# A Programming Paradigm for Distributed-Memory Computers

S. Crivelli
E. R. Jessup

CU-CS-818-96

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

## Abstract

Programming models are useful programming tools because they allow the recognition of patterns that are common to many applications and that are not committed to any particular language or machine. In this paper, we survey the different kinds of models and present some examples of them. We also introduce a new programming paradigm called PMESC that assists programmers in the design and implementation of problems on distributed-memory computers. The PMESC paradigm recognizes different phases in the computation involving different programming issues. Structuring programs via the PMESC paradigm allows programmers to separate the code into modules and to develop the modules independently. We show how the PMESC paradigm applies to the different categories of parallel problems, and we derive the frameworks that illustrate salient features in their implementation. We also discuss how to combine PMESC with other models in order to make the most effective use of those abstractions.

# A Programming Paradigm for Distributed-Memory Computers[1]

S. Crivelli E.R. Jessup
Department of Computer Science,
University of Colorado
Boulder CO 80309–0430.

CU-CS-818-96 October 1996

University of Colorado at Boulder

Technical Report CU-CS-818-96
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

# A Programming Paradigm for Distributed-Memory Computers[†]

S. Crivelli E.R. Jessup
Department of Computer Science,
University of Colorado
Boulder CO 80309–0430.

October 1996

## 1 Introduction

As the use of parallel computers becomes accepted as the only plausible way to solve very complex or large problems, it is becoming increasingly important to develop tools that help programmers to take advantage of this computing power. Programming tools are fundamental to the spread of parallel computing because parallelism significantly complicates the development of code. The programmer must now be concerned with many issues for which there are no direct counterparts in sequential programming such as the number and the physical interconnection of the processors, load distribution, and data sharing.

However, the rapidly changing technology of these computers and the lack of standards have made it difficult to develop a sufficient set of tools. The advent of MPI (Message Passing Interface) [21, 36] is an important step toward dramatically reversing this process for it provides a basis for building efficient and portable libraries. In this paper, we discuss another approach that complements this effort: the use of programming models for using parallel computers. In particular, we present a new programming paradigm as a programming tool that assists programmers in the design and implementation of problems on distributed-memory computers.

The PMESC paradigm provides a high-level abstraction that encompasses all the stages involved in the computation. This structurization allows programmers to recognize the modules that compose their applications and to attack them as independent units. It also allows the separation of the machine-dependent aspects of the computation from those that are machine independent and the identification of those modules that should be built on top of MPI. Finally, it provides the basis for highly tuning an application without having to substantially rework the code each time a new machine is introduced.

This paper is organized as follows. In section 2, we discuss the advantages of programming models in general and describe the different kinds of models in particular. We also present examples of each kind of model. In section 3, we describe the current efforts in using programming models. We also discuss the need for a new abstraction that covers at a high level all the steps involved in parallel implementations on distributed-memory computers. In section 4, we introduce the PMESC programming paradigm. In section 5, we present a taxonomy of

[†]This work was funded by DOE contract DE-FG03-94ER25215, by NSF contract ASC-9357812, and by a grant from the Intel Corporation.

the parallel algorithms and discuss the different approaches for parallelization. In sections 6, 7, and 8, we describe these approaches in more detail, give some examples, and show how the PMESC paradigm applies to each one of them. In section 9, we analyze the composition of programming models. Finally, we draw our conclusions in section 10.

## 2 The Programming Models

Programming models are useful programming tools. First, they allow the recognition of programming issues that are common to many applications and that are independent of the applications themselves. These abstractions facilitate the transfer of experience and knowledge gained from previous implementations as well as the reuse of code. Second, programming models are not committed to any particular machine, and, therefore, they promote the writing of portable code. Third, they are not committed to any particular language. Rather, they are represented by pseudo-codes that are easy to read and understand. Because these models are not associated with any language or machine, they allow users to finely tune their codes as much as they want.

Programming models are both educational and research tools. They are educational tools because they bring together the ideas about the different algorithms that exist and the different approaches for parallelization. Thus, they help programmers to identify the essence of a problem and to use known methodologies to attack that problem. Programming models are research tools because they help scientists to recognize the building blocks that compose large-scale applications. Consequently, they facilitate the writing of code that is highly customized but that can be modified and ported without significant redesign.

Programming models come in different variations. Some are programming paradigms, some archetypes, and some others templates. Next, we describe each one of these variations.

### 2.1 Programming Paradigms

Programming paradigms are a combination of problem-solving methodologies and the program design techniques associated with them. Programming paradigms are not actual algorithms but rather the strategies that can be used for structuring the algorithms. Thus, paradigms are the high-level abstractions that are common to many algorithms [27, 31] and that can be used as starting points when developing new ones. Because the essential computational structure of these methodologies is already known, programmers using paradigms only need to work on the problem-specific details in order to produce a new program design.

Paradigms represent the algorithms in the same way as higher-order functions, i.e., functions that take functions as arguments, represent general computational frameworks in the context of functional programming languages [32]. Like higher-order functions, paradigms are not concerned with the lowest level details of particular problems. Instead, they capture the higher-level computational structure of whole classes of algorithms. Implementations of particular problems are mere instances of these general frameworks [8].

As experience in parallel processing has grown, a number of programming paradigms have arisen, each of which represents the structure of a particular programming style or technique that can be efficiently applied to a particular class of problems. Next, we describe some examples of those programming paradigms.

### 2.1.1 The Divide-and-Conquer Paradigm

This paradigm is used both in sequential and parallel computing. It applies to problems that can be divided into two or more smaller subproblems of the same type. Thus, the subproblems are just smaller instances of the original problem. They are divided recursively until they reach a threshold size. Once these smallest base-case problems are solved, their solutions are combined to produce the solutions to the larger ones. This process repeats recursively until the solution to the original problem is found. The divide-and-conquer strategy can be used to solve a wide variety of problems such as sorting [2, 7, 31], matrix multiplication [31], Fourier transforms [28, 31], and eigenvalue problems [10, 12, 37].

In a parallel computation based on the divide-and-conquer paradigm, the problem is subdivided into independent subproblems of the same type that can be solved in parallel. Because the data and the results are distributed among the processors, their combination requires interprocessor communication. Figure 1 shows the framework corresponding to the parallel divide-and-conquer paradigm.

When using the divide-and-conquer paradigm, the programmer is responsible for addressing the following issues:

- Determining the size of the base-case problems, making sure that they are large enough to justify the overhead of communication for data exchange and scheduling processors,
- Implementing the function *Solve* that solves the base-case problems,
- Implementing the procedure *Split* that takes a problem and subdivides into smaller subproblems,
- Implementing or using an existing procedure *Combine* that combines the solutions to the subproblems to obtain the solution to the original problem.

### 2.1.2 The CAB Paradigm

The CAB paradigm [31] consists of three phases: Compute, Aggregate, and Broadcast. The problem is partitioned into subproblems, and each processor is responsible for the solution of a subproblem. This solution corresponds to the Compute phase. The Aggregate phase is a gathering operation that combines data from the processors, producing a global value. This global value, or some information based upon it, is sent to the processors in the Broadcast phase. Depending on this information, processors may proceed or not with the next stage. Thus, this paradigm applies to those computations that can be divided into the C, A, and B stages. Examples of the CAB paradigm are given by the parallel implementation of iterative methods such as Jacobi and Gauss-Seidel [31]. Figure 2 depicts the framework corresponding to the CAB paradigm.

When using the CAB paradigm, the programmer is responsible for addressing the following issues:

- Partitioning the problem into subproblems,
- Implementing the function *Compute* that solves the subproblems,
- Deciding what information needs to be shared by all the processors,

```
procedure Divide-and-Conquer (in Problem T, out Solution S)
begin
    Problem T1, T2, ..., Tn;
    Solution S1, S2, ..., Sn;

    if ( T = base-case )
      S = Solve (T);
    else
      Split (in T, out T1, T2, ..., Tn);
      begin parallel computation
            Divide-and-Conquer (in T1, out S1);
            Divide-and-Conquer (in T2, out S2);
            ...
            Divide-and-Conquer (in Tn, out Sn);
      end parallel computation
      Combine (in S1, S2, ..., Sn, out S);
    end if
end
```

**Figure 1**: Framework for the Parallel Divide-and-Conquer Paradigm. Source: [6]

- Implementing or use an existing *Combine* routine for gathering the partial results,
- Implementing or use an existing *Broadcasting* routine,
- Deciding when to terminate.

In the CAB paradigm what is important is the identification of the phases that compose any problem, not the order in which they appear in the computation. Thus, the paradigm may also be Broadcast-Compute-Aggregate (BCA) or any other combination of the three.

### 2.1.3 The PCAM Paradigm

The PCAM paradigm [15] is a *design* methodology that separates issues that should be considered early in the program design process from those that should be considered late. PCAM structures this process as four distinct stages that must be analyzed in this order: Partitioning, Communication, Agglomeration, and Mapping. The Partitioning stage deals with the decomposition of the computation to be performed *and* the data to be operated upon into small tasks. The Communication stage determines the intertask communication necessary and defines the corresponding communication structures. The algorithm or algorithms resulting from the Partitioning and Communication stages are not specific to any parallel computer.

The next two phases, Agglomeration and Mapping, take into consideration the machines to use. They revise the algorithms determined in the Partitioning and Communication stages, selecting those that are efficient on a particular class of parallel computer. Thus, the Agglomeration stage combines the tasks identified in the Partitioning stage into larger tasks in order to reduce communication costs and improve the performance. Observe, that although Partitioning

```
procedure CAB (in Problem T, out Solution S)
begin
        Problem Ti;
        Solution Si;

        \* Split Problem T into subproblems Ti *\
        Partition (in T, out Ti);

        do
            \* Processor i computes problem Ti *\
            Si = Compute (Ti);

            \* Processor 0 gathers the partial results Si and combines them in S *\
            Aggregate (in 0, Si, out S);

            \* Processor 0 broadcasts global result S *\
            Broadcast (in 0, inout S);

            if (S meets the solution requirements)
                    continue;
            else
                    terminate;
            end if
        until (termination)
end
```

**Figure 2**: Framework for the CAB paradigm

and Agglomeration are both concerned with the splitting of the work into pieces, Partitioning takes place at an early stage of the design process while Agglomeration comes later and takes into account the machine or machines to be used for the implementation. Finally, the Mapping stage analyzes the assignment of tasks to processors in such a way that maximizes load balance and minimizes communication costs. Thus, the first two stages of the PCAM paradigm focus on natural parallelism and scalability while the last two stages focus on practical parallelism, locality, and other ways to pursue performance.

Unlike the other paradigms described in this paper, PCAM represents steps involved in the design process, rather than in the more concrete process of writing code. Because it organizes the design, rather than the code itself, it is not suitable for representation with a pseudo-code.

Examples of programs that can be designed using the PCAM paradigm are those for the solutions of a set of partial differential equations by a finite difference method and of a floorplan optimization problem by the branch-and-bound method [15].

#### 2.1.4 Other Paradigms

Other examples of programming paradigms include the mesh paradigm [6], the queue paradigms [8, 13, 40], and the systolic paradigm [31]. The mesh paradigm is used in sequential and parallel computing. It provides the basis to implement some iterative methods that present an n-dimensional mesh structure. These methods compute the new values of some variables at each point of the mesh, based on the old values and the values at neighboring points. The parallel approach includes the following steps. First, the mesh is partitioned into regions that are assigned to the processors. Neighboring regions share a boundary. Then, processors solve the problem in their assigned regions as well as on the corresponding boundaries. Finally, neighboring processors exchange information about the boundaries they share. The computation in the regions and their boundaries repeats in a loop until the values of the computed variables reach the desired accuracy.

The queue paradigms provide the abstractions to deal with a common programming issue that arises in distributed-memory computing: how to decompose the original problem into subproblems and distribute them among the processors. These subproblems, which can be created dynamically during the computation, are stored in a queue. Depending on which processors store and handle the queue, the queue paradigm can be centralized, distributed, or a combination of both.

The systolic paradigm addresses another typical problem in distributed-memory computing: how to redistribute data among the processors efficiently. The paradigm provides an efficient communication structure that lets data flow through the processors so they can all have access to them. Instances of this paradigm are computations involving large matrices that are stored by blocks in the processors' memories. The blocks flow through the processors, and processors perform their computations in response to the data received.

### 2.2 Programming Archetypes

An archetype is a combination of a problem-solving method, a program design strategy associated with that method, and sample problems to which the method can be applied written in one or more programming languages for one or more target machines. In other words, an archetype is a programming paradigm accompanied by a set of examples and suggestions for turning the high-level abstraction provided by the paradigm into a real application. These suggestions are intended to reduce the effort required to implement correct code as they come from the experience of others who have developed similar applications. They may cover a wide range of aspects involved in the implementation process, such as common problems and errors, test suites, debugging, and performance tuning for different target machines.

For example, the divide-and-conquer archetype proposed in [6, 14, 31] defines the divide-and-conquer paradigm but also provides discussion of several important programming issues including approaches to parallelization, language to use, performance, debugging, and execution. In addition, it illustrates the use of the divide-and-conquer paradigm via such applications as merge sort. This corroborates the idea that archetypes are designed to provide support to programmers in every step of the program development process.

Another example of an archetype is the mesh archetype proposed in [6, 14]. It includes a description of the mesh paradigm as well as some applications based on it. It also supplies subroutines and functions to perform global reductions, exchange of boundary data, and some parallel I/O (the latter is still in experimental stage.) Languages and libraries provided for this archetype are Fortran M [15, 16], Fortran 77, and PVM [17]. Applications that fit this archetype include some fluid dynamics computations that use iterative and/or finite-element solvers. In particular, the archetype includes an application that simulates flow in a toroid.

## 2.3 Templates

Among the abstractions discussed in this paper, templates are the lowest-level ones and the closest ones to the actual algorithms. In fact, templates are general descriptions of algorithms that can be translated directly into code by substituting a programming language for the natural language in which the template is expressed. As do programming paradigms and archetypes, templates provide a framework on which to build a program. However, the frameworks provided by the latter are more detailed and low-level than those provided by the former. Consequently, because they provide a higher level of detail, templates cannot be as general as paradigms and archetypes. Nevertheless, templates are becoming popular because they provide pseudo-code versions of the algorithms while still offering whatever degree of customization the user may desire.

A number of templates have been proposed in the literature [1]. An example is given by the Jacobi method. This provides an iterative mechanism for solving the linear system $Ax = b$ based on the relation

$$x^{(k)} = (b_i - \sum_{j \neq i} a_{i,j} x_j^{(k-1)})/a_{i,i}.$$

Figure 3 shows the template corresponding to the Jacobi method.

Given this template, the programmer has to translate the pseudo-code into the target language such as Fortran or C. Using the Basic Linear Algebra Subprograms [11] facilitates this process. Although the pseudo-code closely resembles the actual code, it still allows the programmer a great deal of freedom. For instance, it allows programmers to tune the data structures for their applications. The Jacobi template leaves the programmers with even more to decide in the parallel case. Some of these decisions include how to partition the work among the processors and how to combine the partial results to check for convergence. The CAB and mesh paradigms, which also apply to parallel implementations of Jacobi, provide more insight on how to handle these issues than the template does. The template, however, provides a more detailed description of the Compute phase that takes place in every processor.

```
**procedure** Jacobi (**in** Solution $x^{(0)}$, **out** Solution $x$)
\* $x^{(0)}$ initial guess to solution $x$ *\
**begin**
        **for** $k = 1, 2, \ldots$
            **for** $i = 1, n$
                $y_i = 0;$
                **for** $j = 1, n; j \neq i$
                    $y_i = y_i + a_{i,j} x_j^{(k-1)};$
                $y_i = (b_i - y_i)/a_{i,i};$
            **end**
            $x^{(k)} = y;$
            check convergence; continue if necessary;
        **end**
        $x = y;$
**end**
```

**Figure 3**: Template for the Jacobi method. Source: [1]

## 3 Current Efforts in Using Programming Models

The programming models described in section 2 are all well-suited to certain types of problems but are inappropriate for others. A natural question is whether it is possible to create a universal model, i.e., one that could be applied without restriction to solve any parallel problem on any parallel machine. Unfortunately, this is an impossible goal as different types of computers and problems require different approaches for parallelization. Furthermore, the provision of a single, universal programming framework is incompatible with the goal of efficiency.

In his book *Algorithmic Skeletons: Structured Management of Parallel Computation* [8], M. Cole proposes an alternative solution: to build a system that presents not a single model but rather a collection of them. In this system, the user would be presented with a menu listing the available models. The user would select the model that is appropriate for the problem at hand as well as the language in which she or he wants to write the code. The system would respond by displaying the generic program that describes the structure of the model in the chosen language. Finally, the system would ask the user to provide problem specific details of the data structures and procedures so that it can turn the generic model into the user application.

Although Cole's notion of an abstract machine based on algorithmic skeletons is in an embryonic state, there is a great deal of work being done directed at gathering different collections of skeletons and strategies of resolution. Rather than providing an automatic way to produce code from specifications these libraries provide program development methods which require ample user participation and creativity.

One of these projects is being developed at the University of Tennessee [1]. It provides a collection of templates for solving large sparse systems of linear equations by iterative methods. The templates are accompanied with useful information, such as discussions of convergence and stopping criteria, suggestions on how to choose and implement an efficient method, and tips on parallel implementations [1].

Another effort directed at building high-level libraries is taking place at Mississippi State University [35]. The Multicomputer Toolbox is a set of parallel libraries for solving linear systems. The Toolbox includes sparse, dense, direct and iterative linear algebra, and a stiff ODE/DAE solver. These diverse methods are linked together through a uniform calling interface. An important concept involved in the design of the Toolbox is that of poly-algorithms. A poly-algorithm consists of the use of two or more algorithms to solve the same problem and a high level decision-making process that determines which algorithm performs best in a given situation. Such poly-algorithms are the key to performance and portability of these libraries [35].

A third project has been initiated at the California Institute of Technology [6, 14]. It aims at studying the usability of a library of parallel programming archetypes as a tool to reduce the efforts required to produce correct and efficient programs. The project is monumental for it does not circumscribe itself to any particular category of problem like the Tennessee one.

This archetype library covers three categories including scientific applications, combinatorics and optimization, and reactive systems. The scientific applications category is composed of the mesh-based archetype. The combinatorics and optimization category is represented by the divide and conquer archetype. Finally, the category of reactive distributed systems deals with archetypes for more specific problems such as multiprocess synchronization, distributed clocks, collective communication operations, and mutual exclusion. Future plans include archetypes for commercial applications that help small business to operate through the network.

## 3.1 What is needed

The programming models presented in the literature do not represent all the kinds of problems that exist in distributed-memory computing. For instance, the divide-and-conquer, CAB, and mesh paradigms support computations that require synchronization among the processors. In all these paradigms, the computation is divided up into stages, cycles or iterations. Processors need to wait for the partial results computed by other processors before they can proceed with the next cycle. However, there are other kinds of problems for which no cycles can be distinguished and no synchronization points are necessary. Moreover, synchronization should be eliminated in these problems for it introduces a source of unnecessary overhead.

Another type of problem that is not represented by these models is one in which the work is generated dynamically and unpredictably. These problems are difficult to partition and distribute among the processors in such a way that keeps the load balanced. One of the queue paradigms supports a centralized approach in which a master processor handles the pieces of work or tasks as they are generated and assigns them to the other processors upon request. However, this approach can only be efficiently applied to applications involving a small number of processors [26]. The distributed queue paradigm, on the other hand, provides a scalable approach in which each processor is responsible for its own work but does not resolve the load imbalance problem that may arise among them.

The PCAM paradigm provides a more general approach that applies to all of these problems. However, it represents the steps that programmers have to go through in the design process rather than the modules that they need to implement when writing their codes. Therefore, it is desirable to find a general model that represents, at a high level, the methodologies to design *and* implement a wide variety of scientific problems on distributed-memory computers. This model should help programmers to understand the big picture first and then to identify how their problems fit within it. Thus, the mesh, divide-and-conquer, queue, and CAB paradigms, would be particular instances of this main structure. Although general, this model should provide enough detail to help programmers understand the main categories of problems and to recognize their basic structure. It should also provide enough ground to go to a finer level of detail and study the building blocks that comprise those structures and the programming issues involved. Finally, this programming model should provide the basis we need to organize the software support that is already available and to determine what needs to be done.

In the next section, we present the PMESC paradigm that serves as this more general programming model. Although this new paradigm and PCAM have been developed independently, they complement each other well. In fact, while PCAM analyzes the main stages involved in the design process, PMESC separates the main building blocks or modules that compose a parallel code. Thus, the results of all the analyses made at the Partition and Agglomeration stages of the PCAM process, should be considered in the implementation of the Partition module of PMESC. Likewise, the conclusions about convenient algorithms for Mapping and Communication made in the PCAM analyses, should be used in the implementation of the Map and Communicate modules of PMESC. PMESC then completes the implementation process by adding two more modules: one that takes care of the computation itself, called Solve, and other that takes care of embedding the communication structures used for the implementation onto the machine architecture, called Embed. The addition of these modules allows us to derive frameworks or pseudo-codes that help programmers to visualize the approaches to use and the different ways of putting the building blocks together.

## 4 The PMESC Paradigm

Parallel programming on distributed-memory computers presents several sources of difficulty. The first is the task of identifying the processes that can operate concurrently during the process of achieving a correct solution. The second is that of specifying a mapping of processes onto the available processors and indicating the mechanisms by which the mapping can be performed dynamically if necessary. In the most favorable situation, the decomposition and distribution of a problem will lead to a situation in which processors receive roughly equal amounts of independent work and can proceed with the execution until complete. However, in most cases, processors need to communicate either to share some data or to share some work. It is necessary to consider the mechanisms by which both kinds of sharings are to be performed. A final problem is the separation of the applications from the underlying characteristics of the hardware.

To better deal with all of these issues, we present a new programming paradigm. It is called **Partition-Map-Embed-Solve-Communicate**
(PMESC), and it is composed of five phases bearing those names. The **Partition** phase splits a problem into subproblems. The **Map** phase distributes those subproblems among the virtual graph of processors interconnected by some convenient virtual topology. The **Embed** phase embeds the virtual interconnecting topology of the processors into the actual machine architecture. The **Solve** phase performs the computation necessary to solve the subproblems, and the **Communicate** phase takes care of the interprocessor communication necessary for sharing data. Note that because these mechanisms of data communication and task communication are so different we rather treat them as two separate phases: Communicate and Map. The

five phases of the PMESC paradigm become the building blocks of distributed algorithms. To better understand this paradigm we next describe the computational model to which it applies.

### 4.1 The Computational Model

Distributed-memory computing consists of partitioning the work into units of work or tasks and assigning those tasks to the processors. Data is shared through message passing instead of through common memory. Let us assume a set of tasks $T$ and a set of processors $P = \{P_0, P_1, \ldots, P_{p-1}\}$. Let us call $(P, M)$ a graph whose vertices $P$ are connected by edges $M$ such that each edge in $M$ joins two processors in $P$. Let the graph $(P, M)$ represent the physical interconnecting topology of the processors, i.e., the actual machine architecture.

The problem is represented in terms of a graph $(T, G)$, whose vertices correspond to the tasks in $T$ and whose edges, $G$, correspond to their communication requirements. The graph of tasks $(T, G)$ is mapped onto the set of processors $(P, M)$. Thus, a program for that problem consists of a graph $(T, G)$ and a mapping function that assigns those tasks to the processors. In some cases, a program consists of multiple graphs, as tasks and their communication requirements evolve and change with the computation.

However, this model is still incomplete. We need to make it more flexible in order to deal with two seemingly conflicting goals: efficiency and portability. On one hand we want to separate the application from the machine architecture, i.e., to separate the low-level details of the parallel computation from the high-level ones, in order to create portable code. On the other hand, in order to develop highly efficient code we need to program the computer explicitly, matching the data-dependency graph of tasks $(T, G)$ with the physical interconnection topology of the processors $(P, M)$.

In order to be able to choose any variation between high efficiency and maximal portability, experienced programmers introduce a third graph $(P, V)$ which corresponds to the set of processors interconnected by some virtual topology. Under this new model, a program consists of the graphs $(T, G)$, $(P, V)$, and $(P, M)$ and two mapping functions: one that assigns tasks to the processors interconnected by some convenient virtual topology and the other that maps this virtual machine into the real machine architecture.

The programming model supported by PMESC provides the freedom of choosing between maximal efficiency and portability. The programmer can program the computer explicitly, matching the data-dependency topology of the tasks with the interconnection topology of the processors or assume a virtual architecture and embed it into the actual one. The first choice, which allows the writing of highly efficient but non-portable code, corresponds to the case when no virtual topology is used, i.e., $(P, V) = (P, M)$. The second choice, which allows the writing of portable but possibly less efficient code, corresponds to the case when the code is designed on a virtual machine that changes dynamically at the designer's convenience.

In section 3 we establish the need for a general model that fits a wide range of scientific applications on distributed-memory computing. To prove that PMESC is such model we now examine the different categories of problems and the different approaches to parallelizing them.

## 5 The Problems and the Approaches to Parallelization

In this section, we present a taxonomy of the problems. Our goal is to identify the approaches to parallelizing different problems according to a small set of salient features.

## 5.1 The Problems

Three categories of problems can be distinguished in parallel computing depending on the way they can be divided up among the processors: data parallel, task parallel, and a combination of both [15, 23]. Each of these categories can be classified as regular or irregular.

### 5.1.1 Data vs. task parallel.

Data parallel computations are those that present a large data domain that can be decomposed into subdomains to be assigned to the processors. The subdomains can be operated upon in parallel by performing the same computation on each. However, processors must exchange data periodically and synchronously in order to achieve correct results. Because data in neighboring subdomains are usually related, adjacent subdomains are usually assigned to neighboring processors (in the actual machine) in an attempt to control communication costs.

In contrast, task parallel computations do not necessarily present a large amount of data to split but rather are comprised of a large task that can be partitioned into asynchronous subtasks to be assigned to the processors. Different subtasks may involve different computations. These tasks may need to share some data, but usually they can do it asynchronously. Because the tasks are essentially uncoordinated and because the communication between them (if any) is not significant (compared to the data parallel case), tasks can be assigned and reassigned to any processor.

Observe that some applications may involve the two paradigms, data and task parallel. An example is given by those large scale applications implemented on heterogeneous computers. In this case, the original problem may be split into heterogeneous subproblems that are assigned to the different computers, following a task parallel approach. Each computer, in turn, applies the appropriate paradigm —data or task parallel— to the corresponding subproblems.

### 5.1.2 Regular vs. irregular.

Data and task parallel problems can be regular or irregular. Regular computations are those whose computational requirement can be determined or at least estimated a priori. In contrast, the computational requirement of irregular computations becomes evident only during their execution. Irregularity appears in some data parallel computations when it is difficult (if not impossible) to make an a priori partition of the data structure in such a way that each processor receives roughly the same amount of work. Irregularity appears in task parallel computations because the number of tasks and usually their computation times vary during program execution in an unpredictable manner.

## 5.2 The Approaches to Parallelization

Different types of problems may require different approaches for their efficient implementation on distributed-memory computers. These approaches can be classified as static and adaptive. The latter can be quasi-dynamic and dynamic [39]. In this section, we briefly describe each one of these categories and the differences between them.

The static approach splits and assigns work to the processors without any regard for the system state. This subdivision is made only once, usually at the beginning of the execution, and it is based on information that the programmer has about the application. For that reason, it is well suited to regular computations. Adaptive methods are an interesting alternative to this straightforward approach. They apply to those computations that are irregular and for which no a priori estimates of load distribution are possible. In this case, it is only during program execution that different processors can become responsible for different amounts of work. Adaptive approaches are especially appropriate for these problems because they react to the variations in the system state, concentrating efforts on those areas that look more promising and making work transfer decisions to keep the load balanced.

Adaptive approaches can be quasi-dynamic and dynamic. Quasi-dynamic approaches apply to those computations that are synchronous and predictable in stages and that may require periodic load balancing checks to achieve good performance. Dynamic approaches apply to those computations that are asynchronous and unpredictable and that may require load balancing checks at any time during the computation.

Next, we describe each category in more detail and present some examples. We also discuss the PMESC framework that corresponds to each one of the approaches.

## 6 The Static Approach

The static approach can be successfully applied to those problems, data or task parallel, that exhibit regular structure. Because these computations allow one to get an a priori estimate of the workload, they can be efficiently partitioned and mapped to the processors at the beginning of the execution. This initial partition and assignment of work achieves good load balance and no adjustments need to be made to improve the efficiency of the parallel implementation.

### 6.1 Example: the solution of Laplace's equation on a rectangle

An example of the static approach is given by the solution of Laplace's equation

$$\frac{\partial^2 V}{\partial^2 x} + \frac{\partial^2 V}{\partial^2 y} = 0$$

on a rectangle. The solution of this equation on a continuous space is approximated by studying the behavior of a finite set of points in that space. Typically, the domain is discretized by laying a grid on top of it. The numerical solution is obtained by following an iterative procedure that computes the value $V_{ij}$ at the point $(i, j)$ in the grid as the average of the values of its four neighbors at the previous iteration [15]

$$V_{ij}^{t+1} = \frac{1}{4}(V_{i-1j}^{t} + V_{i+1j}^{t} + V_{ij-1}^{t} + V_{ij+1}^{t}).$$

The iteration proceeds until the global error estimate, obtained as the maximum of the differences between the new and the old values over all points, is less than a given tolerance. Thus, the problem requires a static approach that assigns equal numbers of points to all the processors.

The efficiency of a static implementation depends heavily on the use of a good partitioning strategy that divides the work evenly and also on an efficient mapping strategy that assigns the pieces of work to the processors in such a way that keeps the communication costs low. Although these are straightforward procedures in this PDE example (as we will see in the next section), other problems require more sophisticated techniques. Many researchers have been working on efficient strategies for domain decomposition of PDE problems [19, 20, 33, 34] and for efficiently mappings of the subdomains onto the processors [3, 4, 5, 23, 29].

```
Initialize;
begin
        Partition;
        Map;
        Embed; \* topology or topologies used in Map and Communicate *\
        begin iterative procedure
                Solve;
                Communicate;
        end iterative procedure
        Communicate;
end
```

**Figure 4**: PMESC framework for the static approach

## 6.2 Applying PMESC to static problems

Figure 4 shows the PMESC framework for the static approach. It begins with the **Partition** of the work into units. In the PDE example, this phase corresponds to the partition of the grid into subgrids of identical size. There are as many subgrids as processors to be used.

The next task is to assign the subgrids to processors. In order to make the approach independent of the computer architecture, the problem should be formulated in terms of a virtual machine whose processors are interconnected by virtual communication channels. As the data in the PDE example form a grid, the most natural virtual machine to use for that problem is a grid or mesh of processors. The **Map** phase assigns the subgrids to the processors of the virtual mesh. The **Embed** phase embeds the virtual machine into the actual machine architecture. In addition to allowing a straightforward mapping, the virtual mesh turns out to be a good one for an efficient program. Neighboring processors in the virtual mesh need to communicate to exchange the data values along the edges of their assigned subgrids. The virtual mesh may be mapped onto both hypercube and mesh machines so that neighboring processors in the virtual mesh are mapped onto neighboring processors in the actual machine.

Next, each processor proceeds independently with its assigned part of the computation. In the PDE example, each processor must execute an iterative procedure that is composed of two phases: **Solve** and **Communicate**. The **Solve** phase computes the values of $V$ at the assigned points of the grid. The **Communicate** phase exchanges values at the boundaries of the subgrids between neighboring processors in the mesh so that processors can proceed with the next iteration. It also performs some global combine operation to check for convergence.

At the end, processors may communicate again to gather the partial results. This is also taken care of by the **Communicate** phase. Note that the **Communicate** phase (like other phases in PMESC) may be composed of different modules. In the PDE example, **Communicate** comprises a module for exchanging of data between neighboring processors and one for globally combining results. These modules may use different virtual topologies which need to be embedded by different procedures.

## 7 The Quasi-Dynamic Approach

The quasi-dynamic approach applies to those computations that can be divided into stages. Each stage is composed of the following steps. First, the problem is decomposed into subproblems. Second, the set of subproblems is mapped into the set of processors. Then, processors execute their assigned subproblems. To do so, processors may need to exchange some data. In the quasi-dynamic approach, this exchange is synchronous to guarantee determinism. Besides exchanging data, processors may need to transfer work to keep the load balanced. These transfers are also synchronous. Because in these computations the workload redistributes itself gradually, the transfers of work are only necessary periodically. When the workload becomes imbalanced the current stage ends and the work estimate of the next stage is estimated. Based on that estimate, the work is redistributed and mapped among the processors, and another stage begins.

### 7.1 Example: adaptive irregular multigrids

There are some PDE problems where the computational effort needs to be concentrated on some regions of the domain. Because the domain is discretized by a mesh, there are some regions where the mesh needs to be more refined than others. Because the location of these regions is not known in advance, adaptive irregular grids that allow localized refinement are necessary. In particular, there are some methods called adaptive multigrid methods, where the domain may be discretized by a hierarchy of grids that have different resolutions [25, 30]. As the computation goes on, the collection of grids may be changed from coarser to finer by applying the refinement procedure.

The adaptive solutions to these irregular problems attempt to balance the load periodically, remapping the adapting grids as they change. These changes are called grid refinements, and they separate the computation into stages. In each stage, a grid is decomposed into units (aggregates of nodes). Units that do not have dependencies or temporal precedence constraints —i.e., those that can be executed simultaneously— are mapped onto the set of processors arranged as a virtual mesh. This virtual topology needs to be embedded into the real machine. With the units assigned to the processors, the problem is solved for that grid using some iterative procedure. During that procedure, processors computing neighboring units may need to exchange some data.

The optimal load distribution for each stage is determined by minimizing the estimated parallel execution time of that stage. This is done by assigning computation and communication costs to the units and using these values as input parameters to some approximate cost function. Several optimization algorithms are proposed in the literature based on the minimization of the estimated parallel execution time of the next stage. Among them, recursive bisection, orthogonal bisection, and simulated annealing are commonly used in many PDE implementations [3, 24, 33, 39]. Any of these procedures requires global synchronization of processors at the end of each stage.

### 7.2 Applying PMESC to quasi-dynamic problems

Figure 5 shows the PMESC framework for the quasi-dynamic approach. The computation is divided in stages, each with its corresponding Partition, Map, Embed, Solve and Communicate procedures. In the multigrid example, each stage deals with the solution of the problem in a

```
Initialize;
begin
    for $i = 1$ : number of stages
        begin stage
            Partition;
            Map;
            Embed;
            Solve;
            Communicate;
        end stage
    end
end
```

**Figure 5**: PMESC framework for the quasi-dynamic approach

particular grid. Partition represents the domain decomposition. It may be global or it may run sequentially on a single processor, depending on the method used. The mapping procedure corresponds to the initial assignment of work to the processors of a virtual machine as well as to the subsequent reassignments that may be needed to keep the load balanced. These reassignments are always synchronous to guarantee deterministic results. In the example, the virtual topology used is a mesh. Thus, the Embed phase represents the embedding of the mesh into the actual machine architecture.

After partitioning, mapping, and embedding, processors can proceed with the computation. These computations correspond to the Solve phase. The Solve phase may represent a simple procedure or a very complex one. In the multigrid case, it corresponds to an iterative procedure to evaluate the solution on a grid. This phase can be represented in turns, by another programming model such as the CAB paradigm [31] or the mesh archetype [14, 6]. We discuss this issue in more detail in section 9. Because data dependencies are so strong in these problems, exchange of data between processors is a very important ingredient in the computation. This exchange is associated with the Communicate phase.

A new stage begins when redistribution of work is necessary. In the multigrid case, this occurs when the procedure requires a grid refinement. The process repeats until computation is completed.

## 8 The Dynamic Approach

There is another kind of unpredictable computation for which no stages can be distinguished and no a priori estimates of load distribution are possible. Dynamic approaches are especially well-suited to these problems because they assume no prior knowledge of the workload and allow work redistribution at any time. Thus, like quasi-dynamic approaches, dynamic ones are adaptive. Like quasi-dynamic approaches, they evaluate the changes in the system state in order to make work transfer decisions. However, unlike quasi-dynamic approaches, they make these evaluations continuously rather than periodically, interleaving remapping with computation.

The dynamic approach is asynchronous as it allows each part of the computation to proceed independently of the other parts. It applies to those applications that can be split into parts that are as autonomous as possible. Therefore, the approach is especially appropriate for task parallel computations, where tasks vary dynamically in size and number.

### 8.1 Example: branch-and-bound

Examples of problems that need a dynamic approach can be found in those computations involving some type of tree search. This type of computation is difficult to partition and map to a distributed-memory computer because different branches of the tree may have different number of nodes and levels. In addition, trees evolve dynamically, making it impossible to achieve an efficient initial mapping of the work among the processors. To illustrate the situation, let us consider the solution of the traveling salesman problem (TSP) by the branch-and-bound procedure. This problem consists of a set $\{1, 2, \ldots, n\}$ of cities connected by a graph. An edge $(i, j)$ of the graph represents the distance $d_{i,j}$ between cities $i$ and $j$. A tour is a traversal of the graph in which each city appears exactly once. A solution to the traveling salesman problem is the tour of least cost. Thus, a solution is given by a permutation $\sigma$ of the set of cities $\{1, 2, \ldots, n\}$ that minimizes $\sum_{i=1}^{n} d_{i,\sigma(i)}$.

The branch-and-bound procedure associates a rooted tree with the problem. The execution of the algorithm corresponds to the traverse of that tree to find the leaves. A leaf represents either a solution of the problem, i.e., a goal leaf, or an unproductive partial solution, i.e., a partial solution that cannot lead to a solution. The nodes of the tree are generated by using the branching procedure, which applied to any problem $\mathcal{P}$, either solves it directly or derives a set of subproblems such that the solution of $\mathcal{P}$ can be found from the solution of the subproblems. Thus, when a branching procedure is applied to node $v$, it either determines that $v$ is a leaf or produces the children of $v$. The search recursively branches nodes until a set of leaves is identified as the desired solution of the problem.

The bounding procedure uses a global bound to prune those branches of the tree that cannot produce a solution [18, 38, 41]. It does so by defining a cost function $c$ that assigns a value $c(v)$ to each node $v$ based on the values of the nodes in the path from the root to $v$. The solution of the problem is given by the leaf with the minimum cost function. The global bound is the solution of least cost found so far. The bounding procedure is based on the monotonicity property of the lower bounds that states that the lower bound of a subproblem of $\mathcal{P}$ is at least as large as the lower bound of $\mathcal{P}$. This property ensures that any subproblem with associated cost bigger than the cost of one solution found does not lead to a feasible solution, and, therefore, it can be ruled out.

An efficient parallel branch-and-bound algorithm not only distributes the subproblems among the processors but also checks that processors do not waste time exploring unproductive subproblems. Therefore, a parallel branch-and-bound algorithm may not achieve good speedup by merely keeping the load balanced. In branch-and-bound algorithms, the order in which nodes are expanded matters. Thus, nodes are given priorities according to the costs associated with them. Nodes with lower cost have higher priorities because they are more likely to produce a solution. A priority queue must be maintained to implement these problems so that the node with the highest priority can be easily found.

The parallel implementation of this algorithm is clearly task-parallel. Processors split their work into units or tasks and store them in a priority queue from where they select them for execution. Processors execute the tasks asynchronously and independently until they become overloaded or idle, in which case they activate the load balancing mechanism to transfer tasks. Processors having some work to give away split their local queues and send one part to another processor. Because the send is asynchronous, the sending processor can proceed with execution immediately. The communication necessary to share information is also asynchronous. That is one of the main characteristics of the dynamic problems that distinguishes them from the quasi-dynamic ones. In this particular example, communication is necessary to update the bound that processors use to prune some branches of the tree and to concentrate on the most promising ones.

## 8.2 Applying PMESC to dynamic problems

Figure 6 shows the PMESC framework for the dynamic approach. First comes the Partition phase with the initial subdivision of the work into units of work or tasks. In the branch-and-bound algorithm, the Partition phase corresponds to the expansion of the root to create a set of frontier nodes. Then comes the Map phase, which distributes the units of work among the processors of a convenient virtual machine. In the example, the virtual topology used is a tree. The Embed phase embeds that tree into the actual machine.

Once processors receive their tasks, they create a queue of tasks ready for execution. In the TSP example, the queue is prioritized according to the cost function associated with the nodes. Because the queue stores the tasks, its handling is represented by the Partition phase. Thus, Partition represents the process of selecting a task from the queue as well as that of storing a task in the queue. According to the queue paradigms, queues can be centralized, distributed, and a combination of both [8, 15, 22]. We assume the distributed case for it represents the most scalable as well as the most challenging approach for distributed-memory computers [26]. In this case, processors maintain a local queue of tasks. Processors execute the tasks in their local queue as long as the system is moderately loaded —i.e., the queue length is between some lower and upper bounds.

While moderately loaded, each processor takes for execution the task with the highest priority —i.e., the node with the least expensive cost— it has available in the queue. The execution of the tasks corresponds to the Solve phase. Processors may also need to share some information. In the branch-and-bound example, this corresponds to the updating of the bound based on the solution of least cost that the processor knows so far. To this end, upon finding such solution, a processor sends a message to the other processors so that they can update their bounds. This is essential to discarding unproductive work. Updating of variables or, in general, sharing of information, is taken care of by the Communicate phase. Also, the virtual topology used for transferring tasks may not be necessaryly the same as the one used to sharing information. Thus, each phase involving interprocessor communication —i.e., Map

```
Initialize;
begin
    Partition;
    Map;
    Embed;

    while (queues are not empty)
        while (lower bound < local queue length < upper bound)
            Partition;
            Solve;
            if (exchange of data is required)
              Communicate;
        end
        Map;
    end
    Communicate;
end
```

**Figure 6**: PMESC framework for the dynamic approach

and Communicate— may need a different virtual machine. Embedding these virtual machines into the real ones is taken care of by the Embed phase.

When a processor becomes overloaded —i.e., when the queue length exceeds the upper bound— or underloaded —i.e., when the queue length is less than the lower bound— the processor activates the load balancing mechanisms to transfer some work to or to get some work from another processor. This is work redistribution and, therefore, it corresponds to the Map phase.

The cycle repeats until all the local queues become empty. In that case, the processors may invoke some mechanism to combine the partial results. This corresponds to the Communicate phase.

## 9 Composing Models

The programming paradigms or models that we describe in this paper are not mutually exclusive. In fact, one model can be nested into another model, allowing a hierarchy or composition of programming abstractions. A typical case of model composition occurs when one applies a model to describe a problem and then uses other models to describe some of its subproblems.

Programming models are meant to provide programmers with a better understanding of the problems and their characteristics. Therefore, model composition is important for making the most effective use of these abstractions. The more models we can apply to a problem the better understanding we may have about how to efficiently implement that problem.

Next, we describe one example that shows how the composition of two models describes a problem better than each one of them.

```
Initialize;
begin
    for $i = 1$ : number of grid refinements
        begin new grid
            Partition of the grid into units;
            Map units to the processors interconnected as a virtual grid;
            Embed grid into real machine;
            Solve iterative method
                begin Solve phase
                    for $i = 1$ : number of iterations
                        Compute solution on units;
                        Aggregate neighbors' boundaries;
                        Broadcast own boundaries;
                    end
                end Solve phase
            Communicate data for new refinement;
            [Embed if topology used is other than grid;]
        end grid
    end
end
```

**Figure 7**: PMESC and CAB frameworks for the multigrid example

## 9.1 Composing PMESC with the CAB Paradigm to Solve Adaptive Irregular Multigrids

The problem of adaptive multigrids presented in section 7 provides us with a good case where we can apply model composition. For this example, the PMESC framework provides a high level description of the problem structure, which separates the work in stages. PMESC highlights the main steps involved in the computation of the whole problem, i.e., the set of grids. However, although it structures the computation of all the grids, it does not specify how to organize the computation of each one of them. The solution of the individual grids can be explained in more detail by another structure: the CAB paradigm.

In fact, the grid problem is about computing the values of variables at every point of a grid based on the values at neighboring points by following a iterative procedure. This is composed of three phases: (1) a compute phase in which processors estimate the solution of the problem on the different units and their boundaries, (2) an aggregate phase in which each processor receives the values of the boundaries from its neighbors in the grid, and (3) a broadcast phase in which each processor sends its values to their neighbors in the grid.

Figure 7 shows the framework for the adaptive multigrid example that combines PMESC and the CAB paradigm.

## 10 Conclusions

In this paper, we analyze the programming models that have arisen in distributed-memory computing as a consequence of theoretical research and practical experience. Emerging from this review are three key results: a new abstraction for structuring the algorithms on distributed-memory computers, a set of methodologies for implementing those algorithms, and a library for managing task-parallel algorithms that need a dynamic approach for parallelization. Of these, the present paper addresses the first two results. We present and evaluate the library in [9].

Our efforts are driven by several goals: to enforce the development of high quality algorithms, to support the development of libraries, to enforce good programming techniques, and to make the overall program design and implementation less time consuming. We believe that the PMESC programming paradigm and other concurrent efforts mentioned in this paper as well as tools such as MPI are fundamental to achieving these goals.

## References

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994.

[2] K. Batcher, *Sorting networks and their applications*, in Proceedings of the AFIPS Spring Joint Computer Conference, vol. 32, AFIPS Press, 1968, pp. 307–314.

[3] M. Berger and S. Bokhari, *A partitioning strategy for non-uniform problems on multiprocessors*, IEEE Transactions on Computers, C-36(5) (1987), pp. 570–580.

[4] F. Berman and L. Snyder, *On mapping parallel algorithms into parallel architectures*, J. Par. Dist. Comput., 4 (1987), pp. 439–458.

[5] S. Bokhari, *On the mapping problem*, IEEE Trans. Computers, C-30 (1981), pp. 207–214.

[6] K. Chandy, *Concurrent program archetypes*, Tech. Rep. RND-93-005, California Institute of Technology, 1993.

[7] K. Chandy and S. Taylor, *An Introduction to Parallel Programming*, Jones and Bartlett Publishers, Inc, 1992.

[8] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, The MIT Press, Cambridge, Massachusetts, 1989.

[9] S. Crivelli and E. Jessup, *The PMESC programming library for distributed-memory MIMD computers*, Tech. Rep. CU-CS-814-96, Dept. of Computer Science, University of Colorado at Boulder, 1996. Submitted to Journal of Parallel and Distributed Computing.

[10] J. Cuppen, *A divide and conquer method for the symmetric tridiagonal eigenproblem*, Numer. Math., 36 (1981), pp. 177–195.

[11] J. Dongarra, J. DuCroz, , S. Hammarling, and R. Hanson, *An extended set of Fortran basic linear algebra subprograms*, ACM Trans. Math. Soft., 14 (1988), pp. 1–32.

[12] J. Dongarra and D. Sorensen, *A fully parallel algorithm for the symmetric eigenvalue problem*, SIAM J. Sci. Stat. Comput., 3(2) (1987), pp. 139–154.

[13] J. Dongarra and D. Sorensen, *SCHEDULE: Tools for developing and analyzing parallel Fortran programs*, in The Characteristics of Parallel Algorithms, L. Jamieson, D. Gannon, and R. Douglass, eds., The MIT Press, Cambridge, Massachusetts, 1987, pp. 363–394.

[14] *The Etext Document*, 1996. Document in progress. The etext document can be obtained at http://www.etext.caltech.edu.

[15] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Publishing Company, 1995.

[16] I. Foster and K. Chandy, *FORTRAN M: A language for modular parallel programming*, Journal of Parallel and Distributed Computing, 26(1) (1995).

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing*, The MIT Press, 1994.

[18] A. Grama and V. Kumar, *Parallel processing of discrete optimization problems: A survey*, tech. rep., Department of Computer Science, University of Minnesota, 1992.

[19] W. Gropp and D. Keyes, *Domain decomposition on parallel computers*, Tech. Rep. YALEU/DCS/RR-723, Yale University, 1989.

[20] ——, *Domain decomposition with local mesh refinement*, SIAM Journal of Scientific and Statistical Computing, 13(4) (1992).

[21] W. Gropp, E. Lusk, and A. Skjellum, *USING MPI Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1995.

[22] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, N.Y., 1984.

[23] L. H. Jamieson, *Characterizing parallel algorithms*, in The Characteristics of Parallel Algorithms, L. Jamieson, D. Gannon, and R. Douglass, eds., The MIT Press, Cambridge, Massachusetts, 1987, pp. 65–100.

[24] M. Jones and P. Plassmann, *Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes*, in Proceedings of the 1994 Scalable High-Performance Computing Conference, IEEE Computer Society, 1994, pp. 478–485.

[25] J. D. Keyser and D. Roose, *A software tool for load balanced adaptive multiple grids on distributed memory computers*, in Proceedings of the Sixth Distributed Memory Computing Conference, 1991, pp. 122–128.

[26] V. Kumar, A. Grama, and V. Rao, *Scalable load balancing techniques for parallel computers*, Journal of Parallel and Distributed Computing, 22(1) (1994), pp. 60–79.

[27] A. KWAN AND L. BIC, *Using parallel programming paradigms for structuring programs on distributed memory computers*, in Proceedings of the Sixth Distributed Memory Computing Conference, 1991.

[28] J. LIPSON, *Elements of Algebra and Algebraic Computing*, Addison-Wesley, 1981.

[29] S. MANOHARAN AND P. THANISCH, *Assigning dependency graphs onto processor networks*, Parallel Computing, 17 (1991), pp. 63–73.

[30] S. MCCORMICK, *Multilevel Adaptive Methods for Partial Differential Equations*, SIAM, 1995.

[31] P. NELSON AND L. SNYDER, *Programming paradigms for nonshared memory parallel computers*, in The Characteristics of Parallel Algorithms, D. G. L.H. Jamieson and R. Douglass, eds., MIT Press, 1987.

[32] D. SCHMIDT, *Denotational Semantics A Methodology for Language Development*, Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[33] H. SIMON, *Partitioning of unstructured problems for parallel processing*, Computing Systems in Engineering, 2 (1991), pp. 135–148.

[34] H. SIMON, W. V. DALSEM, AND L. DAGUM, *Parallel computational fluid dynamics: Current status and future requirements*, Tech. Rep. RNR-92-004, NASA Ames Research Center, 1992.

[35] A. SKJELLUM, A. LEUNG, S. SMITH, R. FALGOUT, C. STILL, AND C. BALDWIN, *The Multicomputer Toolbox - First-generation scalable libraries*, in Proceedings of HICSS-27, IEEE Computer Society Press, 1994, pp. 644–654. HICSS-27 Minitrack on Tools and Languages for Transportable Parallel Applications.

[36] M. SNIR, S. OTTO, S. HUSS-LEDERMAN, D. WALKER, AND J. DONGARRA, *MPI The Complete Reference*, MIT Press, 1996.

[37] P. SWARZTRAUBER, *A parallel algorithm for computing the eigenvalues of a symmetric tridiagonal matrix*, Mathematical Computing, 20 (1993).

[38] H. TRIENEKENS, *Parallel Branch and Bound Algorithms*, PhD thesis, Department of Computer Science, Erasmus University, Rotterdam, 1990.

[39] R. WILLIAMS, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice and Experience, 3(5) (1991), pp. 457–481.

[40] M. WU AND D. GAJSKI, *Hypertool: A programming aid for message passing systems*, IEEE Transactions on Parallel and Distributed Systems, 1(3) (1990), pp. 330–343.

[41] Y. ZHANG, *Parallel Algorithms for Combinatorial Search Problems*, PhD thesis, Computer Science Division (EECS), University of California at Berkeley, 1989.