The PMESC Programming Library for Distributed-Memory MIMD Computers

S. Crivelli
E.R. Jessup

CU-CS-814-96

Abstract

This paper presents the PMESC library for managing task-parallel problems on distributed-memory MIMD computers. Efficient parallel programming of task-parallel problems, where the number and execution times of the tasks can vary unpredictably, demands an asynchronous and adaptive approach. In such an approach, however, such fundamental programming issues as load sharing, data sharing, and termination detection can present difficult programming problems. PMESC offers support for all of these issues in a portable and efficient way while still allowing users to customize their codes. The library is based on a model in which message passing and machine architecture are almost absent thus making it accessible to a wide variety of users.

The PMESC Programming Library for Distributed-Memory MIMD Computers

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

CU-CS-814-96 October 1996

The PMESC Programming Library for Distributed-Memory MIMD Computers

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

October 1996

1 Introduction

Many physical phenomena exhibit irregular and unpredictable behavior. The numerical methods used to study the mathematical models of such phenomena typically give rise to separate computations evolving at different rates. These computations are nonuniform because they depend on variant amounts of data or because they are composed of tasks whose number and execution time change unpredictably. When the tasks can be carried out in parallel, computations of the latter sort are called *irregular task-parallel.* Examples of irregular task-parallel methods include global optimization [44, 45], combinatorial optimization [1, 2, 21, 25, 37, 38, 46], adaptive numerical quadrature [9, 13, 14, 15, 39, 40, 41], and the bisection method for computing eigenvalues of symmetric matrices [3, 6, 9, 29, 34]. An efficient approach to dealing with these problems needs to be adaptive so that it can react dynamically to the changes occurring during computation. However, dynamic approaches are difficult to implement on multiprocessors owing to the complexity of such issues as processor load sharing and termination detection. The difficulty of writing dynamic programs is compounded by the need for portability if the programs are to remain useful as parallel architectures change.

The time consuming and challenging process of producing codes that are both portable and reasonably efficient is eased by the use of high-level parallel programming tools. Obvious characteristics of a well-designed programming tool include efficiency and portability. Other desirable features pertain to usability of the tool. First, the tool should be flexible. It should provide a variety of approaches to handle the main programming issues, and it should provide users with the freedom to choose between them easily as they tune their codes. Second, the tool should be accessible to a variety of users. It should be usable by scientists, engineers, and other non-computer scientists who want to use the advanced architectures without becoming experts in the technical details of the machines.

In this paper, we introduce the PMESC library for managing medium- to coarse-grained task-parallel problems on distributed-memory MIMD computers. The library supports a dy-

*This research was supported by National Science Foundation grant ACS–9357812, Department of Energy grant DE–FG03–94ER25215, and the Intel Corporation.

namic, asynchronous approach that lets processors work independently as much as possible while still maintaining an effective use of the computational resources.

PMESC meets the goal of flexibility by providing building blocks built on top of MPI. These building blocks represent different alternatives to different programming issues. The concept of providing changeable blocks to fit the requirements of unstructured computations was pioneered by Charm [16, 30], a language originally designed for task-parallel computations. Beyond that, however, the design philosophies of PMESC and Charm differ. While both tools offer a variety of routines to handle such programming concerns as load sharing, exchange of information, and program termination, PMESC alone allows the user to choose when and how often every routine is invoked. In addition, only PMESC permits the user to change routines at runtime as needed during the process of code development and testing. Furthermore, while Charm is geared toward experienced users of parallel computers, PMESC is also accessible by inexperienced programmers. It assists the less knowledgable user in writing portable and reasonably efficient code without the burden of learning a new language, a complex interface, or the machine architecture details. It assists the expert by allowing the reuse of code and by providing a convenient platform for testing new applications, for comparing different strategies, and for trying different virtual and real architectures.

A prototype of PMESC, implemented as a C library, runs on any platform that supports MPI. (The embedding routines must be modified when PMESC is ported to an architecture not yet supported.) We have tested it on the Intel iPSC/860, the Thinking Machines CM5, and the IBM SP1. In this paper, we present computational results on those machines for the parallel bisection problem. Performance achieved with this portable application shows considerable gains with respect to the statically scheduled approach without significant programming effort. For similar results on other application problems and for more results on the listed machines, see [9].

The paper is organized as follows. In section 2, we characterize task-parallel problems and present two examples. In section 3, we describe the PMESC paradigm that allows us to recognize the different phases in the computation and to treat them as separate modules. In section 4, we introduce the library based on the PMESC paradigm, and, in section 5, we provide the specification of the library. In section 6, we present computational results and discuss performance issues. Finally, in section 7, we summarize the main features of PMESC and discuss some future work.

2 Background

Task-parallel problems present a complex task that can be decomposed into medium- to coarse-grained subtasks for parallelization. These subtasks then can be executed asynchronously and independently by different processors. If processors need to exchange some data or work, they can usually do it asynchronously. Task-parallel problems can be regular or irregular. Regular problems are those whose computational requirements can be estimated a priori. Their efficient solution requires a static approach that partitions and maps work to the processors only once, usually at the beginning of the execution. Irregular problems, on the other hand, are those whose computational requirements cannot be estimated in advance. Their efficient computation requires a more sophisticated approach that dynamically balances the non-uniform workloads and asynchronously checks for termination.

Examples of irregular task-parallel problems can be found in computations involving some type of tree search (among those are most of the examples listed in Section 1 of this paper.) This type of computation is difficult to partition and map to a distributed-memory computer because different branches of the trees may have different number of nodes and levels. In addition, the trees evolve dynamically, making it impossible to achieve an efficient initial mapping of the work among the processors. We next present two examples of task-parallel computations. First is the parallel bisection problem, representing a case in which tasks can be executed in any order and processors do not need to communicate during their execution. Then we describe the traveling salesman problem, illustrating a different situation in which tasks are given priorities and processors need to exchange information from time to time.

2.1 Parallel Bisection

An example of an irregular task-parallel problem is given by the computation of the eigenvalues of a matrix by the bisection procedure [4, 24]. The problem consists of computing some or all of the eigenvalues of a symmetric, tridiagonal matrix $\mathcal{T}$ with subdiagonal elements different from zero. The number of eigenvalues of $\mathcal{T}$ smaller than any real number 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 matrix property translates into a simple computational procedure by which one can determine 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. The process is applied recursively to every non-empty subinterval until the length of the subinterval is less than a given threshold. In that case, the midpoint of that subinterval is taken as a computed eigenvalue. Thus, the bisection procedure can be associated with a tree, where the root corresponds to the initial interval, the nodes to its subintervals, and the leaves either to computed eigenvalues or to empty subintervals that are discarded.

The bisection procedure presents a straightforward case for parallelization. If the initial intervals containing the desired eigenvalues are distributed among the processors, each processor can apply the sequential bisection procedure independently and asynchronously to each of its assigned intervals. All non-empty intervals must be bisected, and the order does not matter. However, there is a problem that may prevent 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, any initial partition and assignment of the work will probably end up with different processors having different workloads.

2.2 The Traveling Salesman Problem (TSP)

Another typical example of task-parallel problem is given by the computation of the TSP by the branch-and-bound procedure. The problem presents 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 of 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 $\sum_{i=1}^{n} d_{i,\sigma(i)}$.

The branch-and-bound procedure searches through a tree of partial solutions. Thus, like bisection, it can be associated with a tree. In this case, the nodes of the tree represent partial solutions to the given problem. They are generated by using a branching procedure which, when applied to any problem $\mathcal{P}$, either solves it directly or else derives a set of subproblems such that the solution of $\mathcal{P}$ can be found from the solution of the subproblems. Thus, when a branching procedure is applied to node v, it either determines that v is a leaf or else produces the children of v. A leaf represents either a possible solution of the problem or an unproductive partial solution, i.e., a partial solution that cannot lead to a solution. The solution of the problem is given by the leaf with minimum cost.

Unlike the bisection case, the branch-and-bound method *reduces* the size of the search tree dynamically. In fact, it uses a bounding procedure to prune those branches of the tree that cannot produce a solution [25, 47]. It does so by defining a cost function c that assigns a value $c(v)$ to each node v based on the values of the nodes in the path from the root to v. The bounding procedure is based on the monotonicity property of the cost function that states that the cost of a subproblem of $\mathcal{P}$ is at least as large as the cost of $\mathcal{P}$. This property ensures that any subproblem with associated cost bigger than the cost of one solution can be ruled out.

Consequently, unlike the bisection case, a parallel branch-and-bound algorithm may not achieve good speedup by merely keeping the processors workload balanced. It must also direct the computation toward the branches of the tree with less cost. To that end, nodes are given priorities according to the costs associated with them. Nodes with lower cost should be given higher priority because they are more likely to produce a solution.

The examples presented are task-parallel because the initial work can be split into smaller pieces or tasks that can be executed asynchronously by different processors. A task consists of following one or more paths down the tree starting from a given node. Because these paths are created dynamically, it is impossible to assign similar loads to all the processors at once. This situation usually results in a poor distribution of work that needs to be fixed as the computation evolves. To that end, processors need to activate load sharing mechanisms that transfer work as necessary.

In the next section we present an abstraction that allows us to recognize the different phases involved in the computation of task parallel problems. Identifying these phases and providing efficient computational techniques to handle them is the basis of the PMESC approach.

3 The PMESC Paradigm

A high-level analysis of task-parallel problems shows that their implementation on distributed-memory computers involves the same basic components. Different components may be resolved via different algorithms and may present wholly different programming issues. Some components must be developed by the user while others can reasonably be supplied in a library. In either case, codes for individual components are easier to implement and debug than is a monolithic piece of code. Furthermore, separation into components promotes a modular design in which the pieces or building blocks can be easily changed and reused. To that end, we present the PMESC paradigm for structuring the algorithms that allows the separation of different phases in the computation involving different programming issues.

The **Partition-Map-Embed-Solve-Communicate** (PMESC) paradigm is composed of five phases bearing those names. The **Partition** phase splits a problem into subproblems. The

Map phase assigns (and reassigns as necessary) those subproblems to the set of processors interconnected by some convenient virtual topology. The **Embed** phase embeds the virtual topology or topologies used for the application into the actual machine architecture. The **Solve** phase executes the subproblems. The **Communicate** phase takes care of the interprocessor communication necessary to exchange information. The P, M, E, S, and C phases can appear in any order and number. In the PMESC model, the identification of the phases is what matters, not their sequence.

Figure 1 shows the PMESC phases in a task-parallel problem. First comes the Partition phase with the initial subdivision of the work into units of work or tasks. This partitioning must account for 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 next phase is the Map phase, which distributes the units of work among the processors of a convenient virtual machine. In the tree search examples of section 2, a convenient virtual topology is a tree. The Embed phase takes care of embedding that tree into the actual machine.

Once processors receive their tasks, they create a queue of tasks ready for execution. Because the queue stores the tasks, its handling is represented by the Partition phase. Thus, partition represents the process of selecting a task from the queue as well as that of storing a task in the queue. This queue can be centralized—i.e., stored and maintained by a master processor—or distributed—i.e., split into local queues stored and maintained by all the processors [32]. Figure 1 illustrates the distributed case because it represents the most scalable as well as the most challenging approach to implement on distributed-memory computers.

In the distributed-queue approach, processors execute the tasks in their local queues. They do so by taking a task from the queue and executing it, eventually producing new tasks that are placed in the queue. The Solve phase represents the execution of those tasks. Processors may also need to share some information. The TSP example depicts a situation in which is necessary to exchange the solution of least cost found so far in order to prevent some processors from working on unproductive branches of the tree. This interprocessor communication, necessary to exchange data rather than work, is represented by the Communicate phase. The exchange of tasks is always a Map phase.

The innermost loop continues as long as the processor is moderately loaded. A processor can determine whether it is moderately loaded according to various criteria. In the framework shown in Figure 1, a processor is considered moderately loaded when the length of its queue is between some lower and upper bounds.

When a processor becomes overloaded or underloaded, the queue length exceeds the upper bound or is less than the lower bound, respectively. In those cases, the processor activates the load sharing mechanisms to transfer some work to or to get some work from another processor. This is work redistribution and, therefore, it corresponds to the Map phase. The cycle repeats until all of the local queues are empty.

The PMESC paradigm provides the programmer with a simple model for task-parallelism. In the next section, we introduce the library that supports this model.

```
Initialize;
begin
    Partition;  \* splits the work into tasks *\
    Map;        \* assigns tasks to processors using virtual machine *\
    Embed;      \* embeds virtual machine into real one *\

    while (queues are not empty)
        while (lower bound < local queue length < upper bound)
            Partition; \* gets task from queue *\
            Solve;     \* executes task *\
            if (exchange of data is necessary)
              Communicate;
        end
        Map; \* reassigns work to processors *\
    end
end
```

Figure 1: PMESC framework for task-parallel problems

4 The PMESC Library

PMESC is a programming tool designed to offer a set of implementation strategies to task-parallel application programmers. It was developed by finding abstractions common to different applications and introducing the routines that were necessary to implement them. Thus, it provides a high-level environment which focuses on the applications and their requirements. The library fulfills those requirements by offering utilities at all levels of complexity from embedding of virtual into real architectures to automatic load sharing as well as asynchronous exchange of global information. An important feature of PMESC is that it frees the users from dealing with the low-level details of the computation without hiding parallelism from them.

The library is composed of a set of routines that allows users to build a modular, easily changeable interface between the problem and the machine. It is designed in two layers. At the lower level, PMESC provides routines for embedding a virtual architecture into the real one, and MPI provides routines for point-to-point message passing. At the higher level, PMESC provides the high-level abstractions for handling the remaining programming issues including task redistribution, termination checking, and so on. The low- and high-level routines together form the basis for a flexible model of computation in which the underlying topology of the hardware can be completely ignored. The user may call the low-level routines directly or may avoid them entirely by programming only in high-level routines and allowing those routines to call the low-level routines as necessary.

Each level of PMESC is distinct and independent of the other, with the higher level built on top of the lower one. As a result of this 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 PMESC library is an ongoing project. It was designed to meet the requirements prescribed by Parallel Tools Consortium (Ptools) [35] of being usable as a standalone environment as well as a building block for future integrated parallel programming environments. We expect to expand and modify the library in response to the development and evolution of new paradigms and techniques.

4.1 Motivations for a New Library

Although there has been a great deal of effort directed at providing software for irregular data-parallel computations [5, 7, 8, 22, 26, 28, 31, 36, 43], up to this point, only a few tools have been developed specifically to address irregular task-parallel ones. Among them, the most complete and mature product is Charm [16, 30]. Charm is a parallel programming system that supports an explicitly parallel C- and C++-based language for data- and task-parallel computations. For data-parallel computations, it provides static load sharing and induces data locality. For task-parallel computations, it includes management of processes, support for prioritization and information sharing and dynamic load sharing strategies. Charm allows machine independent parallel programming over the class of MIMD machines —shared and non-shared memory. It provides an environment where the user has to specify the creation of tasks or **chares** and the communications between them explicitly, leaving the management of chares —load sharing, scheduling, etc.— to the system.

PMESC borrows from Charm the explicit definition of the tasks and the choices of strategies to use for some programming issues. However, Charm does not provide complete control to users. Rather, it leaves some important decisions to the system, reducing the user's ability to finely tune the program while still using the tool. In contrast, PMESC is a run-time library that allows programmers to decide *which* strategies to use and *when* to use them.

4.2 A Program for the Parallel Bisection Example

In this section, the use of the PMESC library is illustrated by a simple program for the parallel bisection algorithm presented in section 2. Figure 2 depicts a pseudo-code for the sequential version `SeqBisection`. This function calls itself recursively every time it finds an interval containing eigenvalues. The process repeats until the length of the intervals is less than the desired accuracy `Threshold`.

Figure 3 shows the task-parallel version of bisection using PMESC. It begins with the header file `pmesc.h` that contains all the constants and variables needed by PMESC. In PMESC, programmers have to create the parallel actions called tasks explicitly. Therefore, the next step is to define those tasks. In the parallel bisection example, a task may consist of applying a single step of the recursive procedure to an interval containing at least one eigenvalue. One can think of this task as traversing a single level down the tree of intervals from one node to its children. With the tasks identified, the programmer must define the variable type Task 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 (which determine the number of eigenvalues in that interval), and a threshold to decide when to stop the subdivision.

The main code starts with the usual definition of variables. The program then invokes the GetID and GetNumProc functions to get the processor id and the number of processors assigned to the computation. The function PMESC_Init specifies the virtual topologies to be used for the load sharing, termination, and communication procedures. These virtual architectures can be changed at any time during the computation. In this particular case, an all-connected topology is specified for load sharing and a tree for termination checking. None is specified for

```
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 *\

  \* calls itself for every nonempty subinterval
           until it finds an eigenvalue *\
  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 2: Pseudo-code for the sequential bisection procedure

```
#include "pmesc.h"

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;      \* desired accuracy *\
};
typedef struct task Task;

main(int argc, char **argv)
{
  int my-id, num-proc, l_b, U_b, queue-length, limit;
  int signal = 0;
  Task T;
  LIFOq fq;                        \* defines LIFO queue *\
  PMESC_flag Bflag;

  GetID(&my_id);                   \* gets processor id *\
  GetNumPr(&num_proc);             \* gets number of processors *\

  PMESC-Init(AllC, Tree, None, my_id, num_proc); \* specifies virtual topologies *\

  Read l_b and U_b;                \* gets lower and upper bounds for queue *\

  InitPart_and_Map(my_id, num_proc);

  while ( signal ) {
        queue_length = DEQUEUE (&fq, &T, size_of_task);
        while (l_b < queue_length ≤ U_b) {          \* processor moderately loaded *\
              ParBisection (&fq, T);
              DEQUEUE (&fq, &T, size_of_task);      \* gets task from queue *\
        }

        if (queue_length > U_b) {                   \* processor heavily-loaded *\
              MAP_RaR (&fq, BUSY, tasks_to_send, size_of_task, my_id, num_proc, limit, &Bflag);
              ParBisection (&fq, T);
        }
        if (queue_length ≤ l_b)                     \* processor lightly-loaded or idle *\
              signal = MAP_RaR (&fq, IDLE, tasks_to_send, size_of_task, my_id, num_proc, limit, &Bflag);
  }
}
```

Figure 3: Pseudo-code for the main procedure corresponding to the parallel bisection example using a distributed queue approach

communication because communication is not necessary. Next, the lower and upper bounds for the queue, `l_b` and `U_b`, are entered as parameters so that the user can try different values. Then comes the `InitPart_and_Map` function which makes processor ROOT read the initial interval or intervals and pass them to the other processors so that they can create their first tasks. The latter function corresponds to the initial Partition and Map phases of the PMESC paradigm and to the Partition and Map functions of the PMESC framework depicted in Figure 1. The PMESC_Init function corresponds to the Embed phase.

As Figure 1 indicates, after defining the tasks and assigning work to the processors, the program enters a cycle including Partition, Solve, and Communicate phases as applicable. The Partition and Solve phases appear in the program of Figure 3. (Partition, Solve, and Communicate appear in the program of Figure 5.) At every step of the cycle, the program takes a task from the queue —using the PMESC Partition function `DEQUEUE`— and executes it —using the Solve function `ParBisection` supplied by the user. Note that the `DEQUEUE` function takes as an argument a pointer to the queue. This queue is represented as a structure LIFOq which means that the Last task In is the First Out. The `ParBisection` procedure is based on the sequential bisection procedure `SeqBisection`. `ParBisection` selects a task from the queue and applies bisection to it. However, unlike `SeqBisection`, `ParBisection` does not call itself recursively. Instead, it creates new tasks and places them in the queue. When `ParBisection` finishes with the current task, it terminates. Figure 4 shows the pseudo-code corresponding to this procedure.

When the queue length is greater than the upper bound, the program calls the load sharing routine. For this example, we assume a random, receiver-initiated approach. Therefore, the routine to use is MAP_RaR with input parameter set to BUSY—for overloaded. When the queue length is less than the lower bound, the program also calls the load sharing routine, this time with the input parameter set to IDLE. In that case, the load sharing routine actively searches for work, and, if it cannot find any, it automatically calls another routine to check for termination. MAP_RaR returns -1 when it detects termination. Otherwise, it returns 0. The process of transferring tasks between processors and, eventually, detecting termination is part of the Map phase of the PMESC paradigm.

4.3 A Program for the Traveling Salesman Problem

In this section, we discuss the PMESC program for the traveling salesman problem presented in section 2. Figure 5 shows the corresponding code. As in the bisection program, this code follows the lines of the PMESC framework presented in Figure 1. Thus, for the sake of brevity, we only discuss those aspects of the code that are different from the bisection example.

In this problem, a task consists of the branch and bound procedures to a given partial solution. 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. Thus, the structure Task contains an $n \times n$ matrix (n being the number of cities) that represents the graph and an integer that contains the cost. The matrix 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.

The process begins with a single partial solution, i.e., with a single task. Our implementation of the `InitPart_and_Map` function makes a designated processor, ROOT, execute the original

```
ParBisection (LIFOq *fq, Task *T)
{
  Task T1;
  double mid; \* midpoint of interval $[a,b]$ *\
  int cm;      \* eigenvalue count at mid *\

  \* creates a Task for every nonempty subinterval it finds *\
  \* Tasks are placed in queue *\
  if ( | 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;
      ENQUEUE (fq, T, size_of_task);
    }
    else if (T->ca = cm and cm < T->cb) {
          T->a = mid;
          T->ca = cm;
          ENQUEUE (fq, T, size_of_task);
    }
    else {
        T1.a = mid;
        T1.ca = cm;
        T1.b = T.b;
        T1.cb = T.cb;
        ENQUEUE (fq, &T1, size_of_task);
        T.b = mid;
        T.cb = cm;
        ENQUEUE (fq, T, size_of_task);
    }
  }
}
```

Figure 4: Pseudo-code for the ParBisection function

task, create new tasks, and assign a task to each one of the processors. Again, `InitPart_and_Map` corresponds to the initial Partition and Map phases.

Because tasks are given different priorities, the queue is defined as PRIORity rather than as LIFO. Also, in order for all the processors to share some of the high priority tasks, they must exchange tasks periodically. Thus, the main loop breaks not only when the processor becomes overloaded or underloaded but also when `accum` reaches a given threshold. Other ways to deal with the balancing of prioritized tasks are discussed in [9, 12, 11, 27, 33, 42]. Load balancing is always a Map phase.

Another characteristic of this example is the use of the routine `COMM_Up` to exchange the value of the upper bound. This bound is determined by the least cost of the solutions found so far and can be updated every time a new solution with lesser cost is found. The communications necessary to perform the updates correspond to the Communicate phase of the PMESC paradigm. For this phase, processors may use a virtual topology that is different from the one used for the load sharing mechanisms. This virtual topology is specified in the PMESC initialization call `PMESC_Init` which represents the Embed phase.

5 Specification

The PMESC library is composed of a set of routines that address different issues involved in the implementation of task-parallel computations. 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 interprocessor communication make up the Communicate module.

In this section we provide a specification of PMESC. We also discuss the choices we made in selecting the strategies to attack each issue efficiently.

5.1 Handling the Task Queue

In task-parallel applications, queues are used to store the tasks. Queues can be centralized or distributed. Centralized queues are maintained by a master processor while distributed queues are maintained by all the processors. So far, the library provides support only for distributed queues because they permit more scalable applications than do centralized queues. Queues can also be non-prioritized and prioritized. PMESC offers a queueing strategy for both cases. In the former, the last task queued is the first out (LIFO). In the latter, the highest priority task is the first out.

A LIFO queue is appropriate for problems such as bisection where tasks can be executed in any order without affecting either the correctness of the results or the overall performance. On the other hand, a priority queue is recommended for problems such as the TSP where tasks are assigned different priorities.

PMESC provides routines for the creation of the task queue structure, dynamic allocation and deallocation of memory for this structure, addition of tasks to the queue, selection of tasks from the queue, and partition of the queue structure.

```
#include "pmesc.h"
#define n                        \* number of cities *\
struct task {
          double Graph[n][n]; \* matrix of costs *\
          double cost;        \* task priority *\
};
typedef struct task Task;

main(int argc, char **argv)
{
  int my_id, num_proc, l_b, U_b, queue_length, limit;
  int signal = 0;
  int accum = 0;
  Task T;
  double bound = INFINITY;        \* pseudo-global variable
                                  containing the best solution found so far *\
  PRIORq fq;                      \* defines priority queue *\
  PMESC_flag Bflag;
  GetID(&my_id);                  \* gets processor id *\
  GetNumPr(&num_proc);            \* gets number of processors *\
  PMESC_Init(AllC, Tree, Tree, my_id, num_proc); \* specifies virtual topologies *\
  Read l_b and U_b;               \* gets lower and upper bounds for queue *\
  InitPart_and_Map(my_id, num_proc);

  while ( signal ) {
        queue_length = DEQUEUE (&fq, &T, size_of_task);
        while (l_b < queue_length ≤ U_b &&
              accum < Thr) {
              TSP (&fq, T);
              COMM_Up (my_id, num_proc, &bound, value); \* updates pseudo-global variable bound *\
              accum++;
              DEQUEUE (&fq, &T, size_of_task);          \* gets task from queue *\
        }
        if (queue_length > U_b)                         \* processor heavily-loaded *\
              MAP_RaS (&fq, BUSY, tasks_to_send, size_of_task, my_id, num_proc, limit, &Bflag);
              TSP (&fq, T);
              COMM_Up (my_id, num_proc, &bound, value); \* updates pseudo-global variable bound *\
        if (queue_length ≤ l_b)                         \* processor lightly-loaded or idle *\
              signal = MAP_RaS (&fq, IDLE, tasks_to_send, size_of_task, my_id, num_proc, limit, &Bflag);
        if (accum > Thr-1) {                            \* exchanges high priority tasks *\
              MAP_RaS (&fq, PRIOR, tasks_to_send, size_of_task, my_id, num_proc, limit, &Bflag);
              TSP (&fq, T);
              accum = 0;
        }
  }
}
```

Figure 5: Pseudo-code for the main procedure corresponding to the parallel traveling salesman example using a distributed queue approach

5.2 Dynamic Load Sharing

The basis of the solution of task-parallel problems is the creation of tasks to be executed in parallel. The actual gains in performance depend heavily on their efficient distribution among the processors. The load sharing routines are responsible for the assignment of tasks to processors so that the system resources can be utilized efficiently. By load sharing we mean the reallocation of tasks to keep the computational workload distributed, rather than *evenly* distributed, among the processors. This approach is reasonable as the tasks sizes are mostly uneven and are usually unknown.

Depending on the amount of system state information used, load sharing mechanisms can vary from simple to complex ones. Eager et al. [19] show that simple dynamic load sharing strategies that use small amounts of information yield performance close to that expected from more complex approaches. Therefore, PMESC provides simple mechanisms that collect very small amounts of information and use it in very simple ways.

Depending on how the control is distributed among the processors, load sharing strategies can be centralized, hierarchical, distributed or hybrid [17, 13, 18, 20, 23, 14, 32, 45]. In general, centralized and even hybrid strategies are easy to implement but they do not scale well [32]. For that reason, the PMESC library supports distributed strategies that let each processor manage its own load.

The PMESC strategies for load sharing are referred to as Random, Threshold, and Ring. The Random strategy distributes tasks to processors selected at random. A selected processor accepts the tasks only if its local queue is below some threshold. Otherwise, the tasks are transferred to another randomly selected processor. To avoid instability, the number of times a task can be transferred is restricted by using a transfer limit [19].

Threshold is a variant of the random strategy that selects a processor at random and probes it to determine whether a transfer of some tasks to it would put its queue length above a given threshold. If not, the tasks are transferred to that processor. If so, another processor is selected at random and probed in the same way. This continues until either a destination processor is found or else the number of probes exceeds a probe limit. Eager et al. show that the performance of this strategy is insensitive to the choice of probe limit [19].

The Ring strategy assigns tasks among the neighbors in a ring. It also imposes a transfer limit. The strategy may be useful for those platforms in which the cost of communicating between distant processors can be high.

Load sharing strategies come in two different versions: sender- and receiver-initiated. In the sender-initiated approach, heavily-loaded processors send out some work without it being requested. In receiver-initiated strategies, idle or lightly-loaded processors send out a request for more work. In the latter case, work is sent only upon request. Studies show that both, receiver- and sender-initiated approaches for load sharing are better than the static approach. They also show that the receiver-initiated approach outperforms the sender-initiated one at high system loads and that the sender-initiated approach is preferable at light to moderate system loads [18].

5.3 Termination Detection

Termination detection is a difficult programming issue in distributed-memory computers. This is particularly true in computations where processors work asynchronously and tasks migrate among them for load sharing. PMESC provides an efficient routine based on the prefix algorithm

[9]. It assumes that processors are connected by a virtual tree, but it can use any other virtual topology that is appropriate for a combine operation. (When a new virtual architecture is added, however, corresponding Embed routines must be developed for the library.) Each processor keeps a count of the messages it has sent to request work minus the messages received asking for 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. If so, 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 the root then broadcasts a termination message. Otherwise, there is at least one requesting message that was sent but not 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 $2\log_2(p)$ steps to complete. Refer to [10] for a complete discussion of this and other strategies.

5.4 Asynchronous Sharing of Information

In task-parallel computations, processors do not synchronize. Thus, if they need to share some information, they must define a pseudo-global variable. To implement a pseudo-global variable, each processor has its own copy of the variable in its local memory. To keep the copies updated, processors exchange their values from time to time. There are two different ways to update pseudo-global variables: centralized and distributed [9]. In the centralized case, processors communicate their pseudo-global values to their parents in a tree but only update them when they receive a new value from the root. Thus, to complete an update, the tree is traversed twice: once from the leaves to the root and then again from the root to the leaves. In the distributed case, processors communicate their values to their parents and update them along the way. The latter approach allows processors to use new information coming from their children without having to wait until it comes from the root.

PMESC implements centralized and distributed approaches. Although more costly, the centralized approach must be applied to those cases in which the information needs to be accumulated by a central processor (like a global sum.) The distributed approach, on the other hand, is more convenient in cases in which the partial results can be used. The TSP presents an example in which the distributed approach can be used because the variable shared is a minimum (i.e., the least cost of the solutions found so far.) Any partial minimum, if better than the processor's own, is valid and so should be used quickly to prune the unproductive branches of the search tree.

5.5 Embedding

Another fundamental task in the implementation of a parallel algorithm is the embedding of virtually connected processors into 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 efficient embedding routines that exploit the hardware characteristics while keeping them hidden from the user. So far, the library includes algorithms for embedding rings, trees, and arrays onto the supported platforms.

6 Numerical Results and Performance

PMESC provides support for task-parallel computations using a static or 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 case. Besides executing tasks from their queues, processors continuously check their loads, dynamically transfer tasks to keep them balanced, and asynchronously check for termination. The overhead associated with these issues can be considerable. Therefore, the dynamic approach should only be considered for irregular cases. The question is how irregular a problem must be in order to justify the overhead associated with the dynamic approach.

In this section, we discuss some experiments that answer this question and show the efficiency and flexibility of PMESC. First, we show that the dynamic approach outperforms the static one even on problems that are slightly irregular. Second, we illustrate how PMESC allows users to tune their codes. We show how different choices of load sharing strategy, lower and upper bounds for the queue length, and granularity, can improve the performance. We use the bisection example for our experiments because it can be easily adapted to represent different situations by just changing the matrix type or size. The results of these experiments are representative of those for a variety of other task-parallel problems including the TSP. For more experimental results, see [9].

6.1 When the Dynamic Approach Outperforms the Static One

Our experiments show advantages of the dynamic over the static approach on problems that present different "degrees" of irregularity. The degree of irregularity of a problem is given by the level of difficulty in making an even distribution of the work among the processors. The more unpredictable and unevenly distributed the computation is, the higher its degree of irregularity.

The bisection procedure applied to the test matrix $[1, 2, 1]$ that has 2's on the diagonal and 1's on the subdiagonal illustrates a highly irregular case. The eigenvalues of this matrix, 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, tend to concentrate at the endpoints of the interval $[0, 4]$ as n increases. The case $n = 10^4$ is definitely irregular as most eigenvalues are concentrated but still computationally different. Figure 6 shows the distribution of the eigenvalues in the interval $[0, 4]$. Problems that present this characteristic are difficult to partition efficiently because no matter how the initial interval is distributed among the processors, most of the work is likely to be assigned to a few processors.

Figure 7 shows the execution time of each processor using the static and dynamic approaches to compute all eigenvalues of the matrix [1,2,1] for $n = 10^4$ with PMESC on the CM5. In this case, the initial interval is partitioned into equally-sized subintervals that are assigned to the

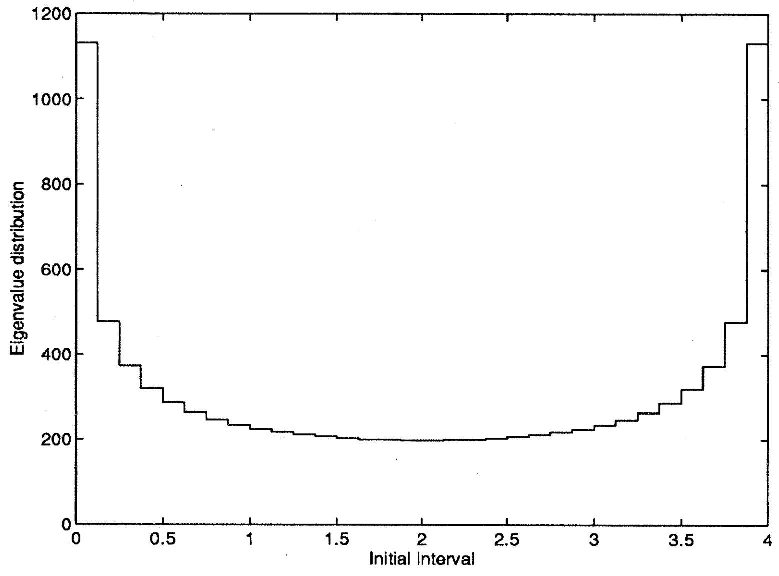

Figure 6: Spectral distribution of the matrix $[1, 2, 1]$ of order 10,000

processors. The figure shows that, in the static case, a few processors receive considerably more work than the rest. These heavily loaded processors continue their computations long after the remaining processors have finished their computations and become idle. As the total time is determined by the slowest processor, the dynamically scheduled program, which keeps the processor loads better balanced, finished faster. Indeed, the figure shows that the dynamic approach outperforms the static one by a factor of more than one half.

Note that processors are not synchronized in the dynamic approach. 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. Figure 8 shows that the advantages of the dynamic with respect to the static approach remain, independent of the matrix size.

The parallel bisection procedure applied to a random matrix illustrates a less irregular case than that represented by the matrix $[1, 2, 1]$. The random matrix has uniformly distributed random elements between -1 and 1 in the diagonal and subdiagonal. Figure 9 shows that the random matrix presents a more evenly distributed spectrum than does the $[1, 2, 1]$ matrix. However, Figure 10 shows that, in spite of the more even distribution of the work, the results obtained for the random matrix on the CM5 also favor the dynamic over the static approach. Because the random matrix eigenvalues are more evenly distributed than those of the $[1, 2, 1]$ matrix, the initial distribution of work does a better job in the former than it does in the latter. Dynamic load sharing improves the initial distribution of the work for the random matrix but not as dramatically as in the $[1, 2, 1]$ case. Figure 11 compares the dynamic and static approaches applied to random matrices of different sizes.

Thus, the experiments show that the dynamic approach outperforms the static one even in problems that are slightly irregular. In the next section we present another experiment that illustrates the PMESC approach to improving the efficiency of a program.

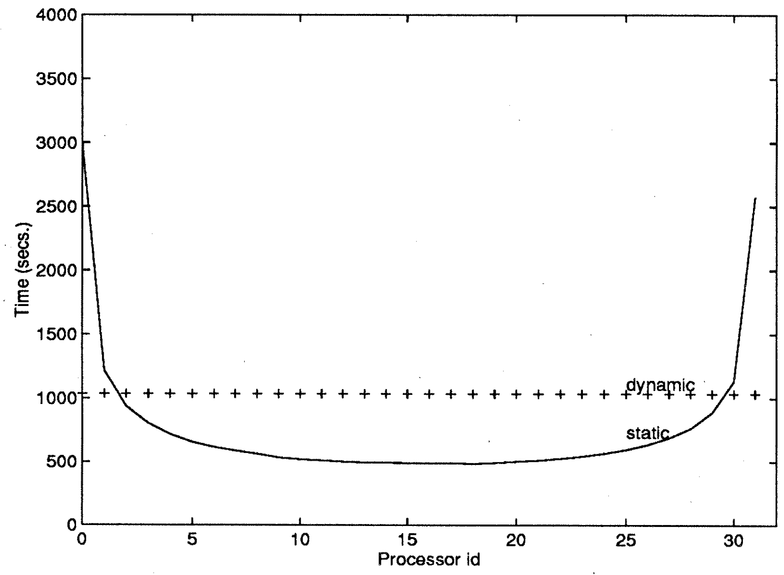

Figure 7: Execution times for each of the 32 processors under the static and dynamic approaches to solve bisection for the [1, 2, 1] matrix of size 10,000 (on the CM5.)

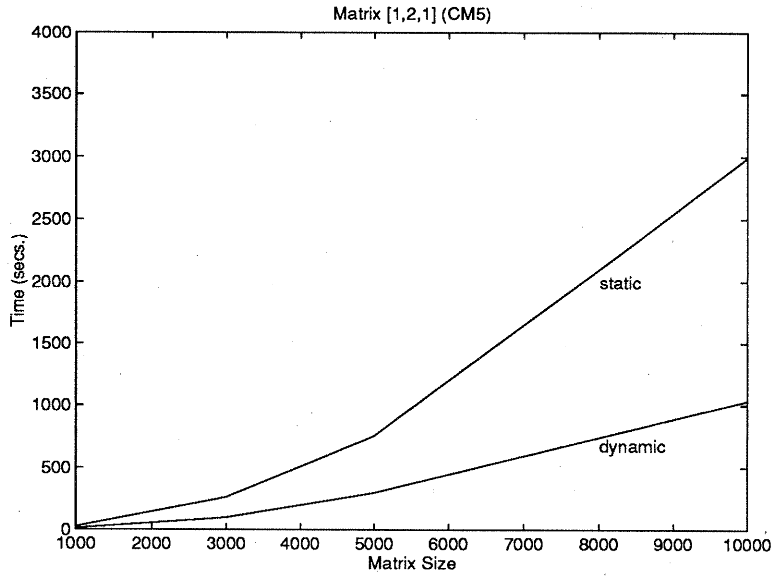

Figure 8: Total execution times for the static and dynamic approaches applied to [1, 2, 1] matrices of different sizes (on the CM5.)

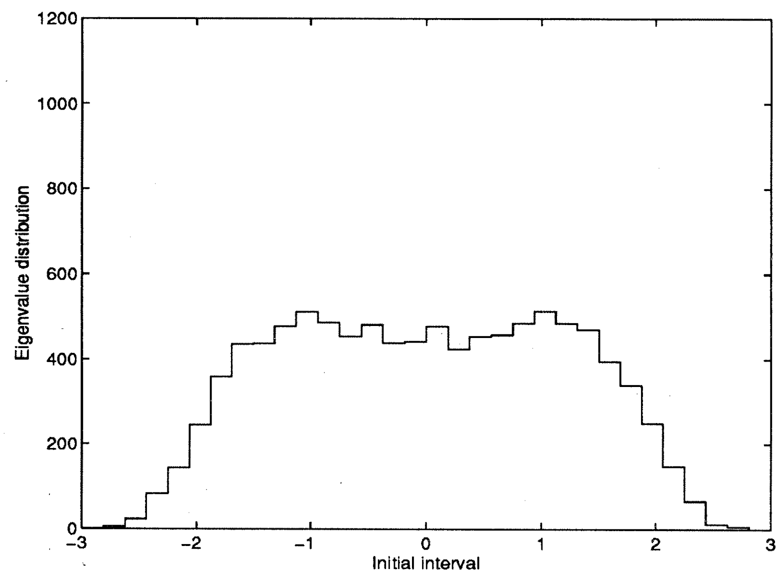

Figure 9: Spectral distribution of the random matrix of order 10,000

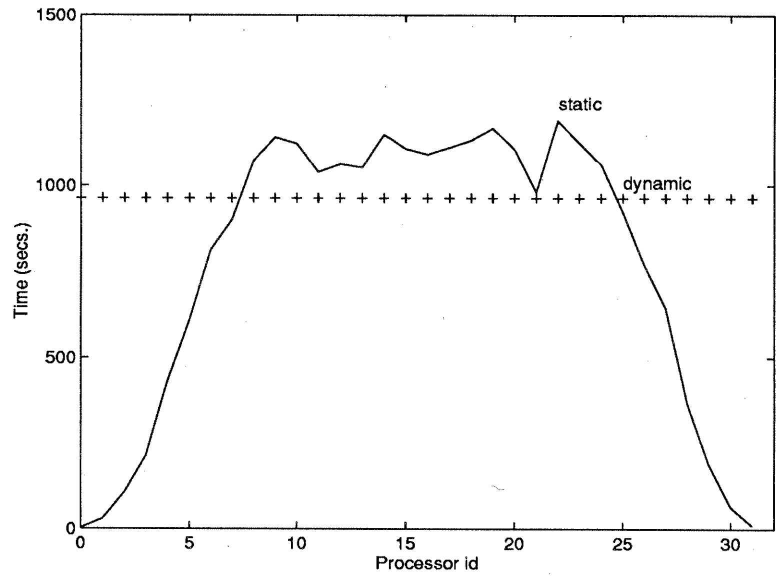

Figure 10: Execution time for each of the 32 processors under the static and dynamic approaches to solve bisection for the random matrix of size 10,000 (on the CM5.)

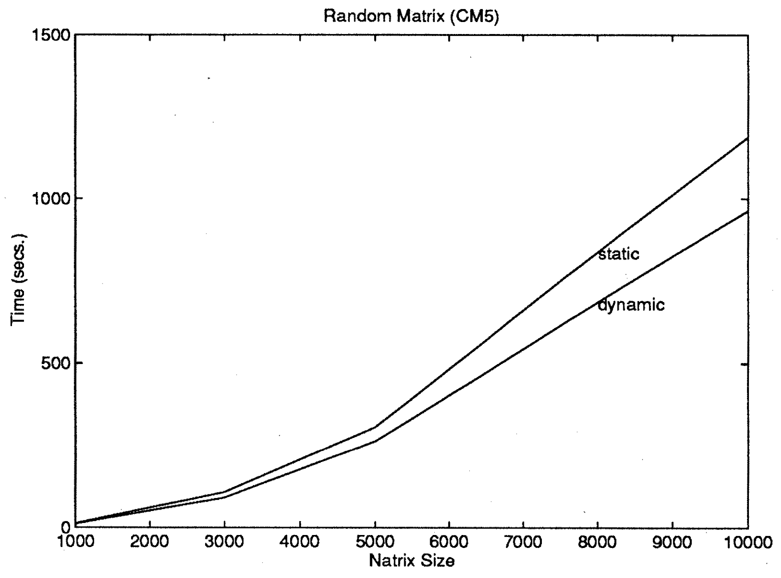

Figure 11: Total execution times for the static and dynamic approaches applied to random matrices of different sizes (on the CM5.)

6.2 Achieving Efficiency with PMESC

PMESC is based on the idea that no single strategy can be optimal for all of the problems or on all of 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 which ones are more suitable for the particular application at hand. Some decisions can be made a priori by studying the problem and its characteristics. However, some other decisions can only be made after running the program and analyzing the results. Thus, high efficiency is not an attribute that can be guaranteed the first time the user runs an application with PMESC, especially if little is known about the behavior of the application in advance.

PMESC provides a set of flags (`PMESC_flag`) that allow users to obtain useful information and, based upon that information, to make changes at runtime. For instance, a flag `PMESC_flag.req` set after a call to MAP_PrR indicates that the number of failed requests for work have reached a given `limit`. (`PMESC_flag` and `limit` are output and input parameters, respectively, of the Map routines.) The user may ignore this signal or make the changes necessary to correct the problem. A possible solution is to change the load sharing routine. Another solution is to make the processor wait a certain amount of time and then proceed with the execution. If the code is written using this information and considering all of the alternatives, the changes can be executed without having to modify, recompile, or rerun the code.

There are different programming decisions to make when trying to improve the efficiency of a PMESC application. Among them are the frequency of load sharing calls (controlled by the queue lower and upper bounds), the load sharing strategy (controlled by the load sharing routines), and the granularity (controlled by the size of the tasks.) We next discuss the various programming options.

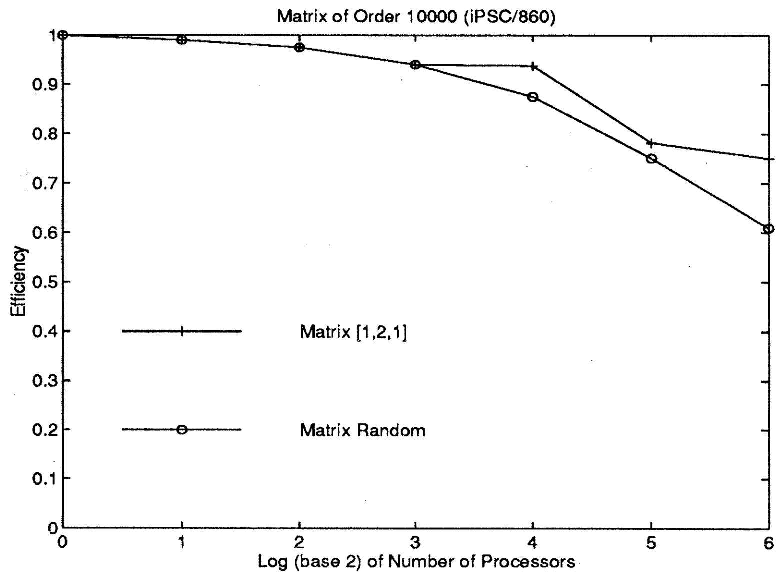

Figure 12: Efficiencies achieved with PMESC for the [1, 2, 1] and random matrices of order 10000 using the sender-initiated approach on the iPSC/860.

6.2.1 Changing the load sharing routine

To illustrate this issue, let us again consider the parallel bisection example. Figure 12 shows the efficiencies obtained as a result of applying PMESC using the Random sender-initiated load sharing approach to both matrices, [1, 2, 1] and random, on the iPSC/860. The experiments show efficiencies ranging from 60% in the worst case to 99% in the best case.

These efficiencies can be improved in the random matrix case (when using more than 16 processors) by changing the load sharing strategy from sender- to receiver-initiated. This alteration just amounts to changing the load sharing routine from MAP_RaS (for Random sender-initiated) to MAP_RaR (for Random receiver-initiated.) Figure 13 shows the efficiencies obtained on the iPSC/860, for the random matrix, with the sender- and receiver-initiated approaches.

However, the results obtained for the random matrix cannot be extrapolated to other cases. In fact, the Random sender-initiated approach yields better performance than the Random receiver-initiated one when applying the bisection procedure to the matrix [1, 2, 1]. Figure 14 compares the efficiencies achieved in the [1, 2, 1] case on the iPSC/860 using both approaches.

Observe that those results are predicted by the theory developed by Eager and his colleagues [18]. 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] matrix case corresponds to the lightly-loaded system. Most eigenvalues are concentrated in a few processors while the rest are underloaded. In contrast, the random matrix 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 expected then that the sender-initiated approach would be more suitable for the [1, 2, 1] matrix while the receiver-initiated one would be better for the random matrix.

Changing the load sharing routine is not the only resource the user has to improve the efficiency of the application. We next analyze other alternatives.

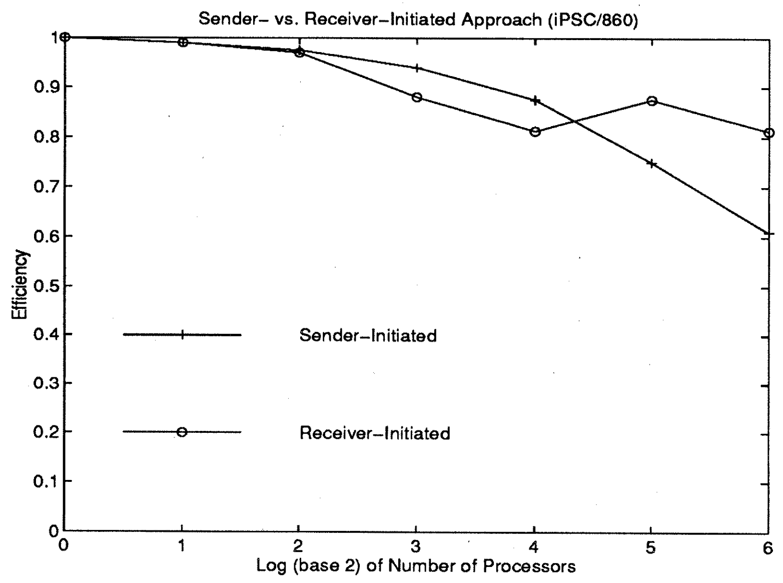

Figure 13: Efficiencies achieved with PMESC for the random matrix of order 10000 using the sender- and receiver-initiated approaches on the iPSC/860.

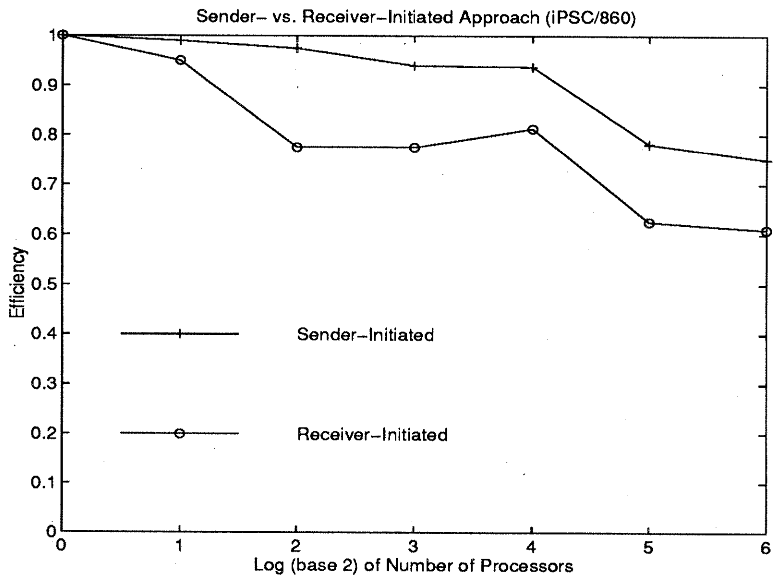

Figure 14: Efficiencies achieved for the $[1,2,1]$ matrix of order 10,000 using the sender- and receiver-initiated approaches on the iPSC/860.

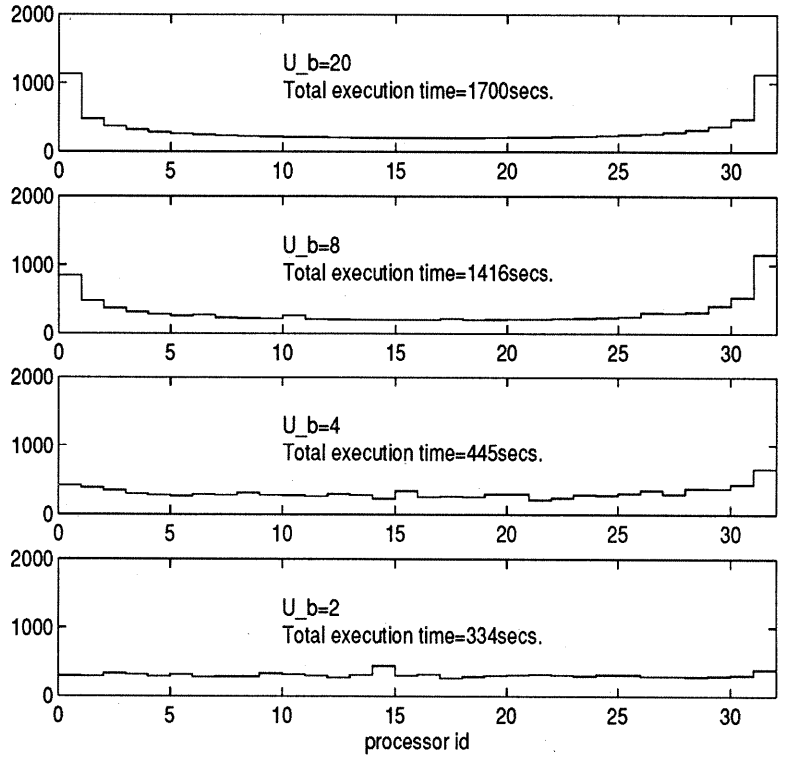

Figure 15: Eigenvalues computed by each of the 32 processors using different values of the parameter *U_b* when applying bisection to the $[1, 2, 1]$ matrix of order 10,000 (on the SP1.)

6.2.2 Changing the frequency of the load sharing calls

The performance of an application can also be improved by changing the frequency of the load sharing calls. We can easily control this frequency by varying the input parameters *l_b* and *U_b* that represent the lower and upper bounds of the queue length. To illustrate the tremendous impact of this simple change in the performance, we perform an experiment that consists of varying the upper bound *U_b* in the bisection example. Figure 15 shows the different distributions of work obtained when applying bisection to the matrix $[1, 2, 1]$ of order 10,000 using 32 processors on the IBM SP1 with values of *U_b* varying between 20 and 2. The value $U\_b = 20$ corresponds to the static approach in which no load redistribution occurs. Decreasing values of the parameter *U_b* results in more frequent calls to the load sharing routine. Thus, the lower the upper bound is, the better balanced the workload is.

6.2.3 Controlling granularity

The grain size is defined as the average computation required by the tasks. PMESC is a library for moderate to coarse grain sizes. Tasks that are too short create an unnecessary overhead that may harm the performance of the application. Grain sizes can be selected based on previous experience and results. For instance, in the parallel bisection example, we defined a task as the traversing of a single level down the tree of intervals. The computation time required for this action depends on the matrix size. If this partitioning creates a large number of short-

```
ParBisection (LIFOq *fq, Task *T)
{
  Task T1;
  double mid; \* midpoint of interval [a,b] *\
  int cm;      \* eigenvalue count at mid *\

  \* creates a Task for every nonempty interval it finds *\
  \* rather than storing all the Tasks, it continues with one until an eigenvalue is found *\
  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 (fq, &T1, size-of-task);
            T.b = mid;
            T.cb = cm;
        }
  }
}
```

Figure 16: Pseudo-code for the ParBisection function with larger grained tasks

lived units, then it is convenient to define a new task. A larger task may consist of applying bisection to a non-empty interval, then to one of its non-empty subintervals and so forth, until an eigenvalue is computed. This new task can be thought of as traversing a single path from a given node in the tree to one of the leaves. Figure 16 shows the ParBisection routine that computes the new tasks.

7 Conclusion and Future Directions in PMESC

The PMESC library is a tool designed to fulfill the needs of task-parallel application programmers. It provides users with the means to implement their problems using a dynamic and fully asynchronous approach without having to deal with the complexities associated with such an approach. The library is based on a parallel processing model in which explicit message passing and machine architecture are almost absent. This model facilitates the programmers' task because it matches the way that they envisage their applications.

Code written with PMESC is portable across any platform that supports PMESC. The library itself is portable, with minor or no changes, to any machine that supports MPI. Our numerical results show that PMESC also meets the goal of efficiency: the dynamic approach is efficient even when applied to slightly irregular problems. Furthermore, the experiments show that the library meets the goals of flexibility and ease of use by allowing users to tune their codes through small changes of parameters and routines. We illustrate how users can easily try different strategies for load sharing by just changing the mapping routine, how they can control the overhead associated with load sharing by altering the lower and upper bounds for the queue length, and how they can control the grain size by modifying the tasks that are executed. Moreover, an important feature of PMESC is that all of these changes can be performed at runtime.

The library is expected to grow in response to the needs of the users. Future additions include a hierarchical strategy for load balancing that implements a centralized approach at the lower level and a distributed approach at the higher level with changing machine size. This strategy will allow users to scale their applications not only up but also down. Some other issues to be considered include support for fault tolerance and support for obtaining system state information to make decisions regarding load balancing instead of the queue length criteria presently used.

References

[1] S. Arvindam, V. Kumar, and V. Rao. Floorplan optimization on multiprocessors. In *Proceedings of the 1989 International Conference on Computer Design*, pages 307–314. IEEE Computer Society, 1989.

[2] P. Banerjee. *Parallel algorithms for VLSI computer-aided design.* Prentice-Hall, 1994.

[3] R.H. Barlow and D.J. Evans. A parallel organization of the bisection algorithm. *The Computer Journal*, 22:267–69, 1977.

[4] W. Barth, R.S. Martin, and J.H. Wilkinson. Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection. In *Handbook for Automatic Computation: Linear Algebra*, pages 249–256. Springer Verlag, 1971.

[5] A. Beguelin, J. Arabe, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome user's guide version 1.0. Technical report, School of Computer Science, Carnegie Mellon University, 1996.

[6] H.J. Bernstein and M. Goldstein. Parallel implementation of bisection for the calculation of eigenvalues of tridiagonal symmetric matrices. *Computing*, 37:85–91, 1986.

[7] 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.

[8] D.Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA Ames Research Center, 1993.

[9] S. Crivelli. *A programming paradigm and library for distributed-memory computers.* PhD thesis, Dept. of Computer Science, University of Colorado at Boulder, 1995.

[10] S. Crivelli and E.R. Jessup. Asynchronous mechanisms for terminating task-parallel problems on distributed-memory computers. Paper in preparation, 1996.

[11] S.K. Das, M.C. Pinotti, and F. Sakar. Corrections to optimal and load balanced mapping of parallel priority queues in hypercubes. *IEEE Transactions on parallel and distributed systems*, 7(8), 1996.

[12] S.K. Das, M.C. Pinotti, and F. Sakar. Optimal and load balanced mapping of parallel priority queues in hypercubes. *IEEE Transactions on parallel and distributed systems*, 7(6), 1996.

[13] 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.

[14] E. de Doncker and J.A. Kapenga. Parallel systems and adaptive integration. *Parallel Computing, North-Holland*, 7:211–225, 1988.

[15] 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.

[16] Department of Computer Science, University of Illinois, Urbana-Champaign. *Charm 4.3 Programming Language Manual*, 1994.

[17] 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.

[18] 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.

[19] 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.

[20] D.G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *Computer*, pages 65–77, May 1990.

[21] R.A. Finkel. Large-grain parallelism – Three case studies. In D. Gannon L.H. Jamieson and R.J. Douglass, editors, *The Characteristics of Parallel Algorithms.* MIT Press, 1987.

[22] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[23] 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.

[24] W. Givens. Numerical computation of the characteristic values of a real symmetric matrix. Technical Report ORNL-1574, Oak Ridge National Laboratory, 1954.

[25] A.Y. Grama and V. Kumar. Parallel processing of discrete optimization problems: A survey. Technical report, Department of Computer Science, University of Minnesota, 1992.

[26] W. Gropp and B. Smith. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. To appear in Scientific Programming.

[27] A. Hac. Performance analysis of the priority queues for the load-building algorithms. *The Journal of Systems and Software*, 19(1), 1992.

[28] R. Hempel, H.-C Hoppe, U. Keller, and W. Krotz. Parmacs v6.0 specification, November 1993.

[29] I. Ipsen and E.R. Jessup. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM J. Sci. Stat. Comput.*, 11(2):203–229, 1990.

[30] L.V. Kale. A tutorial introduction to CHARM. Technical Report 92-6, Department of Computer Science, University of Illinois, 1992.

[31] S.R. Kohn and S.B. Baden. Irregular coarse-grain data parallelism. *Scientific Programming*, 5(3), 1996.

[32] 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.

[33] R-C Liu and S-D Wang. Performance modeling and analysis of load balancing policies with priority queueing. *The Journal of Systems and Software*, 20(2), 1993.

[34] S. Lo, B. Phillipe, and A. Sameh. A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 8:s155–s165, 1987.

[35] C.M. Pancake. A collaborative effort in parallel tool design. *Environments and Tools for Parallel Scientific Computing, (SIAM)*, 1995.

[36] Parasoft Corporation, Pasadena, California. *Express C User's Guide, Version 3.0*, 1990.

[37] V.N. Rao and V. Kumar. Parallel depth-first search, part I: Implementation. *International Journal of Parallel Programming*, 16(6):479–500, 1987.

[38] V.N. Rao and V. Kumar. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[39] J.R. Rice. A metalgorithm for adaptive quadrature. *Journal of the ACM*, 22(1):61–82, 1975.

[40] J.R. Rice. Parallel algorithms for adaptive quadrature III. Program correctness. *ACM Transactions on Mathematical Software*, 2(1):1–30, 1976.

[41] H.D. Shapiro. Increasing robustness in global adaptive quadrature through interval selection heuristics. *ACM Transactions on Mathematical Software*, 10(2):117–139, 1984.

[42] 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.

[43] A. Skjellum, A.P. Leung, S.G. Smith, R.D. Falgout, C.H. Still, and C.H. Baldwin. The Multicomputer Toolbox - First-generation scalable libraries. In *Proceedings of HICSS-27*, pages 644–654. IEEE Computer Society Press, 1994. HICSS-27 Minitrack on Tools and Languages for Transportable Parallel Applications.

[44] S.L. Smith. *Adaptive asynchronous parallel algorithms in distributed computation.* PhD thesis, Department of Computer Science, University of Colorado at Boulder, 1991.

[45] S.L. Smith. Performance analysis of dynamic scheduling techniques for irregularly structured computation. In *Software for Parallel Computation.* Springer-Verlag in cooperation with NATO Scientific Affairs Division, 1993.

[46] 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.

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