A Programming Paradigm and Library for Distributed-Memory Computers

Silvia Albornoz Crivelli

CU-CS-787-95

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A PROGRAMMING PARADIGM AND LIBRARY FOR DISTRIBUTED-MEMORY COMPUTERS

by

SILVIA ALBORNOZ CRIVELLI

B. S., Universidad Nacional del Litoral, 1979

M. S., University of Colorado, 1991

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1995

Crivelli, Silvia Albornoz (Ph. D., Computer Science)

A Programming Paradigm and Library for Distributed-Memory Computers

Thesis directed by Professor Elizabeth R. Jessup

This thesis is concerned with the implementation of task-parallel problems on distributed-memory computers. These problems are difficult to implement efficiently in parallel for they are asynchronous and unpredictable. A few languages and libraries have been proposed that are specifically designed to support this kind of computation. However, one big challenge still remains: to make those tools understood and used by scientists, engineers, and others who want to exploit the power of parallel computers without spending much effort in mastering those tools. Therefore, our primary goal is to make parallel programming on distributed-memory computers easy to understand and to make efficient parallel code easy to design. To that end, we present the PMESC programming paradigm and the PMESC library. The paradigm provides a methodology for structuring task-parallel problems that allows the separation of different phases in the computation. The library provides support for those phases that are application-independent allowing the users to concentrate on the application-specific ones. We evaluate the portability, performance, and ease of use of the PMESC library by subjecting it to a suite of tests in the context of different platforms and different examples. The tests show good results for a wide variety of situations and library uses.

DEDICATION

To my parents, my husband, Luis, and my two pearls, Bruno and Agostina, for their love and support.

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor, Liz Jessup. I was taught that anything can be achieved in life with conviction and hard work. However, I believe this is only partially true: one also needs luck. Liz represents a big portion of my luck. Her patience, guidance, encouragement, and support have been invaluable.

I would also like to thank Bobby Schnabel, for his valuable comments, Ben Zorn, for always having a word of encouragement and for his interesting course about languages, Dirk Grunwald, for his advice in load balancing, and Mike Lightner, for his helpful suggestions.

I would like to thank Clayton Lewis for his advice in testing the library, for the walkthrough analysis, and for helping me with the preparation of the user's manual. I truly appreciate that. Many thanks to Scott Baden for his guidance in building libraries and to Bill Gropp for running my experiments on the Delta.

Special thanks to Rich Neves and Greg Hill for helping me with the walkthrough sessions. Sharon Smith for her helpful comments on adaptive problems and for leaving a door open for me to explore. Thanks to Betty Eskow for her constant support and encouragement. I would also like to thank Carolyn Schauble for her technical and moral support and to Mark Maybee for being always ready to answer any question.

Finally, I would like to thank Luis Crivelli, for sharing with me his experience in data-parallel problems as engineer and software developer. Thanks

to him also for respecting my choices and supporting me throughout.

CONTENTS

FIGURES

TABLES

CHAPTER 1

INTRODUCTION

As parallel computers become 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. The reason why tools are so fundamental to parallel computing is that 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 interconnecting topology of the processors, load distribution, and data sharing. In addition, the rapidly evolving technology of these computers makes it necessary to develop software that can be ported from one machine to another without significant redesign.

Therefore, it is desirable to establish a machine-independent parallel programming model that is efficient for different applications on different architectures. Two complementary approaches to achieving this goal have been studied: parallel languages [14, 15, 23, 31, 42, 53, 77, 91] and libraries that expand existing standard languages [6, 7, 12, 15, 30, 36, 39, 41, 54, 71, 90]. Extensive efforts have been directed in both directions, and significant advances have been made. However, the results are still disappointing: a recent survey applied to a significant cross-section of the parallel user community shows that users simply do not find the current generation of tools useful for their program development needs [69]. They complain that tools are difficult to learn, tedious to use, and fail to provide the information that users need. The consensus is that the inadequacy of the parallel software is one of the reasons to blame for holding back the high-performance computing industry [67].

The big challenge is then not only to develop tools that are efficient and portable but also to make them understood and widely used by engineers, mathematicians, and other scientists who really want to exploit the power of these computers without having to spend a great deal of time mastering them. Our contribution in that direction is in the category of library packages. In particular, we want to address a kind of problem that is hard to implement but for which little support has yet been provided. Our primary goals are efficiency, portability, and ease of use. With this library, we do not intend to solve the 'software crisis' dilemma. Rather, our intention is to provide a working prototype library that meets the requirements of the Parallel Tools Consortium (Ptools) [70] of being usable as a standalone environment as well as a building block for future integrated parallel programming environments. Furthermore, because programming paradigms and techniques evolve, we prefer to present it as a basis subject to future refinement and growth, rather than as a final product.

1.1 Motivation

Parallel problems can be data-parallel or task-parallel or even a combination of both. Data-parallel problems present a large domain that can be decomposed into fine-grained units to be executed synchronously. Task-parallel problems present a complex task that can be decomposed into medium- to coarse-grained units to be executed asynchronously. Some other problems can be implemented efficiently by using a two-level subdivision that combines a task-parallel approach at the higher level with either a task- or a data-parallel approach at the lower one. Examples in this category are given by recent applications on heterogeneous computers. These are implemented by dividing very large scale problems into asynchronous and heterogeneous subproblems that are assigned to the different computers. Each computer, in turns, uses the appropriate approach on each of the subproblems.

A high-level analysis of all these problems shows that their implementation on distributed-memory computers involves the same basic steps:

- First, find parallelism. It is necessary to define the natural units of parallelism. This step is usually called domain decomposition in data-parallel computations and partitioning in task-parallel ones.
- This natural parallelism must take into consideration the limitations of parallel computers, e.g., the ratio between communication and computation costs. Too little parallelism results in idle processors; too much parallelism may result in high overhead associated with handling of the short-lived units. Thus, it is sometimes necessary to combine short-lived units into longer ones that are more convenient for distributed memory computers. The natural parallelism is thus transformed into practical parallelism.
- The units must be assigned to processors. For some parallel algorithms, static assignment is appropriate while, for others, dynamic load balancing is needed.
- Finally, if the units are not entirely independent, they need to share information. This sharing is synchronous in data-parallel problems and

asynchronous in task-parallel problems.

Although the high-level abstractions are basically the same, the actual implementation of data- and task-parallel problems is completely different. The strategies are different, the levels of complexity are different, the degree to which each one of the steps can be automated is different. Consequently, a tool intended for one type of problem cannot be applied efficiently to the other if it can be applied at all.

Although there has been a great deal of effort directed at providing general interfaces to solve data-parallel problems [31, 41, 54, 52, 53, 71, 77, 93], up to this point only a few attempts have been made to develop general tools for solving task-parallel ones. In fact, most of the software that has been developed for task-parallel problems is aimed at particular subsets of these problems such as discrete optimization problems [30, 90], global optimization problems [88], or parallel adaptive quadrature problems [51, 76]. The more general systems that have been developed to date [23, 71] present other drawbacks. One of the problems is that they require user experience. Another is that they restrict the control that the programmer has over the parallel application, reducing her or his ability to increase the efficiency of a program using that tool. That is, they hide parallelism from the programmer, rather than considering it a fundamental ingredient of programming design.

Therefore, it is still necessary to provide the parallel programmer with a tool to fully exploit the capabilities of distributed-memory computers to solve task-parallel, asynchronous algorithms. The necessity is based on the fact that these types of problems are difficult to parallelize and even more difficult to parallelize efficiently. This tool should provide support for some issues such as partitioning, load balancing, termination checking, and communication, allowing the programmer to concentrate on the application itself. This tool must also be easy to use.

1.2 Our Approach

This research started with the implementation on distributed-memory MIMD computers of the bisection method for computing eigenvalues. (Refer to Chapter 6 for a description of the bisection procedure.) The problem presents straightforward parallelism, but it is usually irregular and unpredictable. Therefore, it requires an adaptive approach that can dynamically respond to the changes in the computation. Much relevant work in the area of adaptive algorithms had been done by that time. One paper by Eager et al. demonstrates that adaptive algorithms that use dynamic load balancing techniques can, in fact, outperform static algorithms that use no redistribution at all. Also the thesis of S. Smith on "Adaptive Asynchronous Parallel Algorithms in Distributed Computation" [88] identifies several problems in the literature that, like bisection, exhibit irregular structure and require an adaptive approach.

At the same time, many tools were being developed to solve adaptive algorithms [4, 52, 30, 31, 36, 41, 50, 71, 90, 93], However, most of those tools were directed at data-parallel problems. Such tools spend a great deal of time on the partitioning of the domain. They also implement a synchronous approach as the only way to achieve correct results. Therefore, the type of support that they provide cannot be applied efficiently to bisection or other problems that are task-parallel. These problems do not need synchronization to guarantee correctness, and so they can benefit from the use of an asynchronous approach.

Nevertheless, some of the tools aimed at data-parallel problems did provide some background useful in attacking task-parallel ones. In particular, S. Baden developed a library (that evolved into LPARX) based on the separation of the execution process into three phases: partition, mapping, and computation. The use of these abstractions and the analysis of the implementation of different task-parallel problems led us to a new, more flexible and more general template: the PMESC paradigm.

Our research focused then on the task-parallel problems, the building blocks that compose them, and the strategies to implement those building blocks. Many different algorithms and strategies had been proposed in the literature. However, the conclusions about which one to implement for the library were not definitive: some of them perform well on some problems, some are better for other problems. The solution was to provide not one but several strategies for the same programming issue so that the programmer could select the ones that were more appropriate for the particular application.

The PMESC paradigm introduced in this thesis and the library based on it are the result of this research. The PMESC paradigm provides a mechanism for structuring problems by separating the different programming issues involved in their computations. The PMESC library provides a set of building blocks that users put together to conform with the applications and the machines. Both offer an environment that frees the programmers from dealing with application-independent issues allowing them to concentrate on the application-specific ones. PMESC is intended for both the inexperienced and the experienced programmer. It assists the inexperienced programmer in writing portable and quite efficient code within the context of a familiar language (Fortran or C) and without the burden of learning the machine architecture details. It assists the experienced programmer by providing a platform for testing new applications and for comparing different strategies.

1.3 Organization

The remainder of the thesis is organized as follows. Chapter 2 discusses paradigms in general, and the importance of recognizing them. It discusses related work and presents some examples. It also introduces the PMESC paradigm as a "way of thinking" about implementations of task-parallel problems for distributed-memory computers. Chapter 3 examines the different types of problems and the different approaches for parallelizing them. It shows how the PMESC paradigm applies to the different cases and presents illustrative examples of each case. Chapter 4 sets the ground for the PMESC library. It describes the different strategies investigated in this research for embedding of different topologies into different machine architectures, global combine operations, termination detection, and dynamic load balancing.

Chapter 5 introduces the PMESC library. It discusses related work, the motivations for developing a new library, the library approach for handling task-parallel problems, and the modules that compose the library. Chapter 6 demonstrates how to use the library in a variety of situations. It describes in detail the implementation of a suite of examples with the PMESC library. Chapter 7 examines the criteria to use to evaluate the portability, efficiency, and ease of use of PMESC. It begins the evaluation of the library by examining its portability and ease of use. Chapter 8 evaluates the performance that can be achieved with the PMESC library. Chapter 9 compares PMESC with a similar package called Charm. Finally, Chapter 10 presents conclusions about this work and topics for future research.

CHAPTER 2

PROGRAMMING PARADIGMS

As experience in paralleling processing has grown, a number of 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. Paradigms facilitate the efficient use of parallel computers for they provide models that can be used to formulate the applications. Because the essential computational structure of these techniques is already known, programmers only need to work on the problem specific details to produce a new program.

The purpose of this chapter is to emphasize the significance of paradigms as programming tools and to present a new paradigm. The chapter is organized as follows. Section 2.1 discusses paradigms in general. Section 2.2 presents some examples. Section 2.3 analyzes the viability of an universal approach. Section 2.4 presents the PMESC paradigm. Section 2.5 describes the PMESC programming model. Finally, Section 2.6 discusses the PMESC framework.

2.1 Paradigms

Paradigms are not actual algorithms, but rather they are problem solving strategies that are frequently used in structuring the algorithms. Thus, paradigms are the high-level methodologies, skeletons, or frameworks that we recognize as common to many algorithms [57]. They 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 [80]. Like higher-order functions, paradigms do not concern themselves 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 skeletons.

An important feature of many paradigms is that they enforce a modular style of programming that is highly beneficial in parallel processing. The independence of the modules breaks the implementation process into several smaller parts, each dealing with a particular task. The programmer may concentrate on implementing each individually, without having to consider the whole range of possibilities associated with the implementation of the entire code. This modularity makes programming not only easier and less error prone but also more flexible as it allows changing of the modules without a great deal of redesign.

Paradigms are useful tools for software developers for 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 from previous implementations as well as the reuse of code. In particular, paradigms encapsulate information about useful communication patterns. Thus, they can be used to identify which patterns are more useful and, therefore, should be well supported by computer designers and software developers.

In the next section, we present examples of paradigms that are suited

```
while (tasks are available)
        {
        Get a task from queue;
        Execute the task;
        Place any newly created tasks on the queue;
        }
```

Figure 2.1: Framework for the queue paradigms

for parallel programming on distributed-memory computers.

2.2 Examples of Paradigms

A number of programming paradigms for parallel computation have been presented in the literature [57]. In particular, the compute-aggregate-broadcast (CAB) paradigm [57, 65, 66], the queue paradigms [16, 24, 95], the divide-and-conquer paradigm [1], and the systolic paradigm [16, 65, 66]. We next describe these paradigms and provide an example for each case.

2.2.1 The queue paradigms. The queue paradigms provide the abstractions to solve a common problem that arises when implementing algorithms on distributed-memory computers: how to decompose the original problem into subproblems or tasks and distribute them among the processors. These tasks, which can be dynamically created by each processor during program execution, are stored in a queue. The procedure can be represented by the framework shown in Figure 2.1. (Note that this framework, like the rest of the pseudo-codes in this thesis, represent code that runs on each one of the processors.)

Depending on what processors store and handle the queue, the queue paradigm can be centralized, distributed, and a combination of both [44]. Also, depending on whether the tasks have been assigned different priorities or not, the queue paradigm can be prioritized or non-prioritized [23]. In the centralized approach, a master processor stores and maintains the queue of tasks that are ready for execution. When a slave processor finishes a task, it sends to the master the new tasks that it may have created and asks for more tasks. In the case of a priority queue, the master processor accumulates the tasks in the queue according to their priorities and sends out the tasks with the highest priorities first. Figure 2.2 shows the framework for the centralized queue paradigm.

In the distributed approach, each processor maintains a local queue of tasks ready for execution. Processors execute the tasks in their local queues until the system becomes imbalanced. In that case, some work is transferred to compensate for the uneven distribution. Figure 2.3 shows the framework corresponding to this approach.

Finally, the hybrid approach uses aspects of both the centralized and distributed ones. It occurs when storage and maintenance of the queue is handled by different processors. An example of this case is called centralized mediation [88]. In this paradigm, processors store their local queues locally as in the distributed case. However, when a processor becomes overloaded (i.e., when the length of its local queue exceeds some threshold) it sends part of its queue to a processor called the mediator. Likewise, when a processor becomes idle or underloaded (i.e, when the length of its local queue is less than a threshold) it requests some work from the mediator. The mediator processor sends the tasks received from overloaded processors to the underloaded processors that requested them. A variation of this hybrid approach, that we called centralized controller, is intended to reduce the overhead of moving large amounts

```
if (**master** processor)
{
 while (queue is non-empty)
 {
       maintain the queue (based on priorities if necessary);
       if (slave processor sends tasks)
       {
         receive tasks;
         add tasks to queue;
         select tasks from the queue;
         send tasks to processor;
       }
       if (slave processor requests tasks)
       {
         select tasks from the queue;
         send tasks to processor;
       }
 }
 send termination message to all processors;
}

if (**slave** processor)
{
 while (termination message is not received)
 {
       receive tasks from master processor;
       execute task;
       send newly created tasks to master processor;
       request more tasks;
 }
}
```

Figure 2.2: Framework for the centralized-queue paradigm

```
{
 while (local queue is non-empty)
 {
       select task from local queue;
       execute task;
       add newly created tasks to local queue;
       if (a processor requests tasks)
         send tasks to processor;
 }
 get tasks from another queue;
}
```

Figure 2.3: Framework for the distributed-queue paradigm

of data to the mediator. Processors store their local queues but these are controlled by the controller. Processors keep it informed of all the tasks that they create and execute, but they never actually send tasks to the controller. When the controller decides that a processor has to release some work, it sends a message to that processor requesting that it transfer some tasks to another designated processor. That way, the controller does not handle the tasks. It only handles the information about them.

For distributed-memory computers, a centralized queue and even a hybrid queue result in non-scalable applications [56]. In contrast, the distributed queue paradigm supports an approach that keeps the queue structure and the communication necessary to maintain it distributed among the processors and, therefore, seems to be more suitable for distributed-memory computers [56].

Instances of the centralized queue paradigm can be found in those multiprocessor operating systems that follow a master/slave organization [28]. In this case, a designated central processor executes all the privileged operations while slaves perform jobs that are delegated by the master.

2.2.2 The CAB paradigm. An example that illustrates clearly how programming paradigms provide a method for structuring parallel computation is given by the CAB paradigm. The CAB paradigm consists of three phases: compute, aggregate, and broadcast. In the compute phase, each processor executes its assigned portion of the problem, producing some results. The compute phase may differ widely from application to application. It may be a very complex algorithm or just a few operations. In the aggregate phase, the partial results are combined into a global result. In the broadcast phase,

```
Initialize;
for $i = 1$ : number of cycles
{
   begin {cycle}
          Compute;
          Aggregate;
          Broadcast;
   end    {cycle}
}
```

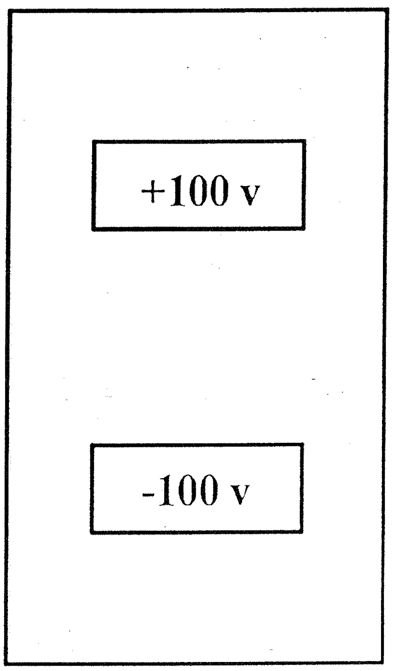
Figure 2.4: Framework for the CAB paradigm

global information necessary for the next compute phase is sent to each processor. A CAB algorithm can be represented by the framework in Figure 2.4.

The aggregate phase is usually a tree-based gathering operation that combines data from the processors, producing a global value [66]. This global value, or some information based upon it, is sent to the processors in the broadcast phase. Processors then can proceed with the next compute phase. Thus, this paradigm applies to those computations in stages that require synchronization at the end of each stage.

An instance of the CAB paradigm is given by the parallel implementation of the Jacobi iterative method for solving Laplace's equation on a rectangle [65]. The example in [65] solves the electric field problem depicted in Figure 2.5. The domain is discretized by laying a mesh on top of it. The numerical solution is obtained by computing the voltage $V_{i,j}$ at the points (i, j) in the mesh. To begin, an initial guess is computed for each point. Then, successive iterations compute a new value at each point as the average of the values of its four neighbors. The iterative process terminates when the difference between the new and the old value at every point is less than a given tolerance.

Figure 2.6 depicts the pseudo-code for this problem. The compute phase comes first. It connects the processors in a mesh so that each processor



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

$$V_{ij}^{n+1} = \frac{1}{4}(V^n_{i-1j} + V^n_{i+1j} + V^n_{ij-1} + V^n_{ij+1})$$

0 on boundary

Figure 2.5: Electric field problem: A CAB algorithm

computes a point of the discretized domain. Observe that in order to make the approach independent of the computer architecture, problems are formulated in terms of virtual machines whose processors are interconnected by virtual communication channels. Virtual machines are possible because the messages are automatically routed from any processor to any other processor. For instance, the natural virtual topology to use for this phase is a mesh as processors exchange their value $V_{i,j}$ with their four neighbors. Then, they calculate the new value, $V'_{i,j}$, by taking the average of the values received. Finally, they compute the difference between the new and the old value. The aggregate phase comes next. It connects the processors in a virtual binary tree and computes the global maximum of the differences between old and new values as follows. Every processor that corresponds to a leaf of the tree sends its difference to its parent. A processor that corresponds to an internal node of the tree evaluates the maximum of its difference and its children's differences and send it to its parents. The root processor, i.e., the processor that corresponds to the root of the tree, computes the global maximum. Finally, depending on the global maximum, the root sends a message to the other processors which corresponds to the broadcast phase. If the global maximum is less than the tolerance, the root broadcasts a termination message. Otherwise, it broadcasts a continuation message.

In the CAB paradigm what is important is 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.

```
{
 while (Continuation message received)
        *** Compute phase ***
        {
        Mesh(Neighbors); *Obtain neighbors in a mesh*
        Exchange(V's) with Neighbors;
        NewV = $(V_{i-1,j}+V_{i+1,j}+V_{i,j-1}+V_{i,j+1})/4$;
        Dif = abs(NewV - V);
        V = NewV;
        }

        *** Aggregate phase ***
        {
        Tree(Parent,Children); *Obtain neighbors in a tree*
        if (internal processor)
          Receive(Dif) from children;
        Dif = max(Dif, Received Dif's);
        if (not root)
          Send(Dif) to Parent;
        }

        *** Broadcast phase ***
        {
        if (root)
        {
          if (Dif < Tolerance)
            Broadcast (Termination);
          else
            Broadcast (Continuation);
        }
        else
          Receive Continuation or Termination message;
 }
}
```

Figure 2.6: Pseudo-code for the Jacobi procedure

2.2.3 The systolic paradigm. In this paradigm, a problem is divided into subcomputations that are assigned to processors. The data flows through the processors until computation is completed. Systolic algorithms present characteristics that make them similar to CAB algorithms such as locality of communication and regular communication structure. However, the key characteristic of these algorithms is the concept of flow of data.

A simple example of systolic algorithm is given by the computation of the matrix-vector multiply $Ab = c$, where $c_i = \sum_{j=0}^{n-1} a_{ij}b_j$. Processors $P_0, \ldots, P_{n-1}$ are virtually connected in a ring structure, and the matrix $A = [a_{ij}]$, $0 \leq i, j \leq n-1$ is statically distributed by rows among the processors. The elements of b flow through the processors, visiting all of them. Initially, the i-th row of A and the i-th element of b are stored in the memory of processor P_i. At each step, a multiplication between an element of the row of A and the visiting element of b is performed by each processor and accumulated into a variable s_i that stores the intermediate result of the inner product. Then, the element b_i is cyclically shifted to next processor in the ring. After n steps, the ith element of vector c is in processor P_i. The process is illustrated in Figure 2.7.

2.2.4 The divide-and-conquer paradigm. This paradigm is well known in sequential and parallel computing. It applies to problems that can be divided into two or more smaller subproblems that can be solved independently. The subproblems are just smaller instances of the original problem, and their results have to be combined to produce the final result. Thus, the paradigm can be represented by a recursive procedure.

In a sequential computation, the subproblems are solved serially. Each

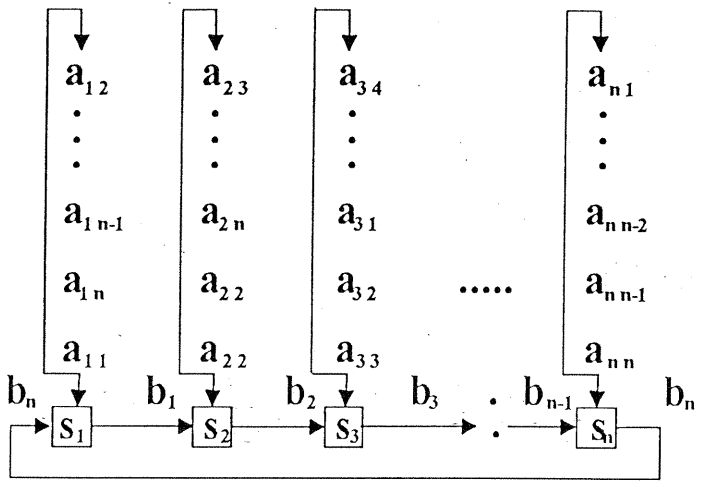


Figure 2.7: Matrix-vector multiply: A systolic algorithm

subproblem is completely solved before starting with the next one. In a parallel computation, the subproblems can be solved in parallel. Because the data and the results are distributed among the processors, their combination requires interprocessor communication.

An instance of this paradigm is given by the divide-and-conquer algorithm to multiply two dense $n \times n$ matrices $AB = C$ using n^2 processors, assuming that n is some integer power of two. The processors are interconnected by a $n \times n$ mesh and labeled P_{ij} for $0 \leq i, j \leq n-1$. The matrices A and B are initially distributed among the processors, with processor P_{ij} containing elements a_{ij} and b_{ij}.

Consider the case that $n = 2$. Each processor multiplies the elements it has, e.g., P_{00} computes $a_{00}b_{00}$, P_{11} computes $a_{11}b_{11}$, and so on. Then, each processor sends its a_{ij} value to the next processor in the same row and its $b_{i,j}$ value to the next processor in the same column. Every processor multiplies the newly received values and adds the result to the previous one to obtain the element c_{ij} of matrix C.

For the general case, the algorithm splits each $n \times n$ matrix and converts it into a 2×2 matrix whose elements are $\frac{n}{2} \times \frac{n}{2}$ submatrices. That way, it can apply to this matrix the same technique used for the case $n = 2$. (The only difference is that, in this case, the elements of the 2×2 matrix are submatrices.) The procedure splits the matrix recursively until the submatrices are 2×2.

The execution of a divide-and-conquer algorithm can be associated with a dynamically evolving tree of processes that has to be traversed down for the subdivision of the problem and then up for the computation of the final result. The algorithms are basically synchronous for it is not possible to solve any problem without having completed its subproblems.

Most paradigms presented in this section (except for the queue paradigms which do not specify) provide support for developing problems that are synchronous. However, we will see that there are other kinds of problems for which no synchronization points are necessary. Moreover, synchronization should be eliminated in these problems for it introduces a source of unnecessary overhead. It is necessary to find a more general methodology to address a wider spectrum of problems. The paradigm that we introduce in this chapter is intended to fill that hole.

2.3 A General Paradigm

A number of general programming techniques have emerged that help users to design and implement parallel applications. Each one is well suited to a certain type of problem but is inappropriate for others. A natural question is whether it is possible to create a universal technique, i.e., a paradigm that could be applied without restriction to solve any parallel problem on any parallel machine. Unfortunately, this seems to be 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" [16], M. Cole proposes an alternative solution: to build a new system that he calls **the skeletal machine**. This so-far-imaginary system would present not a single skeleton but a collection of independent ones.

In this system, the user would be presented with a menu listing the available skeletons. The user would select the skeleton that is appropriate for the problem at hand as well as the language in which she or he wants to specify the procedures. The system would respond by displaying the generic program that describes the structure of the skeleton 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 skeleton into the user application.

Although the notion of an abstract machine based on algorithmic skeletons is in an embryonic state, we believe that gathering a collection of paradigms is of fundamental importance if we want to take a first step in making parallel computers more accessible to a wide range of users. Most existing systems, either languages or libraries, provide no methodology to deal with parallel problems. Our approach is to propose a new paradigm and a library that is based on that paradigm. That way, we can provide both the methodology and the tool that implements it. We next present that paradigm.

2.4 The PMESC Paradigm

Parallel programming on distributed-memory computers involves several sources of difficulty. Among them is the task of problem decomposition or identification of parallelism. It is necessary to identify the processes that can operate concurrently in order to achieve a solution. The second problem is that of distribution, the physical realization of the parallelism identified in the decomposition phase. We must specify a mapping from processes into the available processors and indicate the mechanisms by which the mapping can be performed dynamically if necessary. Processors must then execute their assigned work. In the most favorable situation, the decomposition and distribution of a problem will lead to a situation in which processors have received a fair amount of work and can proceed with the execution until complete. However, in most cases, processors will 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. Because these mechanisms are so different we rather treat them as two separate problems. Finally, another problem to consider 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. These phases become the building blocks of distributed algorithms. To better understand the building blocks of this paradigm we next describe the computational model to which it applies.

2.5 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, M being a set of pairs of processors. 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., the low-level details of the parallel computation from the high-level ones, to achieve portable code. However, this approach may be very inefficient. 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). Of course, the main drawback of this approach is that it is not portable.

In order to be able to choose any variation between high efficiency and maximal portability, some 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 the next section we present the PMESC framework that we use as the backbone to develop distributed implementations.

2.6 The PMESC Framework

Figure 2.8 depicts the PMESC framework. The framework shows at a high-level the phases involved in the implementation of a parallel problem. First comes the identification of the units of work, i.e., the Partition phase. These units may be created either at the beginning of the execution or dynamically as the execution evolves. In the next step, units are assigned to the processors. This step corresponds to the Map phase. Upon assignment of the work, processors begin execution. This step corresponds to the Solve phase of the PMESC paradigm which may vary widely from problem to problem. Sometimes, some interchange of work may be necessary to keep the load balanced,

```
{
  Partition;
  Map;
  begin {cycle}
          Solve;
          Map;            (to balance loads if necessary)
          Communicate; (to share data if necessary)
  end   {cycle}
  Communicate;
}
```

Figure 2.8: Framework for the PMESC paradigm

and this step corresponds to the Map phase. Also, in some applications, interprocessor communication may be necessary to share information. This step corresponds to the Communicate phase. The process repeats in a cycle until the work is completed. At the end, processors may communicate to gather the partial results. The PMESC phases can appear in any order and number. In the PMESC paradigm, the identification of the phases is what matters, not their sequence.

In the next chapter, we describe different categories of parallel problems and show how the PMESC paradigm applies to all of them.

CHAPTER 3

A TAXONOMY OF PARALLEL ALGORITHMS

In this chapter, we describe the different categories of problems that exist and the different approaches for their resolution on distributed-memory computers. For each approach, we present illustrative examples and discuss the PMESC phases. The chapter is organized as follows. Section 3.1 presents a classification of the problems: data-parallel vs. task-parallel, regular vs. irregular. Section 3.2 categorizes the approaches for parallelization: static, quasi-dynamic, and dynamic. Sections 3.3, 3.4, and 3.5 analyze each one of these approaches in the context of different examples. These sections also show how the PMESC paradigm applies to the different approaches.

3.1 The Problems

Two categories of problems can be distinguished in parallel computing: data and task-parallel [47]. Each of these categories can be classified as regular or irregular.

3.1.1 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. Because data dependencies are strong, adjacent subdomains are usually assigned to neighboring processors (in the actual machine.) Thus, locality is an important characteristic of these problems. The subdomains can be executed in parallel by performing the same computation on each. However, processors must exchange data periodically and synchronously in order to achieve correct results. Models that represent this type of problem are SIMD (Single Instruction, Multiple Data) and SPMD (Single Program, Multiple Data).

In contrast, task parallel computations do not necessarily have a large data structure but rather a large task that can be partitioned into asynchronous subtasks to be assigned to the processors. Different subtasks may involve different computations. These tasks are self-contained for they do not need the outcome of other tasks. Eventually, processors may need to exchange some data or work, but 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), locality is not an issue for these problems. Task-parallel problems can be represented by the SPMD model and the MPMD (Multiple Program, Multiple Data) model.

Observe that some applications may involve the two paradigms, data- and task-parallel. An example is given by a problem where the number of subproblems is less than the number of processors available. In that case, more than one processor is dedicated to solving the same subproblem. The original problem is then subdivided twice. At the higher level, the problem is split coarsely into a convenient number of subproblems. At the lower-level, the subproblems are split finely among the processors in a group. The first subdivision is task-parallel while the second one is data-parallel. Another, more challenging case is presented by those large scale applications that are being 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.

3.1.2 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.

3.2 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 [93]. 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 well-suited 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 require periodic load balancing checks to achieve good performance. Dynamic approaches apply to those computations that are asynchronous and unpredictable and that require continuous, instead of periodic, load balancing checks.

In the remaining sections of this chapter we analyze each category in more detail, present some examples, and show how the PMESC paradigm applies to each one of them.

3.3 The static approach

The static approach can be successfully applied to those problems 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.

3.3.1 Example: block matrix-vector multiply. An example of the static approach is given by the block matrix-vector multiply $Ax = b$, where the matrix size is a multiple of the number of processors [34]. Figure 3.1 illustrates how the matrix A is divided into p = number of processors blocks and the blocks assigned to the processors. Each processor then computes its assigned portion of the vector b in roughly the same amount of time.

Static problems are not necessarily so simple. Many complex, static problems can be found on data parallel computations such as the solution of some PDE problems. In this case, the efficiency of the static implementation depends heavily on the use of a good partitioning strategy for domain decomposition and also on an efficient mapping strategy to assign the different subdomains to the processors. Many researchers have been working on efficient strategies for domain decomposition of PDE problems [37, 38, 84, 85]. Also, very good results have been obtained for efficiently mapping the subdomains onto the processors [8, 9, 11, 47, 63].

3.3.2 Applying PMESC to static problems.

Figure 3.2 illustrates how PMESC applies to static problems. In the diagram, the horizontal axis represents the processors and the vertical axis represents time. Thus, each column corresponds to the execution time spent by each processor. Connected columns represent phases in the computation involving some communication. For the static approach the diagram is very simple. There is an initial partition of the work into units. This procedure may run on a single processor or in parallel, depending on the application. Then the mapping procedure assigns those units to the processors connected by some convenient virtual topology —e.g., ring, binary tree, etc. The virtual topology is embedded into the actual machine architecture —e.g., hypercube, mesh. Next, each processor proceeds independently with its assigned part of the computation. In the matrix-vector multiply example, the Solve phase consists of performing the dot product of the assigned rows of A with x. At the end, processors may communicate to collect

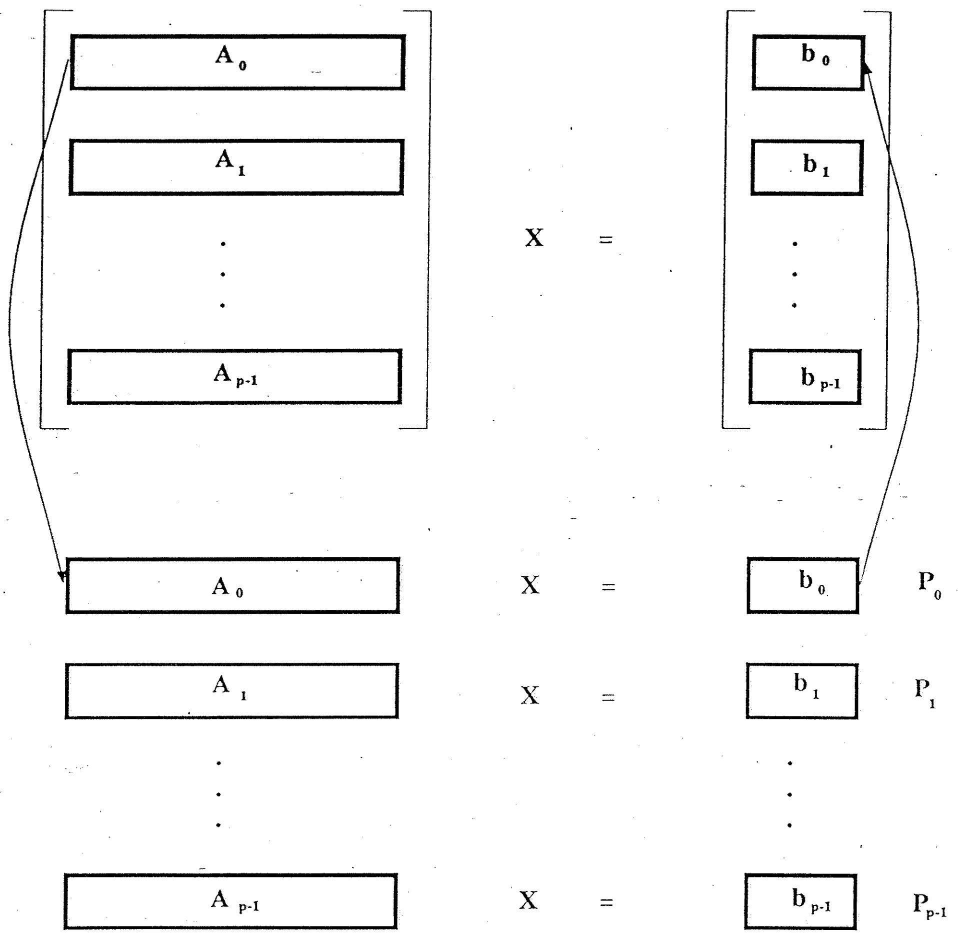


Figure 3.1: Block matrix-vector multiply

the partial results. For this phase, they may use a different virtual topology. Figure 3.3 shows the PMESC phases in the static approach.

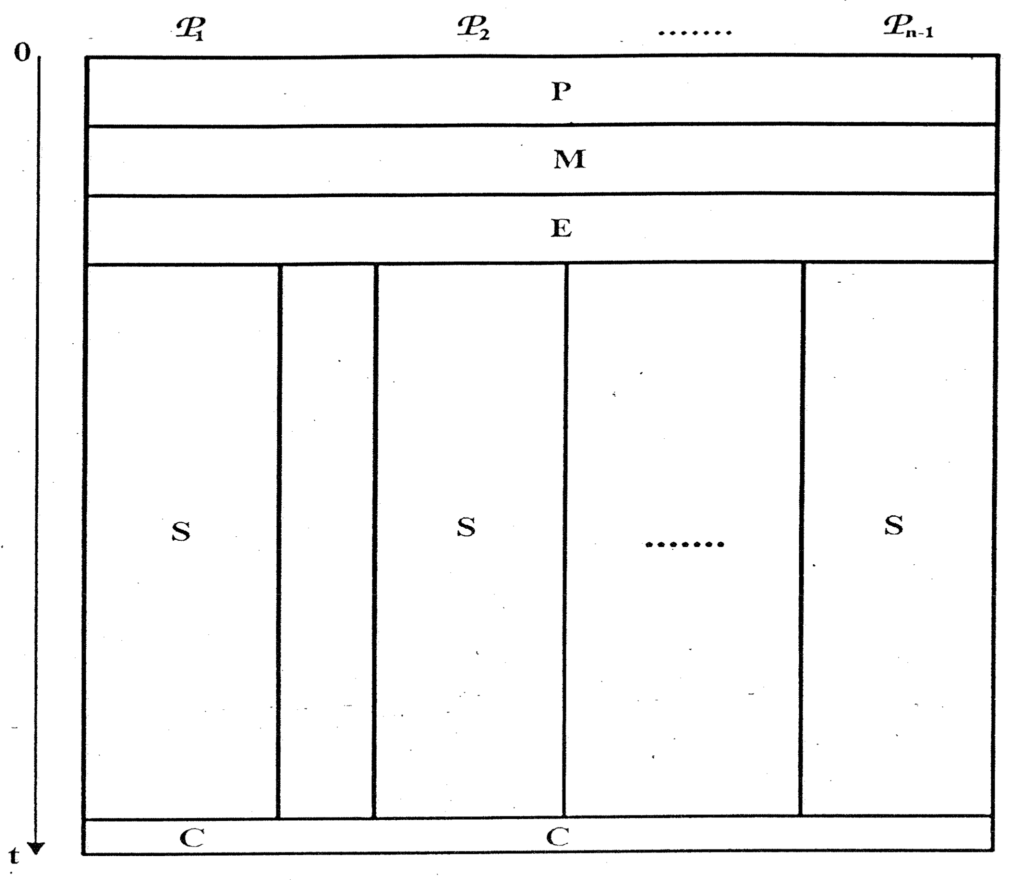


Figure 3.2: Diagram for the static approach

```
Initialize;
begin {
        Partition;
        Map;
        Embed;
        Solve;
        Communicate;
        [Embed;]
        }
end
```

Figure 3.3: PMESC phases for the static approach

3.4 The Quasi-Dynamic Approach

The quasi-dynamic approach applies to those data-parallel computations for which computational efforts should be concentrated on different parts of the domain during different parts of the execution. These variations occur slowly enough that imbalance appears in a gradual and predictable way. Thus, periodic, instead of continuous, remappings of work are necessary. The remapping of work occurs at convenient synchronization points in the computation that distinguish different stages. After each stage, the work estimate of the next stage can be determined. According to that estimate, the work is redistributed and mapped among the processors. Each stage is composed of the following steps:

- the domain is decomposed into subdomains,
- the set of subdomains is mapped into the set of processors,
- each processor runs the application on its assigned portion of the domain,
- each processor transfers work —if necessary— to keep the load balanced. Transfers are synchronous. They may be global, i.e., involving all the processors or non-global, i.e., involving only neighboring processors.
- processors exchange data —if necessary. This exchange is also synchronous.

After each stage, a new domain decomposition is considered, and another stage begins.

3.4.1 Example 1: a fast N-body problem. Examples of quasi-dynamic applications are found in computational fluid dynamics and particle physics. For instance, the following example originally presented in [3, 4] describes the flow of a fluid by computing the motion of a set of particles over a series of timesteps. These particles, which move under mutual interaction, congregate and disperse irregularly around the domain unpredictably with time. Because computational effort is associated with the number of particles, it is impossible to make a static distribution of work that keeps the load balanced.

The particles move under the influence of a potential which may be computed in two parts: a global part and a local part. Because the local attractions are physically more important than the global ones, we can ignore the global part. Therefore, communications only involve neighboring processors.

To partition the two-dimensional problem, a mesh is laid on top of the computational domain. Each meshbox contains the particles that lie in the region covered by the box. A uniform way to partition the work consists of splitting the domain into rectangular regions and assigning those regions to the processors. However, this approach is very inefficient because the number of particles in each box varies over the mesh and so does the amount of work associated with each box. Figure 3.4 illustrates a situation where the domain is decomposed into 16 rectangular regions to be assigned to 16 processors. Because particles are nonuniformly distributed in space, some processors become overloaded while others are idle.

Figure 3.4 also shows a better partitioning strategy that splits the domain into irregularly sized regions that have roughly the same amount of particles and so require similar computational times. This strategy compensates for the uneven distribution of work and can substantially improve the

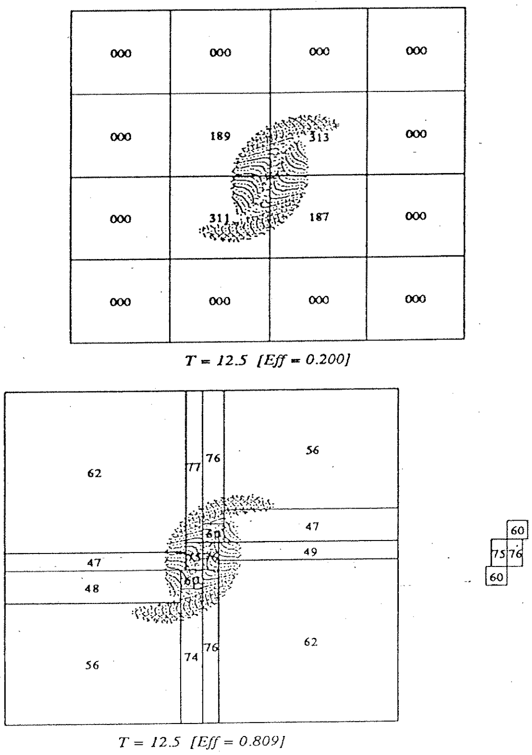


Figure 3.4. Uniform (top) vs. adaptive (bottom) partitioning of the domain into 16 subdomains. At the depicted time each subdomain's share of the work-load is shown in the subdomain assigned to it, normalized to 1000 units of total work. Source [4].

efficiency. However, it only solves the problem for a while. As time passes, particles move, redistributing themselves around the domain as illustrated in Figure 3.5. Therefore, partitioning and mapping of work must be periodically reconsidered. Otherwise, some processors would become overloaded while others would be idle waiting.

Because particles move slowly in time, their redistribution is a gradual process that only involves neighboring processors. Therefore, load balancing in this case is non-global. It is also synchronous in order to guarantee correct results. If mapping were asynchronous, then it would be possible for some particles to be in different positions on different processors at the same time. Consequently, results would be nondeterministic.

Thus, in the quasi-dynamic approach, the computation is divided into stages which are separated by synchronization points. At the end of each stage a new partition and mapping of work are considered based on the estimates of the workload for the next stage. This type of approach is reasonable when the irregularities appear gradually. There is enough predictability about the problem structure to be sure that load balancing will improve the situation for a certain amount of time.

3.4.2 Example 2: 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

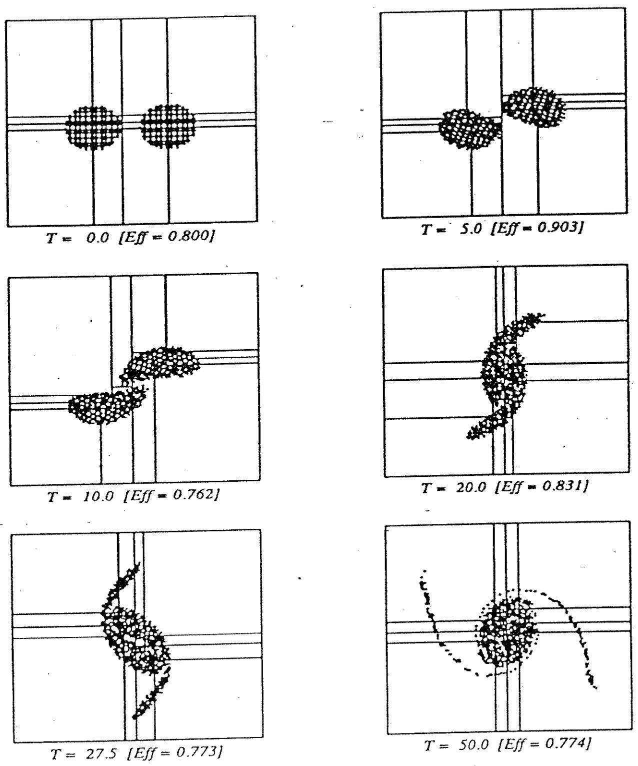


Figure 3.5. Dynamic distribution of the particles into 16 subdomains. Source [4].

discretized by a hierarchy of grids that have different resolutions [52]. 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, the partitioning consists of the decomposition of the grid into units (aggregates of nodes). The mapping of those units into the set of processors consists of the identification of the units that can be executed simultaneously —i.e., those units that do not have dependencies or temporal precedence constraints —and the subsequent distribution of them among the processors.

The optimal load distribution for each stage is determined by minimizing the estimated parallel execution time of that stage. For grid-oriented applications, the computation load for the next stage can be predicted reliably. The mapping procedure assigns computation and communication costs to the units, and these values are used 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. Any of these procedures requires global synchronization points.

3.4.3 Applying PMESC to quasi-dynamic problems. An important characteristic of the quasi-dynamic approach is that it applies to problems in which the variations in the load distribution occur gradually. Because these changes happen slowly and predictably, load imbalance can be check periodically and corrected with anticipation. The Solve phase, then, can be structured in between the periodic checks.

The two previous examples can be represented by the diagram in Figure 3.6. It shows how the computation is divided in stages, each with its corresponding Partition, Map, Embed, Solve and Communicate procedures. Again, the horizontal axis corresponds to the processors and the vertical to the execution time.

The first procedure in Figure 3.6 corresponds to the Partition phase. In the quasi-dynamic approach, it represents the domain decomposition. In the diagram, partition appears as a global procedure, but it may also run sequentially on a single processor, depending on the method used. Then comes the mapping procedure. It corresponds to the initial assignment of work to the processors as well as to the subsequent reassignments needed to keep the load balanced. These reassignments are always synchronous to guarantee deterministic results.

In the particle example, mapping is non-global, i.e., it involves neighboring processors only. In the multigrid example, mapping is global. Data dependencies are strong in these problems, and so processors exchange not only some work but also some data. This exchange is associated with the Communicate phase. In both examples the virtual topology used is a mesh. Thus, the Embed phase represents the embedding of the mesh into the actual machine architecture.

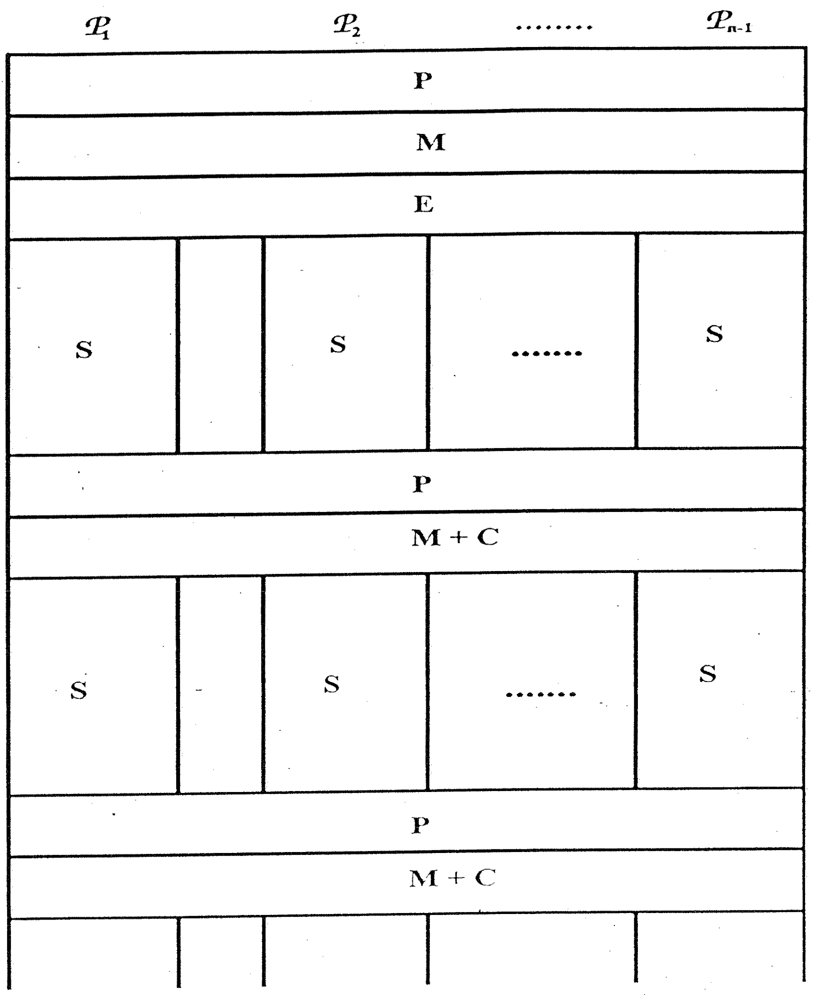


Figure 3.6: Diagram for the quasi-dynamic approach

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

Figure 3.7: PMESC phases for the quasi-dynamic approach

After partitioning, mapping, and communicating, processors can proceed independently with their computations for a while. These computations correspond to the Solve phase that continues until the system becomes imbalanced again. In that case, redistribution of work is necessary, and a new stage begins. The process repeats until computation is completed. Figure 3.7 shows the PMESC phases for the quasi-dynamic approach.

3.5 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 also 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.

Examples of problems that need a dynamic approach can be found in those computations involving some type of tree search [35]. 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. Combinatorial search methods, for instance, are complex, dynamic techniques used to solve problems that arise in such fields as artificial intelligence [96]. They consist of associating a rooted tree with a given problem. The execution of the algorithm corresponds to a search in that tree to find the goal 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 an expansion 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 solutions of its subproblems. Thus, when an expansion procedure is applied to node v, it either determines that v is a leaf or produces the children of v. The search recursively expands nodes until a set of leaves is identified as the desired solutions of the problem.

An efficient parallel implementation of a tree search must be adaptive and asynchronous, i.e., dynamic [48, 74, 75]. It needs to be adaptive in order to redistribute the subtrees as necessary to keep the load balanced. It also should be asynchronous in order to exploit the facts that tasks are independent and that little communication is required between the processors executing those tasks. In the next sections we discuss some search problems and the dynamic algorithms to solve them. We will see that a parallel search must address the issues of partitioning and distributing the work among the processors, solving the subproblems associated with the nodes, sharing some information, and embedding the tree into the actual machine architecture. Therefore, we will see that they are PMESC algorithms.

3.5.1 Example 1: backtrack search.

There is a class of dynamically created problems that can be solved by backtrack search [96]. Backtrack search is a search through a tree of partial solutions. When a partial solution cannot lead to a solution, it is necessary to terminate the unproductive search and backtrack to a point where another partial solution can be started. One example of backtrack search is given by the eight queens problem. This problem consists of finding all possible arrangements of eight queens on a 8-by-8 chess board in such a way that no queen can attack another, i.e., no two queens can be on the same row, column or diagonal of the board.

The backtrack search method generates partial solutions consisting of the arrangement of some queens so that no two of them can attack each other. Given a partial solution, it may be extended by placing another queen at the first available row in such a way that the newly placed queen does not attack the others. If such an arrangement is not possible, it means that the partial solution cannot be extended to a solution, and therefore it is necessary to backtrack. Backtracking is done by removing the last queen placed on the board and continuing with other possible placements that have not yet been attempted. This process repeats until all possible arrangements of the eight queens on the board are found.

3.5.2 Applying PMESC to parallel backtrack search. The execution of the backtrack search is based on the node expansion operation. When this operation is applied to a node v, it either determines that v is a leaf or generates the children of v. The node expansion operation can be applied simultaneously to several subtrees, which makes backtrack search especially suited for parallelism. Thus, to parallelize the backtrack search we need to assign a subtree to each processor. Good speedup can be expected if the processors are kept busy with nodes to expand. In other words, speedup can be obtained by keeping the load balanced. A parallel backtrack search procedure can be described as follows:

- **Partition:** The root is expanded by creating a set of frontier nodes, i.e., nodes that have been generated but not expanded.
- **Map:** The frontier nodes are distributed among the processors.
- **Partition:** Each processor creates a queue of nodes ready for expansion.
- **Map:** A processor is busy if its local queue of nodes is non-empty. Otherwise, it is idle. A processor is overloaded if its queue length is greater than a given threshold. A load balancing strategy must be applied to redistribute the load if necessary.
- **Solve:** The assigned subtree is traversed by the node expansion operation. Each node expansion operation generates more frontier nodes which are added to the local queue.
- **Communicate:** The final solutions, i.e., the goal leaves, are gathered at the end of the execution.
- **Embed:** The logical structure of the tree is mapped onto the physical machine.

Thus, parallel backtrack search can be described in terms of the five basic building blocks: Partition, Map, Embed, Solve, and Communicate.

3.5.3 Example 2: branch-and-bound. In parallel backtrack algorithms, each subtree can be searched without any knowledge of the outcome of other subtrees. However, this is not the case for all search algorithms. There is another method that uses the outcomes of other searches not to find the solution of the problem but rather to achieve good performance. This method, called branch-and-bound, uses a global bound to prune those branches of the tree that cannot produce a solution [89, 96]. It defines a cost function c which 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. An example is the traveling salesman problem (TSP). 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 σ of the set of cities $\{1, 2, \ldots, n\}$ that minimizes $\sum_{i=1}^{n} d_{i,\sigma(i)}$.

A branch-and-bound procedure consists of two parts bearing those names. The bounding procedure computes lower bounds on the costs associated with each subproblem. Thus, to bound a problem $\mathcal{A}$ the bounding procedure solves a simpler problem $\mathcal{B}$. In the TSP case, $\mathcal{A}$ represents the minimization problem subject to the constraint that the tour must have only one cycle, while $\mathcal{B}$ represents the same minimization problem with the tour allowed more than one cycle. Thus, if the solution S of $\mathcal{B}$ contains only one cycle, then S is also a solution to problem $\mathcal{A}$. Otherwise, the cost associated with S is used as a lower bound on the solution to $\mathcal{A}$.

The branching procedure takes a solution S that contains more than one cycle and breaks the smallest cycle by deleting one edge. Let us call D the smallest cycle in S and $e_1, e_2, \ldots, e_d$ its edges. The branching procedure creates subproblems $\mathcal{A}_i$ by deleting edge e_i, for $i = 1, \ldots, d$. The process repeats until all the leaves are found. Leaves correspond to subproblems that can be solved directly, i.e., tours that contain only one cycle. Then, a solution to the original problem can be found by solving the subproblems corresponding to the leaves and taking the solution of least cost.

When a leaf is found, its cost is used to prune the tree. This is done 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.

3.5.4 Applying PMESC to branch-and-bound search. A parallel branch-and-bound algorithm not only has to distribute the subproblems among the processors but also must ensure that processors do not waste their time exploring unproductive subproblems. Therefore, unlike the parallel backtrack search, a parallel branch-and-bound algorithm may not achieve good speedup by merely keeping the load balanced. Another difference from the backtrack search is that, 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 branch-and-bound algorithm can be described as follows:

- **Partition:** The root is expanded by creating a set of frontier nodes.
- **Map:** The frontier nodes are distributed among the processors.
- **Partition:** Processors maintain a priority queue of nodes ready for expansion.

- **Solve:** Each processor expands the node of least expensive cost it has available in the queue.
- **Communicate:** An updated bound for the cost is maintained by each processor based on the solution of least cost that the processor knows so far. Upon finding a goal leaf, a processor sends a message to the other processors so that they can update their bounds. This is essential to discarding unproductive work.
- **Embed:** The logical structure of the tree is mapped onto the physical machine.

Thus, the parallel implementation of the branch-and-bound search can be represented by the PMESC paradigm.

3.5.5 Applying PMESC to dynamic problems. The diagram in Figure 3.8 illustrates how PMESC applies to problems implemented with a dynamic approach. There is an initial partition and assignment of work to the processors using a convenient topology. This topology is embedded into the real machine architecture. Because this type of approach applies to task-parallel computations, processors split the work into units or tasks and store them in a queue from where they select them, one at a time, for execution. Processors execute the tasks asynchronously and independently until they become overloaded or idle, in which case they activate the mapping procedure in order to transfer some 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.

In some applications, the dynamic approach involves interprocessor communication, e.g., branch-and-bound. Communication is also asynchronous

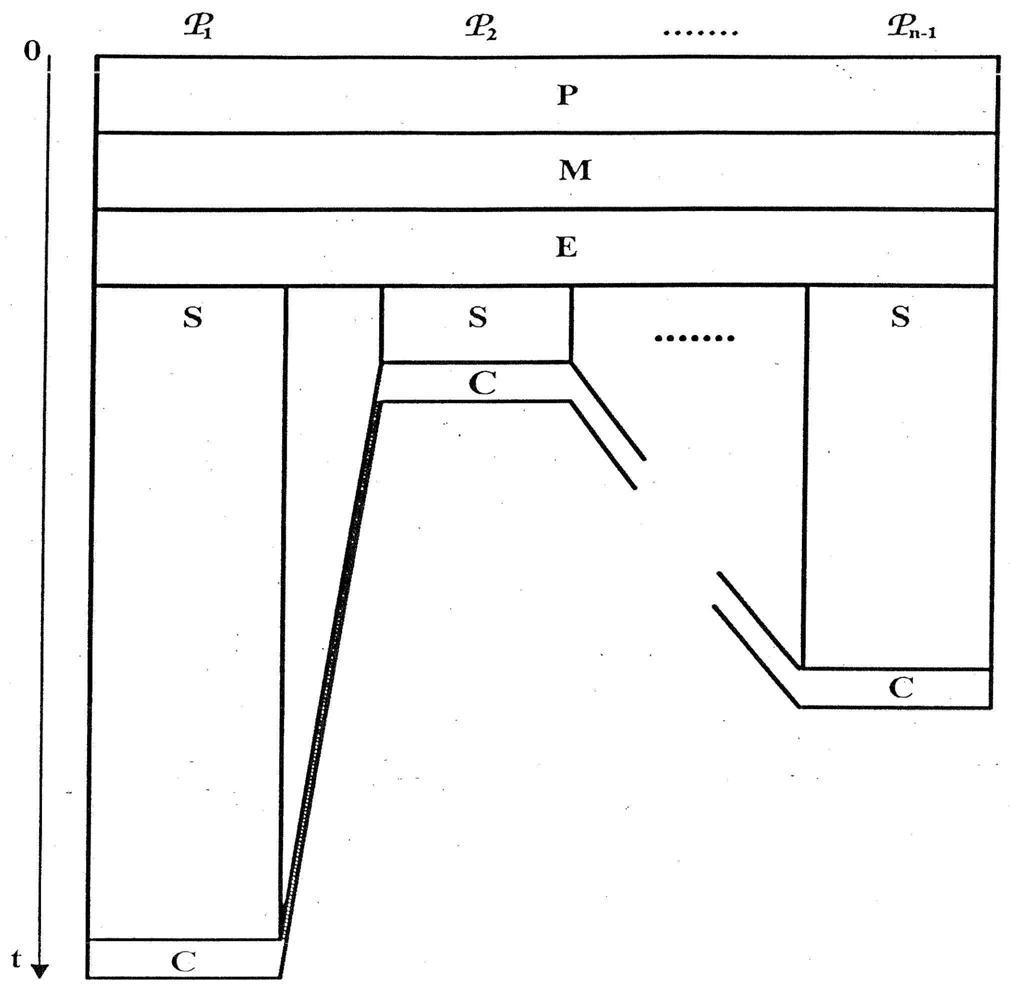


Figure 3.8: Diagram for the dynamic approach (before load balancing)

and can occur either during the execution or at the end or both. Communication corresponds to a procedure that distributes or gathers information that processors may need. In the backtrack procedure, communication occurs at the end of the execution to gather results. In the branch-and-bound procedure, communication occurs during execution to update the bound that processors use to prune some branches of the tree and to concentrate on the most promising ones. However, communication is asynchronous. That is what distinguishes these problems from the quasi-dynamic ones. Figure 3.9 shows how the overall system performance can be improved by shuffling some work from the heavily loaded processors to the idle processors via repartitioning and remapping. Note that the time t_1 in Figure 3.8 is less than the time t in Figure 3.9. In a real situation, t_1 can be substantially less than t.

Figure 3.10 shows a high-level pseudo-code for the dynamic approach based on the PMESC phases. This template shows that mapping and communication between processors, if necessary, are performed independently of the algorithm itself, i.e., the Solve phase. Thus, the Solve phase is always a sequential procedure which not only makes programming easier but also allows the reuse of sequential code.

A final aspect involved in the solution of dynamic problems is the termination checking. Because processors work asynchronously, idle processors need to check whether to continue searching for work or terminate. To that end, an idle processor activates the mechanisms for load balancing and these invoke, if necessary, the termination checking procedure. For that reason, termination checking is considered part of the Map phase.

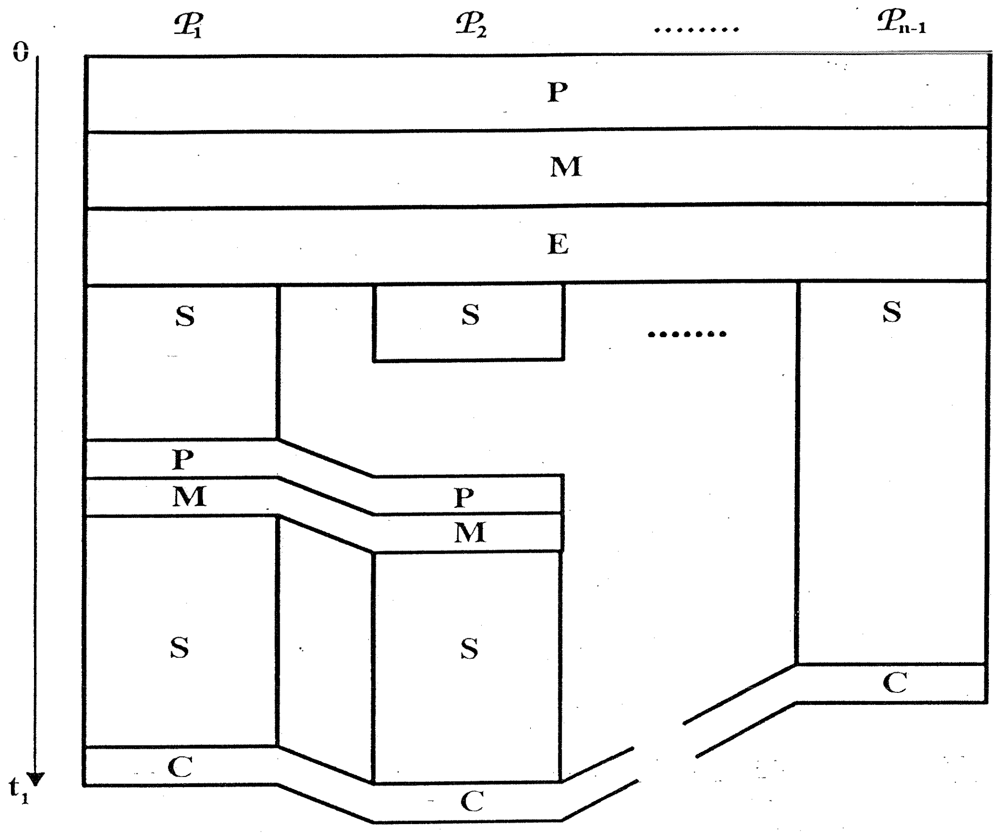


Figure 3.9: Diagram for the dynamic approach (after load balancing)

```
Initialize;
Partition;
Map;
Embed;

while (distributed-queue is non-empty)
        begin {cycle}
                Partition;
                Solve;
                if (communication is required)
                {
                  Communicate;
                  Embed;
                }
        end {cycle}
        Map;
        Embed;
end;
Communicate;
Embed;
```

Figure 3.10: PMESC phases for the dynamic approach

3.6 Conclusion

In this chapter, a taxonomy of the problems is presented in an attempt to provide a common terminology as well as to identify the approaches to parallelizing those problems. The taxonomy given allows the classification of the parallel problems and the approaches for parallelization according to a small set of salient features. This classification is also important to characterizing and understanding the different programming tools and to identify areas worthy of additional effort.

We present applications in each category and analyze their implementations. We show that the PMESC template can be applied to the different approaches. However, although the PMESC phases are the same, their implementation is different. We study the main programming issues involved in each one of the PMESC phases and identify the building blocks that compose those phases. Because the purpose of this thesis is to develop a library for the particular class of task-parallel problems, our next step is to find efficient algorithms to implement those building blocks. We describe those algorithms in the next chapter.

CHAPTER 4

COMMUNICATION AND EMBEDDING ALGORITHMS

Chapter 3 describes a category of problems that have irregular structure and can benefit from the use of adaptive asynchronous parallel algorithms. These algorithms involve complex programming issues whose efficient implementation is crucial to achieving good performance. This chapter presents a collection of strategies that we use in this thesis to address the issues of interprocessor communication and embedding of virtual into real architectures. The chapter describes several strategies proposed in the literature as well as our own ones, developed to improve the functionalities of existing mechanisms or their results.

This chapter is organized as follows. Section 4.1 presents different algorithms for embedding some commonly used virtual topologies into the available real architectures. Section 4.2 reviews the issues concerning the implementation of load balancing strategies. Section 4.3 covers different ways to detect termination of asynchronous processors. Section 4.4 describes an algorithm for efficient broadcasting and gathering. Section 4.5 analyzes an approach to maintain "global" information in a distributed environment. Chapter 5 discusses how these strategies are actually used to implement the PMESC library.

4.1 Embedding Algorithms

The programming approach that we adopt does not require that programmers formulate their computations to fit the physical communication links of the processors exactly. Instead, programmers can design their codes in terms of virtual processors and virtual communication channels that connect them, i.e., a virtual machine. It then becomes necessary not only to embed that virtual machine into the real machine architecture but also to embed it efficiently to avoid excessive network traffic. In this section, we describe the algorithms for efficiently embedding some virtual topologies. In particular, we discuss the embeddings of rings, trees, and arrays (uni- and bi-dimensional) onto hypercubes, meshes, and fat trees. We selected those virtual topologies because they are widely used in all kinds of scientific applications. In some cases, we provide different variations of a virtual topology depending on the real machine architecture. For instance, we provide a spanning tree for the hypercube, a tree of spanning trees (i.e., a spanning tree along each dimension) for the 2-D mesh, and a quaternary tree for the fat tree.

4.1.1 Embedding trees into hypercubes. A spanning tree can be embedded into the hypercube by using the algorithm presented in [79]. Because the neighbors' ids (identifiers) differ in exactly one bit, spanning trees can be spawned by toggling successive bits of the binary representation of the processors' ids. Bits can be toggled in any order, i.e., least significant bit first or most significant bit first. We illustrate the case in which the least significant bit is toggled first. In this case, the first level of the tree has the processor $0^{d-1}1$, where 0^{d-1} represents $d-1$ concatenated zeroes and d is the hypercube dimension. This processor is the root of the spanning tree. The i^{th} level of the tree contains processors whose most significant bits through bit $i+1$ are zero while the i^{th} bit is one. Each processor located at the level i spawns one child of level $i+1$, one of level $i+2$ and so on through level d. The resulting tree for a hypercube of dimension 4 is depicted in Figure 4.1. The algorithm is represented in Figure 4.2.

4.1.2 Embedding trees into 2-D meshes.

An extension of the algorithm shown in Figure 4.2 for hypercubes can be used in the 2-D mesh case. This algorithm, presented in [5], embeds a spanning tree along one dimension and then along the other, using the roots of the spanning trees corresponding to the first dimension as nodes of the spanning trees corresponding to the second dimension. In the mesh, every processor is identified by a single number in the range from zero to one less than the number of processors. This enumeration increases from left to right and from top to bottom. However, in order to find the neighbors of a processor in a virtual tree, it is necessary to identify this processor by its coordinates in the mesh. Each coordinate represents a dimension. The one-dimensional procedure shown in Figure 4.2 can then be applied to the coordinates in each dimension one at a time. The algorithm is depicted in Figure 4.3. It shows how the two-dimensional coordinates are computed and how they are used to embed a tree in each dimension.

4.1.3 Embedding quaternary trees into fat trees.

The topology of the CM5 is a quaternary fat tree. Figure 4.4 shows the interconnection pattern. Processors are located at the leaves of the tree. The internal nodes of the tree represent the data network, which provides point-to-point data communications between processors. This network is composed of several router chips. Each router chip is connected to four child chips and either two or four parent chips.

Figure 4.5 shows a routine that we created to embed a quaternary tree into the fat tree topology of the CM5. The algorithm is straightforward.

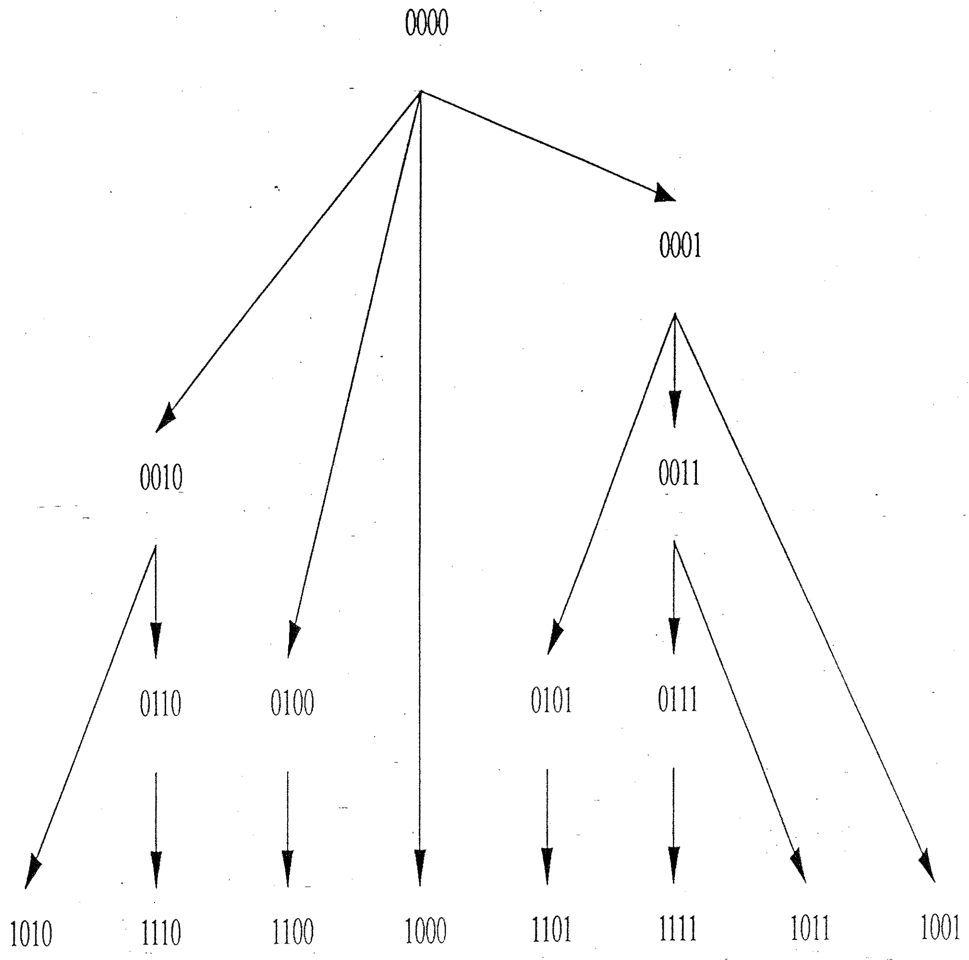


Figure 4.1: Spanning tree for a hypercube of dimension 4.

```
Tree (me, dim0, dim1, num_neigh, neighbors)
int me;              \* processor's id *\
int dim0;            \* hypercube dimension *\
int dim1;            \* dummy variable for symmetry with the 2-D case *\
int num_neigh;       \* number of neighbors of processor me in the tree
                        including its parent and children *\
int *neighbors;      \ vector containing the neighbors' ids *\
{
  int i, L;
  int parent;

  for (i=0; i<dim0; i++) {
      \* find first one (1) from right to left
         in the binary representation of me *\
      if ( ((me >> i) & 01) == 01) {
         parent = me & ~(1 << i); \* toggle first one (1)
                                     to determine parent's id *\
         break;
      }
      \* count the number of zeroes
         to determine the level of me in the tree
         and thus, the number of children *\
      else
         L++;
  }
  num_neigh = L+1;
  neighbors = (int *)malloc(num_neigh*sizeof(int));
  neighbors[0] = parent;
  for (i=0; i<L; i++)
      neighbors[i+1] = me | (1 << i); \* toggle zeroes
}                                        to determine the children's ids *\
```

Figure 4.2: Algorithm for embedding a spanning tree into a hypercube.

```
Tree (me, p0, p1, num_neigh, neighbors)
int me;               \* processor's id *\
int p0;               \* number of rows in the mesh *\
int p1;               \* number of columns in the mesh *\
int num_neigh;        \* number of neighbors in the tree *\
int *neighbors;       \* vector containing the neighbors' ids *\
{
  int i, L;
  int parent;
  int x[2], dim[2];
  int L[3];

  \* find coordinates (x[0], x[1]) corresponding to processor me *\
  x[0] = me / p1; \* row *\
  x[1] = me % p1; \* column *\

  \* function log2 returns the closest integer greater than or equal to *\
  \* the logarithm in base 2 of the argument *\
  dim[0] = log2(p0);
  dim[1] = log2(p1);

  L[0] = L[1] = L[2] = 0;

  neighbors = (int *) malloc( (dim[0]+dim[1]+1) * sizeof(int) );

  for (j=1; j>=0; j-) {
     for (i=0; i<dim[j]; i++) {
        \* find first one from right to left
           in the binary representation of one of the coordinates *\
        if ( ((x[j] >> i) & 01) == 01) {
          parent = x[j] & ~(1 << i); \* toggle first one *\
          break;                     \* to determine the parent's id *\
        }
        \* count the number of zeroes
           to determine the number of children of me in each dimension *\
        else
           L[j]++;
     }
     for (i=0; i<L[j]; i++)
        neighbors[L[j+1]+i+1] = x[j] | (1 << i);
     if (x[j] != 0) break; \* only the roots of the spanning trees in the first *\
  }                        \* dimension are considered for the second dimension *\
  num_neigh = L[0]+L[1]+1;
  neighbors[0] = parent;
}
```

Figure 4.3. Algorithm for embedding a tree of spanning trees into a 2-D mesh.

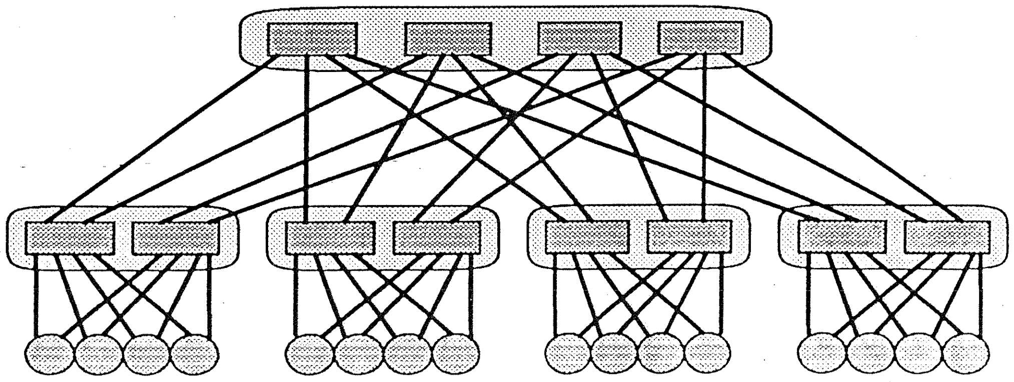


Figure 4.4. The physical topology of the CM5 is a quaternary fat-tree. Each internal node of the tree is made up of several router chips. Each router chip is connected to 4 child chips and either 2 or 4 parent chips. Source [59].

Processors are clustered in groups of four. The neighbors of a processor are all the other processors in its group. The parent processor is the lowest numbered processor in the group.

4.1.4 Embedding rings into hypercubes. This algorithm, presented in [79], embeds a ring of l vertices into a hypercube of $p = 2^d$ processors in a way that preserves the property that two adjacent vertices in the logical ring correspond to actual neighbors in the physical hypercube. Let us consider first the case $l = 2^d$. An efficient way to embed a ring into a hypercube is to connect the processors according to the binary reflected Gray code. There are different ways to generate Gray codes. We follow the procedure explained in [79]. We start with the sequence of the two digits $S_1 = \{0, 1\}$. It corresponds to the one-bit Gray code. To build a two-bit Gray code, we take the one-bit sequence and insert a zero in front of each number, then we take the one-bit sequence in reverse order and insert a one in front of each number. Thus, we get the sequence

$$S_2 = \{00, 01, 11, 10\}.$$

To get a three bit Gray code we repeat the process, taking the above sequence S_2 and inserting a zero in front of each number, then taking the reverse of S_2 and inserting a one in front of each number. The new sequence is

$$S_3 = \{000, 001, 011, 010, 110, 111, 101, 100\}.$$

In general, if we call S_i^R the sequence obtained by reversing S_i, and denote $0S_i$ or $1S_i$ the sequences obtained by inserting a 0 or 1 respectively in front of each element of the sequence S_i, then the expression for a binary reflected Gray code of order d becomes

$$S_d = \{0S_{d-1}, 1S_{d-1}^R\}. \tag{4.1}$$

```
Tree (me, num_proc, p1, num_neigh, neighbors)
int me;             \* processor's id *\
int num_proc;       \* number of processors *\
int p2;             \* dummy argument used for symmetry with the 2-D
                       mesh case *\
int num_neigh;      \* number of neighbors in the tree *\
int *neighbors;     \* vector containing the neighbors' ids *\
{
  int i, c;

  c = (int) me / 4;

  num_neigh = 4;
  neighbors = (int *)malloc(num_neigh*sizeof(int));

  for (i=1; i<num_neigh; i++)
      neighbors[i] = ( (me+i) % 4 ) + c*4; \* the children's ids *\

  neighbors[0] = c*4; \* the parent's id *\
}
```

Figure 4.5: Algorithm for embedding a quaternary tree into a fat tree.

```
Ring (me, num_proc, p1, num_neigh, neighbors)
int me;             \* processor's id *\
int num_proc;       \* number of processors *\
int p1;             \* dummy argument used for symmetry with
                       the 2-D mesh case *\
int num_neigh;      \* number of neighbors in the ring *\
int *neighbors;     \* vector containing the neighbors' ids *\
{
  num_neigh = 2;
  neighbors = (int *)malloc(num_neigh*sizeof(int));

  neighbors[0] = Gray(me, num_proc, -1);
  neighbors[1] = Gray(me, num_proc, 1);

  return;
}
```

Figure 4.6: Algorithm for embedding a ring into a hypercube.

Note that the first and last processors in this sequence are neighbors in the virtual ring as well as neighbors in the actual hypercube.

Suppose now that l is not exactly a power of two but an even number (the ring cannot be embedded into the hypercube if l is odd [79].) Let $j = l/2$, and let $S_k[j]$ denote the first j elements in the sequence S_k. Then, the Gray code sequence for this case is given by

$$\{0S_{d-1}[j], 1S_{d-1}[j]^R\}.$$

Note that when $l = 2^d$ this formula coincides with 4.1. The Ring and the Gray code routines for a hypercube are given in Figures 4.6 and 4.7.

4.1.5 Embedding rings into 2-D meshes. We next describe an algorithm proposed in [5] to embed a ring into a 2-D mesh. Figure 4.8 depicts a mesh of $16 = p_1 \times p_2 = 4 \times 4$ processors. Processors in a partition (rectangular submesh) are identified by a logical processor number in the range from zero to one less than the number of nodes in the partition, i.e., num_proc. As in the hypercube case, we want an algorithm that embeds a ring so that

```
Gray (me, num_proc, par)
int me;
int num_proc;
int par;
{
  int i, k, mi, Mi, c;
  int *s;

  s = malloc(num_proc*sizeof(int)); \* contains the gray sequence *\
  s[0] = 0;
  s[1] = 1;

  for (i=1; (1<<(i+1)) < num_proc; i++) {
      mi = 1 << i; \* each sequence has twice as many elements *\
      Mi = 2*mi;   \* as the previous one *\
      c = mi;
      for (k=mi; k<Mi; k++) {
          c--;
          \* generates half of the new sequence by reversing the order of the *\
          \* previous one and inserting a 1 in front of each number. *\
          \* The other half is exactly as the previous sequence *\
          s[k] = s[c] | (1 << i);
      }
  }
  \* the last sequence is generated separately to cover the case that
  \* num_proc is not a power of two *\
  mi = (num_proc-2)/2+1;
  Mi = num_proc + 1;
  c = mi;
  for (k=mi; k<Mi; k++)
      c--;
      s[k] = s[c] | (1 << i);

  for (i=0; i<num_proc; i++) {
      if (me == s[i]) { \* searches position of me in the Gray sequence *\
         pos = i;
         break;
      }
  }
  if (par == 0) return (pos); \* returns position of me in the Gray sequence *\
  if (par == 1) return (s[(pos+1)%num_proc]); \* returns the successor of me *\
  if (par == -1) return (s[(pos-1)%num_proc]); \* returns the predecessor of me *\
}
```

Figure 4.7: Algorithm for computing the Gray sequence.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 4.8: Distribution of processors in a mesh.

all neighbors in the ring are neighbors in the partition. The sequence starts at processor 0 and goes from top to bottom of the mesh in a serpentine fashion. It goes from left to right in the even rows (starting at row 0) and from right to left in the odd ones, without including the processors located at the first column except for those in the first and last rows. When the sequence gets the lower left corner of the 2-D mesh, it goes straight up from bottom to top. In the example of 4×4 processors, this algorithm should produce the following sequence of neighboring processors in the ring:

$$\{0, 1, 2, 3, 7, 6, 5, 9, 10, 11, 15, 14, 13, 12, 8, 4\},$$

with processor 4 connected to processor 0.

The Ring function for the mesh that produces this sequence is given in Figure 4.9. When applied to a processor id, the function OUT returns 0 if the id is out of the range of processors available, i.e., 0 to one less than the number of processors, and 1 otherwise. This algorithm does not apply when the number of rows is odd.

4.1.6 Embedding rings into trees. There is no technique that we can use to embed a ring into a fat tree in such a way that all neighbors in the ring are actual neighbors in the fat tree. Thus, to send a message from processor A to processor B, the message must go up the tree until it gets to the level where it can go down and reach processor B. However, because the tree is fat, the bandwidth is kept balanced and is not critically dependent on

```
Ring (me, p0, p1, num-neigh, neighbors )
int me;               \* processor's id *\
int p0;               \* number of rows *\
int p1;               \* number of columns *\
int num_neigh;        \* number of neighbors in the ring *\
int *neighbors;       \* vector containing the neighbors' ids *\
{
  int x0, x1;
  num_neigh = 2;
  neighbors = (int *)malloc(num_neigh*sizeof(int));

  x0 = me / p1; \* the row where processor me is located *\
  x1 = me % p1; \* the column where processor me is located *\

  \***** if x1=0 then me is in the first column *****\
  \* the sequence goes from bottom to top *\
  \* the predecessor of me is the processor below *\
  \* and the successor is the processor above *\
  \* if me is in the last row, its predecessor is the processor to the right *\
  \* if me is in the first row, its successor is the processor to the right *\
  if (x1 == 0) {
    if ( OUT(me + p1) ) neighbors[0] = me + 1;
    else neighbors[0] = me + p1;
    if ( OUT(me - p1) ) neighbors[1] = me + 1;
    else neighbors[1] = me - p1;
  }

  \***** if the row is even, the succession goes from left to right *****\
  \* the predecessor of me is me-1 and the successor me+1 *\
  \* if me is in the last column, the successor is the processor below *\
  else if (x0 % 2 == 0) {
    neighbors[0] = me - 1;
    if ( x1 == (p1-1) ) neighbors[1] = me + p1;
    else neighbors[1] = me + 1;
  }

  \***** if the row is odd, the succession goes from right to left *****\
  \* the predecessor of me is me+1 and the successor me-1 *\
  \* if me is in the last column, the the predecessor is the processor below *\
  else {
    if ( x1 == (p1-1) ) neighbors[0] = me - p1;
    else neighbors[0] = me + 1;
    neighbors[1] = me - 1;
  }
}
```

Figure 4.9: Algorithm for embedding a ring into a 2-D mesh.

```
Ring(me, num_proc, p1, num_neigh, neighbors)
int me;
int num_proc;
int p1; *\ dummy argument used for symmetry with the
          2-D mesh case *\
int num_neigh;
int *neighbors;
{
  num_neigh = 2;
  neighbors = (int *)malloc(num_neigh*sizeof(int));

  if (me == 0) neighbors[0] = num_proc - 1;
  else neighbors[0] = me - 1;

  neighbors[1] = (me + 1) % num_proc;
}
```

Figure 4.10: Algorithm for embedding a ring into a fat tree.

the distance between communicating processors. Thus, we implement a simple algorithm that maps the ring into the processors in numerical order. The algorithm is shown in Figure 4.10.

4.1.7 Embedding linear arrays. To embed a linear array into a hypercube, 2-D mesh or fat tree, we can use a similar procedure to that for embedding rings. The only difference is that the procedure for arrays does not return a predecessor of the first processor or a successor of the last processor.

4.2 Dynamic Load Balancing Strategies

Load balancing is a critical factor in the efficient implementation of parallel applications that evolve dynamically. The four important components of a load balancing procedure are the **control policy**, the **transfer policy**, the **location policy**, and the **information source policy** [88]. The control policy determines what processors are responsible for the distribution of the work. The transfer policy concerns decisions such as when to send a task from one processor to another and which task to send. The location policy determines which processor should receive or send the tasks. Finally, the information source policy decides what kind of information should be used to make task transfer decisions. We devote the rest of this section to discussing each one of these components in more detail.

4.2.1 The control policy. Load balancing strategies can be centralized, distributed, hybrid or hierarchical [29, 88]. In centralized strategies [51, 24], a single processor, called master, distributes the work. In distributed strategies, all the processors actively participate in the distribution of the work [20, 23]. Hybrid strategies contain aspects of both centralized and distributed [88]. In hierarchical strategies [29, 32], processors are organized in different levels. According to their level in a hierarchy, processors have different responsibilities in the distribution of work.

The centralized approach uses a master-slave model. The master processor is responsible for the distribution of the tasks among the processors. Slave processors communicate with the master either to request work or to send some work away. When the queue is centralized, the master processor is also responsible for maintaining the queue. The approach is not scalable. The master processor quickly becomes a bottleneck as the number of processors increases [56, 88].

In the distributed approach, each processor is responsible for the distribution of the load. To avoid unnecessary communication costs, processors run their tasks locally as much as possible. However, the local workload may become too heavy or too light, and so they must transfer work to keep the load balanced. This approach is well-suited to task-parallel problems for they are composed of independent subproblems or tasks that can be moved from one processor to the other without regard to the communication restraints of data-parallel computations.

The hybrid approach combines the characteristics of the centralized and distributed ones. The centralized mediator approach studied in [88] is an example in this category. In this strategy, processors do not deal with the master all the time to request or send some work. Rather, they store and execute their tasks locally as much as possible and only communicate with the master when the system becomes imbalanced. Thus, when a processor becomes overloaded, rather than sending work directly to other processors (as in the distributed case), it sends it to a designated processor called the centralized mediator. Likewise, if a processor becomes lightly loaded or idle, it sends a request to the centralized mediator. This strategy requires less communication through the central processor than does the centralized one. For that reason, the centralized mediation strategy is expected to scale up to a larger number of processors than the centralized one. However, the mediator processor also becomes a bottleneck as the number of processors increases.

Finally, the hierarchical approach [29, 32] is intended to overcome the bottleneck problem that arises in the centralized and hybrid approaches by dividing the processors into groups and applying either centralized or hybrid strategies to each group. For example, a two-level centralized hierarchical approach organizes the processors into groups at the first level and applies a centralized strategy to each group to keep them balanced. The strategy resolves the imbalance between groups at the second level by applying the centralized approach to the masters of those groups.

4.2.2 The transfer policy.

Two strategies have been proposed to distribute tasks to and receive tasks from other processors: sender-initiated and receiver-initiated [26] (also called bidding and drafting strategies [78].) In sender-initiated strategies, processors send out work when they determine they are overloaded. In this case, work is sent to other processors without it being requested by them. In receiver-initiated strategies, processors initiate the load balancing process when they determine that they need more work. In this case, work is sent to these processors upon request.

In a sender-initiated strategy, a heavily-loaded processor sends a number of its tasks to another processor. The receiving processor accepts the tasks. If it becomes heavily loaded as well, it forwards the tasks to another processor. In a receiver-initiated strategy, a lightly-loaded processor sends a request to another processor called the requestee processor. If the requestee processor has extra work to give away, it sends a number of its tasks to the requestor. Otherwise, it forwards the message. If the requestor processor still has some work to do, it proceeds with the execution of its tasks. On the other hand, if the requestor processor is already idle, it may send another request or check for termination.

Eager et al. [26] show that both receiver- and sender-initiated strategies for dynamic load balancing are better than the static approach. They also show that receiver-initiated strategies outperform sender-initiated strategies at high system loads and that sender-initiated strategies are preferable at light to moderate system loads [27].

There are some applications in which tasks are assigned different priorities. Such applications can be more efficiently implemented when the priorities determine the execution order of the tasks, i.e, when higher priorities tasks are executed first. In this type of problem, a load balancing scheme that only balances the load without balancing the high priority tasks over the system, might result in the concentration of these tasks on a few processors. This situation leads to wasteful computations and substantially higher memory requirements that can seriously degrade the performance. A solution to this problem is to make each processor transfer some of its highest-priority tasks not only when it is heavily loaded but also when it is moderately-loaded as well. In this case, processors periodically transfer some tasks to other processor to ensure an even distribution of the best parts of the search space. Obviously, if the frequency of transfers is high, then the communication costs can be very large.

The Token strategy [86] provides another alternative to dealing with prioritized tasks. This hierarchical approach splits the processors into clusters of managees controlled by load manager processors. When a managee creates a task, it sends a token containing the priority of the new task to the load manager. Each managee informs its manager of the status of its load by both, piggybacking load information with each token it sends to the manager and by periodically sending load information. When a manager decides that one of its managees needs work, it asks the owner of the highest priority task to send some work to the lightly loaded processor. The load managers balance tokens and priorities among themselves by exchanging their high priority tokens.

4.2.3 The location policy. There are many ways to determine which processor will be sent the tasks or will be asked for tasks. In the **Gradient** strategy [61], for instance, the requests are sent through adjacent processors in a mesh-connected topology. This is done by assigning a **pressure** to each processor that is a function of the processor utilization and memory availability. A processor also defines a **proximity function** that indicates its distance from the nearest idle processor. This function is computed by each processor based on its pressure and the proximity function of its adjacent nodes. The value of the proximity function at each node determines a **gradient plane**. Idle and overloaded processors can be found by following the track of least and largest proximity functions respectively in the gradient plane. **Contracting Within Neighborhood** [82] is another strategy that transfers work between neighbors. In this case, requests are sent to the topologically adjacent processor with the least load. This differs from the Gradient strategy, where requests are sent in the direction of an inferred global workload minimum.

A different approach is given by the **Random** strategy. In this case, a processor is selected at random either to transfer or to request some work. When a processor receives a task, it accepts it for processing only if its queue length is less than a given threshold. Otherwise, the processor forwards the task to another randomly selected processor. To avoid instability, there is a **transfer limit** for the number of times a task can be forwarded. If this limit is exceeded, the destination processor of the last transfer must execute the received tasks regardless of its state. The strategy ignores communication locality and system state. It is appealing both for its simplicity and for its fast distribution of the work without the overhead of acquiring system information.

The performance of this strategy has been studied theoretically in [26, 96]. The studies in [96] conclude that it can be efficiently applied to search methods such as backtrack search and branch-and-bound. Also, Grunwald et al. show experimentally in [40] that a random policy generally yields good performance when it is used for the initial placement of work.

A variation of the Random strategy is called **Threshold**. In this case, a processor is also selected at random but it is **probed** to determine whether a transfer of some tasks to it would put its queue length above a given threshold. If not, the work is transferred to the selected processor. If so, then another processor is selected at random and probed in the same way. This process repeats until either a processor is found to accept the tasks or the number of probes exceeds a **probe limit**. The Threshold strategy avoids the useless task transfers that sometimes occur in the Random strategy. Eager. et al. [26] show that the performance of this strategy with a small probe limit of 3 or 5 is almost as good as the performance with a large (and consequently more expensive) probe limit. Another key result of [26] is that the Threshold strategy provides substantial performance improvement over the Random one.

In any case, Eager et al. [26] conclude that any load balancing strategy is better than none and that simple strategies are usually as effective as more complex ones. Grunwald et al. [40] also remark that no single strategy is unambiguously best across all the problems.

4.2.4 The information policy. Load balancing strategies can vary from simple to complex in their use of system state information. The advantage of a complex policy is the possibility of making a better decision regarding work placement. The disadvantages are the overhead cost and the risk of using inaccurate or stale information. Thus, there is a trade-off between the amount of system information used to make a transfer decision and the cost of acquiring that information.

Information about the system state can be acquired from two sources: local and non-local. An example of a strategy that uses local information is Random. The motivation for this type of strategy is that the cost of acquiring information from other processors is not too high compared to the benefits of using it. Examples of strategies that use non-local information are Contracting within Neighborhood [82], Gradient [61], and Threshold [26].

Eager et al. [26] demonstrate that the potential of dynamic load balancing can be realized by strategies that use only small amounts of local or non-local information about the system state. A similar conclusion is reached in [40].

4.3 Checking for Termination

Termination detection is a programming issue that arises in the solution of dynamic problems due to their asynchronicity. For these problems, load balancing is continuously checked and transfer of work accordingly made from the heavily loaded processors to the lightly loaded ones. In this context, idle processors waiting for heavily loaded processors to send some work away may wait forever if not informed that the work has been completed. Therefore, because it is impossible for an idle processor to correctly decide whether or not to quit based on its local information, a global check for termination becomes necessary. Observe that this is a different situation than that of the load balancing case. In the load balancing case, one has the alternative of using either local or global information to transfer work. If the decision to use local information leads to an incorrect transfer of work, the processor has the chance to recover from that mistake by transferring work again. In the termination checking case, this is no longer true. If a processor makes a wrong decision and quits, it cannot start again. Therefore, the approach for termination must be based on global information. A global check can be made by a single processor, i.e., centralized, or by all of them, i.e., distributed. Because the communication traffic associated with some centralized approaches can be reasonably low, we are interested in these types.

A first approach [56], based on the prefix algorithm [58], assumes that processors are connected by a virtual tree, but it can use any other virtual topology that is appropriate for a combine operation. The process starts at the leaves. When a leaf processor becomes idle, it requests more work from another processor. While waiting, it sets its DONE flag and passes it on to its parent with a termination message. When an internal processor, i.e., one that is not a leaf, becomes idle, it sends a request for more work to another processor. While waiting, it checks if it has received the termination message from its children. If that is the case, and if it is still idle, it sets its DONE flag and sends a termination message to its parent. The process repeats until it reaches the root. Once the root sets its own DONE flag, it broadcasts an order to quit back down to the rest of the processors. This process takes $2\log_2(p)$ steps to complete, where p is the number of processors. Observe that termination messages need to be given lower priority than the rest of the messages. In other words, if an idle processor waiting for work receives two messages, one carrying work and one carrying a termination message, it just ignores the termination message.

However, there is a problem with this algorithm: the information received by the root may be stale by the time it is received, and, therefore, the order to quit could be wrong. In fact, suppose that a heavily loaded processor A sends some work away to processor B which has just set its DONE flag and passed it on to its parent. Because node B is idle, but has not received the quit order yet, it accepts that work. In this case, if node A finishes first and passes its DONE flag to its parent, the root will receive the wrong information that every node has terminated (node B may still be working.)

To overcome this problem, [94] proposes a new approach. It starts at the root of the virtual tree of processors. When the root becomes idle, it broadcasts a request for information about the **message count**, i.e, the difference between the number of messages sent and the number of messages received by every processor. Each node receives the request, but only passes it down the tree when it becomes idle. When it reaches the leaves, these send back their answers. Internal processors wait for their children to respond. They add their counts to their children's counts and pass them back up. The process repeats until it reaches the root. If the count that arrives back at the root plus its own count is zero, then all nodes have finished. In this case, the root broadcasts a quit message. Otherwise, there is still some work being done. This algorithm needs $2\log_2(p)$ steps to combine the information and $\log_2(p)$ steps to terminate.

We propose a new approach that is more efficient than the one presented in [94] for it takes $2\log_2(p)$ steps to combine the information and terminate. In our approach, each processor keeps a count of the messages it has sent —i.e., those requesting work— minus the messages received —i.e., those carrying work. When a leaf processor becomes idle, it sends a termination message with this count to its parent. When an internal processor becomes idle, it checks if it has received the termination messages from its children. In that case, it adds its children's counts to its own count and passes this value upwards. The process repeats until it reaches the root. The value that the root obtains by adding its own count to its children's count, represents the overall amount of messages sent minus messages received. If this value is zero, all the processors have finished, and, therefore, the root broadcasts a quit message. Otherwise, there is at least one requesting message that has been sent but has not been received before the message count was passed. In that case, the root proceeds until it receives a new count. When the processors that received the uncounted messages finish their work, they pass a new count on to their parents. This algorithm takes only $\log_2(p)$ steps to combine the information and another $\log_2(p)$ steps to complete termination.

We are investigating work from other areas that may be pertinent to a full evaluation of which is the most efficient algorithm to use to detect termination. We discuss them in detail in [18].

4.4 Broadcasting and Gathering

For broadcasting and gathering we discuss the procedure presented in [5, 79] that can be efficiently implemented on meshes and hypercubes as well as on fat trees. This algorithm does not cause any contention problems and has the same logarithmic time complexity on all of these architectures.

In the hypercube and fat tree cases, it is implemented by embedding a minimum spanning tree as discussed in Section 4.1. Figure 4.11 shows the broadcasting procedure for a hypercube of eight processors. The figure illustrates that the order in which bits are toggled, i.e., least significant bit to most significant bit or vice versa, does not affect the performance of the broadcasting on this type of architecture.

Figure 4.12 shows the same broadcast on a linear array of p nodes. In this case, network conflicts appear when the bits are toggled least-significant bit first. When this order is reversed, i.e., when bits are toggled most-significant bit first, network conflicts are resolved as indicated in Figure 4.13. These results can be extended to arrays of two dimensions. In the mesh case, the idea is to partition the two-dimensional array along one dimension, thereby reducing the problem to that for linear arrays. These, in turn, are recursively partitioned, doubling the number of partitions at each step, and creating distinct subarrays which can proceed independently with the broadcast (gather) procedure. In this way, a minimum spanning tree broadcast can be efficiently performed on meshes as well. The only difference between the hypercube and the 2-D mesh approaches is in the way that the minimum spanning tree is derived. While the order in which bits are toggled to derive the tree is irrelevant for hypercubes, it is important for meshes due to contention problems.

4.5 Updating of Pseudo-global Variables

Global variables cannot be used in asynchronous parallel programs running on distributed-memory machines. However, many desired uses of global variables can be captured by pseudo-global ones. To implement a pseudo-global variable, each processor has its own copy of the variable in its local memory. To keep their copies updated, processors exchange their values from time to time. There are two different ways to update pseudo-global

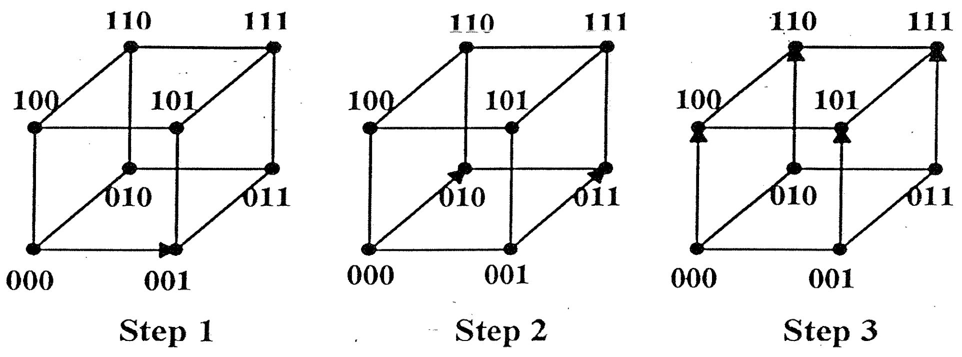


Figure 4.11. Broadcasting (gathering) procedure from (to) processor 0 on a hypercube (least significant to most significant bit.) Source [5].

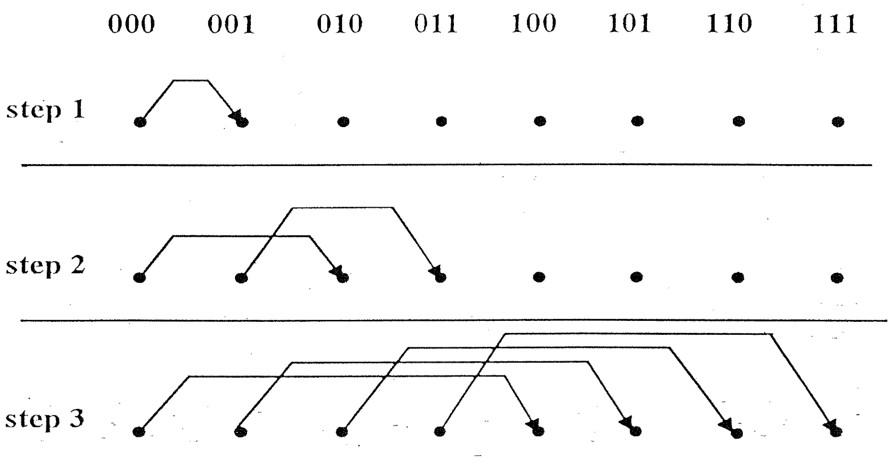


Figure 4.12. Broadcasting (gathering) from (to) processor 0 on a linear array (least significant bit first.) Source [5].

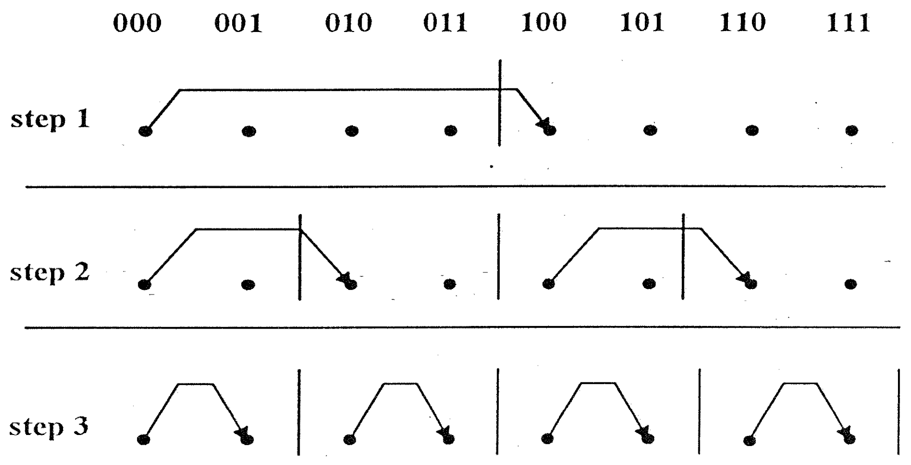


Figure 4.13. Broadcasting (gathering) from (to) processor 0 on a linear array (most significant bit first.) Source [5].

variables: centralized and distributed. In the centralized case, slave processors communicate their values to the master, and the master communicates its updated value to the slave processors. In the distributed case, every processor updates its own copy and passes it to the other processors.

An example of the distributed approach makes each processor broadcast its copy periodically and all the processors update their copies as they receive new values. This approach may lead to a large amount of traffic in the network. An example of centralized approach proposed by Shu and Kale [81] uses a tree-based structure. In this algorithm, each processor keeps its own copy updated. Every processor that is not the root periodically sends its copy to its parent in the tree. The root periodically broadcasts its copy to all processors. Figure 4.14 depicts the pseudo-code for this updating procedure.

We propose a hybrid approach that, like the centralized approach of [81], uses a tree as the virtual topology of the processors. In this case, every processor updates its copy of the pseudo-global variable and communicates it to its neighbors in a virtual tree (rather than broadcasting it to all the other processors.) This approach presents two variations. One, called **regular**, in which a processor always updates its copy with the updating value and communicates it to its neighbors, following the centralized approach of [81] that traverses the tree in both directions. The other, called **monotonic**, in which a processor updates its copy and communicates it to its neighbors only if the updating value is less than the current one. This follows a distributed approach that only traverses the tree in one direction. As in many other issues involved in dynamic computations, the nature of the problem at hand determines the optimal approach to use. Refer to [18] for a more complete

```
\* Procedure to update a pseudo-global variable *\

\* Processors are arranged as a tree structure *\

if (processor is not the root)
  {
    when a new value is received, update own copy;
    periodically, send copy to parent;
  }

else
  {
    when a new value is received, update own copy;
    periodically, broadcast own copy to all processors;
  }
```

Figure 4.14: Pseudo-code for the updating procedure.

evaluation of the strategies for updating pseudo-global variables.

CHAPTER 5

THE PMESC LIBRARY

This chapter presents a new library for implementing task-parallel problems that need a static or a dynamic approach for computation on distributed-memory computers. The library is based on the PMESC paradigm for this allows us to distinguish the phases in the computation and treat them as independent modules. Thus, the paradigm recognizes the modules or building blocks that compose the parallel computations and the library proposes different strategies to efficiently implement those modules. Although the paradigm applies to all kinds of parallel algorithms, the library concentrates on the task-parallel ones. It has been designed to take care of those modules that are application-independent and so can be reused. That way, it provides a tool that frees the programmer from dealing with the complexity of such issues as load balancing, interprocessor communication, and program termination, allowing her or him to concentrate on the application-specific ones.

The chapter begins with a discussion of the existing tools and the motivations for developing a new one, and then proceeds with the presentation of the PMESC library. Section 5.1 classifies the tools that are available for different kinds of parallel applications. Section 5.2 concentrates on those tools that address task-parallel problems and discusses the reasons to create a new one. Section 5.3 introduces the PMESC programming library. Section 5.4 describes the PMESC approach for coping with some of the issues involved in dynamic computations. Section 5.5 describes the PMESC modules. Section 5.6 presents the PMESC routines organized in the Partition, Map, Embed, and Communicate modules. Finally, Section 5.7 gives a first approach to using the library.

5.1 Related Work

Many parallel tools and languages have been proposed in the academic and commercial worlds. They have one goal in common: they want to make the parallel programmer's task easier while maintaining both portability and good performance. The means to achieving this, however, vary widely from one system to the other. Some are languages, some are libraries of routines, and some are libraries of macros. Most of them are based on the data-parallel paradigm and only a few on the task-parallel paradigm. Some address a single stage of the programming process while others address most of them.

In this section we investigate different parallel tools (libraries and languages) that the user has available today to cope with the difficult task of implementing **task-parallel** programs on distributed-memory computers. These tools provide different levels of abstraction, from the low-level communication mechanisms that allow machine independence to the high-level ones that take care of more sophisticated programming issues. In terms of the PMESC phases, they can be classified as tools that provide support for the Communicate phase alone, or for the Communicate and the Embed phases, or for most phases in the computation.

Several systems have been designed to take care of the Communicate phase. They address the first problem that every programmer faces when writing a portable application: different machines have different sets of low-level communication primitives for message-passing. These systems achieve machine independence by providing a uniform, higher-level set of communication routines that remain unchanged across different computers. Some of these tools are intended for a relatively small subset of machines, like PICL [33] and Linda [14]. Others are intended for a wide range of shared-, distributed-memory, and even heterogeneous systems. Examples are PVM [6], p4 [12], Chameleon [39], MPI [64], and PARMACS [41].

Some systems address not only the Communicate but also the Embed phase of the PMESC paradigm. They allow programmers to get rid of the tedious job of mastering not only the low-level communication primitives but also the physical interconnections of the processors. They provide a virtual machine that more resembles the logical interconnections defined by the application than the physical interconnections imposed by the machine architecture. Examples in this regard are MPI, PARMACS [41] and STRAND88 [31].

Finally, many other tools are intended to assist the parallel programmer at a higher level, providing more than just machine independence. While most of these tools are especially designed for quasi-dynamic, i.e., data-parallel, applications, a few provide a similar type of support for dynamic, i.e., task-parallel, ones. Although both types are adaptive, they require different mechanisms for parallelization.

Examples of tools that address data-parallel problems are the LPARX system [55], Dome [7], and PARMACS on the library side as well as Dino2 [77] and Mentat [36] on the language side. Among all the systems proposed so far, only two address task-parallel computations in general: Charm [23] and Express [71]. Charm has been designed to provide full support for task-parallel computations, covering all the application-independent issues involved in their implementation at a high-level. The Charm system is based on the SPMD model rather than on the MPMD one. It handles such issues as queueing of tasks, dynamic load balancing, and information sharing. Express provides low- and high-level abstractions for both data- and task-parallel applications. It is based on both the SPMD and MPMD models, and, therefore, it is intended for a wider variety of problems. It does not support some of the issues specific to task-parallel problems such as queueing and sharing of information. We next describe these tools in more detail.

5.2 Tools for Task-Parallel Applications

In this section, we describe the tools that the user has available today for task-parallel applications. We discuss them in terms of the PMESC phases so that it is easier to compare and contrast them with PMESC.

5.2.1 Express. Express [71], developed by ParaSoft Corporation, is a set of tools designed to assist the user in writing efficient and portable parallel code. It provides support for data- and task-parallel problems. Express includes low- and high-level communication facilities, load balancing, automatic loop parallelization, data distribution, and domain decomposition, and a set of performance and debugging tools. Express is a commercial product that is marketed along with several Express-based applications. It has been designed for MIMD supercomputers with shared and distributed memory, vector supercomputers, networks of workstations and workstations. Programs written with Express are completely portable across all the machines for which it provides support without changing a single line of code.

Express presents a three-layered approach. At the lowest level it provides an architecture specific layer that supports allocation of processors, loading of programs and asynchronous message passing. At a higher level are the utilities designed to address some of the issues involved in data- and task-parallel applications. Examples are the facilities for automatically decomposing problems with regular structure and those for handling dynamic load balancing. These routines can be used ignoring the characteristics of the hardware. At the highest level is a complete I/O system that allows a uniform access to the operating system facilities of the host.

The lowest level of the Express system is based on an asynchronous, point-to-point message passing system. It offers both synchronous and asynchronous functionality to support different types of data- and task-parallel applications. Express also provides global communication functions that include broadcast, gather and synchronization. Along with the obvious attributes of a message such as length, data, source and destination, Express also associates a type with each. This feature is useful for task-parallel applications for it allows processors to make different decisions depending on the message type. In a multitasking environment different message types can be used to specify that certain messages are intended for one task rather than another. Different types can also be used to handle tasks that have different priorities associated with them. Thus, Express provides more features than other communication systems like PVM which support only synchronous mechanisms.

On top of the communication layer comes a higher-level one for which no knowledge of the underlying topology of the processors is required. Users at this level can work on a virtual machine, absolutely ignoring the physical interconnections of the processors. This level was originally designed for data-parallel computations, and, therefore, that is the type of computation for which it provides the strongest support. Express provides utilities for automatic domain decomposition and balancing of data that are based on a virtual grid of processors. However, the newest versions include features for dynamic balancing of tasks as well. Express does not provide support for the other phases involved in task-parallel computation such as handling the task queue structure, information sharing to maintain a pseudo-global variable, and asynchronous detection of termination.

Finally, the highest level layer provides a runtime I/O system that extends the functionality of the standard C and Fortran libraries to fit within a parallel environment. Furthermore, it provides support for graphics, debugging, and performance analysis. In summary, Express offers support for a wide spectrum of problems and applications. In terms of the PMESC phases, Express provides strong support for the Embed and Map phases. It covers a considerable part of the Communicate phase but it does not include information sharing. Neither does it support the Partition phase. Table 5.1 shows the Express features, its **do's** and **don't's** so we can compare it and contrast it later with Charm and PMESC.

5.2.2 Charm. Charm [23], developed by Kale and his students at the University of Illinois, is a parallel programming system that supports an explicitly parallel C-based language for computations with regular and irregular structure. For regular computations, it provides static load balancing and induces data locality. For irregular computations, it includes management of processes, support for prioritization and information sharing, and dynamic load balancing strategies. Charm is not public domain software, but it is available

Table 5.1: Express features

Features	Express	Charm	PMESC
synchronous + asynchronous communication	$\surd$		
virtual topologies + embeddings	$\surd$ (only grids)		
dynamic load balancing	$\surd$		
global communication	$\surd$		
information sharing			
termination detection			
queueing			
prioritized tasks			
I/O	$\surd$		
graphics	$\surd$		
debugging	$\surd$		
performance analysis	$\surd$		

free of charge.

Charm allows machine independent parallel programming over the class of MIMD machines —shared and non-shared memory. Unlike Express, Charm does not provide a low-level communication layer. However, it provides higher-level mechanisms and strategies for task-parallel computations than does Express, supporting almost all the stages involved in their implementation (not just load balancing.) In fact, Charm covers more specific issues such as queueing and load balancing for prioritized tasks. It provides an environment where the user has to explicitly specify the creation of tasks or **chares**, and the communications between them, leaving the management of chares —load balancing, scheduling, etc.— to the system.

Chares have some properties that distinguish them from processes in general. They are medium-grained processes that can dynamically create other chares, send messages to other chares, and share information with other chares using specific information-sharing primitives. The system is free to schedule the chares on any processor it chooses to use. It follows a message-driven approach in which a chare is allocated to a processor only when a message for that chare is received. Each chare has associated with it a data area and some entry functions that can access this data area. The entry functions can be executed by addressing a message to an entry function of a particular chare, which can be uniquely identified with its chare-id. Having processed a message, a chare suspends until another message meant for it is selected. However, when a chare blocks, waiting for a message, another chare may execute on that processor. The system running on each processor picks a message from a queue of messages. Besides the data, a message includes a code entry point and a chare-id. The system identifies the chare-id and then jumps to the entry point. When the entry point code finishes, control returns to the system, which then selects another message and repeats the cycle. The Charm implementation on shared-memory machines has one queue for messages shared by all the processors. On nonshared-memory machines, the queues are local queues, and the load balancing strategy attempts to balance the size of these queues.

Charm provides two mechanisms for I/O; CkScanf and CkPrintf. The difference between CkPrintf and the C function printf is that the former is atomic. This means that data printed by a single CkPrintf is guaranteed to appear on the output without interference from other CkPrintfs being executed by other processors.

In summary, Charm is a more structured environment than Express. It provides wider support on high-level issues involved in task-parallel computations and less support for lower-level issues. In terms of the PMESC paradigm, it provides only little support for the Communicate and Embed phases, and strong support for the Partition and Map phases. Table 5.2 shows a comparison between both systems, Express and Charm.

5.2.3 Why to propose a new tool. In Sections 5.2.1 and 5.2.2, we described the tools the programmer has available today to develop task-parallel computations on parallel computers. Each of these tools has its strengths and weaknesses. Express provides a great deal of support at the lowest and highest levels. It also allows the user to efficiently program on a virtual architecture of processors, ignoring the real one. However, Express lacks of strong support at the mid-level. The Express programmer has to design and take care of the efficient implementation of most issues involved in task-parallel

Table 5.2: Express and Charm features

Features	Express	Charm	PMESC
synchronous + asynchronous communication	$\surd$		
virtual topologies + embeddings	$\surd$ (only grids)	$\surd$ (only spanning trees, only in the high-level context of the chares)	
dynamic load balancing	$\surd$	$\surd$	
global communication	$\surd$	$\surd$ (only in the context of the chares)	
information sharing		$\surd$	
termination detection		$\surd$	
queueing		$\surd$	
prioritized tasks		$\surd$	
I/O	$\surd$	$\surd$	
graphics	$\surd$	$\surd$	
debugging	$\surd$	future plans	
performance analysis	$\surd$	$\surd$	

computation such as handling the queue of tasks, detecting termination of the asynchronous processors, and maintaining global information. Furthermore, the user has to take care of the load balancing of tasks when these have associated priorities. Express keeps the number of tasks in each processor balanced, but it does not provide support for keeping the priorities balanced. Load balancing of prioritized tasks is a very important issue in those problems where efficiency is achieved not just by keeping the processors busy all the time but also by keeping them busy doing productive work.

In contrast, Charm provides no support at the lowest-level. If the user wants to implement a new approach from scratch, she or he must use another communication system. Instead, Charm provides a great deal of support for high-level programming issues, but in a very structured manner. The user then, is restricted to using the chare-based model without much room for change. This constraint is certainly a drawback in this type of problem where the decisions about what technique or approach are more efficient depend on the application and, sometimes, on the target machine. In order to achieve good performance, the user must be given alternatives so she or he can ultimately decide what to use and how to use it.

The PMESC library offers the high-level facilities provided by Charm but at the same time provides the low-level power of machine independence provided by Express. In other words, it combines the strengths of both Express and Charm in a single system that allows portability with good performance and, above all, is easy to use. We want this system to offer the user the framework and high-level abstractions to design and implement task-parallel applications. It should also give the user all the power to create her or his own alternatives from scratch without the burden of the machine-dependent communication primitives or the architecture-dependent processor interconnections. In the next sections, we discuss the library prototype that we propose to deal with these issues.

5.3 A New Library: PMESC

Rather than starting from the hardware and building a communication system, PMESC began with the applications and their requirements. Our goal with PMESC is to fulfill those requirements by offering utilities at all levels of complexity from low-level message passing primitives to automatic load balancing as well as a communication interface that is totally independent of the underlying hardware connectivity. We next discuss the design decisions made to achieve this goal.

5.3.1 The levels of PMESC. PMESC is a two-layered library. At the lower level, it provides support for synchronous and asynchronous message passing. The user can either use these facilities or ignore them totally depending on the degree of involvement she or he wants to achieve with the application. At the higher-level, PMESC provides the high-level abstractions to handling more specific programming issues. These routines form the basis for a flexible model of computation in which the underlying topology of the hardware can be completely ignored. Unlike Express, PMESC does not support a higher level for I/O. We will consider this possibility in a future implementation of the library. For I/O, the user can use either the functions provided by the C or Fortran language or the routines provided by the machine vendors.

Each level of PMESC is distinct and independent of the other, with the higher level being built on top of the lower one. As a result of that design, we can port the library to a wide variety of computers by taking a vertical "top-down" approach in which the low-level may need to change while the high-level, built on top of it, does not. The user code, built on top of these levels, should remain unchanged across different computers. The only exception may be the lines corresponding to input and output of data (only if the routines provided by the machine vendors are used.)

5.3.2 The programming model (for viewing the computers).

There are two completely disjoint programming models. PMESC supports the "Hostless" model. In this model, the same or different pieces of code are executed in parallel on all participating nodes by just invoking the name of the executable code. The parallel program may use most operating system services and runtime libraries as though it were running on the host computer itself. For instance, it can invoke the C functions for I/O from any node. The Intel machines iPSC/860 and Delta as well as the CM5 support this approach.

The alternative model is called "Host-Node". It entails writing a program for the native host computer and another for the parallel computer nodes. The host program takes care of allocating processors and downloading applications, and eventually it may execute some portions of the application. The nodes typically perform the bulk of the numerical computation. The facilities available from the operating system and runtime libraries are only available to the host. Thus, I/O must be handled by the host program and then sent in messages to the nodes. The iPSC/2 and CM2 support this model.

In this thesis, we will limit ourselves to the hostless approach for it resembles more the nature of the asynchronous and independent problems.

```
# include "types.h" \* Define system constants and macros *\

main (argc, argv)
      int argc;
      char * argv[];
{
  \* User Program *\
}
```

Figure 5.1: A typical PMESC program.

However, a future implementation of the library should include the host-node model if future computers demand it. That way, the user can decide which approach to use.

5.3.3 The language used. PMESC is a library of routines. It can be added to any existing high level language supported by the target machine. So far, the PMESC prototype is available in C. Most constants and variables needed by PMESC are defined in the header file **types.h** which should be included in all PMESC programs. A typical PMESC program written in C has the skeletal form shown in Figure 5.1.

Note that the user is not restricted to the C language. It is possible to write a Fortran program that interfaces with a C program.

5.3.4 Some "open" design decisions. Some design decisions have been left open to the user because they depend on the application. Questions like:

- Should the queue of tasks be centralized or distributed?
- What is the most efficient possible initial distribution of work among the processors?
- Should tasks be prioritized?
- Should processors share some information?

- Which load balancing strategy incurs the least overhead?
- When should the load balancer be invoked?

could not be answered without considering the problem and the target machine. Thus, the nature of the application should determine the set of building blocks to be used. Only one system so far has addressed some but not all of these questions: Charm. We next discuss the PMESC approach to deal with these issues. In Chapter 9, we compare and contrast this approach to the one proposed by Charm.

5.4 The PMESC Approach

We begin by explaining the PMESC programming model for viewing dynamic task-parallel problems in general, and then we discuss the open design decisions in particular. Task-parallel problems can be compared to a group of individuals working on a common project. The work is split and assigned to the members. Each member, in turn, subdivides his or her work into smaller pieces. Because they can execute only one piece at a time, they place the other pieces in a pile where they will pick them later. In our case, individuals are represented by processors, pieces of project by tasks, and the pile of subprojects by the queue of tasks, but the fundamental idea is exactly the same.

The first step towards using the PMESC library is to identify the units of work or tasks. Tasks are stored in a queue of tasks from which they are selected for computation. The tasks and the queue play a key role in the implementation and performance of task-parallel problems on distributed-memory computers. Tasks are important for they define the granularity of the parallel problem. PMESC is especially designed for medium- to coarse-grain problems, and it incurs a high overhead on fine-grain applications. The queue is also important for it originates two different approaches, centralized (maintained by a single processor) and distributed (split into local queues maintained by all the processors), that lead to different memory usage, network usage, and programming complexities. With these fundamental concepts in mind we can begin making decisions.

5.4.1 Should the queue be centralized or distributed? One of the first decisions that users must make is whether the queue of tasks is centralized or distributed. PMESC supports both approaches. If the queue is centralized, a master processor is responsible for storing and maintaining the queue of tasks. Figure 5.2 depicts the framework for this approach. The pseudo-code shows that the burden of manipulating the queue is on the master processor who spends its time receiving and sending messages. The other processors take tasks from the queue by sending a request to the master processor. When they receive a task, they execute it, eventually creating more tasks along the way. When a processor creates new tasks it stores them locally until it finishes with the execution of the current task. When execution is finished, the processor sends the newly created tasks to the master along with a request for more. This process repeats until the queue becomes empty in which case the master processor sends a termination message to the other processors. Because tasks are assigned to the processors upon request, no load balancing strategy is necessary with this approach. Also, because there is a single queue, no special mechanisms are required for either detecting termination or using prioritized tasks.

In the distributed approach, each processor is responsible for maintaining, i.e., storing and handling, its own local queue. Figure 5.3 presents the framework for this case. The figure shows that every processor works on the execution of its own tasks until its local queue becomes empty. However, like in the human workgroup case, some individuals execute their portions faster than the others either because they are less efficient at their task or because their task is more complex. To fix this problem, idle individuals are assigned tasks from the busy ones in the hope that everyone will end together. That way, one can reduce overall execution time by dynamically distributing the work so that each processor is kept busy all the time. The load balancing mechanisms are a fundamental ingredient for achieving good performance.

5.4.2 What is the most efficient initial distribution of work among the processors? One characteristic of dynamic problems is that they are unpredictable. It is impossible to assure that any initial distribution of work will evenly assign work to all the processors. Therefore, PMESC lets the user decide on this matter. The user may either try to implement a code that takes care of the initial distribution or do nothing and let the load balancing mechanisms (in the distributed queue case) or the queueing mechanisms (in the centralized queue case) take care of it. This decision depends on the knowledge that the user has of the problem and on how fast she or he wants processors to begin the parallel execution. In general, sending some work to every processor so that all can start cooperating right at the beginning of the execution may be the best alternative though it is not always possible. PMESC provides machine-independent communication routines as well as virtual topologies that help the user to implement this code in a portable way if she or he decides to do so. We provide some illustrative examples in Chapter 6.

```
[Define global variables;]

main ()
{
     Define local variables;

     InitialPartition;
     InitialMap of the work;
     Store initial task(s) in queue;

     if (MASTER) {
       while (queue is non-empty) {
            Handle queue {
                 if (task received)
                   Store it in the queue;
                 else if (task requested) {
                   Get task from queue;
                   Send task to requesting processor;
                 }
            }
       }
       Send termination message;
       Gather results;
     }

     else {
       while (termination message not received) {
            Get task from queue;
            Solve task;
            Store newly created tasks in the queue;
       }
       Send final results to MASTER;
     }
}
```

Figure 5.2: Syntax of the Centralized Approach

```
[Define global variables;]

main ();
{
        Define local variables;

        InitialPartition;
        InitialMap of the work;
        Store initial task(s) in queue;

        while (termination not detected) {
                while (moderately loaded) {
                        Get task from queue;
                        Solve task;
                        Stores newly created tasks in the queue;
                }

                Balance loads;
                Check for termination;
        }

        Gather results;
}
```

Figure 5.3: Syntax for the Distributed Approach

5.4.3 Should tasks be prioritized? This is another decision that the user must make for it depends on the particular problem. Some problems can be split into subproblems that can be executed in any order without affecting either the correctness of the results or the overall performance. There are however, other cases where the assignment of priorities to the tasks may have a tremendous impact in the performance (and even the feasibility) of the problem. This consideration may apply to both sequential and parallel implementations. An example is given by the Traveling Salesman Problem (TSP). (Please refer to Chapter 6 for a description of this problem.) PMESC provides support for handling prioritized and non-prioritized tasks.

5.4.4 How processors share some information? In task-parallel applications, processors never synchronize. Thus, if they need to share some information, they must define a pseudo-global variable. The need for a pseudo-global variable depends on the problem. Refer to Chapter 6 for some illustrative examples.

Although maintaining a pseudo-global variable involves a high communication overhead, keeping it updated reduces the computation costs. The frequency of the updatings of the pseudo-global variable then becomes a subject of trade-off. If the frequency is too low, many wasted computations may result. If the frequency is too high, the updating procedure introduces a large overhead. The PMESC approach to coping with this problem is to provide an updating routine and let the user decide when to invoke it. Chapter 6 shows different uses of this routine. It also shows an example of how to determine the updating frequency.

5.4.5 Which load balancing strategy is the most efficient?

Depending on how the control is distributed among the processors, load balancing strategies can be centralized, hierarchical, distributed or hybrid. (Please refer to Chapter 4 for a description of these strategies.) In general, centralized and even hybrid strategies do not perform well for large number of processors. For that reason, PMESC supports distributed strategies. The PMESC strategies for load balancing are the random approach that distributes tasks globally, a ring-based approach that distributes tasks among the neighbors in a ring, and a strategy for prioritized tasks based on the random strategy. Each strategy comes in two different versions: sender- and receiver-initiated.

The PMESC approach for load balancing is to let the user decide for the particular problem. The user does not need to know which strategy to use in advance. She or he should try different ones and then, based on the results, the user can decide. Theoretical studies show that, the random approach (or any of its variations) performs reasonably well on most applications [27, 40]. Also, in general, the sender-initiated approach outperforms the receiver-initiated one on moderately- to lightly-loaded systems while the opposite is true when the system is heavily-loaded [26].

5.4.6 When should the load balancer be invoked? First, the user must decide how to measure the processor's load. One way to do it is by counting the number of tasks in the queue. When this count is between some lower and upper bounds, the processor is considered moderately-loaded. Otherwise, when the count is less than the lower bound, the processor is lightly-loaded and when it is greater than or equal to the upper bound, the processor is heavily-loaded. In any case, the processor calls the load balancing procedure to initiate some transfer of work or to check if some work has been transferred to it or both (depending on whether the load balancing strategy is sender- or receiver-initiated.) Therefore, the frequency of the load balancing calls can be controlled by the lower and upper bounds. There is always a compromise between the frequency of these calls and the amount of overhead introduced by the load balancing procedure. Too frequent calls may incur high overhead, while less frequent calls may reduce the overhead but also increase load imbalance. By changing the parameters representing the lower and upper bounds, the user must be able to find a reasonable solution.

5.5 An Introduction to the PMESC Library

In this section we summarize the programming issues that PMESC is intended to address. We also organize them as different modules that match the programming phases of the PMESC paradigm.

5.5.1 The PMESC features. The PMESC library addresses the following programming issues:

(1) To provide support for handling the task queue structure.

(2) To provide support for dynamic load balancing.

(3) To provide support for efficient termination checking.

(4) To provide support for efficient updating of pseudo-global variables.

(5) To provide support for global operations such as broadcast and gather.

(6) To provide support for point-to-point communication in a way that is machine-independent.

(7) To provide support for programming on virtual machines.

Table 5.3 shows the main features of Express, Charm, and PMESC. As discussed in Section 5.2 PMESC combines the most important aspects of Express and Charm into a single tool that is easier to use and gives the user more control over the application. In the next subsection, we discuss the PMESC modules that actually support these features.

5.5.2 The PMESC modules. The PMESC library is composed of a set of routines or building blocks that allow one to build a modular, easily changeable interface between the problem and the machine. These routines are classified according to the programming phase in the PMESC paradigm for which they provide support. Thus, the routines that handle the task queue structure make up the Partition module, the routines that take care of balancing the load and checking for termination comprise the Map module, the routines that match virtual into real architectures compose the Embed module, and the routines that take care of handling low- and high-level interprocessor communication make up the Communicate module.

To build the library, it is necessary to implement these modules. To do that, we need to define precisely what programming issues are involved in each module and what strategies to use to efficiently attack those issues. We will consider the Partition, Map, Embed, and Communicate modules, discuss their functionalities, and the methodologies to solve them.

5.5.3 The Partition module. The routines of the Partition module perform one of the following procedures:

- create the task queue structure,
- dynamically allocate and deallocate memory for the task queue structure,
- add tasks to the queue,
- select tasks from the queue,

Table 5.3: Express, Charm, and PMESC features

Features	Express	Charm	PMESC
synchronous + asynchronous communication	√		√
virtual topologies + embeddings	√ (only grids)	√ (only spanning trees, only in the high-level context of the chares)	√ (trees, rings, arrays)
dynamic load balancing	√	√	√
global communication	√	√ (only in the context of the chares)	√
information sharing		√	√
termination detection		√	√
queueing		√	√
prioritized tasks		√	√
I/O	√	√	
graphics	√	√	future plans
debugging	√	future plans	future plans
performance analysis	√	√	future plans

- partition the queue.

Depending on the application, queues can be either non-prioritized or prioritized, centralized or distributed. The Partition module supports all these approaches. It is implemented by defining a linked list that represents the queue structure and different functions that access and modify the linked list.

5.5.4 The Map module.

The routines of the Map module take care of one of the following programming issues:

- distribution of tasks among the processors for load balancing,
- termination checking.

Chapter 4 discusses the issues involved in the selection of load balancing and termination checking strategies. In general, to avoid the bottleneck problem caused by centralized approaches, we have chosen distributed ones. The only exception to that rule has been made in the selection of the termination checking strategy. (Please refer to Chapter 4 for a discussion of this problem.) With respect to the load balancing procedures, we have made the following decisions. Regarding the control issue, the strategies proposed by PMESC use a distributed approach. The centralized strategy that we implemented produced such poor performance that we decided not to include it. However, hybrid and even hierarchical approaches should be included in the future. Regarding the task transfer issue, all the strategies are implemented in both versions, sender-initiated and receiver-initiated. Regarding the location issue, PMESC provides a random approach that distributes tasks globally and a ring-based approach that distributes them among the neighbors in a ring. It also provides a strategy for prioritized tasks that is based on the random strategy. Our decision has been based on the results of [26, 40, 96] discussed in

Chapter 4. Finally, regarding the information source issue, we have used only local information to avoid another source of overhead.

5.5.5 The Embed module. Another fundamental task in the implementation of a parallel algorithm is the allocation of virtually connected processors to a given architecture. Considering embedding as an independent procedure allows one to program on a virtual machine, thereby hiding architectural details from the application. Thus, if the programmer is concerned with efficiency and the high communication costs that the algorithm may incur, he or she should try a virtual topology of processors that can be efficiently mapped onto the actual machine. PMESC provides such efficient embedding routines that exploit the hardware characteristics while keeping them hidden from the user.

Many efficient algorithms [5, 10, 43, 79, 58] are available for a wide variety of interconnection structures and architectures. So far, we have included the virtual topologies that we needed for the implementation of the PMESC routines, i.e., rings for the ring-based load balancing routine, trees for the global combine routines, and for the updating and termination checking routines, and arrays. The algorithms are described in Chapter 4.

5.5.6 The Communicate module. The routines of the Communicate module take care of the following programming issues:

- synchronous and asynchronous point-to-point communications,
- asynchronous global combine operations,
- asynchronous updating of pseudo-global variables.

We classify them into two different levels: the low level that takes care of the point-to-point communication and the high level that takes care of the rest.

The high level of the Communicate module includes routines for global interchange of data. They are Broadcast, Gather, and Update (of pseudo-global variables.) The mechanisms used for broadcasting and updating are described in Chapter 4. Gathering is similar to broadcasting. However, messages flow in the opposite direction. Instead of going from the root downwards to the leaves (as in broadcasting), messages go upwards from the leaves to the root.

5.6 The PMESC Routines

The purpose of this section is to describe the main routines that compose the PMESC library modules. Thus, each subsection contains a brief discussion of the main issues approached by each module and presents the routines that are intended to address those issues. The arguments of the routines are marked as IN, OUT or INOUT. The meanings of these are:

- the routine uses but does not update an INput argument,
- the routine may update an OUTput argument,
- the routine uses and updates an INOUT argument.

5.6.1 The Partition module. The routines of the Partition module are invoked to handle the task queue structure. Users define a variable Task (equivalent to a C structure or a Pascal record) and decide which type of queue to use (**TYPE1**: centralized or distributed, **TYPE2**: FIFO or prioritized), and the routines of the Partition module take care of the rest. Task is a data structure that contains all the information that a processor needs in order to execute that particular task. Remember that tasks can be executed by any processor, not necessarily by the processor that generates the task. The partition routines are:

- int ENQUEUE (TYPE1, TYPE2, * Task)

IN	TYPE1	queue type can be centralized or distributed
IN	TYPE2	queue type can be FIFO or prioritized
OUT	Task	a data structure that contains the information to execute a task

ENQUEUE places a Task in the queue according to its priority or in FIFO order if the queue has no priorities. It returns the current length of the queue.

- **int DEQUEUE (TYPE1, TYPE2, * Task)**

IN	TYPE1	queue type can be centralized or distributed
IN	TYPE2	queue type can be FIFO or prioritized
OUT	Task	a data structure that contains the information to execute a task

DEQUEUE selects a Task from the queue (priority or FIFO) and returns the current length of the queue.

- **int LENGTH (TYPE1, TYPE2)**

IN	TYPE1	queue type can be centralized or distributed
IN	TYPE2	queue type can be FIFO or prioritized

LENGTH returns the length of the queue.

- **PARTITIONQ (TYPE1, TYPE2, TYPE3, int *num-tasks, int *p-node)**

IN	TYPE1	queue type can be centralized or distributed
IN	TYPE2	queue type can be FIFO or prioritized
IN	TYPE3	SPLIT indicates that the queue is split to send tasks. RECV indicates that the queue

		receives tasks
INOUT	num-tasks	number of tasks to be sent
IN	p-node	node to which tasks will be sent

PARTITIONQ **SPLIT**s the queue and sends **num-tasks** Tasks to **p-node** or **RECEIV**es **num-tasks** Tasks from **p-node**.

5.6.2 The Map module.

The routines of the Map module take care of the dynamic load balancing and termination checking procedures. The user decides which strategy to use by selecting the corresponding routine. He or she can also decide whether the selected strategy is sender- or receiver-initiated by just changing one of the arguments of the routine.

The Map routines are:

- **MAP_Ra (TYPE1, TYPE2, int node, int work_nodes)**

IN	TYPE1	load balancing strategy can be **SENDER** or **RECEIVER**
IN	TYPE2	indicates whether processor is **BUSY** or **IDLE**
IN	node	the calling processor id
IN	work-nodes	number of processors used

  MAP_Ra transfers work between the calling processor **node** and a processor **R**andomly selected in a sender or receiver-initiated fashion. **TYPE1** represents those two options: **SENDER** and **RECEIVER**. **TYPE2** indicates whether the calling processor is **BUSY** or **IDLE**. **work_nodes** is the number of processors used for the parallel application.

- MAP_Ri (TYPE1, TYPE2, int node, int work_nodes)

IN	TYPE1	load balancing strategy can be **SENDER** or **RECEIVER**
IN	TYPE2	indicates whether processor is **BUSY** or **IDLE**
IN	node	the calling processor id
IN	work-nodes	number of processors used

MAP_Ri transfers work between the calling processor **node** and one of its neighbors in a virtual Ring in a sender or receiver-initiated fashion. **TYPE1** is to represent those two options: **SENDER** and **RECEIVER**. **TYPE2** indicates whether the calling processor is **BUSY** or **IDLE**. **work_nodes** is the number of processors used for the parallel application.

- MAP_Pr (TYPE1, TYPE2, int node, int work_nodes)

IN	TYPE1	load balancing strategy can be **SENDER** or **RECEIVER**
IN	TYPE2	**BUSY** and **IDLE** indicate balancing of the load, **PRIOR** indicate balancing of priorities
IN	node	the calling processor id
IN	work-nodes	number of processors used

MAP_Pr transfers work between the calling processor **node** and a processor Randomly selected in a sender or receiver-initiated fashion. **TYPE1** represents the two options: **SENDER** and **RECEIVER**. This routine implements a strategy for Prioritized tasks that keeps not only the load but also the priorities balanced. **TYPE2** indicates whether the calling processor is **BUSY** or **IDLE** or ready to balance priorities **PRIOR**. **work_nodes** is the number of processors used for the parallel application.

- **TERM-CHECK (flag):** checks for termination according to the procedure described in Chapter 4. Users do not generally need to call this routine; the MAP routine does when necessary.

5.6.3 The Embed module. The routines of the Embed module take care of embedding the set of processors interconnected by different virtual topologies into the actual machine architecture. Thus, a call to Array, Ring, S-Tree, and Quat-Tree by a given processor will retrieve the id of its closest neighbors in a unidimensional array, bidirectional ring, binary tree, and quaternary-tree respectively.

Users can also choose indirectly the virtual topology (without specifically invoking any of the embedding routines.) They do so when selecting some strategies that use a virtual topology of processors. For instance, for the load balancing procedure, users can choose a ring or a fully-connected virtual topology of processors by selecting MAP_Ri or MAP_Ra respectively.

The embed routines are:

- **Array** (int *node, int *dim, int *neighbor)

IN	node	the calling node
IN	dim	number of processors
OUT	neighbor	the id of the virtual neighbor

Array returns the virtual **neighbor** of processor **node** in a unidimensional array of dim processors.

- Ring (int *node, int *dim, int *neighbors)

IN	node	the calling node
IN	dim	number of processors
OUT	neighbors	the ids of the virtual neighbors

Ring returns the virtual **neighbors** of processor **node** in a bidirectional ring of **dim** processors.

- **S-Tree (int *node, int *dim, int *neighbors)**

IN	node	the calling node
IN	dim	number of processors
OUT	neighbors	the ids of the virtual neighbors

S-Tree returns the virtual parent and children, i.e., **neighbors**, of processor **node** in a tree of **dim** processors.

- **Quat-Tree (int *node, int *dim, int *neighbors)**

IN	node	the calling node
IN	dim	number of processors
OUT	neighbors	the ids of the virtual neighbors

Quat-Tree returns the virtual **neighbors** of processor **node** in a quaternary tree of **dim** processors.

- **Tree (int *node, int *dim, int *neighbors)**

IN	node	the calling node
IN	dim	number of processors
OUT	neighbors	the ids of the virtual neighbors

Tree returns the virtual neighbors (**neighbors**) of processor **node** in a tree of **dim** processors. Tree is equivalent to S-Tree on the hypercubes and to Qua-Tree on the CM5.

5.6.4 The Communicate module. The Communicate module takes care of all mechanisms involving interprocessor communication either point-to-point or global. The Communicate module has two different levels. One that takes care of the basic point-to-point communications such as send and receive and the other that takes care of global combine operations such as broadcast and gather.

The high-level communication routines are:

- **BROADCAST (int *source, char *msg)**

IN	source	the processor that initiates the broadcasts
OUT	msg	the message to be broadcast

  It broadcasts a message **msg** from processor **source**.

- **GATHER (int *source, FLAG, int *msg)**

IN	source	the processor that gathers the messages
IN	FLAG	options for a global sum or for concatenation of all the messages
INOUT	msg	as input, the values to be combined; as output the result of the operation available on the source processor

  Depending on **FLAG**, it either performs a global sum or gathers the copies of msg in all the processors as the components of a vector msg in the processor **source**. The values of **FLAG** are SUM and VECTOR, respectively.

- UPDATE (my_node, work_nodes, global, newvalue, FLAG)

IN	my_node	processor id
IN	work_nodes	number of processors

INOUT	global	the pseudo-global variable to be updated
IN	newvalue	the new value
IN	FLAG	option to perform the update

It updates the pseudo-global variable **global** located in the private memory of processor **my_node** with the value **newvalue**. If FLAG is set to REGU, it updates **global** with the content of **newvalue**. If FLAG is set to MONO, it only updates **global** when its content is greater than the content of **newvalue**. If FLAG is UP or DOWN, it follows a centralized approach in which the processor reads the message when FLAG is UP but only updates the pseudo-global variable when FLAG is DOWN.

The low-level communication routines take care of the low-level, machine-dependent mechanisms for message passing. The implementation of these routines may be different on different machines, but this is transparent to the users. Thus, although the names of the library routines for message passing remain unchanged throughout all the machines, the library uses different communication primitives for implementing them (i.e., the CMMD low-level communication primitives on the CM5 and NX on the Intel machines.)

Observe that, like Express, PMESC associates a type with each message. Because PMESC uses types internally to indicate different kinds of activity, the following message type restrictions apply:

Type range	Purpose
0–30,000	For normal use
30,000 and over	For internal use

The low-level communication routines are:

- **int THEREIS-msg (int *type, int *node)**

OUT	type	the message type
OUT	node	the sending node

It returns 1 if there is a pending message as well as the **type** of the message and the sending processor **node**. Returns 0 otherwise.

- **CANCEL-msg (int *type)**

IN	type	the message type

It cancels an asynchronous receive operation of type **type**. When **CANCEL-msg** returns the user knows the following: a) the asynchronous receive operation is no longer active, b) the message buffer may be reused, and c) the message identification (which on most machines is a limited resource) is released.

- **ASEND (int *type, char *msg, int *size, int *node)**

IN	type	the message type
IN	msg	the message
IN	size	the message length
IN	node	the node to which the message is sent

It asynchronously sends a message **msg** of type **type** and size **size** to processor **node**.

- **SSEND (int *type, char *msg, int *size, int *node)**

IN	type	the message type
IN	msg	the message
IN	size	the message length

IN	node	the node to which the message is sent

It synchronously sends a message **msg** of type **type** and size **size** to processor **node**.

- **ARECV (int *type, char *msg, int *size)**

IN	type	the message type
OUT	msg	the message
INOUT	size	the message length

It asynchronously receives a message **msg** of type **type** and size **size**.

- **SRECV (int *type, char *msg, int *size)**

IN	type	the message type
OUT	msg	the message
INOUT	size	the message length

It synchronously receives a message **msg** of type **type** and size **size**.

5.7 A First Approach to Using the PMESC Library

The basic idea behind PMESC is not only to facilitate the programmer's job but also to allow her or him to have complete control over the application. To that end, PMESC provides the building blocks and the frameworks and the user puts them together. The templates presented in section 5.4 provide good starting points for the design and implementation of task-parallel applications in terms of the PMESC library. Users have to take care of the initial partition of the work into pieces, the initial distribution of those pieces among the set of processors, the algorithm itself (i.e., the Solve phase), and the main program that puts all the building blocks together. Having those functions implemented, the rest of the job is performed by the library. Thus, the partition routines handle the task queue —e.g., DEQUEUE selects tasks from the queue, ENQUEUE stores them in the queue. The map routines distribute tasks to keep the load balanced and check for termination. The communication routines implement pseudo-global variables as well as broadcast and gather procedures. The embed routines embed virtual topologies into real ones. We now discuss in more detail the issues involved in the implementation of a problem with PMESC.

To begin, the user has to partition the initial work into pieces and identify the units of work or tasks. The definition of these units is a very important part in the design process that has been purposely left to the users so they can adjust the granularity of the problem. PMESC is designed to coarse- to medium-grain problems. Short-lived tasks increase the overhead and harm the performance. Thus, the initial partition process requires two steps. One is to define a C structure called Task that contains all the data that a processor needs in order to execute a task. The definition of the type Task is usually included in the file "user-types.h". The other step is to write the code (i.e., the InitialPartition function in Figure 5.3) that effectively partitions the initial work into pieces. Notice the distinction between pieces and tasks. Pieces are the initial chunks of work into which the original problem is subdivided. Tasks are the units of work that are queued, executed by the Solve function, and exchanged by the processors to keep the load balanced. The idea is to divide the original problem at a coarse level first, creating as many pieces as processors, so that each processor receives some work to start execution. Once processors get started they split the subproblems at a finer level thereby creating the tasks. In some applications pieces and tasks may be the same.

This depends on the problem, and, therefore, it is left to the programmer's decision.

After separating the pieces, users have to implement the procedure that distributes them among the processors (i.e., the InitialMap function in Figure 5.3.) In general, this is a simple code in which a processor sends one or more pieces of work to other processors. The PMESC library provides routines for send and receive that allow users to write a portable code. Once pieces are distributed, processors are ready to create the queue of tasks and begin execution.

The user has to make some decisions. One decision is what kind of approach to use for the queue of tasks: centralized or distributed. In our experiments, we confine ourselves to the distributed approach for it scales better. Another decision is how to measure the workload (i.e., how to consider that a processor is heavily-, moderately- and lightly-loaded.) Often, a good measurement of the workload is the length of the queue of tasks. If the queue length is between some given lower and upper bounds, the processor is considered moderately-loaded. Otherwise, it is lightly-loaded when the queue length is less than the lower bound and heavily-loaded when the queue length is greater than the upper bound. The lower and upper bounds should be input parameters that the user can easily change in order to find the ones that are appropriate for the particular application.

Another procedure to be taken care of by the user corresponds to the application itself (i.e., the Solve function in Figure 5.3.) The function Solve is, by definition, a sequential procedure. Therefore, the user can implement it and debug it on a workstation and then transfer it to the target platform. Also, the user may reuse existing code. In any case, the code has to be prepared or modified in such a way that it executes a task and terminates. If it creates new tasks, these are stored in a queue by using the PMESC routine ENQUEUE.

Finally, the user must design the main code that puts all the pieces together. If the distributed queue approach is chosen, the user have to decide what kind of strategy use for load balancing (i.e., random, ring or prioritized, sender or receiver initiated.) Figure 5.4 illustrates the pseudo-code for the main program using a distributed queue approach. The core of this main procedure consists of a loop in which every processor selects a task from the queue (by using DEQUEUE) and executes it (by using Solve). Solve takes an instance of a variable of type Task as argument and eventually generates more instances of Task that are placed in the queue (by using ENQUEUE.) This process repeats as long as the queue length is between the lower and upper bounds. Otherwise, the main procedure calls the load balancing routine every time the queue length either exceeds the upper bound (by using MAP_xx (BUSY, other arguments)) or falls behind the lower bound (by using MAP_xx (IDLE, other arguments).) When a processor becomes idle, the load balancer (not the main code) calls the procedure that checks for termination (i.e., TERM-CHECK.) The load balancing routine returns -1 if termination has been detected and 0 otherwise. Once computation is completed, a designated processor may gather the final results (by using GATHER.) GATHER can either combine the results in a vector or add them up.

An optional feature is the use of pseudo-global variables to share some information among the processors. The user defines a pseudo-global variable that provides similar functionality to a global variable without incurring a great deal of overhead. To implement a pseudo-global variable each processor has its own copy and updates it from time to time. To propagate the updated value to the other processors, the user calls UPDATE. UPDATE asynchronously communicates the pseudo-global value and so processors may not have the latest update at any given moment. Because there is a communication cost associated with the updates, the frequency of the calls to UPDATE should depend on the relevance of the pseudo-global variable to the overall computation and its performance.

We described in this section the high-level issues involved in the design and implementation of a distributed task-parallel application using PMESC. We will discuss them again in more detail in Chapter 6 in the context of different examples.

```
#include "types.h"
#include "user-types.h"

[Define global variables;]
[Define pseudo-global variable P-G;]

main ();
{
      Define local variables;
      int signal = 0;
      Task T;

      (User) InitPart;
      [(User) InitMap;]
      Read lower and upper bounds;

      for (;;) {
          queue_length = DEQUEUE(T);

          \* main loop *\
          while (lower_bound <= queue_length < upper_bound) {
                (User) Solve(T); \* executes T and stores new tasks in queue *\
                [UPDATE(P-G);]
                queue_length = DEQUEUE(T);
          }

          \* branch for heavily-loaded processors *\
          if (queue_length >= upper_bound)
          {
                MAP_xx(BUSY, other arguments);
                Solve(T);
                [UPDATE(P-G);]
          }

          \* branch for lightly-loaded or idle processors *\
          else if (queue_length < lower_bound)
          {
                signal = MAP_xx(IDLE, other arguments); \* returns -1
                                                        if termination detected *\
                Solve(T);
                [UPDATE(P-G);]
                if (signal)
                   break;
          }
      } \* end of infinity loop *\
      GATHER;
}
```

Figure 5.4: Syntax of a Distributed Approach using PMESC

CHAPTER 6

CHOOSING EXAMPLES

In Chapter 5 we introduced the PMESC environment. We discussed the PMESC features, presented its routines, and gave a first approach to using it. The purpose of this Chapter is to discuss in more detail the different uses of PMESC. In order to do that, we have selected a set of example problems that illustrate different situations and environment features. We use these examples again in Chapter 7 to assess the ease of use of PMESC as well as its portability and performance.

The examples are the *N* queens problem, parallel bisection, parallel local adaptive quadrature, and the traveling salesman problem. We have chosen them for a number of reasons. First, they represent different areas of research. In fact, these problems may arise in a wide variety of fields, including chemistry, engineering, physics, robotics, artificial intelligence, and many more. Second, they illustrate different levels of complexity. The wide spectrum goes from the simplicity of the "toy" N queens example, to the "real life" situations represented by bisection and quadrature, to the difficulty of the NP-complete traveling salesman problem. Finally, the examples provide enough ground to illustrate the most significant library uses. To this end, we organize the problems in different categories and for each category we present a representative example. Our goal is to use the examples as models as well as starting points to unexperienced users.

Should the user have a more complex, very large scale problem, she or he must partition it into smaller subproblems of the type we discuss in this chapter. Large problems, then, can present several levels of subdivision. In terms of PMESC, this means that the Solve phase in one level can be decomposed into new P, M, E, S, and C phases in the sublevel, thus allowing a finer subdivision of the subproblems or tasks. These very complex problems do not illustrate features of PMESC that are different from the ones we illustrate in this chapter and, therefore, for the sake of clarity, we rather concentrate on those that only require a single level of subdivision. However, our plans include the implementation of large scale problems with PMESC so that we may analyze the performance of the library under higher levels of complexity.

The chapter is organized as follows. Section 6.1 describes different categories of dynamic problems that represent different uses of PMESC. Section 6.2 reviews the steps involved in the implementation of those problems in general. Finally, Sections 6.3 to 6.6 discuss in detail the PMESC implementation of each of the examples.

6.1 Categories of Problems

In order to easily identify the model that matches our particular problem as well as to recognize the PMESC features necessary to implement it, we have identified three categories of dynamic task-parallel problems. These categories are distinguished according to the information shared, the priority of the tasks, and the length of the tasks. Next, we describe each one of these categories.

6.1.1 According to the information shared. It is possible to distinguish two extreme situations in the category of dynamic problems. On one hand, there are fully asynchronous problems on which processors can work wholly independently of each other. No information needs to be shared during the computation, except for that necessary at the beginning or at the end. On the other hand, there are problems that require information sharing during the course of the computation. In distributed-memory computers, this is accomplished through the use of pseudo-global variables. A pseudo-global variable may be necessary, for instance, to maintain the iteration number in an iterative procedure or the bound in a branch-and-bound procedure. Obviously, this approach creates an extra source of overhead and PMESC provides support for its efficient implementation. The example problems show both situations, i.e., fully asynchronous and asynchronous with information sharing, as illustrated in Table 6.1.

In between the two extremes —fully asynchronous and asynchronous with information sharing— there are other problems that can be split into sets of tasks. The tasks in each set are asynchronous with information sharing, but the sets are fully asynchronous among themselves. These problems may correspond to some implementations that separate the processors into groups that perform different activities. Processors in each group may need to share some data during the computation, while processors in different groups only exchange data at the end of the computation. PMESC does not provide support for this kind of problem, and, therefore, we will not discuss them further.

6.1.2 According to the priority of the tasks.

Task-parallel problems can also be classified according to the priority associated with the tasks. Some problems can be separated into tasks that can be solved in any order without affecting either the result or the efficiency of the computation.

Table 6.1: Categories of dynamic problems and representative examples

	Info. Sharing		Priority		Length of Tasks	
	With	Without	With	Without	Fine	Medium-Coarse
Queens		✓		✓	✓	✓
Bisection		✓		✓	✓	✓
Quadrature	✓	✓	✓	✓		✓
TSP	✓		✓			✓

Some other problems need to be separated into tasks that must be executed in a certain order either to guarantee correct results or to achieve good performance. In the latter case, tasks are assigned priorities and stored in a priority queue. The task with highest priority is always selected first from the queue. PMESC provides support for both types of problems, i.e., for tasks with and without priorities. The example problems illustrate both situations as indicated in Table 6.1.

6.1.3 According to the length of the tasks. According to the length of the tasks into which they are decomposed, parallel problems can be either coarse- (tasks involve a great amount of computation time), medium- (tasks involve computation time larger than the communication time between processors), or fine-grained (tasks involve little computation taking time comparable to or less than the communication time.) PMESC is intended for medium- to coarse-grained problems and introduces a great deal of overhead on fine-grained problems. The N queens and bisection problems illustrate two different ways to turn a fine-grained problem into a medium-grained one. Table 6.1 shows that the examples cover the different categories.

In the next section, we review the steps involved in the implementation of dynamic applications in general by using the PMESC environment.

6.2 Using the PMESC Environment

The PMESC frameworks presented in Chapter 5 provide templates or skeletons to design and implement task-parallel applications in terms of the PMESC environment. The user is responsible for implementing some modules and selecting the library routines. The user has to take care of the partitioning of the work into tasks, i.e., identification of tasks and definition of the structure Task, the initial distribution of the work among the set of processors, i.e., InitPart-and-Map, and the implementation of the algorithm itself, i.e., Solve. In the main code, the user puts all the pieces together by following the framework guidelines and her or his own design decisions.

We now highlight the steps involved in the implementation of a dynamic task-parallel application using PMESC. For a more complete description of these issues refer to Chapter 5.

(1) Partition of the initial problem into subproblems to be assigned to the processors. This is a coarse subdivision performed only at the beginning of the execution.

(2) Identification of tasks. Tasks are units of work that processors execute, create, and eventually transfer to keep the load balanced. They should be medium- to coarse-grained.

(3) Definition of Task. Task is a type definition represented by a C structure. An instance of the variable of type Task represents a task and contains all the information necessary to execute that task.

(4) Decisions about the queue. The queue of tasks can be centralized or distributed. In general, we use a distributed queue as it is suitable for large number of processors. Also, depending on the application,

the queue can be LIFO or prioritized. All the definitions of the input parameters necessary for the PMESC routines (including the definition of the variable type Task) should be put together into a file called user-types.h. This file should be included in all the modules written by the user.

(5) Implementation of InitialPartition and InitialMap (sometimes combined into a single function called InitPart-and-Map.) These functions perform the initial partition and distribution of the work. They may either create an instance of Task and place it in the queue of a particular processor or create more instances and assign them to the processors so that they can all begin execution.

(6) Implementation of Solve. This function takes a task and executes it, eventually creating new tasks along the way.

(7) Decisions about the load balancing (only if the distributed queue is used.) Load balancing mechanisms can be centralized, hierarchical, distributed, or hybrid. So far, PMESC only supports distributed ones. The strategies are based on a random selection of processors or on a selection in a ring of processors. The routines that implement these approaches are MAP_Ra, MAP_Ri, and MAP_Pr (for prioritized tasks.) All the strategies can be sender- or receiver-initiated.

(8) Implementation of the main code. The main code should follow the lines of the frameworks depicted in Figures 5.2 and 5.4 of Chapter 5 which correspond to the centralized and distributed queues respectively. In the first case, processors select and execute tasks following a loop or cycle that only breaks when the queue becomes empty. In

the second case, processors execute the main loop until it becomes necessary to balance the loads or, in the prioritized case, to balance the loads and the priorities.

In the next sections, we discuss each one of these steps in more detail in the context of the examples.

6.3 The N Queens Problem

6.3.1 Description of the problem. An example of a dynamic, fully-asynchronous problem is given by the parallel implementation of the N queens problem. We first describe the sequential problem and then discuss the parallel approach. The N queens problem consists of placing N queens on an $N \times N$ chess board so that no queen can attack another. That is, no two queens are on the same row, column or diagonal of the board. Suppose that the chess board is represented by an $N \times N$ matrix. Then, if one queen is in position (i_1, j_1) and the other is in position (i_2, j_2), they can attack each other if either $i_1 = i_2$, or $j_1 = j_2$, or $|i_1 - i_2| = |j_1 - j_2|$.

The sequential algorithm starts with an empty board and places queens row by row. For any partially filled board, it finds all the positions in the next row that do not conflict with the already placed queens and calls itself recursively for exploring all such positions further. Figure 6.1 shows the pseudo-code for the SeqQueens function.

The process can be associated with a dynamically evolving tree. A node of the tree corresponds to a board filled up to certain row and its children to the same board with a new queen added to the next position. The problem can be parallelized by assigning different branches of the tree to the different processors. It is dynamic for it is impossible to determine a priori the amount

```
*************************************************************************
Recursive procedure for the N queens problem
*************************************************************************

SeqQueens (Row, Queens)
int Row;        \* The row to be filled *\
int Queens[N]   \* Vector containing the positions
                   of the already placed queens *\
{
  int Column;
  if (Row == N) { \* A solution has been found *\
    return;
  }

  for (Column from 0 to N-1) {
      \* Checks if the queens are in a non attacking position *\
      if (NonAttack (Queens, Row, Column) ) {
        Queens[Row] = Column \* Places a new queen *\
        SeqQueens (Row+1, Queens)
      }
  }
}

*************************************************************************
Function NonAttack checks if the new queen is in a non-attacking position
It returns 0 if the queen is in attacking position,
it returns 1 otherwise
*************************************************************************
int NonAttack (queens, row, col)
int queens[], row, col;
{
  int i, j;
  for (i from 0 to row-1) {
      j = queens [i];
      if ( j == col or | row - i | == | col - j | )
        return(0);
  }
  return(1);
}
```

Figure 6.1: Pseudo-code for the sequential N queens function

```
# define N
struct task {
        int NextRow;  \* next position to be filled *\
        int Queens[N]; \* the positions of the already placed queens *\
};
typedef struct task Task;
```

Figure 6.2: Structure Task for the N queens problem

of work associated with each subtree. Furthermore, the problem is fully asynchronous for the outcome of the search through one subtree is irrelevant to the other searches. In the next section, we discuss the parallel implementation of the N queens problem using PMESC.

6.3.2 The parallel approach. The first step towards the implementation of a task-parallel application is the identification of the tasks. In this particular case, a task may consist of finding all the possible positions of a queen in the next row given a partially filled board. This task corresponds to a single step in the recursive procedure SeqQueens.

Having identified the tasks, the next step is to define the corresponding structure Task that contains all the information that a processor needs to execute a task. The structure Task is composed of two fields: one that corresponds to the location of the already placed queens and the other that corresponds to the next row to be filled. Figure 6.2 shows the structure Task corresponding to the N queens problem.

Next, it is necessary to make some decisions regarding the queue. One decision is whether the queue should be centralized or distributed. Because we are interested in a scalable approach, we assume a distributed queue. Thus, the input parameter TYPEP1 corresponding to the routines of the Partition module, i.e., the routines that handle the queue, must be set to ‘d’. Another

```
****************************************************************************
File user-types.h for the N queens problem
****************************************************************************

#define N
#define TYPEP1 'd'
#define TYPEP2 'L'

struct task {
        int NextRow;  \* Next position to be filled *\
        int Queens[N]; \* Vector containing the positions
                          of the already placed queens *\
};
typedef struct task Task;
```

Figure 6.3: File user-types.h for the N queens problem

decision regarding the queue is whether it should be LIFO or prioritized. In this case, because all possible positions must be tried in any order without affecting either the performance or the correctness of the results, the queue should be LIFO. Therefore, the input parameter TYPEP2 corresponding to the routines of the Partition module, must be set to 'L'. These and other definitions performed by the user should be included in a file called "user-types.h". Figure 6.3 shows the file "user-types.h" for this example.

Once the tasks, the structure Task, and the queue have been defined, it is necessary to implement the InitPart-and-Map function that creates the first task or tasks and places it or them in the queue. The programmer may either assign some tasks to all the processors so they can all create their local queues and quickly begin the parallel execution or leave the initial task in a single processor, ROOT, and let the others obtain their work by using the load balancing mechanisms. For this example, we follow the second approach. The pseudo-code for this simple function is shown in Figure 6.4. In the pseudo-code, ROOT is any processor selected by the user.

```
#include "types.h"
#include "user-types.h"

InitPart-and-Map (my-id, num-proc)
int my-id, num-proc;
{
  int i;
  Task T;

  if (my-id == ROOT) { \* **ROOT is defined in "user_types.h"** *\
    T.NextRow = 0;
    ENQUEUE(TYPEP1, TYPEP2, T);
  }
}
```

Figure 6.4: Pseudo-code for the InitPart-and-Map function

Next comes the Solve phase. In this case, it consists of the implementation of the function that performs the algorithm and that we called ParQueens. ParQueens is based on its sequential counterpart, i.e., the SeqQueens function. Like the SeqQueens function, ParQueens loops through all the columns in a row searching for every feasible position —i.e., non-attacking position. Unlike SeqQueens that calls itself recursively, ParQueens creates a new task for every feasible position and terminates. The pseudo-code for the ParQueens function is presented in Figure 6.5.

Now, remember that all these procedures correspond to different modules that need to be put together by a main program. We implement this code by following the skeleton shown in Figure 5.4 of Chapter 5 that corresponds to the distributed queue approach. For this example, we assume the ring-based, sender-initiated load balancing approach. Therefore, we use the MAP_Ri routine of the Map module with parameter TYPE1 set to SENDER. We add this new definition to the user-types.h file. Fig 6.6 shows the pseudo-code for the main program. The procedure contains a loop that corresponds to the case where the processor is moderately loaded. The loop breaks when the processor becomes either overloaded or idle. In that case, the processor invokes the load balancing routine. Because the selected approach is sender-initiated, an overloaded processor will try to send some work away to a neighboring processor in a ring even without its being requested. An idle processor calls the load balancer to check whether it has received any work or to check for termination.

At this point, the code is completed. There is however, a problem with this implementation: it creates a large number of short-lived tasks. This

```
**************************************************************************
The ParQueens function
**************************************************************************
#include "types.h"
#include "user-types.h"

ParQueens (T)
Task T;
{
      Task newT;
      int i, row, column;

      row = T.NextRow;

      if (row == N) new solution found;
      else {
           for ( column from 0 to N-1 )
               if ( Non-Attack ( T.Queens, row, column ) ) {
                 newT.NextRow = row + 1;
                 for ( i from 0 to row-1 )
                     newT.Queens[i] = T.Queens[i];
                 newT.Queens[row] = column;
                 ENQUEUE (TYPEP1, TYPEP2, newT);
               }
      }
}
```

Figure 6.5: Pseudo-code for the Solve function of the N queens problem

```
*************************************************************************
Main procedure
*************************************************************************
#include "types.h"
#include "user-types.h"

main()
{
        int i, grain, l-b, U-b;
        int me, num-proc;
        int queue-length = 0;
        Task T;

        \* l-b and U-b are lower and upper bounds for the system load *\
        Read l-b, U-b;

        my-id = Get processor id;
        num-proc = Get number of processors;

        InitPart-and-Map(my-id, num-proc);

        for (;;) {
            queue-length = DEQUEUE(TYPEP1, TYPEP2, T);
            while (l-b < queue-length < U-b) {
                  ParQueens(T);
                  queue-length = DEQUEUE(TYPEP1, TYPEP2, T);
            }

            if (queue-length > U-b) {
            \* processor is busy and gives some work away *\
                  MAP (TYPE1, TYPE2, TYPE3, BUSY);
                  ParQueens(T);
            }

            \* processor is idle. checks if more work is received
            and if not, checks for termination *\
            else if (MAP (TYPE1, TYPE2, TYPE3, IDLE) break;
        }

        GATHER results;
        if (my-id == ROOT) print results;
}
```

Figure 6.6. Pseudo-code for the main program of the dynamic N queen problem

is not recommended for PMESC which is designed for medium- to coarse-grain problems. We can overcome this problem by introducing a threshold that controls the grain size from the Solve function. Figure 6.7 depicts the new ParQueens function that allows the user to control the grain size of the application. It does so by not creating new tasks when the number of rows to be filled in the board is less than a given threshold. Instead, the function proceeds recursively as in the sequential case.

A different way of adjusting the granularity of a problem is changing the definition of the tasks. We illustrate this situation in the bisection example.

6.4 Parallel Bisection

6.4.1 Description of the problem. Another example of a dynamic, fully asynchronous problem is given by the computation of the eigenvalues of a matrix by the parallel bisection procedure. We begin by describing the sequential method and then discuss the parallel one. The problem consists of computing some or all the eigenvalues of a symmetric, tridiagonal, $n \times n$ matrix

$$\mathcal{T} = \begin{pmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-1} & \alpha_{n-1} & \beta_n \\ & & & \beta_n & \alpha_n \end{pmatrix},$$

where $\beta_j \neq 0$, $j = 2, \ldots, n$ by using bisection. This method is based on a matrix property that the number of eigenvalues of $\mathcal{T}$ smaller than any real λ is equal to the number of negative terms in the sequence of the determinants of the leading principal minors of the matrix $\mathcal{T} - \lambda I$. This property allows us

```
**************************************************************************
The ParQueens function
**************************************************************************
#include "types.h"
#include "user-types.h"

ParQueens (T)
Task T;
{
      Task newT;
      int i, row, column;

      row = T.NextRow;

      if (row == N) new solution found;

      else if ( ( N - row ) < grain ) \* Henceforth it does not create new tasks *\
             SeqQueens (T);

      else {
           for ( column from 0 to N-1 )
               if ( Non-Attack ( T.Queens, row, column ) ) {
                 newT.NextRow = row + 1;
                 for ( i from 0 to row-1 )
                     newT.Queens[i] = T.Queens[i];
                 newT.Queens[row] = column;
                 ENQUEUE (TYPEP1, TYPEP2, newT);
               }
      }
}
SeqQueens (T)
Task T;
{
  int row, column;

  row = T.NextRow;

  if (row == N) new solution found;

  for (column from 0 to N-1)
      if (Non-Attack (T.Queens, row, column) ) {
        T.NextRow = row + 1;
        T.Queens[row] = column;
        SeqQueens (T);
      }
}
```

Figure 6.7: Pseudo-code for the Solve function of the N queens problem

to compute the number of eigenvalues of $\mathcal{T}$ in any interval as the difference between the eigenvalue counts at the endpoints of that interval.

The sequential bisection procedure begins with an interval or set of intervals known to contain the desired eigenvalues. It takes an interval containing at least one eigenvalue and splits the interval into halves. It then determines the number of eigenvalues in each half by computing the eigenvalue count at the interval midpoint. If only one half contains eigenvalues, the initial interval is replaced with that half. Otherwise, the interval is replaced with one half while the other is kept in a queue. The process repeats recursively until the length of the interval is less than a given threshold. In that case, the midpoint of that interval is taken as one computed eigenvalue. When the bisection procedure finishes the computation of an eigenvalue, it selects another interval from the queue to start a new computation. This process repeats until the queue becomes empty. The process can be associated with a dynamic tree of intervals where each node of the tree corresponds to an interval, the children nodes to its subintervals, and the leaves to the computed eigenvalues. It can be represented by the pseudo-code in Figure 6.8.

The bisection procedure presents a straightforward case for parallelization. If the initial interval (intervals) containing the desired eigenvalues is (are) split among the processors, each processor can apply the sequential bisection procedure independently and asynchronously to each of its assigned intervals. No information has to be shared by the processors and no priorities need to be associated with the tasks. All non-empty intervals must be bisected, and the order does not matter. There is however, a problem that may prevent

```
***************************************************************************
Recursive bisection procedure
***************************************************************************

Initialize;
a             = left end of initial interval;
b             = right end of initial interval;
ca            = eigenvalue count at a;
cb            = eigenvalue count at b;
Threshold     = desired accuracy;
eigen-count   = function that performs the eigenvalue count;

RecBisection (a,b,ca,cb,Threshold)
{
  double mid; \* midpoint of interval [a, b] *\
  int cm;     \* eigenvalue count at mid *\

  while ( | b - a | > Threshold) {
        mid = (a + b) / 2;
        cm = eigen-count (mid);
        if (ca < cm and cm = cb) {
          RecBisection (a,mid,ca,cm,Threshold);
        }
        else if (ca = cm and cm < cb) {
              RecBisection (mid,b,cm,cb,Threshold);
        }
        else {
              RecBisection (a,mid,ca,cm,Threshold);
              RecBisection (mid,b,cm,cb,Threshold);
          }
  }
}
```

Figure 6.8: Pseudo-code for the sequential bisection function

parallel bisection from achieving optimal performance: it is impossible to estimate the amount of work associated with the computation of the eigenvalues until they are isolated in an interval. Therefore, there are cases where there is no way to efficiently partition the initial task among the processors at the beginning of the computation so that they can all finish at roughly the same time. This characteristic is what makes parallel bisection a dynamic problem. In the next section we discuss the parallel approach and how to use PMESC to implement it.

6.4.2 The parallel approach. To begin, the programmer needs to identify the units of work or tasks. In the parallel bisection example, the original work consists of computing some or all the eigenvalues of a matrix by applying bisection to an initial set of intervals containing them. A tentative task may consist of applying a single bisection to a non-empty interval — i.e., an interval that contains at least one eigenvalue. One can think of this task as a single step of the recursive procedure RecBisection or as traversing a single step down the tree of intervals i.e., a step that goes from the parent to one of its children. However, as in the N queens example, these tasks may involve little computation (depending on the matrix size.) To overcome this problem, we consider another task that consists of applying bisection to a non-empty interval, then to one of its non-empty subintervals and so forth, until an eigenvalue is computed —i.e., until the length of the subinterval is less than the threshold. This new task can be thought as traversing a single path from any node in the tree of subintervals to one of the leaves (one computed eigenvalue.) This is the approach we choose.

With the tasks identified, the user must define the variable type Task

```
struct task {
          double a;         \* left endpoint of interval *\
          double b;         \* right endpoint of interval *\
          int ca;           \* eigenvalue count for a *\
          int cb;           \* eigenvalue count for b *\
          double Threshold; \* acceptance criterion *\
};
typedef struct task Task;
```

Figure 6.9: Struct Task for the bisection problem

that represents them. A Task is a C structure that contains the data that a processor needs to execute a task. In this case, it is composed of the endpoints of the interval to which bisection will be applied, the eigenvalue count at each of the endpoints (to determine the number of eigenvalues in that interval), and a threshold to decide when to stop the subdivision. The structure Task for the parallel bisection case is shown in Figure 6.9.

Next, it is necessary to make some decisions regarding the queue of tasks. One decision is whether the queue should be centralized or distributed. In this example we use both approaches to illustrate the main differences between the two implementations. We use a centralized queue first and then a distributed one. Another decision regarding the queue is based upon whether the tasks should be assigned different priorities or not. In the bisection case, all the non-empty intervals must be bisected but the order is irrelevant. Thus, a LIFO queue is appropriate for this case. Therefore, the arguments TYPEP1 and TYPEP2 in the routines that interact with the queue (in the Partition module) must be set to 'c' (for centralized) and 'L' (for LIFO) respectively. These and other PMESC-related definitions are included in the file "usertypes.h" shown in Figure 6.10.

The next step is to implement the code for the InitPart-and-Map function. A possible implementation of this function, makes processor ROOT

```
#define ROOT 0
#define TYPEP1 'c'   \* sets queue to centralized *\
#define TYPEP2 'L'   \* sets queue to LIFO *\

struct task {
            double a;
            double b;
            int ca;
            int cb;
            double Threshold;
};
typedef struct task Task;
```

Figure 6.10: File user-types.h for the parallel bisection example

read the initial interval or intervals, create as many subintervals as processors and send them to the processors so they can all quickly begin with parallel execution. Another possible implementation of InitPart-and-Map will leave the initial interval or intervals in the central queue maintained by ROOT. The programmer decides what is best for the particular application. Our implementation of the InitPart-and-Map function follows the first approach, and it is presented in Figure 6.11.

Having defined the tasks, assigned subintervals to the processors, and created the central queue, the next step is to implement the ParBisection algorithm that corresponds to the Solve function in the PMESC framework. Our implementation is based on the sequential bisection procedure RecBisection. ParBisection selects a task from the queue and applies bisection to it. However, unlike RecBisection, ParBisection does not call itself recursively. Instead, when it creates a new task, it places it in the queue. When ParBisection finishes with the current task, it terminates. The pseudo-code corresponding to this procedure is given in Figure 6.12.

Once the pseudo-code for the Solve function has been finished, it

```
**************************************************************************
Function for the initial partition and assignment of work
**************************************************************************
#include "types.h"
#include "user-types.h"

InitPart-and-Map (my-id, num-proc)
{
  Task T;
  if (my-id = ROOT) {
    Read Interval;
    Split Interval into SubIntervals;
    send a SubInterval to each processor;
  }
  else
    receive Interval;

  T.a = left endpoint of Interval;
  T.b = right endpoint of Interval;
  T.ca = eigen-count (T.a);
  T.cb = eigen-count (T.b);

  ENQUEUE(TYPEP1, TYPEP2, T);
}
```

Figure 6.11. Pseudo-code for InitPart-and-Map function corresponding to the parallel bisection example

```
*****************************************************************
Function corresponding to the Solve phase
*****************************************************************
#include "types.h"
#include "user-types.h"

ParBisection (T)
Task T;
{
  Task T1;
  double mid;
  int cm;

  while ( | T.b - T.a | > T.Threshold) {
        mid = (T.a + T.b) / 2;
        cm = eigen-count (mid);
        if (T.ca < cm and cm = T.cb) {
          T.b = mid;
          T.cb = cm;
        }
        else if (T.ca = cm and cm < T.cb) {
               T.a = mid;
               T.ca = cm;
             }
             else {
               T1.a = mid;
               T1.ca = cm;
               T1.b = T.b;
               T1.cb = T.cb;
               ENQUEUE (TYPEP1, TYPEP2, T1);
               T.b = mid;
               T.cb = cm;
             }
        }
  }
}
```

Figure 6.12. Pseudo-code for the Solve function corresponding to parallel bisection

is necessary to design the main procedure that puts all the pieces together in order to complete the parallel implementation. Because we choose a centralized queue approach for this example, we use the corresponding framework depicted in Figure 5.2 of Chapter 5. Observe that, because the queue is centralized, no load balancing is necessary for this approach. The pseudo-code for the main code is given in Figure 6.13.

Next, we show how to turn this centralized queue approach into a distributed queue approach. For the distributed queue case, we can use the same definition of Task, as well as the same InitPart-and-Map and ParBisection functions as in the centralized case. One change consists of setting the parameter TYPEP1 in the file "user-types.h" to 'd' (for distributed.) Another, more substantial, change consists of modifying the main code so that it follows the lines of the distributed queue framework shown in Figure 5.4 of Chapter 5. Now, because the distributed approach requires load balancing, we need to decide which approach to use. For this example, we assume a random, receiver-initiated approach. Therefore, the routine to use is MAP_Ra with its parameter TYPE1 set to RECVR. The pseudo-code corresponding to the main procedure using a distributed queue is shown in Figure 6.14. Because the approach is receiver-initiated, a call to MAP_Ra from a BUSY processor will only transfer work if it has been requested by another processor. An IDLE processor will actively search for work, and, if it cannot find any, it will check for termination.

6.5 Parallel Local Adaptive Quadrature

6.5.1 Description of the problem.

Calculation of the area under a curve by parallel adaptive quadrature provides another simple example

```
****************************************************************
The main procedure using a centralized queue
****************************************************************
#include "types.h"
#include "user-types.h"
Initialize;

main()
{
  int my_id, num_proc, queue_length;
  Task T;

  my_id = Get processor id;
  num_proc = Get number of processors;

  InitPart-and-Map (my_id, num_proc);

  queue_length = DEQUEUE (TYPEP1, TYPEP2, T);
  while (0 < queue_length) {
        ParBisection (T);
        queue_length = DEQUEUE (TYPEP1, TYPEP2, T);
  }
  GATHER (results);
}
```

Figure 6.13. Pseudo-code for the main procedure corresponding to parallel bisection using a centralized queue approach

```
*******************************************************************
The main procedure using a distributed queue approach
*******************************************************************
#include "types.h"
#include "user-types.h"

main()
{
  int my-id, num-proc, l-b, U-b, queue-length;
  Task T;

  my-id = GetID();            \* get processor id *\
  num-proc = GetNumPr(); \* get number of processors *\

  Read l-b and U-b; \* gets lower and upper bounds for queue *\

  InitPart-and-Map(my-id, num-proc);

  while ( signal != -1 ) {
        queue-length = DEQUEUE (TYPEP1, TYPEP2, &T);
        while (l-b < queue-length ≤ U-b) { \* processor moderately loaded *\
              ParBisect (T);
              DEQUEUE (TYPEP1, TYPEP2, &T); \* gets task from queue *\
        }

        if (queue-length > U-b) \* processor heavily-loaded *\
              MAP_Ra (RECVR, BUSY, my-id, num-proc);

        if (queue-length ≤ l-b) \* processor lightly-loaded or idle *\
              signal = MAP_Ra (RECVR, IDLE, my-id, num-proc);
  }
}
```

Figure 6.14. Pseudo-code for the main procedure corresponding to the parallel bisection example using a distributed queue approach

of a dynamic problem. First, we describe the sequential procedure and then we present the approaches for parallelizing it. The problem consists of calculating a numerical approximation Q to an integral

$$I_f = \int_D f(x)dx,$$

where $f(x)$ is a given function and D a domain. The discussion is based on the one-dimensional case, but it can be extended to multivariate integration.

A simple way to solve the above problem is to split the domain $D = [a, b]$ into subintervals and fit the space under the curve with a series of trapezoids with width equal to the size of the subintervals. The sum of the areas of those quadrilaterals estimates the area underneath the curve, and, consequently, the thinner the intervals, the better the approximation. This approach is called the **trapezoidal rule**.

According to the trapezoidal rule, if we split the interval $[a, b]$ into n subintervals of size $h = (b - a)/n$ and define $x_0 = a$, $x_i = x_0 + ih$, $x_n = b$, then I_f can be approximated by

$$Q_n = h \sum_{i=1}^{n-1} f(x_i) + \frac{h}{2}(f(x_0) + f(x_n)).$$

The error, $E_n = I_f - Q_n$, is the sum of the errors on all subintervals [49],

$$E_n = -\frac{f''(\eta)h^2(b-a)}{12}, \tag{6.1}$$

where η is in $[a, b]$. However, because $f''(\eta)$ is usually unknown, it is necessary to find other ways to determine if Q_n is a good approximation to I_f.

According to equation (6.1),

$$I_f - Q_n = -\frac{f''(\eta)h^2(b-a)}{12},$$

when n subdivisions are used. If the interval is subdivided at $2n$ division points, we have

$$I_f - Q_{2n} = -\frac{f''(\zeta)(h/2)^2(b-a)}{12}, \tag{6.2}$$

where ζ is a point in $[a,b]$ not necessarily equal to η. Assuming that $f''(x)$ is reasonably constant over $[a,b]$ so that $f''(\eta) \approx f''(\zeta)$, we have

$$I_f - Q_{2n} \approx \frac{1}{4}(I_f - Q_n).$$

Thus, we expect the error resulting from using $2n$ subdivision points to be roughly 4 times as small as the error obtained by using n subdivision points. Multiplying equation (6.2) by 4 and subtracting $(I_f - Q_{2n})$ from both sides, we get a computable error estimate:

$$E_{2n} = I_f - Q_{2n} \approx \frac{1}{3}(Q_{2n} - Q_n)$$

In general, a more conservative upper estimate is used for the error:

$$|I_f - Q_{2n}| \leq |Q_{2n} - Q_n|/2.$$

Thus, in order to determine how good an approximation Q_n is of I_f, we halve the interval, apply the trapezoidal rule to each half to obtain a new approximation Q_{2n}, and estimate the error in Q_{2n} as the difference $|Q_{2n} - Q_n|/2$. This procedure can be recursively applied to the whole interval $[a,b]$ until the error achieves a given tolerance.

However, suppose that $f(x)$ is "badly behaved" (i.e., rapidly varying) only on a small subinterval $[\alpha, \beta]$ of $[a,b]$. In that case, the trapezoidal rule with relatively few subdivisions will produce fairly accurate results for $\int_a^\alpha f(x)dx$ and $\int_\beta^b f(x)dx$, although that will not be the case for $\int_\alpha^\beta f(x)dx$. In that case, doubling the number of division points (splitting the subintervals) for the whole

```
AdaptiveQuad (Interval, Area, Tolerance)
{
  [SubInterval1, SubInterval2] = Subdivide (Interval);
  SubArea1 = TrapezoidRule (SubInterval1);
  SubArea2 = TrapezoidRule (SubInterval2);
  NewArea = SubArea1 + SubArea2;
  Error = | NewArea - Area |;

  if (Error < Tolerance) return (Area);
  else {
       AdaptiveQuad (SubInterval1, SubArea1, Tolerance);
       AdaptiveQuad (SubInterval2, SubArea2, Tolerance);
  }
}
```

Figure 6.15: Pseudo-code for the serial adaptive quadrature procedure

interval $[a, b]$ will waste computations on those areas outside $[\alpha, \beta]$ that do not need further refinement.

Adaptive quadrature uses a quadrature rule —like the trapezoidal rule— to approximate I_f to within a given error tolerance without incurring unnecessary calculations. Although we describe the serial adaptive quadrature procedure based on the trapezoidal rule, extensions to other rules are straightforward. The adaptive technique concentrates attention on those difficult areas that need further subdivisions. For instance, in the above example, the adaptive procedure would use a large number of subdivision points in $[\alpha, \beta]$ and a relatively small number of subdivisions in $[a, \alpha]$ and $[\beta, b]$. An estimate of the error is used to decide which subintervals need to be subdivided and which not. The process repeats recursively until an acceptance criterion is satisfied [22, 76]. Figure 6.15 shows the pseudo-code corresponding to the serial adaptive quadrature procedure.

The type of information used to decide which subintervals need further subdivision and which not, i.e., the acceptance criterion, gives rise to two different approaches: local and global. In the local approach, an interval is accepted or subdivided further based on its own error estimate. In the global approach, an interval is accepted only when the global error estimate, i.e., the error over the entire domain, meets the desired tolerance.

The local adaptive algorithm can be characterized as follows:

- A subinterval R is accepted by a processor when

$$E(R) \leq TOL\frac{S(R)}{b-a}, \tag{6.3}$$

  where $S(R)$ is the length of the subinterval and TOL the error tolerance.
- Subintervals can be accepted or active, i.e., not yet accepted. Only active subintervals need further subdivision.
- The process terminates when there are no more active subintervals.

The global adaptive algorithm can be characterized as follows:

- All subintervals remain active throughout computation.
- To avoid unnecessary subdivisions, intervals are assigned priorities. Higher priorities are assigned to those regions that have bigger error estimates.
- Process terminates when the estimated error over the entire domain satisfies the acceptance criterion $E([a,b]) \leq TOL$.

The adaptive procedure (local or global) can be associated with a dynamically evolving tree of intervals where each node of the tree corresponds to an interval and its children nodes to the subintervals. Because the decisions about when to subdivide and terminate are based on error information obtained dynamically, it is difficult to estimate a priori either the number of subintervals or their computation time. A dynamic approach is especially suited to parallelizing this problem for it asynchronously checks the workload and redistributes subintervals as necessary to keep the load balanced.

6.5.2 The parallel approach. The parallel implementation of the local and global adaptive quadrature procedures requires different approaches. The advantage of the local method is that the subdivision and acceptance criteria are based solely on local information, and, therefore, no communication is required to share information. The disadvantage is that it may perform extra subdivisions and, consequently, more function calls than the ones needed to reach the tolerance for the entire domain. In contrast, the global method performs fewer subdivisions but requires interprocessor communication to compute the global error. The local method is similar to the bisection procedure: tasks are fully asynchronous and independent. The global method is different for it involves the use of a pseudo-global variable as well as prioritized tasks. We begin with the implementation of the local approach and then describe the global one. As discussed in Section 6.2, there are three procedures that users have to implement: InitPart-and-Map, Solve, and the main program.

To begin, the programmer needs to define the units of work or tasks. In the adaptive quadrature example, the original work is to estimate the integral of $f(x)$ in a given interval $[a, b]$ to a given accuracy TOL. A subunit of work or task consists of giving a subinterval on $[a, b]$ and an estimate of the integral in that interval, obtaining a new estimate by splitting the interval into two halves and applying the quadrature procedure to both halves. A task, then, is equivalent to a single step of the recursive procedure shown in Figure 6.15. To execute a task, a processor needs the interval, the current estimate

```
struct task {
        double a;        \* left endpoint of the interval *\
        double b;        \* right endpoint of the interval *\
        double Area;  \* current estimate of the integral between a and b *\
        double TOL;   \* the threshold *\
};
typedef struct task Task;
```

Figure 6.16: Structure Task for the local adaptive quadrature problem

of the integral in that interval, Area, and the tolerance, TOL. The structure Task then, is defined as shown in Figure 6.16.

With tasks identified and the structure Task defined, the user has to make some decisions regarding the queue that stores those tasks. One decision is whether the queue should be LIFO or prioritized. For the local adaptive problem, tasks do not need to be prioritized, and so a LIFO queue is appropriate. Another decision is whether the queue should be centralized or distributed. In the centralized case, a master processor keeps track of all the queues while in the distributed case every processor is responsible for handling its own queue. Because it scales well, we follow the distributed approach for our example. The "user-types.h" file that shows our decisions so far, is depicted in Figure 6.17.

The next step is the implementation of the InitPart-and-Map function which partitions the initial work into pieces and distributes them among the processors. In this example, a master processor (ROOT) sends the initial interval to the other processors so they can compute their assigned subintervals and create the first tasks to begin the execution. A pseudo-code for the InitPart-and-Map function is presented in Figure 6.18. Observe that this function, as well as the other functions defined by the user, includes the files "types.h" and "user-types.h". The former contains definitions used internally by PMESC.

```
***************************************************************************
user-types.h
***************************************************************************
# define TYPEP1 'd'
# define TYPEP2 'L'

struct task {
        double a;      \* left endpoint of the interval *\
        double b;      \* right endpoint of the interval *\
        double Area;   \* current estimate of the integral between a and b *\
        double TOL;    \* the threshold *\
};
typedef struct task Task;
```

Figure 6.17. The user-types.h file corresponding to parallel adaptive quadrature using the local approach

The latter contains the variables and parameters defined by the user.

The next step is to implement the algorithm that executes the tasks. This step corresponds to the Solve phase of the PMESC paradigm and to the Solve function in the PMESC framework. The ParAdaptQuad routine given in Figure 6.19 is the parallel version of the AdaptiveQuad function that generates tasks. Instead of splitting the intervals and calling itself recursively to execute them —as AdaptiveQuad does— the ParAdaptQuad function takes a task, executes it, creates new tasks, stores them in a queue and terminates.

Once the pseudo-code for the Solve function has been finished (i.e., ParAdaptQuad), it remains to design the main procedure that puts all these building blocks together. Basically, this procedure is as follows: processors create their own tasks, store them in their local queues, and work on them as long as they are moderately loaded. Because under a local approach each processor bases its decisions upon its own local error estimate, no sharing of information, i.e., no pseudo-global variable, is necessary. The main code for the local approach follows the line of the framework depicted in Figure 5.3 of Chapter 5.

A decision that the user has to make is what load balancing strategy to use —i.e., MAP_Ra for the random strategy or MAP_Ri for the ring-based strategy— and whether she or he wants it to be sender- or receiver-initiated. Because we assume that intense computation is concentrated around singular points, it is likely that neighboring processors will have similar loads. For that reason, we choose the random strategy that tends to distribute work globally. Regarding the choice of sender- or receiver-initiated, the user can try both by just changing an argument of the load balancing routine. The main procedure

```
************************************************************
Function for the initial partition and assignment of work
************************************************************
# include "types.h"
# include "user-types.h"

InitPart-and-Map (my-id, num-proc)
int my-id, num-proc;
{
  double x0, xn, xi, xj, h;
  Task T;

  if (my-id = ROOT) {
    Read Interval;
    Send Interval to all other processors;
  }
  else
    Receive Interval;

  \* Determine end points of my interval *\
  x0 = left endpoint of Interval;
  xn = right endpoint of Interval;
  h = (xn - x0)/num-proc;
  xi = x0 + my-id * h;
  xj = xi + h;

  \* Create first instance of Task *\
  T.a = xi;
  T.b = xj;
  T.Area = TrapezoidRule (T.a, T.b); \* Compute area between T.a and T.b *\
  T.TOL = Tolerance;

  ENQUEUE (TYPEP1, TYPEP2, T);
}
```

Figure 6.18. Pseudo-code for the InitPart-and-Map function corresponding to parallel adaptive quadrature

```
****************************************************************************
The Solve function corresponding to parallel adaptive quadrature
****************************************************************************
# include "types.h"
# include "user-types.h"

double ParAdaptQuad (T)
Task T;
{
  Task T1, T2;

  T1.a = T.a;
  T1.b = T2.a = (T.a + T.b)/2;
  T2.b = T.b;
  \* Compute area for each subinterval and
  determine if interval meets acceptance criterion *\
  SubArea1 = TrapezoidRule (T1.a, T1.b);
  SubArea2 = TrapezoidRule (T2.a, T2.b);
  NewArea = SubArea1 + SubArea2;
  LocalError = |NewArea – T.Area|;
  if (LocalError < T.TOL) return(T.Area); \* Interval is accepted *\
  else {
      \* Interval is subdivided *\
      T1.Area = SubArea1;
      T1.TOL = T.TOL;
      T2.Area = SubArea2;
      T2.TOL = T.TOL;
      ENQUEUE (TYPEP1, TYPEP2, T); \* The newly created tasks *\
      ENQUEUE (TYPEP1, TYPEP2, T); \* are stored in the queue *\
  }
}
```

Figure 6.19. Pseudo-code for the parallel adaptive quadrature procedure using the local approach

that completes the implementation of the local approach is represented by the high-level pseudo-code in Figure 6.20.

We now discuss the global adaptive quadrature procedure. One of the two main differences between the global and the local approach is that the former assigns priorities to the tasks. The priority of a task is the error estimate of the integral over the subinterval associated with the task. The interval corresponding to the task with highest priority is selected first for subdivision. The queue of tasks to use in this case is a priority queue, and the structure Task must include a new record that represents the priority associated with the task. Also, the load balancing strategy MAP_Pr that keeps the highest-priority tasks balanced among the processors is appropriate for this case. Figure 6.21 shows the pseudo-code for the main procedure using the MAP_Pr routine. Observe that the main loop of this code (i.e., the loop that corresponds to the case where the processor is moderately loaded) presents a new breaking point. In the local case this loop only breaks when the processor becomes overloaded or underloaded. However, in the global case, the main loop also breaks when the variable `accum` that counts the number of tasks that are executed without calling the load balancer, reaches a given threshold (determined by the user.) As explained in Chapter 4, this is to keep not only the load but also the priorities balanced.

The other difference from the local approach is the use of a pseudo-global variable to share information. Remember that in the global approach the acceptance criterion is based on the global estimated error calculated as the sum of the local errors over all the subintervals. Thus, in this case, processors must compute their initial local error estimates and add them up to obtain an

```
#include "types.h"
#include "user-types.h"

main()
{
  int my_node;       \* Processor id *\
  int work_nodes;    \* Number of working processors *\
  int l-b, U-b;      \* Lower and upper bounds for the queue length *\
  int queue-length;
  Task *T;

  Read l-b and U-b; \* gets lower and upper bounds for queue *\
  my_node = GET_NODE();
  work_nodes = GET_PROC();

  InitPart-and-Map (my_node, work_nodes);

  for (;;) {
      queue-length = DEQUEUE (TYPEP1, TYPEP2, T);
      while (l_b < queue-length ≤ U_b) { \* processor moderately loaded *\
            ParAdaptQuad (T);
            DEQUEUE (TYPEP1, TYPEP2, T); \* gets task from queue *\
      }
      if (queue-length > U_b) { \* processor heavily-loaded *\
            \* transfers tasks to another processor *\
            MAP_Ra (TYPE1, BUSY, my_node, work_nodes);
            ParAdapQuad (T);
      }
      if (queue_length ≤ l_b) { \* processor lightly-loaded or idle *\
            \* checks for more work or for termination *\
            signal = MAP_Ra (TYPE1, IDLE, my_node, work_nodes);
            if (signal) break;
            ParAdapQuad (T);
      }
  GATHER (ParSum);
}
```

Figure 6.20. Pseudo-code for the main procedure of parallel adaptive quadrature using a local approach

```
#include "types.h"       \* This file contains PMESC definitions *\
#include "user-types.h"  \* This file contains user definitions *\

main()
{
  int my_node;       \* Processor id *\
  int work_nodes;    \* Number of working processors *\
  int queue_length;
  int l_b, U_b;      \* Lower and upper bounds for the queue length *\
  int accum = 0;     \* Controls the task transfer frequency *\
  Task *T;

  my_node = GET_NODE();
  work_nodes = GET_PROC();

  InitPart-and-Map (my_node, work_nodes);
  for (;;) {
      queue_length = DEQUEUE (TYPEP1, TYPEP2, T);
      while (l_b < queue_length ≤ U_b &&
            accum < Thr) {
            ParAdaptQuad (T);
            queue_length = DEQUEUE (TYPEP1, TYPEP2, T);
      }
      if (queue-length > U_b) {
      \* processor heavily-loaded *\
            MAP_Pr (TYPE1, BUSY, my_node, work_nodes);
            ParAdapQuad (T);
      }
      if (queue_length ≤ l_b) {
      \* processor lightly-loaded or idle *\
            signal = MAP_Pr (TYPE1, IDLE, my_node, work_nodes);
            if (signal) break;
      }
      if (accum ≥ Thr) {
      \* time to send some tasks away to keep priorities balanced *\
            signal = MAP_Pr (TYPE1, PRIOR, my_node, work_nodes);
            if (signal) break; \* processor detects termination
            if (queue_length > -1) ParAdapQuad (T);
      }
  } \* end of infinity loop *\
  GATHER (Results);
}
```

Figure 6.21. Pseudo-code for the main procedure of parallel adaptive quadrature using a global approach

initial global error. Once the global error has been initialized, it is necessary to keep it updated. This is done by adding to the initial global error estimate the updates obtained after successive subdivisions of the intervals, i.e., after successive applications of the Solve phase. These updates are performed via the routine UPDATE as explained next.

Let us assume that $Local\_E$ is the current error estimate on any processor and that $Local\_E'$ is the new error estimate after performing the Solve phase. Then, the difference $Delta\_E = Local\_E - Local\_E'$ is an update or contribution to the global error. Now, because all the processors make their contributions to the global error in an asynchronous way, it is essential to avoid the same contribution to be added more than once. To solve this problem we use a feature of the routine UPDATE that implements a centralized approach (see Chapter 4 for more details.) In this approach, processors do not actually perform the updates until a single processor ROOT does it. However, all of them can use the local updates for an earlier detection of termination.

Figure 6.22 shows a section of the main code that illustrates this use of the routine UPDATE. In the code, we can see that processors invoke this routine upon calculation of the local error. The routine UPDATE uses a virtual tree of processors. In the centralized approach of UPDATE, the tree is traversed in both directions, up and down. When the message goes upwards, it carries the local updates from the processors to the ROOT. When the ROOT processor receives the updates, it adds them to the global error. If the updated global error meets the acceptance criterion, then ROOT broadcasts a termination message to the other processors. If it does not meet the criterion, then ROOT sends the updated value downwards. Thus, when the message goes downwards, it carries the new version of *Global_E*. In this case, processors update their own copies and pass them down the tree. When a processor (that is not the ROOT) computes a local update it adds it to its version of the global error to check if this achieves the acceptance criterion. If it does, the processor broadcasts a termination message to the other processors (without waiting for the ROOT.)

Updating in two directions has a double purpose. Upwards, it allows the processor to pass the improvements to the global error and eventually use them to make an earlier detection of the acceptance criterion without actually updating their own copies of the global error. Downwards, it allows the processors to update their copies of the pseudo-global variable. This way, only one processor can add the contributions (to avoid adding the same contribution twice) but all of them can decide when to terminate. It is important to emphasize that this complex mechanism is transparent to the user. The user only calls UPDATE with the corresponding arguments, and the rest is done automatically.

To avoid excessive communication costs associated with updating the pseudo-global variable, processors should only update their *Delta_E* value when it is significant enough, e.g., when $Delta\_E < TOL/p$. Otherwise, they should buffer it. In the next section, we study another use of the routine UPDATE.

6.6 The Traveling Salesman Problem (TSP)

6.6.1 Description of the problem.

The TSP is a discrete optimization problem in which a salesman must visit N cities and return to the starting point in such a way that minimizes the total cost of the trip. All the cities, except for the starting point, must be visited only once. Let us represent

```
#include "types.h"      \* This file contains PMESC definitions *\
#include "user-types.h" \* This file contains user definitions *\
Initialize;

main()
{
  int work_nodes;    \* Number of working nodes *\
  int my_node;       \* Processor id *\
  int queue-length;
  int l_b, U_b;      \* Lower and upper bounds for the queue length *\
  int accum = 0;     \* Controls the task transfer frequency *\
  double Global_E;   \* Local copy of the pseudo-global variable *\
  double Delta_E;    \* Local update or contribution *\
  Task *T;

  my_node = GET_NODE();
  work_nodes = GET_PROC();

  Read l_b and U_b;
  T = (Task *)malloc(sizeof(Task));
  InitPart_and_Map (my_node, work_nodes);

  for (;;) {
     queue-length = DEQUEUE (TYPEP1, TYPEP2, T);

     \* Main loop *\
     while (t_b < queue-length ≤ T_b && accum < Thr) {
           ParAdapQuad (T);
           Delta_E = Current error estimate - new estimate;
           if (Global_E+Delta_E < T.TOL)
             BCAST (termination); \* Processor detects termination *\
           else
             \* Processor sends update to parent *\
             UPDATE (my_node, work_nodes, &Global_E, &Delta_E, UP);
           queue-length = DEQUEUE (TYPEP1, TYPEP2, T);
     }
  \* Proceeds as in Figure 6.21 *\
  } \* end infinity loop *\

  \* Processor terminates *\
}
```

Figure 6.22: Partial pseudo-code for the main procedure

the cities by the set $\{1, 2, \ldots, N\}$ and the traveling cost associated with any pair of cities i and j by $c_{i,j}$. (When $i = j$, $c_{i,j}$ is assumed to be infinite.) Then, a solution of the problem is given by a permutation σ of the set of cities that minimizes $\sum_{i=1}^{N} c_{i,\sigma(i)}$.

The problem can be solved by using the branch-and-bound algorithm. This algorithm finds the solutions by searching through a tree of partial solutions that are dynamically created. The branch-and-bound procedure consists of two phases. A phase that generates new partial solutions by branching out from the current ones, and a phase that computes the cost associated with any partial solution to decide whether to discard it or to pursue it further. One can think of partial solutions as graphs whose vertices correspond to the cities and whose edges correspond to their interconnections in the traveling circuit.

The process proposed by Little et al. in [62] begins with an initial partial solution and an infinite upper bound for the cost. Given a partial solution, the branching procedure generates new partial solutions by including and excluding the best edge (determined by some selection criterion) that is not in the partial solution. The bounding procedure computes the cost associated with the partial solution. This cost is considered a lower bound of the cost of the possible solution that can be obtained by extending that partial solution. A partial solution is discarded if its lower bound is larger than the current upper bound. The upper bound is given by the least cost of the solutions computed so far and it is updated every time a new solution is found. The process repeats recursively until the solutions are found.

An interesting feature of branch-and-bound procedures is that the cost associated with the partial solutions, i.e., the nodes of the tree, can be used to direct the search towards the most promising branches of the tree. This is accomplished by assigning higher priorities to those nodes that have less cost associated with them. Nodes with higher priorities should be selected first for they are more likely to produce a solution. In the next section, we discuss the approach for parallelizing this problem in terms of PMESC.

6.6.2 The parallel approach. The first step towards implementing a task-parallel approach consists of dividing up the work into basic units of work. In this case, a task is chosen as a single step of the recursive procedure. A task then consists of giving a partial solution and applying the branch and bound procedures to it. The cost of the partial solution, obtained by the bounding procedure, is taken as the task priority.

To perform a task, a processor needs the graph corresponding to the partial solution and the cost associated with it. The structure Task containing all this information can be represented by the structure given in Figure 6.23. In the structure, the $N \times N$ matrix Graph (N being the number of cities) is generated by placing the cost $c_{i,j}$ in the position (i,j) if (i,j) is an edge in the graph corresponding to the partial solution and 0 otherwise. Kale et al. [23] propose a more efficient implementation that uses two vectors of order N and two functions that rebuild the matrix based on those vectors. That way, only two vectors instead of the entire matrix have to be sent when transferring tasks among the processors. (We use this approach in our final implementation of TSP given in Appendix A.)

Now, we need to decide what type of queue better suits the application. Because we want to use a large number of processors, we select a distributed queue. To this end, the input parameter TYPEP1 corresponding

```
define N \* number of cities *\

struct task {
            double Graph[N][N];
            int cost;
};
typedef struct task Task;
```

Figure 6.23: Structure Task for the TSP

to the routines of the Partition module, must be set to 'd'. Another decision is whether the queue should be LIFO or prioritized. In this particular case, tasks are prioritized and, therefore, the queue that stores those tasks must be prioritized. The input parameter TYPEP2 corresponding to the routines in the Partition module must be set to 'p'. The definition of Task as well as other definitions made by the user, are included in the file "user-types.h". This file is shown in Figure 6.24.

The process begins with a single partial solution, i.e., with a single task. In order for all the processors to begin execution, it is necessary to create more tasks. One possible way to initially distribute work among the processors is to organize them as a virtual binary tree. A selected processor ROOT begins the execution of the original task, creates new tasks, and assigns a task to each of its neighbors in the tree. The rest of the processors wait until they receive a task, execute that task, create new tasks, and assign them to their neighbors in

```
#define N \* number of cities *\
#define TYPEP1 'd'
#define TYPEP2 'p'

struct task {
            double Graph[N][N];
            int cost;
};
typedef struct task Task;
```

Figure 6.24: File user-types.h for the TSP

the tree. Figure 6.25 shows the pseudo-code for this InitPart-and-Map function.

Next, we must implement the Solve function. For this phase, we do not implement a new code as we did in the other examples. Instead, we modify the code used by Kale and his students [23] in their Charm implementation of the TSP. The motivations for this are twofold. First, we use this example to compare PMESC and Charm in Chapter 9. Second, we show that PMESC allows the reuse of existing code.

Charm divides the computation into processes called chares. A chare is executed when a message (similar to a PMESC task) containing the name of that chare and all the information necessary to execute it is selected from the queue. In this example, the chare **start** corresponds to the computation of the algorithm itself, i.e., the Solve phase of PMESC. The code for the chare start is shown in Figures 6.26, 6.27, 6.28, and 6.29. (See Appendix A for a complete version of the Charm-based TSP code.) A message to execute this chare contains a matrix representing a partial tour and the cost associated with it. The chare evaluates the cost of the new node, i.e., the cost of adding a new city to the tour, with the function SOLVE_RELAXATION and adds this cost to the partial cost. If the new cost is less than the upper bound and the partial solution corresponds to a solution, the chare updates the upper bound (with NewValue(mono_bound, updatefn(tmsg)), prints the new solution and terminates. Otherwise, if it is not a solution it branches out two new nodes (with BRANCH_OUT) and terminates.

The changes to turn the chare start code into the Solve function are the following. First, we need to transform the Charm code into a regular C function that takes a Task as an argument. Then, the PMESC Solve procedure

```
#include "types.h"
#include "user-types.h"

InitPart-and-Map (my-id, num-proc)
int my-id, num-proc;
{
  Task T, T1;

  Bi-Tree (my-id, num-neighbors, neighbors[num-neighbors]);

  if (me == ROOT)
    T = read (initial partial solution);
  else
    synchronous receive T;

  ENQUEUE (TYPEP1, TYPEP2, T);

  for (i from 1 to num-neighbors) {
      DEQUEUE (TYPEP1, TYPEP2, T);
      Solve (T);
      DEQUEUE (TYPEP1, TYPEP2, T1);
      asynchronous send T1 to neighbor[i];
  }
}
```

Figure 6.25. Pseudo-code for one possible implementation of the InitPart-and-Map function corresponding to the parallel TSP

```
chare start {
      entry LEAF : (message MSG1 *msg)
      {
        int *x;
        int n;
        int cost;
        MONO_MSG *tmsg;
        int A[Max][Max];
        int soln[Max][Max];
        int rows[Max], columns[Max];

        n = ReadValue(N);
        DETERMINE_MATRIX(msg, ReadValue(matrix), A, n);
        cost = SOLVE_RELAXATION(A, soln, n, rows, columns);
        cost += msg->cost;
        if (cost < ((MONO_MSG *) MonoValue(mono_bound))->bound)
          if (isSolution(msg, soln, n))
          {
            tmsg = (MONO_MSG *) CkAllocMsg(MONO_MSG);
            tmsg->bound = cost;
            NewValue(mono_bound, updatefn(tmsg));
            printout_new_solution(soln, n);
          }
          else
          {
            BRANCH_OUT (msg, A, n, cost, rows, columns);
          }
        CkFreeMsg(msg);
        ChareExit();
      }
}
```

Figure 6.26: Charm code for the TSP

```
BRANCH_OUT (msg, A, n, cost, rows, columns)
MSG1 *msg;
int A[Max][Max], rows[Max], columns[Max];
int n, cost;
{
  int *x;
  int i, j, temp, best, index;
  int best_i, best_j, cycle_start, cycle_end;
  MSG1 *msg1, *msg2;
  int temp1[Max], temp2[Max], temp3[Max][Max];

  best_i = best_j = (-1);
  best = find_best_zero(A, n, &best_i, &best_j);
  if (best_i != -1) {
    msg1 = (MSG1 *)CkAllocPrioMsg(MSG1, sizeof(int));
    msg2 = (MSG1 *)CkAllocPrioMsg(MSG1, sizeof(int));
    msg1->cost = msg2->cost = msg->cost;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
            index = i*n + j;
            if (IsFree(msg->isDefined, index)) {
              Reset(msg1->isDefined, index);
              Reset(msg2->isDefined, index);
            }
            else {
              Set(msg1->isDefined, index);
              Set(msg2->isDefined, index);
            }
            if (IsFree(msg->in, index)) {
              Reset(msg1->in, index);
              Reset(msg2->in, index);
            }
            else {
              Set(msg1->in, index);
              Set(msg2->in, index);
            }
        }
```

Figure 6.27: Function BRANCH_OUT in the Charm code (Part 1)

```
index = best_i*n + best_j;
if ((cost < ((MONO_MSG *) MonoValue(mono_bound))->bound &&
  (best >= INFINITY)) ||
  (cost + best <
  ((MONO_MSG *) MonoValue(mono_bound))->bound))
{
  Set(msg1->isDefined, index);
  Reset(msg1->in, index);

  A[best_i][best_j] = INFINITY;
  if (least_in_row(A, best_i, n) == INFINITY)
    if (rows[best_i] < INFINITY)
      msg1->cost += rows[best_i];
  if (least_in_col(A, best_j, n) == INFINITY)
    if (columns[best_j] < INFINITY)
      msg1->cost += columns[best_j];

  x = (int *)CkPriorityPtr(msg1);
  *x = cost + best;
  A[best_i][best_j] = 0;

  CreateChare(start, start@LEAF, msg1);
  Accumulate(acc_nodes, addfn());
}
```

Figure 6.28: Function BRANCH_OUT in the Charm code (Cont.)

```
    \* include best edge and exclude its symmetric counterpart *\
    if (rows[best_i] < INFINITY)
      msg2->cost += rows[best_i];
    if (columns[best_j] < INFINITY)
      msg2->cost += columns[best_j];
    Set(msg2->isDefined, index);
    Set(msg2->in, index);
    index = best_j*n + best_i;
    Set(msg2->isDefined, index);
    Reset(msg2->in, index);

    temp = A[best_j][best_i];
    A[best_j][best_i] = INFINITY;
    if (least_in_row(A, best_j, n) == INFINITY)
      if (rows[best_j] < INFINITY)
        msg2->cost += rows[best_j];
    if (least_in_col(A, best_i, n) == INFINITY)
      if (columns[best_i] < INFINITY)
        msg2->cost += columns[best_i];
    A[best_j][best_i] = temp;

    \* Get the edge that completes the cycle and exclude it *\
    get_cycle_edge(msg2, best_i, best_j, n, &cycle_start, &cycle_end);
    index = cycle_end*n + cycle_start;
    Set(msg2->isDefined, index);
    Reset(msg2->in, index);
    temp = A[cycle_end][cycle_start] = INFINITY;
    A[cycle_end][cycle_start] = INFINITY;
    if (least_in_row(A, cycle_end, n) == INFINITY)
      if (rows[cycle_end] < INFINITY)
        msg2->cost += rows[cycle_end];
    if (least_in_col(A, cycle_start, n) == INFINITY)
      if (columns[cycle_start] < INFINITY)
        msg2->cost += columns[cycle_start];
    A[cycle_end][cycle_start] = temp;

    A[best_j][best_i] = A[cycle_end][cycle_start] = INFINITY;

    for (i=0; i<n; i++)
      A[best_i][i] = A[i][best_j] = INFINITY;
    x = (int *)CkPriorityPtr(msg2);
    *x = SOLVE_RELAXATION(A, temp3, n, temp1, temp2) + cost;

    CreateChare(start, start@LEAF, msg2);
    Accumulate(acc_nodes, addfn());
  }
}
```

Figure 6.29: Function BRANCH_OUT in the Charm code (Cont.)

follows the lines of the Charm code: given the task, it applies the function DETERMINE_COST to compute its cost. If the cost is less than the current bound, it determines whether a solution has been found. If that is the case, it updates *bound* by using the PMESC routine UPDATE. Otherwise, it invokes the branching procedure BRANCH_OUT (see Figures 6.27, 6.28, and 6.29.) This function also needs minor changes. In the Charm code, BRANCH_OUT sends a message to a chare every time it creates new nodes in the tree. Instead, in the PMESC code, BRANCH_OUT creates new instances of the variable Task. The code corresponding to the Solve function for the TSP using PMESC is shown in Figure 6.30. The BRANCH_OUT function is depicted in Figures 6.31, 6.32, 6.33, and 6.34.

An interesting aspect of this example is the use of the routine UPDATE. Remember that in branch-and-bound procedures an upper bound is maintained to keep track of the least cost of the solutions found so far. This upper bound is used to discard those partial solutions that have a lower bound greater than or equal to the upper bound. Should they produce a solution, it would be more expensive than one that has already been found, and, therefore, that solution would not be optimal. Thus, in branch-and-bound procedures the cost of the current best solution has to be shared among the processors which gives rise to an important issue that concerns efficiency. PMESC provides support for sharing information through the implementation of pseudo-global variables. In this particular case, the variable *bound* is pseudo-global, and a call to UPDATE spreads its value among the processors. Because the pseudo-global variable in this case is monotonic, PMESC allows its efficient updating using a distributed approach. (Refer to Chapter 5 for more details on the

UPDATE routine.) In the distributed case, every processor can perform the updates as soon as it receives them but it can also ignore them when they are bigger than its own value. Another interesting feature of the routine UPDATE applied to monotonic variables is that it automatically invokes a routine of the Partition module to update the queue. This routine discards those tasks in the queue whose cost is greater than or equal to the recently updated variable. (As discussed before, it is not worthwhile to pursue those tasks further.)

Finally, it is necessary to implement the pseudo-code corresponding to the main procedure that runs on every processor. To do this, we must decide what approach to use for load balancing. In this case, because the tasks have priorities associated with them, we use the MAP_Pr routine. Also, because we assume that the system will be highly-loaded most of the time, we assume a receiver-initiated approach. The pseudo-code for this function is given in Figures 6.35 and 6.36. Observe that the main code shows similar characteristics to the main code corresponding to the global adaptive quadrature example. It presents a variable *accum* that counts the number of tasks that are executed in a row without calling the load balancing routine. The main loop or cycle breaks when the processor becomes overloaded or underloaded or when *accum* reaches a given threshold.

There are other ways to deal with the balancing of priorities. A possible variation consists of associating a level with every node in the tree. Then, a processor performs the main loop until it either becomes overloaded or underloaded, or until the level of the selected task reaches a given top level. In the first case, it calls the load balancer to balance the load. In the second, it calls the load balancer to balance the priorities. This approach prevents

```
#include "types.h"
#include "user-types"

extern int my_node;
extern int work_nodes;
extern int bound;

TSP (T)
Task *T;
{
  int cost;

  cost = DETERMINE_COST (T, N);
  cost += T->cost;

  if (cost < bound) {
    if (isSolution (T, N)) {
      bound = T.cost;
      UPDATE (my_node, work_nodes, &bound, bound, NUM-UP);
      print solution;
    }
    else
      BRANCH-OUT (T, N);
  }
}
```

Figure 6.30: Pseudo-code for the Solve function corresponding to the TSP

```
\ ************************************************************ \
This is the procedure BRANCH_OUT taken from the Charm code
and adapted for PMESC. This is done by changing the Charm calls for
PMESC calls. The lines of code changed are marked in the code
\ ************************************************************ \
BRANCH_OUT (msg, A, n, cost, rows, columns)
Task *msg;
int A[Max][Max], rows[Max], columns[Max];
int n, cost;
{
  int x;
  int i, j, temp, best, index;
  Task *msg1, *msg2;
  int best_i, best_j, cycle_start, cycle_end;
  int temp1[Max], temp2[Max], temp3[Max][Max];

  best_i = best_j = (-1);
  best = find_best_zero(A, n, &best_i, &best_j);
  if (best_i != -1) {
    msg1 = (Task *) malloc(sizeof(Task));
    msg2 = (Task *) malloc(sizeof(Task));
    msg1->cost = msg2->cost = msg->cost;
    msg1->Level = msg2->Level = Level+1;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
            index = i*n + j;
            if (IsFree(msg->isDefined, index)) {
              Reset(msg1->isDefined, index);
              Reset(msg2->isDefined, index);
            }
            else {
              Set(msg1->isDefined, index);
              Set(msg2->isDefined, index);
            }
            if (IsFree(msg->in, index)) {
              Reset(msg1->in, index);
              Reset(msg2->in, index);
            }
            else {
              Set(msg1->in, index);
              Set(msg2->in, index);
            }
        }
```

Figure 6.31: Function BRANCH_OUT adapted for PMESC (Part 1)

```
index = best_i*n + best_j;
if ( ((cost < bound) &&
  (best >= INFINITY)) ||
  (cost + best < bound) )
{
  Set(msg1->isDefined, index);
  Reset(msg1->in, index);

  A[best_i][best_j] = INFINITY;
  if (least_in_row(A, best_i, n) == INFINITY)
    if (rows[best_i] < INFINITY)
      msg1->cost += rows[best_i];
  if (least_in_col(A, best_j, n) == INFINITY)
    if (columns[best_j] < INFINITY)
      msg1->cost += columns[best_j];

  x = cost + best;
  msg1->Prio = x;
  A[best_i][best_j] = 0;

  \****************************************\
  \* This is a PMESC call that replaces
  the Charm call CreateChare(start,start@LEAF,msg1) *\
  ENQUEUE(TYPEP1,TYPEP2,msg1);
  \****************************************\
  acc_nodes++;
}
```

Figure 6.32: Function BRANCH_OUT adapted for PMESC (Cont.)

```
\* include best edge and exclude its symmetric counterpart *\
if (rows[best_i] < INFINITY)
  msg2->cost += rows[best_i];
if (columns[best_j] < INFINITY)
  msg2->cost += columns[best_j];
Set(msg2->isDefined, index);
Set(msg2->in, index);
index = best_j*n + best_i;
Set(msg2->isDefined, index);
Reset(msg2->in, index);

temp = A[best_j][best_i];
A[best_j][best_i] = INFINITY;
if (least_in_row(A, best_j, n) == INFINITY)
  if (rows[best_j] < INFINITY)
    msg2->cost += rows[best_j];
if (least_in_col(A, best_i, n) == INFINITY)
  if (columns[best_i] < INFINITY)
    msg2->cost += columns[best_i];
A[best_j][best_i] = temp;
```

Figure 6.33: Function BRANCH_OUT adapted for PMESC (Cont.)

```
    \* Get the edge that completes the cycle and exclude it *\
    get_cycle_edge(msg2, best_i, best_j, n, &cycle_start, &cycle_end);
    index = cycle_end*n + cycle_start;
    Set(msg2->isDefined, index);
    Reset(msg2->in, index);
    temp = A[cycle_end][cycle_start] = INFINITY;
    A[cycle_end][cycle_start] = INFINITY;
    if (least_in_row(A, cycle_end, n) == INFINITY)
      if (rows[cycle_end] < INFINITY)
        msg2->cost += rows[cycle_end];
    if (least_in_col(A, cycle_start, n) == INFINITY)
      if (columns[cycle_start] < INFINITY)
        msg2->cost += columns[cycle_start];
    A[cycle_end][cycle_start] = temp;

    A[best_j][best_i] = A[cycle_end][cycle_start] = INFINITY;

    for (i=0; i<n; i++)
      A[best_i][i] = A[i][best_j] = INFINITY;
    x = SOLVE_RELAXATION(A, temp3, n, temp1, temp2) + cost;
    msg2->Prio = x;

    \************************************************ \
    \* This is a PMESC call that replaces
    the Charm call CreateChare(start,start@LEAF,msg2) *\
    ENQUEUE(TYPEP1,TYPEP2,msg2);
    \************************************************ \
    acc_nodes++;
  }
}
```

Figure 6.34: Function BRANCH_OUT adapted for PMESC (Cont.)

a processor from going too deep down the tree without sharing some of its highest-priority tasks with other processors. In practice, this is implemented as follows. To the structure Task is added one more field called *Level* that represents the actual level of the node in the tree. The root has level 0. Children have the level of their parent incremented by one. In the main code, the main loop compares the level of the task to a given *Threshold.* If the level of the current task selected for execution exceeds this value, then the cycle breaks to call the load balancer. Appendix A shows the complete C code corresponding to the TSP using this approach.

```
#include “types.h”        \* This file contains PMESC definitions *\
#include “user-types.h” \* This file contains user definitions *\
Initialize;

int bound; \* Pseudo-global copy of the bound *\

main()
{
  int work_nodes;  \* Number of working nodes *\
  int my_node;     \* Processor id *\
  int queue-length;
  int l_b, U_b;    \* Lower and upper bound for the queue length *\
  int accum = 0;   \* Controls the task transfer frequency *\
  double matrix [N][N];
  Task *T;

  my_node = GET_NODE();
  work_nodes = GET_PROC();

  Read l_b and U_b;

  T = (Task *)malloc(sizeof(Task));

  if (my-id == ROOT) {
    T->Graph = read initial partial solution;
    T->cost = DETERMINE-COST (T.Graph, N);
    ENQUEUE (TYPEP1, TYPEP2, T);
  }

  bound = INFINITY;
  InitPart-and-Map (my_node, work_nodes);
```

Figure 6.35: Pseudo-code for the main procedure (Part 1)

```
for (;;) {
    queue-length = DEQUEUE (TYPEP1, TYPEP2, T);
    \* Main loop *\
    while (t_b < queue-length ≤ T_b &&
          accum < Thr) {
          TSP (T);
          queue-length = DEQUEUE (TYPEP1, TYPEP2, T);
    }

    if (queue-length > T_b) {
    \* Processor heavily loaded *\
      if (MAP_Pr (TYPE1, BUSY, my_node, work_nodes))
        UPDATE (my_node, work_nodes, &bound, bound);
      TSP (T);
    }

    if (queue-length ≤ l-b)
    \* Processor lightly loaded or idle *\
      signal = MAP_Pr (TYPE1, IDLE, my_node, work_nodes);
      if (signal == -1) break;
      if (signal == 1) UPDATE (my_node, work_nodes, &bound, bound);
      if (queue_length > -1) TSP(T);
    }

    if (accum ≥ Thr) {
    \* Time to send some tasks away to keep priorities balanced *\
      signal = MAP_Pr (TYPE1, PRIOR, my_node, work_nodes);
      if (signal == -1) break;
      if (signal == 1) UPDATE (my_node, work_nodes, &bound, bound);
      if (queue_length > -1) TSP(T);
    }
} \* end infinity loop *\

\* Processor detected termination *\
GATHER (results);
}
```

Figure 6.36: Pseudo-code for the main procedure (Cont.)

CHAPTER 7

TESTING THE LIBRARY: PORTABILITY AND EASE OF USE

Although parallel computers have been available for quite some time, they are still underutilized. The factor most commonly blamed for the slow transition to parallelism is the lack of adequate software support [13, 68]. A number of key ingredients must be combined in order to develop tools that shorten and smooth this transition. One important ingredient is ease of use. It has to be possible for noncomputer scientists to develop their applications within reasonable time and effort. Existing software support requires that the researcher spend a great deal of time learning machine-dependent numerical techniques, fitting the problems into the machine architectures, studying the impact of the architecture on the efficiency, and many other computational aspects that are not even close to their areas of interest and expertise.

Another ingredient is portability. The instability of the parallel computing market has to be compensated for by the existence of flexible and portable software. Nobody knows which machines will succeed and which machines will not. Therefore, nobody wants to commit a great deal of time and money mastering specific software tools and techniques for specific platforms that may suddenly be discontinued from the market.

A last but not least ingredient is performance. As Ken Neves of Boeing Computer Services says: "Nobody wants parallelism. What we want is performance. It is the fact that going to parallelism is the only way to continue to enhance performance that makes parallelism a necessity [83]."

All in all, in order to be attractive to all audiences, a tool must be portable, easy to use, and reasonably efficient. Although these seemed to be conflicting goals in the past, several systems [6, 12, 71] have demonstrated, in recent years, that a message passing system can be implemented to be portable and reasonably efficient. This Chapter evaluates the PMESC library. Our conclusions drawn here are based on the examples described in Chapter 6. However, because those examples show a variety of situations and library uses, we have reason to believe that we can expect similar results on other problems of the same kind. As for large scale problems, we can extend our conclusions regarding portability for they are independent of the problems used. However, we cannot extend our conclusions about performance. The implementation of more complex problems will be necessary to address this issue.

This chapter is organized in four sections. Section 7.1 presents the suite of tests that are used to evaluate PMESC. Section 7.2 briefly describes the different distributed-memory computers that we have used in our experiments. Section 7.3 discusses portability in its two different aspects: porting the library code and porting the programmer's code. Finally, Section 7.4 examines the usability of PMESC from the perspective of its ease of use.

7.1 How to Test a Tool

In response to all the criticism surrounding parallel software development the Parallel Tool Consortium (Ptools) was founded [70]. It brings together representatives from all the areas of interest to address the issues of what users need and how their feedback can be incorporated effectively into the tool development process. According to Ptools, tool developers should begin with a prototype based on the following guidelines:

- the tool must respond to concrete needs in the user community,
- users should be actively incorporated to the design process cycle as the only way to ensure software usefulness and ease of use. This has been successfully tried in some cases [72].
- the tool prototype should be supported on more than one platform and behave consistently across all of them,

With these guidelines in mind, we propose that programming libraries should be evaluated according to the the following criteria:

- Tests of portability
    - ∗ Difficulty of porting the library efficiently
    - ∗ Difficulty of porting the user's code efficiently
- Tests of ease of use
    - ∗ Designer walkthrough method
    - ∗ User walkthrough method
- Tests of efficiency
    - ∗ Efficiency under different levels of complexity: the static vs. the dynamic approach
    - ∗ Efficiency obtainable by the naive user
    - ∗ Efficiency obtainable by the expert user
- Comparison with relevant packages

We have subjected PMESC to these tests and will discuss them throughout this and the following chapters. We first present the platforms that we have used for the tests.

7.2 Selecting different computers

There are three types of parallel architectures:

- SIMD (Single Instruction, Multiple Data) with distributed memory,
- MIMD (Multiple Instruction, Multiple Data) with distributed memory,
- MIMD with shared memory.

In this thesis, we address the problem of building a software tool for MIMD computers with distributed memory. In this type of computer, each processor has its own private memory, so that it is necessary to explicitly arrange for data transfer between processors if they need to work together on a single problem. Examples of distributed memory MIMD machines are the Intel iPSC/2, iPSC/860, Delta, and Paragon, the Thinking Machine Corporation CM5, and the Ncube 6400. We conducted our experiments on the Intel machines iPSC/2, iPSC/860, and Delta, and on the CM5. Our porting experiments included all four platforms. The library was developed on the iPSC/2 and then ported to the iPSC/860, the Delta and the CM5. The performance experiments included the iPSC/860 and the CM5. Next, we briefly describe each of the test systems.

7.2.1 iPSC/2. The iPSC/2 is a second generation Intel hypercube consisting of up to 128 node processors attached to an Intel 301 host processor. The system that we use at the University of Colorado at Boulder has 32 nodes. The operating system for the host processor is UNIX System V and for the node processors is NX v2.2. The host and node processors are 80386/80387 processors running at a 16 MHz clock rate. Each node has 64 kilobytes of cache memory and 4 megabytes of main memory, expandable to 16 megabytes. Nodes are interconnected by a hypercube network. A d-dimensional hypercube consists of 2^d nodes interconnected in such a way that each node has d neighbors. If nodes are binary numbered, two nodes are connected when their binary numbers differ in only one bit.

The system has an internode communication bandwidth of 2.8 MB/sec. Node communication is supported by direct-connect routing modules on each node. The direct-connect network is a variation of the wormhole routing approach that can be thought of as a switching network. In order to communicate between two nodes, a series of switches are closed and the communication path established. Only the sending and receiving processors are involved in the communication which reduces the overhead for multi-hop communications. Subcube allocation is supported, allowing multiple users to access the hypercube simultaneously. Because this machine is practically obsolete, we cannot provide its Linpack TPP performance as we do for the other ones. For more information on this machine refer to [46].

7.2.2 iPSC/860. The iPSC/860 hypercube uses the same communication hardware as the IPSC/2, but with a 40 MHz i860 RISC processor replacing the 80386/80387. This processor includes 8 MBytes of main memory. It has an 8-KByte instruction cache, a 16-KByte data cache, and multiple arithmetic units, permitting multiple operations per cycle. The iPSC/860 supports two kinds of hosts: local and remote. The local host is the System Resource Manager (SRM). A remote host is any workstation other than the SRM. The system that we use at Oak Ridge National Laboratory (ORNL) runs with a local host. The host processor runs UNIX System V/386 Release 3.2 and the nodes run NX 3.3.2.

The system is composed of up to 128 node processors with an internode communication bandwidth of 5.6 MBytes per second. According to [2], it achieves a TPP Linpack performance of 2.6 Gflops [25]. Additional information about the iPSC/860 can be found in [46].

7.2.3 Intel Touchstone Delta. The Intel Touchstone Delta is a distributed-memory MIMD computer developed by the Defense Advanced Research Projects Agency (DARPA) and the Intel Corporation. It consists of 576 i860-based nodes interconnected via a two-dimensional mesh network that has 16 rows and 36 columns. Thirty two of the columns constitute the 512 computing nodes, each with 18 MBytes of memory. The other columns are composed of I/O nodes as well as service nodes for two HIPPI interfaces. According to [2], it achieves a TPP Linpack performance of 13.9 Gflops [25].

The interconnection network employs a Mesh Routing Chip (MRC) at each node which provides an internode communication bandwidth of 25 MBytes/sec in each direction. Each MRC allows five channels, one for the node and four for its adjacent neighbors in the two-dimensional mesh. The channels consist of two unidirectional buses, one for data flowing into the MRC and one for data flowing out of the MRC. The system supports explicit message-passing, with a latency of about 75 microseconds via worm-hole routing using a packet-based protocol. Thus, after the sending node transmits a message to its MRC, the message moves from MRC to MRC until it reaches the receiving node, without interrupting the intermediate processors.

Every node in the system has a unique physical node number given by the the slot in which the node board resides. The mesh can be divided into partitions of different sizes, allowing multiple users to access the machine simultaneously. The nodes in a partition are identified by a logical node number in the range from zero to one less than the number of processors in the partition.

All partitions are rectangular. The node in the upper left corner is node 0. The node numbers then increase sequentially from left to right and top to bottom.

The operating system running on the nodes is NX/M. The software for interprocess communication is compatible with that of the iPSC/860. For more details about the Delta refer to [45].

7.2.4 CM5. The CM5 is a product of Thinking Machines Corporation that contains between 32 and 16,384 processing nodes. Each node contains a 32MHz SPARC processor, 32 megabytes of memory, and a 128-megaflop vector-processing unit. In the CM5, nodes can fetch from the same address in their respective memories to execute the same instruction (SIMD style) or from individually chosen addresses to execute independent instructions (MIMD style). In addition to the processing nodes, the CM5 contains a few control processors that act as hosts to which users can log in. This allows the system administrator to divide the nodes into groups called partitions and to assign a separate control processor to each partition as host. The control processor provides a conventional Unix-like operating system. Each processing node runs an operating system microkernel that supports a subset of the full functionality available on the control processor acting as its host. The CM5 at NCAR that we used for our experiments has only one partition with 32 nodes.

The basic architecture of the CM5 is a fat-tree, i.e., a tree with higher communication bandwidth as it goes further from the leaves. Processing nodes and control processors are located at the leaves of the fat-tree. A user partition corresponds to a subtree in the network. Messages local to a given partition are routed within the partition's subtree. To route a message from one processor to another, the message is sent up the tree to the least common ancestor of the two processors, and then down to the destination. Thus, the network provides many comparable paths for a message to go from the source to the destination processor. As it goes up the tree, a message may have several choices as to which parent connection to take. This decision is resolved pseudorandomly, i.e., by selecting from among those links that are unobstructed by other messages. After the message has attained the height of the least common ancestor of the source and destination processors, it takes the single available path to its destination. The pseudorandom choice at each level balances the load on the network. The CM5 designers claim that, thanks to this automatic load balancing, users can program the network in a straightforward manner and obtain high performance. According to them, there is a guaranteed 5 MBytes per second processor to processor data transfer regardless of other processor communication loads. Near communications can attain up to 20 MBytes per second for they may not have to be routed as far up the tree.

The CM5 achieves a TPP Linpack performance of 59.7 Gflops with 1024 nodes [2].

7.3 Portability

According to the Parallel Tools Consortium [70], one of the three aspects of the "parallel software crisis" is that "tools vary widely across current platforms, so the steep learning curve must be repeated each time a user migrates to a new machine." The situation is aggravated by the fact that parallel machines are in a rapidly evolving state, and so users can not justify climbing the learning curve unless the tool is supported on more than one platform and behaves in a consistent way across all of them.

Thus, one of the most important goals of PMESC is to achieve a high level of portability across all kinds of parallel processors. Although the PMESC prototype is geared toward distributed-memory multiprocessors, our future plans include porting it to a wider class of architectures like networks of workstations, MIMD machines with shared-memory, and even combinations of them.

In this section, we discuss two different aspects of the porting procedure: porting the user's code written using PMESC and porting the PMESC library itself. These are two independent procedures in which developers take care of porting the library and users concentrate on porting their own application codes. We have experimented with porting both the library and the application code as developers and users, respectively. Our experience as developers shows that, like in the case of any other library or system, porting the library is a time consuming process that requires a great deal of knowledge about the machine hardware and the low-level mechanisms for message passing. However, the modular design of PMESC allows the easy identification of the parts that need to be changed and the reuse of many others. Our experiments as users indicate that porting code is straightforward when using PMESC. These experiments include porting code from the iPSC/2 to the iPSC/860, to the Delta and to the CM5.

7.3.1 Porting the PMESC library. Porting the library requires knowledge of the machine architecture to which the library will be ported, of efficient embedding algorithms for matching different virtual topologies into that architecture, and of the low-level set of message passing routines that the machine vendors provide. According to our experience, the porting procedure may be sometimes easier than the others (depending on the target machine), but it is always time consuming. However, there are some advances in this field directed to facilitating this process. We discuss them in the next subsection, dedicated to the issue of porting the low-level routines of the Communicate module.

Porting the PMESC library does not mean changing its entire code. Its layered design allows us to build higher level routines and abstractions upon a few, well-defined lower-level ones that depend on the machine and/or the architecture. These routines have been isolated in the Embed and low-level Communicate modules, and so these modules may need to be changed when porting the library. The Map and high-level Communicate modules, built on top of them remain unchanged across different computers. The Partition module is, by definition, machine independent. The Solve module consists of user code, and so it is not considered part of the library.

Table 7.1 shows the PMESC routines that compose the Partition, Map, Communicate and Embed modules organized in two different layers. In the lower layer, the low-level routines of the Communicate module are based on the low-level primitives for message passing that vary with the machine. They must be changed when porting the library from one platform to another. Examples of these routines are **ASEND**, **ARECV**, **SSEND**, and **SRECV**, which perform asynchronous sends and receives and synchronous sends and receives respectively. Also, in the same layer, the routines of the Embed module are closely related to the machine architecture, and, therefore, they must be addressed when porting the library from one architecture to another. An example of these routines is **Ring** which embeds a bidirectional ring into the corresponding machine architecture. The implementation of this routine is the same on the iPSC/860 and the Ncube 6400 for both machines have the same architecture, i.e., hypercube. However, its implementation on the iPSC/860 hypercube multiprocessor is different from that on the mesh-connected Delta multiprocessor. We discuss this in more detail in Section 7.3.3.

We next discuss the porting process of the machine- and architecture-dependent PMESC routines.

7.3.2 Porting the low-level routines of the Communicate module. As explained in Chapter 5, the Communicate module takes care of the low-level communication mechanisms. The current implementation of this module is based on the low-level set of primitives for message passing that comes with each machine, and, consequently, it must be changed in order to port it from one system to another. The lack of a standard set of commands for message passing has made the process of porting this module more difficult for one has to learn the different communication libraries that come with the different machines. To illustrate the difficulty associated with porting the low-level routines of the Communicate module, we show the implementation of one of these routines on two different platforms. Figures 7.1 and 7.2 show the routine THEREIS_msg of the Communicate module for the iPSC/860 and the CM5 respectively. Both versions of the routine test whether there is any pending message. If there is at least one, they return 1 as well as the 'type' of the message and the sending processor 'node'. They return 0 otherwise. However, the system calls that they invoke are completely different. For instance, to check if there is any pending message, the routine that runs on the iPSC/860 uses the iPSC/860 system call **iprobe()** while the routine that runs on the CM5 uses the CMMD call **CMMD_mcb_pending**.

Table 7.1: The two layers of PMESC

Partition, Map, and upper-level Communicate routines	
ENQUEUE	
DEQUEUE	
LENGTH	
PARTITIONQ	
UPDATEQ	
MAP_Ra	
MAP_Ri	
MAP_Pr	
TERM-CHECK	
UPDATE	
BCAST	
GATHER	
Embed routines	Lower-level Communicate routines
Ring	ASEND
S-Tree	ARECV
Quat-Tree	SSEND
Tree	SRECV
Array	THEREIS-msg
	CANCEL-msg
	CHECK
	GET-id
	GET-num

```
int THEREIS_msg (type, node)
int *type;
int *node;
{
  int probe;

  int probe = iprobe (-1);

  if (probe) {
\* infotype and infonode are iPSC/860 system calls that return *\
\* the type and source node of the most recently probed message *\
     *type = infotype();
     *node = infonode();
  }

  return (probe);
}
```

Figure 7.1: Routine THEREIS-msg for the iPSC/860

However, the advent of a message-passing interface such as MPI [64], that is highly portable across all the machines, will certainly simplify this process. If MPI becomes a widely accepted standard, then PMESC will be based on MPI rather than on the particular packages for message passing. Thus, the PMESC library will be portable across all the machines that support MPI.

7.3.3 Porting the Embed module. In many parallel applications, the physical topology of the processors does not reflect the communication patterns of the problems, and, thus, virtual topologies can be used to separate the problems from the machines. That way, instead of changing the entire application when porting from machine to machine, we only need to change the embedding functions. The Embed module creates and embeds most of the commonly used virtual topologies into the different machine architectures. It allows the user to select from among a set of topologies the one that is most suitable for the particular application without having ~~to~~ match it

```
#include <cm/cmmd.h>

CMMD_mcb THEREIS-msg (type, node)
int *type
int *node;
{
\* mcb is a message identifier of type CMMD_mcb (integer) *\
  CMMD_mcb mcb;

\* CMMD_mcb_pending is a CMMD function *\
\* that tests whether there is a message waiting *\
  mcb = CMMD_mcb_pending (CMMD_ANY_NODE, CMMD_ANY_TAG);
  if (mcb != NULL) {
\* CMMD_mcb_tag returns the type of the message *\
    *type = CMMD_mcb_tag (mcb);
\* CMMD_mcb_source returns the source of the message *\
    *node = CMMD_mcb_source (mcb);
  }

  return (mcb);
}
```

Figure 7.2: Routine THEREIS-msg for the CM5

into the actual machine architecture. It also allows other PMESC routines that involve interprocessor communication to be implemented on a virtual machine and, consequently, to be portable across different machines.

Porting of the Embed module, requires not only a deep knowledge of the machine architectures details but also an understanding of the algorithms for efficiently embedding different virtual topologies into those machine architectures. To illustrate the complexity of the porting procedure, we show the different implementations of an embedding routine on two different architectures. Figures 7.3 and 7.5 depict the routine Ring for embedding a ring into a hypercube and a 2D-mesh respectively. Figure 7.4 shows the routine Gray called by the routine Ring that runs on the hypercube. The input parameters of Ring are the processor identifier, the number of processors, and the number of neighbors in the logical ring, and its output parameters are the identifiers of those neighbors. As can be seen, each architecture requires a different algorithm. This example illustrates the tedious work involved in implementing and porting the routines of the Embed module. This work is taken care of by the developers of the library and saves a great deal of time to the programmer. Chapter 4 provides a complete description of the different embedding algorithms that we implemented for the different machines we used.

7.3.4 Porting the upper-level routines of the Communicate and Map modules.

The upper-level routines of the Communicate module and the Map module are implemented on a virtual machine to facilitate their design as well as their portability. This idea is based on the PMESC paradigm which reflects a technique used by experienced users for writing their codes.

```
Ring (me, num_proc, p1, num_neigh, neighbors)
int me;               \* processor's id *\
int num_proc;         \* number of processors *\
int p1;               \* dummy argument used to keep the same
                         interface as in the 2-D mesh case *\
int num_neigh;        \* number of neighbors in the ring *\
int *neighbors;       \* vector containing the neighbors' ids *\
{
  num_neigh = 2;
  neighbors = (int *)malloc(num_neigh*sizeof(int));

  neighbors[0] = Gray(me, num_proc, -1);
  neighbors[1] = Gray(me, num_proc, 1);

  return;
}
```

Figure 7.3: Routine Ring for embedding a ring into a hypercube

```
Gray (me, num_proc, par)
int me;
int num_proc;
int par;
{
  int i, k, mi, Mi, c;
  int *s;

  s = malloc(num_proc*sizeof(int)); \* contains the gray sequence *\
  s[0] = 0;
  s[1] = 1;

  for (i=1; (1<<(i+1)) < num_proc; i++) {
      mi = 1 << i; \* each sequence has twice as many *\
      Mi = 2*mi;   \* elements as the previous one *\
      c = mi;
      for (k=mi; k<Mi; k++) {
          c--;
          \* generates half of the new sequence by reversing the order *\
          \* of the previous one and inserting a 1 in front of each number. *\
          \* The other half is exactly as the previous sequence *\
          s[k] = s[c] | (1 << i);
      }
  }
  \* the last sequence is generated separately to cover
  the case that num_proc is not a power of two *\
  mi = (num_proc-2)/2 + 1;
  Mi = num_proc + 1;
  c = mi;
  for (k=mi; k<Mi; k++) {
      c--;
      s[k] = s[c] | (1 << i);
  }
  for (i=0; i<num_proc; i++) {
      if (me == s[i]) { \* searches position of me in the gray sequence *\
        pos = i;
        break;
      }
  }
  if (par == 0) return (pos); \* returns position of me in the gray sequence *\
  if (par == 1) return (s[(pos+1)%num_proc]); \* returns the successor of me *\
  if (par == -1) return (s[(pos-1)%num_proc]); \* returns the predecessor of me *\
}
```

Figure 7.4: Routine for computing the Gray sequence on a hypercube

```
Ring (me, p0, p1, num-neigh, neighbors )
int me;              \* processor's id *\
int p0;              \* number of rows *\
int p1;              \* number of columns *\
int num_neigh;       \* number of neighbors in the ring *\
int *neighbors;      \* vector containing the neighbors' ids *\
{
  int x0, x1;
  num_neigh = 2;
  neighbors = (int *)malloc(num_neigh*sizeof(int));

  x0 = me / p1; \* the row where processor me is located *\
  x1 = me % p1; \* the column where processor me is located *\

  \***** if x1=0 then me is in the first column *****\
  \* the sequence goes from bottom to top *\
  \* the predecessor of me is the processor below *\
  \* and the successor is the processor above *\
  \* if me is in the last row, its predecessor is the processor to the right *\
  \* if me is in the first row, its successor is the processor to the right *\
  if (x1 == 0) {
    if ( OUT(me + p1) ) neighbors[0] = me + 1;
    else neighbors[0] = me + p1;
    if ( OUT(me - p1) ) neighbors[1] = me + 1;
    else neighbors[1] = me - p1;
  }

  \***** if the row is even, the succession goes from left to right *****\
  \* the predecessor of me is me-1 and the successor me+1 *\
  \* if me is in the last column, the successor is the processor below *\
  else if (x0 % 2 == 0) {
    neighbors[0] = me - 1;
    if ( x1 == (p1-1) ) neighbors[1] = me + p1;
    else neighbors[1] = me + 1;
  }

  \***** if the row is odd, the succession goes from right to left *****\
  \* the predecessor of me is me+1 and the successor me-1 *\
  \* if me is in the last column, the the predecessor is the processor below *\
  else {
    if ( x1 == (p1-1) ) neighbors[0] = me - p1;
    else neighbors[0] = me + 1;
    neighbors[1] = me - 1;
  }
}
```

Figure 7.5: Algorithm for embedding a ring into a 2-D mesh

```
{
  —
  —
  Ring (me, other arguments, neighbors);
  —
  SRECV (other arguments, neighbors[0]);
  ASEND (other arguments, neighbors[1]);
  —
}
```

Figure 7.6: A PMESC sample code

By using a virtual machine, we can isolate the machine- and architecture-dependent parts in the lower level, leaving the upper level unchanged. Another advantage of using a virtual machine is that, as new parallel architectures become available, we can easily port the library to them by simply changing the lower-level part. To illustrate this point, Figure 7.6 presents a sample code that could be part of any of the upper-level routines or even the user's code. In this code, an algorithm is implemented on a virtual ring of processors. A call to Ring retrieves the processor identification of the neighbors of processor 'me' in the virtual ring. Those identifications are then used by 'me' to send and receive information. This code does not have to be changed when porting it to any computer as long as we do not want to change the virtual machine, i.e., the ring that it uses.

There are, however, some situations in which it may be convenient to change the virtual machine as well. A situation like this may arise, for instance, when some routines are implemented on a virtual machine that is either impossible to embed onto some machine architecture or possible but very inefficient because it leads to much network contention or high communication cost. In either case, because our goals with PMESC are to achieve portability and efficiency, we provide the means for easily changing the virtual topology of the PMESC routines. This feature can be exploited by advanced users in order to maximize the efficiency of their implementations. To illustrate this point, we next discuss two different alternatives that the user may need to analyze when selecting an efficient broadcasting procedure.

In order to broadcast data among p processors, one can think of them as connected by a tree or by a linear array. Both virtual architectures can be embedded into all the machine architectures we have available. However, while broadcasting via a tree takes $O(\log(p))$ communication steps, broadcasting via a linear array takes $O(p)$. A first glance at these results may lead one to choose the tree. Nevertheless, a more detailed analysis shows that this alternative is not always better. In fact, each internal node of the tree has to send out more than one message for each one message it receives. Each internal node of the array, on the other hand, sends out only one message for each message it receives. The linear array thus offers a higher bandwidth at each processor. If the source node sends out only a little data, then latency time dominates, and, so, the tree-based algorithm should be chosen. If the source node sends out a lot of data, then bandwidth is more important than latency. In that case, the linear array should be chosen. The broadcast routine, BROADCAST, of PMESC uses a spanning tree. If the user wants to use an array, she or he must modify this routine so that it calls Array instead of Tree.

7.3.5 Porting the PMESC-based user's code. A program written using PMESC should require minor or no source code changes when porting around the different systems. This portability is based on the very definition of PMESC. The PMESC paradigm separates all the application-dependent aspects from the machine-dependent ones. The PMESC library,

Table 7.2. Percent of lines of user's code to change in porting from the iPSC/860 to the CM5 when using the PMESC library (assuming the use of machine-dependent I/O functions)

Using the PMESC Library			
Bisection	Quadrature	TSP	N-queens
1	1	0.5	2

based on this paradigm, takes care of most of the machine-dependent issues that need to be changed when porting an application from one machine to another.

Most of the user's code corresponds to the application itself, i.e., the Solve module, which, by definition is machine independent. The only portion of the user's code that may involve some interprocessor communication is the initial mapping of tasks to the processors. However, by using the PMESC communication routines, the user should be able to port this code as well in a straightforward manner. Thus, the only code to change might be the one corresponding to input and output of data and only in the case that this is machine-dependent. The changes in this case constitute a small percentage of the total code as it is illustrated in Table 7.2.

Table 7.2 shows the percentage of lines of user's code to change or add to port the four examples presented in Chapter 6 from the iPSC/860 to the CM5. Table 7.2 shows that this percentage is small in all the examples when using the PMESC library. (No changes are necessary to port around the Intel machines.) Table 7.3 shows that more dramatic changes are necessary to port the examples without using the library.

Table 7.3. Percent of lines of user's code to change in porting from the iPSC/860 to the CM5 without using PMESC

Without Using the PMESC Environment			
Bisection	Quadrature	TSP	N-queens
50	46	40	59

7.4 Ease of use

In her paper entitled "Where are we Headed?", Cherri Pancake [67] emphasizes that the audience for high-performance computing is not the computer science community, but scientists, engineers, and other technical programmers. She points that "they have turned to parallel processing because their problems are too big, or their time constraints too pressing, for conventional architectures." Thus, parallelism is for them a necessity, a means to solving bigger problems as well as to achieving good performance. However, parallelism remains underutilized by most of the non-computer-scientist community. The blame can be attributed to both the fact that parallelizing compilers alone are not good enough and show disappointing speedups and the fact that hand-coded parallelism is difficult and time-consuming. To attack this problem, considerable effort and expense have been devoted to developing tools that support parallel programming. However, recent evidence has shown that users do not find these tools useful for their program development needs. Workshops sponsored by such associations as ARPA, NSF, and ACM have arrived at the conclusion that "A lot of smart people are developing parallel tools that smart users just won't use" [67].

A recent survey applied to a significant cross-section of the parallel user community shows that virtually all parallel users prefer hand-coded instrumentations over current tool offerings [17]. They complain that tools are hard to learn, tedious to use, and fail to provide the information users really need and want. As Pancake points, "At best, the tool is considered clumsy and hard to learn; at worst, the user assumes it cannot provide the desired information." Scientific users must become quasi-computer scientists to understand and use the tools. To aggravate the problem, tools vary widely across current parallel platforms, so the steep learning curve must be repeated each time a user migrates to a new machine. The user community attributes the problem to the fact that parallel tools are being developed without user input. Most of the users of parallel tools do not approach programming in the same way as their designers do. Thus, in order for tools to be useful and easy to use, they should be designed according to how the user actually programs, not to how the developer thinks the user should program. User participation becomes essential for it allows designers to understand their working environment, programming habits, and application domains.

In order to evaluate the usability of PMESC, we use a method called **Programming Walkthrough**. This method, developed by Polson et. al. [73], provides a first approach to evaluating the usability of a programming tool from the perspective of its ease of use. In a walkthrough analysis, the tool designers select a sample problem and examine the steps that a user would have to go through in the process of writing a program for that problem. For each step, designers also evaluate the knowledge required to choose among different alternatives. All this information about the knowledge and reasoning required to use a tool is then used to identify weak points in the tool design as well as to construct useful user documentation. In fact, the walkthrough method is based on the idea that following the users' mental processes reveals what they need to know in order to use the tool. Users need to know, for instance, in what order certain decisions should be made or what is the sequence of steps. Some of the required knowledge is part of the general programming background that every user of the tool is expected to have. It is called basic knowledge. Some other knowledge is specific to the tool and, consequently, must be made available to its users. It is called specific knowledge [92]. Therefore, the sequence of user steps and decisions, as well as the specific knowledge behind each one of them, become the target point for evaluation.

7.4.1 Programming walkthrough. We performed two walkthrough analyses. Because, by that time, most of the major design decisions about PMESC had already been completed, the most substantial contribution of the analyses was in the area of the user documentation needed. Thus, we began the production of a user's manual. We prepared an inventory of the information that users really need to know to use the PMESC library effectively. Based on that, we developed a first draft containing not only the definitions of the library routines but also the guiding knowledge obtained thanks to the walkthrough sessions. This designer walkthrough led us to organize the first draft of the user's manual as shown in Figure 7.7.

Although the walkthrough analyses allowed us to gain some insight into what was necessary to develop a user-friendly tool, we thought that some user feedback was still needed. It is easy for a designer to overlook some trouble points or to misjudge some decisions because of the familiarity naturally acquired with the tool during the design process. So, we decided to evaluate PMESC from the perspective of users who are not familiar with it. We now describe another method for tool evaluation that involves user participation.

(1) Brief introduction and motivations for PMESC.
(2) Description of the problems that PMESC is intended to address.
(3) The PMESC paradigm:
    (a) Description of the framework for the centralized queue approach showing the cascade of routine calls to illustrate the different layers of PMESC.
    (b) Description of the framework for the distributed queue approach showing the cascade of routine calls to illustrate the different layers of PMESC.
(4) The PMESC library:
    (a) Description of the routines.
        (1) The Partition module.
        (2) The Map module.
        (3) The Embed module.
        (4) The Communicate module.
    (b) Description of the different alternatives available when using PMESC.
(5) Using PMESC step by step
    (a) Description of the steps involved in the implementation.
    (b) Presentation of the examples.
    (c) Implementation of the examples.
    (d) Discussion of alternatives.

Figure 7.7: Contents of the first draft of the user's manual

7.4.2 Programming walkthrough with user participation.

The **Goal-Centered User Evaluation** is an extension of the programming walkthrough method that evaluates a tool with user participation [72]. We refer to it as **user walkthrough**. This approach consists of performing the walkthrough analyses with users (instead of designers.) To avoid missing information that may be relevant, the user walkthroughs rely on the use of thinking aloud. The thinking aloud technique [60] complements the walkthrough by requiring users to say everything that comes to their minds when using a tool. The spontaneity of these comments is fundamental to discovering potential problems.

The user walkthrough method allows designers to follow the user's actions when implementing a problem using a given tool. It involves a number of steps. First, the designers must find the users to perform the walkthrough sessions. Designers also must select a sample problem or set of sample problems to be implemented with the tool during the sessions. Then, the designers must provide a user's manual that users should read before the walkthrough session. Finally, users implement the problem. The user is required to walk through the sequence of actions she or he considers necessary or convenient to take. At each step, the designer tries to determine the reasoning behind those actions. This process allows designers to detect potential difficulties that users may find, redundancies, concepts that require expertise, and other faults that may arise in the development of the problems. By following the sequence of tasks that the users carry out with the application, the designers can observe the tool's functionality from different points of view. The designers then, have a better idea of how the users go about their work and consequently, they can make the necessary corrections to adjust the tool to the users' minds.

In the next section, we describe the results of the user walkthrough analyses of PMESC.

7.4.3 The user walkthrough sessions.

We conducted a number of user walkthrough sessions to obtain some feedback about the ease of use aspect of PMESC. The people that volunteered to do the tests were all from the Computer Science Department at the University of Colorado at Boulder. However, they had different backgrounds, ranging from no experience in parallel computing to some experience in parallel computing but not in dynamic problems to experience in both parallel computing and dynamic problems. To give them the insight into the library and the problems, we asked them to read the user's manual. The manual described two example problems one of which was implemented with PMESC and the other of which was left as an exercise. The implemented example was parallel adaptive quadrature, and the test example was the N queens problem. The user walkthrough sessions consisted of implementing that problem using PMESC.

The sessions showed some problems that we had not seen before. While only a few of these problems were concerned with PMESC itself, many were related to the PMESC user's manual. The following is a summary of the problems and the modifications suggested by the users.

- Facilities for performing I/O operations should be added.
- The cascade of routine calls presented in sections 3.a and 3.b of the user's manual was confusing. Users had trouble differentiating the routines they had to call explicitly from the ones that were called automatically.

- The user was given many alternatives for handling different programming issues (load balancing, termination checking, embedding, and more). This versatility is something that we regard as an important feature of PMESC. However, in the way that we presented it, it had a negative impact among inexperienced users for it created a source of confusion. The suggestion was to adopt defaults that accommodate the features that the average user will understand. The other options should be left for advanced users and discussed in later sections of the manual.
- Some users adopted solution paths that we had not considered. One example is given by the initial distribution of the work. A user suggested a new way of doing it through the load balancing mechanisms. This way may slow the initiation of the parallel computation, but it is a valid procedure to consider.
- The presentation of the problems that PMESC is supposed to address was unclear and misleading (in section 2.) Most users were confused by the target problems and consequently the use of PMESC itself. Furthermore, the examples were presented too late in the manual (in section 5.) A user suggested introducing one example right after the discussion of the problems in order to provide a better picture of the situation.
- The specific vocabulary used for the manual assumed familiarity with parallel processing and parallel procedures, which created confusion. The suggestion was to define expressions like "virtual topology", "embedding", and "load balancing" in a separated section of the manual.

- Overall the biggest mistake was to present the simple features and at the same time introduce the more complex ones. The user walkthrough analyses allowed us to see clearly that we needed to separate different layers of complexity and then introduce them hierarchically, from the simpler to the more complex ones.

We discovered that additional knowledge about the types of problems and the programming issues involved in their implementation as well as a reorganization of the material we had prepared was necessary for understanding and using PMESC. The walkthrough analyses had taught us an important lesson: the usability of a tool is influenced by the presentation of the specific knowledge required to use it. No matter how good a tool might be, the user has to understand the big picture first, and, in that picture, recognize her or his problem and the applicability of the tool to implement that problem. Thus, developing a good user's manual is an important part of making the library easy to use.

7.4.4 Constructing user documentation.

Developing a good manual required an iterative process. We started with a draft that we thought would cover most of the issues. First designers and then users walked through a couple of sample problems, noting all the things that required modification or correction. We worked on a new draft and repeated the process. The analysis was very helpful for it guided us to the points where the users had trouble understanding how to proceed or how to use the appropriate functionality of the library. The user evaluation of the PMESC interface allowed us to focus attention on the knowledge that the users need to use the library and to organize all this knowledge in a way that is easy to understand. The basic and specific knowledge required for using PMESC can be summarized as follows:

- Basic knowledge
  - (1) The user must be able to read and understand C for all the examples are presented in that language.
  - (2) The user must have the basic idea that, on MIMD distributed-memory computers, each processor runs its version of the program and that the only way to share information is through message passing. Some inexperienced parallel programmers did not have a clear idea of the fundamental differences between shared- and distributed-memory computers. For instance, some of them thought that a C global variable was automatically shared by all the processors. This knowledge is essential to understanding PMESC and the motivation to use it.
- Specific knowledge
  - (1) Before selecting PMESC: The user has a problem and has to find the right tool to solve it. The following knowledge is necessary to find out whether PMESC is the right tool:
    - ∗ The user must understand that PMESC only applies to problems that can be parallelized by separating them into units or tasks that can run asynchronously on any processor.
    - ∗ The user must become familiar with concepts such as virtual topology, load balancing, and embedding.
    - ∗ The user must understand the dynamic structure of the irregular problem: tasks are created dynamically during the computation. It is impossible to predict their number or the

amount of work associated with them. Load balancing is the only way to achieve good performance, but it is also a source of overhead.

* The user must understand that, because processors are not coordinated (asynchronous) it may be necessary to check for termination periodically though asynchronously.

(2) After selecting PMESC: The user has to implement the problem with PMESC. The user should be able to do the following:

* The user should be able to define a C structure that contains all the information necessary to run a task.
* The user should be able to write or modify the code that executes the algorithm itself.
* The user should be able to write the main program that runs on every processor and that calls the PMESC routines.
* The user should be able to use PMESC to store tasks in or retrieve tasks from the queue.
* The user should be able to use PMESC to balance the loads.
* The user should be able to use PMESC to check for termination.
* The user should be able to use PMESC to communicate between processors.

The final draft of the user's manual, obtained after this analysis, is provided in [19]. Figure 7.8 shows the final organization of the user's manual. This new draft presents more sections than the previous one. Among them, a section that gives definitions, another that discusses the taxonomy of the problems, and another that presents an example, are included at the beginning of the manual. It also includes a new section that describes in detail the specific knowledge necessary to use the library right after presenting it. At the end, the final draft presents more examples and uses them to discuss different alternatives. The discussion of the alternatives at the end is another difference from the first draft, which discussed them right after presenting the library and before introducing the examples.

(1) Brief introduction and motivations for PMESC.
(2) Definitions and preliminaries.
(3) A taxonomy of the problems.
(4) Description of the problems that PMESC is intended to address.
(5) An example.
(6) The PMESC paradigm as a model.
(7) The PMESC library.
    (a) The library features.
    (b) Description of the framework for the centralized queue approach (only what the user sees rather than the cascade of routine calls.)
    (c) Description of the framework for the distributed queue approach (only what the user sees.)
    (d) Specific knowledge necessary to use the library.
    (e) Description of the routines.
        (1) The Partition module.
        (2) The Map module.
        (3) The Embed module.
        (4) The Communicate module.
(8) Using PMESC step by step.
(9) Examples.
    (a) Implementation of the examples.
    (b) Discussion of different alternatives.

Figure 7.8: Contents of the final draft of the user's manual

CHAPTER 8

TESTING THE LIBRARY: PERFORMANCE

A survey conducted among parallel, vector, and serial programmers who attended the Supercomputing '93 conference shows that their expectations of performance are extremely optimistic [17, 69]. When asked about how much improvement in performance would be likely if 100 processors were used to compute their applications in parallel, most of the respondents expected a factor of at least 50. In addition, the survey reveals that users will not adopt a tool until they can be assured of its performance payoff. Thus, performance is another important ingredient in the usability of a tool.

In this chapter we analyze the performance of PMESC in the context of the examples described in Chapter 6. Therefore, our conclusions derived here can only be applied to problems of similar characteristics. We know that the flexibility, portability, and ease of use of PMESC are not without cost. Our goal is to keep that cost as low as possible. This chapter is organized as follows. Section 8.1 examines the cases where the complex dynamic mechanisms used in PMESC outperform the simple static ones. Section 8.2 analyzes the efficiency of the PMESC library by studying the overhead associated with different functionalities it presents. Section 8.3 discusses different ways of improving the performance of a PMESC application. Finally, Section 8.4 summarizes the results of this chapter.

8.1 Static vs. Dynamic Approach with PMESC

PMESC provides support for task-parallel computations using the static or the dynamic approach. In the former, the initial work is partitioned and assigned to the processors only once. Processors then create their local queues of tasks, execute all the tasks in their queues, and terminate. There is no communication among them except at the beginning (to distribute the work) and at the end (to gather results.) Because the interprocessor communication is so scarce, the overhead associated with this approach is minimal. The situation is different in the dynamic approach. Besides executing tasks from their queues, processors continuously check their loads, dynamically transfer tasks to keep it balanced, and asynchronously check for termination. The overhead associated with this can be considerable. Obviously, if we have to implement a regular problem we use the static approach. However, if we have an irregular problem the dynamic approach should be more appropriate. The question is whether the overhead associated with the dynamic PMESC approach makes this choice actually better than the static one and how much better.

The experiments that we describe next show consistent advantages of the dynamic over the static approach on problems that are irregular. We analyze different "degrees" of irregularity to show the performance of the dynamic approach under more favorable, i.e., highly irregular, and less favorable, i.e., less irregular, situations. To show the advantage of the dynamic over the static approach, we first compare the total times under each one of the approaches without trying to justify those times. Then, we analyze the times for the dynamic approach in detail.

Our first experiment applies the parallel bisection example (described in Chapter 6) to the test matrix [1, 2, 1], i.e., a matrix that has 2's in the diagonal and 1's in the subdiagonals. Its eigenvalues are given by

$$\lambda_i = 2(1 + \cos\frac{i\pi}{n+1}), \quad i = 1, \ldots, n$$

where n is the size of the matrix. These eigenvalues are distributed in the interval [0, 4] and, as n increases they tend to concentrate at the endpoints of that interval. The case $n = 10,000$ is highly irregular for most eigenvalues are concentrated but still computationally different. Problems that present this characteristic are difficult to partition efficiently for no matter how the initial interval is distributed among the processors most of the work is likely to be assigned to a few ones. This situation is illustrated in Figures 8.1 and 8.2 that show the execution time of each processor by using the static and the dynamic approach with PMESC on the iPSC/860 and the CM5 respectively. In this case, the initial interval is partitioned into equally-sized subintervals that are assigned to the processors. The figures show that, in the static case, a few processors receive considerable more work than the rest. As the total time is determined by the slowest processor, by the time this finishes its work most of them had been idle for a long time.

The plots show that the dynamic approach outperforms the static one by a factor of more than one half. Note that in the dynamic approach processors are not synchronized. However, because they do not terminate until they receive the termination message (broadcast by the processor that detects termination) their times usually differ by only a few seconds. Figures 8.3 and 8.4 show that the advantages of the dynamic with respect to the static approach remain independent of the matrix size.

Obviously, the advantages of the dynamic over the static approach

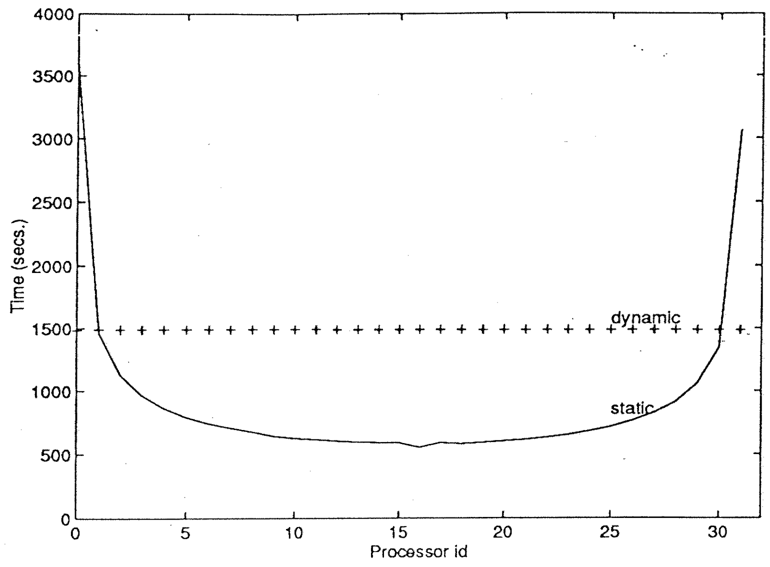


Figure 8.1. Execution times for each of the 32 processors under the static and dynamic approaches to solve bisection for the $[1, 2, 1]$ matrix of size 10,000 (on the iPSC/860.)

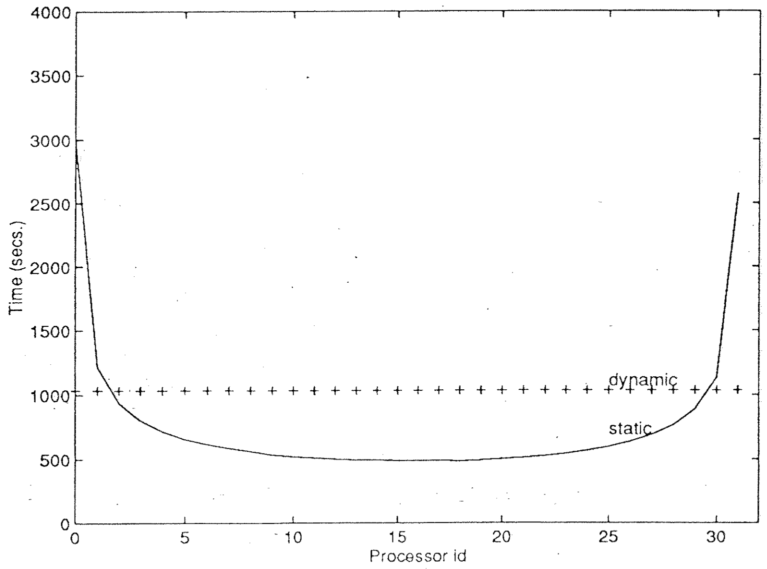


Figure 8.2. Execution times for each of the 32 processors under the static and dynamic approaches to solve bisection for the $[1, 2, 1]$ matrix of size 10,000 (on the CM5.)

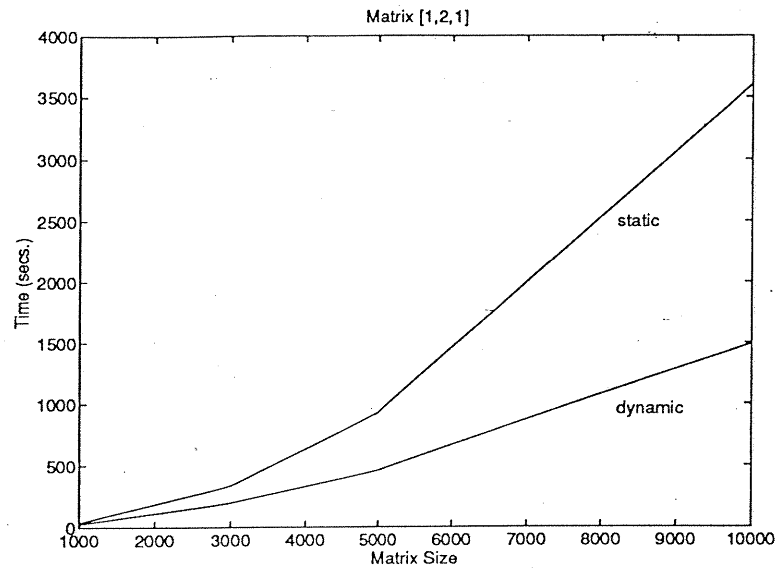


Figure 8.3. Total execution times for the static and dynamic approaches applied to $[1, 2, 1]$ matrices of different sizes (on the iPSC/860.)

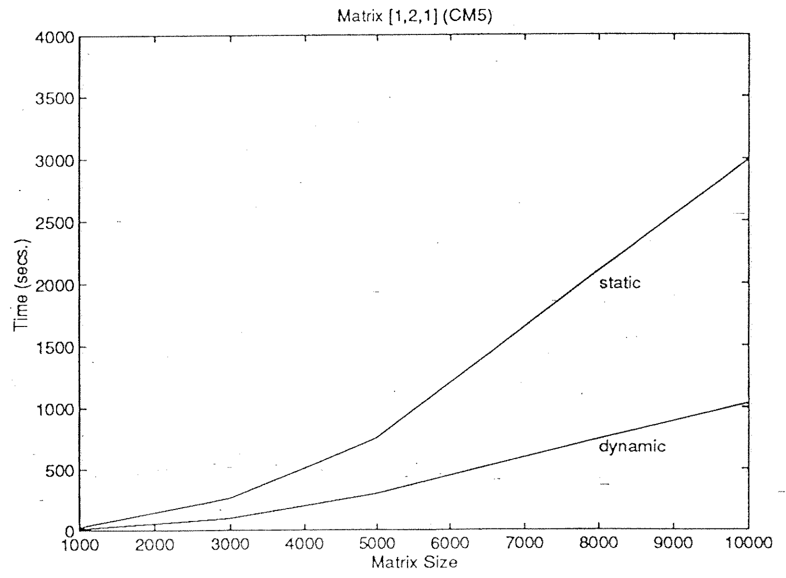


Figure 8.4. Total execution times for the static and dynamic approaches applied to $[1, 2, 1]$ matrices of different sizes (on the CM5.)

depend highly on the application. To illustrate this point, we present another experiment. It consists of applying parallel bisection to the random matrix. This matrix has uniformly distributed random elements between -1 and 1 in the diagonal and subdiagonals. Figures 8.5 and 8.6 show that the results obtained for the random matrix on the iPSC/860 and the CM5 also favor the dynamic over the static approach although not as dramatically as in the $[1, 2, 1]$ matrix case. The difference is due to the eigenvalue distribution along the spectrum. In the random matrix case, eigenvalues are more evenly distributed than in the $[1, 2, 1]$ matrix case, and, therefore, the initial distribution of work does a better job in the former than it does in the latter. Dynamic load balancing improves the initial distribution of the work for the random matrix but not as dramatically as in the $[1, 2, 1]$ case. Figures 8.7 and 8.8 compare the dynamic and static approaches applied to random matrices of different sizes.

Another experiment uses the parallel adaptive quadrature example described in Chapter 6. Figures 8.9 and 8.10 show the time distribution resulting from applying static and dynamic approaches to the parallel quadrature case on the iPSC/860 and the CM5 respectively. The plots show that because the work is concentrated in a difficult area of the domain is also unevenly distributed among the processors. The static curve shows this uneven distribution while the dynamic one shows halving of the total time when compared to the static case.

A regular problem that is suited to a static approach presents a "worst-case" scenario for the dynamic one. In this case, the calls to the load balancing routine introduce an unnecessary overhead that prevent the dynamic implementation from achieving good performance. The following experiment

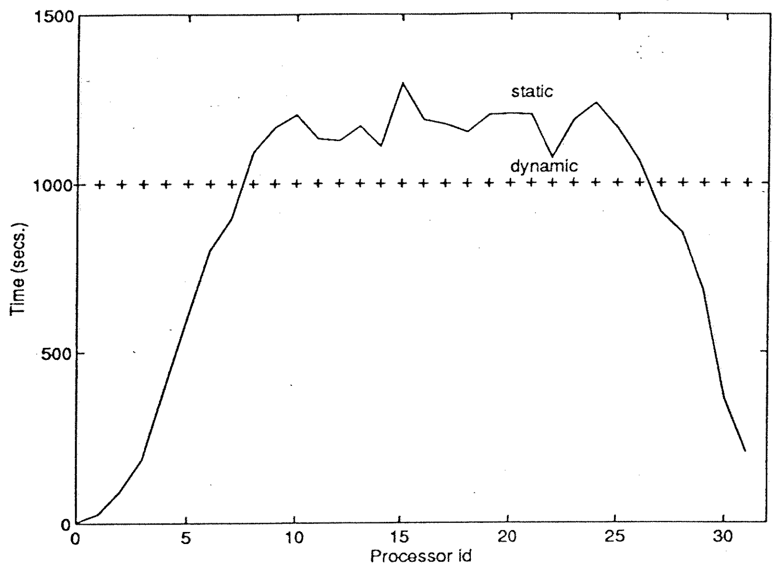


Figure 8.5. Execution time for each of the 32 processors under the static and dynamic approaches to solve bisection for the random matrix of size 10,000 (on the iPSC/860.)

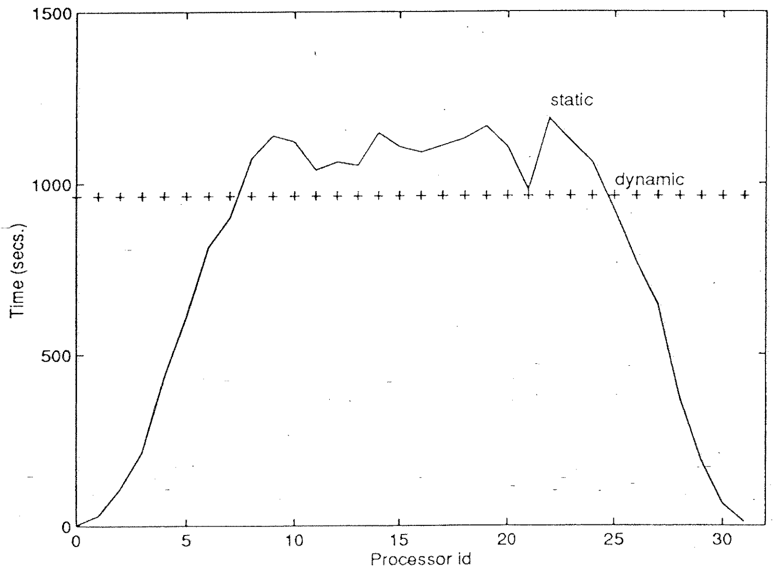


Figure 8.6. Execution time for each of the 32 processors under the static and dynamic approaches to solve bisection for the random matrix of size 10,000 (on the CM5.)

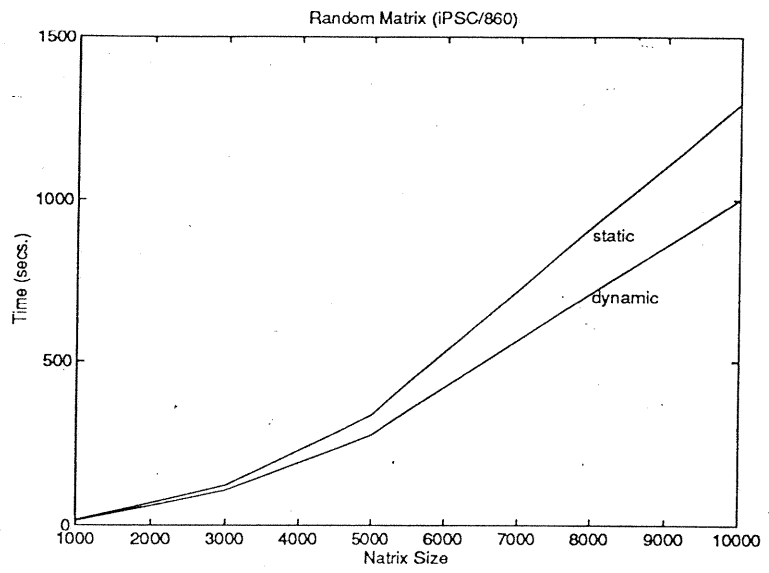


Figure 8.7. Total execution times for the static and dynamic approaches applied to random matrices of different sizes (on the iPSC/860.)

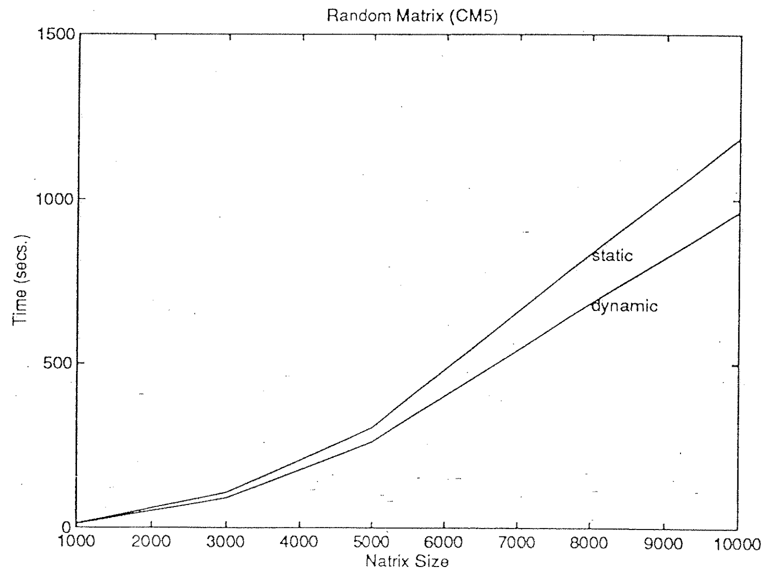


Figure 8.8. Total execution times for the static and dynamic approaches applied to random matrices of different sizes (on the CM5.)

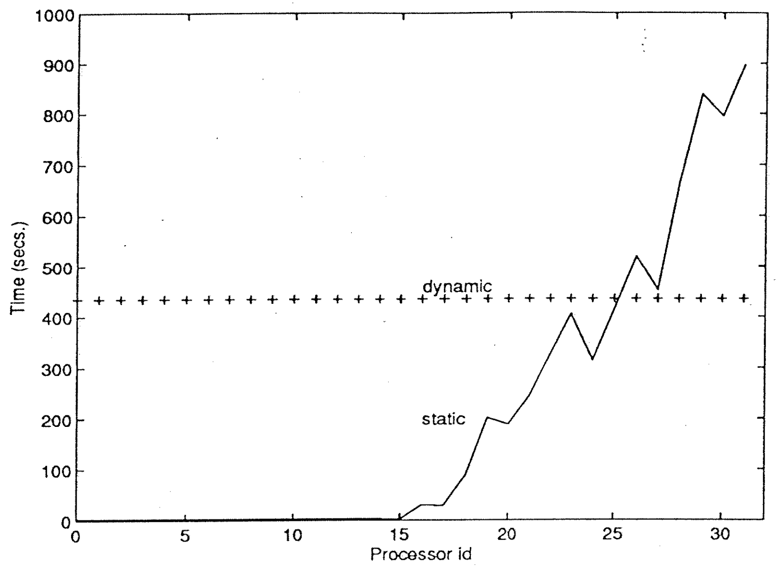


Figure 8.9. Execution time for each of the 32 processors under the static and dynamic approaches to solve the adaptive quadrature problem (on the iPSC/860.)

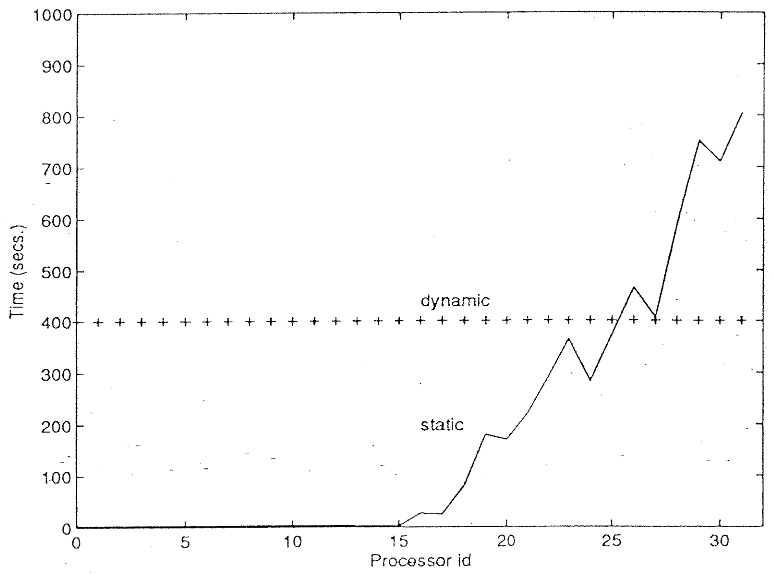


Figure 8.10. Execution time for each of the 32 processors under the static and dynamic approaches to solve the adaptive quadrature problem (on the CM5.)

illustrates the case. It corresponds to a synthetic problem that assigns three tasks involving equal amounts of work to each processor. Figures 8.11 and 8.12 show the execution times of each processor when using the static and the dynamic approaches onto the worst case problem on the iPSC/860 and the CM5. In this case, the strategy used for load balancing is sender initiated and the upper bound for the queue length is set to two. Thus, any processor having more than two tasks in its local queue tries to send some work to another processor. This communication introduces considerable overhead as all the processors have more than two tasks in their queues. Fortunately, the user can reduce the frequency of the load balancing calls. In this particular case, an upper bound greater than three reduces this frequency to none (because processors always have three tasks or fewer in their queues), yielding a total execution time identical to the static case.

This last experiment illustrates the importance of using the specific knowledge to successfully apply the library (see Chapter 7.) In particular, it shows the importance of understanding the problem types so that the appropriate approach can be chosen. If the problem is regular, the static approach is appropriate. If the problem is irregular, the dynamic approach outperforms the static one. If the problem is regular and the dynamic approach is used by mistake, the performance can be improved by reducing the frequency of the load balancing calls. Therefore, to achieve good efficiency it is important (though not essential) to use the specific knowledge as well as the knowledge acquired from experience. The former is obtained by reading the documentation and by analyzing the characteristics of the problem and the machine. The latter is obtained by trying different strategies and interpreting the results.

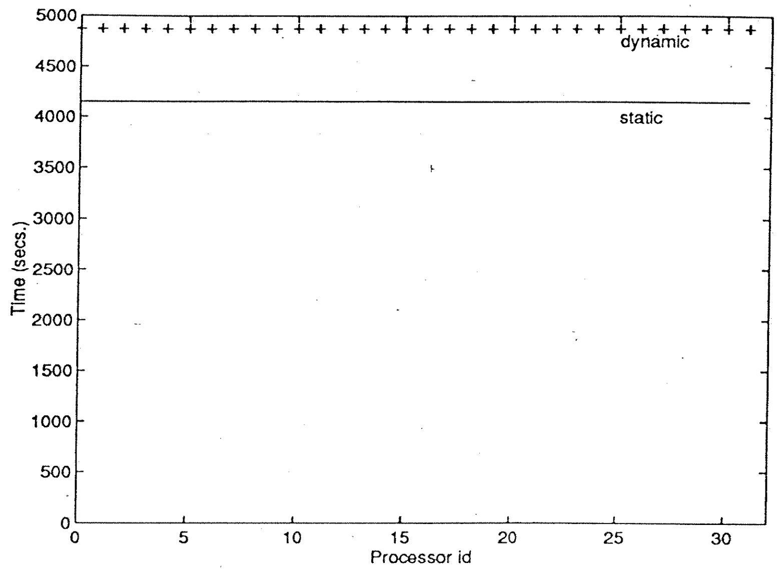


Figure 8.11. Execution time for each of the 32 processors under the static and dynamic approaches to solve the synthetic problem (on the iPSC/860.)

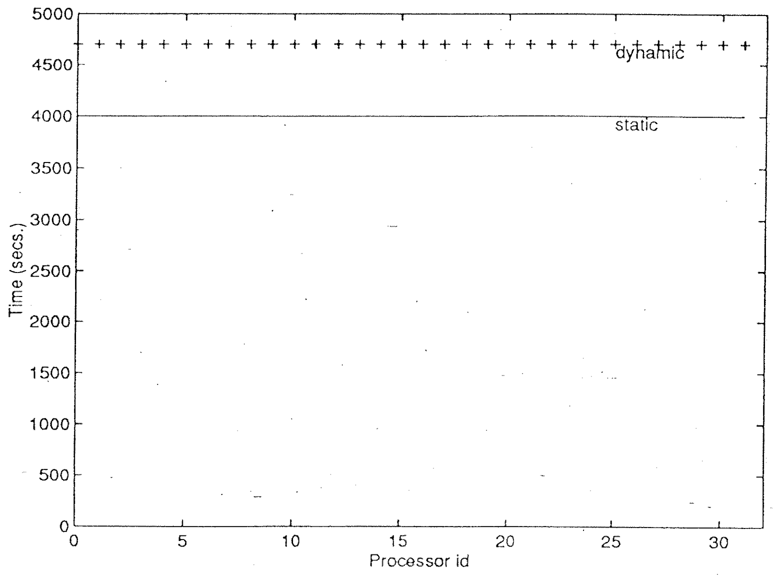


Figure 8.12. Execution time for each of the 32 processors under the static and dynamic approaches to solve the synthetic problem (on the CM5.)

Because experience seems to be an important ingredient here, we now compare the efficiencies obtained by naive and experienced users.

8.2 Efficiency Achieved the First Time (on an Irregular Problem)

As discussed in Chapter 4, there is no single strategy for load balancing, termination checking or updating of information that can be optimal for all the problems and on all the machines. The PMESC approach to coping with this problem consists of providing a set of reasonably easy-to-use and easy-to-change strategies so that the user can decide the one that is the most suitable for the particular application in mind. Some decisions can be made a priori by studying the problem and its characteristics as well as the strategies provided to support them. However, some other decisions can only be made after running the program and analyzing the results. Thus, efficiency is not an attribute that can be guaranteed the first time the user runs an application with PMESC, especially if the application is irregular.

Among the decisions that can be made a priori is what kind of approach to use: static or dynamic. If the static approach is chosen, there are no more decisions to make as load balancing, termination checking, and information sharing are not part of the problem. If, on the other hand, the dynamic approach is chosen, there are other decisions ahead. For instance, we know that ring is a better strategy than random for load balancing on a ring-connected network of workstations where the penalty of communicating distant processors is high. We also know that sender-initiated strategies for load balancing outperform receiver-initiated ones at light to moderate system loads and the opposite is true at high system loads. However, because the unpredictable nature of the dynamic problems it is usually impossible to be certain about all the issues before executing the problem the first time.

Therefore, in order for the experienced users to know whether they have achieved the most by using the library, or for the naive users to better understand the problem and improve its performance, they need to try different alternatives and compare results. Fortunately, this process does not require either a great deal of time or expertise. It basically consists of changing some parameters and routines. In this section, we discuss the efficiency that can be achieved with a first approach using the PMESC library. In the next section, we analyze what can be done to improve the first approach.

Suppose that we have to implement the parallel bisection example. As we prepare the main code, we have to make some decisions. To begin, we choose a distributed-queue approach and use the random, sender-initiated strategy for load balancing (which are the defaults.) To control the frequency of calls to the load balancing routine, we select parameters l_b and U_b, corresponding to the lower and upper bounds for the queue length, equal to zero and four respectively. Therefore, the main routine will invoke the load balancer every time the queue length is less than or equal to 0 or greater than four. Observe that in all of these selections, we have not assumed any knowledge about the problem's behavior. All we know is that it does not require sharing of information. Thus, the performance obtained with this approach can be compared to the one obtained by a first-time user.

Figures 8.13 and 8.14 show efficiencies that can be achieved by using PMESC. The efficiency is a performance measure defined as

$$\text{Efficiency} = \frac{\text{Speedup}}{p},$$

where p is the number of processors and Speedup is the ratio

$$\text{Speedup} = \frac{\text{execution time using 1 processor}}{\text{execution time using } p \text{ processors}}.$$

Figure 8.13 shows the efficiencies obtained as a result of applying our first approach to both matrices, $[1, 2, 1]$ and random, on the iPSC/860. Figure 8.14 shows the efficiencies achieved by running the same experiments on the CM5. Overall, the experiments show very encouraging results. Efficiencies range from 60% in the worst case to 99% in the best case. However, as we mentioned before, we have to try other alternatives to be sure that those efficiencies are the best possible. In the next section, we address this issue.

8.3 How Can We Improve the Efficiency?

In general, there are different aspects to look at when trying to improve the efficiency of a PMESC-based application when the dynamic approach is chosen. Among the aspects to consider are the frequency of load balancing calls (controlled by the queue lower and upper bounds), the load balancing strategies (controlled by the load balancing routines), the frequency of updating calls (controlled by the updating routine) and the granularity (controlled by the size of the tasks.)

In the next sections, we discuss each one of these aspects and present some examples. To illustrate our analysis, we present a series of diagrams that show the percentage of time that a program written with PMESC spends on activities other than the computation itself. This execution time breakdown allows us to determine the overhead incurred by using the library, the main sources for that overhead, and how it affects the efficiency of an application.

Diagrams 8.15, 8.16, 8.17, 8.18, 8.19, and 8.20 show the execution

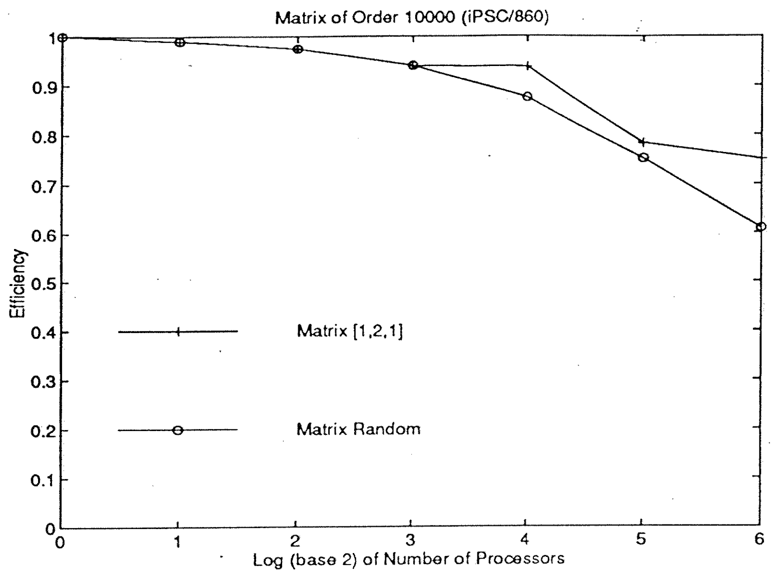


Figure 8.13. Efficiencies achieved with PMESC for the [1, 2, 1] and random matrices of order 10000 using the sender-initiated approach on the iPSC/860.

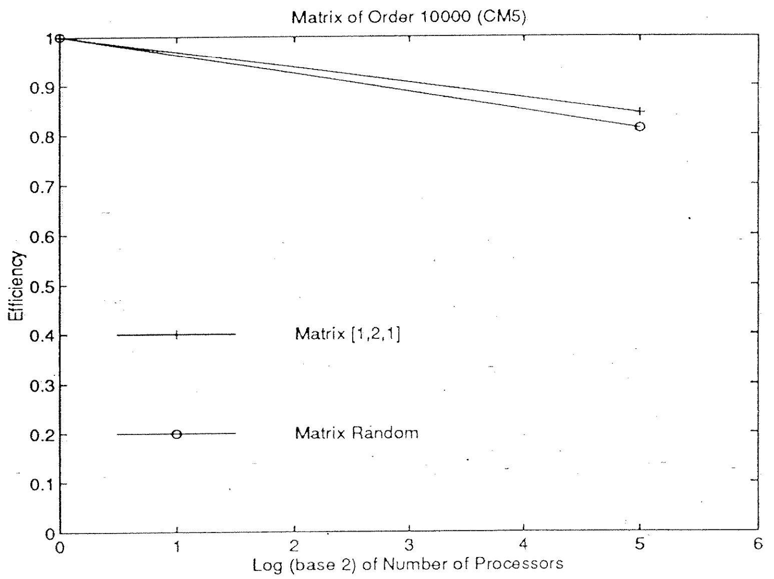


Figure 8.14. Efficiencies achieved with PMESC for the [1, 2, 1] and random matrices of order 10000 using the sender-initiated approach on the CM5.

times for dynamic bisection broken-down into four categories: **Computation**, **Map**, **Global Communication**, and **Idle**. These times represent averages over all the processors. Observe that these categories do not exactly match the PMESC phases. This is because we are not interested in measuring the time associated with each one of the PMESC phases. Rather, we want to analyze the significance of the overhead and idle time with respect to the computational time as well as the variations obtained in those times as we change strategies. Thus, Computation measures the percentage of time that processors spend on the execution of the tasks, i.e., the Solve phase of PMESC. In addition, it includes some PMESC overhead associated with fetching tasks from and storing tasks in the local memory, i.e., the time corresponding to the Partition phase. Most of the PMESC overhead is associated with the routines that involve message-passing. We separate this overhead into two categories, Map and Global Communication. Map represents the percentage of time that processors spend balancing the loads and checking for termination, i.e, the time associated with the Map module. Global Communication represents the time that processors spend either updating pseudo-global variables or doing some kind of global combine operation, i.e, the time associated with the high level Communicate module. Both categories, Map and Global Communication, include the time associated with low-level communication, i.e., the low-level Communicate phase, as well a the time for embedding whatever virtual topologies they may use into the machine architecture. The embedding algorithms are executed only once, i.e., one per virtual topology used, and only represent some small fraction of the mapping or communication time. For that reason we do not treat them separately. Finally, high overhead is not the only reason to get good or bad performance. It may also happen that a given strategy does not incur a great deal of overhead but does not perform an efficient distribution of the load either. In that case, the idle time is high, and the overall performance is then poor. Thus, we include another category, called Idle, that represents the percentage of the total execution time that processors spend doing nothing.

8.3.1 Load balancing.

Changing the load balancing strategy is an important feature of PMESC that may allow us to get substantial improvements in performance. To illustrate this point, Figures 8.15 and 8.16 show the execution time breakdown resulting from applying bisection to the random matrix of order 10,000 on the iPSC/860 and the CM5 respectively. A random, sender-initiated approach was selected for load balancing. Observe that, because we cannot use a subpartition on the CM5, the diagram shows the time breakdown for only two numbers of processors: 1 and 32. The diagram in Figure 8.15 shows an increasing Map overhead as the number of processors grows. This indicates a problem with the load balancing mechanisms. The actions to take in this case are either to change the load balancing procedure or to change the frequency of the calls to the load balancing procedure. In this case, the first alternative produces substantial gains. Figures 8.17 and 8.18 show the execution time breakdown corresponding to the random, receiver-initiated approach on the iPSC/860 and CM5 respectively. The diagrams show that the receiver-initiated approach dramatically reduces the overhead and increases the computation time. This is confirmed in Figure 8.21 which shows the efficiencies obtained on the iPSC/860, for the random matrix, with the sender- and receiver-initiated approaches. Figure 8.22 compares the efficiency achieved using both approaches on the CM5. Thus, in the random matrix case, we can substantially improve the efficiency by changing the load balancing routine from sender- to receiver-initiated.

However, the results obtained for the random matrix cannot be extrapolated to other cases. For instance, let us consider again the matrix $[1, 2, 1]$. Figures 8.19 and 8.20 show the execution time breakdown corresponding to this matrix when using a sender-initiated approach on the iPSC/860 and the CM5 respectively. The idle time in this example is not as high as in the random matrix, and the overhead is low. Still, we try the receiver-initiated alternative to see if we can improve the already good efficiency that we had obtained with the sender-initiated one. The results in this case, are not favorable. Figure 8.23 compares the efficiencies achieved in the $[1, 2, 1]$ case on the iPSC/860 using both approaches.

Observe that what happened with these examples is not surprising considering the theoretical results obtained by Eager and his colleagues [26]. In their paper, they conclude that receiver-initiated strategies outperform sender-initiated ones in heavily-loaded systems, while the opposite is true in moderately- to lightly-loaded systems. The $[1, 2, 1]$ case corresponds to the lightly-loaded system. Most eigenvalues are concentrated in a few processors while the rest are underloaded. On the other hand, the random case corresponds to a heavily-loaded system. Eigenvalues are more evenly distributed than in the $[1, 2, 1]$ case, and most processors are busy most of the time. It was justified then that the sender-initiated approach was more suitable for the $[1, 2, 1]$ matrix while the receiver-initiated performed better for the random matrix. Thus, if the user knows about the theoretical results of Eager et al. and knows the characteristics of the problem in hands, the user can make the

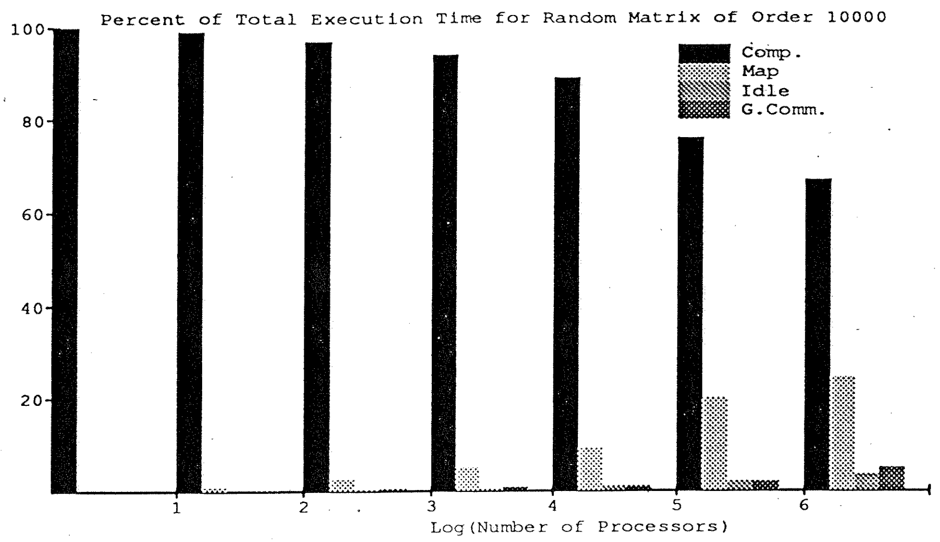


Figure 8.15. Execution time breakdown for the parallel bisection problem applied to the random matrix of order 10000 using a sender-initiated approach on the iPSC/860.

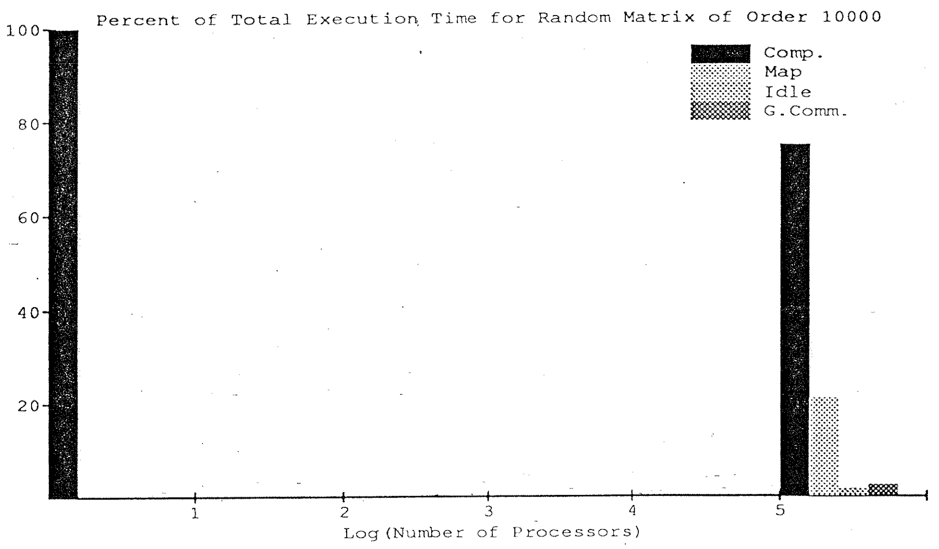


Figure 8.16. Execution time breakdown for the parallel bisection problem applied to the random matrix of order 10000 using a sender-initiated approach on the CM5.

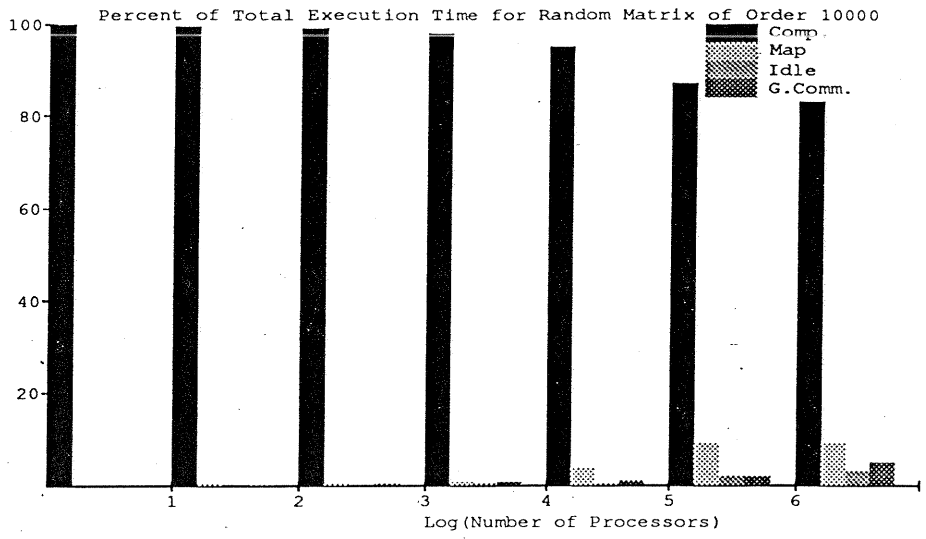


Figure 8.17. Execution time breakdown for the parallel bisection problem applied to the random matrix of order 10000 using a receiver-initiated approach on the iPSC/860.

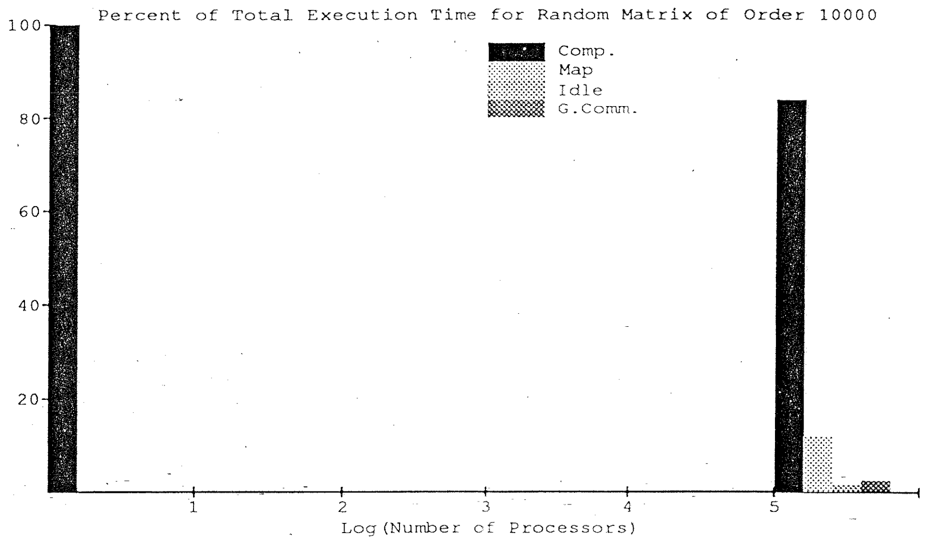


Figure 8.18. Execution time breakdown for the parallel bisection problem applied to the random matrix of order 10000 using a receiver-initiated approach on the CM5.

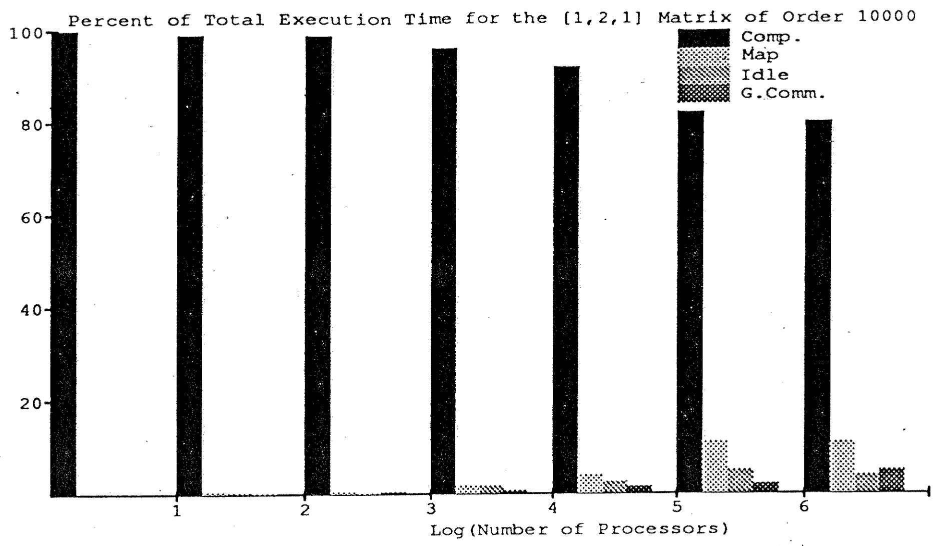


Figure 8.19. Execution time breakdown for the parallel bisection problem applied to the [1, 2, 1] matrix of order 10000 using a sender-initiated approach on the iPSC/860.

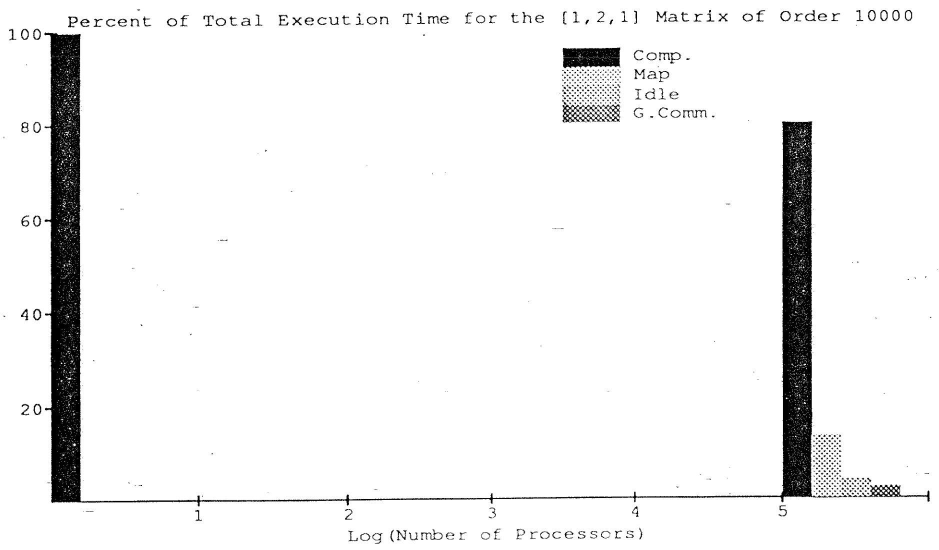


Figure 8.20. Execution time breakdown for the parallel bisection problem applied to the [1, 2, 1] matrix of order 10000 using a sender-initiated approach on the CM5.

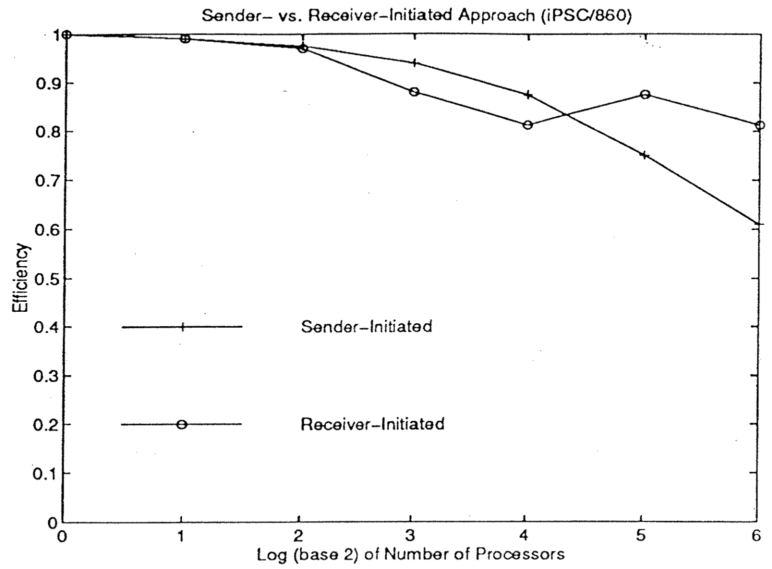


Figure 8.21. Efficiencies achieved with PMESC for the random matrix of order 10000 using the sender- and receiver-initiated approaches on the iPSC/860.

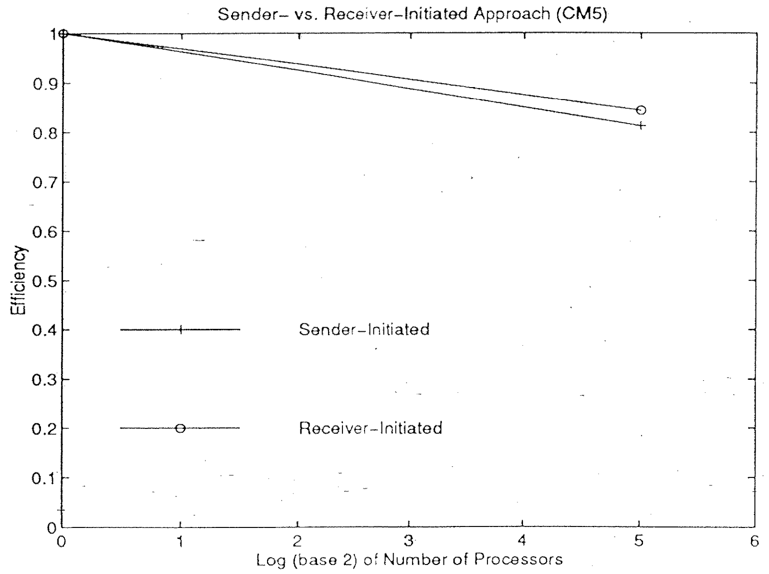


Figure 8.22. Efficiencies achieved with PMESC for the random matrix of order 10000 using the sender- and receiver-initiated approaches on the CM5.

right choice about which strategy to use for load balancing without having to try different ones. On the other hand, if the user does not know about the problem and the relevant results that can be applied to it, then the user has to try different alternatives and then decide based on her or his own results.

Now, because we have seen that conclusions made about one problem cannot be easily extended to other problems, we have reasons to be concerned with the problem size that we use to test the different alternatives. In fact, it is not wise to test the effect of using different routines and input parameters on a very large problem. It is advisable to try them on a smaller subproblem and study then if they hold as the size of the problem increases. Our experiences with the bisection problem indicate consistent results along different problem sizes. Figures 8.24, 8.25, 8.26, and 8.27 illustrate that the the conclusions about sender- vs receiver-initiated approaches remain the same across a wide range of matrix sizes even when the irregularity of the problem becomes more and more accentuated as the matrix size increases.

Finally, changing the frequency of the load balancing calls may also have an impact on the performance. We can easily control that frequency by changing the input parameters l_b and U_b that represent the lower and upper bounds of the queue length. To illustrate this point, we use the synthetic example presented in section 8.1. In that case, a value of the upper bound equal to two generated considerable overhead and caused a poor performance of the dynamic approach. However, this situation is improved by just changing the upper bound. Figure 8.28 shows the execution times for the synthetic example using the static and the dynamic approaches with $U\_b = 2$ and $U\_b = 3$. The upper bound $U\_b = 3$ reduces the overhead, putting the dynamic approach at

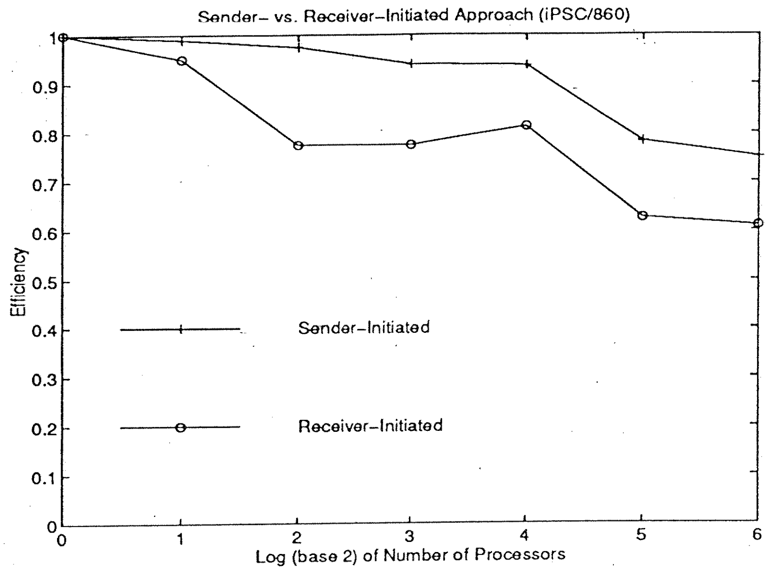


Figure 8.23. Efficiencies achieved for the [1, 2, 1] matrix of order 10,000 using the sender- and receiver-initiated approaches on the iPSC/860.

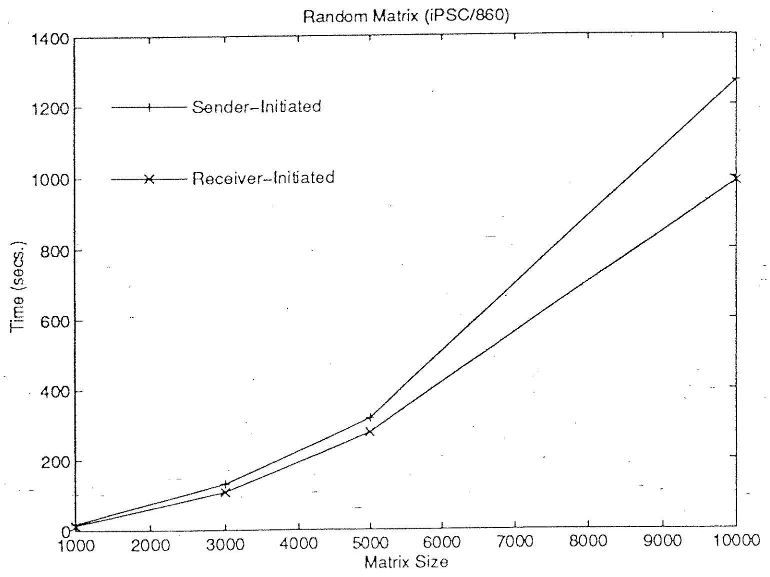


Figure 8.24. Timings for computing the bisection problem for the random matrix using the sender- and receiver-initiated approaches on the iPSC/860.

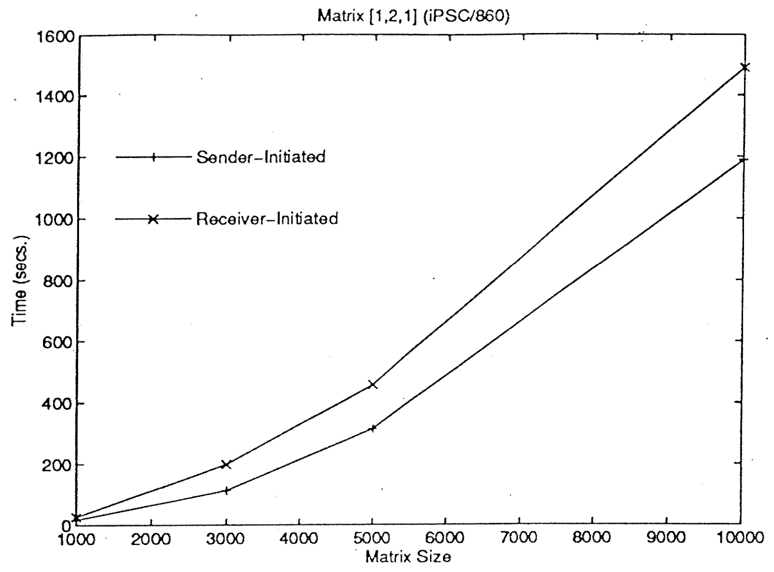


Figure 8.25. Timings for computing the bisection problem for the matrix [1, 2, 1] using the sender- and receiver-initiated approaches on the iPSC/860.

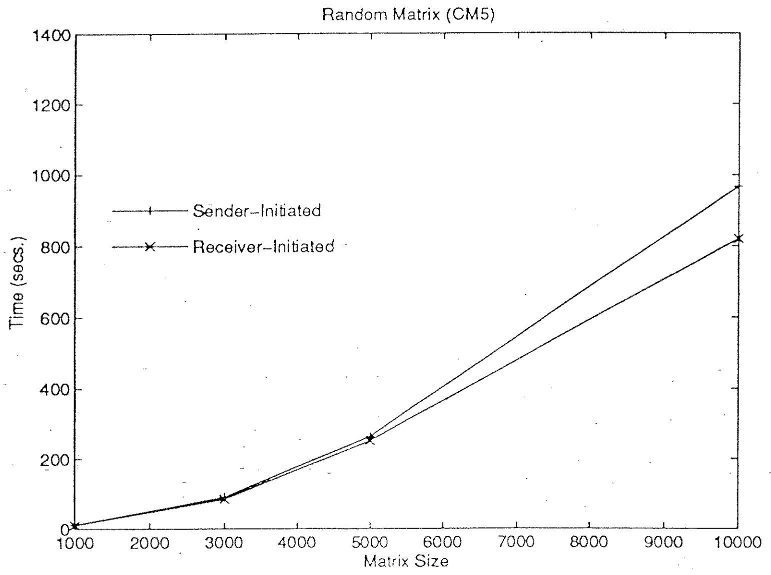


Figure 8.26. Timings for computing the bisection problem for the random matrix using the sender- and receiver-initiated approaches on the CM5.

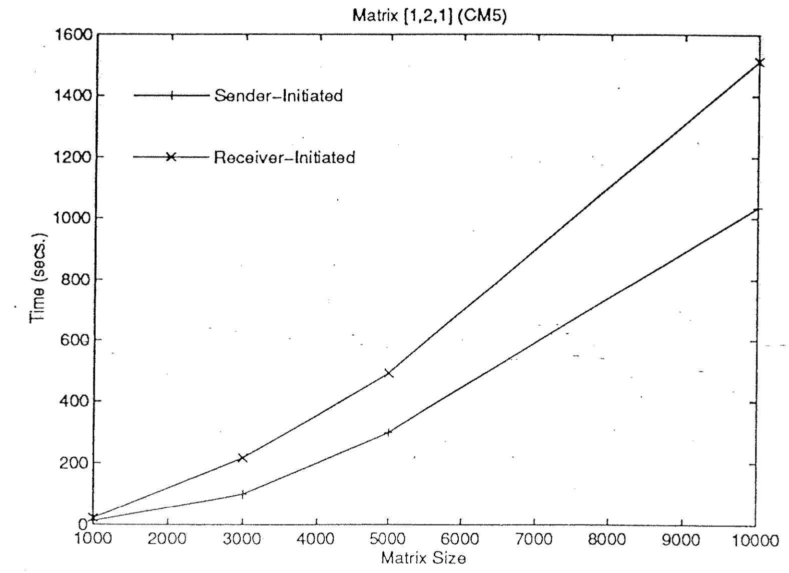


Figure 8.27. Timings for computing the bisection problem for the matrix [1, 2, 1] using the sender- and receiver-initiated approaches on the CM5.

the same level of the static one.

8.3.2 Checking for termination. The termination checking strategy used in PMESC adds very little overhead to the working processors. As explained in Chapter 4, the termination checking is started by idle processors, but it is absolutely ignored by busy ones. The latter just disregard the termination messages without even receiving them. (Every kind of message has associated a type. Processors can disregard messages after just determining their types.) Thus, the overhead is on the idle processors who send a termination message to their parents in the tree while waiting for extra work to arrive. For this reason, we do not consider the termination checking issue an important source of overhead that may deter at all the overall performance. Termination detection could be a problem if it interrupted the processors from what they are doing, but that is not the case in our implementation.

8.3.3 Updating. In the bisection example, the time that processors spend doing global communication corresponds to the gathering of results. This time is minimal compared to the time for load balancing. The situation repeats in the local adaptive quadrature case in which processors work independently most of the time. However, as was discussed in Chapter 6, there are some other problems in which the global communication time represents a more significant fraction of the total execution time. We now discuss this issue.

Sharing of information adds an extra source of overhead to parallel applications. An example that requires sharing is given by global adaptive quadrature. In this problem, the routine UPDATE (of the Communicate module) must be used to update the global error which becomes a pseudo-global

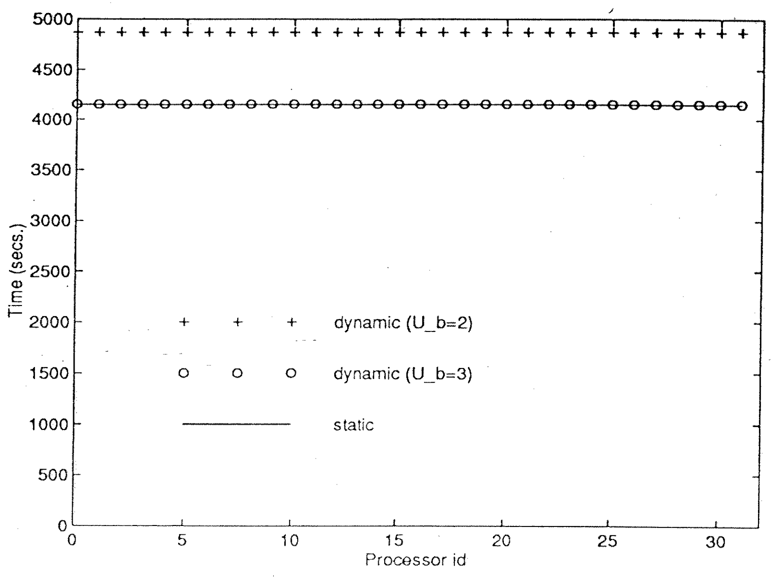


Figure 8.28. Execution time for each of the 32 processors under the static and dynamic approaches to solve the synthetic problem (on the iPSC/860.)

variable (refer to Chapter 6 for a description of this problem.) It is called pseudo-global for each processor has a copy of it which may not necessarily contain the most recent updates made by other processors. To keep the pseudo-global variable updated, processors exchange their copies from time to time. However, because the pseudo-global variable is used to avoid unproductive computation, the frequency of those exchanges becomes a critical ingredient. If the updatings are too close, the overhead becomes too high. On the other hand, if they are too infrequent, the time spent on unnecessary computation may become too high. The solution is application-dependent, and, as in other cases, it is left to the programmer's decision. In the global quadrature case, we follow the approach proposed in [21]. In this case, a processor invokes the routine that updates the global error only if the variation of the new local error with respect to the previous one is significant enough, i.e., greater than a given tolerance. This local tolerance is computed as the global error tolerance divided by the number of processors.

Figures 8.29 and 8.30 show the execution time breakdown corresponding to the local and global adaptive quadrature procedures run on the iPSC/860. The figures show a moderate increase of the global communication time in the global adaptive case. The increase is moderate because we use an efficient approach that avoids the unnecessary overhead caused by the global updatings when the local error is considered too small. Figure 8.31 shows the case when the updating routine is invoked every time the local error is changed. In this case, processors spend more time communicating than in the previous case, thus reducing the performance of the implementation.

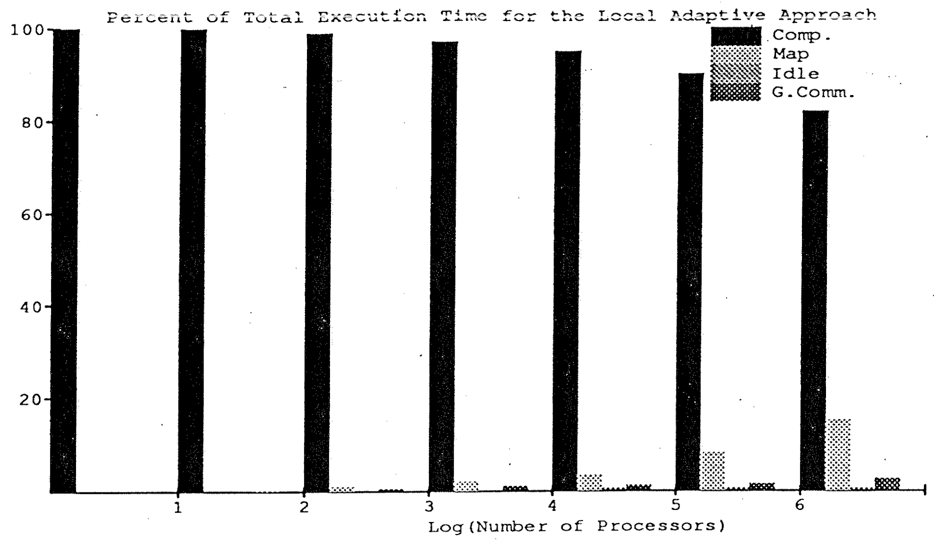


Figure 8.29. Execution time breakdown for the local adaptive quadrature procedure on the iPSC/860

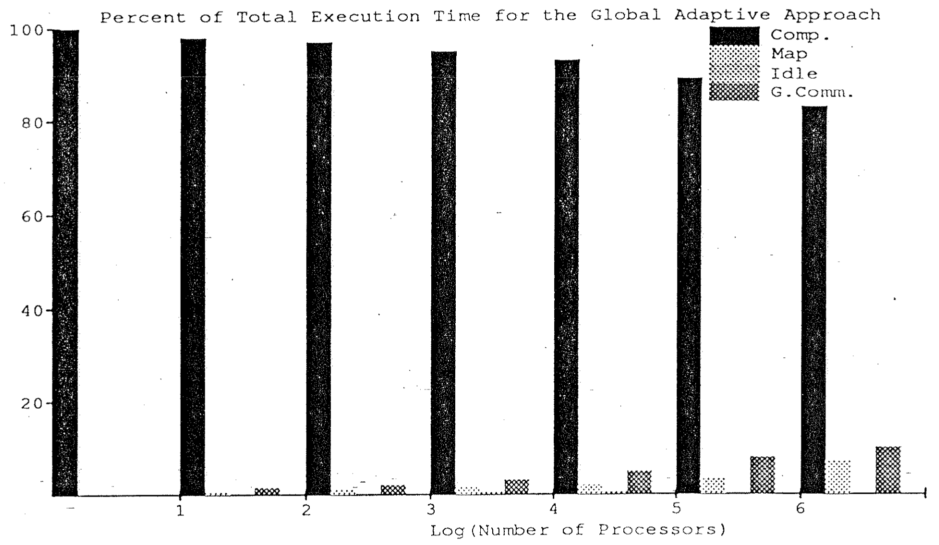


Figure 8.30. Execution time breakdown for the global adaptive quadrature procedure on the iPSC/860

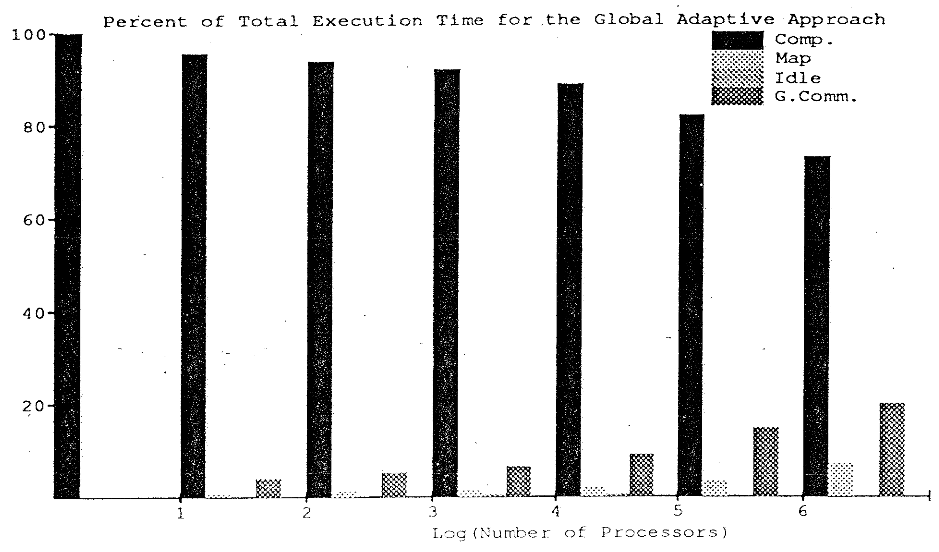


Figure 8.31. Execution time breakdown for the global adaptive quadrature procedure on the iPSC/860 for the case where the global error is updated every time the local error is changed.

8.3.4 Granularity. PMESC is not an library for fine-grained problems. Short-lived tasks cause unnecessary overhead and yield poor performances, and, consequently, they should be avoided. In Chapter 6, we discussed different ways to control the size of the tasks, but we did not specify which sizes should be considered "reasonable" and which should not. In this section we address this point by running a problem with different grain sizes and determining the efficiency achieved in each case. To this end, a convenient example to use is the N-queen problem for it allows us to change the grain size by just changing an input parameter. (Refer to Chapter 6 for more details on this example.)

Figure 8.32 shows the efficiency achieved by running the 10-queen problem using different grain sizes. According to the figure, a task that takes 2 msecs. to complete is too short to justify the use of PMESC. The efficiency improves for tasks that take 8 msecs. and it is even better for tasks that take 50 msecs. However, the performance deteriorates when the grain becomes too coarse. In this case, the system does not create enough tasks to keep all processors busy most of the time, and even if it does, it moves them more slowly as working processors need more time to respond to the requests.

8.4 Conclusions

In this chapter we present a suite of tests that evaluate the performance that can be obtained by using the PMESC library. This evaluation is composed of several parts. First, we compare the total execution times resulting from using the static and the dynamic approach with PMESC and illustrate the gains obtained with the dynamic approach when solving irregular problems. Then, we determine whether efficiency can be achieved and how to

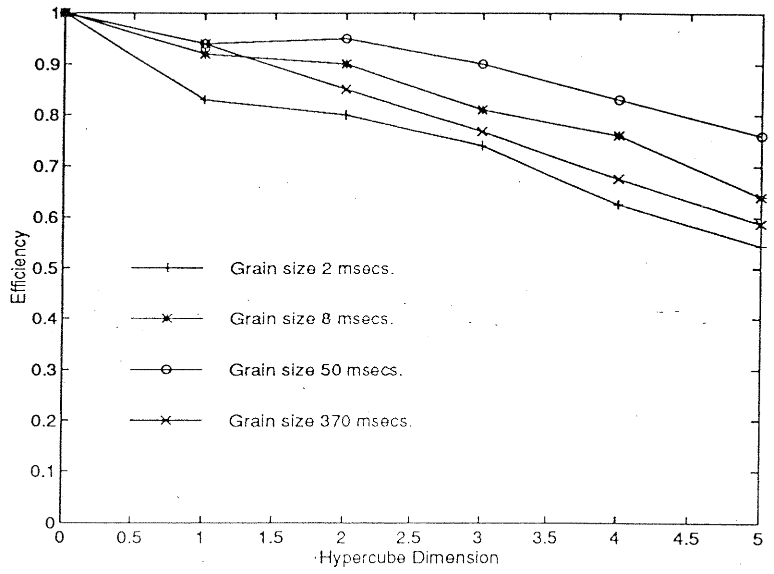


Figure 8.32: Efficiencies for the 10-queen problem varying the grain size.

achieve it. The dynamic approach supported by PMESC requires complex and costly procedures, and so its efficiency is not straightforward. Load balancing and information sharing constitute significant sources of overhead. We analyze their effect on the efficiency and how they can be changed to improve it.

The PMESC approach to obtaining efficiency is to change routines and parameters. Our tests show that efficiency can be achieved by all kinds of users. However, in order to perform a meaningful search of possibilities, these changes should be guided by analysis of previous and current results. In this regard, we emphasize the importance of using the knowledge about the problems and the strategies to solve them as much as possible. We realize that the specific knowledge discussed in Chapter 7 is important not only to make the library easy to use but also to achieve efficiency within reasonable time.

Our results are based on the examples and cannot be extended to all the problems. Although the examples cover different situations they do not represent all of them. Therefore, in order to make a more general statement about the performance of PMESC, it is necessary to study other problems, analyze their results, and eventually incorporate new techniques that might be necessary to improve their performance. For instance, problems whose tasks involve large amounts of data may need a load balancing routine that uses pipelining to do the transfers.

CHAPTER 9

TESTING THE LIBRARY: PMESC VS. CHARM

In this chapter, we complete the evaluation of PMESC by comparing it to a competitive package. As mentioned in Chapter 5, Charm, Express, and now PMESC are the only systems that address dynamic problems at a high level of abstraction. In that chapter, we compare and contrast the general characteristics of these approaches. In this chapter, we compare and contrast their flexibility and ease of use in the context of an example. Because Express is not available to us, we limit ourselves to discussing Charm vs. PMESC. Also, because there is no working version of Charm available at the time of this writing, we do not include experiments with it.

The chapter is organized as follows. In Section 9.1, we describe the structure of a Charm program and show the syntax that emphasizes its main components. Then, in Section 9.2, we analyze the main issues from the perspective of each tool. We discuss the way they address important programming aspects of dynamic problems such as load balancing, partitioning, and information sharing. Finally, in Section 9.3 we summarize the conclusions of this chapter.

9.1 Charm

Charm is a parallel programming system that supports a parallel C-based language. The main objectives of Charm are portability over the class of MIMD computers and efficiency. In Charm, the user is responsible for the creation of the units of work while the system decides when and where to execute them.

A Charm program consists of one or more modules. Each module is composed of a declaration section and a code section. The declaration section contains the definition of **messages** (similar to Tasks in PMESC) as well as C type definitions and declarations of shared variables. The code section contains the definitions of chares and C functions. Chares are procedures that are activated by Charm system calls. They interact via messages and the information sharing abstractions. Examples of chares are the **main** chare corresponding to the main procedure, and the **start** chare corresponding to the PMESC Solve procedure. A chare definition starts with the name of the chare, a declaration of local variables, and entry points definitions. In addition, it may also include private functions, which are like C functions except that they are allowed to access the local variables of the chare and can only be invoked within the chare. Each entry point definition includes the declaration of a message variable and a C-code block. The C-code block may contain Charm calls in addition to the usual C code. Figure 9.1 shows the syntax of a Charm program.

The execution of a Charm program begins at the **DataInit** entry point of the main chare or, in its absence, at the **CharmInit** entry point. The DataInit entry point is used to read input, create chares, and initialize shared variables. Parallel execution itself begins after this, at the ChareInit entry point. The **QUIESCENCE** entry point is optional, and it is executed when the system becomes idle. At runtime, many chares may be active. New chares are created using the **CreateChare** system call. Chares may send messages to other chares by using the **SendMsg** calls. The system maintains a "work-pool" consisting of new chares and messages for existing chares. The system may select multiple items from the queue and schedule them for execution in a nondeterministic way. Executing a message for an existing chare involves retrieving its data area and executing the code corresponding to the entry point specified in the SendMsg call. Executing a new chare involves allocating the data area for that chare, and beginning execution at the entry point specified in the CreateChare call.

9.2 The main issues under both tools

Let us compare both tools by observing the way they accomplish the main issues of partitioning, load balancing, information sharing, and gathering of results on the TSP code. Please refer to Chapter 6 for a description of this example.

9.2.1 Partition. Both, PMESC and Charm, allow users to partition the work by defining the units of work and, thus, the granularity of the problem. Both systems use similar definitions of the units of work.

- **Charm:** In Charm, the user defines a structure message as shown in Figure 9.2.
- **PMESC:** In PMESC, the user defines a structure Task like the one illustrated in Figure 9.3.

Thus, partitioning is done in the same way in both cases.

9.2.2 Load balancing. PMESC and Charm provide a variety of load balancing strategies. Charm allows the user to decide which to use but hides all the details regarding the execution of the procedure itself. The user does not know when the procedure is invoked or how frequently it is invoked

```
module Module1 {

        Type declarations
        Message definitions, information sharing declarations

        chare main {
                Local variable declarations

                [entry DataInit: C-code-block]
                entry ChareInit: C-code-block
                [entry QUIESCENCE: C-code-block]
                Other Entry points, functions and their code
        }

        chare Example {
                Local variable declarations

                \* Entry point definitions *\

                entry EP1: (message MESSAGE_TYPE1 *msgPtr)
                        C-code-block
                ..
                entry EPn: (message MESSAGE_TYPEn *msgPtr)
                        C-code-block

                \* Local function definitions *\

                private Function1(..)
                        C-code-block
                ..
                private Functionm(..)
                        C-code-block
        }

        BranchOffice Chares and other function declarations

}
```

Figure 9.1: Syntax of a Charm program. Source [23]

```
# define MAX \* Maximum number of cities *\
        message {
        int cost;
        unsigned int isDefined[MAX];
        unsigned int in[MAX];
        }
```

Figure 9.2: Partition in Charm

```
# define MAX \* Maximum number of cities *\
        struct task {
                unsigned int isDefined[MAX];
                unsigned int in[MAX];
                int cost;
        }
        typedef struct task Task;
```

Figure 9.3: Partition in PMESC

or what information is used to invoke it and has no way to change any of these issues without actually modifying the system itself.

The PMESC approach allows more user participation. It provides the routines MAP_xx that implement different load balancing strategies. The user program calls MAP_xx whenever it is necessary to transfer work. To determine when this is necessary, the user is free to use any measurement of the system load. One approach is based on the queue length. The program checks that the length of the local queue is between the user-defined lower and upper bounds. If the queue length is less than the lower bound the processor is lightly loaded or idle. In that case, the program may call MAP_xx(IDLE,other parameters) to get more work or, if no work is found, to check for termination. If the queue length is greater than the upper bound the processor is heavily loaded. In that case, the program may call MAP_xx(BUSY, other parameters).

Let us assume that we want to use a random load balancing strategy. We next describe how to proceed in both systems.

- **Charm:** In Charm, the creation of an executable parallel program requires translating the Charm program to produce a C file, compiling that file to produce an object file and then linking it with the necessary components of the run-time system. Among these components is the load balancing routine. The user can easily select this routine by copying its name into the makefile that comes along with Charm. Figure 9.4 shows the makefile corresponding to using the random strategy. Because the user does not invoke the load balancer, she or he cannot control the time or the the frequency of the load balancing calls. This is entirely up to the system.

```
#Choose queueing strategy
#queue=stack
#queue can be one of the following
#        stack, fifo, fl, bfifo, ififo, bstack, istack

#Choose load balancing strategy
#balance=rand
#balance can be one of the following
#          acwn, rand, tok

normal: pgm.o
        charmc -balance rand -queue stack pgm.o -o pgm -lm

pgm.o:  pgm.p
        charmc -c pgm.p
```

Figure 9.4: A Charm makefile

- **PMESC:** To implement a random strategy, the user must invoke the MAP_Ra routine from the main code. Then comes the choice (not available in Charm) of whether the strategy is sender- or receiver-initiated. Let us assume a random, sender-initiated load balancing routine, i.e., the default case. In that case, the main code part corresponding to load balancing is as illustrated in Figure 9.5.

  The user can also control the frequency of the load balancing calls by just changing the parameters corresponding to the lower and upper bounds of the queue_length. By controlling the frequency of these calls, the user can also control the overhead incurred by the load balancing procedure. Less frequent calls decrease the overhead at the cost of increasing the load imbalance. The user can play with these parameters and see how they affect the overall performance. For instance, the user can lessen the frequency of MAP_xx calls by just increasing the value of *U_b*. Chapter 7 presents an experiment that illustrates the importance of this feature to achieving good efficiency.

9.2.3 Information sharing. In the TSP example, there is a pseudo-global variable called **bound** that keeps track of the solution of least cost found so far. It is used to prevent processors from working on unproductive branches of the tree. Both systems provide facilities for the implementation of pseudo-global variables. However, the mechanisms they use to update these variables are different.

- **Charm:** Charm provides specific abstract data types for information sharing between chares. For branch-and-bound computations like TSP,

```
main()
{
        Task I;

        for (;;) {
            queue_length = DEQUEUE (&I);

            \* Main loop *\

            \*********************************************** \
            \* The user can decide what strategy *\
            \* to use for load balancing, whether it is *\
            \* sender- or receiver-initiated, and *\
            \* when to invoke it *\
            \************************************************ \

            \* Processor heavily loaded *\
            if (queue_length > U_b) \* Decides when to call the load balancer
            {
              \* Decides which strategy to use *\
              MAP_Ra (SENDER, BUSY, my_node, work_nodes)
              TSP (I);
              queue_length = DEQUEUE (&I);
            }

            \* Processor lightly loaded *\
            else if (queue_length <= L_b) \* Decides when to call the load balancer
            {
              if (MAP_Ra (SENDER, IDLE, my_node, work_nodes) == -1)
                          break;
            }
        } \* end infinity loop *\
}
```

Figure 9.5: Load balancing in PMESC

Charm provides **monotonic variables** (their values increase monotonically.) A monotonic abstract data type has associated with it a message containing the data area of the monotonic data type, an initialization function (that is used to set up the message) and an update operator. A variable of monotonic data type is created using the **CreateMono** call. Figure 9.6 shows how a monotonic data type is declared in the TSP example. The initialization function is **initf** and the update operator **updatefn**. The name of the functions, initfn and updatefn, are keywords and the scope of the functions is inside the block defined by the declaration of the monotonic variable.
The monotonic variable is created and initialized in the CharelInit entry point of the main chare as shown in Figure 9.7. Figure 9.8 shows how the monotonic variable is updated in the entry RECEIVE of the main chare. The mechanism that performs the updates is centralized. It uses a virtual tree of processors. The upward flow is for each processor to send the monotonic variable to its parent. The downward flow is for the root to broadcast its updated value to all processors [81].

- **PMESC:** PMESC updates the variable **bound** by explicitly calling the routine UPDATE as shown in Figure 9.9. The main code that runs on every processor may invoke UPDATE either when it detects an updating message from other processor or when it changes its own value of the pseudo-global variable. Thus, PMESC requires 1 line of code to declare a pseudo-global variable and another line of code to update it. Charm requires 26 lines of code to declare and create a pseudo-global variable, and 6 lines of code to update it.

```
\* This code is to declare a monotonic data type *\
message {
        int bound;
}MONO_MSG;
monotonic {
        MONO_MSG *msg;

        \************************************\
        \* This is initfn **********\
        \************************************\

        MONO_MSG *initfn(data)
        MONO_MSG *data;
        {
          msg = (MONO_MSG *)CkAllocMsg(MONO_MSG);
          msg->bound = data->bound;
          return(msg);
        }

        \************************************\
        \* This is updatefn **********\
        \************************************\

        updatefn(new)
        MONO_MSG *new;
        {
          if (msg->bound > new->bound)
          {
            msg->bound = new->bound;
            return(1);
          }
          return(0);
        }
} MONO_INT;
```

Figure 9.6: Declaration of shared variables in Charm

```
readonly int mono_bound;

chare main {

        entry CharmInit:
        {
                MONO_MSG *mono_msg;

        [Stuff deleted]

        \* Create pseudo-global variable *\
        \******************************************************* \
        mono_msg = (MONO_MSG *)CkAllocMsg(MONO_MSG);
        mono_bound = CreateMono(MONO_INT, mono_msg);
        ReadInit(mono_bound);
        \******************************************************* \

        [Stuff deleted]
}
```

Figure 9.7: Creating a pseudo-global variable in Charm

```
chare main {

        entry RECEIVE : (message SolnMsg *msg)
        {
        MONO_MSG *tmsg;

        if (msg->cost <= ((MONO_MSG *) MonoValue(mono_bound))->bound)
        {
        \* Update the pseudo-global variable *\
        \******************************************************* \
          tmsg = (MONO_MSG *)CkAllocMsg(MONO_MSG);
          tmsg->bound = msg->cost;
          NewValue(mono_bound,updatefn(tmsg));
        \******************************************************* \
        }

        [Stuff deleted]
}
```

Figure 9.8: Updating a pseudo-global variable in Charm

PMESC provides centralized and distributed mechanisms for updating pseudo-global variables. These are described in Chapter 4.

9.2.4 Gathering of results.

Usually, at the end of the computation, it is necessary to accumulate partial results. In the TSP case, for instance, one may want to count the total number of nodes that have been searched. Both systems provide different mechanisms to support this feature.

- **Charm:** In Charm, gathering or accumulation of information, is accomplished through the definition of an **accumulator** abstract data type. This type has associated with it a message containing the data area of the accumulator data type, an initialization function (that is used to set up the message) and two user-defined operators. The initialization function, **initfn**, is called on every node upon creation of the accumulator variable using the **CreateAcc** call. The first operator, **addfn**, adds to the accumulator variable in some user defined fashion. The second operator, **combinefn**, takes two accumulator variables as operands and combines those variables element by element.

  Figure 9.10 shows how the accumulator variable is declared in the TSP example. Also, Figure 9.11 shows how the accumulator variable is created and initialized in the CharelInit entry point of the main chare. Also, in the procedure branch_out (the equivalent to the Solve phase in PMESC), the accumulator is incremented every time a new node is created. Figure 9.12 shows how the accumulator is incremented.

  Finally, the code shown in Figure 9.13 shows how the number of nodes is summed up.

```
int bound; \* Declare pseudo-global variable *\

main ()
{
        int my_node, work_nodes;
        int cost;
        Task T;
        [Stuff deleted]

        \***************************************************** \
        \* MAP detects an updating message *\
        \****************************************************** \

        \* Updates pseudo-global variable *\
        if (MAP_xx(parameters) == 1)
          UPDATE(my_node, work_nodes, &bound, bound);

        [Stuff deleted]

        \***************************************************** \
        \* A new solution is found with cost less than bound *\
        \****************************************************** \

        \* Updates pseudo-global variable *\
        if (cost < bound)
          UPDATE (my_node, work_nodes, &bound, cost);

        [Stuff deleted]
}
```

Figure 9.9: Updating pseudo-global variables in PMESC

```
\* This code is to declare an accumulator data type *\
message {
        int nodes;
} ACC_MSG;

accumulator {

        ACC_MSG *msg;

        \*****************************************\
        \* This is the initialization function *\
        \****************************************\

        ACC_MSG *initfn(data)
        ACC_MSG *data;
        {
          msg = (ACC_MSG *)CkAllocMsg(ACC_MSG);
          msg->nodes = data->nodes;
          return(msg);
        }

        \*****************************************\
        \* This is the increment function *\
        \*****************************************\

        addfn()
        {
          (msg->nodes)++;
        }

        \********************************************\
        \* This is the combine function *\
        \********************************************\

        combinefn(b)
        ACC_MSG *b;
        {
          (msg->nodes) += (b->nodes);
        }

} ACC_INT;
```

Figure 9.10: Declaration of an accumulator variable in Charm

```
readonly int acc_nodes;

chare main {

        entry CharmInit:
        {
                ACC_MSG *acc_msg;

                [Stuff deleted]

        \* Create accumulator variable *\
        \******************************************************\
                acc_msg = (ACC_MSG *)CkAllocMsg(ACC_MSG);
                acc_msg->nodes = 0;

                acc_nodes = CreateAcc(ACC_INT, acc_msg);
                ReadInit(acc_nodes);
        \******************************************************\

                [Stuff deleted]
        }
}
```

Figure 9.11: Creating an accumulator variable in Charm.

```
branch_out(msg, A, n, cost, rows, columns)
{
  [Stuff deleted]

  CreateChare(start, startLEAF, msg1);
  \* Increment accumulator variable *\
  Accumulate (acc_nodes, addfn());

  [Stuff deleted]
}
```

Figure 9.12: Incrementing an accumulator variable in Charm.

```
chare main {
        entry QUIESCENCE:
        {
        \* Gather accumulator variables *\
          CollectValue(ReadValue(acc_nodes), MAINEP1, &chareid);
        }
}
```

Figure 9.13: Global sum in Charm.

- **PMESC:** To compute the number of tasks created by all the processors using PMESC, the user must define a variable that accumulates the partial sum in each processor. The main procedure that runs on every processor invokes the routine GATHER that sums the contents of these variables, **acc_nodes**, as shown in Figure 9.14. The variable **acc_nodes** is incremented by the routine branch_out every time a new task is generated (as in the Charm case.) Thus, PMESC requires 1 line of code to declare the accumulator variable, 1 line to increment it, and another line to sum the local values. Charm requires 26 lines of code to declare and create the accumulator variable, 1 line of code to increment it, and another line to collect all the values.

9.2.5 Size of the interface. A final comparison takes into account the number of lines of code related to each system. For instance, the number of Charm-related lines included in the TSP code is 117 which represents 22% of the total user code. Sixty of those 117 lines, i.e., 11% of the total code, are Charm calls, and 18 of those calls are different. In contrast, the number of PMESC calls included in the TSP code is 18 which represents 3% of the total user code. Only 6 of those calls are different.

9.3 Conclusions

PMESC and Charm have two goals in common: efficiency and portability. However, in this thesis we emphasize the importance of a third goal: ease of use. Therefore, we devote this chapter to comparing PMESC and Charm from the perspective of their flexibility and ease of use. We do that by analyzing the mechanisms used by both systems to attack some of the programming issues. We contrast the number of lines of code used by both tools to update

```
\* Declare accumulator variable for counting tasks *\
int acc_nodes;

main()
{
    [Stuff deleted]

    \* Gather accumulator variables *\
    GATHER (ROOT, SUM, &acc_nodes);
}

branch_out(T,A,n,cost,rows,columns)
Task *T;
{
    [Stuff deleted]

    ENQUEUE(TYPEP1, TYPEP2, T1);
    \* Increment accumulator variable *\
    acc_nodes++;

    [Stuff deleted]
}
```

Figure 9.14: GATHER collects partial sums in every processor.

pseudo-global variables and to perform global combine operations. In both cases, we show that Charm requires more lines of code than PMESC does to perform the same feature. Furthermore, PMESC has a more mnemonic setup than Charm does. For instance, to update the pseudo-global variable **bound** with a new value **cost**, a PMESC-based code would use

```
UPDATE (my_node, work_nodes, &bound, cost);
```

while a Charm-based code would call

```
tmsg = (MONO_MSG *)CkAllocMsg(MONO_MSG);
tmsg->bound = msg->cost;
NewValue (mono_bound, updatefn(tmsg)).
```

Overall, the complete TSP codes presented in Appendices A and A show that the Charm program has a much more complicated structure than does the equivalent PMESC program.

We show that PMESC gives the user more control over the mechanisms for load balancing than Charm does. Both tools allow users to decide which strategies are more convenient for their applications. However, PMESC provides more freedom, allowing users to activate those mechanisms whenever they (not the system as in Charm) consider it necessary.

Finally, Charm is a more mature tool than PMESC is. It presents a set of associated tools for analyzing the performance of a Charm code as well as for obtaining useful information about the program. These tools are Dagger [23], which allows the specification of dependences between messages and subcomputations within a single process, and Projections [23, 87], which allows performance visualization.

CHAPTER 10

CONCLUSIONS AND FUTURE RESEARCH

This thesis investigates the implementation of task-parallel problems on distributed-memory MIMD computers. The main issues that we address (and also the main contributions) fall into the broad categories of programming abstractions and programming tools. In our discussion of abstractions, we emphasize the importance of finding common patterns of resolution to the understanding of the problems and their efficient implementation. In our discussion of libraries, we show that although they abound in the area of data-parallel problems they are scarce in the area of task-parallel ones. We also investigate the main failures of current tools from the perspective of what users need and want. We conclude that in order for a tool to be useful it has to address the problems of portability, efficiency, and ease of use. Portability is important because computer technology is evolving, and rewriting programs for different platforms is expensive and prone to error. Efficiency is important because the fundamental reason for using parallelism is speed. Finally, ease of use is important for most potential users are not computer experts and do not want to spend a lot of time mastering the tools.

10.1 Contributions

Our first contribution is the identification of the PMESC paradigm. Note that we do not claim to have invented the paradigm. Rather, we observe and abstract an idea that has been used by experienced users in the development of code. By establishing this idea clearly, we make it accessible to a wide audience of users eager for models that help them understand and learn parallel computation. We discuss the importance of using PMESC as a model to design and implement code that is modular, efficient, and portable.

We also present the PMESC library to provide support for the implementation of task-parallel problems using static and dynamic approaches. The PMESC library is a set of routines or building blocks that users can put together. It allows them to customize their codes and to have complete control over the application. The library is efficient and portable but is also easy to understand and use. It is portable because it is based on the PMESC paradigm that allows the separation of the problems from the machines. It is efficient because it presents a modular design that allows changing of the building blocks to adjust to the different requirements of the problems and to the different computers. It is easy to use because it was designed based on the applications and the user requirements. We use a suite of tests to evaluate these three aspects of the library. Not only are these evaluation techniques useful for assessing the usability of the library but they were also instrumental in improving the library and its documentation.

A final set of contributions arises as a direct consequence of developing a library. This process requires the research into and the implementation of a variety of strategies to address different programming issues. We evaluate several strategies and in some cases we propose new ones. Examples of our new work are the hybrid strategy for updating pseudo-global variables (based on the centralized approach proposed in [81]) and a dynamic load-balancing strategy for prioritized tasks (based on the Random strategy proposed in [27]). These strategies are evaluated in detail in [18].

10.2 Future Research

This thesis suggests several areas for future research. One area for future work is to test PMESC with a larger set of applications that involve much larger communication requirements or memory requirements than those considered in this thesis. One interesting application would be a large global optimization problem.

Another area for future work is to build PMESC on top of MPI. This would make the library automatically portable across all the platforms that support MPI. A point to address though is whether this change will affect the performance of the PMESC library.

Another area for future work is the development of helpful tools for debugging and analyzing performance. These tools should be able to provide recommendations for improving the performance.

Yet another interesting avenue of future research is in the area of dynamic load balancing. So far we have implemented distributed approaches, but we are also interested in hierarchical ones. The idea is to be able to scale up as well as down. The hierarchical approach could bring the simplicity of a centralized strategy to a large number of processors, organizing them in clusters at one level and then organizing the clusters at a second level. However, it could also adapt to a small number of processors by reducing the levels to one and the number of clusters to one.

A final area, perhaps the most interesting one, for future work would be the use of PMESC in the implementation of large scale problems combining the data- and task-parallel paradigms on heterogeneous networks of parallel computers. A data-parallel tool would be used to implement the subproblems that so require it on the different computers of the network. PMESC would be used to resolve the imbalance problem that may arise between computers.

BIBLIOGRAPHY

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. **The Design and Analysis of Computer Algorithms**. Addison-Wesley, Reading, Massachussets, 1974.

[2] G.S. Almasi and A. Gottlieb. **Highly Parallel Computing, Second Edition**. The Benjamin/Cummings Publishing Company, Inc., 1994.

[3] S.B. Baden. Dynamic load balancing of a vortex calculation running on multiprocessors. Technical Report LBL-22584, Mathematics Department, Lawrence Berkeley Laboratory, University of California, 1986.

[4] S.C. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. **SIAM J. Sci. Stat. Comp.**, 12(1):249–256, 1991.

[5] M. Barnett, D.G. Payne, R. van de Geijn, and J. Watts. Broadcasting on meshes with worm-hole routing. Submitted to Journal of Parallel and Distributed Computing, original version available as Technical Report number TR-91-38, Department of Computer Science, University of Texas at Austin, 1994.

[6] A. Beguelin, J. Dongarra, G.A. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM: Parallel Virtual Machine. Technical Report TM-11826, Oak Ridge National Laboratory, 1991.

[7] A. Beguelin, E. Seligman, and M. Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, School of Computer Science, Carnegie Mellon University, 1994.

[8] M. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. Technical Report 85-55, ICASE, NASA Langley Research Center, 1985.

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

[10] D.P. Bertsekas, C. Ozveren, G.D. Stamoulis, P. Tseng, and J.N. Tsitsiklis. Optimal communication algorithms for hypercubes. **J. Par. Dist. Comput.**, 11:263–275, 1991.

[11] S.H. Bokhari. On the mapping problem. **IEEE Trans. Computers**, C-30:207–214, 1981.

[12] R. Buttler and E. Lusk. User's guide to the P4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, Mathematics and Computer Science Division, 1992.

[13] B. Buzbee. Report from Trondheim: Trends and needs in supercomputing. **Int. J. Supercomput. Appli.**, 3(4):3–5, 1989.

[14] N. Carriero and D. Gelernter. Linda in context. **Communications of the ACM**, 32(4):444–458, April 1989.

[15] D.Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA Ames Research Center, 1993.

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

[17] C.R. Cook, C. Pancake, and R. Walpole. Are expectations for parallelism too high? A survey of potential parallel users. In **Proceedings of Supercomputing '94**, 1994.

[18] S. Crivelli and E.R. Jessup. Asynchronous mechanisms for implementing task-parallel problems on distributed-memory computers. Technical Report in preparation, Department of Computer Science, University of Colorado, Boulder, 1995.

[19] S. Crivelli and E.R. Jessup. A user's manual for the PMESC library. Technical Report in preparation, Department of Computer Science, University of Colorado, Boulder, 1995.

[20] E. de Doncker and A. Gupta. Distributed and adaptive integration: Algorithms and analysis. In M. Becker, L. Litzler, and M. Trehel, editors, **Proceedings of Transputers '94**, pages 266–277. IOS Press, 1994.

[21] E. de Doncker and J.A. Kapenga. Parallel systems and adaptive integration. **Parallel Computing, North-Holland**, 7:211–225, 1988.

[22] E. de Doncker and I. Vakalis. Convergence results and speedup of parallel numerical integration algorithms. In **Proceedings of the Sixth SIAM**

Conference on Parallel Processing for Scientific Computing, volume II, pages 539–545, 1993.

[23] Department of Computer Science, University of Illinois, Urbana-Champaign. **Charm 4.3 Programming Language Manual**, 1994.

[24] J. Dongarra and D. Sorensen. SCHEDULE: Tools for developing and analyzing parallel fortran programs. In L.H Jamieson, D.B. Gannon, and R.J. Douglass, editors, **The Characteristics of Parallel Algorithms**, pages 363–394. The MIT Press, Cambridge, Massachusetts, 1987.

[25] J.J. Dongarra. Performance of various computers using standard linear equation solvers. Technical Report CS-89-85, Oak Ridge National Laboratory, November, 1993.

[26] D.L. Eager, D.L. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. **Performance Evaluation**, 6(1):53–68, March 1986.

[27] D.L. Eager, D.L. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. **IEEE Transactions on Software Engineering**, SE-12(5):662–675, May 1986.

[28] P.H. Enslow. Multiprocessor organization - a survey. **Computing Surveys**, 9(1):103–129, 1977.

[29] D.G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. **Computer**, pages 65–77, May 1990.

[30] R. Finkel and U. Manber. DIB–a distributed implementation of backtracking. **ACM Transactions on Programming Languages and Systems**, 9(2):235–256, April 1987.

[31] I. Foster and S. Taylor. **Strand: New Concepts in Parallel Programming**. Prentice Hall, Englewood Cliffs, N.J., 1990.

[32] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In **Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**, pages 50–59. ACM SIGPLAN, 1990.

[33] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. PICL–a portable instrumented communication library. Technical Report ORNL/TM-11130, Mathematical Sciences Section, Oak Ridge National Laboratory, 1990.

[34] G.H. Golub and C.F. Van Loan. **Matrix Computations, Second Edition**. The Johns Hopkins University Press, Baltimore and London, 1989.

[35] A.Y. Grama and V. Kumar. Parallel processing of discrete optimization problems: A survey. Technical report, Department of Computer Science, University of Minnesota, 1992.

[36] A. Grimshaw. The MENTAT run-time system: Support for medium grain parallel computation. In **Proceedings of the Fifth Distributed Memory Computing Conference**, pages 1064–1073, 1990.

[37] W.D. Gropp and D.E. Keyes. Domain decomposition on parallel computers. Technical Report YALEU/DCS/RR-723, Yale University, 1989.

[38] W.D. Gropp and D.E. Keyes. Domain decomposition with local mesh refinement. Technical Report YALEU/DCS/RR-726, Yale University, 1989.

[39] W.D. Gropp and B. Smith. Chameleon parallel programming tools user's manual. Technical Report ANL-93/00, Argonne National Laboratory, 1993.

[40] D.C. Grunwald, B.A. Nazief, and D.A. Reed. Dynamic load distribution on point-to-point multicomputer networks. Technical Report CU-CS-542-91, Department of Computer Science, University of Colorado at Boulder, 1991.

[41] R. Hempel, H.-C Hoppe, U. Keller, and W. Krotz. Parmacs v6.0 specification, November 1993.

[42] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. 1993.

[43] C.-T. Ho and M.T. Raghunath. Efficient communication primitives on hypercubes. Technical Report RJ 7932 (72915), IBM Research Division, Yorktown Heights, 1987.

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

[45] Intel Corporation, Beaverton, Oregon. **Touchstone Delta System User's Guide**, 1991.

[46] Intel Supercomputer System Division, Beaverton, Oregon. **iPSC/2 and iPSC/860 User's Guide**, 1991.

[47] L. H. Jamieson. Characterizing parallel algorithms. In L.H Jamieson, D.B. Gannon, and R.J. Douglass, editors, **The Characteristics of Parallel Algorithms**, pages 65–100. The MIT Press, Cambridge, Massachusetts, 1987.

[48] V. Janakiram, E. Gehringer, D. Agrawal, and R. Mehrotra. A randomized parallel branch-and-bound algorithm. **International Journal of Parallel Programming**, 17(3):277–301, 1988.

[49] L.W. Johnson and R.D. Riess. **Numerical Analysis, Second Edition**. Addison-Wesley Publishing Company, 1982.

[50] L.V. Kale. A tutorial introduction to CHARM. Technical Report 92-6, Department of Computer Science, University of Illinois, 1992.

[51] J.A. Kapenga and E. de Doncker. A parallelization of adaptive task partitioning algorithms. **Parallel Computing**, 7:211–225, 1988.

[52] J. De 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**, pages 122–128, 1991.

[53] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. **The High Performance Fortran Handbook**. The MIT Press, 1994. also available via ftp as High Performance Fortran Language Specification, Rice University, Houston, Texas, Version 1.0 Draft, 1993.

[54] S.R. Kohn and S.B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. In **Proceedings of the 1994 Scalable High Performance Computing Conference**, Knoxville, Tenessee, 1994.

[55] S.R. Kohn and S.B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. Technical Report CS94-354, Department of Computer Science and Engineering, University of California at San Diego, 1994.

[56] V. Kumar, A.Y. Grama, and V.N. Rao. Scalable load balancing techniques for parallel computers. **Journal of Parallel and Distributed Computing**, 22(1):60–79, 1994. Also available as Technical Report 91-55 (November 1991), Department of Computer Science, University of Minnesota, Minneapolis, MN. Available via anonymous ftp from ftp.cs.umn.edu at users/kumar/lb-MIMD.ps.

[57] A.W. 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.

[58] F. Thomson Leighton. **Introduction to parallel algorithms and architectures: arrays-trees-hypercubes**. Morgan Kaufmann, 1992.

[59] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S. Yang, and R. Zak. The network architecture of the connection machine cm-5. Technical Report leiserson-9208, Thinking Machines Corporation, Cambridge, Massachusetts, 02142, 1992.

[60] C. Lewis. Using the thinking-aloud method in cognitive interface design. Technical Report IBM Research Report RC 9265, IBM, Yorktown Heights, NY., 1982.

[61] F.C. Lin and R.M. Keller. The gradient model load balancing method. **IEEE Software**, SE-13(1):32–38, 1987.

[62] J.D. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. **Operations Research**, 11:972–989, 1963.

[63] S. Manoharan and P. Thanisch. Assigning dependency graphs onto processor networks. **Parallel Computing**, 17:63–73, 1991.

[64] Message Passing Interface Forum. **Document for a Standard Message-Passing Interface**, 1993.

[65] P.A. Nelson. **Parallel Programming Paradigms**. PhD thesis, Department of Computer Science, University of Washington, 1987.

[66] P.A. Nelson and L. Snyder. Programming paradigms for nonshared memory parallel computers. In D. Gannon L.H. Jamieson and R.J. Douglass, editors, **The Characteristics of Parallel Algorithms**. MIT Press, 1987.

[67] C. Pancake. Where are we headed? **Communications of the ACM**, 34(11):53–64, 1991.

[68] C. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? **IEEE Computers**, 23(12):13–23, 1990.

[69] C. Pancake and C. Cook. What users need in parallel tool support: Survey results and analysis. In **Proceedings of the Scalable High-Performance Computing Conference**, pages 40–47. IEEE Computer Society Press, 1994.

[70] C.M. Pancake. A collaborative effort in parallel tool design. 1995.

[71] Parasoft Corporation, Pasadena, California. **Express C User's Guide, Version 3.0**, 1990.

[72] P.C. Pinkney. The visual browsing tool for astrophysical data management: a case study in scientific user interface design. Master's thesis, Department of Computer Science, University of Colorado at Boulder, 1994.

[73] G.G. Polson, C. Lewis, J. Rieman, and C. Wharton. Cognitive walkthroughs: A method for theory-based evaluation of user interfaces. **International Journal of Man-Machine Studies**, 36, 1992.

[74] V.N. Rao and V. Kumar. Parallel depth-first search, part I: Implementation. **International Journal of Parallel Programming**, 16(6):479–500, 1987.

[75] V.N. Rao and V. Kumar. Parallel depth-first search, part II: Analysis. **International Journal of Parallel Programming**, 16(6):501–519, 1987.

[76] J.R. Rice. Parallel algorithms for adaptive quadrature III. Program correctness. **ACM Transactions on Mathematical Software**, 2(1):1–30, 1976.

[77] M. Rosing and R. Schnabel. Efficient language constructs for large parallel programming – an overview of Dino 2. Technical Report CU-CS-578-92, Department of Computer Science, University of Colorado at Boulder, 1992.

[78] A. Ross and B. McMillin. Experimental comparison of bidding and drafting load sharing protocols. In **The Fifth Distributed Memory Computing Conference**, pages 968—974. IEEE, 1990.

[79] Y. Saad and M.H. Schultz. Data communication in hypercubes. **J. Par. Dist. Comput.**, 6:115–135, 1989. also available as technical report YALEU/DCS/RR-428, Department of Computer Science, University of Yale, October 1985.

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

[81] W. Shu and L.V. Kale. Chare-kernel—a runtime support system for parallel computations. **Journal of Parallel and Distributed Computing**, 11:198–211, 1991.

[82] W. Shu and L.V. Kale. A dynamic scheduling strategy for the chare-kernel system. In **Proceedings of Supercomputing '89**, pages 389–398, November 1989.

[83] H.D. Simon. Are highly parallel systems ready for prime time? **Int. J. Supercomput. Appli.**, 4, 1990.

[84] H.D. Simon. Partitioning of unstructured problems for parallel processing. **Computing Systems in Engineering**, 2:135–148, 1991.

[85] H.D. Simon, W.R. Van Dalsem, and L. Dagum. Parallel computational fluid dynamics: Current status and future requirements. Technical Report RNR-92-004, NASA Ames Research Center, 1992.

[86] A.B. Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In **Workshop on Dynamic Object Placement and Load Balancing**. ECOOP '92, Utrecht, The Netherlands, 1992.

[87] A.B. Sinha and L.V. Kale. Projections: A scalable performance tool. Submitted to Supercomputing '92, 1992.

[88] S.L. Smith. **Adaptive asynchronous parallel algorithms in distributed computation**. PhD thesis, Department of Computer Science, University of Colorado at Boulder, 1991.

[89] H.W.J.M. Trienekens. **Parallel Branch and Bound Algorithms**. PhD thesis, Department of Computer Science, Erasmus University, Rotterdam, 1990.

[90] B.W. Wah and Y.W. Eva Ma. MANIP—a multicomputer architecture for solving combinatorial extremum-search problems. **IEEE Transactions on Computers**, c-33(5):377–390, 1984.

[91] R. Weaver. **Supporting dynamic data structures at the language level on distributed memory machines**. PhD thesis, Dept. of Computer Science, University of Colorado at Boulder, 1992.

[92] R.P. Weaver and C. Lewis. Examining the usability of parallel language constructs from the programmer's perspective. Technical Report CU-CS-492-90, Department of Computer Science, University of Colorado at Boulder, 1990.

[93] R.D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. **Concurrency: Practice and Experience**, 3:457–481, 1991.

[94] C. Wright. Personal Communication, 1992.

[95] M.Y. Wu and D.D. Gajski. Hypertool: A programming aid for message passing systems. **IEEE Transactions on Parallel and Distributed Systems**, 1(3):330–343, 1990.

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

APPENDIX A

CHARM CODE FOR THE TSP

This is the Charm code for the TSP exactly as it comes with the Charm installation package (as one of the examples.)

```
module TSP {
#include <stdio.h>
#include "qd.h"

#define Max 40
#define MaxSquare (Max*Max)/30
#define INFINITY 999999         \* indicates to edge present *\
#define SMALL -999999           \* very small *\
#define RANDON_BOUND 200 \* limit on weight of edges *\
#define ROUTINE1 1
#define ROUTINE2 2
#define TRUE 1
#define FALSE 0

#define IsFree(a,ind) !( (a[(ind/32)]) & (1<<(ind % 32)) )
#define Set(a,ind) a[(ind/32)] = ( a[(ind/32)] | (1<<(ind % 32)) )
#define Reset(a,ind) a[(ind/32)] = ( a[(ind/32)] & ( (1<<(ind % 32))) )
```

```
#define DETERMINE_MATRIX determine_matrix0
#define BRANCH_OUT branch_out0
#define SOLVE_RELAXATION reduce_matrix

message {
        int cost;
        unsigned int isDefined[MaxSquare];
        unsigned int in[MaxSquare];
} MSG1;

message {
        int cost;
        int cycle[Max];
} SolnMsg;

readonly int N; \* number of rows and columns *\
readonly int matrix[Max][Max]; \* initial cost matrix *\

message {
        int nodes;
} ACC_MSG;

accumulator {

        ACC_MSG *msg;
```

```
\*****************************\
\ This is the initialization function *\
\*****************************\

ACC_MSG *initfn(data)
ACC_MSG *data;
{
  msg = (ACC_MSG *) CkAllocMsg(ACC_MSG);
  msg->nodes = data->nodes;
  return(msg);
}

\********************************************* \
\* This is the increment function for the accumulator *\
\* that will count the number of nodes in the tree *\
\********************************************* \

addfn()
{
  (msg->nodes)++;
}

\****************************************** \
\ This is the combine function for the accumulator *\
\******************************************* \

combinefn(b)
ACC_MSG *b;
{
  (msg->nodes) += (b->nodes);
}

ACC_INT; \* counts number of nodes *\


message {
  int bound ;
MONO_MSG;

monotonic {

  MONO_MSG *msg;

  \********************************\
  \* This is initfn *\
  \********************************\

  MONO_MSG *initfn(data)
  MONO_MSG *data;
  {
    msg = (MONO_MSG *)CkAllcMsg(MONO_MSG);
    msg->bound = data->bound ;
    return (msg);
  }

  \********************************\
  \* This is the comparison function *\
  \* for the monotonic variable *\
  \********************************\

  update(new)
  MONO_MSG *new;
  {
    if (msg->bound > new->bound)
    {
      msg->bound = new->bound;
      return(1);
    }
    return(0);
  }
} MONO_INT; \* the current bound *\


readonly int acc_nodes;
readonly int mono_bound;

chare main {

  int cost;
  int temp[Max][Max];
  int temp1[Max], temp2[Max];

  entry CharmInit;
  {
    int *x;
    int i, j;
    int index;
    MSG1 *msg;
    ACC_MSG *acc_msg;
    int choice, upper;
    MONO_MSG *mono_msg;

    CkScanf("%d", &N);
    CkPrintf("The number of cities are: %d\n", N);
    ReadInit(N);
    CkScanf("%d", &choice);
    CkScanf("%d", &upper);

    read_in_matrix(matrix, N);
```

```
cost = reduce_matrix(matrix, temp, N, temp1, temp2);

acc_msg = (ACC_MSG *)CkAllocMsg(ACC_MSG);
mono_msg = (MONO_MSG *)CkAllocMsg(MONO_MSG);

if (choice)
  mono_msg->bound = upper;
else
  mono_msg->bound = INFINITY;

acc_msg->nodes = 0;

mono_bound = CreateMono(MONO_INT, mono_msg);
ReadInit(mono_bound);
acc_nodes = CreateAcc(ACC_INT, acc_msg);
ReadInit(acc_nodes);
ReadInit(matrix);

msg = (MSG1 *)CkAllocPrioMsg(MSG1, sizeof(int));
x = (int *)CkPriorityPtr(msg);
*x = cost;
msg->cost = cost;
for (i = 0; i < N; i++)
  for (j=0; j<N; j++)
  {
        index = i*N + j;
        Reset(msg->isDefined, index);
        Reset(msg->in, index);
      }
    CreateChare(start, start@LEAF, msg);
    MyChareID(&chareid);
    StartQuiescence(Quiescence, &chareid);
}


entry QUIESCENCE: (message QUIESCENCE_MSG *dmsg)
{
    ChareIDType chareid;

    CkPrintf("Totalling number of nodes in the tree \n");
    MyChareID(&chareid);
    CollectValue(ReadValue(acc_nodes), MAINEP1, &chareid);
}


entry MAINEP1 : (message ACC_MSG *msg);
{
    CkPrintf("Total number of nodes in the tree = %d\n", msg->nodes);
    CkPrintf("———————————— \n");
    CkExit();
```

```
  }



  entry RECEIVE : (message SolnMsg *msg)
  {
    int i, j;
    MONO_MSG *tmsg;

    if (msg->cost <= ((MONO_MSG *)MonoValue (mono_bound))->bound)
    {
      tmsg = (MONO_MSG *) CkAllocMsg(MONO_MSG);
      tmsg->bound = msg->cost;
      NewValue(mono_bound, updatefn(tmsg));

      CkPrintf("New Tour of cost %d found at time %d :\n\n\t",
                                                   msg->cost, CkTimer());
      for (i=0; i<N; i++)
          CkPrintf("%2d\n", msg->cycle[i]);
      CkPrintf("%2d\n", msg->cycle[0]);
    }
    CkFreeMsg(msg);
  }
}
```

```
chare start {
  entry LEAF : (message MSG1 *msg)
  {
    int *x;
    int n;
    int cost;
    MONO_MSG *tmsg;
    int A[Max][Max];
    int soln[Max][Max];
    int rows[Max], columns[Max];

    n = ReadValue(N);
    DETERMINE_MATRIX(msg, ReadValue(matrix), A, n);
    cost = SOLVE_RELAXATION(A, soln, n, rows, columns);
    cost += msg->cost;
    if (cost < ((MONO_MSG *) MonoValue(mono_bound))->bound)
      if (isSolution(msg, soln, n))
      {
          tmsg = (MONO_MSG *) CkAllocMsg(MONO_MSG);
          tmsg->bound = cost;
          NewValue(mono_bound, updatefn(tmsg));
          printout_new_solution(soln, n);
      }
      else
      {
            BRANCH_OUT (msg, A, n, cost, rows, columns);
        }
      CkFreeMsg(msg);
      ChareExit();
  }
}


\****************************************** \
\* This function reads in the initial cost matrix *\
\****************************************** \


read_in_matrix(A, n)
int A[Max][Max];
int n;
{
  int i, j;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
    {
      CkScanf("%d", &A[i][j]);
    }
  for (i=0; i<n; i++)
    A[i][i] = INFINITY;
}
```

```
\******************************************* *******\
\* This function prints out a solution. Since the monotonic *\
\* variable does not guarantee the global best on every node *\
\* the solution found may not necessarily be the best *\
\******************************************* *******\

printout_new_solution(soln, n)
int soln[Max][Max];
int n;
{
  int *x;
  int i, j;
  int next;
  int index = 0;
  SolnMsg *msg;
  ChareIDType chareid;

  msg = (SolnMsg *) CkAllocPrioMsg(SolnMsg, sizeof(int));
  x = (int *)CkPriorityPtr(msg);
  *x = 0;
  msg->cost = ((MONO_MSG *) MonoValue(mono_bound))->bound;
  MainChareID(&chareid);
  for (i=0; i<n; i++)
    if (soln[0][i] == 1)
      break;
  msg->cycle[index++] = 0;
  next = i;
  while (next != 0)
  {
    msg->cycle[index++] = next;
    for (j=0; j<n; j++)
      if (soln[next][j])
      {
          next = j;
          break;
      }
  }
  SendMsg(main@RECEIVE, msg, &chareid);
}


\******************************************* *******\
\* This function reduces the cost matrix to a non-negative *\
\* matrix. We use this to fire off the computations, and also *\
\* to determine the first reasonable lower bound. *\
\******************************************* *******\


reduce_matrix(A, soln, n, rows, columns)
int A[Max][Max];
int soln[Max][Max];
int n;
int rows[Max], columns[Max];
{
  int i, j;
  int lowest;
  int cost = 0;

  for (i=0; i<n; i++)
  {
    lowest = INFINITY;
    for (j=0; j<n; j++)
      if (A[i][j] < lowest)
          lowest = A[i][j];

    if (lowest != INFINITY)
    {
      for (j=0; j<n; j++)
          if (A[i][j] != INFINITY)
             A[i][j] -= lowest;
      cost += lowest;
    }
    rows[i] = lowest;
  }
```

```
 for (j=0; j<n; j++)
 {
   lowest = INFINITY;
   for (i=0; i<n; i++)
     if (A[i][j] < lowest)
         lowest = A[i][j];

   if (lowest != INFINITY)
   {
     for (i=0; i<n; i++)
         if (A[i][j] != INFINITY)
           A[i][j] -= lowest;
     cost += lowest;
   }
   columns[j] = lowest;
 }
 for (i=0; i<n; i++)
   for (j=0; j<n; j++)
     soln[i][j] = 0;
 return (cost);
}
\************************************************ \
\* This function determines whether the solution to the *\
\* assignment problem is also a solution to the traveling *\
\* salesman problem. *\
\************************************************* \

isSolution(msg, soln, n)
MSG1 *msg;
int soln[Max][Max];
int n;
{
  int x, y;
  int i, j;
  int next;
  int index;
  int length = 0;
  int cycle_start, cycle_end;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
    {
      index = i*n + j;
      if (!IsFree(msg->in, index))
      {
         x = i;
         y = j;
         soln[i][j] = 1;
         length++;
      }
      }

    if (length != n-1)
      return(0);

    get_cycle_edge(msg, x, y, n, &cycle_start, &cycle_end);
    soln[cycle_end][cycle_start] = 1;
    return(1);
}



\****************************************** \
\* This function gets the sub-branch whose edge *\
\* may contribute to a cycle *\
\****************************************** \


get_cycle_edge(msg, v1, v2, n, cycle_start, cycle_end)
MSG1 *msg;
int v1, v2;
int n;
int *cycle_start;
int *cycle_end;
{
  int i;
  int index;
int end = v2;
int start = v1;
BOOLEAN done = FALSE;

while (!done)
{
  for (i=0; i<n; i++)
  {
    index = end*n + i;
    if (!IsFree(msg->in, index))
    {
        end = i;
        break;
    }
  }
  if (i==n)
    done = TRUE;
}

*cycle_end = end;
done = FALSE;

while (!done)
{
  for (i=0; i<n; i++)
```

```
      {
        index = i*n + start;
        if (!IsFree(msg->in, index))
        {
            start = i;
            break;
        }
      }
      if (i==n)
        done = TRUE;
    }
    *cycle_start = start;
  }



\************************************************** **\
\* This function returns the index of the least element in the *\
\* row *\
\************************************************** **\

least_in_row(A, row, n)
int A[Max][Max];
int row, n;
{
  int j;
  int least = INFINITY;

  for (j=0; j<n; j++)
    if (least > A[row][j])
      least = A[row][j];
  return(least);
}


\************************************************** **\
\* This function returns the index of the least element in the *\
\* column *\
\************************************************** **\

least_in_col(A, col, n)
int A[Max][Max];
int col, n;
{
  int i;
  int least = INFINITY;

  for (i=0; i<n; i++)
    if (least > A[i][col])
      least = A[i][col];
  return(least);
```

```
}


\************************************\
\* This function determines the best zero *\
\************************************\

find_best_zero(A, n, best_i, best_j)
int A[Max][Max];
int n;
int *best_i, *best_j;
{
 int temp;
 int store;
 int i, j, k;
 int best = SMALL;

 for (i=0; i<n; i++)
   for (j=0; j<n; j++)
   {
     temp = 0;
     if (A[i][j] == 0)
     {
         store = A[i][j];
         A[i][j] = INFINITY;
```

```
        temp = least_in_row(A, i, n);
        temp += least_in_col(A, j, n);
        A[i][j] = store;
        if (temp > best)
        {
          best = temp;
          *best_i = i;
          *best_j = j;
        }
      }
    }
  return(best);
}


\**************************************************** \
\* This is the trivial branching procedure. It determines *\
\* the best node in the relaxation problem and then branches *\
\* out in a binary fashion with one branch excluding the *\
\* edge and the other including the edge *\
\**************************************************** \

branch_out0(msg, A, n, cost, rows, columns)
MSG1 *msg;
int A[Max][Max];
int n;
int cost;
int rows[Max], columns[Max];
{
  int *x;
  int i, j;
  int temp;
  int best;
  int index;
  MSG1 *msg1, *msg2;
  int best_i, best_j;
  int cycle_start, cycle_end;
  int temp1[Max], temp2[Max], temp3[Max][Max];

  best_i = best_j = (-1);
  best = find_best_zero(A, n, &best_i, &best_j);
  if (best_i != -1)
  {
    msg1 = (MSG1 *)CkAllocPrioMsg(MSG1, sizeof(int));
    msg2 = (MSG1 *)CkAllocPrioMsg(MSG1, sizeof(int));
    msg1->cost = msg2->cost = msg->cost;

    for (i=0; i<n; i++)
      for (j=0; j<n; j++)
      {
        index = i*n + j;
        if (IsFree(msg->isDefined, index))
        {
          Reset(msg1->isDefined, index);
          Reset(msg2->isDefined, index);
        }
        else
        {
          Set(msg1->isDefined, index);
          Set(msg2->isDefined, index);
        }
        if (IsFree(msg->in, index))
        {
          Reset(msg1->in, index);
          Reset(msg2->in, index);
        }
        else
        {
          Set(msg1->in, index);
          Set(msg2->in, index);
        }
    }

index = best_i*n + best_j;
if ((cost < ((MONO_MSG *) MonoValue(mono_bound))->bound &&
   (best >= INFINITY)) ||
   (cost + best <
   ((MONO_MSG *) MonoValue(mono_bound))->bound))
  {
   Set(msg1->isDefined, index);
   Reset(msg1->in, index);

   A[best_i][best_j] = INFINITY;
   if (least_in_row(A, best_i, n) == INFINITY)
       if (rows[best_i] < INFINITY)
         msg1->cost += rows[best_i];
   if (least_in_col(A, best_j, n) == INFINITY)
       if (columns[best_j] < INFINITY)
         msg1->cost += columns[best_j];

   x = (int *)CkPriorityPtr(msg1);
   *x = cost + best;
   A[best_i][best_j] = 0;

   CreateChare(start, start@LEAF, msg1);
   Accumulate(acc_nodes, addfn());
  }

\* include best edge and exclude its symmetric counterpart *\
if (rows[best_i] < INFINITY)
```

```
  msg2->cost += rows[best_i];
if (columns[best_j] < INFINITY)
  msg2->cost += columns[best_j];
Set(msg2->isDefined, index);
Set(msg2->in, index);
index = best_j*n + best_i;
Set(msg2->isDefined, index);
Reset(msg2->in, index);

temp = A[best_j][best_i];
A[best_j][best_i] = INFINITY;
if (least_in_row(A, best_j, n) == INFINITY)
  if (rows[best_j] < INFINITY)
      msg2->cost += rows[best_j];
if (least_in_col(A, best_i, n) == INFINITY)
  if (columns[best_i] < INFINITY)
      msg2->cost += columns[best_i];
A[best_j][best_i] = temp;

\* Get the edge that completes the cycle and exclude it *\
get_cycle_edge(msg2, best_i, best_j, n, &cycle_start, &cycle_end);
index = cycle_end*n + cycle_start;
Set(msg2->isDefined, index);
Reset(msg2->in, index);
temp = A[cycle_end][cycle_start] = INFINITY;
```

```
      A[cycle_end][cycle_start] = INFINITY;
      if (least_in_row(A, cycle_end, n) == INFINITY)
        if (rows[cycle_end] < INFINITY)
            msg2->cost += rows[cycle_end];
      if (least_in_col(A, cycle_start, n) == INFINITY)
        if (columns[cycle_start] < INFINITY)
            msg2->cost += columns[cycle_start];
      A[cycle_end][cycle_start] = temp;

      A[best_j][best_i] = A[cycle_end][cycle_start] = INFINITY;

      for (i=0; i<n; i++)
        A[best_i][i] = A[i][best_j] = INFINITY;
      x = (int *)CkPriorityPtr(msg2);
      *x = SOLVE_RELAXATION(A, temp3, n, temp1, temp2) + cost;

      CreateChare(start, start@LEAF, msg2);
      Accumulate(acc_nodes, addfn());
    }
  }

\**************************************************** \
\* This function determines the updated cost matrix after *\
\* looking at the edges excluded and included in this branch *\
\****************************************************** \

determine_matrix0(msg, matrix, A, n)
MSG1 *msg;
int matrix[Max][Max];
int A[Max][Max];
int n;
{
  int index;
  int i, j, k;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      A[i][j] = matrix[i][j];
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
    {
      index = i*n + j;
      if (!IsFree(msg->isDefined, index))
          if (!IsFree(msg->in, index))
          {
            for (k=0; k<n; k++)
                A[k][j] = INFINITY;
            for (k=0; k<n; k++)
                A[i][k] = INFINITY;
            A[j][i] = INFINITY;
          }
          else
            A[i][j] = INFINITY;
      }
  }
```

APPENDIX A

PMESC CODE FOR THE TSP

```
\****************************************************************\
\* This is the main procedure that runs on every processor *\
\****************************************************************\


#include <stdio.h>
#include <fcntl.h>
#include "types.h"        \* This file contains PMESC internal definitions *\
#include "user_types.h" \* This file contains user definitions *\


FILE *fp;


int N;
int matrix[Max][Max];
int acc_nodes;
int bound;
int my_node;      \* Processor id *\
int work_nodes; \* Number of working nodes *\



main()
```

```
{
        \* Variables necessary for the Solve phase *\
        int choice, upper;
        int cost;
        int temp[Max][Max];
        int temp1[Max], temp2[Max];

        \**************************************************
        l_b and U_b are the lower and upper bounds for the queue length.
        A processor is considered overloaded when the queue length
        is greater than U_b and idle or lightly loaded when
        the queue length is less than l_b.
        **************************************************\
        int l_b, U_b;
        l_b = 0;

        \**************************************************
        All nodes in the tree have a level associated with them.
        The initial task (the root of the tree) has level 0,
        its children level 1, and so on.
        The level is used to avoid that a processor searches deep
        down the tree concentrating efforts on the same branch that
        may be unproductive after all. A threshold Thr is used to
        limit this search.
        **************************************************\
int Thr;

int *res;
Task *I;

int i, j;
int index;

my_node = GET_NODE();
work_nodes = GET_PROC();

fp = fopen("tsp-input.dat","r");
fscanf(fp,"%d %d", &l_b, &U_b);
fscanf(fp,"%d", &N);
fscanf(fp,"%d %d", &choice, &upper);

read_in_matrix(matrix,N);

\* Initialize bound *\
if (choice)
  bound = upper;
else
  bound = INFINITY;

\* Initialize variable that counts nodes *\
acc_nodes = 0;


I = (Task *)malloc(sizeof(Task));


\* First instance of Task *\
cost = reduce_matrix(matrix, temp, N, temp1, temp2);
I->cost = cost;
I->Level = 0;


for (i=0; i<N; i++)
  for (j=0; j<N; j++)
  {
      index = i*N + j;
      Reset(I->isDefined, index);
      Reset(I->in, index);
  }
\* First Task created *\


\************************************************
Node ROOT begins with the first task and generates
more tasks. The other processors are idle. They call
MAP(IDLE, other arguments) to get some work.
************************************************\
if (my_node == ROOT)
  ENQUEUE(TYPEP1, TYPEP2, I);
```

```
for (;;) {
  queue_length = DEQUEUE(TYPEP1, TYPEP2, I);


\***************************************************
To avoid unproductive computation, every time a processor
chooses a task, it checks its level.
If the level is greater than a given threshold, the processor
calls MAP(PRIOR, other arguments).
This keeps the priorities balanced as well as the load.
The frequency of these messages is controlled by the
threshold which is defined by the user.
***************************************************\
  while ( (l_b <= queue_length && queue_length < U_b+1)
          || (I->Level < Thr+1) )
  \* main loop *\
  {
          TSP(I);
          queue_length = DEQUEUE(TYPEP1, TYPEP2, I);
  }


  if (I->Level ≥ Thr)
  {
\***************************************************
MAP(PRIOR, other arguments) sends away a number of
tasks. It returns 1 when it detects messages carrying the
pseudo-global variable, otherwise it returns 0.
If the returned value is 1, the processor calls UPDATE to
update the bound with the new value.
*************************************************\
        if (MAP(TYPEM1, PRIOR, my_node, work_nodes))
          UPDATE(my_node, work_nodes, &bound, bound,
                   MONO);
        TSP(I);
        queue_length = DEQUEUE(TYPEP1, TYPEP2, I);
\* After reaching the threshold level, the processor
adjusts its threshold to search down the tree. *\
        if (I->Level ≥ Thr)
          Thr = Thr*2;
        TSP(I);
  }

  if (queue_length > U_b)
  {
  \* Processor heavily loaded *\
        if (MAP(TYPEM1, BUSY, my_node, work_nodes))
          UPDATE(my_node, work_nodes, &bound, bound,
                   MONO);
        TSP(I);
  }
```

```
        else
        {
        \**********************************************
        Processor is idle
        and calls MAP(IDLE, other arguments) to get more work or
        check for termination.
        MAP returns -1 when it detects termination.
        **********************************************\
            if (MAP(TYPEM1, IDLE, my_node, work_nodes)
                == -1)
                break;
        }
      \* End of infinity loop *\

      \* Processor detected termination *\
      \* Node ROOT sums up the number of nodes in the tree *\
      GATHER(SUM, ROOT, acc_nodes);
      if (my_node == ROOT)
        printf("total number of nodes: %d\n",acc_nodes);
}

\**************************************************\
\* This function reads in the initial cost matrix *\
\**************************************************\
read_in_matrix(A, n)
int A[Max][Max];
int n;
{
                int i, j;
                for (i=0; i<n; i++)
                  for (j=0; j<n; j++)
                        fscanf(fp,"%d",&A[i][j]);
                for (i=0; i<n; i++)
                  A[i][j] = INFINITY;
}


\**************************************************\
\* The following are the routines of the Solve module. *\
\* They were taken from the TSP code that comes with *\
\* Charm [23]. *\
\* We just removed the Charm calls *\
\* and replace them with PMESC calls when necessary. *\
\* This is indicated in the code. *\
\**************************************************\

#define IsFree(a,ind) !( (a[(ind/32)]) & (1<<(ind % 32)) )
```

```
#define Set(a,ind) a[(ind/32)] = ( a[(ind/32)] — (1<<(ind % 32)) )
#define Reset(a,ind) a[(ind/32)] = ( a[(ind/32)] & ( (1<<(ind % 32))) )

#define DETERMINE_MATRIX determine_matrix0
#define BRANCH_OUT branch_out0
#define SOLVE_RELAXATION reduce_matrix

struct solution {
            int cost;
            int cycle[Max];
};
typedef struct solution SolnMsg;


TSP(msg)
Task *msg;
{
            int n;
            int cost;
            int A[Max][Max];
            int soln[Max][Max];
            int rows[Max], columns[Max];

            DETERMINE_MATRIX(msg, matrix, A, N);
            cost = SOLVE_RELAXATION(A, soln, N, rows, columns);
            cost += msg->cost;
            if (cost < bound)
              if (isSolution(msg, soln, N))
              {
              \*****************************************\
              \* This is a PMESC call *\
                    UPDATE(my_node, work_nodes, &bound, cost,
                              MONO);
              \*****************************************\
                    printout_new_solution(soln, N);
              }
              else
                    BRANCH_OUT(msg, A, N, cost, rows, columns);
}


\**************************************************************\
\* From [23]: *\
\* This function and PRINT_S take care of printing out *\
\* a solution. Since the monotonic *\
\* variable does not guarantee the global best on every node *\
\* the solution found may not necessarily be the best *\
\**************************************************************\

printout_new_solution(soln, n)
```

```
int soln[Max][Max];
int n;
{
                int i, j;
                int next;
                int index = 0;
                SolnMsg *msg;

                msg = (SolnMsg *) malloc(sizeof(SolnMsg));
                msg->cost = bound;
                for (i=0; i<n; i++)
                  if (soln[0][i] == 1)
                        break;

                msg->cycle[index++] = 0;
                next = i;
                while (next != 0)
                {
                  msg->cycle[index++] = next;
                  for (j=0; j<n; j++)
                        if (soln[next][j])
                        {
                                  next = j;
                                  break;
                        }
                }
                PRINT_S(msg);
        }



        PRINT_S(msg)
        SolnMsg *msg;
        {
                int i, j;

                if (msg->cost <= bound)
                {
                  \****************************************** \
                  \* This is a PMESC call *\
                  UPDATE(my_node, work_nodes, &bound, msg->cost,
                          MONO);
                  \****************************************** \
                  printf("New tour of cost %d found:\n \t",msg->cost);

                  for (i=0; i<N; i++)
                          printf("%2d->",msg->cycle[i]);
                  printf("%2d\n",msg->cycle[0]);
                }
                free(msg);
        }
```

```
\******************************************************************* \
\* From [23]: *\
\* This function reduces the cost matrix to a non-negative *\
\* matrix. We use this to fire off the computations, and also *\
\* to determine the first reasonable lower bound. *\
\******************************************************************* \

reduce_matrix(A, soln, n, rows, columns)
int A[Max][Max];
int soln[Max][Max];
int n;
int rows[Max], columns[Max];
{
            int i, j;
            int lowest;
            int cost = 0;

            for (i=0; i<n; i++)
            {
              lowest = INFINITY;
              for (j=0; j<n; j++)
                        if (A[i][j] < lowest)
                              lowest = A[i][j];
```

```
    if (lowest != INFINITY)
    {
            for (j=0; j<n; j++)
                if (A[i][j] != INFINITY)
                  A[i][j] -= lowest;
            cost += lowest;
    }
    rows[i] = lowest;
}

for (j=0; j<n; j++)
{
    lowest = INFINITY;
    for (i=0; i<n; i++)
            if (A[i][j] < lowest)
                lowest = A[i][j];

    if (lowest != INFINITY)
    {
            for (i=0; i<n; i++)
                if (A[i][j] != INFINITY)
                  A[i][j] -= lowest;
            cost += lowest;
    }
```

```
              columns[j] = lowest;
            }
            for (i=0; i<n; i++)
              for (j=0; j<n; j++)
                        soln[i][j] = 0;
            return (cost);
}


\*************************************************************** \
\* [23]: *\
\* This function determines whether the solution to the *\
\* assignment problem is also a solution to the traveling *\
\* salesman problem. *\
\*************************************************************** \

isSolution(msg, soln, n)
Task *msg;
int soln[Max][Max];
int n;
{
            int x, y;
            int i, j;
            int next;
            int index;
```

```
        int length = 0;
        int cycle_start, cycle_end;

        for (i=0; i<n; i++)
          for (j=0; j<n; j++)
          {
                  index = i*n + j;
                  if (!IsFree(msg->in, index))
                  {
                        x = i;
                        y = j;
                        soln[i][j] = 1;
                        length++;
                  }
          }

        if (length != n-1)
          return(0);

        get_cycle_edge(msg, x, y, n, &cycle_start, &cycle_end);
        soln[cycle_end][cycle_start] = 1;
        return(1);
}
```

```
\**********************************************\
\* From [23]: *\
\* This function gets the sub-branch whose edge *\
\* may contribute to a cycle *\
\**********************************************\

get_cycle_edge(msg, v1, v2, n, cycle_start, cycle_end)
Task *msg;
int v1, v2;
int n;
int *cycle_start;
int *cycle_end;
{
            int i;
            int index;
            int end = v2;
            int start = v1;
            int done = 0;

            while (!done)
            {
              for (i=0; i<n; i++)
              {
                        index = end*n + i;
                        if (!IsFree(msg->in, index))
```

```
                {
                        end = i;
                        break;
                }
        }
        if (i==n)
                done = 1;
}

*cycle_end = end;
done = 0;

while (!done)
{
        for (i=0; i<n; i++)
        {
                index = i*n + start;
                if (!IsFree(msg->in, index))
                {
                        start = i;
                        break;
                }
        }
        if (i==n)
                done = 1;
                }
                *cycle_start = start;
        }



\************************************************************* \
\* From [23]: *\
\* This function returns the index of the least element in the row. *\
\************************************************************* \


least_in_row(A, row, n)
int A[Max][Max];
int row, n;
{
                int j;
                int least = INFINITY;

                for (j=0; j<n; j++)
                  if (least > A[row][j])
                        least = A[row][j];
                return(least);
}


\************************************************************* \
\* From [23]: *\
\* This function returns the index of the least element in the column. *\
\*************************************************************** \


least_in_col(A, col, n)
int A[Max][Max];
int col, n;
{
            int i;
            int least = INFINITY;

            for (i=0; i<n; i++)
              if (least > A[i][col])
                    least = A[i][col];
            return(least);
}



\**************************************\
\* From [23]: *\
\* This function determines the best zero *\
\**************************************\


find_best_zero(A, n, best_i, best_j)
int A[Max][Max];
int n;
int *best_i, *best_j;
{
        int temp;
        int store;
        int i, j, k;
        int best = SMALL;

        for (i=0; i<n; i++)
          for (j=0; j<n; j++)
          {
                temp = 0;
                if (A[i][j] == 0)
                {
                        store = A[i][j];
                        A[i][j] = INFINITY;
                        temp = least_in_row(A, i, n);
                        temp += least_in_col(A, j, n);
                        A[i][j] = store;
                        if (temp > best)
                        {
                          best = temp;
                          *best_i = i;
                          *best_j = j;
                        }
```

```
                    }
                }
            return(best);
}


\************************************************************* \
\* From [23]: *\
\* This is the trivial branching procedure. It determines *\
\* the best node in the relaxation problem and then branches *\
\* out in a binary fashion with one branch excluding the *\
\* edge and the other including the edge *\
\************************************************************* \

branch_out0(msg, A, n, cost, rows, columns)
Task *msg;
int A[Max][Max];
int n;
int cost;
int rows[Max], columns[Max];
{
            int x;
            int i, j;
            int temp;
            int best;
int index;
Task *msg1, *msg2;
int best_i, best_j;
int cycle_start, cycle_end;
int temp1[Max], temp2[Max], temp3[Max][Max];

best_i = best_j = (-1);
best = find_best_zero(A, n, &best_i, &best_j);
if (best_i != -1)
{
  msg1 = (Task *) malloc(sizeof(Task));
  msg2 = (Task *) malloc(sizeof(Task));
  msg1->cost = msg2->cost = msg->cost;
  msg1->Level = msg2->Level = Level+1;

  for (i=0; i<n; i++)
            for (j=0; j<n; j++)
            {
                  index = i*n + j;
                  if (IsFree(msg->isDefined, index))
                  {
                    Reset(msg1->isDefined, index);
                    Reset(msg2->isDefined, index);
                  }
                  else
```

```
                {
                  Set(msg1->isDefined, index);
                  Set(msg2->isDefined, index);
                }
                if (IsFree(msg->in, index))
                {
                  Reset(msg1->in, index);
                  Reset(msg2->in, index);
                }
                else
                {
                  Set(msg1->in, index);
                  Set(msg2->in, index);
                }
            }

index = best_i*n + best_j;
if ( ((cost < bound) &&
                (best >= INFINITY)) ||
                (cost + best < bound) )
{
            Set(msg1->isDefined, index);
            Reset(msg1->in, index);

            A[best_i][best_j] = INFINITY;
            if (least_in_row(A, best_i, n) == INFINITY)
                    if (rows[best_i] < INFINITY)
                      msg1->cost += rows[best_i];
            if (least_in_col(A, best_j, n) == INFINITY)
                    if (columns[best_j] < INFINITY)
                      msg1->cost += columns[best_j];


            x = cost + best;
            msg1->Prio = x;
            A[best_i][best_j] = 0;


\****************************************** \
\* This is a PMESC call *\
            ENQUEUE(TYPEP1, TYPEP2, msg1);
\****************************************** \
            acc_nodes++;
}


\* include best edge and exclude its symmetric counterpart *\
if (rows[best_i] < INFINITY)
            msg2->cost += rows[best_i];
if (columns[best_j] < INFINITY)
            msg2->cost += columns[best_j];
Set(msg2->isDefined, index);
Set(msg2->in, index);
```

```
index = best_j*n + best_i;
Set(msg2->isDefined, index);
Reset(msg2->in, index);

temp = A[best_j][best_i];
A[best_j][best_i] = INFINITY;
if (least_in_row(A, best_j, n) == INFINITY)
          if (rows[best_j] < INFINITY)
                msg2->cost += rows[best_j];
if (least_in_col(A, best_i, n) == INFINITY)
          if (columns[best_i] < INFINITY)
                msg2->cost += columns[best_i];
A[best_j][best_i] = temp;

\* Get the edge that completes the cycle and exclude it *\
get_cycle_edge(msg2, best_i, best_j, n, &cycle_start,
               &cycle_end);
index = cycle_end*n + cycle_start;
Set(msg2->isDefined, index);
Reset(msg2->in, index);
temp = A[cycle_end][cycle_start] = INFINITY;
A[cycle_end][cycle_start] = INFINITY;
if (least_in_row(A, cycle_end, n) == INFINITY)
          if (rows[cycle_end] < INFINITY)
                msg2->cost += rows[cycle_end];
```

```
            if (least_in_col(A, cycle_start, n) == INFINITY)
                        if (columns[cycle_start] < INFINITY)
                          msg2->cost += columns[cycle_start];
            A[cycle_end][cycle_start] = temp;

            A[best_j][best_i] = A[cycle_end][cycle_start] = INFINITY;

            for (i=0; i<n; i++)
                        A[best_i][i] = A[i][best_j] = INFINITY;
            x = SOLVE_RELAXATION(A, temp3, n, temp1, temp2)
                + cost;
            msg2->Prio = x;

            \**********************************\
            \* This is a PMESC call *\
            ENQUEUE(TYPEP1, TYPEP2, msg2);
            \**********************************\
            acc_nodes++;
          }
}

\************************************************************\
\* From [23]: *\
\* This function determines the updated cost matrix after *\
\* looking at the edges excluded and included in this branch *\
\****************************************************************** \


determine_matrix0(msg, matrix, A, n)
Task *msg;
int matrix[Max][Max];
int A[Max][Max];
int n;
{
        int index;
        int i, j, k;


        for (i=0; i<n; i++)
          for (j=0; j<n; j++)
              A[i][j] = matrix[i][j];
        for (i=0; i<n; i++)
          for (j=0; j<n; j++)
          {
              index = i*n + j;
              if (!IsFree(msg->isDefined, index))
                      if (!IsFree(msg->in, index))
                      {
                        for (k=0; k<n; k++)
                            A[k][j] = INFINITY;
                        for (k=0; k<n; k++)
                        A[i][k] = INFINITY;
                    A[j][i] = INFINITY;
                }
                else
                    A[i][j] = INFINITY;
        }
}
```