M + tag$. The virtual **receive** extracts the message using a physical tag computed in a similar manner. Since threads control scheduling at the user level, it is important for all physical **receive** operations to be preceded with a call to **probe** which basically checks whether the message has arrived or not. If not, the thread must yield the processor to another thread by executing a **yield_thr**. If all the threads in a physical processor are waiting for a message and the message queue is empty, the first message that arrives allows a thread to take control and execute. The busy-waiting only occurs if all the threads within a physical processor block on receiving messages. The **yield_thr** operation is encapsulated in the virtual receive routine. Therefore, all virtual architecture algorithms that require communication must use the virtual communication routines, otherwise deadlock is likely. The prototypes for the virtual point-to-point communication routines and the **vwho** are shown in Figure 15.

Many virtual architecture parallel algorithms contain aggregate or collective communication operations which require participation of all the virtual processors. A library designed based on our scheme allows a number of threads or virtual processors to be mapped to a set of physical processors. This mapping is many-to-one and may map zero or more virtual processors to a physical processor. To support virtual collective communication operations, we use our virtual point-to-point communication routines and assume all the virtual processors participate in the collective communication operations. However, in the collective communication calls which require a root processor, any one of the participants could be the root of the call.

We would like to support *virtual collective communication* routines which may be used by the library designer to implement algorithms that require such operations. There are a number of existing libraries which support collective operations. However, these collective communication libraries are designed to function at the physical processor level and do not support dynamic process (or thread) creation. For instance, a broadcast operation may only require the broadcast message to be sent to a subset of the physical processors since this is where the virtual processors may reside. Similarly a multicast may send a message from one virtual processor to a subset of the virtual processors that happen to reside over all the physical processors of the system.

In our design of the collective communication library, the virtual point-to-point communication routines are used to build the virtual collective communication routines. The algorithms used to implement the virtual collective communication library routines are based on the logarithmic algorithms found in communication libraries such as [4]. The collective routines in these communication libraries do not allow for dynamic creation of processes, and function at the physical level. Our enhancements to these routines allow communication in the virtual architecture as opposed to the physical architecture. All collective communication library routines require participation of all the virtual processors of the currently active library call.

Figure 16 shows the list of virtual collective communication routines supported in our library system. **Vbroadcast** broadcasts *data* from the virtual processor, *root*, to all the virtual processors described by *handle*. **Vbroadcast** does not impose a barrier synchronization among the virtual processors in the handle. **Vreduce** performs a reduction over all the virtual processors in *handle* using the reduction function *rfunc* (a user defined associative function) and the *data*. The result is stored in the *root* virtual processor in the parameter *result*. **Vreduce** imposes barrier synchronization among the virtual processors of the currently invoked library call. **Vcombine** performs a reduction over all the virtual processors in *handle* using the reduction function *rfunc* and the *data*. The result is stored in all the virtual processors in the parameter *result*. **Vcombine** enforces barrier synchronization among the virtual processors of the currently invoked library call. Consider a line of 4 virtual processors and assume the processor i holds value i, where i ranges from 0 to 3. Further assume that a collective communication operation is executed by all the virtual processors and the virtual processor 0 is the root of the call. If the collective call is a **vbroadcast**, all 4 virtual processors will hold the value 0 after the call. If the collective call is a **vreduce** and the reduction function returns the maximum of two integers, virtual processor 0 will hold the value 3 after the call to **vreduce**. However, if the collective call is a **vcombine**, all virtual processors will hold the value

void vbroadcast(char *msg, int size, int tag, int root, LibHandlePtr handle)
broadcast the data in msg to all the virtual processors in handle. The root of the broadcast, message tag, and message size are root, tag, and size, respectively.

void vreduce(char *data, int Tag, int *result, int root, void (* rfunc), LibHandlePtr handle);
Perform a reduction over all the virtual processors in handle using the reduction function rfunc() and data. Stores the result at the root processor in location pointed to by result.

void vcombine(char *data, int tag, int *result, void (* rfunc), LibHandlePtr handle);
Perform a reduction over all the virtual processors in handle using the reduction function rfunc() and data. The result is stored in the result parameter of all the virtual processors.

void vprefix(char *data, int tag, int *result, int root, void (* pfunc), LibHandlePtr handle);
Perform a prefix reduction over all the virtual processors in handle using the prefix function pfunc(), data, and root. The result is stored in the result parameter of all the virtual processors.

void vsync(LibHandlePtr handle);
Implement a barrier synchronization among the virtual processors associated with handle.

Figure 16: Virtual collective communication library prototypes

3 after the call. **Vprefix** performs a prefix reduction over all the virtual processors in the *handle* using the *data*, the prefix function *pfunc*, and the *root* of the prefix operation. The prefix results are stored in the *result* parameter of all the virtual processors. **Vprefix** enforces barrier synchronization among the virtual processors of the currently invoked library call. In the previous example, if the collective call was **vprefix** and the prefix function basically returned the sum of its two parameters, virtual processors 0, 1, 2, and 3 would hold values 6, 6, 5, and 3, respectively. **Vsync** enforces a barrier synchronization among all the virtual processors of the handle.

Algorithms 4 and 5 are used to perform the virtual broadcast. The algorithms view the virtual

Algorithm 4 Supporting recursive routine for virtual broadcast

```
Require: Number of virtual processors, buffer, size, tag, and handle
  If number of virtual processors, c, is 1, return
  Divide number of virtual processors, c, into two halves ( for odd size, one half will be one larger)
  The root sends the buffer to the virtual processor $\lfloor \frac{c}{2} \% C \rfloor$, where $C$ is the total number of virtual
  processors in this phase (it also attaches the number of virtual processors it should broadcast to)
  Call recursively with the root and the first half of virtual processors
  Call recursively with the middle virtual processor as the root and the second half of virtual
  processors
```

processors in form of a ring. This is because we would like to allow any virtual processor to be the root of the broadcast. The second part of the algorithm is the one that interfaces with the library routine (**vbroadcast**). If the virtual processor is the root processor, it calls the first algorithm which sends the message to the virtual processor half way around the ring. It will then repeat the

Algorithm 5 Virtual broadcast algorithm

```
Require: Number of virtual processors, buffer, size, tag, and handle
  if I am the root of the broadcast then
    Call the above algorithm
  else
    Receive message
    Extract the number of virtual processors to broadcast this message to
    Call the above algorithm with the number
  end if
```

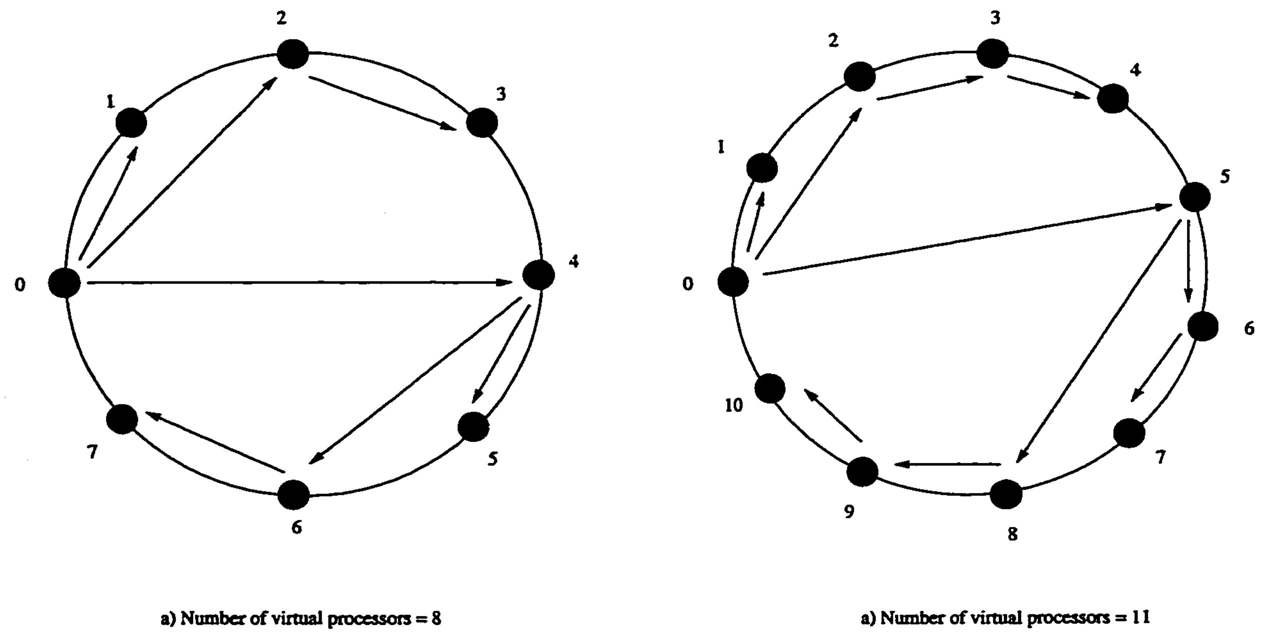

0
1
2
3
4
5
6
7
a) Number of virtual processors = 8
0
1
2
3
4
5
6
7
8
9
10
a) Number of virtual processors = 11

Figure 17: Virtual broadcast communication operation

process for the first half of the ring. The **else** section of the second part of the algorithm is meant for the virtual processors that are recipients of the message. These virtual processors will then call the first algorithm to recursively repeat the broadcast on the portion of the ring that is assigned to them. The point-to-point communication operations to perform the broadcast on 8 and 11 virtual processors are shown in Figure 17. In both examples, the root of the virtual broadcast is the host virtual processor.

The **vsync** routine, which implements a barrier synchronization among all the virtual processors, uses a reduction followed by a broadcast operation. The argument to **vsync** is just the handle of the parallel phase. The reduction operation is performed using zero size messages. The implementation of the collective virtual communication routines can be found in Appendix C.

3.1.3 Library Routine Handle: A Dynamic Entity

A *Handle* is a dynamic entity that represents a virtual architecture library routine. Each invocation of a library routine requires creation of a new handle which will then contain information about the

```
struct LibHandle                                 /* Handle */
{
    ArchitecturePtr architecture;                /* Pointer to architecture type */
    int NumParams;                               /* Number of parameters */
    enum ParamType * ClassParams;                /* in/out/inout parameter */
    BlockingPtr contraction;                     /* Blocking of data */
    MappingPtr mapping;                          /* Mapping of the reduced architecture */
    DistributionPtr init_layout;                 /* Initial data layout */
    DistributionPtr fin_layout;                  /* Final data layout */
    UserDataPtr local_data;                      /* User data buffers */
    struct LibHandle * next;                     /* Pointer to the next handle */
} LibHandle;
```

Figure 18: Definition of a library routine handle

call. This information consists of the topology, dimensionality, size, mapping, and the contraction factor. There is also information about the initial and final data layout and the actual user data for the computation. A new handle is allocated upon entering a parallel library call.

Each physical processor holds a distinct copy of the handle for the current phase and the history. The history is a list of all the handles that have been generated since the start of the execution of the program. Newly created handles are appended to the beginning of the list. If a handle may not participate in data redistribution, it is disposed. This happens when all distributed data of a handle are used for redistribution and marked as stale. All physical processors hold similar copies of the current handle and the history of handles with the exception of the content of the distributed data. Since our library system supports many-to-one mapping, necessary space is allocated for all the virtual processors that are mapped to a physical processor. We will discuss the overhead of maintaining handles and the history list in the later part of this chapter.

The declaration for a handle is shown in Figure 18. Figure 19 shows this structure in a pictorial form. The components of the handle are used in supporting contraction, automatic data distribution, and redistribution. **Architecture** contains information about the virtual topology and its size. **NumParams** and **ClassParams** hold information about the number and the type of each parameter of the library routine, respectively. The parameters are of type **IN**, **OUT**, or **INOUT**. **IN** type parameters do not need to be accumulated back at the virtual host processor since their contents do not change. **OUT** and **INOUT** parameters must be accumulated at the virtual host processor. There is no initial data distribution associated with the **OUT** parameters and they cannot be subject of redistribution on entering a parallel library call. The redistribution module checks for these types of parameters and bypasses them. The type of a parameter specifies whether it is call by value or call by reference. **Contraction** specifies the contraction of the data along each dimension of the data. **Mapping** describes the assignment of the virtual processors to the physical processors, the rank of each virtual processor on the physical processor it is mapped to, and the actual thread identifier of the represented thread. **Init_Layout** and **Fin_Layout** hold the initial and final data layout of each parameter, respectively. The data blocks are numbered in a row or

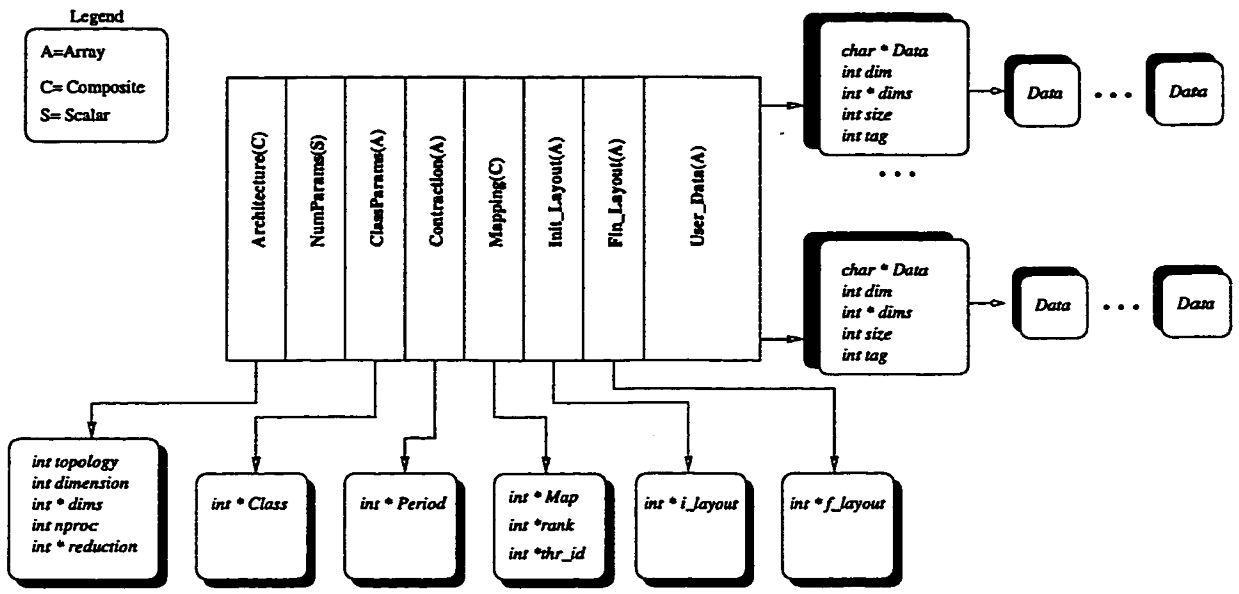


Figure 19: Pictorial view of a library handle

column major form. As long as the library designer keeps the numbering of the blocks consistent, it does not pose any problem. **local_data** contains the actual distributed data along with all the corresponding attributes such as size, dimensionality, and the message tag used for communicating the contents of the data buffers. The data for the threads that are handled by a processor is allocated in the handle **local_data** field. Therefore, the allocation of the data for the threads must be done after the mapping stage. The **next** field points to another handle. This field is used to keep a list of handles for previously executed parallel library calls. We will elaborate on this in the data redistribution section.

The **Architecture** component defines the reduced and the fine grain virtual architectures and the mapping table. For each one, the total size, dimensionality, and the size along each dimension is kept in the structure (Figure 20). This structure contains the topology identifier (one of the enumerated values MESH, TORUS, LINE, RING, HYPERCUBE, *etc.*), number of dimensions and the size along each dimension in the fine grain and the reduced form. Once a parallel library receives the size of the problem instance, it can determine the values for the fine grain architecture. When the library routine performs grain adjustment, the coarse grain architecture attributes will be set.

The **Distribution** component defines the mapping of blocks onto the reduced virtual architecture. The mapping table assumes a linear numbering of the reduced virtual processors as well as the contracted data blocks (Figure 21). For each distributed parameter of a library call an instance of this structure is allocated. The **mapping** structure describes the mapping of the reduced virtual architecture to the physical architecture. This mapping depends on the physical system topology and size, and is determined at the time of the library routine invocation (Figure 22). Each entry of the mapping table consists of the physical processor identifier, the rank of the virtual processor, and the actual thread identifier of the represented virtual processor. The user data is also maintained in the handles (Figure 23). The handle resides in the global memory of each processor, therefore it

```
typedef struct Architecture         /* Virtual Architecture Declaration */
{
    enum Types topology;            /* Topology */
    int dimensionality;             /* Dimensionality of virtual architecture */
    int * fine_dims;                /* Size of each dimension before contraction */
    int fine_nproc;                 /* Total number of virtual processors - fine grain */
    int * period;                   /* Describes the reduction factor */
    int * coarse_dims;              /* Size of each dimension after contraction */
    int coarse_nproc;               /* Number of virtual processors - coarse grain */
} Architecture;

typedef Architecture * ArchitecturePtr;
```

Figure 20: Structure for virtual architecture declaration

```
typedef struct Distribution         /* Data distribution declaration */
{
    int *data_map;                  /* Array of data block to processor mapping */
} Distribution;

typedef Distribution * DistributionPtr;
```

Figure 21: Structure for data mapping

```
typedef struct Mapping              /* Processor mapping declaration */
{
    int *map_table;                 /* Array of virtual to physical processor mapping */
    int *rank;                      /* Local index of the virtual processor */
    int *thr_id;                    /* Actual thread identifier of the thread */
} Mapping;

typedef Mapping * MappingPtr;
```

Figure 22: Structure for processor mapping

```
typedef struct UserData             /* Local Data */
{
    char * data;                    /* Pointer to local data partitions */
    int dimensionality;             /* Dimensionality of the local data data */
    int *dims;                      /* Size of each dimension */
    int size;                       /* Total size of data partition */
    int tag;                        /* For communication purpose */
} UserData;

typedef UserData * UserDataPtr;
```

Figure 23: Structure for distributed data

```
typedef struct BlockingPtr              /* Contraction parameters */
{
    int dimensionality;                 /* Dimensionality of the data */
    int fine_nblock;                    /* Number of blocks in fine grain */
    int *fine_dims;                     /* Dimensions in fine grain */
    int *period;                        /* Blocking period in each dimension */
    int coarse_nblock;                  /* Number of blocks in coarse grain */
    int *coarse_dims;                   /* Dimensions in the coarse grain */
} Blocking;

typedef Blocking * BlockingPtr;
```

Figure 24: Structure describing data contraction

is accessible between two phases for data redistribution. There is one instance of this structure for each distributed parameter of the library call. The **data** field contains data for all the threads that run on the processor on which the handle resides. Each thread uses its rank (available through the mapping table or the parameter) to access its own data in the **data** field. Care must be taken to access the data properly since user level threads do not check for illegal memory access.

The **blocking** structure describes the contraction factor for the data along each dimension (Figure 24). There is also one instance of this block for each distributed parameter of the library call. The fields specify the period along each dimension of the data. The number of blocks in the coarse and fine grain data. This declaration for contraction allows square or rectangular blocking of data, as long as the period along each dimension is constant. Most block matrix algorithms use regular blocking of data (square or rectangular blocking).

3.1.4 Representation of Data and Processor Mapping

The virtual architecture algorithms are based on distributed vectors, matrices, or multidimensional objects. The size of these data items is passed as parameter in every call to a library routine. In order to perform automatic data distribution and data redistribution we encapsulate data distribution of the matrices in the library. The objective is to internally represent the location of data blocks in the virtual architecture and the location of the virtual processors in the physical system. This representation must facilitate the thread representation of virtual processors in our design and provide necessary information for data redistribution.

Parallel library routines operate on distributed data. Global data is mapped to the physical architecture through three levels of mapping. The first level of mapping assigns elements of the global index to the local indices of the blocks. The second level of mapping assigns the blocks to the virtual processors. The third level of mapping is from the virtual processors to the physical processors. The data blocks mapped to the virtual architecture follow the third level of mapping to the physical architecture.

Mapping from the global indices to the local indices of blocks is basically a special case of block-cyclic mapping supported in HPF [52]. Our virtual architecture algorithms acquire blocks of data in

m	0	1	2	3	4	5	6	7	8	9
β	0	0	0	1	1	1	2	2	2	3
i	0	1	2	0	1	2	0	1	2	0

(a) $r = 3$

m	0	1	2	3	4	5	6	7	8	9
β	0	1	2	3	4	5	6	7	8	9
i	0	0	0	0	0	0	0	0	0	0

(b) $r = 1$

m	0	1	2	3	4	5	6	7	8	9
β	0	0	0	0	0	0	0	0	0	0
i	0	1	2	3	4	5	6	7	8	9

(c) $r = 10$

Figure 25: Examples of block(r) data mapping for $M = 10$

a specific distribution dictated by the algorithm. Although the distribution function is hard-coded in the implementation, the data distribution function is kept in the handle for possible redistribution in the parallel phases that follow a phase at runtime. The mapping of the virtual processors to the physical processors also plays a role in the performance of the library routines.

The fundamental data objects in our library routines are partitioned matrices. For simplicity we will first consider a vector and then expand our analysis to two dimensional objects. The analysis can be easily extended to higher dimensional objects. We use $[a]$ to denote the set $\{0, 1, \ldots, a-1\}$ for an integer a, $[a] \times [b]$ to denote the Cartesian product of $[a]$ and $[b]$ for integers a and b, and $f : X \rightarrow Y$ to denote a function, f, from domain X to range Y, where X and Y are two sets.

Block(r) mapping of one dimensional data of m scalars can be described using a mapping function from the global index $[m]$ to an index pair in $[\beta] \times [r-1]$, or

$$[m] \rightarrow [\beta] \times [r-1],$$

where β is $\lfloor m/r \rfloor$. A value of $r = 1$ assigns a single element to each block giving rise to m blocks. On the other hand, a value of $r = m$ assigns all elements to a single block. The inverse mapping of elements within a block to the global index can then be described in the form:

$$[\beta] \times [r-1] \rightarrow [m].$$

Figure 25 illustrates examples of block(r) mapping for $r = 3$, $r = 1$, and $r = 10$, where $m = 10$.

Block(r, s) mapping of two dimensional data of m by n scalars can be described by applying two independent functions in row and column directions. Therefore, if the rows are grouped using block(r) and the columns are grouped using block(s), then $[\beta] \times [r-1] \rightarrow [m]$ and $[\gamma] \times [s-1] \rightarrow [n]$ can be used to derive the Block(r, s) mapping, where β is $\lfloor m/r \rfloor$ and γ is $\lfloor n/s \rfloor$. The tensor product of the row and column block mappings can be written as:

$$[m] \times [n] \rightarrow ([\beta] \times [\gamma]) \times ([r] \times [s]).$$

Values of $r = 1$ and $s = 1$ assign a single element to each block giving rise to $m \times n$ blocks. On the other hand, values of $r = m$ and $s = n$ assign all elements to a single block. The inverse mapping

m	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5
n	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
β	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
γ	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
i	0	0	0	0	1	1	1	1	2	2	2	2	0	0	0	0	1	1	1	1	2	2	2	2
j	0	1	2	0	0	1	2	0	0	1	2	0	0	1	2	0	0	1	2	0	0	1	2	0

(a) $r = 3$ and $s = 3$

m	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5
n	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
β	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5
γ	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) $r = 1$ and $s = 1$

m	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5
n	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
β	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
γ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5
j	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

(c) $r = 6$ and $s = 4$

Figure 26: Examples of block(r, s) mapping for $m = 6$ and $n = 4$

of elements within a block to the global index can then be described in the form:

$$([\beta] \times [\gamma]) \times ([r-1] \times [s-1]) \rightarrow [m] \times [n].$$

Figure 26 illustrates an example of block(r, s) mapping for three different pairs of values for r and s, where $m = 6$ and $n = 4$. The first case is shown in Figure 27 in block form.

Data distribution can be described using a function that specifies the location of each block of data in the logical architecture. We identify these functions at runtime since the specific function depends on the virtual architecture scaling factor and the logical architecture size. Assuming the data consists of B blocks in one dimension and a logical line or ring of P processors is used, we represent the initial data distribution with $\mathcal{D}_{init}$ and the final data distribution with $\mathcal{D}_{fin}$ that map block indices to the logical processors. These can be expressed as $\mathcal{D}_{init} : [B] \rightarrow [P]$ and $\mathcal{D}_{fin} : [B] \rightarrow [P]$. Mapping of the distributed data onto the global data can be described as an inverse mapping function $\mathcal{D}_{init}^{-1} : [P] \rightarrow [B]$ for initial data distribution and $\mathcal{D}_{fin}^{-1} : [P] \rightarrow [B]$ for final data distribution.

Now, for two dimensional blocks of data, assuming the data consists of R by S blocks and a logical mesh or torus of P by Q processors is used, the distribution functions can be stated as

0,0	0,0	0,0	0,1
0,0	0,0	0,0	0,1
0,0	0,0	0,0	0,1
1,0	1,0	1,0	1,1
1,0	1,0	1,0	1,1
1,0	1,0	1,0	1,1

Figure 27: Representation of block(3, 3) mapping for $m = 6$ and $n = 4$

	$\mathcal{D}_{init}(i,j)$	$\mathcal{D}_{fin}(i,j)$
A	$(i,(i+j) \bmod N)$	$(i,(i+j) \bmod N)$
B	$((i+j) \bmod N, j)$	$((i+j) \bmod N, j)$
C	-	(i,j)

Table 4: Initial and final distribution of data in the torus based matrix multiply

$\mathcal{D}_{init} : [R] \times [S] \rightarrow [P] \times [Q]$ and $\mathcal{D}_{fin} : [R] \times [S] \rightarrow [P] \times [Q]$. Mapping of the distributed data onto the global data can be described as inverse mapping functions $\mathcal{D}^{-1}_{init} : [P] \times [Q] \rightarrow [R] \times [S]$ and $\mathcal{D}^{-1}_{fin} : [P] \times [Q] \rightarrow [R] \times [S]$. The functions described can be easily modified to accommodate data distribution of blocked data with higher dimensions and logical architectures with other topologies.

The level of mapping just described defines the location of data blocks in the virtual architecture. A second level mapping defines the location of the virtual processors in the physical system. *Processor mapping* is a function, $\mathcal{M}$, from virtual processors to the physical processors. The data blocks follow the virtual processor on which they are mapped through this mapping function. The mapping of virtual to physical architecture is used by the virtual communication layer to find the physical location of the data blocks. The effect of the virtual to physical mapping function on the performance of the routine has been discussed on Page 85.

Let's consider the vector dot product algorithm. Elements at position i of the vectors are mapped to the i^{th} virtual processor. The distribution of both A and B can be represented by the function $\mathcal{D}_{init}(i) = i$ and the inverse mapping function is $\mathcal{D}^{-1}_{init}(i) = i$. Assume the mapping of n virtual processors onto the p physical processors, where p divides n, is in block form. Therefore, the i^{th} physical processor is mapped virtual processors $\frac{i*n}{p}$ to $\frac{(i+1)*n}{p} - 1$, inclusive.

Another interesting example is the matrix multiplication based on a torus topology described on Page 81. The initial data distribution of A and B can be described using the functions shown in Table 4. C does not have an initial distribution, but its final distribution is the identity function. Symbols i and j are function variables and N is the size of one dimension of the matrix. Figure 28 shows the initial and final data distribution functions for matrix multiply in a pictorial form. The virtual to physical processor mapping function used in the diagram maps the ith virtual processor

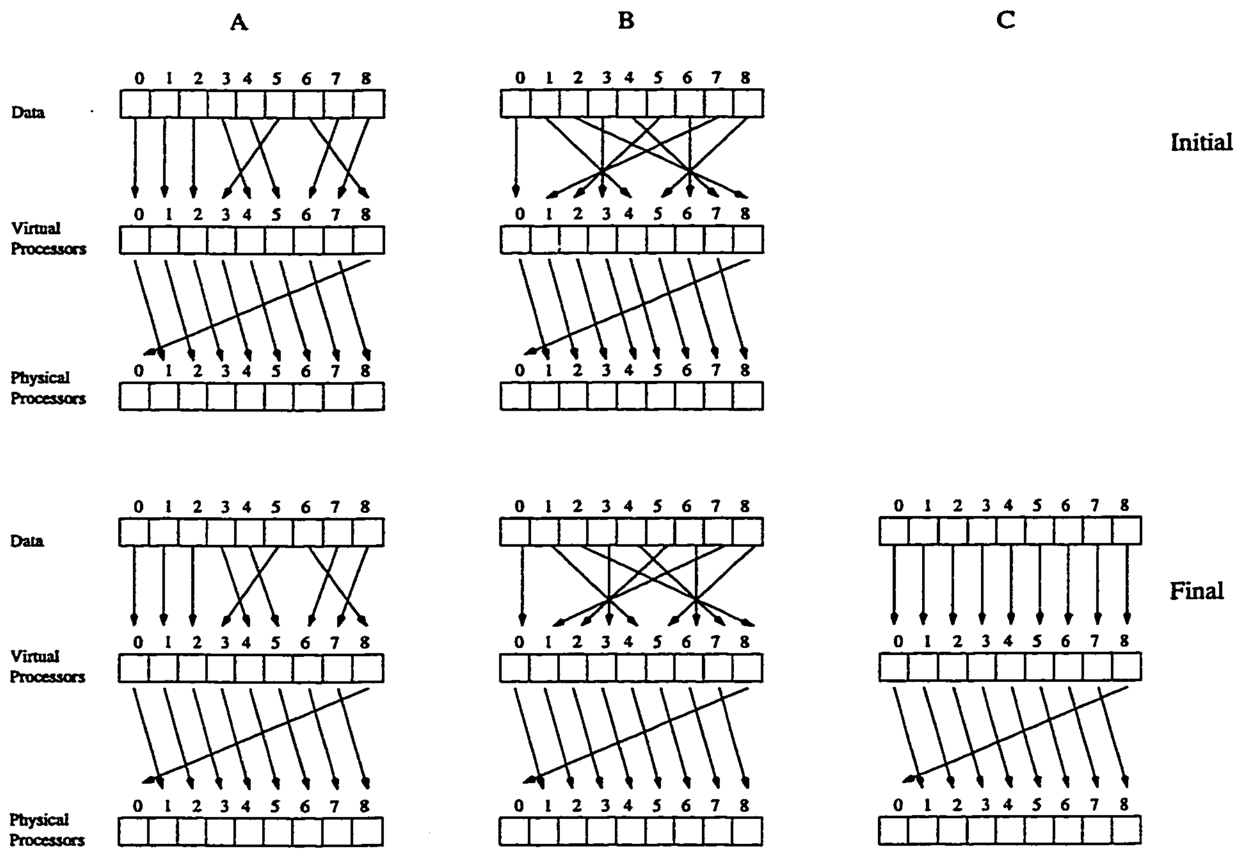


Figure 28: Initial and final data distribution for matrix multiply - 3 × 3 blocks of data onto a 3 × 3 mesh of processors

to the physical processor number $(i + 1)\%9$, where i takes on values from 0 to 8. This mapping function is only used for illustration here and does not bear any significance in the torus based matrix multiplication.

The data mapping and the processor mapping functions, together, determine the location of the data blocks in the target system. This information is used by the data distribution and redistribution components of our library routines. If the processor mapping function for a parallel phase is $\mathcal{M}$, and the data mapping function for a parameter in that phase is $\mathcal{D}$, the physical location of the data blocks can be obtained using $\mathcal{M} \circ \mathcal{D}$. If a parameter has a final distribution of $\mathcal{D}_{fin}$ in a phase which has processor mapping $\mathcal{M}_1$, and an initial data distribution of $\mathcal{D}_{init}$ with processor mapping $\mathcal{M}_2$ in a subsequent phase, the data can be redistributed using $\mathcal{D}_{init} \circ \mathcal{M}_2 \circ \mathcal{M}_1^{-1} \circ \mathcal{D}_{fin}^{-1}$.

The internal representation of the data objects in the handle consists of the size and dimensionality of each block and the location of each block in the virtual architecture. Even though the data distribution function is hard coded in a component of the library routine, the information on the initial and final data layout of each parameter is used between two parallel phases to determine the source and the destination sets for data blocks in a processor. The distribution of data is represented in two structures called **init_layout** and **fin_layout** (see Figure 19).

The mapping table, on the other hand, must facilitate communication among threads (virtual processors), mapping of several virtual processors to a physical processor, and data references by threads within one physical processor. Since we represent a virtual process by a thread, and a thread may only obtain its actual thread identifier, the mapping table must provide correspondence between virtual processors, their actual thread identifiers, and the physical processor which houses that virtual processor. A thread references its data items (in a contiguous location allocated for all the local threads) using its rank or local thread identifier. Furthermore, the virtual communication layer consults the mapping table to find the physical location of a destination virtual processor. It also consults the mapping table, on the receiving end, to deliver a message to the proper thread.

3.1.5 Handle Allocation and Initialization Modules

This module is the first one called by the main process in each physical processor. The information about the library call parameters such as their number, the type of each parameter (pass by value or reference) is assigned in this module. This module also allocates all necessary structures for which we need only one instance in the handle. When this module is called, the mapping is not yet determined, therefore the allocation of data for the threads must be delayed until the mapping is done. Algorithm 6 shows the two steps in initializing a handle of library call.

Algorithm 6 The handle allocation and initialization

```
Require: An empty handle
1: Allocate all structures
2: Initialize number of parameters and type of each
Ensure: An initialized handle
```

3.1.6 Virtual Architecture Initialization and Contraction Module

This module is the first one that obtains information on the problem instance size. The information is only available at the host and must be broadcast to all other processors. Once the processors obtain the size information, all other fields of the handle are setup simultaneously by all the physical processors. The syntax of the physical broadcast has been previously explained on Page 52. The broadcast is executed by all the nodes and the contents of the buffer is sent from the root physical processor (zero in this case) to all other processors. Algorithm 7 summarizes these steps.

Algorithm 7 The architecture initialization module high level algorithm

```
Require: a handle with all allocated components
1: Broadcast(size,message,tag,0)
2: Initialize topology, dimensionality, and size of each dimension
3: Initialize the fine grain architecture size
Ensure: An initialized architecture structure
```

The task of the contraction module is to close the size gap between the virtual and the physical system. This module determines the granularity at which the parallel library routine communicates. Since the input size is known in this module and the physical system is fixed, this module can determine the best contraction factor (discussed in section 3.3). Once the contraction factor is computed, the reduced virtual architecture size will be fixed and a mapping from the virtual architecture to the physical architecture is performed prior to the execution. Algorithm 8 shows the steps in performing

Algorithm 8 The contraction module high level algorithm

```
Require: a handle with all known attributes of the call
1: Get handle for the current call
2: Scale the algorithm down to the physical system size
3: for all distributed parameters of the call do
4:     Allocate arrays for period, fine and coarse grain dimensions
5:     Initialize attributes of fine and coarse grain data and the period
6: end for
Ensure: A contraction factor that minimizes execution time of the library routine
```

contraction. Step 2 in the algorithm is crucial to our library design. This is where the runtime grain adjustment is done. A series of guidelines to approximate the execution time function can be found on Page 42. The execution time function can be evaluated for all the grain sizes that divide the problem size. The grain size that minimizes the execution time will be used to execute the call.

3.1.7 Mapping Module

The task of this module is to close the gap between the virtual topology and the physical topology through systematic mapping. Once this module is done, the physical nodes are each assigned zero or more virtual processors to emulate. The mapping of the virtual processors to the physical processors is supported by a set of routines in a mapping library. Algorithm 9 shows the steps in performing the mapping. The contents of the mapping table are fixed after the grain size of the computation is

Algorithm 9 The mapping module high level algorithm

```
Require: a handle with all known attributes of the call and the contraction factor
 1: Get handle for the current call
 2: Get reduced virtual topology size and dimensionality from the handle
 3: Get physical system size and topology
 4: Perform assignment of virtual processors to the physical processors
 5: Place information in the handle
Ensure: A proper mapping of the virtual architecture to the physical one
```

selected. Mapping of the scaled down virtual architecture to the physical system is done at runtime. The mapping table information is then used to create the necessary number of threads on each processor. For each thread, the necessary space is allocated in the handle. The total space in a physical processor for each parameter is proportional to the number of threads on that processor and the contraction factor.

The mapping module calls a routine in the mapping library to determine the contents of the mapping table. If the system is partitionable, this is simply a call to a specific routine in the mapping library since both the virtual and physical topologies are known. However, if the system is reconfigurable, this module must check the topology of the target system before calling the appropriate mapping function. Although reconfigurable systems require setting up more mapping functions, the application program may benefit form virtual and physical topology match which reduces the communication overhead. Since reconfigurable systems can be configured for different topologies, the mapping library must take into consideration all the possible physical topologies. For instance, if the virtual architecture algorithm is based on a torus and the system is partitionable based on a mesh, only one mapping function from torus to mesh suffices for the implementation of this algorithm in the library. However, if the system is reconfigurable, one must consider mapping of torus to all possible physical topologies. The library routine implementor must consider number of mapping functions equal to the number of possible physical topologies. If the physical system is configured as a torus, the library routine will benefit from the match between the virtual and the physical topology.

The mapping module consists of a set of routines which initialize the mapping table of a handle. Each routine is designed to define the mapping from one topology to another. A library routine is aware of its topology and can query the physical system topology. Hence, it can call the specific mapping routine in the mapping library. Each routine in the mapping library is designed to initialize the mapping table with the physical processor on which each virtual processor runs. The indices of the mapping table are the virtual processor identifiers and the contents are the physical processor identifiers. Other information are added to the entries when the threads are created (see algorithm 14 on Page 76). The mapping library (which has a header file containing the definition of a handle) is compiled once and exists in object form. The object code is then linked in with the compiled SPMD code, the virtual communication library, the low level communication library, and the threading library. There are no references from the mapping library to any of the routines in the other modules. The only reference to the routines in the mapping module is from the individual library implementation of the virtual architecture algorithms. The call is made after determining the contraction factor. This call is made by the main flow of execution and before any threads are created. In fact, the mapping table is used for the creation of the threads on each physical processor (see Figure 30 on Page 77).

If the parallel library routine requires a very unique mapping function, this may be implemented in the library routine itself. Only commonly known mapping functions must be added to the mapping library as they may be used by several library routines. If a new virtual topology or physical topology is to be added to the mapping library, the procedure is very simple since the mapping library is basically a set of functions that map various virtual topologies to physical topologies. It is finally important to note that our proposed design supports many-to-one mapping. Some processors may even have no mapped virtual processor. The runtime granularity adjustment has high potential of using a fraction of existing processors when the problem size is small and the system size is large.

3.1.8 Data Distribution Module

This module utilizes the information from the previous modules, namely the contraction factor and the mapping table to decompose, assemble, and send the data to the appropriate virtual processors. The location of the initial data is strictly determined by the parallel algorithm used for the implementation of the library routine. Each node is then blocked waiting for the initial data from the host processor. Algorithm 10 shows the steps in performing the initial data distribution by the host

Algorithm 10 The initial data distribution module

```
Require: A handle with all known attributes of the call, contraction factor, global data, and the
    mapping table
 1: Get handle for the current call
 2: if I am the host virtual processor then
 3:     Perform blocking of global data
 4:     Distribute partitions to the processors using virtual send
 5: end if
 6: Get initial data using virtual receive
 7: Place data in the proper location in the handle (use rank via vwho)
Ensure: Data will be distributed based on the initial data distribution function
```

and the nodes. In our implementation of the library routines, the steps 6 and 7 are executed in the local computation routine. In effect, the host virtual processor executes steps 3 and 4 and the rest of the virtual processors execute steps 6 and 7.

In block matrix algorithms, blocks of data must be assembled into contiguous space and broadcast to the virtual processors. In most systems, the host has a larger memory than the nodes to hold the original data in full. These partitions are not statically defined. The size of the partitions is dynamically determined and the assembling of data incurs runtime overhead prior to the commencement of a parallel phase. A similar problem exists when data is to be received by the host. The partitions are received by the host processor into contiguous space and must be broken up to be placed in the appropriate locations of the global matrix. An example of a 6×6 matrix in single element row major order form and its counterpart with a contraction factor of 2 is shown in

Figure 29. Note that the change of storage is only required for multi-dimensional data. Once the computed data is sent to the host, the storage must be changed back to comply with the default storage mechanism. It is noteworthy to mention that the storage cannot be changed in place, as the data may be needed in a subsequent sequential phase. An additional copying is necessary here, however this copy can be disposed of as soon as partitions are launched to be sent to the processors.

The data distribution module creates a bottleneck that is inevitable. Our design of the libraries has a tendency of using large message sizes which allows some toleration of the data distribution latency. Once a partition is launched and is in transit the main virtual processor can assemble the next partition and create a pipelining effect. The data distribution overhead is examined at the end of this chapter. Chapter 4 describes our data redistribution support to reduce this overhead.

3.1.9 Data Accumulation Module

This module, which requires cooperation between the host and the node programs, collects back the results of the parallel computation and assembles the results in the global data. Algorithm 11 shows

Algorithm 11 The final data accumulation module

```
Require: Data sent to host by the processors, contraction factor, global data, and mapping table
1: Get handle for the current call
2: if I am the host then
3:     Get data from the processors
4:     Perform expansion of data to change storage method
5: else
6:     Send result to the host (use rank via vwho)
7: end if
Ensure: Computed result will be in the global data in proper storage form.
```

the steps in performing the final data accumulation by the host.

3.2 Conversion of the User Program to Single Program Multiple Data Form

Assume the users' sequential programs consist of parallel library calls that are based on our design of the library system. The parallel library calls may be nested within any language constructs or functions. For simplicity, we use a pseudo-C like language, however we are not bound to a specific language.

Our objective is to define a translation process from the user programs to the SPMD form which interfaces with our parallel library routines. The generated SPMD form must be based on standard language constructs, a threading library, and a communication library previously described. A sequential program with parallel library calls can then be transformed to an SPMD form, compiled using the resident compiler, linked with the thread library, communication libraries, and redistribution module to obtain a parallel executable image.

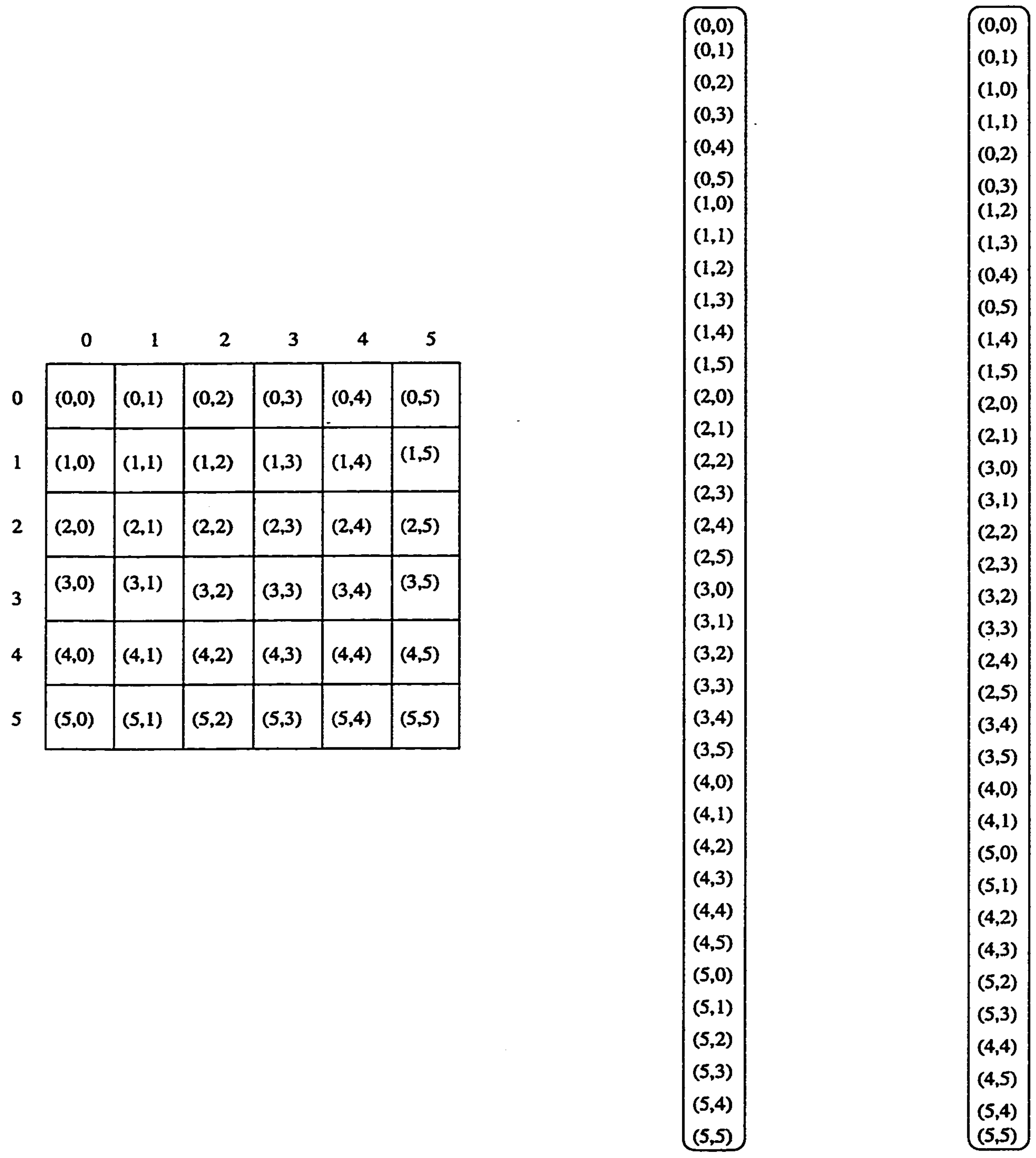


Figure 29: Change of storage to accommodate block matrix algorithms(Single element row major storage to row major storage with block size of 2 × 2)

Our approach is to allow all the physical processors to execute the sequential portions of the program just like the host does. This is with the exception of some statements, such as I/O, that can be only executed by the host and special cases to be discussed shortly. We must ensure that any condition used in the user's program evaluates to the same value on all the physical processors. This will ensure that all the physical processors do or do not participate in a parallel library call (of course some may not be assigned any virtual processors). For instance, if the user program calls a parallel library routine in a loop a number of times, the node programs must iterate over the loop the same number of times and participate in the parallel computation (coherency). If the loop bound is a value input at runtime, the nodes will not see this value and therefore may not follow the same execution path. The translation proposed below ensures coherency among the physical processors. We refer to all non-scalar variables which are parameters of the parallel library calls as *poly* variables and all other variables of the program as *mono* variables. We refer to *writing* a variable to mean that its contents are altered. For instance when the variable occurs on the left side of an assignment, it is being written to. *Reading* a variable means that its contents are used in a computation or decision making. For instance, when the variable occurs on the right side of an assignment, it is being read from. We distinguish between read/write operations of a variable and performing I/O on the variable even if they are very similar as far as altering the values of the variables are concerned. Furthermore, we use the term *guard* to refer to restricting the context of an SPMD statement to be executed only by the physical host processor. The guards are implemented using a call to **who** and an **if** statement which checks the value of the virtual processor identifier. If the value is 0, the guarded statements are executed. As mentioned before, the physical host processor is the only one with the I/O capability and possibly has larger memory capacity. Mono and poly variables must not be mistaken with scalar and vector data, as a mono variable could be a scalar or a vector. The physical processors must have the same image of the mono variables as the host. All sequential computation done by the host can also be done by the nodes to have a consistent image over all the physical processors. Our translation of the user program to SPMD form consists of the following set of rules:

- All I/O statements must be guarded. These statements can only be executed by the physical host.

- All inputs of mono variables must be broadcast after the inputs. The call to the physical broadcast must be inserted after the guarded input statements because all physical processors must participate in the collective communication operation.

- All write operations to poly variables must be guarded. Poly variables could be static or dynamic. When dynamic allocation is used to create them, the allocation will only take place on the host (due to memory limitations of the nodes). Therefore, access to these variables in the nodes will cause memory access violation. This is because, in our design, dynamic poly variables are guarded to be allocated only at the host. This is, however, not the case when the data is static. These data will be allocated by all the nodes on the stack or in the data section of the virtual memory.

- All read operations on poly variables must broadcast the same value to all the physical processors. The elements of poly variables are distributed across the handles that reside on the physical processors. The host processor is the only one that has a full image of the poly data between parallel phases. Reading elements of poly variables will only result in a correct value on the host. Therefore this value must be broadcast to all the physical processors.
- Parallel library calls are translated to a sequence of calls to the library components and the threading library to create one thread for each virtual processor.

There is a performance penalty at the statements that input mono variables. These statements must be guarded and the value must be broadcast to all the nodes. Considering I/O operations are slow and these cases are scarce in the numerical applications, the broadcast overhead would be tolerable.

Another problem has to do with the poly data. If the poly data is static, it will occupy unnecessary space on the nodes. Since in most programs the problem instance size is input at runtime, this is quite rare (and not recommended in our design). If the poly data is dynamic, its allocation is detected and guarded to be executed only by the host physical processor. The same holds for input of poly data. No broadcast is, however, required. The poly data is distributed across the processors when the library routine is called.

The source to source transformation consists of the steps in Algorithm 12. Determining the read

Algorithm 12 Source to source transformation - sequential program with library calls to SPMD program

```
1: Construct an abstract syntax tree
2: Identify parallel library calls and the type of their parameters
3: Identify all mono and poly variables
4: Identify the variables in the read and write sets of each statement
5: Identify Input/output statements
6: for all input/output statements do {Insertion of guards for I/O statements}
7:     Insert guard
8: end for
9: for all writes to poly variables do {Insertion of guards for writes to poly data}
10:    Insert guard
11: end for
12: for all input statements of mono data do {Insertion of broadcast for input statements of mono
    data}
13:    Insert a call to broadcast after its guard
14: end for
15: for all reads of poly variables do {Insertion of broadcast for reads from poly data}
16:    Transform statement to copy to temp, broadcast temp, and use temp for read.
17: end for
18: for all library call statements do {Translate library calls}
19:    Generate statement to assemble parameters of the call in a list
20:    Generate sequential prologue and thread creation (Algorithm 13)
21: end for
22: Scan the tree and dump the SPMD code
```

and write sets of statements requires interprocedural analysis if the user program has sequential calls. The sequential call may read or write the parameters or the global data.

The SPMD code generation is described in two steps. The first is the manipulation of the abstract syntax tree (AST), and the second is the dumping of the SPMD code. These two steps may be combined to perform fewer tree traversals. For clarity we have described these in phases. Once an AST is built, the library calls, I/O statements, and dynamic allocations are identified. The data is also categorized in two groups, mono and poly data. In our design, the parameters of the call are packed in a single list and passed to the thread. A naming convention is used to obtain the name of the thread for each library routine. This method is used throughout the transformation. The parallel library calls have a specific prefix (**cpcc**). The library component names are constructed from the concatenation of the suffix and the action of the component. For instance, a call to **cpcc_matmpy** has the initial data distribution component **matmpy_distribute_initial**. The suffix is **matmpy** and the action of the component is **distribute_initial**.

As it was previously mentioned, in our parallel library, virtual processors are emulated using threads. The user program is converted to an SPMD form which creates necessary threads at each parallel phase and joins them at the end of the phase. Prior to creation of the threads, the main process allocates the necessary space in the handle for all the threads. Although the data of one thread is discriminantly available to other local threads[5], the library designer must ensure the boundaries of data for each thread are protected. Since data from multiple threads reside in the handle, a local thread identifier is assigned to each created thread and is stored in the mapping table. The local index of each thread may be used by the thread to access its own private data buffers. Algorithm 13 shows the translated form of a single library call **libcall(params)**. The

Algorithm 13 Translation of a single library call

```
1: argList ← parameters
2: allocate_handle(&curHandle);
3: libcall_init_handle(curHandle);
4: libcall_init_arch(curHandle,argList);
5: libcall_contraction(curHandle);
6: libcall_mapping(curHandle);
7: libcall_allocate_local(curHandle,argList);
8: libcall_data_layout(curHandle);
9: redistribute(curHandle,&history);
10: rank ← 0
11: for all virtual processors vp mapped to current physical processor do
12:    thr_param ← append to argList the vpid and rank
13:    thr_create(0, 0, thread_lib_call, thr_param, THR_DETACHED, &new_thr)
14:    rank←rank+1
15: end for
16: for all virtual processors vp mapped to current physical processor do
17:    thr_join(0,0,0);
18: end for
```

thread_lib_call is described in Algorithm 14. Note the term *rank* is the same as local index of a thread. This value is used by a thread to access its own data partitions. Furthermore, the virtual

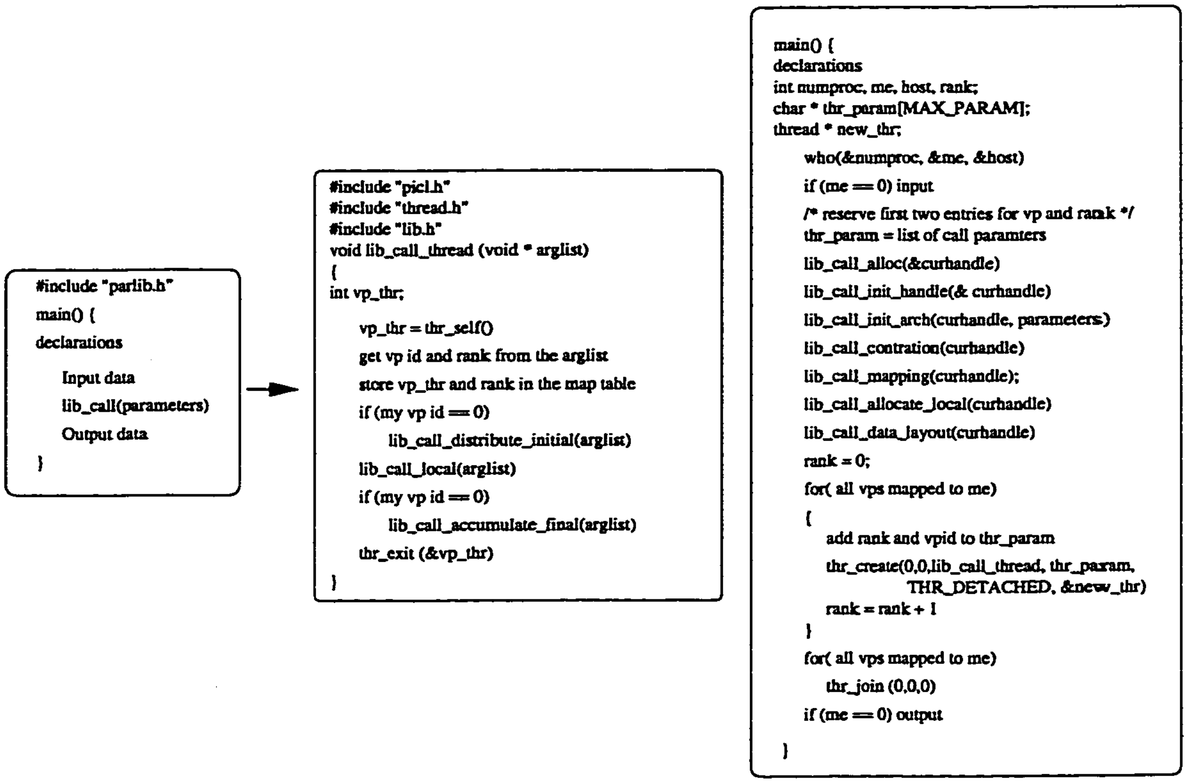
[5]Threads reduce the overhead of scheduling at the cost of loss of memory violation checking by the operating system. It is the job of the programmer to ensure that a thread does not refer to another thread's data.

Algorithm 14 The thread of a library call

```
Require: Arguments: argList(parameters, vpid, rank)
 1: self ← actual thread id (call to thr_self)
 2: Store rank and self thread ID at index vpid in the mapping table
 3: if I am the host virtual processor then
 4:     libcall_distribute_initial(argList, curHandle)
 5: end if
 6: libcall_local(curHandle);
 7: if I am the host virtual processor then
 8:     libcall_accumulate_final(argList, curHandle)
 9: end if
10: thread_exit(&self);
```

processor identifier of a thread can be obtained using the actual thread identifier query (by a call to **thr_self**) followed by searching the mapping table which houses the physical processor identifier, the rank (or local thread identifier), and the actual thread identifier of the resident virtual processors. The index of the mapping table corresponds to the virtual processor identifier. The index of the mapping table is the virtual processor identifier (see description of **vwho**). The arguments of the library call are either scalar values or pointer to data objects. These arguments are placed in an array and padded with the virtual processor ID and the rank. The array is then provided as the only argument of the thread. This array can be passed to other library routines without being decomposed. The library routine is implemented based on a special ordering of the parameters. Therefore, the locations of all the parameters within the array are known. The current handle does not need to be added to this list since it is available globally to all the threads and the called library components (as an external variable).

The transformation for a parallel library call is shown in figure 30. The parameters of the library call are assembled in an array of pointers starting from the third entry. The first and second entries are used for the virtual processor identifier and the rank of the thread. When the main process initializes the handle for the library call, it creates the threads by scanning the mapping table and creating one thread for each virtual processor mapped to the physical processor. This is done simultaneously by all the physical processors. After creation of the threads, the main process waits for the threads to come back by executing **thr_join**. The parameters of **thr_join** are 0 since the identifiers and the statuses of the threads that join are not important to capture. In the next example, we look at a slightly more complex sequential program (Figure 31). In this example a value which is the upper limit of a **for** loop is entered. The body of the **for** loop consists of two parallel library calls. You could think of the first one being a matrix multiply and the second one being matrix copy. The first input statement reads the value of *num_iter* which is used as the upper limit of the **for** loop. This value must be broadcast to all the nodes so they all iterate the same number of times. The input values for the parameters of the parallel library call do not need to be broadcast since they will be distributed as part of the parallel library call. The two parallel library calls are translated in the form previously described in Algorithms 13 and 14. It is important to note that even though we have shown the threads of the library calls as part of the code generation,



```
#include "parlib.h"
main() {
declarations
    Input data
    lib_call(parameters)
    Output data
}
```

```
#include "picl.h"
#include "thread.h"
#include "lib.h"
void lib_call_thread (void * arglist)
{
int vp_thr;
    vp_thr = thr_self()
    get vp id and rank from the arglist
    store vp_thr and rank in the map table
    if (my vp id == 0)
        lib_call_distribute_initial(arglist)
    lib_call_local(arglist)
    if (my vp id == 0)
        lib_call_accumulate_final(arglist)
    thr_exit (&vp_thr)
}
```

```
main() {
declarations
int numproc, me, host, rank;
char * thr_param[MAX_PARAM];
thread * new_thr;
    who(&numproc, &me, &host)
    if (me == 0) input
    /* reserve first two entries for vp and rank */
    thr_param = list of call paramters
    lib_call_alloc(&curhandle)
    lib_call_init_handle(& curhandle)
    lib_call_init_arch(curhandle, parameters)
    lib_call_contration(curhandle)
    lib_call_mapping(curhandle);
    lib_call_allocate_local(curhandle)
    lib_call_data_layout(curhandle)
    rank = 0;
    for( all vps mapped to me)
    {
        add rank and vpid to thr_param
        thr_create(0,0,lib_call_thread, thr_param,
                        THR_DETACHED, &new_thr)
        rank = rank + 1
    }
    for( all vps mapped to me)
        thr_join (0,0,0)
    if (me == 0) output
 }
```

a) User program

b) Thread component of the virtual architecture parallel library routine

c) Transformed user program (SPMD form)

Figure 30: An example of source to source transformation

```
main() {
declarations

     input num_iter
     Input data for parameters
     for(i=0;i<num_iter; i++)
     {
        lib_call1(..., a, ...)
         if (a[i] > 5)
           lib_call2(parameters)
     }
     Output data
}
```

a) User program

```
include files
void * thread_lib_call1 ( void *arglist)
{

}

void * thread_lib_call2 ( void * arglist)
{

}
main()
{
declarations
int numproc, me, host;
     who( &numproc, &me, &host);
     if (me == 0) input num_iter;
     broadcast (num_iter, sizeof(num_iter), 0)
     if (me == 0) input data for parameters
     for(i=0;i<num_iter; i++)
     {    translation of lib_call1
          if (me == 0) temp = a[i];
          broadcast (temp, sizeof(temp), 0)
          if (temp > 5)
                  translation of lib_call2
     }

     if (me == 0) output data
 }
```

b) Translated SPMD form

Figure 31: An example of translation of a write to a poly variable

these threads have fixed code for each call and are integrated into the library system (See Algorithm 14). This will simplify the source to source transformation and the compilation process.

Algorithms 13 and 14 are used for translation of the library calls regardless of the context of the call. If the call occurs inside a loop or a conditional, our translation scheme ensures that all physical processors maintain a consistent view of the mono data upon decision making. Therefore, a conditional will evaluate similarly or a loop iterates the same number of times on all the physical processors. Revisiting Figure 31, the variable *num_iter* is a mono variable. Our translated form of inputing this variable guards the input statement and inserts a call to the PICL broadcast to ensure that all physical processors have the same view of this variable for the rest of the SPMD code. The portion of interest is the **for** loop which uses *num_iter* as the loop bound. All physical processors will iterate the same number of times and will participate in the two parallel library calls (but not necessarily assigned a virtual processor). This discussion applies to all other conditional and iterative programming constructs. In the same figure, reference to the ith element of the poly variable a, in the condition of the **if** statement, must be translated to assignment of this variable to a temporary location enclosed in a guard. The guard is to ensure this is done by the host only. After the guarded statement, all the physical processors execute the broadcast on the temporary variable. The **if** statement, executed by all the physical processors, is transformed to use the temporary variable in the condition as opposed to the poly variable. The poly variable may not have a corresponding location in the node processors (poly variables may be dynamically allocated). The temporary scalar variables can be easily inserted in the declaration section of the SPMD code. This declaration allocates a copy of the variable, for each physical processor, on the runtime stacks of the main processes. The copying operation can be viewed as creating a new mono variable, reading its contents from a poly data, and broadcasting it to all the physical processors.

User programs may contain calls to functions (not parallel library calls). These calls may pass mono variables as parameters to the library routine. Passing the mono variables as parameters is done by all the physical processors and does not need any transformation in the SPMD form. If the parameter of a non-library function call is a poly variable, the value must be copied to a temporary variable and broadcast to all the physical processor prior to the call. This is the normal processing for any read operation of poly variables. If the poly variable is passed by reference, any write operation to the formal parameter must be broadcast to all the physical processors. A compiler can determine, through interprocedural analysis, whether a formal parameter of a function corresponds to a poly variable. It can then translate the function body similar to the translation of the main program bearing in mind that specific formal parameters are poly variables. If the function body contains a parallel library call, all physical processors will still participate in the call.

3.3 Performance Analysis

In this section we show, through a motivating example, the necessary steps that must be followed to examine the performance characteristics of a virtual architecture algorithm. The performance characteristics are examined with respect to the grain size of the computation and the mapping of the virtual processors. For our library design, performance of virtual architecture parallel routines must be studied on a case by case basis. Without loss of generality, the framework used in this section can be applied to other algorithms of interest.

At the time of the implementation of a scientific parallel library routine the problem instance size is not known. This also holds for the system size and topology in many reconfigurable and partitionable systems. Algorithms which possess high range of scalability are more desirable for library implementation on these systems. These algorithms are more likely to deliver better performance when the physical system size and topology are known at load time and when the problem size is known at runtime. Through selection of highly scalable parallel algorithms, constructing library routines that adjust the parameters of the computation at runtime to reduce the execution time, and performing systematic runtime mapping of the logical processors, one can build parallel library routines which reduce the execution time of the caller sequential programs.

We consider two common communication models for message passing parallel computers. The first model captures the communication cost in multicomputers that use *packet switching.* The second model captures the communication cost in multicomputers based on *wormhole* routing (also known as *cut_through* routing). Communication between two virtual processors in the same physical processor incurs a cost proportional to the size of the message, which is basically the cost of a memory-to-memory copy. This is commonly not considered as part of a communication model, however in our analysis of the communication cost we require the model to be extended to define communication within a physical processor. The communication cost t_{comm} in packet switching is:

$$t_{comm}(S,d) = \begin{cases} t_s + (t_h + t_w S)d & \text{if } d > 0 \\ S t_m & \text{if } d = 0, \end{cases} \tag{9}$$

where t_s is the startup cost, t_w is the time to transfer one word across physical channel, t_h is the time delay at each router, d is the distance the message must travel, t_m is the time it takes to perform a memory to memory copy, and S is the size of the message in words. In wormhole routing, the communication cost is composed of the time for the head of the message to arrive at the destination and the time for the rest of the message to arrive in a pipeline fashion. Therefore, the communication cost can be expressed as,

$$t_{comm}(S,d) = \begin{cases} t_s + t_w(d + S) & \text{if } d > 0 \\ S t_m & \text{if } d = 0, \end{cases} \tag{10}$$

assuming a flit is one byte.

The primary difference between the two communication models is that in packet switching the packets are buffered at the intermediate nodes in the path. In wormhole routing, however, when the head of the packet arrives at an intermediate node, it is allowed to proceed without full buffering of the packet. The rest of the packet then follows the head. The value of $t_h d$ is usually much smaller than t_s and $t_w S$.

In the library implementation of an algorithm, the first step is examination of its scalability on the target system. If the algorithm is scalable, it is implemented in a parallel library in a parameterized form. Upon invocation, the library routine adjusts its granularity and mapping to minimize the execution time. If better execution time can be obtained by using a fraction of the available processors, this would be a logical decision to be made by the library routine. The following examples show the shortcomings in the current parallel libraries which are being overcome in the design proposed in this chapter. The first example shows the poor scalability of algorithms that are based on fixed granularity. The second one shows how execution time may be reduced by using only part of the, as opposed to the whole, available partition for the library routine execution. The third one exhibits drop in the performance of a routine as a result of naive mapping of the processors. The analysis is amenable to any of the algorithms in our domain. We have chosen the torus based matrix multiplication [54] virtual architecture parallel algorithm.

The pseudo code for $C = A \times B$, where A, B, and C are the global $n \times n$ matrices, is presented in the following algorithm. This algorithm represents the code executed by all the virtual processors. The symbols a, b, and c refer to the distributed data blocks of A, B, and C, respectively. The initial distribution of data is immaterial in this discussion, however it may be obtained from section 3.1.4. Assume the logical architecture is an $\frac{n}{g} \times \frac{n}{g}$ torus, g and n are powers of 2, and g divides n.

```
$a \leftarrow$ Initial $g \times g$ partition of $A$
$b \leftarrow$ Initial $g \times g$ partition of $B$
$c \leftarrow \mathbf{0}$
for $i = 1$ to $\frac{n}{g}$ do
    Send partition $a$ to the processor to my west
    Send partition $b$ to the processor to my north
    $c \leftarrow c + a \times b$
    Receive partition $a$ from the east
    Receive partition $b$ from the south
end for
Send computed $c$ back to host
```

The execution consists of $\frac{n}{g}$ steps each consisting of a computation step, comprising of a matrix multiplication and a matrix addition, and two communication steps. In the following analysis, the algorithm is considered in isolation, that is, without inclusion of data distribution and accumulation times. We are interested in the performance of the parallel portion of the program. Although, we will observe that any data distribution strategy will perform better using the design described in this thesis than for the existing libraries.

Now, we develop an expression for the communication overhead of the algorithm on a $\sqrt{p} \times \sqrt{p}$ processor torus. The expression will be a function of n, g, and p, and will involve the physical system parameters, previously defined in equations 9 and 10. We will then show that, the algorithm based on coarse grains is more scalable than the fine grain algorithms based on the isoefficiency metric of scalability defined by equation 4. When the number of processors p increases, fine grain algorithms require larger increases in the problem size, than coarse grain algorithms, to maintain a fixed efficiency.

Since the communication is between neighboring nodes, packet switching and wormhole routing are equivalent and either one of equations 9 or 10 may be used in the analysis. We assume a block mapping of the virtual processors to the physical torus in both dimensions. Assuming $\frac{n}{g} > \sqrt{p}$, each physical processor emulates $\frac{n}{g\sqrt{p}} \times \frac{n}{g\sqrt{p}}$ virtual processors. The physical processor at index (i, j), where $0 \leq i, j < \sqrt{p}$, emulates all virtual processors (u, v) such that $i * \frac{n}{g\sqrt{p}} \leq u < (i+1)\frac{n}{g\sqrt{p}}$ and $j * \frac{n}{g\sqrt{p}} \leq v < (j+1)\frac{n}{g\sqrt{p}}$.

The overhead function is composed of two terms. The first term accounts for the communication of all the boundary elements a and b to the neighbors. There are a total of $\frac{2n}{g\sqrt{p}}$ messages in each iteration. The second term account for the local communication among the virtual processors. There are a total of $\frac{2n}{g\sqrt{p}}\left(\frac{n}{g\sqrt{p}} - 1\right)$ local communication operations, each of size g^2. The cumulative overhead over $\frac{n}{g}$ iterations and p processors can be stated as,

$$T_o = \frac{2n^2\sqrt{p}}{g^2}t_s + 2n^2\sqrt{p}\left(\frac{n}{g\sqrt{p}} - 1\right)t_m,$$

where t_m is the time it takes to perform a memory to memory copy. Note the special case of $g = \frac{n}{\sqrt{p}}$, when each physical processor will emulate only one virtual processor and the overhead above simplifies to $2p\sqrt{p}t_s$, and when $g = 1$ the overhead becomes $2n^2\sqrt{p}t_s + 2n^3t_m - 2n^2\sqrt{p}t_m$. Using the isoefficiency equation, $W = KT_o$ we have:

$$W = \frac{2Kn^2\sqrt{p}}{g^2}t_s + 2Kn^2\sqrt{p}\left(\frac{n}{g\sqrt{p}} - 1\right)t_m, \tag{11}$$

where $K = \frac{E}{1-E}$, and t_c is the time to perform one iteration of the operation in the innermost loop of the sequential matrix multiplication algorithm. Since W is t_cn^3 for matrix multiplication, solving for n yields,

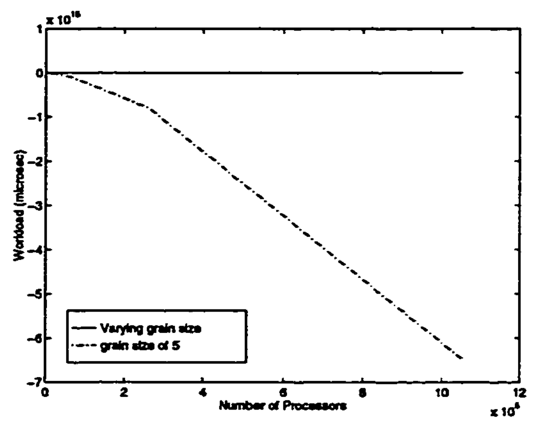
$$\begin{aligned} t_cn^3 &= \frac{2Kn^2\sqrt{p}}{g^2}t_s + 2Kn^2\sqrt{p}\left(\frac{n}{g\sqrt{p}} - 1\right)t_m \\ \Rightarrow t_cn &= \frac{2K\sqrt{p}}{g^2}t_s + 2K\sqrt{p}\left(\frac{n}{g\sqrt{p}} - 1\right)t_m \\ \Rightarrow n &= \frac{g}{g-2K}\left(\frac{2t_sK\sqrt{p}}{t_cg^2} - 2K\sqrt{p}t_m/t_c\right) \\ \Rightarrow n &= \frac{2Kt_s - 2Kg^2t_m}{g(g-2K)t_c}\sqrt{p}. \end{aligned}$$

Plugging this value of n back in equation 11 results in the isoefficiency function,

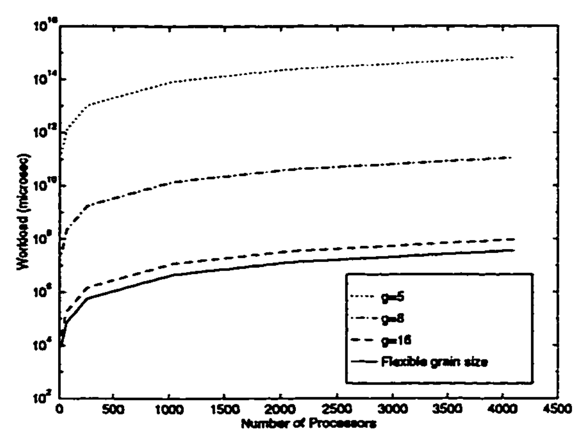
$$W = \left[\frac{2Kt_s}{g^2}\left(\frac{2Kt_s - 2Kg^2t_m}{g(g-2K)t_c}\right)^2 + \frac{2Kt_m}{g}\left(\frac{2Kt_s - 2Kg^2t_m}{g(g-2K)t_c}\right)^3 - 2Kt_m\left(\frac{2Kt_s - 2Kg^2t_m}{g(g-2K)t_c}\right)^2\right]p^{3/2},$$

The first term in the brackets is due to the nonlocal communication and the second and third terms are due to the local overhead. For the special case when $g = \frac{n}{\sqrt{p}}$, each physical processor will emulate only one virtual processor and the isoefficiency function becomes $W = 2Kt_sp^{1/3}$.

For a given problem and system size, the coefficient is a function of g. We plot this coefficient for different values of g using different initial values for efficiency. Figure 32(a) demonstrates that the matrix multiplication based on the fixed grain size is poorly scalable for an efficiency value of 0.8. The values of t_s and t_m were used from the specification of Intel Paragon. These values are



(a) E=0.8



(b) E=0.7

Figure 32: Comparison of the isoefficiency functions for $g = 5, 8, 16$ and $g = \frac{n}{\sqrt{p}}$

$t_s = 30$ μsec, $t_c = 0.02$ μsec, and $t_m = 0.08$ μsec. The value of t_m, which is the time to perform a memory to memory copy, is assumed to take four clock cycles of the 50 MHz processor of the Paragon. Value of t_c, which is the time to perform the statement in the innermost loop of the sequential matrix multiplication, was obtained using the simulator by dividing the total sequential execution time by n^3 and averaging the values over several runs($t_c = 3.7$ μsec).

The negative values for workload in figure 32(a) are indicative of the fact that, for an initial efficiency of 0.8, the fixed grain size algorithms cannot deliver the necessary performance to keep the efficiency constant when the system size increases, no matter what the rate of increase of the problem size is. Figure 32(b) has been plotted for an efficiency of 0.7. The workload axis has been scaled down logarithmically. Both types of algorithm can sustain a fixed performance for this value of efficiency[6]. However, this figure demonstrates that the algorithm based on the largest grain size of $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ is far more scalable than the fixed grain size algorithms with $g = 5, 8,$ and 16. The grain size of 5 has been selected for comparison since in ScaLAPACK it is claimed to deliver the best performance on average. The scalability analysis presented here pertains to all block matrix algorithms in our domain.

The ultimate goal in parallel libraries is to minimize the execution time of the routines. Once a scalable algorithm is implemented in a parallel library, the only degrees of freedom are the granularity of the computation and the mapping of the virtual processors to the physical processors. There are many system attributes that are important factors in the execution time of the library routine. These include the speed of the processors, the topology and the size of the physical system, the routing technology, and the routing algorithm. The system attributes affect the decision making on the granularity and mapping across different systems. For instance, lower communication latency

[6]The logarithmic scale may create an illusion that the rates are the same and only an offset exists, however this is really not the case. Had a linear scale been used, the fixed grain size algorithms would exhibit a drastically larger slope than the varying grain size algorithm.

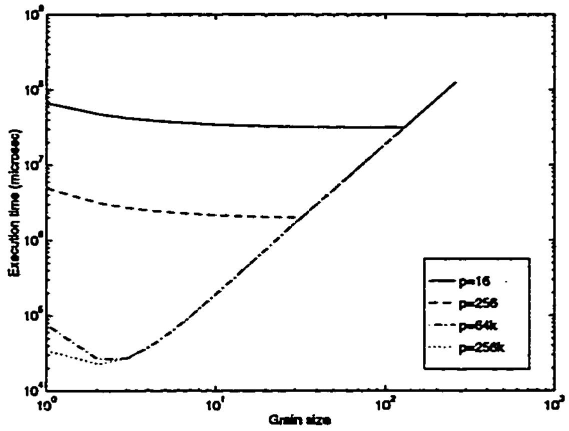


Figure 33: Optimal grain size selection

of one system may allow one library routine to execute with a higher degree of parallelism than a system that has a high communication latency. In porting a parallel library from one system to another, these attributes are changed in the library prior to the one-time compilation of the library for that system.

Now, we develop an expression (in terms of the grain size and the system size) for the execution time of the torus based matrix multiplication. We will evaluate this expression for different problem sizes and system sizes under varying grain size. The processor scheduling cost has been neglected in favor of the algorithms based on smaller granularity,

$$T_{par} = \begin{cases} \frac{n}{g}\left[\frac{gn^2}{p}t_c + \frac{2n}{g\sqrt{p}}t_s + \left(\frac{2n^2}{p} - \frac{2ng}{\sqrt{p}}\right)t_m\right] & \text{if } \frac{n}{g} > \sqrt{p} \\ \frac{n}{g}\left(t_c g^3 + 2t_s\right) & \text{Otherwise.} \end{cases} \tag{12}$$

The first function is always decreasing with increasing grain size and is only defined up to $\frac{n}{\sqrt{p}}$. The second function , however, has a global minimum. Differentiating the second segment of T_{par} with respect to g yields,

$$\frac{\partial T_{par}}{\partial t} = 2nt_c g^2 - \frac{2nt_s}{g^2}.$$

Solving $\frac{\partial T_{par}}{\partial t} = 0$ results in $g_{min} = \left(\frac{ts}{tc}\right)^{1/3}$, where g_{min} is the grain size which minimizes the second segment of the T_{par}. Since these functions are only valid over specific values of g, one can find the minimum execution time by examining all permissible values near g_{min}. The best grain size obtained from the second segment can then be compared with $\frac{n}{\sqrt{p}}$. If g_{min} is larger than $\frac{n}{\sqrt{p}}$, g_{min} is the grain size which minimizes the objective function, otherwise $\frac{n}{\sqrt{p}}$ is the optimal grain size.

Figure 33 shows the execution time of the torus based matrix multiplication for a matrix size of 512×512, i.e $n = 512$. When $p = 16$, the minimum execution time occurs at $g = 128$ which implies a virtual architecture of 4×4. Each physical processor will then emulate one virtual processor. Increasing or decreasing the grain size will deteriorate performance. When $p = 256$, a value of $g = 32$ results in the lowest execution time. The virtual architecture is of size 16×16 for this value of g. For this case, each physical processor also emulates a single virtual processor. Using

$64k$ ($k = 1024$) processors to perform the 512×512 multiplication, a value of $g = 2$ yields the lowest execution time. This value can also be obtained through differentiation, i.e. $(\frac{30}{3.7})^{1/3} \approx 2$. This implies a virtual architecture of size $64k$ processors. Finally when $256k$ processors are used, decreasing the granularity to take advantage of the additional parallelism increases the execution time of the program. Unlike the first three cases, only a fraction of the processors (25%) are used to perform the computation, which implies a large drop in speedup and efficiency. If all the processors are used to perform the computation, the execution time will nearly double. As new techniques are developed which reduce the communication delay between processors, using this scheme, larger number of processors will be used to perform the same computation. Porting a parallel library, based on this scheme, to another system will have a different g_{min}. Updating the values of t_s, t_c, and t_m for the new system and minimization of T_{par} will result in the best granularity for that system. This value of granularity determines the degree of parallelism for the execution of the library routine.

This analysis implies that for a parallel library routine, the execution time function must be developed and integrated in the library. The execution time is a function of the problem size, granularity, mapping, and the physical system attributes. At runtime, the grain size is then selected so that the execution time of the routine is minimized. This value is not necessarily the grain size that scales down the virtual architecture size to that of the physical system size. A fraction of processors may have to sit idle and their participation in the computation would increase the execution time of the library routine due to an increase in the communication overhead. In multicomputers with fixed system topology, such as meshes, the mapping of the virtual processors can be considered in the execution time function. On reconfigurable systems, the execution time function can be derived based on the virtual topology. Overlooking the effects of mapping introduces slight inaccuracy in the execution time function. This inaccuracy is due to the missing factors that account for communication distance and the link contention. The mapping function can then be considered in the delayed mapping phase, to be shortly discussed.

Once the size of the scaled virtual architecture is determined, its mapping to the physical processor determines the communication pattern of the program on the underlying physical architecture. This mapping affects the communication overhead of the program by determining the distance between the communicating nodes. For large messages, the distance increases the latency of communication by amounts that are significant with respect to the startup cost of communication. In this section we examine the impact of the distance between communicating nodes and the message size on the communication overhead of the program for both packet switching and wormhole routing schemes. This study is strongly relevant to our library design where scaling block matrix algorithms increases message sizes. Even though the messages are broken up into packets in both schemes, without loss of generality we consider the messages as a single entity as this is the case when very large packet sizes are used.

Let's consider the time delay introduced, as a result of a conflict on a physical channel in terms of the message size in packet switching and wormhole routing. Figure 34 shows that additional latency introduced as a result of a channel conflict between messages of a given size. The value of $ts = 30$ μsec and $tw = 0.023$ μsec were used in the diagram. The dotted line denotes the latency of

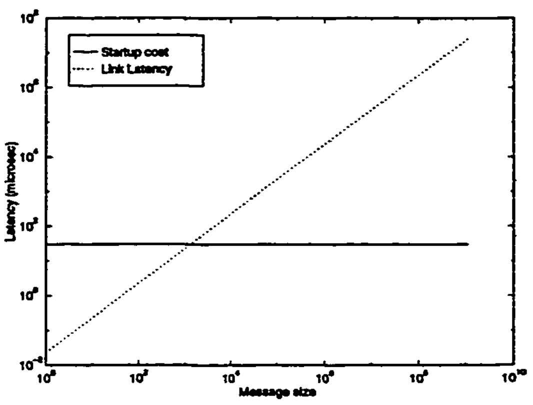


Figure 34: Link latency as a function of the message size

the link and the solid line denotes the startup cost. At near 1024 word message size, the latency due to contention on a physical link is approximately equal to a startup cost. This implies that as a result of a conflict, on average, each message will suffer an additional latency of $\frac{t_s}{2}$ due to this conflict. This applies both to packet switching and wormhole routing. The accumulation of this additional latency over all communication paths and over the course of execution of the program will introduce serious impact on the communication overhead of the program[7]. The overhead of a program due to contention can be captured if the communication behavior of the program is studied using the mapping function in mind. Once a suitable mapping function is determined, it can be built into the library routine. The contention must be considered under the worst case conditions. Time of communication being a factor in the contention makes development of an exact model very difficult. Therefore if two communication paths intersect, it must be assumed that contention is inevitable unless there is a synchronization point between the two communication operations. Most systems have bidirectional channels which must be considered while examining the severity of contention. An example mapping of an $\sqrt{p} \times \sqrt{p}$ torus to a $\sqrt{p} \times \sqrt{p}$ mesh that requires some messages to go through the same physical link in our matrix multiplication example is:

$$\mathcal{M}_1(i,j) = \begin{cases} (\frac{i}{2}, \frac{j}{2}) & \text{if even } i \text{ and even } j \\ (\frac{i}{2}, \frac{\sqrt{p}}{2} + \lfloor \frac{j}{2} \rfloor) & \text{if even } i \text{ and odd } j \\ (\frac{\sqrt{p}}{2} + \lfloor \frac{i}{2} \rfloor, \frac{\sqrt{p}}{2} + \lfloor \frac{j}{2} \rfloor) & \text{if odd } i \text{ and odd } j \\ (\frac{\sqrt{p}}{2} + \lfloor \frac{i}{2} \rfloor, \frac{j}{2}) & \text{if odd } i \text{ and even } j, \end{cases} \tag{13}$$

for even $\sqrt{p}$. This mapping creates contention along every row and column of the matrix in both directions. Contention is higher in the middle of the rows and columns than around the edges and the level of contention increases with the system size. We will later show, through our simulation results, that this mapping results in significantly higher execution time than dilation cost 2 mapping in equation 15 or identity mapping.

Another interesting example is mapping of a hypercube to a line. Consider a d dimensional hypercube, i.e. $p = 2^d$. Assuming in each phase i, processors with a 0 in the ith position communicate

[7]In iterative algorithms the execution consists of repetitive computation and communication steps. Cumulative overhead is calculated over all the iterations and all the processors.

with their neighbor along the ith dimension. For simplicity, let's consider $d = 3$. When identity mapping function is used, the following list shows the contention during each phase:

- Phase 1 - No contention.
- Phase 2 - Contention on links (1,2) and (5,6) with one conflict in each direction.
- Phase 3 - Contention on links (1,2), (2,3), (3,4), (4,5), (5,6) with 2, 3, 4, 3, 2 conflicts, respectively, in each direction.

On the other hand, if the following mapping function is used:

$$\mathcal{M}(i) = \begin{cases} \frac{i}{2} & \text{if even } i \\ p - \lfloor \frac{i}{2} \rfloor & \text{Otherwise,} \end{cases} \tag{14}$$

The contention during each phase will occur on the following links:

- Phase 1 - Contention on links (1,2), (2,3), (3,4), (4,5), and (5,6) with 2, 3, 4, 3, 2 conflicts, respectively, in each direction.
- Phase 2 - No contention.
- Phase 3 - Contention on links (1,2) and (5,6) with one conflict in each direction.

Clearly, one mapping is not preferred over another. If a hypercube physical system is used, the execution would be contention free.

One other effect of mapping on the performance of a program is through creation of bottleneck in the communication of the program during each phase. When the program has inherent synchronization using communication, the bottleneck in communication slows down each iteration as all the processors require data from one another during each iteration. Applications such as torus based matrix multiplication, Jacobi relaxation, and FFT have this characteristic. We show the impact of mapping in the communication overhead of a program using our torus based matrix multiply example. The scaled down architecture is first considered under the systematic mapping with dilation cost of 2 borrowed from [58]. If p is a perfect square, for even $d = \sqrt{p}$, the mapping function, $\mathcal{M}_2$, can be stated as:

$$\mathcal{M}_2(i) = \begin{cases} (2i, 2j) & \text{if } i < \frac{d}{2} \text{ and } j < \frac{d}{2} \\ (2i, 2d - 2j - 1) & \text{if } i < \frac{d}{2} \text{ and } j \geq \frac{d}{2} \\ (2d - 2i - 1, 2j) & \text{if } i \geq \frac{d}{2} \text{ and } j < \frac{d}{2} \\ (2d - 2i - 1, 2d - 2j - 1) & \text{if } i \geq \frac{d}{2} \text{ and } j \geq \frac{d}{2}, \end{cases} \tag{15}$$

where the row and column numbers start from 0. The restriction of even $\sqrt{p}$ can be easily lifted by changes to the above function. The cumulative overhead function is developed in terms of the system size and is compared with the same program under a naive mapping such as identity. Using the overhead function, we study the degradation in performance due to identity mapping when system size varies.

Consider the torus based matrix multiplication of virtual architecture size $\sqrt{p} \times \sqrt{p}$ running on a mesh of the same size[8]. Using the identity mapping, the neighboring relationship among all the virtual processors is preserved except for the processors at the boundary. Since the torus based matrix multiply has inherent synchronization through communication after each iteration, the cumulative overhead of the mapped program consists of $\sqrt{p}$ iterations each consisting of communication of two $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ partitions to a node $\sqrt{p}-1$ hops away. Using packet switching, the communication overhead of the program, $T_o^{sf,\sqrt{p}-1}$, can be expressed as:

$$T_o^{sf,\sqrt{p}-1} = 2\sqrt{p}\left(t_s + \frac{n^2}{p}(\sqrt{p}-1)t_w\right),$$

where $T_o^{r,d}$ denotes the communication overhead of the program using the routing scheme r under a mapping with dilation cost of d. It is important to note that contention has not been considered in the calculation of the overhead function. The effect of contention has been separately discussed. The overlapping of the two communication operations is not considered in the preceding equation either. For the two send operations to be overlapped, the messages must be large enough so that the transmit time of the message exceeds the startup cost. Using the mapping function in equation 15 would yield the following overhead function,

$$T_o^{sf,2} = 2\sqrt{p}\left(t_s + \frac{2n^2}{p}t_w\right).$$

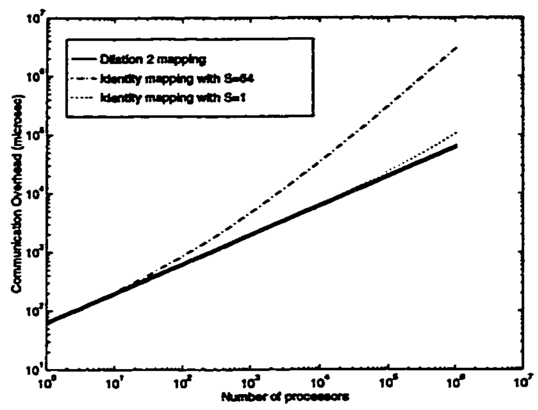
Considering the overhead functions when the system uses wormhole routing, the following expressions are obtained,

$$T_o^{wh,\sqrt{p}-1} = 2\sqrt{p}\left(t_s + (\sqrt{p}-1+\frac{n^2}{p})t_w\right)$$
$$T_o^{wh,2} = 2\sqrt{p}\left(t_s + (2+\frac{n^2}{p})t_w\right).$$

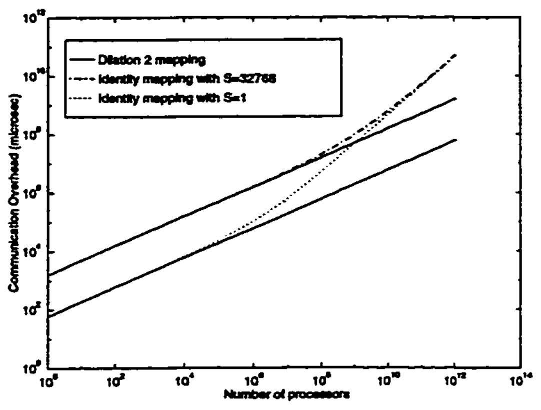
The value $t_s = 30$ μsec was used to derive the data presented. A word is assumed 4 bytes which yields a value of $t_w = 0.023$ μsec. Value of t_h is very small and has been omitted in the computation. Figure 35(a) demonstrates that for a system as small as 32 x 32 processors using packet switching, the communication overhead of the matrix multiplication differs by nearly 50% between the identity mapping and the systematic mapping with dilation cost of 2 for message size $S = 1$. For message size of $S = 64$ significant difference in the total overhead is observed on systems as small as 16 processors. Figure 35(b) demonstrates the same results for a system based on wormhole routing. At near 1024 x 1024 processors a 50% difference in the total overhead is observed for $S = 1$. This difference widens as the number of processors increases. Unlike packet switching networks, for larger messages this gap narrows and is observed at a much larger system size. Larger messages are more attractive in wormhole routing because the communication latency becomes independent of the distance.

The effects of mapping on the communication overhead are tightly coupled with the communication pattern of the virtual architecture and the underlying physical system. Therefore, this effect

[8]Considering the same size virtual and physical architecture is pertinent to our parallel libraries, since we initially close the size gap through runtime grain size adjustment.



(a) Packet switching



(b) Wormhole routing

Figure 35: Comparison of the cumulative overhead between two different mapping functions for n=1024

must be studied on a case by case basis. On the other hand, the communication pattern of most virtual architecture programs matches their virtual topology. Therefore, the analysis just performed on the mapping of a virtual torus to a physical mesh pertains to algorithms and systems that are based on these virtual and physical topologies, respectively. Since different mapping functions result in different contention levels, for each virtual architecture routine, its contention level must be examined on the target system after the mapping. The mapping function that results in the least contention in the worst case, must be selected and embedded in the library. For example, the dilation 2 mapping presented, results in no contention and does not create communication bottleneck which increases the overhead as a function of the system size. This mapping function has a dilation cost of 2 for any system size.

Several conclusions can be made from the analytical results presented. To begin with, highly scalable algorithms are more desirable for parallel library design since they can maintain a desirable efficiency as the system size increases. Through the examples presented in this section, it has been shown that scalability of algorithms is highly sensitive to the granularity of the computation. Mapping of the virtual processors affects the execution time of a program by increasing its total overhead through contention and communication bottleneck. Therefore, delayed grain size adjustment and mapping are critical to the performance of a library routine.

3.4 Experimental Results and Performance Evaluation

We have supported our library design by implementing a source to source transformer which translates sequential programs with library calls to their equivalent SPMD programs, a compiler which generates intermediate code from the SPMD programs, and a multicomputer simulator to simulate the execution of the translated SPMD programs and provide performance results. The data that will be presented in this subsection is meant to show that our library design delivers much higher

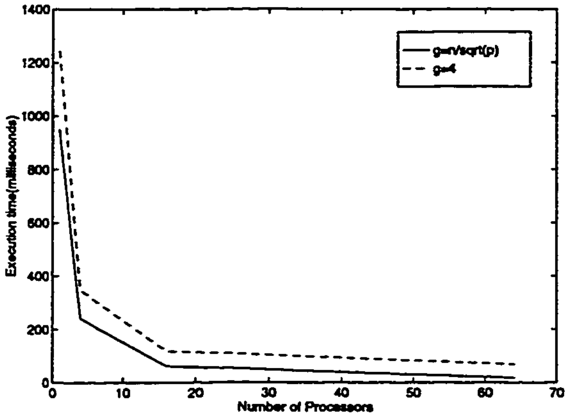
$p \rightarrow$ $g \downarrow$	1×1	2×2	4×4	8×8
$\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$	949.50 ms 1.00 1.00	240.60 ms 3.95 0.99	61.30 ms 15.49 0.97	16.8 ms 56.52 0.88
4×4	1243.30 ms 0.76 0.76	347.20 ms 2.73 0.68	117.60 ms 8.07 0.50	68.10 13.94 0.22

Table 5: Simulation data: Execution, speedup, and efficiency for different problem and system sizes for torus based matrix multiplication.

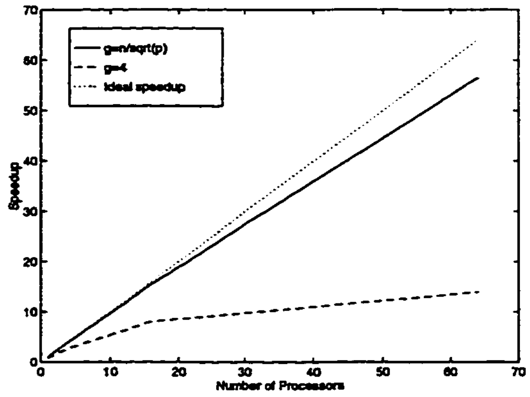
speedup and efficiency than ScaLAPACK. It is also meant to show that execution time of a program is a function of the system attributes and must be evaluated at runtime. Depending on the values of the attributes, the system size which delivers the least execution time can be calculated and used by the library routine. The used partition of physical processors may be part of the allocated physical system and may differ from one parallel library phase to another. Porting a library from one system to another requires updating of these parameters so that the library routine can harness the full power of the new system. We would also like to show that mapping functions that reduce contention level in the physical architecture significantly reduce the execution time of the program.

The following data is collected from our library implementation of the matrix multiply routine and the Cooley-Tukey Fast Fourier Transform algorithm [33]. The parameters of the simulation were set to startup cost of $t_s = 30$ μsec, link delay of $t_w = 0.023$ μsec, and node delay of $t_h = 0.01$ μsec. Physical meshes were used to conduct the simulations. We examined the speedup and efficiency of these applications by first scaling down the virtual architecture to the physical system size. We then collected data for a fixed grain size of 4. Table 5 shows the values of execution time, speedup, and efficiency of matrix multiplication for two 64 × 64 matrices. The numbers in each entry in the table correspond to the values of execution time, speedup, and efficiency in that order from top to bottom. Figure 36 illustrates the same data in a pictorial form. The data shows that the speedup and efficiency from our library routine exceeds those of ScaLAPACK. In ScaLAPACK the granularity of computation is fixed prior to the call. The user program defines the granularity and the size of the virtual architecture. In contrast, in our library routines the granularity is decided by the routine itself and the granularity of computation is transparent to the programmer. The grain size of 4 has been selected for comparison since it is claimed by ScaLAPACK to provide the best average performance.

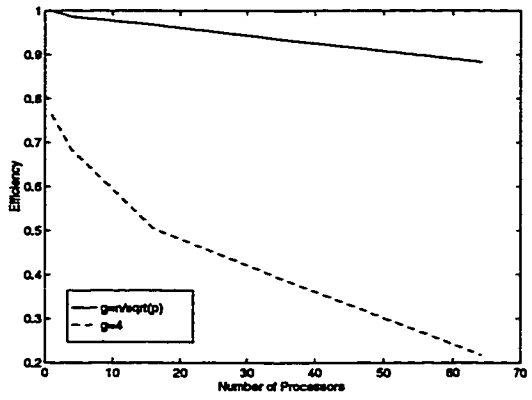
The Cooley-Tukey algorithm for an n-point single dimensional FFT was implemented using our library design scheme. Assume $n = 2^r$ and number of physical processors $p = 2^d$, for some integers r and d. A d dimensional hypercube is used to solve the problem ($d \leq r$) and each processor is allocated n/p elements of the array. To reduce the communication overhead of the routine, the ith element of the input and the output vector are mapped onto processor number $(b_{d-1} \ldots b_0)$, where $(b_{r-1} \ldots b_0)$ is the binary representation of i. With this mapping, processors do not need to



(a) Execution time



(b) Speedup



(c) Efficiency

Figure 36: Comparison of $g = 4$ and $g = \frac{n}{\sqrt{p}}$. a) Execution time b) Speedup c) Efficiency

$p \rightarrow$ $g \downarrow$	4	16	64
$\frac{n}{p}$	23.20 ms 3.68 0.92	5.50 ms 15.53 0.97	1.50 ms 56.93 0.89
4	26.00 ms 3.28 0.82	8.50 ms 10.05 0.63	3.80 ms 22.47 0.35

Table 6: Simulation data: Execution, speedup, and efficiency for different problem and system sizes for Cooley-Tukey algorithm.

$g \rightarrow$ $n \downarrow$	1x1	2x2	4x4	8x8	16x16	32x32	64x64
$n = 64\times64$	720.50 ms	301.50 ms	117.60 ms	98.40 ms	61.30 ms	240.70 ms	949.50 ms
$n = 16\times16$	18.80 ms	6.40 ms	1.50 ms	4.31 ms	16.10 ms	-	-
$n = 4\times4$	0.40 ms	0.30 ms	-	-	-	-	-

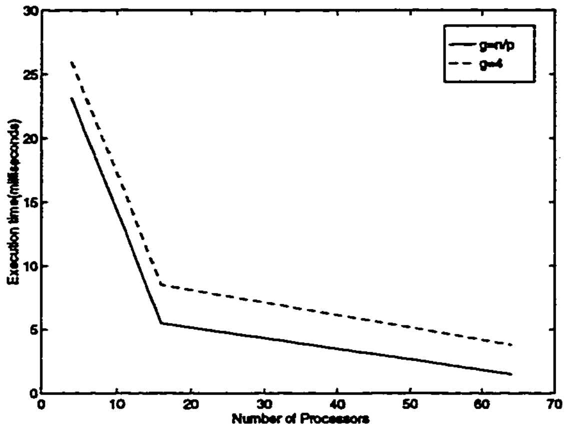
Table 7: Simulation data: Execution time as function of grain size.

communicate in the first $r - d$ iterations of the algorithm because all the elements needed by each processor are locally available to it. In the remaining d iterations processors communicate with one another and in the lth ($0 \leq l < d$), all the n/p values required by a processor are available to it from a single processor (one which is its neighbor along the lth dimension). The following data was obtained from the library implementation of the stated FFT algorithm. Table 6 shows the values of execution time, speedup, and efficiency for a 512 point FFT. The sequential execution of the routine was $T_{seq} = 85.4$ msecs. The numbers in each entry in the table correspond to the values of execution time, speedup, and efficiency in that order from top to bottom. Figure 37 illustrates the same data in a pictorial form.

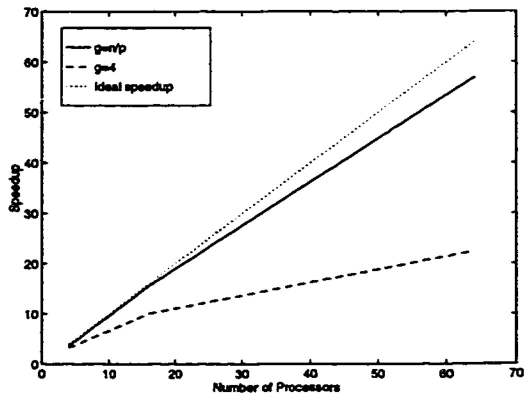
In our second set of experiments, execution times from different problem and system sizes were collected from our library implementation of the parallel matrix multiplication using various granularity values and a 16 processor square mesh. The values of grain size which minimize execution time for each instance have been obtained from the execution time function:

$$T_{par} = \begin{cases} \frac{n}{g}\left[\frac{3.7gn^2}{p} + \frac{60n}{g\sqrt{p}} + 0.04\left(\frac{2n^2}{p} - \frac{2ng}{\sqrt{p}}\right)\right] & \text{if } \frac{n}{g} > \sqrt{p} \\ \frac{n}{g}\left(3.7g^3 + 60\right) & \text{Otherwise,} \end{cases}$$

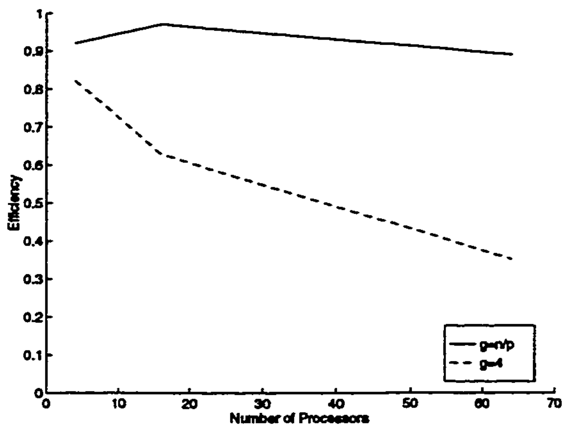
which is derived in Equation 12 on Page 84 by replacing the t_s, t_c, and t_m for the values used in the simulation. The value of $t_c = 3.7$ μsec was obtained by conducting several sequential matrix multiplications with different problem sizes and calculating the average time for the statement in the innermost loop of the sequential matrix multiplication. Table 7 shows the values of execution time as a function of the grain size for three different problem instances. Figure 38 demonstrates the same results pictorially. Following the procedure on Page 84, one can find the best grain size



(a) Execution time



(b) Speedup



(c) Efficiency

Figure 37: Comparison of the FFT algorithm for $g = 4$ and $g = \frac{n}{p}$. a) Execution time b) Speedup c) Efficiency

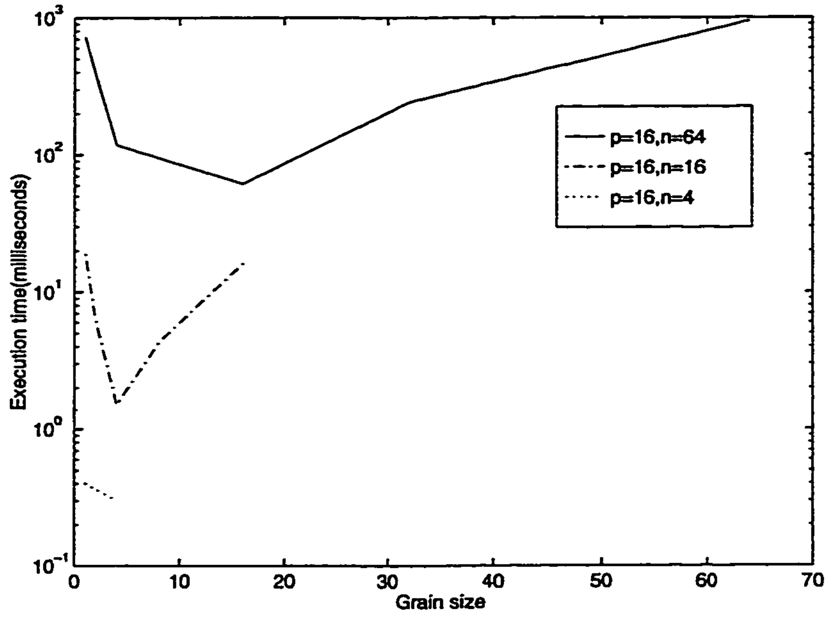


Figure 38: Runtime Granularity Adjustment

for each problem instance and confirm the values in the table. The first two problem instances demonstrate the case where the whole system is utilized for the execution and a larger grain size is used while taking advantage of the full parallelism of the physical system. The third problem instance demonstrates the case where only part of the system is used to perform the computation(4 out of 16 processors). Using all 16 processors and running the program at $g = 1$ will increase the execution time.

We have also developed approximate execution time functions for the matrix addition, Jacobi relaxation, and Fast Fourier Transform. These numerical applications have been developed using our library design. The pseudo code for $C = A + B$, where A, B, and C are global $n \times n$ matrices, is presented in the following algorithm. This algorithm represents the code executed by all the virtual processors. The symbols a, b, and c refer to the distributed data blocks of A, B, and C, respectively. Assume the logical architecture is an $\frac{n}{g} \times \frac{n}{g}$ mesh, g and n are powers of 2, and g divides n.

```
$a \leftarrow$ Initial $g \times g$ partition of $A$
$b \leftarrow$ Initial $g \times g$ partition of $B$
$c \leftarrow a + b$
Send computed $c$ back to host
```

Matrix addition does not require any communication. Once the matrices are distributed across the virtual processors, each virtual processor performs a local matrix addition of its partitions and sends the results back to the host. The approximate execution time function can be described as follows:

$$T_{par} = \begin{cases} \frac{n^2}{p} t_c & \text{if } \frac{n}{g} > \sqrt{p} \\ \frac{n^2}{g^2} t_c & \text{Otherwise.} \end{cases} \tag{16}$$

Values of n, p, g, and t_c are the problem size, the physical system size, the grain size, and the average time to perform addition of a single element, respectively. It is evident that matrix addition uses all the available physical processors regardless of the problem and system size. This is because matrix addition does not have any communication overhead.

The pseudo code for Jacobi relaxation of a global $n \times n$ matrix, A, with tolerance value *tol* is shown below. The symbol a refers to the distributed data blocks of A. Assume the logical architecture is an $\frac{n}{g} \times \frac{n}{g}$ mesh, g and n are powers of 2, and g divides n.

```
$a \leftarrow$ Initial $g \times g$ partition of $A$
Host broadcasts tolerance value, $tol$, to all the virtual processors
repeat
  Send local partition $a$ to the neighbors (Care must be taken at the mesh boundaries)
  Receive partitions from the neighbors (Care must be taken at the mesh boundaries)
  Perform relaxation on the local partition (Care must be taken at the partition boundaries)
  $maxdiff \leftarrow$ Maximum difference of the local relaxation
  $globMaxdiff \leftarrow$ Global combine of $maxdiff$ over all virtual processors
until $globMaxdiff \leq tol$
Send computed $a$ back to host
```

The variable *maxdiff* is used to hold the maximum difference of the last local Jacobi relaxation in each virtual processor. This value is globally combined over all the virtual processors. Each virtual processor will then have the global maximum in the variable *globMaxdiff*. The approximate execution time function for Jacobi relaxation was developed for a single iteration since the total number of iterations is not known and it depends on the tolerance value. Since the communication requirements of the application during all iterations are similar and there is global synchronization after each iteration, minimization of a single iteration ensures minimization of the whole execution time. The function can be described as follows:

$$T_{par} = \begin{cases} \frac{n^2}{p} t_c + \frac{4n}{\sqrt{p}} t_s & \text{if } \frac{n}{g} > \sqrt{p} \\ g^2 t_c + 4t_s & \text{Otherwise.} \end{cases} \tag{17}$$

The value of t_c is the average time to perform a single point relaxation.

The pseudo code for Fast Fourier Transform on a hypercube of a global vector A of size n is shown below. The symbol a refers to the distributed data blocks of A. Assume the logical architecture is a $\frac{n}{g}$ processor hypercube, g and n are powers of 2, and g divides n.

```
$a \leftarrow$ Initial partition (of size $g$) of vector $A$
Perform local FFT of the partition of size $g$
for $i = \log \frac{n}{g} - 1$ downto 0 do
  Send local partition to the neighbor along the $i$th dimension
  Receive partition from the neighbor along the $i$th dimension
  Update values of the local partition using the neighbor's partition
end for
Send computed $a$ back to host
```

The approximate execution time function for Fast Fourier Transform is:

$$T_{par} = \begin{cases} \frac{n}{p}t_c + \frac{nt_s}{gp}\log\frac{n}{g} & \text{if } \frac{n}{g} > \sqrt{p} \\ \frac{n}{g}t_c + t_s\log\frac{n}{g} & \text{Otherwise.} \end{cases} \tag{18}$$

The value of t_c is the time to perform a single point Fast Fourier Transform. Each virtual processor requires number of communications equal to the logarithm of the total number of virtual processors.

The third set of experiments exhibits the effects of mapping on the execution time. There are three sets of data presented to exhibit the effects of mapping. The first one is targeted to show the impact of poor mapping on the execution time of a parallel library routine in packet switching. The second and third experiments show the impacts of poor mapping on the execution time in wormhole networks. Mapping functions have been selected to show two effects of mapping on the performance of the parallel program. The first effect is contention and the second one is the communication bottleneck. In algorithms which consist of several iterations of computation and communication and during each iteration the data moves in form of shift operations, a single link with longer delay than others can create a bottleneck in the shift operation. This additional delay due to the bottleneck can accumulate over all iterations of the algorithm and degrade the performance of the library routine. This bottleneck depends on the dilation cost of the mapping. For example an identity mapping of a torus to a mesh of the same size has a larger dilation cost than the dilation 2 mapping in Equation 15 on Page 87. Therefore, the identity mapping creates a bottleneck that becomes visible on larger systems. The torus based matrix multiplication routine will be run using the three different mapping functions and routing technologies. Parameters of the simulations were as follows, unless stated otherwise:

Packet size = 64 Bytes
Flit size = 1 Byte
Number of lanes = 2
Clock cycle = 20 ns
Link bandwidth = 0.005 μsec per byte
One word = 4 Bytes

For each experiment fixed messages of 32×32 bytes were used. Therefore, each virtual processor was assigned such partition. The functions used for conducting the mapping experiments are all one-to-one. For example, an 8×8 virtual torus was set up to run on an 8×8 physical mesh with each virtual processor holding a 32×32 partition. The problem instance size in this case would be 256×256. All the experiments were conducted on physical meshes of processors. In order to reduce the simulation time, the simulator was changed so that artificial messages of desired size (32×32) were deposited in the network, but the amount of computation performed by each virtual processor was a single scalar operation (rather than a full 32×32 matrix multiplication during each iteration). This may affect the overlapping computation and communication, but reduced the simulation time drastically. Therefore the execution times collected are not actual execution time of the stated problem size. The collected times are in effect the communication overhead, neglecting the very small computation performed.

$p \rightarrow$ *mapping* $\downarrow$	4×4	8×8	16×16
$\mathcal{M}_1$	1093	4618	20564
$\mathcal{M}_2$	945	5679	35603
$\mathcal{M}_3$	1228	7561	44418

Table 8: Simulation data: Contention free *vs.* contended mapping in packet switching.

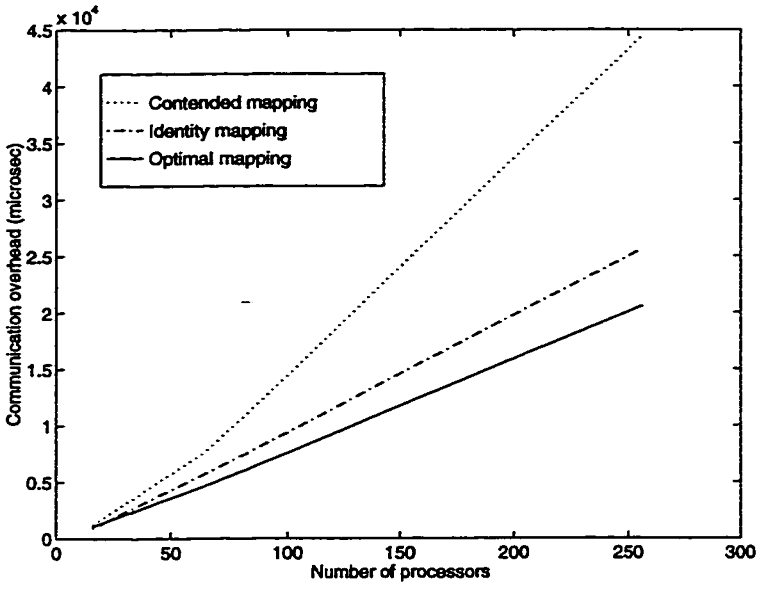


Figure 39: Effects of contention in packet switching

The one-to-one mapping functions $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ are defined by Equations 15 on Page 87, identity, and Equation 13 on Page 86, respectively. $\mathcal{M}_1$ is the dilation 2 contention free optimal mapping. $\mathcal{M}_2$ is the identity mapping which will exhibit the effect of the communication bottleneck in the performance of the routine. $\mathcal{M}_3$ is the contended mapping function which is meant to show degradation in performance due to link contention. Table 8 demonstrates the difference in execution time between $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ in packet switching networks. The values have been plotted in figure 39. The dilation 2 mapping has the lowest rate of increase in the communication overhead with respect to the system size among the three. The identity mapping has a slightly higher rate due to the communication operations between the boundary processors which must travel across the whole dimension. The contended mapping function has the largest increase in the communication cost. The difference in the communication overhead of the three mapping functions is noticeable on a system as small as 36 processors.

Table 9 demonstrates the difference in the execution time between the contention free mapping, $\mathcal{M}_1$, and the contended mapping, $\mathcal{M}_2$ in wormhole routing. The values have been plotted in figure 40. A message size of 1024 was used to conduct the experiments. Although the difference is not as large as that of wormhole routing, this experiment also exhibits degradation in performance due to

$p \rightarrow$ *mapping* $\downarrow$	4x4	8x8	16x16
$\mathcal{M}_1$	404	913	4128
$\mathcal{M}_2$	405	1056	4821
%diff	0.2%	16%	16.8%

Table 9: Simulation data: Contention free *vs.* contended mapping in wormhole routing.

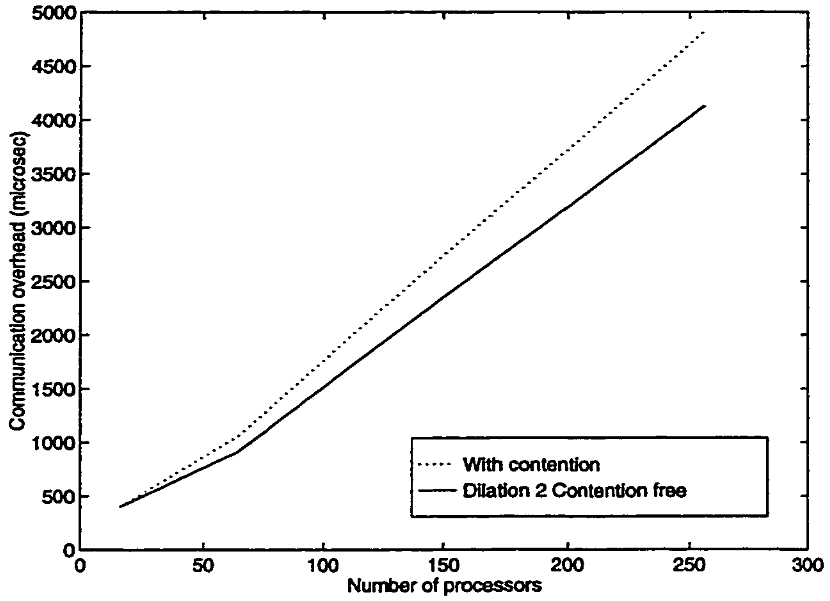


Figure 40: Effects of contention in wormhole routing

contention when poor mapping functions are used. This difference is much smaller and occurs at larger system sizes.

Table 10 demonstrates the difference in execution time, in wormhole networks, between the mapping functions, $\mathcal{M}_1$, which has a fixed size bottleneck regardless of the system size and $\mathcal{M}_2$, which introduces communication bottleneck that elongates with the system size. The overhead function of $\mathcal{M}_1$ grows slower than that of the $\mathcal{M}_2$. For the data above a flit size of 1, packet size of 16, and actual message size of 1 was used. The results have also been presented in figure 41. Large messages are very attractive in wormhole networks because distance becomes insensible in the communication latency. The effect of communication bottleneck is insensible when large messages are used. Small messages, on the other hand, cause the routine execution to be more sensitive to the communication bottleneck. The communication bottleneck effects can also be observed for large messages with very large systems, for which our simulator would run into memory limitation problem, or the simulation time would be intolerably large.

We will now examine the overhead associated with the parallel library design proposed in this chapter. A parallel phase starts by broadcasting the size of the problem instance to all the physical processors. Upon receiving this value, all physical processors, simultaneously, determine the contents

$p \rightarrow$ *mapping* $\downarrow$	4x4	8x8	16x16	32x32
$\mathcal{M}_2$	404	610	1620	6023
$\mathcal{M}_1$	404	610	1619	5810
%diff	0%	0%	0.1%	3.7%

Table 10: Simulation data: Effects of communication bottleneck in wormhole routing.

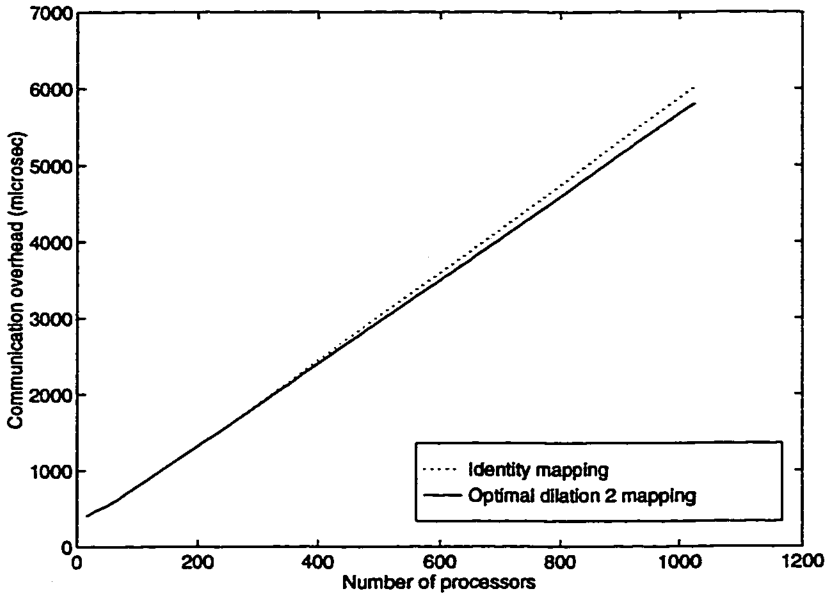


Figure 41: Effects of communication bottleneck in wormhole routing

of the handles. For each parallel phase, there is one handle per physical processor. The overhead of determining the contents of the handle is very minimal since it is done in parallel. When the contraction and mapping are performed, each physical processor scans its local mapping table (same copy on all physical processors) and creates its local threads. A list of previous parallel phases is maintained in the global memory on all the physical processors. This list is visited for possible data redistribution. If distributed data is alive on the system, it is redistributed using the source and the destination handles. Thereafter, the processor with I/O capability distributes the data that was not redistributed. The distribution of data requires change of storage through an additional copy of the data so that the blocks to be distributed would reside in contiguous memory locations. The virtual processors perform the computation once they receive their initial data. Upon completion of the computation, the result is sent back to the processor that originated the parallel phase.

Table 11 show the results of the simulations performed to examine the overhead of our design. The parameters of the simulations were the same as those of the experiments conducted on Page 90. Matrix sizes of 64x64 were used. The execution time of the library call was measured as a stand-alone routine, with inclusion of the initial data distribution, and with the handle initialization overhead. The data clearly states that the use of handles and their initialization introduce very little overhead in

$p \rightarrow$ Overhead included $\downarrow$	2×2	4×4	8×8
None (Library call alone)	240.6 ms	61.3 ms	16.8 ms
Handle setup	241.1 ms	62.5 ms	20.7 ms
Handle setup + Initial data distribution	302.1 ms	124.5 ms	85.20 ms

Table 11: Simulation data: Execution time of the library routine alone, with handle initialization, and with initial data distribution for torus based matrix multiplication.

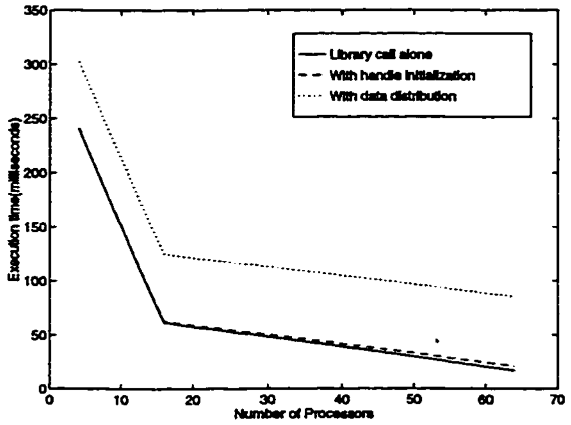
our design. However, initial data distribution overhead is intolerable. For instance, in the presented data for an 8×8 mesh of physical processors, the execution time of the matrix multiplication has increased by a factor of nearly 400% because of the initial data distribution of the two matrices of size 64×64. This overhead consists of the blocking of the data and sending them to the virtual processors. Figure 42 illustrates the same data in a pictorial form. In our design, handles are kept in the global memory to support data redistribution. This very little overhead of handle maintenance is well justified considering the objective of reducing the initial data distribution overhead. Our data redistribution support is described in Chapter 4.

3.5 Impacts on the Library User, Designer, and the Compiler

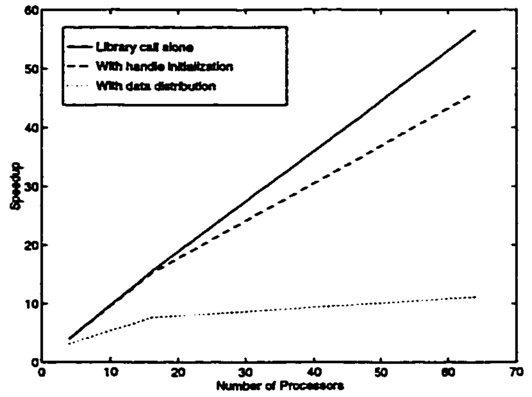
In this section we unfold the design impingements on the library routine user, library routine designer, and the compiler. The programmers' task in using a library routine is quite simple. The system is designed to accept a conventional sequential program that contains parallel library calls. The programmer must first use the source to source transformer to convert the program to an SPMD form. Then, the threading library, virtual and low level communication libraries, the mapping library, and the redistribution module are linked in to create an executable image. Although these libraries are linked in by the user, their internals are completely transparent.

From the view point of the library designer there are various details that must be dealt with when designing the message passing library routines. Each component of the library routine must be carefully coded and tested prior to its usage. The implementation of the library routine must maintain the semantics of the library call and its parameter passing. Various optimizations can be done in the library routines, some of which are mentioned in this thesis. The design of the parallel library routines is monolithic and a major part of the effort is in designing and understanding the first virtual architecture parallel library routine.

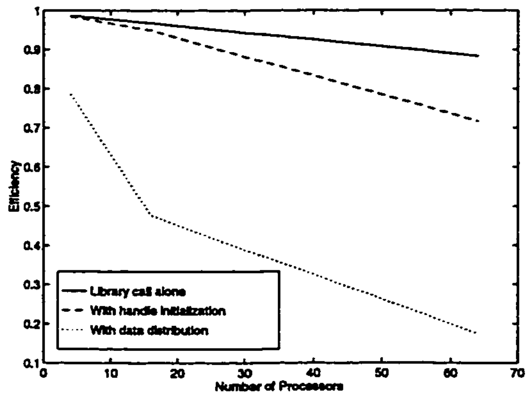
Examination of different parallel algorithms is the first step in setting up a parallel library routine. Obviously, the parallel algorithm with lowest time complexity is the most desirable. Among the selected algorithms, the most suitable ones for library implementation are the ones that can be described in block matrix form. This, of course, holds for the majority of algorithms that operate on vectors and matrices. The next step is to develop a model for the cumulative communication overhead. The cumulative overhead is a function of the communication pattern of the algorithm, system size, possibly problem size, and the system attributes such as components of the communication



(a) Execution time



(b) Speedup



(c) Efficiency

Figure 42: Comparison of the execution time, speedup, and performance of matrix multiplication library routine alone, with handle initialization overhead, and with initial data distribution overhead.

latency. Once closed form expressions are developed, a scalability metric, such as the isoefficiency metric of scalability used in this thesis, can be used to examine the scalability of each algorithm. If all the algorithms have overhead functions with different time complexities, the comparison is simple. However, if more than one algorithm have the lowest time complexity, the one with a lower constant is preferable. Development of the overhead functions based on the virtual architecture is less troublesome than if the physical architecture is considered. The overhead function is dependent on the mapping of the processors to the physical system. If the scalability analysis does not reveal a difference in the overhead function of the two algorithms, the user may consider the algorithms under specific mappings to the physical system in order to decide which algorithm is better suited for parallel library implementation.

Following selection of a scalable algorithm, an execution time model, which captures the computation and communication behavior of the program, must be developed. Factors such as computation and communication overlap and mapping of the processors are difficult to capture in the model. The former depends on the grain size of the computation which is not known statically, and the latter depends on the target system size and topology. The developed model is used for determining an optimal grain size for the execution. Even though the mapping function is not captured in the execution time model, it is considered in the delayed mapping stage. The fact that we use one-to-one mapping in our library routines allows negligence of this factor in the execution time model. If many-to-one mapping were used, the execution time would become more sensitive to the mapping since the issue of local versus nonlocal communication would come into the picture. Significantly higher startup cost of communication compared to the message transmit time, allows negligence of this factor in the model in this stage. Had it been the other way around, the execution time model must have captured the mapping function. Even though we have shown that for very large message sizes, the execution time may be affected by the link latency, lack of knowledge about the problem size and the systems size at library design time will make this difficult, if not impossible, to capture. Finally if the mapping function and the overlapping computation and communication are to be captured in the model, they must be rigorously defined as functions of the message size and the communicating distance between the physical processors.

The third step in the design of the library routine is determining a mapping function that does not degrade the performance of the library routine due to contention and communication bottleneck. This task is simplified on systems based on a fixed topology. Different mapping functions can be examined and the one with the minimum communication and contention may be selected for the delayed mapping phase. If the target architecture topology is not known, specific supported topologies must be considered and examined. If the algorithm is iterative and has inherent synchronization after each iteration, the contention can be examined for each phase. However, during each phase, the worst possible condition must be considered. This technique is used in OREGAMI [56].

The mandatory components of a library routine have been previously discussed in this chapter. The execution time function is embedded in the component which determines the scaling factor of the virtual architecture program. Some components, such as the ones that change the blocking factor, can be reused across different library routines. The physical system attributes can be configured prior to porting the parallel library to a new system. This is the only change required prior to porting, if the new system supports the communication and threading library used in the implementation of the library routine. The mapping library can be easily extended to support other virtual architecture algorithms. For partitionable systems the mapping library is quite smaller than reconfigurable systems since the physical topology is fixed.

As it-was mentioned, the user program basically runs on all the physical nodes. The parallel library calls divide the work by creating threads that run the library routine code. Since the user program could get rather complicated, we must ensure that all physical processors see the same image of variable, with the exception of distributed data. Data that is input must be broadcast to all the nodes, introducing additional overhead in the execution time of the program. There are additional intricacies that must be handled by the transformer, including declaration of additional variables and guarding parts of the code. The additional variables and the guards form the conditional part of the produced code that mask certain processors from executing specific parts of the code, such as the I/O statements. If a parallel library call is the only statement of the body of a conditional or iterative construct, it must be converted to a compound statement prior to the translation.

In order to maintain the semantics of the sequential program, the compiler must detect any changes in the distributed data. Distributed data must be invalidated if the corresponding global data is written to. One method is to insert calls to invalidate the distributed data when the global data is in the write set of a statement in the sequential program. An alternative, which would require more analysis, is to remove guards on such statements and translate them to an equivalent form which performs the same operation on the distributed data. It is important, however, to ensure that the same operation is performed on the global data as well as the distributed data, if the global data is referred to by the sequential code that follows the statement.

Routines that copy matrices or arrays can be easily parallelized. These routines can be implemented in the library to be further used in the programs. In such case, the program will take advantage of redistribution and performs the copying very fast. Copying element by element or manipulation of matrix objects outside of the library routines masks the potential redistribution operations because the generated SPMD code invalidates the distributed data. More details will be provided on invalidation of the distributed data in the next chapter.

Chapter 4

Automatic Data Redistribution

As it was discussed in the previous chapter, our major goal is to provide efficient parallel execution of library calls in a sequential program. The library calls are converted to an SPMD form which must distribute the initial data onto the virtual architecture. In our design of the parallel libraries, we observed that the data distribution component of the library routine must be run sequentially by a single processor designated as the host. The data distribution component of the library routine greatly affects the performance of the program since all other processors sit idle during this phase waiting for their initial data (see Figure 42 on Page 101). In many sequential programs, parameters of one library call are also used in a subsequent call. Therefore, after the first call, the data may stay in memory to be used during the second call. The runtime system can recognize such data items and rearrange them among the physical processors so that the second phase would not have to distribute the data from the host (see Figure 43). If the parallel computation does not require communication among the virtual processors, the virtual processors can proceed as soon as they receive their initial data (see Figure 43(a)). Mesh based matrix addition is an example of this kind. Since the virtual processors do not need data from one another, the computation on a virtual processor can complete independently of the other virtual processors. On the contrary, if the parallel computation requires communication among the virtual processors, the ones that receive their initial data sooner may have to wait until the last virtual processor receives its initial data (see Figure 43(b)). Torus based matrix multiplication is an example of this kind. In torus based matrix multiplication, there is implicit synchronization after each iteration. Data partitions of one virtual processor must be communicated with the two neighboring virtual processors (see the matrix multiplication algorithm on Page 81). When a virtual processor receives its initial data, it must eventually synchronize with the neighboring processors. It is important to note that the redistribution of data is a parallel operation by all the physical processors and is more likely to provide the data to all the virtual processors at about the same time (see Figure 43(c)). Therefore, the virtual processors are less likely to wait for one another after receiving their initial data. With data redistribution, the program can also benefit from simultaneous **send** operations issued by all the physical processors. Hence, the communication latency due to the data distribution will be mostly hidden.

The data item that is the subject of redistribution may be used in two phases of library calls based

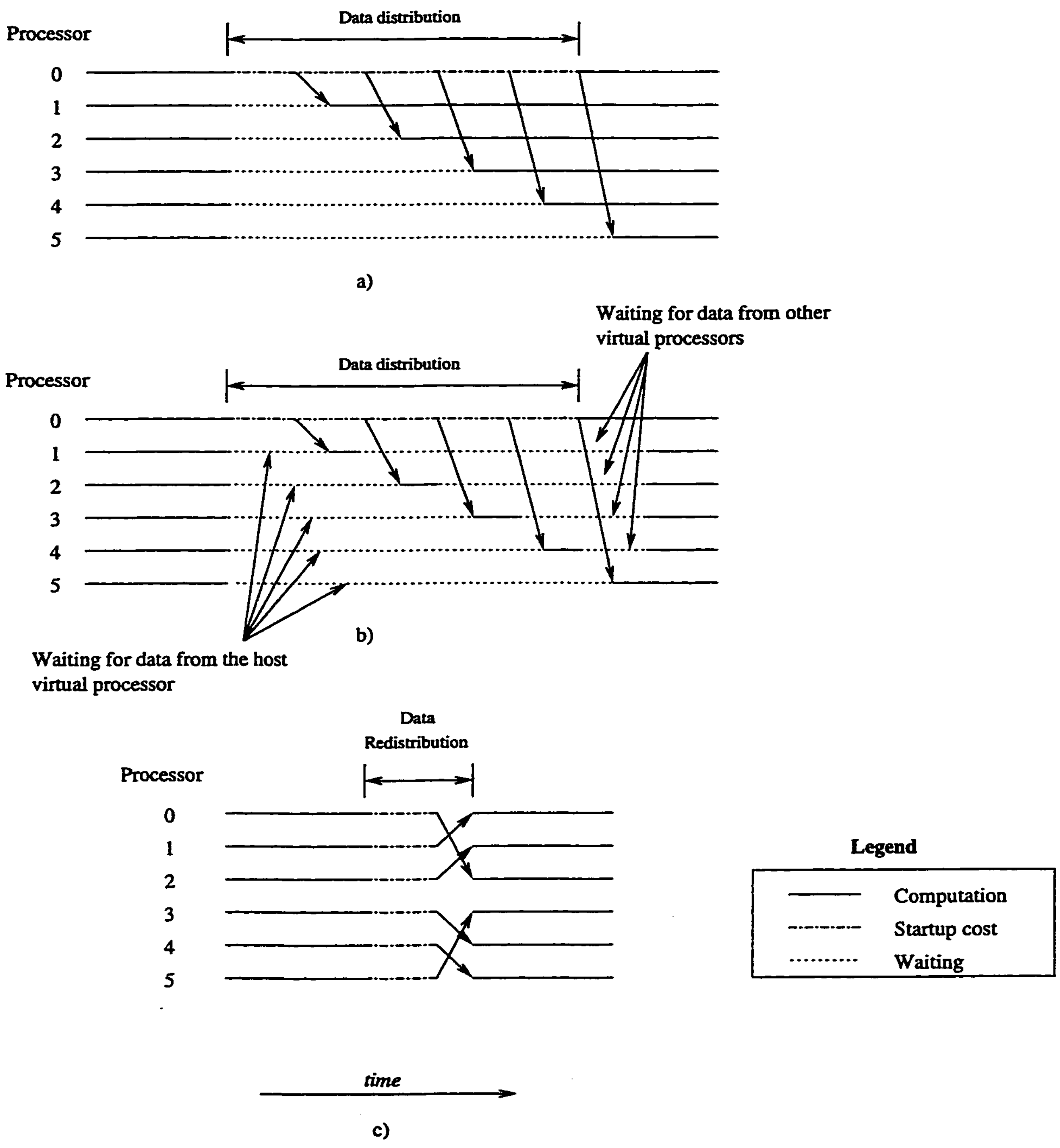


Figure 43: Example of distribution versus redistribution time

on two different virtual architectures, granularities, and mappings. All these attributes of parallel library calls are not known until runtime. Therefore, the computation of the source and destination of the data partitions cannot be done at compile time. Our model is based on the granularity of the two parallel phases. Our analysis shows that redistribution reduces the execution time of the library routine regardless of the grain sizes, data distribution, and the mapping of the two phases. The model captures the communication cost of each processor (virtual for distribution and physical for redistribution) by only considering the startup cost of the communication. We do not capture the effects of contention in the redistribution phase. The processor and data mapping functions of the two phases are not captured in the model. However, they are used by the redistribution module to determine the sources and destinations of the data blocks. We make no assumptions regarding the virtual processor topologies of the source and the destination.

The objective of this chapter is to offer a design of a redistribution module which conforms to our design of the library system presented in the previous chapter. Upon a call to a parallel library routine, the redistribution module must identify parameters which may be redistributed from a previous phase (not necessarily consecutive). If any parameter can be redistributed from a previous phase, the redistribution module must execute the right **sends** and **receives** on each physical processor to rearrange the data for the upcoming phase.

The approach used to achieve the above objectives is through maintaining the contraction factor, initial and final data layout requirements of the virtual architecture algorithms, and the processor mapping of the library calls. Handles are created and initialized upon entering a library routine, and are appended to the *history* list upon exiting the library routine. The history contains the dynamic chain of the *live* handles. A live handle is one that contains at least one valid distributed data item and may become subject of redistribution in a subsequent phase. Before starting a parallel phase, the redistribution component searches the list of handles, which are kept in the order of their creation, and finds the latest occurrence of a data item which still resides in memory. It then uses the information in the corresponding handle to redistribute the data. The old data buffer is released and its attribute is marked as *stale*, and the new data is marked as *redistributed*. Distributed data marked as **stale** means that the distribution of the data in the handle is not the latest. In other words, the data has been redistributed and used by a later parallel phase or does not contain the latest contents of the global data. Data is marked as **redistributed** in the current handle so that the upcoming phase does not perform initial data distribution. The initial data distribution module on page 70 has been modified in Algorithm 15 to support data redistribution. If all the data items of a handle become stale, the handle is considered stale and is released. It is important to note that only one distributed instance of each data item exists in the system at one time. Although it may look as if handles occupy large pieces of memory, this is really not the case. This information is kept in the global memory of each physical processor for each phase in the dynamic flow of the program. As library calls return to the sequential program and vanish, their handles are appended to a history list in the global memory. Upon entering a phase, the history of handles is searched for the distributed parameter in a previous phase. If it is found, the data is redistributed.

In languages such as HPF [52] the forementioned issues do not arise, since the programmer must

Algorithm 15 The initial data distribution module with provisioning for redistribution

```
Require: A handle with all known attributes of the call, contraction factor, global data, and the
    mapping table
1: Get handle for the current call
2: if I am the host virtual processor and the data is not redistributed then
3:     Perform blocking of global data
4:     Distribute partitions to the processors using virtual send
5: end if
6: if the data is not redistributed then
7:     Get initial data using virtual receive
8:     Place data in the proper location in the handle (use rank via vwho)
9: end if
Ensure: Data will be distributed based on the initial data distribution function
```

define the data distribution and redistribution. **DISTRIBUTE** and **REDISTRIBUTE** compiler directives are used to describe the location of the data blocks on the virtual architecture. A limitation in the currently supported redistribution mechanisms is that the number of the source and the target virtual processors must be equal. The block size of the computation is defined by the programmer, and user specified distribution and redistribution operations take place based on the user directives. The parameters of the redistribution are the source and destination logical architectures and the source and destination data distribution functions. The **DISTRIBUTE** directive can only occur in the specification part of the program, whereas **REDISTRIBUTE** directive can be used in the executable portion of the program. Dynamic data distribution is the term normally used for the **REDISTRIBUTE** directive. This directive allows the parameters such as the target block size to be specified dynamically. For instance, **REDISTRIBUTE**(r), where r is a program variable, is an acceptable statement in HPF.

In library systems such as ours, where parallelism is transparent to the programmer, redistribution phase must be strongly coupled with the library system. Since the problem instance size, physical and virtual architecture sizes and topologies, and the mappings are not known until the parallel phases begin, the computation of **send** and **receive** sets cannot be done statically. Furthermore, when data redistribution occurs, the preceding parallel phases do not exist. The information about the parallel phases such as the granularity of the computation, the mapping of the virtual processors, and the data layout is maintained in the list of handles in the global memory (see Section 3.1.3 on Page 58).

4.1 Data Redistribution Design

Our assumption in the design of a redistribution library is that the parallel library routines are based on our design described in the previous chapter. Each library call has embedded information in the corresponding handle. This information includes the granularity, the data mapping, and the processor mapping of each phase. This information is determined for each library call independently. Therefore, the granularity, the data mapping, and the processor mapping of the two phases may have no correlation. The parallel library calls may be used at the first level or nested within conditional or iterative constructs.

Our main objective is to offer a design of a redistribution module which supports the parallel library design discussed in Chapter 3. This includes supporting both the design of the virtual architecture parallel library routines as well as the source to source transformation of the user program. The redistribution module must identify the parallel phase which has a distributed instance of the data of interest. Once the distributed data is identified, appropriate low level **sends** and **receives** are executed by each processor to reshuffle the data. Our design must handle different combinations of the grain size, data mapping, and processor mapping. Furthermore, this must be done in an efficient manner which does not defeat the purpose of redistribution. The redistribution module must allow invalidation of the distributed data upon request from the SPMD program. These requests are inserted in the SPMD program when the user program alters the host copy of the distributed data.

The approach used to perform data redistribution is through usage of the distribution and mapping information in the handles of the source and target library calls. We categorize the redistribution into three cases where two cases are when the source and the target grain sizes are multiples of one another and the third one is when the granularity values do not divide one another. Data redistribution is basically a number of **send** and **receive** operations performed by each physical processor. It is important to note that no threads or virtual processors are active at the time of redistribution. Redistribution is done by all the main processes running on the physical processors.

As it was described in the previous chapter, each library call is represented by a handle. Handles are created upon entering a parallel phase, and are appended to a history list upon exit from the phase. A handle holds complete information about a library call. An important piece of information added to the handle is the virtual address of the actual parameter of the library call in the host processor. This address is the virtual address of the pointer object in the executable image of the program. This virtual address is unique across all the physical processors even if the host processor points to the actual data. This may not be the case with the absolute address since it depends on the virtual memory to physical memory address mapping on each physical processor. This is mainly because there is one executable image which is loaded onto all the physical processors and the user variables occupy the same virtual addresses in all the processors. This address uniquely identifies a user variable. In our design of the libraries we extend the handle with additional fields to facilitate redistribution (see Figure 44). The added field **global_data** is used to uniquely identify the variable in the user program. Since the addresses of the matrix variables are the same in all the images running on the physical processors, each node can find the source of the redistribution by comparing the address of the target with this address. For instance, if A is a matrix in the user program, whether it is global, local, dynamic, or static, it has a fixed virtual address in the executable image (note that we mean the address of A and not A itself). This address is seen by all the physical nodes. We basically avoid a broadcast of this information using the scheme just described. The **redistributed** field is set to TRUE if the data is redistributed so the parallel threads do not do initial data distribution. This field is set to TRUE by the redistribution module that runs before the parallel phase by each physical processor. All handles of the currently upcoming parallel phase will

```
typedef struct UserData             /* Local Data */
{
    char * data;                    /* Pointer to local data partitions */
    int dimensionality;             /* Dimensionality of the local data */
    int *dims;                      /* Size of each dimension */
    int size;                       /* Total size of data partition */
    int tag;                        /* For communication purpose */
    int **global_data;              /* Virtual address of global data */
    int redistributed;              /* Data redistributed */
    int stale;                      /* Data is stale */
} UserData;

typedef UserData * UserDataPtr;
```

Figure 44: Enhanced user structure for data redistribution

have this field set by the redistribution function running on the physical processor which corresponds to the handle. Likewise, the **stale** field is set to TRUE in the source when a data is redistributed. The new distribution of the data becomes active in this case. The structure of the enhanced handle is shown in Figure 45.

It is important to note that the redistribution takes place when no threads are active. This is done by the main process running on each processor. Therefore, the redistribution module uses low level communication routines and not the virtual communication routines. Even though a parallel phase vanishes by transition to outside of the library routine, its handle survives for possible redistributions. During redistribution, when the source handle is found, it and the current handle provide all the necessary information on the attributes of the library calls as well as the data buffers. The granularity, data layout, and mapping of the two handles are used to find the destinations of the blocks of the previous handle and the sources of the blocks of the current handle. Figure 46 shows the three cases of redistribution. In the first case in Figure 46(a), the granularity of the destination virtual architecture is an integer multiple of the granularity of the source virtual architecture. The blocks of data are sent to the destination physical processor and are placed in the designated area for the threads of the upcoming phase. Even though the threads are still not created, their required space and the mapping table is already setup before the redistribution routine is called. When all smaller blocks are placed in the larger partitions of the destination, the storage of these partitions is changed so that the blocks are in single element row major form to be used by the threads. This is called *increasing the block size* in our context. In the second case (see Figure 46(b)), the granularity of the source virtual architecture is an integer multiple of the granularity of the destination virtual architecture. The data partitions must be rearranged to match the block sizes of the second phase. This is known as *decreasing the block size* in our context. Once the storage is changed, smaller blocks of data are sent to the appropriate physical processors. In the third case (see Figure 46(c)), the granularities of the source virtual architecture the destination virtual architecture do not divide one another. The greatest common divisor of the two granularities is used for the grain size of

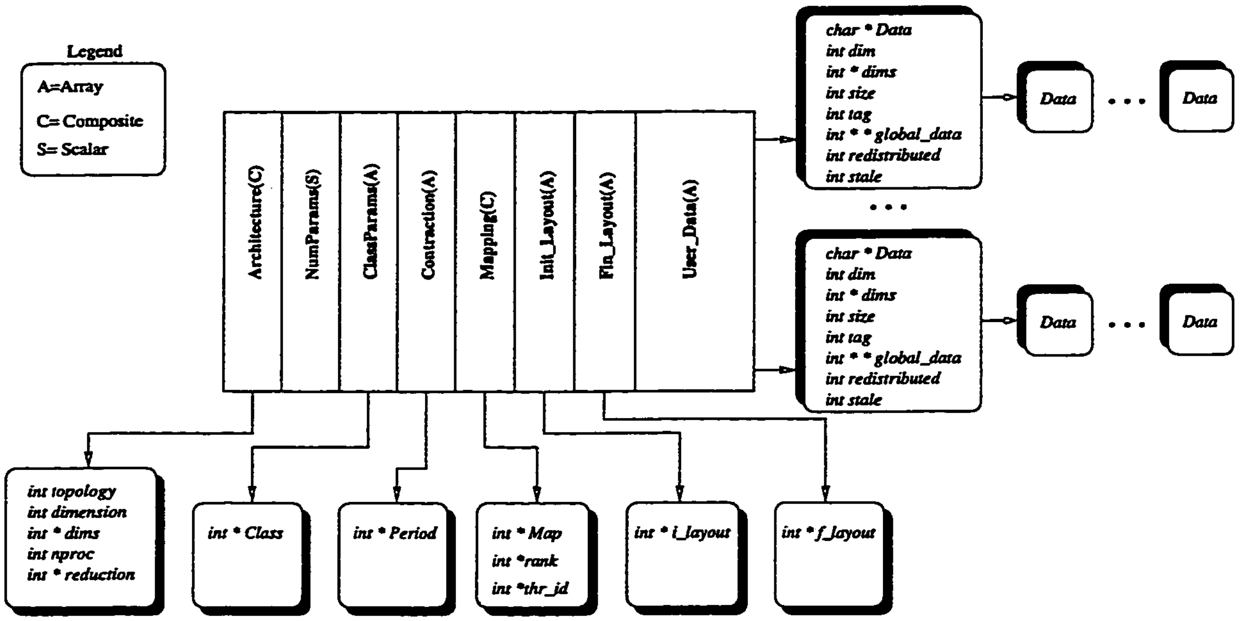


Figure 45: Enhanced Structure of a Library Handle

redistribution[1]. The data partitions are rearranged to match the redistribution granularity through a decrease in the block size. Smaller blocks of data are sent to the appropriate physical processors and placed in the right thread's partition. Once all the subblocks of a partition are received, its blocks size is increased from the redistribution granularity to the destination granularity.

Once the data is redistributed, an indication in the current handle prevents later data distribution. The data buffers of the source handle are released, and if the handle does not contain any more useful data, it is released as well. All operations take place simultaneously by all the physical processors.

Upon creation of a handle, the data in the handle is marked as *live*. A live data means it is currently distributed across the processors and the contents have not been altered since the last write. Live data must stay in the global memory for potential redistribution. Before creation of threads for a new parallel phase, a redistribution routine is called. In this call, all the current parameters of the library routine are considered one at a time for redistribution. The list of handles is traversed in reverse time for the first occurrence of a live data which globally corresponds to the parameter. For this global correspondence, the address of the global data is maintained in the handles. If the global data is live in the local memory of the processors, it is redistributed using the strategy defined in the rest of this chapter. When a data item is redistributed, the old copy is marked as *stale*. Stale data can be released as it should not be used for redistribution. The global address information is recorded in the new handle. In the event that no live occurrence of the global data is obtained, the data cannot be redistributed.

Following are the algorithms to perform redistribution between two parallel library phases. Two possible cases are when the block sizes are multiples of one another and one covers the case where

[1]Hereafter, we refer to this as the redistribution granularity.

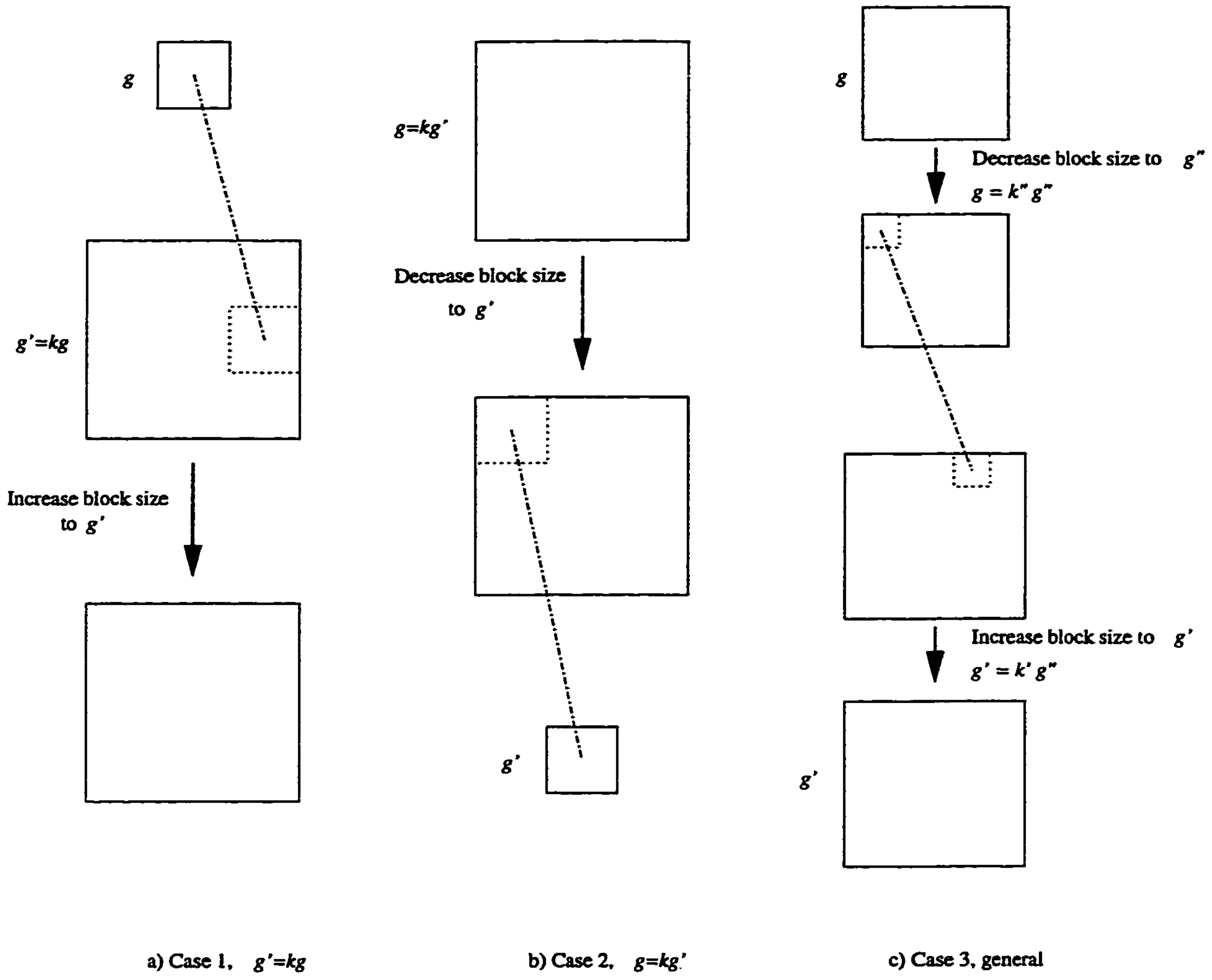

g
$g'=kg$
Increase block size to g'
a) Case 1, $g'=kg$
$g=kg'$
Decrease block size to g'
g'
b) Case 2, $g=kg'$
g
Decrease block size to g''
$g = k'' g''$
Increase block size to g'
$g' = k' g''$
g'
c) Case 3, general

Figure 46: Three cases of redistribution

there is no specific relationship between the block sizes. In these algorithms, the term *global data* refers to the data in the host processor. The terms *block* and *subblock* are used interchangeably depending on the point of reference. Algorithms 16, 17, and 18 show, for case 1, the high level

Algorithm 16 High level algorithm for data redistribution, case 1: $g' = kg$

```
Require: Source and target handles with source and target parameters, $g' = kg$
 1: for all Blocks of the parameter in the source handle do
 2:   Determine the destination
 3:   Send the block using Algorithm 17
 4: end for
 5:
 6: for all Blocks of the parameter in the target handle do
 7:   for all Subblocks within the block do
 8:     Receive the subblock using Algorithm 18
 9:   end for
10:   Increase subblock size of the block from $g$ to $g'$
11: end for
Ensure: Source parameter is sent to the right processors, and target parameter is received from
    the right processors
```

Algorithm 17 Computation of the destination set for case 1, $g' = kg$

```
Require: Mapping table and final data layout of the source parameter, and the mapping table and
    the initial data layout of the target parameter.
 1: Retrieve the block
 2: Use final data layout and map table of the source handle to compute the global block number
    of this block
 3: Use initial data layout and map table of the target handle to compute the destination of this
    block
 4: Attach global block number to the message and send to destination
Ensure: A block is sent to the appropriate processor.
```

Algorithm 18 Computation of the source set for case 1, $g' = kg$

```
Require: Mapping table and initial data layout of the target parameter
 1: Compute global block number of the subblock using the target block number and the subblock
    number within the block. Use the mapping table and initial data layout of the target parameter
 2: Receive the subblock
Ensure: A subblock is received
```

redistribution algorithm, computation of the source set, and computation of the destination set, respectively. In these algorithms, the source messages are tagged with their global block number to be uniquely identified upon receipt. The computation of the source and the destination sets are shown in Figure 47 for $g' = kg$. The left diagram shows the translation scheme used to identify the destination of a block. This translation requires use of the map table to identify the block number of the partitions in the global data. The block number is then attached to the message and sent out to the physical processor requiring the partition. The right diagram, in the same figure, shows the translation scheme used to identify the source of a block. This is done by computing the global

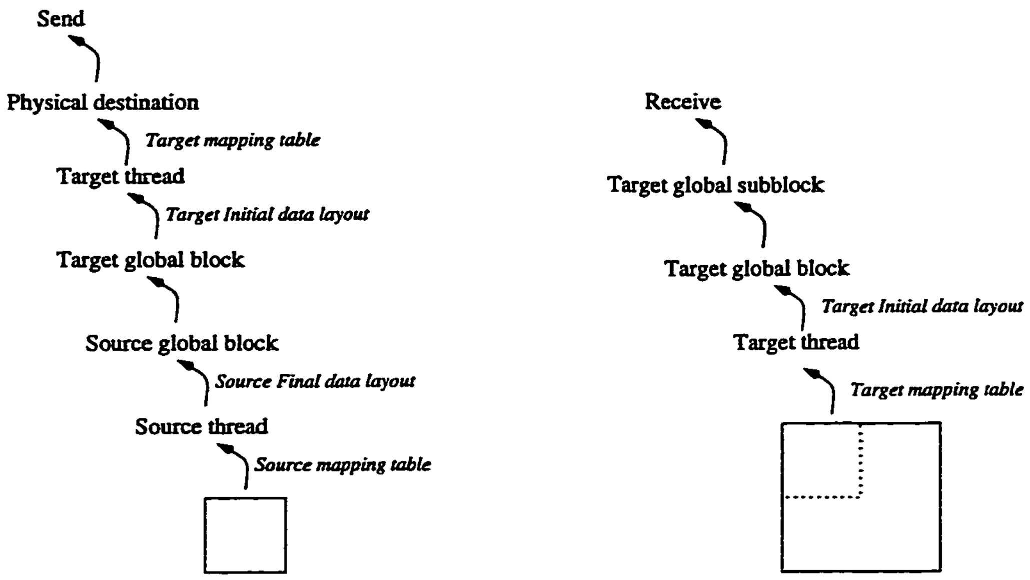


Figure 47: Computation of the source (left) and the destination (right) sets for the case $g' = kg$

block number of the subblock and receiving a message with such block number. Suppose block a is to be redistributed by physical processor s. If the block corresponds to block number a' in the global matrix and the destination physical processor is d, the message (a, a') is send to d. Physical processor d also performs a receive for a message that has a' attached to it. We have used the tags of the PICL like communication library to send the block number. The redistribution module uses the low level communication library as opposed to the virtual communication library. Algorithms 19,

Algorithm 19 High level algorithm for data redistribution, case 2: $g = kg'$

```
Require: Source and target handles with source and target parameters, $g = kg'$
 1: for all Blocks of the parameter in the source handle do
 2:    Decrease block size of the partition from $g$ to $g'$
 3:    for all Sublocks within the block do
 4:       Determine the destination
 5:       Send the subblock using Algorithm 20
 6:    end for
 7: end for
 8:
 9: for all Blocks of the parameter in the target handle do
10:    Receive the block using Algorithm 21
11: end for
Ensure: source parameter is sent to the right processors, and target parameter is received from the
    right processors
```

20, and 21 are the analogous algorithms for case 2. The computation of the source and destination sets are shown in Figure 48 for the case $g = kg'$. Algorithms 22, 23, and 24 correspond to the general case where g and g' are not multiples of one another. The computation of source and destination sets are shown in Figure 49 for the general case (if g and g' are not multiple of one another). The routines for the second and the third case are very similar to the first case. In the first case, the

Algorithm 20 Computation of the destination set for case 2, $g = kg'$

```
Require: Mapping table and final data layout of the source parameter, and the mapping table and
    the initial data layout of the target parameter
1: retrieve the block
2: Use final data layout and map table of the source handle to compute the global block number
   of this subblock.
3: Use initial data layout and map table of the target handle to compute the destination of this
   subblock
4: Attach global block number to the message and send to destination
Ensure: A subblock is sent to the appropriate processor
```

Algorithm 21 Computation of the source set for case 2, $g = kg'$

```
Require: Mapping table and initial data layout of the target parameter
1: Compute global block number in the source using the target block number. Use the mapping
   table and initial data layout of the target parameter
2: Receive the block
Ensure: A block is received
```

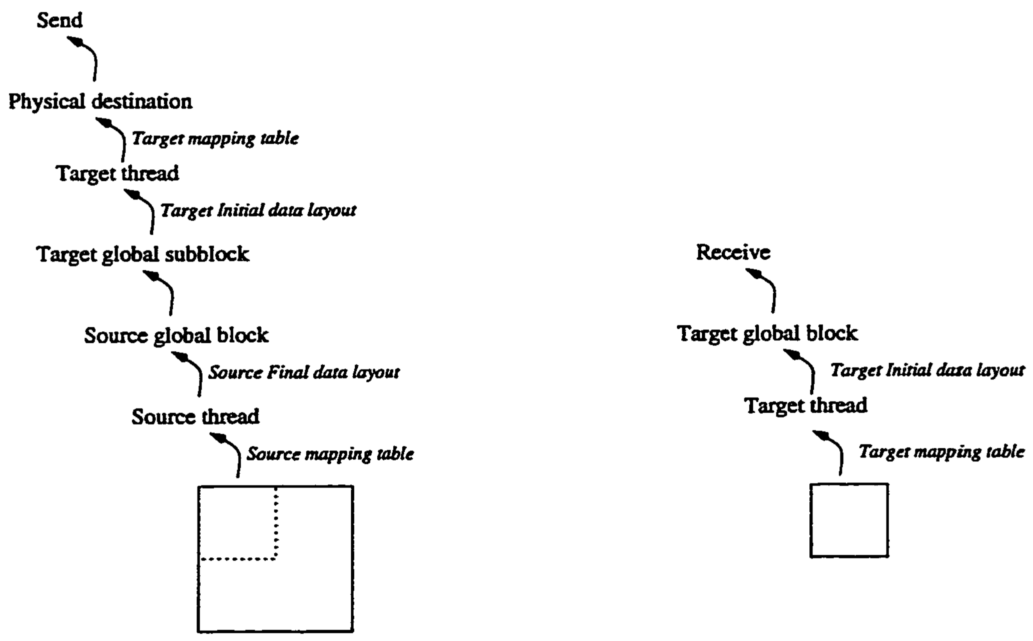


Figure 48: Computation of the source (left) and the destination (right) sets for the case $g = kg'$

Algorithm 22 High level algorithm for data redistribution, case 3: general

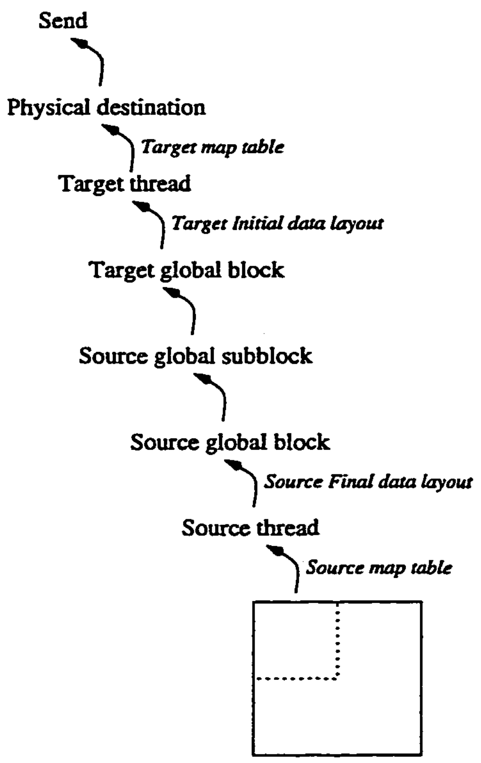
```
Require: Source and target handles with source and target parameters, no specific relationship
    between $g$ and $g'$. Select $g''$ that divides both $g$ and $g'$
1: for all Blocks of the parameter in the source handle do
2:     Decrease block size of the partition from $g$ to $g''$
3:     for all Subblocks within the block do
4:         Determine the destination
5:         Send the subblock using algorithm 23
6:     end for
7: end for
8:
9: for all Blocks of the parameter in the target handle do
10:     for all Subblocks within the partition do
11:         Receive the subblock using algorithm 24
12:     end for
13:     increase block size for the partition from $g''$ to $g'$
14: end for
Ensure: source parameter is sent to the right processors, and target parameter is received from the
    right processors
```

Algorithm 23 Computation of the destination set for case 3, general

```
Require: Mapping table and final data layout of the source parameter, and the mapping table and
    the initial data layout of the target parameter
1: retrieve the subblock
2: Use final data layout and map table of the source handle to compute the global block number
   of this subblock.
3: Use initial data layout and map table of the target handle to compute the destination of this
   subblock
4: Attach the global block number to the message and send to destination
Ensure: A subblock is sent to the appropriate processor
```

Algorithm 24 Computation of the source set for case 3, general

```
Require: Mapping table and initial data layout of the target parameter
1: Compute global block number using the target subblock number. Use the mapping table and
   initial data layout of the target parameter
2: Receive the subblock
Ensure: A subblock is received
```



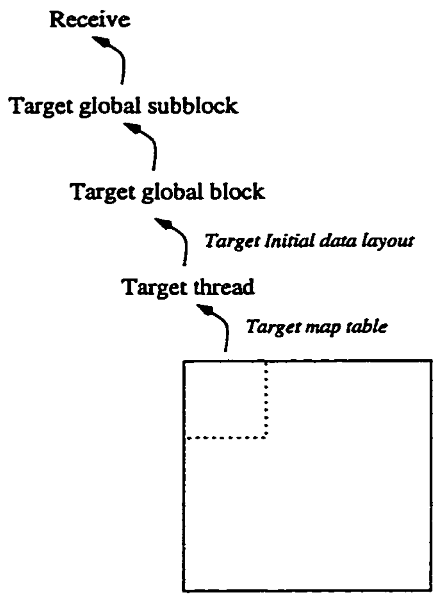


Figure 49: Computation of the source (left) and the destination (right) sets for the case with no multiplicative relationship between g and g'

blocks do not need to be broken at the source and they must be assembled at the destination to form larger blocks. In the second case, larger blocks must be broken up into smaller subblocks and sent to the destination processors. The third case is a hybrid of the above two cases. The blocks are broken up at the source, sent out to the processors, and assembled into larger blocks at the destination.

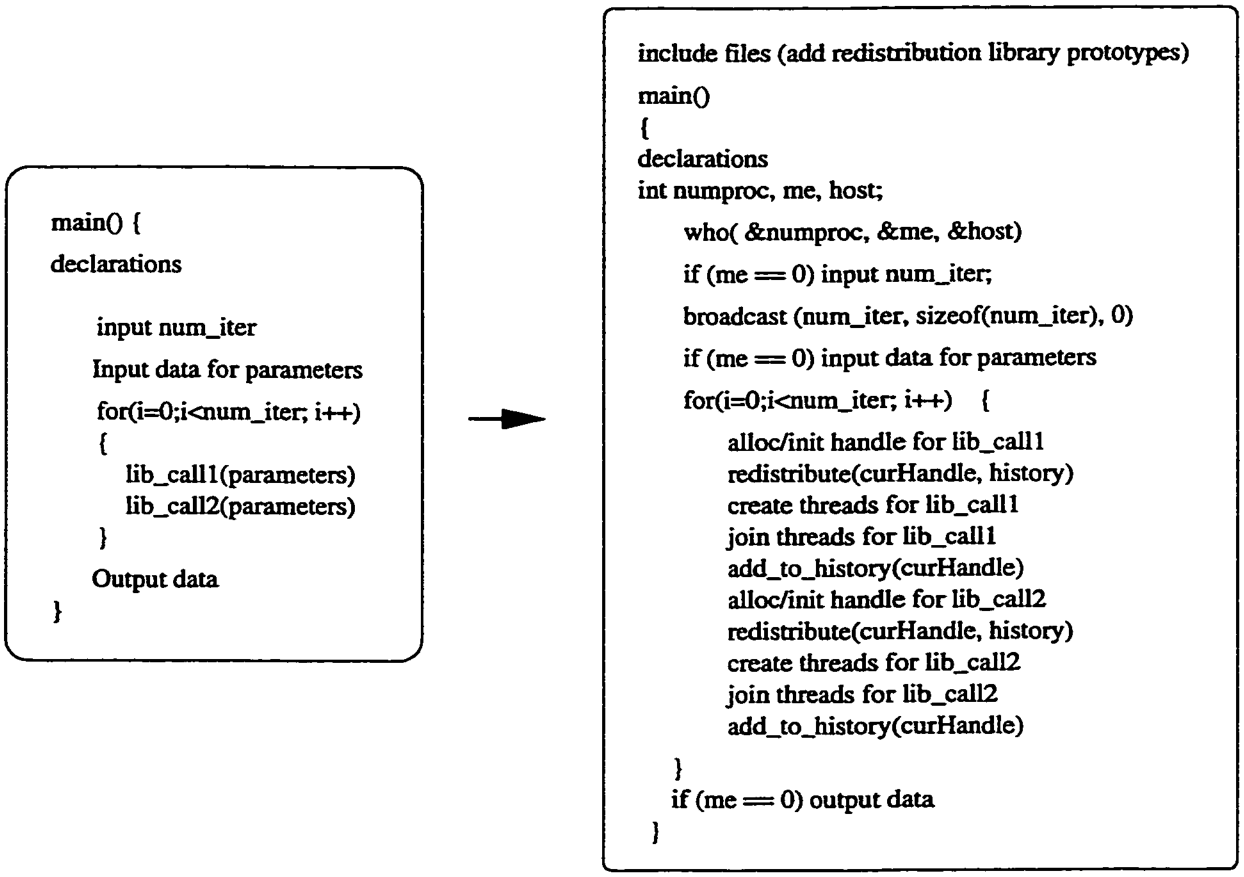
4.2 Data Redistribution and the Source to Source Transformation

We assume the user program consists of several parallel library calls. These calls may be nested within various programming constructs or in user defined functions. The libraries are designed based on the proposed scheme in Chapter 3. Further assume a redistribution module designed based on the proposed scheme in the previous section.

The objective is to offer a source to source transformation which performs data redistribution before a parallel phase. An additional objective is to insert the appropriate code in the SPMD program to invalidate distributed data when they do not match the global copy.

The approach used is to insert a call to the **redistribute** function after the mapping and contraction and before the thread creation. Data redistribution is basically a call to this routine in the redistribution module. Figure 50 shows an example program with its translated form. The redistribution module takes the current handle and the history list as parameter. The parallel library calls invoked at the beginning of the program are less likely to benefit from redistribution, because the data is most likely at the host. As a parallel library routine call completes, its handle is appended to the history list in the global memory. Therefore, more poly data (see description on Page 73) will exist in distributed form. Once we have one instance of each poly variable in the program in distributed form, no more data distribution will occur. All data will be provided through data redistribution.

The approach to invalidation of distributed data is by identifying the writes and I/O on poly data. The algorithm for source to source transformation on Page 74 must be enhanced to insert a call to the **invalidate** routine in the redistribution module. The call to this routine must be inserted in step 9 to 11 of this algorithm. This is where writes to poly data is being guarded. This **invalidate** routine accepts the address of the poly data (virtual address) and the history list. The routine basically searches the history list for a handle which holds the corresponding distributed data. It will then mark the data as stale and releases the associated storage. Figure 51 shows an example translation of a program which writes to a poly variable in between two parallel library calls. The code for **invalidate** can be found in Appendix D. It is important to note that the call to **redistribute** is generated before every parallel library call thread creation regardless of whether redistribution takes place or not. The invalidate routine basically marks the data as **stale** so that the **redistribute** routine does not select it for redistribution.

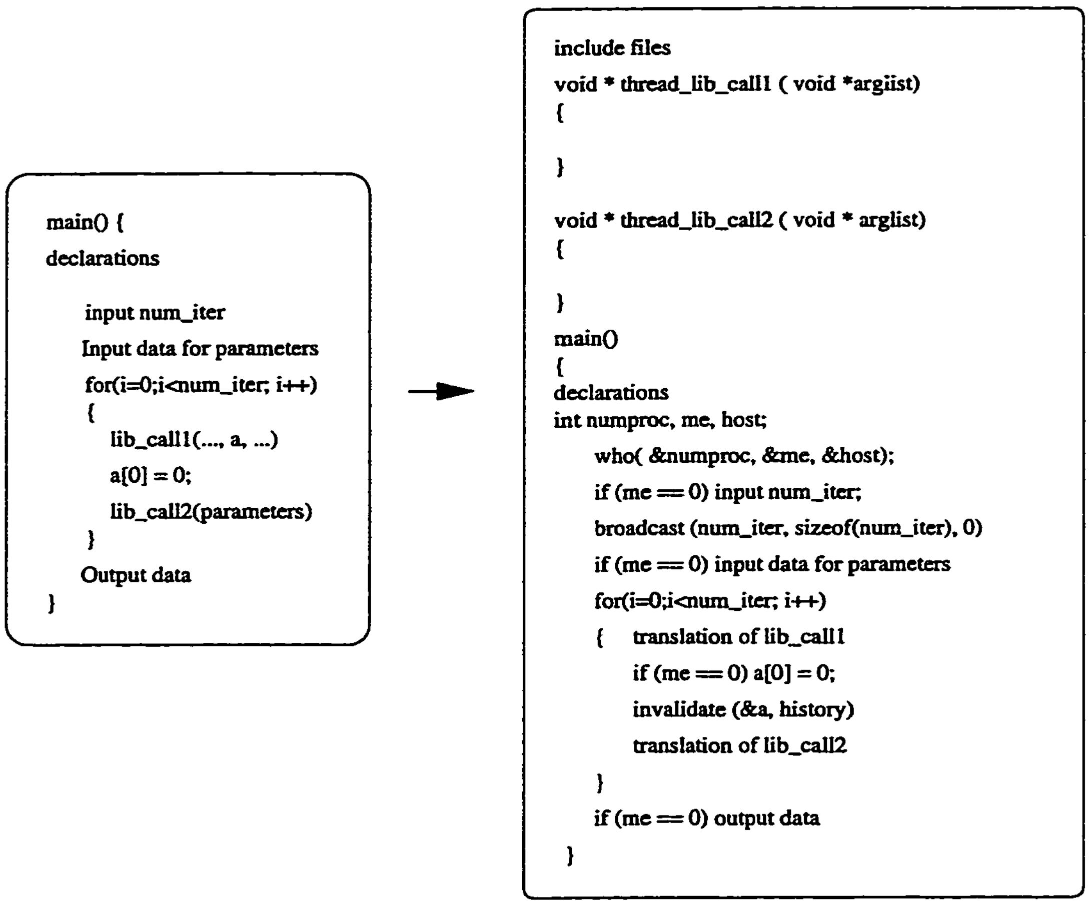


```
main() {
declarations

    input num_iter
    Input data for parameters
    for(i=0;i<num_iter; i++)
    {
       lib_call1(parameters)
       lib_call2(parameters)
    }
    Output data
}
```

```
include files (add redistribution library prototypes)
main()
{
declarations
int numproc, me, host;
    who( &numproc, &me, &host)
    if (me == 0) input num_iter;
    broadcast (num_iter, sizeof(num_iter), 0)
    if (me == 0) input data for parameters
    for(i=0;i<num_iter; i++)   {
        alloc/init handle for lib_call1
        redistribute(curHandle, history)
        create threads for lib_call1
        join threads for lib_call1
        add_to_history(curHandle)
        alloc/init handle for lib_call2
        redistribute(curHandle, history)
        create threads for lib_call2
        join threads for lib_call2
        add_to_history(curHandle)
   }
   if (me == 0) output data
  }
```

Figure 50: Source to source transformation with support of redistribution



```
main() {
declarations

    input num_iter
    Input data for parameters
    for(i=0;i<num_iter; i++)
    {
       lib_call1(..., a, ...)
       a[0] = 0;
       lib_call2(parameters)
    }
    Output data
}
```

```
include files
void * thread_lib_call1 ( void *argiist)
{

}

void * thread_lib_call2 ( void * arglist)
{

}
main()
{
declarations
int numproc, me, host;
    who( &numproc, &me, &host);
    if (me == 0) input num_iter;
    broadcast (num_iter, sizeof(num_iter), 0)
    if (me == 0) input data for parameters
    for(i=0;i<num_iter; i++)
    {   translation of lib_call1
        if (me == 0) a[0] = 0;
        invalidate (&a, history)
        translation of lib_call2
    }
    if (me == 0) output data
 }
```

Figure 51: An example invalidation of distributed data

4.3 Performance Analysis

In this section we analyze the cost of data distribution versus data redistribution of a matrix object that recurs in the parameter list of two library calls. Although in our redistribution module design we have not made any assumptions about the mapping functions, our parallel libraries have a tendency of reducing the virtual architecture size to the physical system size or smaller than it. Therefore, each physical processor will be assigned at most one virtual processor. We consider the worst condition for mapping of the data and processors in the following analysis and still show that redistribution will result in performance gain. By worst case condition we mean that when a data block is to be redistributed, its destination will be another physical processor. Therefore, it incurs a startup communication cost using our approximate model. If the mapping of the data and virtual processors is such that the destination of the data is the same physical processor, the redistribution cost of the block is basically a memory to memory copy. In the following analysis, we approximate the communication models in equations 9 and 10 on Page 80 by considering only a startup cost for sending a message. In the context of data distribution, this is a safe assumption as repeated **send** operations hide the communication latency for large messages. The effects of contention in the redistribution cost are not captured in our redistribution module. The mapping function is used to determine the source and the destination of each data block. Data distribution and redistribution normally require change of storage of matrix data to conform with the block storage. We approximate both data distribution and redistribution by neglecting this cost.

The objective of this section is to show, through performance analysis, that the redistribution of a matrix object results in performance gain regardless of the granularity of the two phases under worse condition scenario previously described.

The approach used in this chapter is through the comparison of the cost of initial data distribution with data redistribution. We model the cost of initial data distribution with the number of nonlocal point-to-point communication operations issued by the host virtual processor. Furthermore, we model the cost of redistribution with the maximum number of nonlocal point-to-point operations over all the physical processors. It is assumed that the data layout and the processor mapping of the source and the target handles are such that communication operations are always nonlocal (this is the worst condition).

We calculate the data redistribution time for three different cases of grain size for a matrix object of size $n \times n$. Furthermore, we calculate the time for data distribution and compare it with all the three cases. Assume the granularities of the distributed data in the first and the second phase are g and g', respectively. We will consider four cases below. The first case analyzes communication time under no data redistribution support. Cases two and three consider data redistribution when the grain sizes are integer multiples of one another. The fourth case considers data redistribution for grain sizes that are not multiple of one another. We consider the cases below with respect to the cost of communication. Assume the startup cost of communication is t_s, the data distribution time is T_{dist}, and the data redistribution time is T_{redist}. The values of k and k', used below, are constants.

No redistribution - The time to distribute the data consists mainly of the the cost of **send** operations to distribute the blocks of data,

$$T_{dist} = \frac{n^2}{g'^2} t_s.$$

Redistribution with $g = kg'$ for $k > 1$ - Larger grains must be broken up and sent out to the destination processors. The redistribution time can be stated as,

$$T_{redist} = k^2 t_s.$$

For redistribution to benefit the execution time, the following inequality should hold,

$$k^2 t_s \leq \frac{n^2}{g'^2} t_s.$$

Comparing the two sides of the inequality, $k^2 \leq \frac{n^2}{g'^2}$ holds. Therefore, this relation is always true.

Redistribution with $g' = kg$ for $k > 1$ - Smaller partitions must be merged into a larger partition,

$$T_{redist} = t_s$$
$$t_s \leq \frac{n^2}{g'^2} t_s.$$

Since $\frac{n}{g'} > 1$, the inequality is always true.

If g and g' are not multiples of one another - Data is broken up at the source, sent out to the destination processors, and merged into the target partitions. If $g = k'g''$, and $g' = k''g''$, where g'' is the greatest common divisor of g and g', and k' and k'' are constants, we have,

$$T_{redist} = k'^2 t_s$$
$$k'^2 t_s \leq \frac{n^2}{g'^2} t_s.$$

Comparison of the terms on the two sides of the inequality yields $\frac{gg'}{\gcd(g,g')} \leq n$ or LCM$(g, g') \leq n$, which is always true since g and g' both divide n. The analysis shows that data redistribution reduces the execution time for all three cases. In this analysis, it is assumed that all data partitions are sent to nonlocal physical processors in both data distribution and redistribution. This assumption favors the data distribution case. In data distribution, nearly all data partitions will be sent to nonlocal processors. In data redistribution, however, it is likely that no nonlocal communication operations will be required. This occurs when the final data layout of the previous phase and the initial data layout of the upcoming phase match as well as the processor mapping of the two phases.

4.4 Performance Results

The assumption in this section is the usage of parallel library implementation and redistribution based on our design described in Chapter 3 and this chapter. The sequential program is run through

$g \rightarrow$	2	4	5	2
$g' \rightarrow$	4	2	2	5
Without redistribution	30.4 msec	25.0 msec	25.4 msec	31.9 msec
With redistribution	24.5 msec	19.8 msec	21.0 msec	26.9 msec
Percent difference	19.4%	20.8%	17.3%	15.7%

Table 12: Simulation data: Performance gain in data redistribution.

our source to source transformer and is converted to an SPMD form. The virtual communication library, mapping library, thread library, and redistribution module are then linked in with the SPMD program.

The objective of this section is to provide performance results of a program with two parallel library calls with and without redistribution. The first parallel phase is a call to matrix multiplication and the second one is to matrix addition.

Our approach in providing the following is through the use of our programming environment and the libraries previously described. The parameters of simulation were a startup cost of $t_s = 30$ μsec and a link bandwidth of $t_w = 0.024$ μsec. Packet switching was used for the simulations. The performance gain can be easily shown with wormhole routing, as well. The example is based on the expression $D = A * B + C$, where the data size of $n = 20$ and a physical square mesh of size $p = 25$ were used. Values of g and g' were manually selected. Table 12 presents the results for some combinations of g and g' covering all three cases of redistribution previously mentioned.

The dynamically managed list of handles introduces space overhead that is inevitable. Certain data items that are kept in the list may never become a source of redistribution. Other distributed data, which are used for redistribution, have only one copy in the list of handles. In the worst case, there will be at most one distributed instance of every global data. As data items become stale, the associated data buffers are released. If all the data items in a handle become stale, the handle is released as well.

Two sources of overhead exist during redistribution. The first is finding the handle which is the source of redistribution. This is largely dependent on the sequence of library calls and the usage of the global data as parameters. In the worst case the history list is scanned and no match is found. The time to collect this information is proportional to the ratio of the problem size and the grain size of redistribution. The second overhead is that of computing the source and the destination sets which is proportional to the ratio of the source grain size and the redistribution grain size.

Chapter 5

An Integrated Parallel Library Programming Environment

The implementation of the numerical library routines, the mapping library, virtual communication library, and the redistribution library were discussed in Chapters 3 and 4. In this chapter we discuss the underlying compile time, run time, and the physical system simulator which were used to experiment with the parallel libraries.

The programming environment used to conduct the experiments in this thesis consists of the following components:

- Source to source transformer and the compiler
- Multicomputer simulator
- Low level communication subsystem
- Process and thread simulation

These components will be discussed in the following sections.

5.1 Source to Source Transformation and the Compiler

The source to source transformer basically inputs a C program and, using conventional compiler technology, constructs its corresponding abstract syntax tree. The tree is then traversed identifying basic blocks[1] and parallel library function calls. Sequential parts of the code are then placed in guards. At basic block boundaries, a guard is terminated and another one is started inside the block. The parallel library function calls are expanded to a sequence of calls to determine the contraction factor, distribute the data, or possibly redistribute data from a previous phase, perform the local computation, and accumulate the result at the virtual host. Additional code is also inserted to update the information regarding the distribution of the library routine parameters. Figure 52

[1] Our definition of basic blocks is the same as the definition in conventional compiler books.

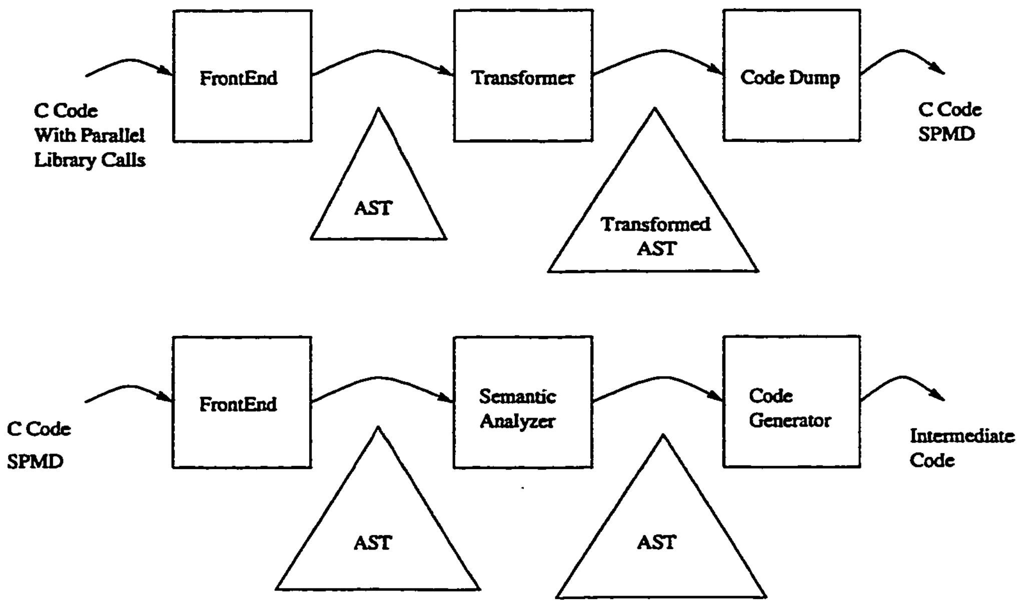


Figure 52: Source to source transformation and compilation process

shows the two step compilation process of the library calls. For a complete example the reader is encouraged to see Appendix E.

In the source to source transformation, a call to **lib_init** is generated at the beginning of the main program and a call to **lib_exit** generated at the end. These two function calls are components that are meant for runtime actions which take place only at the beginning and the end of the user program execution. The transformation for a call is shown in Figure 53. Some of the components of the library routine require specific actual parameters of the library call to perform their task. The transformer identifies the parameters of the library call and provides them, discriminantly, to these components. Even though some of these components may not require all the parameters, for simplicity of the source to source transformation all parameters are provided to the components. This reduction in the compile time overhead is at the cost of a very small overhead of passing unnecessary pointers to the library routine components. For instance, **lib_init_architecture** requires only its last parameter, which is the size of the data, to perform its task, whereas **lib_distribute_initial** would require parameters that contain data for the numerical computation.

Guards are placed around pieces of code that are to be executed only by the host processor. Basic blocks are identified and guards are started at the beginning of these blocks and terminated at the end. A new guard is then started for the next basic block. Parallel library calls terminate guards and once the translation of the library call is complete, a new guard starts (Figure 54). The additional **Forallthread** construct, which takes the mapping table and the total number of threads as arguments, generates all threads mapped to the current processor. Upon entering this construct, the simulator scans the mapping table and creates all the threads that are mapped to the current physical processor. The creation of the threads is discussed in detail in section 5.2 on Page 129. The threads commence execution at the first statement of the body of the **Forallthread**. Execution of

```
main()
{
declarations

    Input data
    lib_call(parameters)
    Output data
}
```

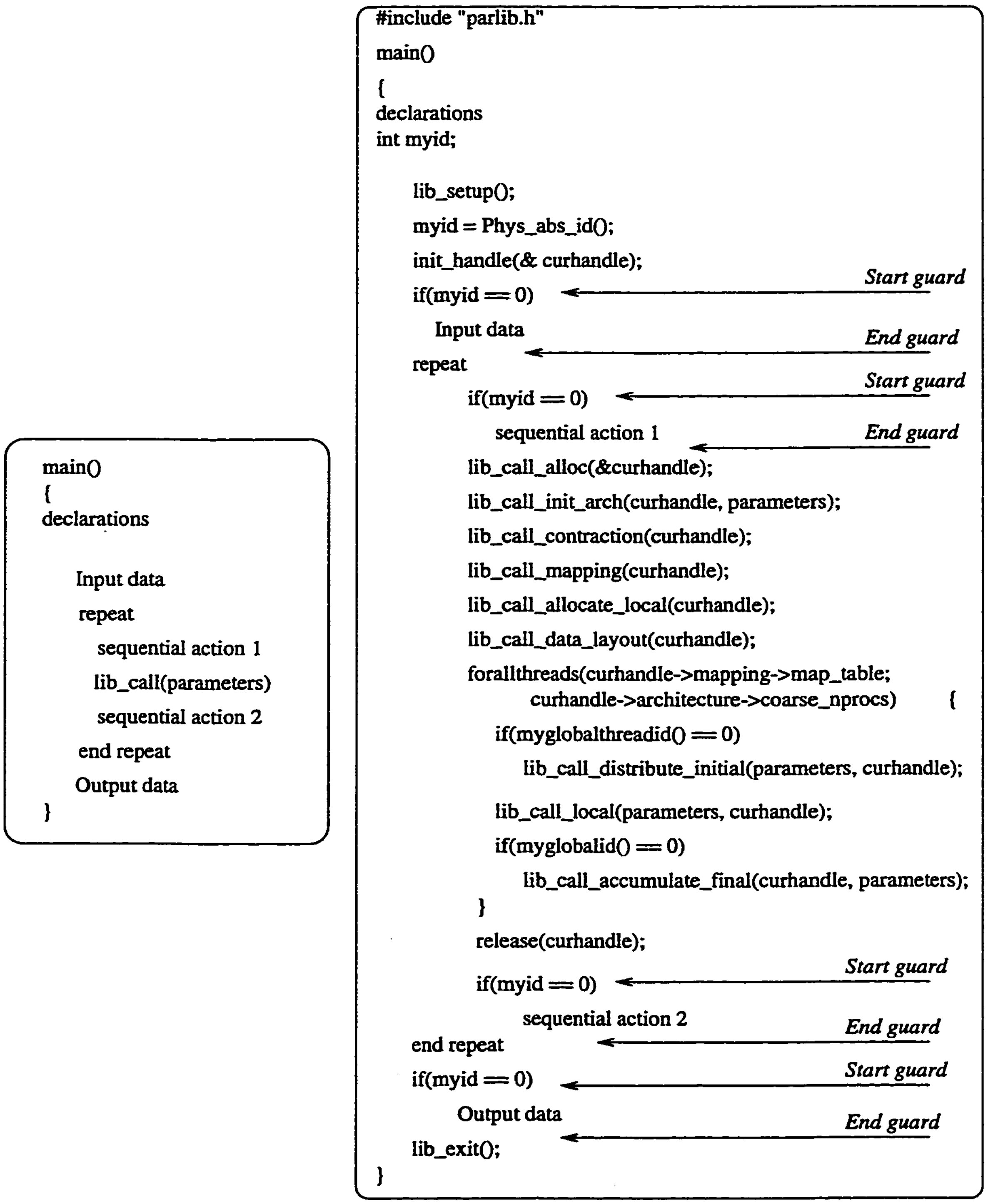
```
#include "parlib.h"

main()
{
   declarations
   int myid;

   lib_setup();
   myid = Phys_abs_id();
   init_handle(& curhandle);
   if(myid == 0)
       Input data
   lib_call_alloc(&curhandle);
   lib_call_init_arch(curhandle, parameters);
   lib_call_contraction(curhandle);
   lib_call_mapping(curhandle);
   lib_call_allocate_local(curhandle);
   lib_call_data_layout(curhandle);
   redistribute(curhandle, history);
   forallthreads(curhandle->mapping->map_table;
                 curhandle->architecture->coarse_nprocs)      {
          if(myglobalthreadid() == 0)
                 lib_call_distribute_initial(parameters, curhandle);
          lib_call_local(parameters, curhandle);
          if(myglobalid() == 0)
                 lib_call_accumulate_final(curhandle, parameters);
   }

   add_handle_to_history(curhandle);
   if(myid == 0)
          Output data
   lib_exit();
}
```

Figure 53: Example of a source to source transformation (Redistribution ON)



```
main()
{
declarations

    Input data
    repeat
      sequential action 1
      lib_call(parameters)
      sequential action 2
    end repeat
    Output data
}
```

```
#include "parlib.h"
main()
{
declarations
int myid;

    lib_setup();
    myid = Phys_abs_id();
    init_handle(& curhandle);
    if(myid == 0)
      Input data
    repeat
        if(myid == 0)
          sequential action 1
        lib_call_alloc(&curhandle);
        lib_call_init_arch(curhandle, parameters);
        lib_call_contraction(curhandle);
        lib_call_mapping(curhandle);
        lib_call_allocate_local(curhandle);
        lib_call_data_layout(curhandle);
        forallthreads(curhandle->mapping->map_table;
              curhandle->architecture->coarse_nprocs)      {
          if(myglobalthreadid() == 0)
            lib_call_distribute_initial(parameters, curhandle);
          lib_call_local(parameters, curhandle);
          if(myglobalid() == 0)
            lib_call_accumulate_final(curhandle, parameters);
        }
        release(curhandle);
        if(myid == 0)
            sequential action 2
    end repeat
    if(myid == 0)
        Output data
    lib_exit();
}
```

Figure 54: An example generation of guards (Redistribution OFF)

the parent is temporarily blocked until all created threads execute the body. At that point, the main process starts execution at the statement immediately after the end of the compound statement. It is likely that some physical processors are not assigned any threads. In this case, the execution of the main process is not halted. In other words, global synchronization is not enforced at the end of a parallel library call. However, at the beginning of a parallel library call, before creation of threads, the size of the problem instance is broadcast to the physical processors. This call to the lower level broadcast enforces an implicit synchronization at the beginning of each call. With this synchronization at the beginning of each library call, two parallel library calls will not be mixed and their execution follows the program order. Data accumulation is another implicit synchronization at the end of a parallel library call. Most virtual architecture algorithms compute values that must be sent back to the host. If the user program refers to the global data after the call, the data accumulation ensures that the data is received prior to executing the guarded portion which uses the returned value. If the parallel library call does not return any value to the host processor (this is very rare), access to variables after the call will not rely on the result of the call anyway. Local synchronization among threads is enforced by the **Forallthread**.

There are additional guards in the body of the **Forallthread** for data distribution and accumulation. The first thread is always mapped to the first physical processor and has access to the main variables. These variables are user defined and contain the data used in the computation. The call to **lib_release** returns the handle and all its constituents to the free memory.

The **Forallthread** construct is translated by the compiler to two intermediate codes **StartThreads** and **EndThreads**. These two codes enclose a group of statements which are executed by the created threads. The **StartThreads** intermediate code indivisibly creates all the threads that are mapped to a physical processor. The created threads start execution at the first statement following the **StartThreads**. Eventually, all the threads will execute the **EndThreads** intermediate code. All the threads will then merge into a single flow of control (that of main). The **StartThreads** is executed by the main process but the **EndThreads** is called by the local threads. Internal to the simulator, a semaphore is used to keep a count of the number of terminated threads. The semaphore is initialized in **StartThreads**. When the number reaches zero, the main process resumes execution.

Aside from the **Forallthread** construct, which can also be implemented using a threading library, our language is based on the full set of C constructs. The C standard library has been implemented as part of the builtin functions. For a full set of language description, the reader may refer to the C language book [51].

In addition to the **Forallthread** construct, our system supports two builtin functions to return the global and the local thread identifiers of a thread. **MyGlobalThreadId** returns the virtual processor identifier of the calling thread. This call is normally used by a thread to check its own identifier and take actions accordingly. The corresponding virtual function, which is described by algorithm 1 on page 54 in Chapter 3, is the **vwho** routine. The local thread identifier (rank) is returned by the **MyLocalThreadId** builtin function. The returned value of this call is normally used by a thread to index into a contiguous memory and find its partition. The contiguous memory is allocated for all the threads running on a physical processor. The **vwho** routine has an extra argument to return the rank of a virtual processor. In other words, in the real system implementation of the libraries the above two builtin functions are merged into the **vwho** routine.

In order to mimic the behavior of an actual multicomputer, each physical processor was given a separate stack space. The stack spaces were used both for the execution of the SPMD images as well as the dynamic memory allocation. Our virtual architecture parallel library routines heavily rely on dynamic data allocation. This is because handles and all their constituents are dynamic entities. The size of the mapping table and the required data buffer size are not known until the problem instance size is broadcast to all the physical processors. Upon receiving the size of the problem, all the SPMD images allocate the required handle from their stack. They will then compute the granularity and the mapping table. At this point, the amount of space per physical processor is known to all the physical processors. This space is required by the virtual processors or threads in a library call.

Since most of the data is dynamic in the proposed library implementation, expression of the local computation may be cumbersome, however it should not be a major issue as for a library routine it is a one-time implementation and exhaustive testing must be performed before usage. Reference to these distributed data cannot use indexing if the dimension is higher than one. All the computation in the library must be described using pointer arithmetic using the grain size of the computation. This will cause the code to be unreadable. Use of macros can greatly facilitate manipulation of the distributed data.

As it was described in the previous chapters, data must be broken up and distributed among the virtual processors prior to the actual computation. The data is generally input by a processor provided with the I/O subsystem. This data is stored in either row major or column major form which does not accommodate the need of block matrix algorithms. In block matrix algorithms, data is needed in form of blocks, the size of which is not known until runtime. Therefore, the data storage pattern must be changed in order to distribute the data onto the processors.

With the support of many-to-one mapping, many partitions of the distributed data may be sent to the same processor. These partitions must be managed locally to be provided to the thread that owns them. Distributed data at each processor is expanded by a factor of the number of threads. Each thread is then given a local index in addition to the global virtual identifier assigned by the mapping function. A local mapping from the global thread identifier to the local thread identifier is used to access the data that corresponds to a thread. The mapping may change from one phase to another, therefore the local mapping must be redefined at the beginning of each parallel phase.

The library routines are based on virtual topologies such as lines, rings, meshes, tori, and hypercubes. It is highly probable that the physical system attributes do not match those of the library routine. A mapping of the virtual processors to the physical processors is decided at the beginning of each parallel phase. This mapping table is simultaneously calculated by all the physical processors and is kept in the handle for the library call. Access to the local data by the threads, intraprocessor communication, and interprocessor communication must be performed using the defined mapping. This mapping is dynamic, however it will not change in the course of a library routine execution. Upon exiting one call and entering another, a new mapping is decided. Figure 55 shows an example

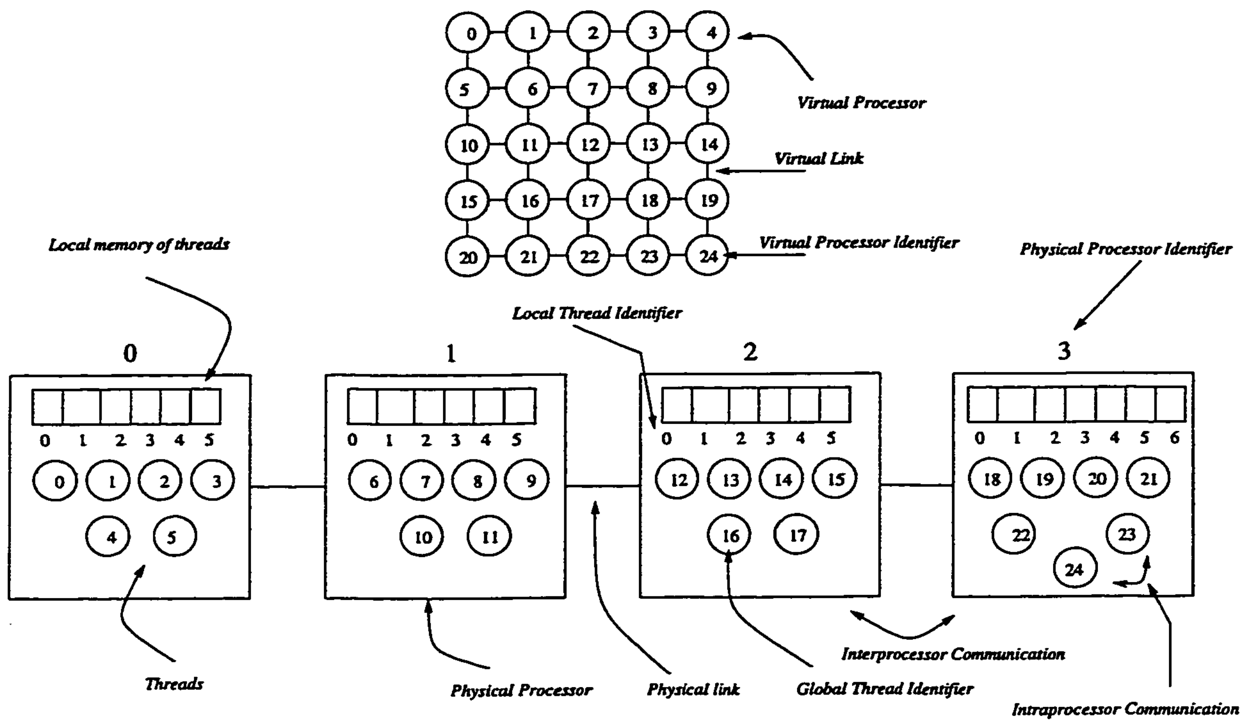


Figure 55: Example mapping of a mesh to a line

mapping of a mesh to a line. The threads are numbered from 0 to 24. This number correspond to the index of the mapping table. Since we have a global view in the simulator, the index is used as the global virtual processor identifier. We do not have an actual thread identifier in the prototype implementation. However, on an actual system, using a threading library, the need for such an identifier in the mapping table has been discussed (see Section 3.1 on Page 45).

Our multicomputer simulator is based on packet switching as well as wormhole routing. Literature such as [58] propose optimal mapping functions among various topologies. These mapping functions ensure the neighboring virtual processors are mapped as close as possible in the target topology. The time to communicate between two nodes is proportional to the distance between the nodes in packet switching networks. In wormhole routed networks [15], the communication time is independent of the distance between the nodes, however reduction of this distance alleviates the contention in the network [62].

5.2 The Multicomputer Simulator

The original multicomputer simulator, developed in our group, was modified to better suit the needs of experimentation on the parallel libraries. One of the major changes was addition of actual imitation of a multicomputer by removing the logical layer and allocating a stack to each physical processor. The multicomputer simulator mimics the multicomputer model by allocating a stack and a program counter to each processor. Further modification was done so that upon startup of an SPMD program all processors get initialized to run the same code (SPMD model). Figure 56 shows

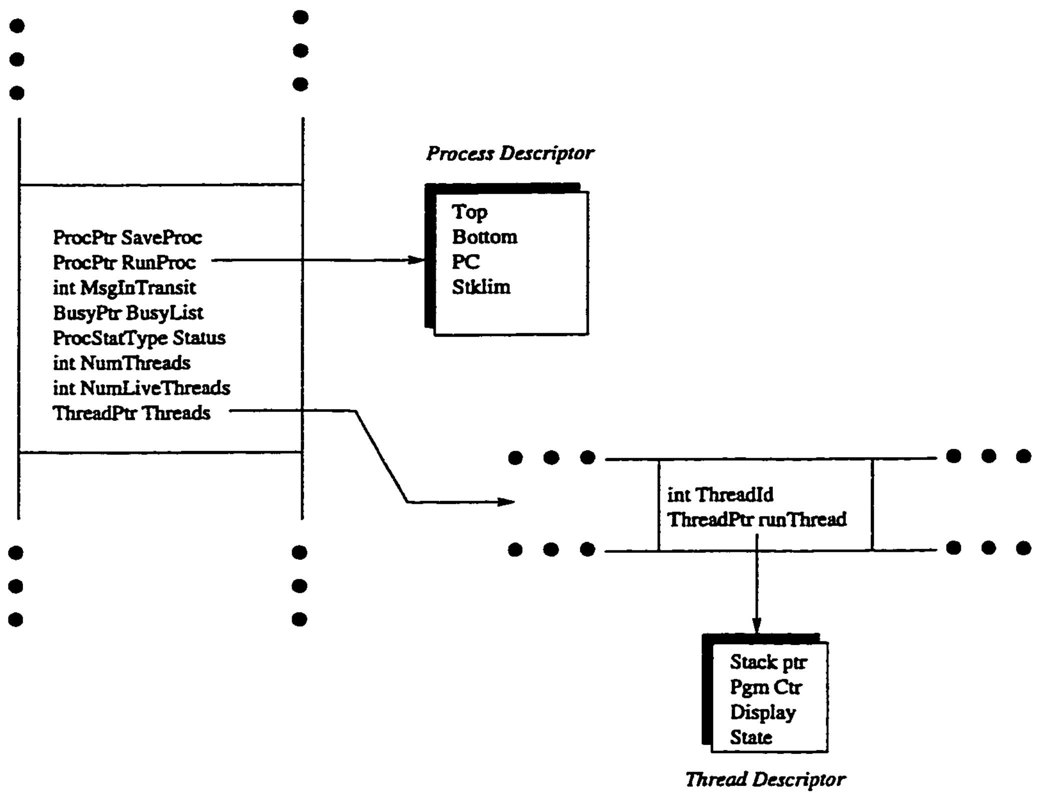


Figure 56: Physical processor and thread table

an entry of a physical processor table. The vertical table on the left of the figure is the physical processor table. There is one entry for each physical processor. The **SaveProc** field is a pointer to a process descriptor structure. When the physical processor is running threads, the main process descriptor which is pointed to by the **RunProc** field is stored in the **SaveProc** field. The **RunProc** field will then point to the first thread in the thread table (the horizontal table in the figure). The thread table, **Threads**, is an entry of the physical processor table. The structure used for threads is similar to the process descriptor. Since these structures are aliases, the **RunProc** pointer can point to the process descriptor or a thread descriptor transparent to the simulator. The thread table index is the local thread identifier and the first field holds the global thread identifier. The physical processor may be running a main process (main) or a set of parallel threads that are created as a result of a library call. At the beginning of a library call the main process is blocked. This process will be scheduled when all the threads are done with their execution. The thread table contains an identifier which corresponds to the virtual processor represented by the thread. Each physical processor table entry has a field for the total number of threads and the number of threads that are running. When the last thread finishes execution, the scheduler releases the thread table, and points **RunProc** to the Process descriptor.

Figure 57 shows how the runtime stack is divided among the threads and the main process. The global region holds the handles which contain the data partitions used by each thread. All threads have access to the current handle and the history list in the physical processor through global data access.

The major difference between processes and threads is their scheduling. The simulator charges a small time for thread switching. Also, a process does not relinquish the processor once it has control unless a **Forallthread** call is entered. However, threads only relinquish the processor if they execute a **receive** which blocks.

5.2.1 Scheduling and Communication Cost Model

Our simulator has two levels of scheduling, a micro-scheduler and a macro-scheduler. The macro scheduler functions at the physical processor level. Each physical processor is given a time slice and once it has completely used the slice, it will have to wait for the next slice allotted to it in a round robin fashion. When a physical processor is granted a time slice, the micro-scheduler searches the list of threads, if any, for one that can use the slice. When the thread is found, it will be scheduled to run and will not relinquish the processor until it either runs out of the time slice or it blocks on a **receive**. This scheduling policy is justified by the fact that threads are meant to keep a low context switching overhead in the local processors. Once a physical processor has an unused time slice, any thread within that processor is equally a good candidate to use the remainder. The simulator charges a processor a small number of cycles for context switching between threads (10 cycles). Once all the processors have used their scheduled time slice, the global clock is increased by the time slice.

When a message is sent by a thread, the destination thread may reside local to that physical processor, or it may be at a distant physical processor. The former is referred to as *intraprocessor communication*, and the latter is referred to as *interprocessor communication*. The cost of

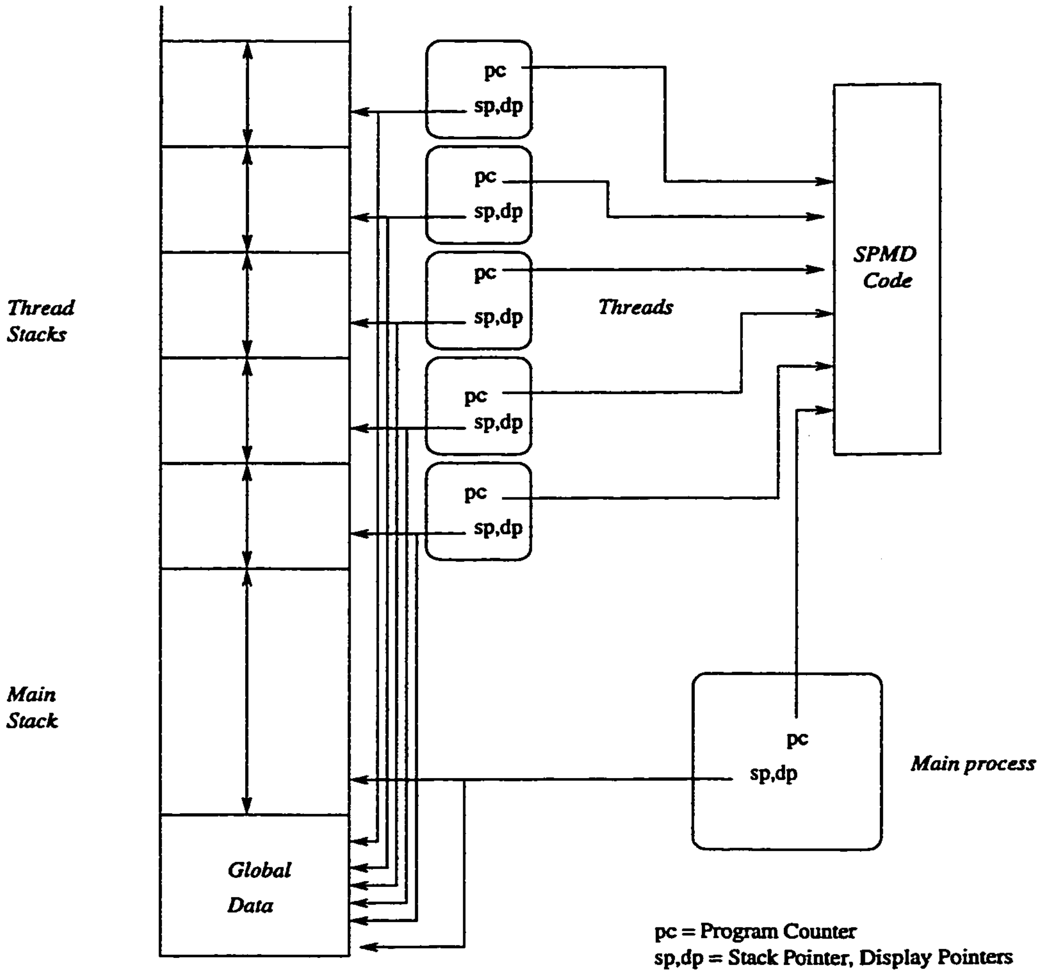


Figure 57: Physical processor and threads stack utilization

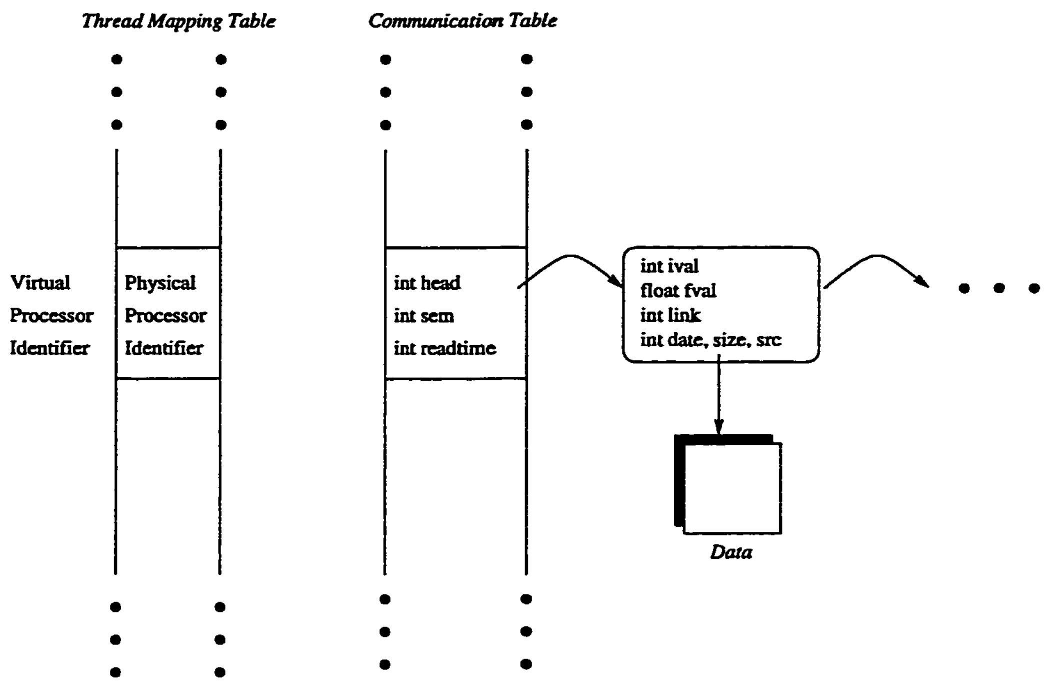


Figure 58: Physical communication structures

intraprocessor communication is basically the time to perform a local memory copy. The penalty for such communication is much lower than the interprocessor communication which is equal to the startup cost plus the time in the network. In our simulator, we have selected an intraprocessor communication cost which is proportional to the message size (2 cycles per word).

In our integrated environment, the mapping table of the currently executing parallel phase is passed to the simulator through the **ForallThread** construct. The mapping table is used by the simulator to distinguish local from nonlocal communication. Nonlocal communication operations are charged based on the routing method used. The simulator supports both packet switching and wormhole routing. The packet switching can be run in the CONGESTION ON mode where the packets occupy time slots on their path to the destination. The wormhole routing simulator is, however, based on step by step simulation of the interconnection network.

5.2.2 Communication Library

The original multicomputer simulator used an abstraction called *channels* to communicate across processes. The current multicomputers are usually provided with some communication library such as PICL. A communication library that provides the basic communication primitives was developed and integrated into the system to support the library design. The library routines are set up to communicate using the primitives provided by our communication library. We used the PICL implementation to assess the feasibility of our library design. Figure 58 shows the virtual communication structures interrelation. We use the communication table both for communication among the physical processors and among the virtual processors. When the main process is executing, each

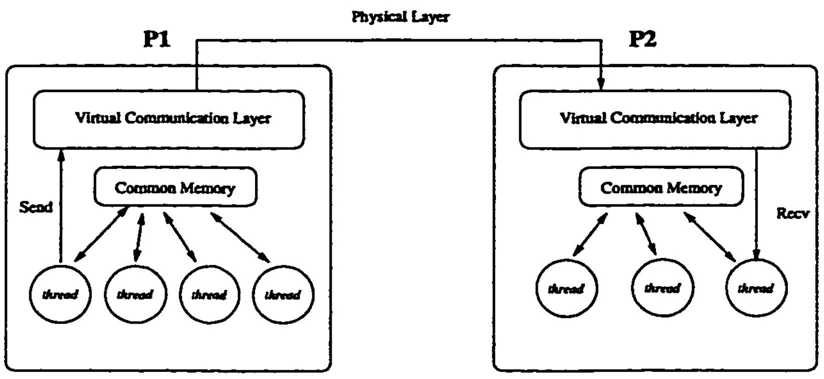


Figure 59: Intraprocessor vs. interprocessor communication

physical processor uses the entry of the communication table corresponding to its physical processor identifier. For instance, entry 3 is used by the physical processor 3. When a parallel phase is active, the entries of the virtual communication table correspond to the virtual processors. For instance, the 10th entry is used by the virtual processor with global identifier 10.

The virtual point-to-point and the physical communication routines are implemented as builtin functions. The virtual point-to-point communication routines use the mapping table to find the destination physical processor for the purpose of charging the correct communication cost. The message is, however, deposited at the appropriate entry of the virtual communication table.

The builtin functions **PHYS_SEND**, **PHYS_RECV**, and **PHYS_ABSID** were implemented in the simulator to mimic the behavior of the PICL communication library. The **PHYS_SEND** builtin function posts a message to a channel. The size and the tag of the message along with the message buffer are placed in a queue. The **PHYS_RECV** scans the queue of a physical processor for a message with a specified tag. If the message is in the queue, it extracts the contents of the buffer and continues by giving control back to the simulator. If the message has not arrived yet, it gives control to the scheduler which schedules the next processor (or thread) for execution. The **PHYS_ABSID** returns the absolute identifier of the currently simulated physical processor.

The cost associated with each communication primitive has been parameterized in the network layer. These parameters are currently adjusted to those of Intel Paragon, but may be easily changed to the set of values for systems listed on Page 31 in Chapter 2. Communication across threads is issued in the logical layer using the virtual communication library routines. These calls are then converted to the physical layer calls which transfer the message using our physical layer simulation of the communication library, PICL. Figure 59 shows the route for a thread to communicate with a nonlocal thread as well as how the threads use the common memory of the processor that houses them to communicate with one another.

Chapter 6

Conclusion

In this thesis we proposed a novel design of parallel virtual architecture library routines that will not only ease programming on multicomputers, but also offers superior performance to the currently known libraries. The novelty of the design lies in the self grain adjusting routines, systematic delayed mapping to improve performance, and the potential of the library routines to still deliver good performance, once ported.

The design goals have been performance, portability, and ease of use. The library routines have a sequential interface and perform data distribution management, granularity adjustment, mapping, and redistribution transparently. The library routines are self adjusting to different size and topologies of the target architecture with runtime grain adjustment. Portability across different parallel systems can be easily done by the recompilation of the mapping library, virtual communication library, the redistribution module, and the numerical library routines. Provided the necessary low level communication library and the threading library are available on the target system, the programmer can easily port all the existing applications to run on the new platform efficiently. Considering that new systems with higher performance will be built in the future, such a library system can easily benefit from the advancement in the hardware technology such as faster communication networks.

To utilize the target system properly, the attributes of the physical system must be projected in the library. The minimization functions select the degree of parallelism based on the problem size, physical system size, the communication latency, and the processor speed of the physical system.

Initial data distribution which is normally done explicitly by the programmer is encapsulated in our library routines. Maintenance of data distribution is not only useful for initial data layout, but also is used for data redistribution. Data redistribution functionality is an important optimization supported in our parallel library routines. The compiler produces the necessary code to detect the data that is subject to redistribution. It will then execute the appropriate communication operations to prepare the data for the upcoming phase.

There are several topics which are interesting research subjects for future research. One is consideration of data distribution time as part of the library routine execution time. This may affect the selection of the best grain size. Another interesting area is the use of parallel I/O in the initial distribution phase of the library call. Initial data distribution is a major bottleneck in the parallel execution time of a library call. Parallel I/O can significantly reduce this overhead. We have, in this thesis, considered the parallel library calls in isolation. A global analysis of the program and possibility of fixing the grain size of the data for the course of execution of the program may reduce the overhead of data redistribution. This can also be addressed under constrained minimization, where a parallel call fixes the granularity which will subsequently affect the grain size adjustment for a subsequent phase. The library design in this thesis has the potential for this enhancement since the information about all the previous parallel phases is maintained in the physical processors.

The collective virtual communication routines, described in this thesis, do not take into account the mapping of the virtual processors to the physical processors. These routines may be enhanced so that instead of sending the message to the virtual processor half way around the ring, the virtual processor with the least physical distance is used. This enhancement can improve the performance of these collective communication routines.

In our selection of the best grain size, we assume that the problem instance size is such that the virtual architecture can be reduced in size to equal the physical architecture size. If this is not the case, the grain size adjustment phase must decide between two granularity values. One of these leaves some physical processors idle during execution of the library routine. The other will keep all physical processors busy with some of these processors being assigned two virtual processors. The cost model for a parallel library routine can be enhanced to take into consideration this type of scenarios and select the better one of the two grain sizes.

The library design in this thesis has a sequential call interface. The library routines are called from sequential programs. This selection of the language eases use of multicomputers at the cost of a high overhead at runtime, specially in the data redistribution phases. Runtime code generation is a technique normally used to generate symbolic expressions at compile time which will then be fully evaluated at runtime. This technique greatly reduces the runtime overhead by performing much of the computation at compile time. Although the mapping functions of the phases are not known at compile time, the data layout of the library routines may be made available to the compiler in symbolic forms.

The compilation process described in this thesis has covered all aspects of the translation, however existing user programs may contain awkward code that manipulates distributed data and cannot be easily translated to an efficient SPMD form. For instance, loops that manipulate the contents of the poly variables and may cause a large number of broadcasts to be generated by the source to source transformer. These instances of reads of poly variables do not need to be broadcast to all other processors. These pieces of code can be identified and transformed so that they can be translated to an efficient SPMD form.

Appendix A

A Programming Example: Matrix Multiplication

Following is the outline of the complete library routine for the torus based matrix multiplication used throughout the thesis. The definition of a handle is given in chapter 3.

```
void
matmpy_init_handle(LibHandlePtr handle)
{
int i;

   handle->NumParams = MATMPY_NPARAM-1;
   handle->ClassParams = NULL;
   handle->ClassParams = (enum ParamType *)
                         malloc((MATMPY_NPARAM-1)*sizeof(int));

   handle->ClassParams[0] = INPARAM;
   handle->ClassParams[1] = INPARAM;
   handle->ClassParams[2] = OUTPARAM;

   handle->architecture = NULL;
   handle->architecture = (Architecture *)
                           malloc(sizeof(Architecture));
   handle->mapping = NULL;
   handle->mapping = (Mapping *) malloc(sizeof(Mapping));

/*
 * The following are related to the data items in the parameters.
 * The number of allocated structures is the same as the number of
 * parameters that are non-scalar.
 */

   handle->contraction = NULL;
   handle->contraction = (Blocking *)
           malloc((MATMPY_NPARAM-1) * sizeof(Blocking));
   handle->init_layout = NULL;
   handle->init_layout = (Distribution *)
           malloc((MATMPY_NPARAM-1) * sizeof(Distribution));
   handle->fin_layout = NULL;
   handle->fin_layout =  (Distribution *)
           malloc((MATMPY_NPARAM-1) * sizeof(Distribution));
   handle->local_data = NULL;
   handle->local_data =  (UserData *)
           malloc((MATMPY_NPARAM-1) * sizeof(UserData));
}

/*
 * Compute execution time.
 */

int
compute_time(int n,int p,int g)
{
int etime;
float sqrtp;

   sqrtp = pow((double)p,0.5);
   if(n/g > sqrtp)
   {
      etime = n/g*(g*n*n*TC/p+2*n*TS/(g*sqrtp) +
                        (2*n*n/p-2*n*g/sqrtp)*TM);
   }
   else
   {
      etime = n/g*(pow((double)g,3.0)*TC + 2*TS);
   }
   return etime;
}
```

```
/*
 * Minimize the grain adjustment function.
 */

int
minimize(int n,int p)
{
int i,g, exectime, minexec, gmin;

  minexec = INFINITY;
  gmin = -1;
  for(i=0;i<=(int)(log10((double)n)/log10(2.0));i++)
  {
     g = (int)pow(2.0,(double)i);
     exectime = compute_time(n,p,g);
     if(exectime < minexec)
     {
       gmin = g;
       minexec = exectime;
     }
  }
  return(gmin);
}

/*
 * Initialize components of the library.  Only one processor
 * knows the size of the data instance.  This processor must
 * broadcast the info to all other processors.
 */

void
matmpy_init_arch(LibHandlePtr handle,int *A,int *B, int *C,int size)
{
int i, src, nproc, d[2];
int xsize;

/*
 * Physical processor 0 sends the size to all the
 * other processors so that they can all set up the
 * handles simultaneously.
 */

   nproc = phys_nproc();
   if(phys_abs_id() == 0)
   {
      for (i=0; i<nproc; i++)
         send(&size,sizeof(int),MATMPY_SIZE_TAG,i);
   }

   recv(&size,sizeof(int),MATMPY_SIZE_TAG,&src);

/*
 * All processors fill in the info in the handle.
 */

   handle->architecture->topology = TORUS2;
   handle->architecture->dimensionality = 2;
   handle->architecture->fine_dims = NULL;
   handle->architecture->fine_dims = (int *)malloc(2);
   handle->architecture->period = NULL;
   handle->architecture->period = (int *) malloc(2);
   handle->architecture->coarse_dims = NULL;
   handle->architecture->coarse_dims = (int *) malloc(2);

/*
 * The library routine is for Square matrix
 * multiplication.  Keep information on the
 * fine grain architecture.
 */

   handle->architecture->fine_dims[0] = size;
   handle->architecture->fine_dims[1] = size;
   handle->architecture->fine_nproc = size*size;

   d[0] = d[1] = handle->architecture->fine_dims[0]/
         minimize(handle->architecture->fine_dims[0], nproc);

   handle->architecture->coarse_dims[0] = d[0];
   handle->architecture->coarse_dims[1] = d[1];
   handle->architecture->period[0] = size/d[0];
```

```
    handle->architecture->period[1] = size/d[1];
    handle->architecture->coarse_nproc =
                   handle->architecture->coarse_dims[0] *
                   handle->architecture->coarse_dims[1];
}
/*
 * Allocates necessary storage for local data.
 */

void
matmpy_allocate_local(int * A, int * B, int *C,
                          int  size, LibHandlePtr handle)
{
int i;
int nbrofmythreads;
int myid;

/*
 * Allocate local data partitions.  Each processor must allocate
 * partitions based on the mapping and the needs of the algorithm.
 * Also note that processors that are not mapped any virtual
 * processors do not need any local data.
 */

   myid = phys_abs_id();
   nbrofmythreads = 0;
   for(i=0;i< handle->architecture->coarse_nproc; i++)
      if(handle->mapping->map_table[i] == myid) nbrofmythreads++;

   for(i=0; i<MATMPY_NPARAM-1; i++)
   {
       handle->local_data[i].redistributed = FALSE;
       handle->local_data[i].stale = FALSE;
       handle->local_data[i].dims = NULL;
       handle->local_data[i].dims = (char *) malloc (2);
       handle->local_data[i].dim = 2;
       handle->local_data[i].dims[0] = handle->contraction[i].period[0];
       handle->local_data[i].dims[1] = handle->contraction[i].period[1];
       handle->local_data[i].size = handle->contraction[i].period[0] *
                                     handle->contraction[i].period[1];
```

```
/*
 * We must allocate space for all the threads that are local to
 * this processor.  The number of threads per processor depends
 * on the contraction factor, the granularity of the computation,
 * and the mapping function.
 */

       handle->local_data[i].data= NULL;
       handle->local_data[i].data=(char *)
              malloc (nbrofmythreads*handle->local_data[i].size);
   }
   handle->local_data[0].global_data = &A;
   handle->local_data[1].global_data = &B;
   handle->local_data[2].global_data = &C;
}

/*
 * Determines the data contraction factor for the instance.
 */

void
matmpy_contraction(LibHandlePtr handle)
{
int i;

/*
 * For all data determine handle->contraction[i].period[*]
 * the array period determines the blocking of data.  All
 * regular blockings can be described using the period.
 * Period[i] is the period for the ith dimension, that is
 * after how many elements a new block begins.
 */

   for(i=0; i<MATMPY_NPARAM-1; i++)
   {
     handle->contraction[i].period = NULL;
     handle->contraction[i].period = (int *) malloc(2);
     handle->contraction[i].fine_dims = NULL;
     handle->contraction[i].fine_dims = (int *) malloc(2);
```

```
      handle->contraction[i].coarse_dims = NULL;
      handle->contraction[i].coarse_dims = (int *) malloc(2);

      handle->contraction[i].dim = 2;
      handle->contraction[i].fine_nblock =
                          handle->architecture->fine_nproc;
      handle->contraction[i].fine_dims[0] =
                          handle->architecture->fine_dims[0];
      handle->contraction[i].fine_dims[1] =
                          handle->architecture->fine_dims[1];
      handle->contraction[i].period[0] =
                          handle->architecture->period[0];
      handle->contraction[i].period[1] =
                          handle->architecture->period[1];
      handle->contraction[i].coarse_nblock =
                          handle->architecture->coarse_nproc;
      handle->contraction[i].coarse_dims[0] =
                          handle->architecture->coarse_dims[0];
      handle->contraction[i].coarse_dims[1] =
                          handle->architecture->coarse_dims[1];
    }
}

/*
 * Determine the mapping.
 */

void
matmpy_mapping(LibHandlePtr handle)
{
    handle->mapping->map_table = NULL;
    handle->mapping->map_table = (int *)
        malloc(handle->architecture->coarse_nproc);
    TorusToMesh(handle);
}

/*
 * Fills in the information on data layout. This
 * information describes, for each parameter, the
 * initial and final data distribution.  These are
 * functions from data block numbers to the virtual
 * processors.
 */

void
matmpy_data_layout(LibHandlePtr handle)
{
int i,j, r;

  r = handle->architecture->coarse_dims[0];

/*
 * Determine the initial data layout for all in parameters. For
 * each data block determine its location in the virtual
 * architecture when the computation commences.
 */

  handle->init_layout[0].data_map = NULL;
  handle->init_layout[0].data_map = (int *)
            malloc(handle->architecture->coarse_nproc);
  handle->init_layout[1].data_map = NULL;
  handle->init_layout[1].data_map = (int *)
            malloc(handle->architecture->coarse_nproc);
  handle->init_layout[2].data_map = NULL;

/*
 * Initial data layout.  Some parameters do not have initial data
 * layout(such as the third parameter of matmpy).
 */

  for(i=0; i<handle->architecture->coarse_dims[0]; i++)
      for(j=0; j<handle->architecture->coarse_dims[0]; j++)
          handle->init_layout[0].data_map[i*r+j] =
                                   i*r+(j-i<0?j-i+r:j-i);


  for(i=0; i<handle->architecture->coarse_dims[0]; i++)
      for(j=0; j<handle->architecture->coarse_dims[0]; j++)
          handle->init_layout[1].data_map[i*r+j] =
                                   (i-j<0?i-j+r:i-j)*r+j;
/*
 * Determine the final data layout for all the parameters.
 * For each block, determine its location in the virtual
 * architecture after the computation is done.
 */

  handle->fin_layout[0].data_map = NULL;
  handle->fin_layout[0].data_map = (int *)
         malloc(handle->architecture->coarse_nproc);
  handle->fin_layout[1].data_map = NULL;
  handle->fin_layout[1].data_map = (int *)
         malloc(handle->architecture->coarse_nproc);
  handle->fin_layout[2].data_map = NULL;
  handle->fin_layout[2].data_map = (int *)
         malloc(handle->architecture->coarse_nproc);

  for(i=0; i<handle->architecture->coarse_dims[0]; i++)
      for(j=0; j<handle->architecture->coarse_dims[0]; j++)
          handle->fin_layout[0].data_map[i*r+j] =
                                i*r+(j-i<0?j-i+r:j-i);

  for(i=0; i<handle->architecture->coarse_dims[0]; i++)
      for(j=0; j<handle->architecture->coarse_dims[0]; j++)
          handle->fin_layout[1].data_map[i*r+j] =
                                (i-j<0?i-j+r:i-j)*r+j;

  for(i=0; i<handle->architecture->coarse_dims[0]; i++)
      for(j=0; j<handle->architecture->coarse_dims[0]; j++)
          handle->fin_layout[2].data_map[i*r+j] = i*r+j;

}

/*
 * Called by each processor to send the final results
 * to the processor with I/O capability.
 */

void
matmpy_send_final(LibHandlePtr handle)
```

```
{
int mylocaltid, j;

/*
 * Processors send Final data to 0.
 */

    mylocaltid = mylocalthreadid();
    vsend(mylocaltid*handle->local_data[2].size +
                 handle->local_data[2].data,
                 handle->local_data[0].size, MATMPY_C_TAG, 0, handle);

}

/*
 * Called by each processor to get the initial data.
 */

void
matmpy_get_initial(LibHandlePtr handle)
{
int src, j;
int mylocaltid;
int myglobaltid;

/*
 * Get Initial data for the computation
 * into the user data field of parameters 0 and 1.
 * In many to one mapping the local thread id must
 * be used to offset into the data.
 */

    mylocaltid = mylocalthreadid();

    if(!handle->local_data[0].redistributed)
        vrecv(mylocaltid*handle->local_data[0].size +
                    handle->local_data[0].data,
                    handle->local_data[0].size,
                    MATMPY_AINIT_TAG, &src, handle);
```

```
/*
 * If data is already redistributed, do not attempt to receive it.
 */

    if(!handle->local_data[1].redistributed)
       vrecv(mylocaltid*handle->local_data[1].size +
                  handle->local_data[1].data,
                  handle->local_data[1].size,
                  MATMPY_BINIT_TAG, &src, handle);
}

/*
 * The actual parallel computation.
 */

void
multiply_partitions(int *a, int *b, int *c, int size)
{
int i, j, k;

    for(i=0; i<size; i++)
        for(j=0; j<size; j++)
        {
            for(k=0; k<size; k++)
               *(c+i*size+j) += *(a+i*size+k) * *(b+k*size+j);
        }
}
void
matmpy_local(LibHandlePtr handle)
{
int i,j;
int myglobaltid, mylocaltid,myrow, mycol, left, above, src;

/*
 * Get my physical id and find my virtual id from the map table.
 * My left and above neighbors are then found and used to communicate
 * through vsend
 */

   mylocaltid = mylocalthreadid();
```

```
    myglobaltid = myglobalthreadid();
    myrow = myglobaltid/handle->architecture->coarse_dims[0];
    mycol = myglobaltid%handle->architecture->coarse_dims[0];
    left = myrow * handle->architecture->coarse_dims[0] +
         (mycol==0?handle->architecture->coarse_dims[0]-1:mycol-1);
    above = (myrow==0?handle->architecture->coarse_dims[0]-1:myrow-1) *
         handle->architecture->coarse_dims[0] + mycol;

    matmpy_get_initial(handle);

    for(i=0;i<handle->local_data[2].size; i++)
        *(i+mylocaltid*handle->local_data[2].size+
                    handle->local_data[2].data) = 0;

    for(i=0;i<handle->architecture->coarse_dims[0];i++)
    {
        vsend(mylocaltid*handle->local_data[0].size+handle->local_data[0].data,
                  handle->local_data[0].size, MATMPY_A_TAG, left, handle);
        vsend(mylocaltid*handle->local_data[1].size+handle->local_data[1].data,
                  handle->local_data[1].size, MATMPY_B_TAG, above, handle);
        multiply_partitions(
(int *) (mylocaltid*handle->local_data[0].size+handle->local_data[0].data),
(int *) (mylocaltid*handle->local_data[1].size+handle->local_data[1].data),
(int *) (mylocaltid*handle->local_data[2].size+handle->local_data[2].data),
handle->contraction[2].period[0]);

        vrecv(mylocaltid*handle->local_data[0].size+handle->local_data[0].data,
              handle->local_data[0].size, MATMPY_A_TAG, &src, handle);
        vrecv(mylocaltid*handle->local_data[1].size+handle->local_data[1].data,
              handle->local_data[1].size, MATMPY_B_TAG, &src, handle);
    }

    matmpy_send_final(handle);
}
/*
 * Following two routines are called by the processor
 * executing the main thread.  Sends the initial data
 * to the processors after performing blocking (contraction)
 * of data.
 */
```

```
void
contract_data(int *arr,LibHandlePtr h)
{
int *tmp, *cur1, *cur2, i, j, k, p0, p1, xblk, yblk;

  tmp = NULL;
  tmp = (int *) malloc (h->architecture->fine_nproc);

  for(i=0,cur1=tmp,cur2=arr; i<h->architecture->fine_nproc;i++)
      *cur1++ = *cur2++;

  p0 = h->contraction[0].period[0];
  p1 = h->contraction[0].period[1];

  for(i=0, cur2=arr;i<h->contraction[0].coarse_nblock;i++)
  {
     xblk = (int) (i/h->contraction[0].coarse_dims[0]);
     yblk = i%h->contraction[0].coarse_dims[0];
     for(j=xblk*p0; j<(xblk+1)*p0;j++)
         for(k=yblk*p1; k<(yblk+1)*p1;k++)
            *cur2++ = *(tmp+j*h->contraction[0].fine_dims[0]+k);
  }
  free(tmp);
}
void
matmpy_distribute_initial(int * A, int * B, int *C,
                              int size, LibHandlePtr handle)
{
int myid, mytid, i, j, k, r;
int *tempA, *tempB;

/*
 * Block the data based on the values in period[*]
 * Send the data to the procs based on the map table
 * and their init_data_layout.
 */

    if(!handle->local_data[0].redistributed)
    {
```

```
   tempA = NULL;
   tempA = (int *) malloc (size * size);
   for(i=0;i<size*size;i++)
     *(tempA+i) = *(A+i);
   contract_data(tempA,handle);
}

if(!handle->local_data[1].redistributed)
{
   tempB = NULL;
   tempB = (int *) malloc (size * size);
   for(i=0;i<size*size;i++)
     *(tempB+i) = *(B+i);
   contract_data(tempB,handle);
}

myid = phys_abs_id();
mytid = myglobalthreadid();
if(mytid==0)
{
   r = handle->architecture->coarse_dims[0];
   if(!handle->local_data[0].redistributed)
       for(i=0; i<handle->architecture->coarse_dims[0]; i++)
           for(j=0; j<handle->architecture->coarse_dims[1]; j++)
           {
               vsend((char *)(tempA+(i*r+j)*handle->local_data[0].size),
                   handle->local_data[0].size,MATMPY_AINIT_TAG,
                    i*r+(j-i<0?j-i+r:j-i),handle);
           }

   if(!handle->local_data[1].redistributed)
       for(i=0; i<handle->architecture->coarse_dims[0]; i++)
           for(j=0; j<handle->architecture->coarse_dims[0]; j++)
           {
               vsend((char *)(tempB+(i*r+j)*handle->local_data[1].size),
                   handle->local_data[1].size,MATMPY_BINIT_TAG,
                    (i-j<0?i-j+r:i-j)*r+j,handle);
           }
 }
```

```
      if(!handle->local_data[0].redistributed) free(tempA);
      if(!handle->local_data[1].redistributed) free(tempB);
}
/*
 * These two routines are called by the processor
 * executing the main thread.  Receive the final
 * data from the processors and expands the data
 * (opposite of contraction).
 */

expand_data(int *arr, LibHandlePtr h)
{
int *tmp, *cur1, *cur2, i, j, k, p0, p1, xblk, yblk;

  tmp = NULL;
  tmp = (int *) malloc (h->architecture->fine_nproc);

  for(i=0,cur1=tmp,cur2=arr; i<h->architecture->fine_nproc;i++)
      *cur1++ = *cur2++;

  p0 = h->contraction[0].period[0];
  p1 = h->contraction[0].period[1];

  for(i=0,cur1=tmp;i<h->contraction[0].coarse_nblock;i++)
  {
     xblk = i/h->contraction[0].coarse_dims[0];
     yblk = i%h->contraction[0].coarse_dims[0];
     for(j=xblk*p0; j<(xblk+1)*p0;j++)
         for(k=yblk*p1; k<(yblk+1)*p1;k++)
            *(arr+j*h->contraction[0].fine_dims[0]+k) = *cur1++ ;
  }
  free(tmp);
}

/*
 * Called by the host processor to accumulate the results.
 */

void
matmpy_accumulate_final(int *A, int *B, int * C,
                         int size, LibHandlePtr handle)
{
int src, i, j, *tmp, *cur1, *cur2;

       tmp = NULL;
       tmp = (int *) malloc (handle->local_data[2].size);
       for(i=0; i<handle->architecture->coarse_nproc; i++)
       {

           vrecv(tmp,handle->local_data[0].size,
                  MATMPY_C_TAG, &src,handle);

           cur1 = C+src*handle->local_data[2].size;
           cur2 = tmp;
           for(j=0;j<handle->local_data[2].size;j++) *cur1++ = *cur2++;
       }

       /*
        * Data must be transformed from block form to contiguous form.
        */

       expand_data(C, handle);
}
```

Appendix B

Mapping Library

This appendix outlines some of the mapping function called from the SPMD program.

```
/************************************
 * Mapping Library.                *
 ***********************************/

int TorusToMesh(LibHandlePtr h)
{
int i, j, phyproc;
int phydims[2];
int perproc[2];
int d,mi,mj;

/*
 * Number of coarse grain virtual procs is a multiple
 * of number of number of physical procs.
 */

    phyproc=phys_nproc();
    phys_dims(phydims);

#ifdef IDENTITY
    if(phyproc >= h->architecture->coarse_nproc)
    {

        for(i=0;i<h->architecture->coarse_dims[0];i++)
            for(j=0;j<h->architecture->coarse_dims[1];j++)
            {
              h->mapping->map_table[i*h->architecture->coarse_dims[0]+j] =
                i*phydims[0]+j;
            }
    }
    else
    {
        perproc[0]= h->architecture->coarse_dims[0]/phydims[0];
        perproc[1]= h->architecture->coarse_dims[1]/phydims[1];
        for(i=0;i<h->architecture->coarse_dims[0];i++)
            for(j=0;j<h->architecture->coarse_dims[1];j++)
            {
              h->mapping->map_table[i*h->architecture->coarse_dims[0]+j] =
                 floor((double)i/perproc[0])*phydims[1] +
                      floor((double)j/perproc[1]);
            }
    }
#endif
#ifdef OPTIMAL

        d=h->architecture->coarse_dims[0];
        for(i=0;i<h->architecture->coarse_dims[0];i++)
            for(j=0;j<h->architecture->coarse_dims[1];j++)
            {
              if((i<d/2) && (j<d/2))
              {
                  mi = 2*i;
                  mj = 2*j;
              }
              else if((i<d/2) && (j>=d/2))
              {
                  mi = 2*i;
                  mj = 2*d-2*j-1;
              }
              else if((i>=d/2) && (j<d/2))
              {
                  mi = 2*d-2*i-1;
                  mj = 2*j;
              }
              else if((i>=d/2) && (j>=d/2))
              {
                  mi = 2*d-2*i-1;
                mj = 2*d-2*j-1;
            }
            h->mapping->map_table[i*h->architecture->coarse_dims[0]+j] =
               mi*phydims[0]+mj;
          }
#endif
#ifdef CONTENTION

        d=h->architecture->coarse_dims[0];
        for(i=0;i<h->architecture->coarse_dims[0];i++)
            for(j=0;j<h->architecture->coarse_dims[1];j++)
            {
              if((i/2*2==i) && (j/2*2==j))
              {
                  mi = i/2;
                  mj = j/2;
              }
              else if((i/2*2==i) && (j/2*2!=j))
              {
                  mi = i/2;
                  mj = d/2+j/2;
              }
              else if((i/2*2!=i) && (j/2*2!=j))
              {
                  mi = d/2+i/2;
                  mj = d/2+j/2;
              }
              else if((i/2*2!=i) && (j/2*2==j))
              {
                  mi = d/2+i/2;
                  mj = j/2;
              }
              h->mapping->map_table[i*h->architecture->coarse_dims[0]+j] =
                 mi*phydims[0]+mj;
            }
#endif
}
```

Appendix C

Virtual Communication Library

This appendix outlines the virtual communication routines.

```
/************************************
 * Virtual Communication Library.   *
 ************************************/

#define BARRIER 10000
#define ENDBARRIER 20000

#define NOP  0
#define SUM  1
#define PROD 2
#define MAX  3
#define MIN  4
#define OR   5
#define AND  6
#define XOR  7

char
rfunc(char res1, char res2, int op)
{
  switch(op)
  {
    case NOP:  return res1;
    case SUM:  return res1 + res2;
    case PROD: return res1 * res2;
    case MAX:  return (res1 < res2 ? res2 : res1) ;
    case MIN:  return (res1 < res2 ? res1 : res2) ;
    case OR:   return res1 | res2;
    case AND:  return res1 & res2;
    case XOR:  return res1 ^ res2;
  }
}

/*
 * Virtual Broadcast.  Executed by all the virtual processors.
 * one to all broadcast
 */

bcast(int num, char *buf, int size, int tag, LibHandlePtr handle)
{
int i, n, half1, half2, dest;
char *buf1;
int myindex;

   if(num==1) return;
   myindex = myglobalthreadid();
   n = handle->architecture->fine_nproc;
   half1 = num/2;
   half2 = num - half1;
   dest = (myindex + half1) % n;
   buf1 = (char *) malloc (size + 1);
   for (i=1;i<=size;i++) *(buf1+i) = *(buf+i-1);
   *buf1 = half2;
   vsend(buf1, size+1,tag, dest, handle);
   bcast(half1,buf,size,tag,handle);
}

void
vbroadcast(char *buf,int size,int tag,int root, LibHandlePtr handle)
{
char * buf1;
int i, j, src, n, num;

   if(myglobalthreadid() == root)
   {
       n = handle->architecture->fine_nproc;
       bcast(n, buf, size, tag, handle);
   }
   else
   {
       buf1 = (char *) malloc (size+1);
       vrecv(buf1, size+1, tag, &src, handle);
       num = *buf1;
       for (i=1,j=0;i<=size;i++,j++) *(buf+j) = *(buf1+i);
       bcast(num, buf, size, tag, handle);
   }
}

/*
 * Combine - All to all reduction
 */

collect(int firstmember, int num,int op, char buf,
                    int tag, char *result, LibHandlePtr handle)
{
int m, n, mem1, mem2, mypid, src;
char res1, res2;

   if(num==1) {
      *result = buf;
      return;
   }

   n = handle->architecture->fine_nproc;
   m = num/2;
   mem1 = firstmember;
   mem2 = (firstmember+m) % n;
   mypid = myglobalthreadid();
   if((mypid >= mem1) && (mypid < mem2))
   {
       collect(mem1, m, op, buf, tag, &res1, handle);
       if(mypid == mem1) {
          vrecv(&res2, 1, tag, &src, handle);
          *result = rfunc(res1,res2,op);
       }
   }
   else  /* mypid in mem2 */
   {
        collect(mem2, num-m, op, buf, tag,  &res2, handle);
        if (mypid == mem2)
             vsend(&res2, 1,tag, mem1, handle);
    }
}

void
vcombine(int op, char buf, char *result,int tag, LibHandlePtr handle)
{
int firstmember, num;

    firstmember = 0;
    num = handle->architecture->fine_nproc;
    collect(firstmember, num, op, buf , tag, result, handle);
    vbroadcast(result,1,tag,firstmember, handle);
}

/*
  * All to one reduction
  */

void
vreduce(int op, char buf, char *result,int tag, int root, LibHandlePtr handle)
{
int firstmember, num;

    firstmember = root;
    num = handle->architecture->fine_nproc;
    collect(firstmember, num, op, buf ,tag , result, handle);
}

/*
 * Prefix over the processors.
 */

prefix(int firstmember, int num,int op, char buf, int tag,
                    char *result, LibHandlePtr handle)
{
int i, m, n, mem1, mem2, mypid, src;
char res1, res2;
```

```
   if(num==1) {
      *result = buf;
      return;
   }

   n = handle->architecture->fine_nproc;
   m = num/2;
   mem1 = firstmember;
   mem2 = (firstmember+m) % n;
   mypid = myglobalthreadid();
   if((mypid >= mem1) && (mypid < mem2))
   {
       prefix(mem1, m, op, buf, tag, result, handle);
       vrecv(&res2, 1, tag, &src, handle);
       *result = rfunc(*result,res2,op);
   }
   else  /* mypid in mem2 */
   {
       prefix(mem2, num-m, op, buf, tag,  &res2, handle);
       *result = res2;
       if (mypid == mem2)
       {
           for(i=0;i<m;i++)
              vsend(&res2, 1,tag, mem1+i, handle);
       }
   }
}

void
vprefix(int op, char buf, char *result, int tag, LibHandlePtr handle)
{
int firstmember, num;

    firstmember = 0;
    num = handle->architecture->fine_nproc;
    prefix(firstmember, num, op, buf , tag, result, handle);
}

/*
```

```
 * Virtual sync. Virtual barrier synchronization.
 */

void
vsync(LibHandlePtr handle)
{
int src;
char dummy, dummyres;

    vreduce(NOP,dummy,&dummyres,BARRIER,0,handle);
    vbroadcast(&dummy,1,ENDBARRIER,0,handle);
}
```

Appendix D

Redistribution Library

The following pages outline the complete redistribution library described in chapter 4. The library mainly consists of a redistribution routine called from the SPMD program and the implementation of the three possible cases of redistribution.

```
/*
 * Lowers block sizes from dims[0]xdims[1]
 * (whole block of sizexsize) down to gxg
 * Assumption is that g divides dims[0]
 * and arr is square.  target must be
 * allocated before calling this routine.
 */

void
lower_blocksize(int *arr, int dim, int *dims, int size, int g)
{
int *cur1, *cur2, *temp, i, j, k, c, xblk, yblk;

     if(dim == 1) return;

     temp = (int *) malloc(size*sizeof(int));
     for(i=0,cur1=temp,cur2=arr; i<size;i++)
         *cur1++ = *cur2++;

     c = (int) (dims[0]/g);
     for(i=0, cur2=arr;i<c*c;i++)
     {
        xblk = (int) (i/c);
        yblk = i%c;
        for(j=xblk*g; j<(xblk+1)*g;j++)
```

```
            for(k=yblk*g; k<(yblk+1)*g;k++)
               *cur2++ = *(temp+j*dims[0]+k);
      }
}

/*
 * Lift block sizes from gxg to size(whole blocks of sizexsize)
 * Assumption is that g divides size.
 * Lift block sizes from gxg to dims[0]xdims[1]
 * Assumption is that g divides dims[0]
 * and arr is square.
 */

void
lift_blocksize(int *arr, int dim, int *dims, int size, int g)
{
int *cur1, *cur2, i, j, k, xblk, yblk;
int *temp, c;


  if(dim == 1) return;

  temp = (int *) malloc(size*sizeof(int));
  for(i=0,cur1=temp,cur2=arr; i<size;i++)
      *cur1++ = *cur2++;

  c = (int) (dims[0]/g);

  for(i=0,cur1=temp;i<c*c;i++)
  {
     xblk = i/c;
     yblk = i%c;
     for(j=xblk*g; j<(xblk+1)*g;j++)
         for(k=yblk*g; k<(yblk+1)*g;k++)
            *(arr+j*dims[0]+k) = *cur1++ ;
  }
}

void redistribute_case1(LibHandlePtr src, int srcParam, int g,
                    LibHandlePtr target, int targetParam, int gp)
```

```
{
int i, j, m, nthr_src, nthr_target, nblk_target, nsubblk_target;
int blockgtid, src_glob_blkid, mypid, gsize, gpsize, k;
int dest_glob_blkid, blk, subblk;
int target_glob_blkid, first_subblkid;
int dest_vp, dest_pp, msgsrc;
int target_glob_subblkid;


    mypid = phys_abs_id();

/*
 * Find the number of blocks in src and target
 */

    for(nthr_src=0, i=0;i<src->architecture->coarse_nproc;i++)
       if(src->mapping->map_table[i] == mypid) nthr_src++;

    for(nthr_target=0, i=0;i<target->architecture->coarse_nproc;i++)
       if(target->mapping->map_table[i] == mypid) nthr_target++;

    gsize = (src->local_data[srcParam].dim==1 ? g : g*g);
    gpsize = (target->local_data[targetParam].dim==1 ? gp : gp*gp);
    k = (int) (gp/g);
    nsubblk_target = (int)(gpsize/gsize);

    j = -1;

    for(blk=0;blk<nthr_src;blk++)
    {

/* 1. Find this partitions global thread id */

       do {
          j++;
       } while(src->mapping->map_table[j]!=mypid);
       blockgtid = j;

/* 2. Scan the fin_layout for myglobal thread, index is
 *    my glob block id
 */

        m = 0;
        while(src->fin_layout[srcParam].data_map[m] != blockgtid) m++;
        src_glob_blkid = m;

/* 3. Find the target block number (The one that embodies
 *    this partition)
 */

        if(target->local_data[targetParam].dim==1)
            dest_glob_blkid = (int)(src_glob_blkid / k);
        else
            dest_glob_blkid=target->contraction[targetParam].coarse_dims[0] *
                   ((src_glob_blkid/src->contraction[srcParam].coarse_dims[0])/k) +
                   (src_glob_blkid%src->contraction[srcParam].coarse_dims[0])/k;

/* 4. Now use the target init_layout to find out
 *    which vp gets this block
 */

        dest_vp = target->init_layout[targetParam].data_map[dest_glob_blkid];

/* 5. Now find out which physical gets this */

        dest_pp = target->mapping->map_table[dest_vp];

/* 6. Construct a tag using the global block id (of g) and send */

        send(src->local_data[srcParam].data+blk*gsize,gsize,
            srcParam*src->architecture->coarse_nproc+src_glob_blkid,dest_pp);
    }

/*
 * Finished computing the destination set, and sent data.
 * Now  compute the source set and receive the data into the target.
 */

    j = -1;
    for(blk=0;blk<nthr_target;blk++)
    {
/*      1. Get from mapping table the globtid */

        do {
           j++;
        } while(target->mapping->map_table[j]!=mypid);
        blockgtid = j;

/*      2. Get from data_map the glbblkid */

        m = 0;
        while(target->init_layout[targetParam].data_map[m] != blockgtid) m++;
        target_glob_blkid = m;

/*      3. Get the starting subblock id from the larger */

        if(target->local_data[targetParam].dim==1)
            first_subblkid = target_glob_blkid * k;
        else
               first_subblkid=src->contraction[srcParam].coarse_dims[0] *
        (target_glob_blkid/target->contraction[targetParam].coarse_dims[0])*k +
        (target_glob_blkid%target->contraction[targetParam].coarse_dims[0])*k;

        for(subblk=0;subblk<nsubblk_target;subblk++)
        {


/*         4. Add subblk to it to find the tag to receive */

               target_glob_subblkid = first_subblkid + (subblk/k) *
               src->contraction[srcParam].coarse_dims[0] + subblk%k;

/*         5. Receive data */

             recv(target->local_data[targetParam].data+blk*gpsize+subblk*gsize,
             gsize, srcParam*src->architecture->coarse_nproc+
                        target_glob_subblkid, &msgsrc);

        }
        lift_blocksize(target->local_data[targetParam].data+blk*gpsize,
                       target->local_data[targetParam].dim,
                       target->local_data[targetParam].dims,
                       target->local_data[targetParam].size,
                       g);
    }
}

void redistribute_case2(LibHandlePtr src, int srcParam, int g,
                 LibHandlePtr target, int targetParam, int gp)
{
int i, j, m, nthr_src, nthr_target, nblk_target, nsubblk_src;
int blockgtid, src_glob_blkid, mypid, gsize, gpsize, k;
int blk, subblk;
int target_glob_blkid, first_subblkid;
int dest_vp, dest_pp, msgsrc;
int src_glob_subblkid;

    mypid = phys_abs_id();

/*
 * Find the number of blocks in src and target
 */

    for(nthr_src=0, i=0;i<src->architecture->coarse_nproc;i++)
       if(src->mapping->map_table[i] == mypid) nthr_src++;

    for(nthr_target=0, i=0;i<target->architecture->coarse_nproc;i++)
       if(target->mapping->map_table[i] == mypid) nthr_target++;

    gsize = (src->local_data[srcParam].dim==1 ? g : g*g);
    gpsize = (target->local_data[targetParam].dim==1 ? gp : gp*gp);
    k = (int) (g/gp);
    nsubblk_src = (int)(gsize/gpsize);

    j = -1;

    for(blk=0;blk<nthr_src;blk++)
    {
        lower_blocksize(src->local_data[srcParam].data+blk*gsize,
                      src->local_data[srcParam].dim,
                      src->local_data[srcParam].dims,
                      src->local_data[srcParam].size,
                      gp);
/* 1. Find this partitions global thread id */

        do {
           j++;
        } while(src->mapping->map_table[j]!=mypid);
        blockgtid = j;

/* 2. Scan the fin_layout for myglobal thread, index is my glob block id */

        m = 0;
        while(src->fin_layout[srcParam].data_map[m] != blockgtid) m++;
        src_glob_blkid = m;

/* 3. Get the starting subblock id from the larger */

        if(src->local_data[srcParam].dim==1)
            first_subblkid = src_glob_blkid * k;
        else
              first_subblkid=target->contraction[targetParam].coarse_dims[0] *
        (src_glob_blkid/src->contraction[srcParam].coarse_dims[0])*k +
        (src_glob_blkid%src->contraction[srcParam].coarse_dims[0])*k;

        for(subblk=0;subblk<nsubblk_src;subblk++)
        {
/* 4. Add subblk to it to find the tag to receive */

              src_glob_subblkid = first_subblkid +
                     (subblk/k) * target->contraction[targetParam].
                            coarse_dims[0] + (subblk%k);

/* 5. Now use the target init_layout to find out which vp gets this block */

              dest_vp = target->init_layout[targetParam].
                            data_map[src_glob_subblkid];

/* 6. Now find out which physical gets this */
```

```
                 dest_pp = target->mapping->map_table[dest_vp];


/* 7. Send data */

                 send(src->local_data[srcParam].data+blk*gsize+subblk*gpsize,
                        gpsize, targetParam*target->architecture->
                                    coarse_nproc+src_glob_subblkid, dest_pp);
          }
    }

    j = -1;
    for(blk=0;blk<nthr_target;blk++)
    {
/*       1. Get from mapping table the globtid */

         do {
            j++;
         } while(target->mapping->map_table[j]!=mypid);
         blockgtid = j;

/*       2. Get from data_map the glbblkid */

         m = 0;
         while(target->init_layout[targetParam].data_map[m] != blockgtid) m++;
         target_glob_blkid = m;

/*       3. Receive data */

         recv(target->local_data[targetParam].data+blk*gpsize, gpsize,
                        targetParam*target->architecture->
                         coarse_nproc+target_glob_blkid, &msgsrc);
    }
}

int
gcd(int a,int b)
{
int m;
```

```
    if((a/b)*b == a) return b;
    else if((b/a)*a == b) return a;
    m=(a<b?a:b);
    while(m!=1)
    {
       if(((a/m)*m == a) && ((b/m)*m == b)) return m;
       else m--;
    }
    return 1;
}

void redistribute_case3(LibHandlePtr src, int srcParam, int g,
              LibHandlePtr target, int targetParam, int gp)
{
int dest_glob_blkid;
int i, j, m, nthr_src, nthr_target, nblk_target;
int  nsubblk_src, nsubblk_target;
int blockgtid, src_glob_blkid, mypid, gsize, gpsize;
int blk, subblk;
int target_glob_blkid, first_subblkid;
int dest_vp, dest_pp, msgsrc;
int src_glob_subblkid;
int target_glob_subblkid;
int gdp, gdpsize;
int kdp, kp;

    mypid = phys_abs_id();

/*
 * Find the number of blocks in src and target
 */

    for(nthr_src=0, i=0;i<src->architecture->coarse_nproc;i++)
       if(src->mapping->map_table[i] == mypid) nthr_src++;

    for(nthr_target=0, i=0;i<target->architecture->coarse_nproc;i++)
       if(target->mapping->map_table[i] == mypid) nthr_target++;

    gdp = gcd(g,gp);
    gsize = (src->local_data[srcParam].dim==1 ? g : g*g);
    gpsize = (target->local_data[targetParam].dim==1 ? gp : gp*gp);
    gdpsize = (src->local_data[srcParam].dim==1 ? gdp : gdp*gdp);
    kdp=g/gdp;
    kp=gp/gdp;
    nsubblk_src = (int)(gsize/gdpsize);
    nsubblk_target = (int)(gpsize/gdpsize);

    j = -1;
    for(blk=0;blk<nthr_src;blk++)
    {
       lower_blocksize(src->local_data[srcParam].data+blk*gsize,
                       src->local_data[srcParam].dim,
                       src->local_data[srcParam].dims,
                       src->local_data[srcParam].size,
                       gdp);

       do {
          j++;
       } while(src->mapping->map_table[j]!=mypid);
       blockgtid = j;

/* 2. Scan the fin_layout for myglobal thread, index is my glob block id */
       m = 0;
       while(src->fin_layout[srcParam].data_map[m] != blockgtid) m++;
       src_glob_blkid = m;

/* 3. Get the starting subblock id from the larger */

       if(src->local_data[srcParam].dim==1)
           first_subblkid = src_glob_blkid * kdp;
       else
              first_subblkid=src->contraction[srcParam].coarse_dims[0] *
                 kdp * (src_glob_blkid/src->contraction[srcParam].
                 coarse_dims[0]) * kdp + (src_glob_blkid%src->
                 contraction[srcParam].coarse_dims[0])*kdp;

       for(subblk=0;subblk<nsubblk_src;subblk++)
       {
              src_glob_subblkid = first_subblkid +
                    (subblk/kdp) * src->contraction[srcParam].
                      coarse_dims[0]*kdp + (subblk%kdp);

/* 4. Use grain info to find the target block number */

                dest_glob_blkid=target->contraction[targetParam].coarse_dims[0]
                * (src_glob_subblkid/(kp*target->contraction[targetParam].
                coarse_dims[0])/kp) + ((src_glob_subblkid%(kp*target->
                contraction[targetParam].coarse_dims[0]))/kp);

/* 5. Now use the target init_layout to find out which vp gets this block */

                dest_vp = target->init_layout[targetParam].
                     data_map[dest_glob_blkid];


/* 6. Now find out which physical gets this */

                dest_pp = target->mapping->map_table[dest_vp];


/* 7. Send data */

                send(src->local_data[srcParam].data+blk*gsize+subblk*gdpsize,
                    gdpsize, targetParam*target->architecture->
                    coarse_nproc*kp*kp+src_glob_subblkid, dest_pp);
        }
    }

    j = -1;
    for(blk=0;blk<nthr_target;blk++)
    {
/*      1. Get from mapping table the globtid */

        do {
           j++;
        } while(target->mapping->map_table[j]!=mypid);
        blockgtid = j;

/*      2. Get from data_map the glbblkid */
```

```
        m = 0;
        while(target->init_layout[targetParam].data_map[m] != blockgtid) m++;
        target_glob_blkid = m;

/*      3. Get the starting subblock id from the larger */

        if(target->local_data[targetParam].dim==1)
            first_subblkid = target_glob_blkid * kp;
        else
               first_subblkid=target->contraction[targetParam].coarse_dims[0] *
                  kp * (target_glob_blkid/target->contraction[targetParam].
                  coarse_dims[0]) * kp + (target_glob_blkid%target->
                  contraction[targetParam].coarse_dims[0])*kp;

        for(subblk=0;subblk<nsubblk_target;subblk++)
        {

/*          4. Add subblk to it to find the tag to receive */
               target_glob_subblkid = first_subblkid + (subblk/kp) *
               target->contraction[targetParam].coarse_dims[0]*kp + subblk%kp;

/*          5. Receive data */
              recv(target->local_data[targetParam].data+blk*gpsize+
                   subblk*gdpsize, gdpsize, targetParam*target->
                   architecture->coarse_nproc*kp*kp+target_glob_subblkid,
                   &msgsrc);

        }
        lift_blocksize(target->local_data[targetParam].data+blk*gpsize,
                     target->local_data[targetParam].dim,
                     target->local_data[targetParam].dims,
                     target->local_data[targetParam].size,
                     gdp);

    }
}

/*
 * Redistribute ith data from src to jth data in target.
 */

void redistribute_data(LibHandlePtr src,int srcParam,
               LibHandlePtr target, int targetParam)
{
int gsrc, gtarget;

    gsrc = src->local_data[srcParam].dims[0];
    gtarget = target->local_data[targetParam].dims[0];

    if((int)(gtarget/gsrc)*gsrc == gtarget)
       redistribute_case1(src,srcParam,gsrc,target,targetParam, gtarget);
    else if((int)(gsrc/gtarget)*gtarget == gsrc)
       redistribute_case2(src,srcParam,gsrc,target,targetParam,gtarget);
    else
       redistribute_case3(src,srcParam,gsrc,target,targetParam,gtarget);
}

/*
 * Identify which one of the three cases this is and
 * call the appropriate routine.
 */
```

```
void redistribute_data(LibHandlePtr src,int srcParam,
               LibHandlePtr target, int targetParam)
{
int gsrc, gtarget;

    gsrc = src->local_data[srcParam].dims[0];
    gtarget = target->local_data[targetParam].dims[0];

    if((int)(gtarget/gsrc)*gsrc == gtarget)
       redistribute_case1(src,srcParam,gsrc,target,targetParam, gtarget);
    else if((int)(gsrc/gtarget)*gtarget == gsrc)
       redistribute_case2(src,srcParam,gsrc,target,targetParam,gtarget);
    else
       redistribute_case3(src,srcParam,gsrc,target,targetParam,gtarget);
}
```

```
/*
 * Called from the SPMD program
 */

void
redistribute(LibHandlePtr handle,LibHandlePtr * history)
{
int i,j, found, * * global_data;
LibHandlePtr cur, prev;
int all_stale;

    for(i=0;i<handle->NumParams;i++)
    {
        /* If Out parameters, no distribution */

        if(handle->ClassParams[i] == OUTPARAM) continue;

        global_data = handle->local_data[i].global_data;
        prev = NULL;
        cur = * history;
        found = FALSE;
        while((! found) && (cur != NULL))
        {
           for(j=0;j<cur->NumParams;j++)
              if(cur->local_data[j].global_data == global_data)
              {
                  found = TRUE; /* found the block to redistribute */
                  break;
              }

           if(!found)
           {
               prev = cur;
               cur = cur->next;
           }
        }

        if(found)
        {
            redistribute_data(cur,j,handle,i);
            /* mark new as redistributed as valid */

            handle->local_data[i].redistributed = TRUE;

            /* mark old as invalid */

            cur->local_data[j].stale = TRUE;

            /* Must really return the local_data[j].data back to free pool */
            free(cur->local_data[j].data);

            /* If all parameters of a handle are stale, the handle can be
               removed from the history list. */

            all_stale=TRUE;
            for(j=0;j<cur->NumParams;j++)
               if(!cur->local_data[j].stale)
                   all_stale = FALSE; /* Still has valid distributed data */

            if(all_stale)
            {
               if(prev==NULL)
               {
                  *history = cur->next;
                  free(cur);
               }
               else
               {
                  prev->next = cur->next;
                  free(cur);
               }
            }

         }
         else
            handle->local_data[i].redistributed = FALSE;
      }
   }
```

Appendix E

Example User Program and Compilation

The example presented in this appendix is a user program which consists of several library calls embedded in various programming constructs. The automatically generated SPMD code of the program follows the user program.

```
#include "cpc.h"
#include "parlib.h"

void
print_matrix(int *A, int size)
{
int i, j;

   for(i=0;i<size;i++)
   {
       for(j=0;j<size;j++)
          printf("%d ",*(A+i*size+j));
       printf("\n");
   }
}

void
read_matrix(int fd, int *A, int size)
{
int i,j;

   for(i=0;i<size;i++)
   {
       for(j=0;j<size;j++)
       {
          *(A+i*size+j) = 0;  /* A Kludge to init type info in simulator */
          fscanf(fd,"%d",A+i*size+j);
       }
   }
}
main()
{
int *A, *B, *C, *D, *E;
int fd, size, i;

     /* Read A and B */
     fd = fopen("infile","r");
     fscanf(fd,"%d",&size);
     A = (int *) malloc (size*size);
     B = (int *) malloc (size*size);
     C = (int *) malloc (size*size);
     D = (int *) malloc (size*size);
     E = (int *) malloc (size*size);
     read_matrix(fd,A,size);
     read_matrix(fd,B,size);
     read_matrix(fd,C,size);

     cpcc_matmpy(A,B,C,size);
     cpcc_matadd(C,D,E,size);

     for(i=0;i<=3;i++)
         cpcc_matmpy(D,B,A,size);

     for(i=0;i<=2;i++)
     {
         cpcc_matmpy(D,B,A,size);
     }

     if(i<=5)
         cpcc_matmpy(D,B,A,size);

     if(i<=5)
     {
         cpcc_matmpy(D,B,A,size);
     }

     print_matrix(A,size);
}

/*
 * Start of the SPMD code
 */

  int  main()
 {

      int   *  A;
      int   *  B;
      int   *  C;
      int   *  D;
      int   *  E;
      int   fd;
      int   size;
      int   i;
    int myid; /* Declarations */
    int t1,t2; /* Declarations */


    lib_setup();
    myid = phys_abs_id(); /* One time initial executables*/
    if(myid==0) { /* Begin Guard */
        fd = fopen("infile", "r");
        fscanf(fd, "%d", &size);
        A = (int *  ) malloc((size * size));
        B = (int *  ) malloc((size * size));
        C = (int *  ) malloc((size * size));
        D = (int *  ) malloc((size * size));
        E = (int *  ) malloc((size * size));
        read_matrix(fd, A, size);
        read_matrix(fd, B, size);
        read_matrix(fd, C, size);
    } /* End Guard */
```

```
/* Expanded Parallel Library Call */
allocate_handle(&curhandle);
matmpy_init_handle(curhandle);
matmpy_init_arch(curhandle,A, B, C, size);
matmpy_contraction(curhandle);
matmpy_mapping(curhandle);
matmpy_allocate_local(A, B, C, size,curhandle);
matmpy_data_layout(curhandle);
redistribute(curhandle,&history);
forallthreads(curhandle->mapping->map_table;
                curhandle->architecture->coarse_nproc){
if(myglobalthreadid()==0)
    matmpy_distribute_initial(A, B, C, size,curhandle);
matmpy_local(curhandle);
if(myglobalthreadid()==0)
    matmpy_accumulate_final(A, B, C, size,curhandle);
}
add_handle_to_history(curhandle); /* After each call*/
/* Expanded Parallel Library Call */
allocate_handle(&curhandle);
matadd_init_handle(curhandle);
matadd_init_arch(curhandle,C, D, E, size);
matadd_contraction(curhandle);
matadd_mapping(curhandle);
matadd_allocate_local(C, D, E, size,curhandle);
matadd_data_layout(curhandle);
redistribute(curhandle,&history);
forallthreads(curhandle->mapping->map_table;
          curhandle->architecture->coarse_nproc){
if(myglobalthreadid()==0)
    matadd_distribute_initial(C, D, E, size,curhandle);
matadd_local(curhandle);
if(myglobalthreadid()==0)
    matadd_accumulate_final(C, D, E, size,curhandle);
}
add_handle_to_history(curhandle); /* After each call*/
for (i = 0; i <= 3; i++)
{
    /* Expanded Parallel Library Call */
    allocate_handle(&curhandle);
    matmpy_init_handle(curhandle);
    matmpy_init_arch(curhandle,D, B, A, size);
    matmpy_contraction(curhandle);
    matmpy_mapping(curhandle);
    matmpy_allocate_local(D, B, A, size,curhandle);
    matmpy_data_layout(curhandle);
    redistribute(curhandle,&history);
    forallthreads(curhandle->mapping->map_table;
                curhandle->architecture->coarse_nproc){
    if(myglobalthreadid()==0)
        matmpy_distribute_initial(D, B, A, size,curhandle);
    matmpy_local(curhandle);
    if(myglobalthreadid()==0)
        matmpy_accumulate_final(D, B, A, size,curhandle);
    }
    add_handle_to_history(curhandle); /* After each call*/
}/* Additional */
 for (i = 0; i <= 2; i++)
     {
/* Expanded Parallel Library Call */
allocate_handle(&curhandle);
matmpy_init_handle(curhandle);
matmpy_init_arch(curhandle,D, B, A, size);
matmpy_contraction(curhandle);
matmpy_mapping(curhandle);
matmpy_allocate_local(D, B, A, size,curhandle);
matmpy_data_layout(curhandle);
redistribute(curhandle,&history);
forallthreads(curhandle->mapping->map_table;
    curhandle->architecture->coarse_nproc){
if(myglobalthreadid()==0)
    matmpy_distribute_initial(D, B, A, size,curhandle);
matmpy_local(curhandle);
if(myglobalthreadid()==0)
    matmpy_accumulate_final(D, B, A, size,curhandle);
}
add_handle_to_history(curhandle); /* After each call*/
     }
 if (i <= 5)
{
```

```
/* Expanded Parallel Library Call */
allocate_handle(&curhandle);
matmpy_init_handle(curhandle);
matmpy_init_arch(curhandle,D, B, A, size);
matmpy_contraction(curhandle);
matmpy_mapping(curhandle);
matmpy_allocate_local(D, B, A, size,curhandle);
matmpy_data_layout(curhandle);
redistribute(curhandle,&history);
forallthreads(curhandle->mapping->map_table;
        curhandle->architecture->coarse_nproc){
if(myglobalthreadid()==0)
    matmpy_distribute_initial(D, B, A, size,curhandle);
matmpy_local(curhandle);
if(myglobalthreadid()==0)
    matmpy_accumulate_final(D, B, A, size,curhandle);
}
add_handle_to_history(curhandle); /* After each call*/
}
 if (i <= 5)
     {


/* Expanded Parallel Library Call */
allocate_handle(&curhandle);
matmpy_init_handle(curhandle);
matmpy_init_arch(curhandle,D, B, A, size);
matmpy_contraction(curhandle);
matmpy_mapping(curhandle);
matmpy_allocate_local(D, B, A, size,curhandle);
matmpy_data_layout(curhandle);
redistribute(curhandle,&history);
forallthreads(curhandle->mapping->map_table;
      curhandle->architecture->coarse_nproc){
if(myglobalthreadid()==0)
    matmpy_distribute_initial(D, B, A, size,curhandle);
matmpy_local(curhandle);
if(myglobalthreadid()==0)
    matmpy_accumulate_final(D, B, A, size,curhandle);
}
```

```
    add_handle_to_history(curhandle); /* After each call*/
          }
    if(myid==0) { /* Begin Guard */
     print_matrix(A, size);
    } /* End Guard */
    lib_exit();
}
```

Bibliography

[1] J. Adams and W. Brainerd. A little history and a Fortran 90 summary. *Computer Standards and Interfaces*, 18(4):279–89, August 1996.

[2] Applied Parallel Research, Sacramento, California. *FORGE High Performance Fortran*, October 1995. Available from World Wide Web.

[3] W.C. Athas and C.L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–25, August 1988.

[4] V. Bala et al. CCL: A portable and tunable collective communication library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, February 1995.

[5] F. Berman. *Experience with an Automatic Solution to the Mapping Problem.* The MIT Press, 1987.

[6] T. Brandes and D. Greco. Realization of an HPF interface to ScaLAPACK with redistributions. In *Proceedings of High Performance Computing and Networking. International Conference and Exhibition HPCN EUROPE 1996.*, pages 834–9. German Nat. Res. Center for Computing Science, Springer-Verlag, April 1996.

[7] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic data distribution in Vienna Fortran. In *Proceedings of Supercomputing Conference*, pages 284–293, 1993.

[8] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, August 1992.

[9] S.R. Chapple and L.J. Clarke. The parallel utilities library. In *Proceedings of Scalable Parallel Libraries Conference*, pages 21–30. Mississipi State University, IEEE Computer Society Press, October 1994.

[10] S. Chatterjee, J.R. Gilbert, R. Schreiber, and T.J. Shefler. Automatic distribution in HPF. In *Proceedings of The Second Workshop on Environments and Tolls for Parallel Scientific Computing.*, pages 11–18. Research Institute for Advanced Computing Science, SIAM, May 1994.

[11] J. Choi, J. Dongarra, and D. Walker. PB-BLAS: A set of parallel block basic linear algebra subprograms, 1995.

[12] J. Choi, Dongarra J., R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *IEEE Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992.

[13] Jaeyoung Choi, Jack Dongarra, Roldan Pozo, and David Walker. Constructing numerical software libraries for high-performance computing environments. In *First International Workshop on Parallel Scientific Computing*, pages 147–168, 1994.

[14] L. Clarke, R. Fletcher, S. Trewin, A. Bruce, G. Smith, and S. Chapple. Reuse, portability and parallel libraries. In *Proceedings IFIP WG10.3 - Programming Environments for Massively Parallel Distributed Systems*, 1994.

[15] William Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):575–785, June 1990.

[16] William Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[17] William Dally and Charles Seitz. The torus routing chip. *Distributed Computing*, 1:187–196, 1986.

[18] W.J. Dally. The J-machine: System support for Actors. *Actors: Knowledge-Based Concurrent Computing*, 1989.

[19] E.F. Van de Velde. Data redistribution and concurrency. *Parallel Computing*, 2(3):125–137, December 1990.

[20] L.M. Delves, D. Lloyed, and T. Lahey. Fast forwarding Fortran. *Computer Standards and Interfaces*, 18(4):315–20, August 1996.

[21] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. Philadelphia, PA, May 1979.

[22] J. Dongarra, J. Du Croz, S. Hammarling, J. Wasniewski, and A. Zemla. LAPACK for Fortran 90. *Applied Mathematics and Computer Science*, 6(2):375–82, August 1996.

[23] Jack Dongarra, Roldan Pozo, and David Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Supercomputing Conference*, pages 162–171, 1993.

[24] Jack Dongarra and David Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, June 1995.

[25] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. *ACM*, pages 256–266, 1992.

[26] Charles Flaig. VLSI mesh routing systems. Technical Report 5241:TR:87, California Institute of Technology, May 1987.

[27] High Performance Fortran Forum. *High Performance Language Specification.* On the World Wide Web, 1997.

[28] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. *PICL: A Portable Instrumented Communication Library.* Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[29] Gene Golub and Charles Van Loan. *Matrix Computations.* The John Hopkins University Press, second edition, 1989.

[30] Gene Golub and James Ortega. *Scientific Computation: An Introduction With Parallel Computing.* Academic Press, 1993.

[31] A.Y. Grama, A Gupta, and V. Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(3):12–21, August 1993.

[32] W.D. Gropp, L.C. McInnes, and B.F. Smith. Scalable libraries for solving systems of nonlinear equations and unconstrained minimization problems. In *Proceedings of Scalable Parallel Libraries Conference*, pages 60–67. Mississipi State University, IEEE Computer Society Press, October 1994.

[33] A. Gupta and V Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

[34] S.K.S. Gupta, S.D. Kaushik, C.H. Huang, J.R. Johnson, and P. Sadayappan. A methodology for generating data distributions to optimize communication. In *Proceedings of the 4th IEEE symposium on Parallel and Distributed Processing*, pages 436–441, December 1992.

[35] Sven Hammerling. Challenges of portable parallel libraries for high performance machines. In Jack Dongarra and Jerzy Wasniewski, editors, *First International Workshop on Parallel Processing*, 1994.

[36] P.J. Hatcher and M.J. Quinn. Supporting data-level and processor-level parallelism in data-parallel programming languages. In H. El-Riwini, T. Lewis, and B. Shriver, editors, *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 14–22. IEEE Computer Society Press, 1993.

[37] P.J. Hatcher, M.J. Quinn, A.J. Lapadula, B.K. Seevers, R.J. Anderson, and R.R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):377–383, July 1991.

[38] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[39] R.A. Horn and C.R. Johnson. *Topics in Matrix Analysis.* Cambridge University Press, 1991.

[40] Lennart Johnsson. Massively parallel computing: Data distribution and communication. Technical Report TR-29-92, Parallel Computing Research Group, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, December 1992.

[41] Lennart Johnsson. *Probability and Performance In Parallel Processing.* John Wiley and Sons Ltd, 1993.

[42] Lennart Johnsson. CMSSL: A scalable scientific software library, May 1994. Obtained from Netlib.

[43] Lennart Johnsson and Mathur Kapil. Scientific software libraries for scalable architectures. Technical Report TR-19-94, Parallel Computing Research Group, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, August 1994.

[44] Lennart Johnsson and Kapil Mathur. High performance, scalable scientific software libraries. Technical Report TR-19-93, Parallel Computing Research Group, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, October 1993.

[45] E.T. Kalns and L.M. Ni. DaRel: A portable data redistribution library for distributed-memory machines. In *Proceedings of Scalable Parallel Libraries Conference*, pages 78–87. Mississipi State University, IEEE Computer Society Press, October 1994.

[46] E.T. Kalns and L.M. Ni. Processor mapping technique toward efficient data redistribution. *IEEE Transactions On Parallel and Distributed Systems*, 6(12):1234–1247, December 1995.

[47] S.D. Kaushik, C.H. Huang, J. Ramajunan, and P. Sadayappan. Multi-phase array redistribution: Modeling and evaluation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 441–445, April 1995.

[48] S.D. Kaushik, C.H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase array redistribution: Modelling and evaluation, 1995.

[49] David Keppel. Tools and techniques for building fast portable thread packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[50] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[51] B. Kernighan and T. Ritchie. *The C Language.* Osborne McGraw Hill, 1988.

[52] C. Koelbel et al. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[53] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2:440–451, October 1991.

[54] Bruce Lester. *MultiPascal.* Addison-Wesley, 1990.

[55] Bil Lewis and Daniel Berg. *A Guide to Multithreaded Programming, Threads Primer.* SunSoft Press, 1996.

[56] V. Lo. Oregami. In *University of Oregon*, 1990.

[57] E. Ma and D.G. Shea. E-kernel - An embedding kernel on the IBM Victor V256 multiprocessor for program mapping and network reconfiguration. *IEEE Transactions on Parallel and Distributed Processing*, March 1993.

[58] E. Ma and L. Tao. Embeddings among meshes and tori. *Journal of Parallel and Distributed Computing*, 18:44–55, 1993.

[59] Neal Margulis. *i860 Microprocessor Architecture*. Osborne McGraw Hill, 1993. ISBN 0-07-881645-9.

[60] D. McCallum and M.J. Quinn. A graphical user interface for data-parallel programming. In *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, volume 2, pages 5–13. IEEE, 1993.

[61] U.H. Nguyen, H. Hosseini, and L. Tao. Contention minimization in Wormhole routed networks. In *Proceedings of the Supercomputing Symposium*, page 454, Montreal, Quebec, Canada, July 1995.

[62] U.T. Nguyen. CPSS: An efficient multicomputer simulator. Master's thesis, Concordia University, Montreal, Quebec, Canada, June 1997.

[63] R.L. Page. Procedures and modules in Fortran 90. *Computer Standards and Interfaces*, 18(4):333–47, August 1996.

[64] M.J. Quinn. *Parallel Computing: Theory and Practice*. Addison-Wesley, 1994.

[65] M.J. Quinn and P.J. Hatcher. Data parallel programming on multicomputers. *IEEE Software*, pages 69–76, 1990.

[66] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Proceedings of Frontiers '95: the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, February 1995.

[67] K. Reeuwijk, W. Denissen, H. Sips, and E. Paalvaast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, Sept 1996.

[68] J. Reid. The array features [Fortran 90]. *Computer Standards and Interfaces*, 18(4):323–31, August 1996.

[69] M. Rosing. *Efficient Language Constructs for Complex Parallelism on Distributed Memory Multiprocessors*. PhD thesis, University of Colorado, Boulder, August 1991.

[70] M. Rosing, R.B. Schnabel, and R.P. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(9):30–42, November 1991.

[71] Charles Seitz and William Dally. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(2):547–553, May 1987.

[72] V. Strumpen and P. Arbenz. Improving scalability by communication latency hiding. In *Processing for Scientific Computing*, pages 778–9. Instititue for Scientific Computing, Zurich, Switzerland, SIAM, Feb 1995.

[73] X.H. Sun. The relation of scalability and execution time. In *The 10th International Parallel Processing Symposium, Proceedings of the IPSS '96*, pages 457–462, April 1996.

[74] X.H. Sun and L. Ni. Another view of parallel speedup. In *Proceedings of the Supercomputing '90*, pages 324–333, 1990.

[75] X.H. Sun and D.T. Rover. Scalability of parallel algorithm-machine combination. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):599–613, June 1994.

[76] L. Tao and H. Hosseini. On channel contention in Wormhole routing. In *Proceedings of the Supercomputing Symposium*, pages 99–106, Calgary, Alberta, Canada, June 1993.

[77] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587–594, June 1996.

[78] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide*, version 6.0.2 edition, June 1991.

[79] Y.M. Tsai and C.S. Yang. Fault-tolerant and deadlock-free Wormhole routing in the hypercube network. In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, pages 9–16, December 1992.

[80] D. Walker and S. Otto. Redistribution of block-cyclic data distribution using MPI. *Concurrency: Practice and Experience*, 8(9):707–728, Nov 1996.

[81] Zhiwei Xu and Kai Hwang. MPPs and clusters for scalable computing. In *Proceedings Frontiers '96. The Sixth Symposium on the Frontiers of massively Parallel Computing*, pages 117–23. Nat. Center for Intelligent Computing, IEEE Computer Society Press, June 1996.

[82] B. Yang and D.R. O'Hallaron. Procedure call models for distributed parameters in data parallel programs. In *Proceedings of Scalable Parallel Libraries Conference*, pages 157–164. Mississipi State University, IEEE Computer Society Press, October 1994.