THE PMESC PROGRAMMING LIBRARY FOR DISTRIBUTED-MEMORY MIMD COMPUTERS *

S. CRIVELLI† AND E. R. JESSUP‡

Abstract.
Efficient programming of task-parallel problems, where the number and execution times of the computational tasks can vary unpredictably, demands an asynchronous and adaptive approach. In this sort of approach, however, such fundamental programming issues as load sharing, data sharing, and termination detection can present difficult programming problems. This paper presents the PMESC library for managing task-parallel problems on distributed-memory MIMD computers within the context of the SPMD (Single Program, Multiple Data) programming model. PMESC offers support for all of the application-independent programming issues involved in SPMD task-parallel computation in a portable and efficient way while still allowing users to customize their codes. Because different problems may require different strategies to achieve good performance, PMESC is based on a straightforward model in which different building blocks can be easily put together and changed to accommodate the particular needs of the different applications. The library provides an interface that allows users to program a virtual machine and thereby ignore the details associated with message passing and machine architecture. These features make PMESC accessible to a wide variety of users.

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 [12, 47, 48], combinatorial optimization [2, 4, 21, 26, 39, 40, 51], adaptive numerical quadrature [11, 15, 16, 17, 41, 42, 43], and the bisection method for computing eigenvalues of symmetric matrices [5, 7, 11, 31, 37]. 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.

* This research was supported by National Science Foundation grant ACS–9357812, Department of Energy grant DE–FG03–97ER25325, and the Intel Corporation.

† Physical Biosciences, Life Sciences, and NERSC Divisions, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (crivelli@global.lbl.gov)

‡ Department of Computer Science, University of Colorado, Boulder CO 80309–0430. (jessup@cs.colorado.edu)

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 an SPMD programming model in which each processor executes a copy of the same program. However, unlike other systems designed for data parallelism, PMESC allows a dynamic and asynchronous approach that lets processors work independently as much as possible while still maintaining an effective use of the computational resources. Thus, PMESC provides support for the SPMD task-parallel programming model by allowing unrelated portions of the computation to take place concurrently and allowing processors executing them to communicate in order to share some useful information.

PMESC meets the goal of flexibility by providing building blocks built on top of MPI (the Message Passing Interface [22].) 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 [28, 32] and adopted by its successor Charm++ [33], 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 to inexperienced programmers. It assists the less knowledgeable 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.) In this paper, we present computational results on the Intel iPSC/860 and the IBM SP1 for the parallel bisection method for computing eigenvalues of a symmetric tridiagonal matrix and for the traveling salesman problem (TSP). Performance achieved with these portable, dynamic applications as programmed with PMESC show considerable gains with respect to the statically scheduled approach without significant programming effort. For similar results on other application problems, see [11].

The paper is organized as follows. In section 2, we describe the PMESC paradigm that allows us to recognize the different components in the computation and to treat them as separate program modules. In section 3, we introduce the library based on the PMESC paradigm. In section 4, we define the method of bisection and the TSP and present both in terms of the PMESC paradigm. In section 5, we present computational results and discuss performance issues. In section 6, we compare and contrast PMESC with related packages. Finally, in section 7, we summarize the main features of PMESC and discuss some future work.

2. 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. The identification of those 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 components in the computation involving different programming issues.

The **Partition-Map-Embed-Solve-Communicate** (PMESC) paradigm is composed of five components bearing those names. The **Partition** component splits a problem into subproblems, puts them into queues, and handles all subsequent access to those queues. The **Map** component assigns (and reassigns as necessary) those subproblems to the set of processors interconnected by some convenient virtual topology. The **Embed** component embeds the virtual topology or topologies used for the application into the actual machine architecture. The **Solve** component executes the subproblems. The **Communicate** component takes care of the interprocessor communication necessary to exchange information. The P, M, E, S, and C components can appear in any order and number. In the PMESC model, the identification of the components is what matters, not their sequence.

Figure 1 shows the PMESC components in a task-parallel problem. First comes the Partition component 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 tasks into longer ones that are more appropriate for distributed-memory computers.

The next component is the Map component, which distributes the units of work among the processors of the computer itself or of a convenient virtual machine. For example, in the tree search examples described in section 5, a convenient virtual topology is a tree. Whenever a virtual machine is used, the next component of the computation is to Embed its topology into the topology of 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 component. Thus, Partition encompasses two stages in the computation–one that depends on the application and one that is general. For the latter, 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 [35]. 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 component represents the execution of those tasks. Processors may also need to share some information. Any exchange of data required for solution of a task is represented by the Communicate component. The exchange of tasks themselves is always carried out during a Map component.

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 given 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 load sharing mechanisms to transfer some work to or to get some work from another processor. This work redistribution is an exchange of tasks and, therefore, corresponds to the Map component. The cycle repeats until all of the local queues are empty.

The PMESC paradigm provides the programmer with a simple model for task-parallelism. It allows the user to separate the components that depend on the application from those that can be provided in a library. In the next section, we introduce the library that supports this model. Because the model is straightforward so will be the library use.

3. The PMESC Library . PMESC is a programming tool designed to offer a set of implementation strategies to SPMD, task-parallel application programmers. It provides a high-level environment based on the PMESC paradigm that focuses on the

```
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
```

FIG. 1. *PMESC framework for task-parallel problems*

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 the PMESC library 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) [38, 50] of being usable as a stand-alone 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.

3.1. PMESC Design and Implementation. The PMESC interface is organized around two basic elements: tasks and processors. Tasks are the units of work into which the computation is divided. Processors are the processing nodes of a distributed-memory MIMD computer. They execute only one task at a time, even though they may have other tasks in their queues. (The other fundamental element of a PMESC program, communication, is an action that occurs between processors and so is not considered part of the interface.)

The current implementation of the PMESC library is intended for SPMD programming. Each processor has a copy of the executable code that includes the tasks. Thus, the transfer of a task from one processor to another only involves transfer of the particular data associated with the execution of the particular task. These data can be represented by a C structure. Once a task starts running on a processor, it cannot be preempted by other tasks or assigned to other processors.

Each task can be independent of the others (as illustrated in the bisection example), or it can depend on the results from other tasks (as illustrated in the TSP example.) PMESC handles the latter case by allowing tasks to be assigned different priorities and by allowing processors to exchange results. Because the PMESC tasks are pieces of the total computation to be performed, they do not involve communication. Thus, communication is performed between processors rather than between tasks.

The PMESC library is a set of routines organized in modules or groups. Each routine is classified according to the programming component in the PMESC paradigm for which it provides support. The routines that handle the task queue structure are said to 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 constitute the Embed module, and the routines that take care of handling interprocessor communication make up the Communicate module.

3.1.1. 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. PMESC offers queuing strategies for both prioritized and unprioritized queues. In the former, the highest priority task is the first out. In the latter, the last task queued is the first out (LIFO).

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.

3.1.2. Dynamic Load Sharing. The dynamic load sharing routines of the Map component are responsible for the redistribution of tasks among processors so that the system resources can be utilized efficiently throughout the course of the computation. By load sharing we mean the reallocation of tasks to keep the computational workload distributed, rather than *evenly* distributed, among the processors.

PMESC provides simple dynamic load sharing mechanisms that collect very small amounts of information and use it in very simple ways. Eager et al. [19] show that such strategies yield performance close to that expected from more complex approaches. The PMESC routines are distributed, meaning that each processor controls its own load. Distributed algorithms are more scalable than those involving any degree of centralization [35].

There are three load sharing strategies in PMESC. 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 process 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 Ring strategy may be useful for those platforms in which the cost of communicating between distant processors can be high such as clusters of workstations including separate subnets.

The load sharing strategies also come in two different versions: sender- and receiver-initiated. In the sender-initiated approach, heavily-loaded processors send out some work without its 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].

3.1.3. 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 Map routine for termination detection based on the prefix algorithm [11]. It uses a virtual tree of processors, but it can use any other virtual topology that is appropriate for a combine operation. (When a new virtual architecture is added, corresponding Embed routines must be developed for the library.) Each processor keeps a count of the messages it has sent in order to request work minus the messages it has 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 [3] for a complete discussion of this and other termination detection strategies.

3.1.4. Asynchronous Sharing of Information. In task-parallel computations, processors do not synchronize. Thus, information sharing is carried out in a Communication component by means of pseudo-global variables. 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 [11]. 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 includes 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.

3.1.5. 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 (in the Embed component) allows one to program on a virtual machine, thereby hiding architectural details from the application. The efficiency of a program can be maintained by using a virtual architecture that maps efficiently onto the real architecture. 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.

4. The PMESC Programming Examples . In this section, we discuss two examples that illustrate two different scenarios for PMESC use. The first shows a case in which tasks are independent and interprocessor communication is only needed to exchange work. The second shows a situation in which the partial results are relevant to all the tasks so need to be periodically exchanged among the processors.

4.0.6. A Program for the Parallel Bisection Example. An example of an irregular task-parallel problem is given by the computation of the eigenvalues of of a symmetric, tridiagonal matrix $\mathcal{T}$ by the bisection procedure [6, 25]. 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 splits an interval containing at least one eigenvalue in half and 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. At that point, the subinterval midpoint is accepted 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.

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`.

```
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);
            }
  }
}
```

Fig. 2. *Pseudo-code for the sequential bisection procedure*

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 likely lead to different workloads on different processors.

Figure 3 shows the task-parallel version of bisection using PMESC. This pseudocode follows the outline presented in Figure 1. The user is required to define the parallel actions called tasks and to write the Solve function. The rest of the computation is handled by PMESC library routines.

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 that determines when to stop the subdivision.

The Solve procedure `ParBisection` is based on the sequential bisection procedure `SeqBisection` which applies a sequential bisection routine to the current task. However, unlike `SeqBisection`, `ParBisection` does not call itself recursively but rather 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.

The program itself begins with a call to the function PMESC_Init to specify 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 communication because communication is not necessary. The lower `l_b` and upper `U_b` bounds for the queue are entered. Next comes a call to the PMESC `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 components 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 component.

At each step of the main program loop, every processor calls the PMESC Partition function `DEQUEUE` to retrieve a task from the queue and then calls the Solve function `ParBisection` to execute that task. Note that the `DEQUEUE` function takes as an argument a pointer to the queue (in this case, the structure LIFOq.) In parallel bisection, there is no interprocessor exchange of data and so no calls to PMESC Communication routines. When the queue length is greater than the upper bound,

```
#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);
  }
}
```

FIG. 3. *Pseudo-code for the main procedure corresponding to the parallel bisection example using a distributed queue approach*

the program calls the load sharing (PMESC Map) routine `MAP_RaR` (for Random Receiver-initiated) with input parameter set to BUSY in order to transfer tasks to other processors. When the queue length is less than the lower bound, the program calls `MAP_RaR` with input parameter set to IDLE in order to receive tasks from other processors. If no tasks are available, `MAP_RaR` automatically calls another Map routine to check for termination.

```
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);
    }
  }
}
```

FIG. 4. *Pseudo-code for the ParBisection function*

4.0.7. A Program for the Traveling Salesman Problem (TSP). Another typical example of a task-parallel problem is given by the branch-and-bound procedure for the TSP. 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 [26, 53]. 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 are given higher priority because they are more likely to produce a solution.

The parallel implementation again follows the lines of the PMESC framework presented in Figure 1. Thus, we only discuss here those aspects of the TSP code that differ from the bisection example. Figure 5 shows the corresponding code.

In this problem, a task consists of the branch and bound procedures to a given partial solution. The task priority is the cost of the partial solution obtained by the bounding procedure. 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. The PMESC implementation of the initial Partition and Map components (the `InitPart_and_Map` function) makes a designated processor, ROOT, execute the original task, create new tasks, and assign a task to each one of the processors. Because tasks in the TSP are given different priorities, the task 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 the number of tasks completed by that processor reaches a given threshold. Other ways to deal with the balancing of prioritized tasks are discussed in [11, 13, 14, 29, 36, 44]. Load balancing in the TSP is carried out by the PMESC Map function `MAP_RaS`.

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 component of the PMESC paradigm. For this component, 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 component.

5. Numerical Results and Performance . In the static implementation of a parallel program, processors create their local queues of tasks, execute all of 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 there is little interprocessor communication, the overhead associated with the static approach is minimal and, therefore, it provides a baseline for comparison against PMESC. In the PMESC approach, in addition to 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 operations can be considerable. The performance of the PMESC program depends not only on how the problem is divided among the processors but also on how the overhead time compares to the time to do arithmetic. The value of using PMESC is thus determined by how the time spent to do the computation dynamically compares to the time spent

```
#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;
        }
  }
}
```

FIG. 5. *Pseudo-code for the main procedure corresponding to the parallel traveling salesman example using a distributed queue approach*

to do it using static assignment of tasks to processors. It also depends on how easy it is for the user of PMESC to achieve good performance.

In this section, we present the results of computational experiments to show how the static and dynamic times compare for problems of various degrees of irregularity. We further show how the performance of the PMESC program is influenced by the user's choice of various program parameters. We use the bisection example for our experiments because it can easily be adapted to represent different situations by just

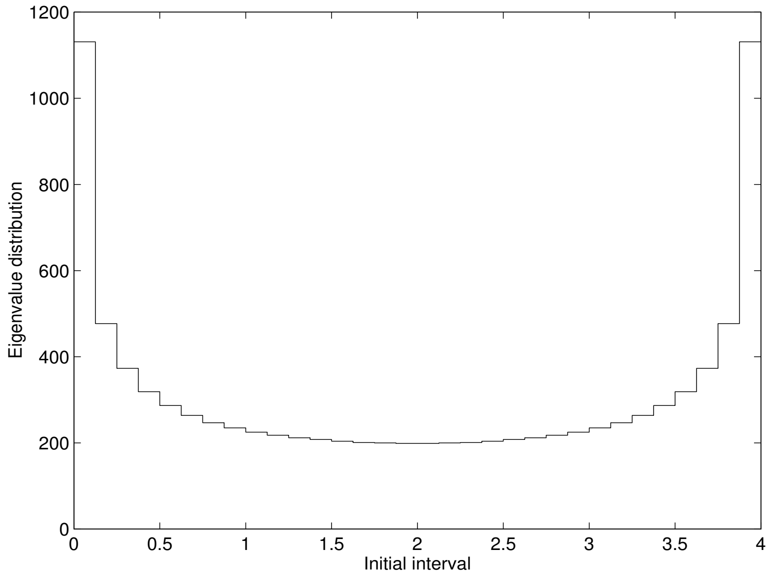

FIG. 6. *Spectral distribution of the matrix* $[1, 2, 1]$ *of order 10,000*

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 [11]. The dynamic bisection program uses the PMESC routines described in Section 4. The static program is constructed using nonblocking MPI communication routines [27, 49].

5.1. Problem irregularity vs. performance . The symmetric, tridiagonal test matrix $[1, 2, 1]$ having 2's on the diagonal and 1's on the off-diagonals presents a highly irregular problem. 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, are increasingly concentrated at the endpoints of the interval $[0, 4]$ as n increases. The eigenvalue distribution of $[1, 2, 1]$ is shown in Figure 6 for the case $n = 10^4$. This distribution presents a difficult partitioning problem because most of the work is likely to be assigned to a few processors no matter how the initial interval is distributed.

Figure 7 shows the execution time of each processor using the dynamic PMESC and static MPI codes to compute all eigenvalues of the matrix [1,2,1] for $n = 10^4$

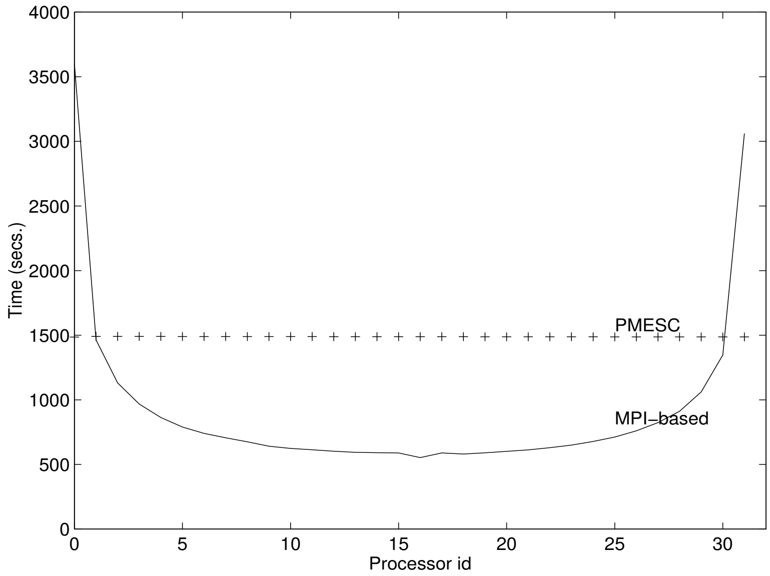

FIG. 7. *Execution times for each of the 32 processors under the dynamic PMESC and MPI-based static approaches to solve bisection for the* $[1, 2, 1]$ *matrix of size 10,000 (on the iPSC/860.)*

on the iPSC/860. In this case, the initial interval is partitioned into p equally-sized subintervals with one subinterval assigned to each of p processors. The timings of the static programs show that a few processors do receive considerably more work than the rest and that these heavily loaded processors continue their computations long after the remaining processors have finished their computations and become idle. As the total time for any parallel program is determined by the time required by the slowest processor, the dynamically scheduled program, which keeps the processor loads better balanced, finishes faster. Indeed, the figure shows that the dynamic approach outperforms the static one by a factor of nearly three despite the greater overhead of the PMESC program.

Note that processors are not synchronized in the dynamic approach. However, because they do not terminate until they receive the termination message which broadcast by the processor that detects termination, their times usually differ by only a few seconds. Figure 8 shows that the advantages of PMESC with respect to the static approach remain for a range of problem sizes.

Because dynamic approaches are most powerful in the context of large, irregular problems, the more interesting test is a comparison of the two strategies for a fairly

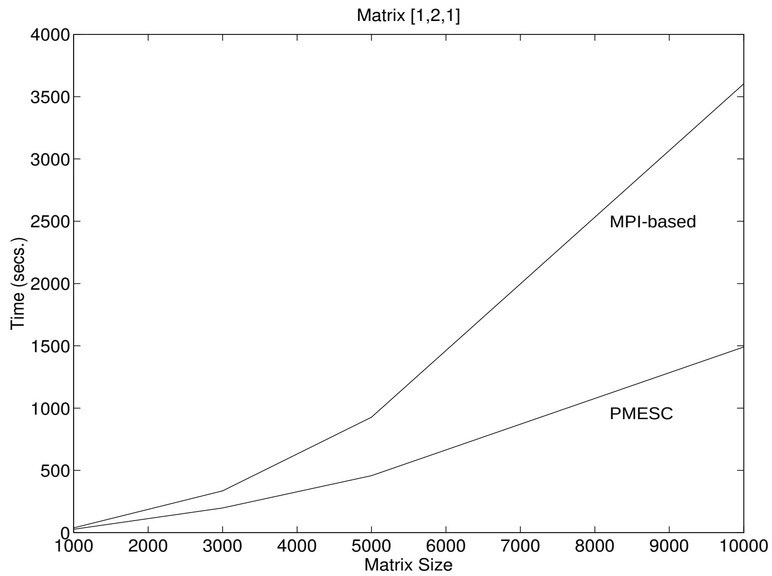

FIG. 8. *Total execution times for the dynamic PMESC and static MPI-based approaches applied to* $[1, 2, 1]$ *matrices of different sizes (on the iPSC/860.)*

regular problem. Our next test case is a random symmetric tridiagonal matrix with uniformly distributed random elements between -1 and 1. Figure 9 shows that the random matrix presents a more evenly distributed spectrum than does the $[1, 2, 1]$ matrix. Nonetheless, Figure 10 shows that the results obtained for the random matrix on the iPSC/860 also favor PMESC over the static approach. As expected, the gain is not as great in this case because an even partitioning of the initial interval provides the static program with a reasonably balanced load. Although dynamic load sharing can do less to improve the initial distribution of work than it could in the more irregular case, Figure 11 shows that PMESC retains its advantage, even for smaller problem sizes.

Thus, our experiments show that dynamic PMESC outperforms the static MPI code for both problems with a wide range of irregularity. In the next section, we show how decisions the user makes about the program can alter its performance.

5.2. Achieving Efficiency with PMESC . PMESC is based on the idea that no single strategy can be optimal for all problems or on all machines. The PMESC approach to coping with this concept consists of providing a set of reasonably easy-to-use and easy-to-change strategies so that the user can decide which ones are most

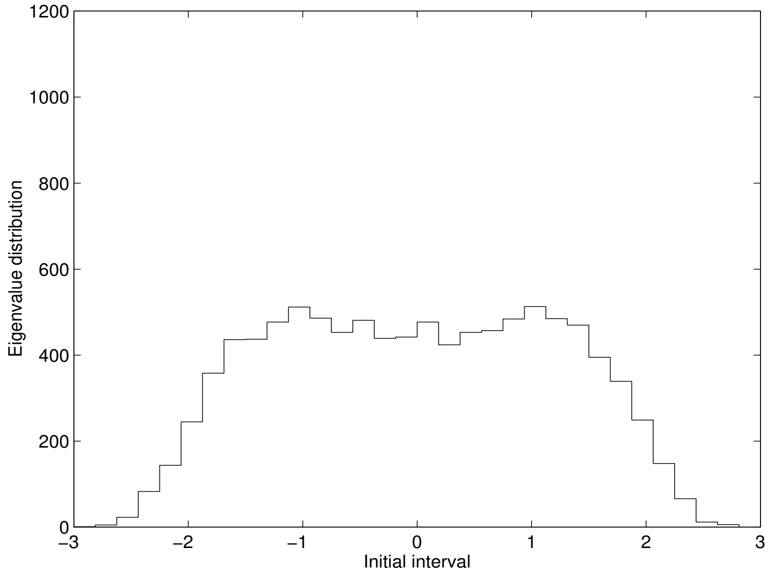

Fig. 9. *Spectral distribution of the random matrix of order 10,000*

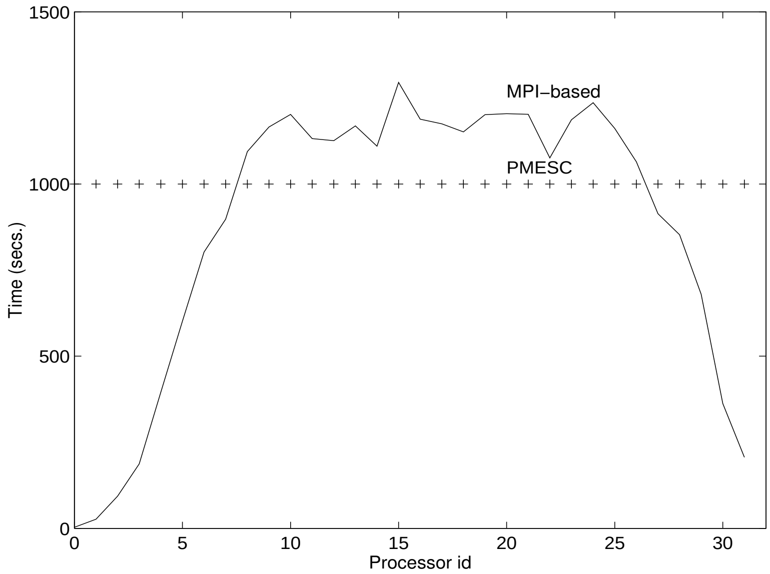

Fig. 10. *Execution time for each of the 32 processors under the dynamic PMESC and static MPI-based approaches to solve bisection for the random matrix of size 10,000 (on the iPSC/860.)*

suitable for the particular application at hand. While some decisions can be made a priori by studying the problem and its characteristics, 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

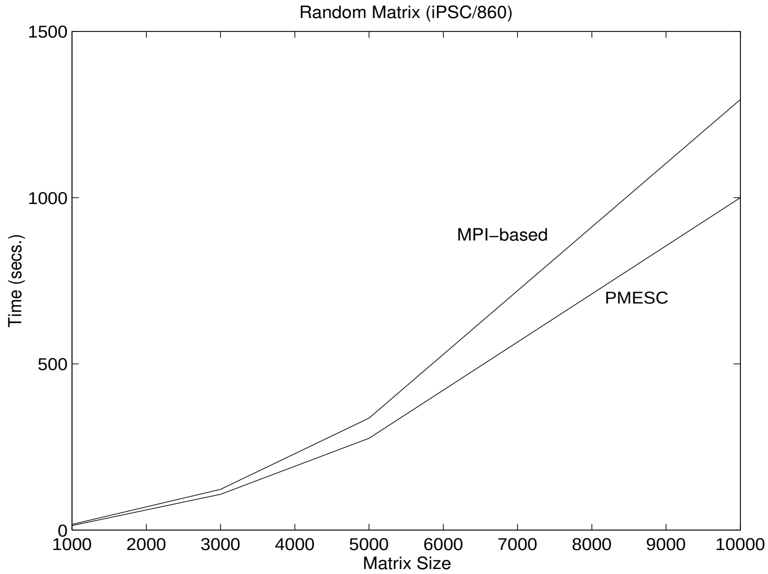

FIG. 11. *Total execution times for the dynamic PMESC and static MPI-based approaches applied to random matrices of different sizes (on the iPSC/860.)*

with PMESC, especially if little is known about the behavior of the application in advance.

In this section, we examine the effects of three possible programming decisions. They are the load sharing strategy (controlled by the load sharing routines), the frequency of load sharing calls (controlled by the queue lower and upper bounds), and the granularity (controlled by the size of the tasks.)

5.2.1. Changing the load sharing routine. To illustrate the effects of the various load sharing strategies, we again consider the parallel bisection example. Figure 12 shows the efficiencies of a dynamic PMESC program that uses the Random sender-initiated load sharing approach for both test 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. When using more than 16 processors, these efficiencies can be improved in the random matrix case by changing the load sharing strategy from sender- to receiver-initiated. Figure 13 shows the results of this change. (This alteration just amounts to changing the load sharing routine from MAP_RaS (for Random sender-initiated) to MAP_RaR (for Random receiver-initiated) in the pseudocode of Figure 3.) The results obtained for the random matrix, however, cannot

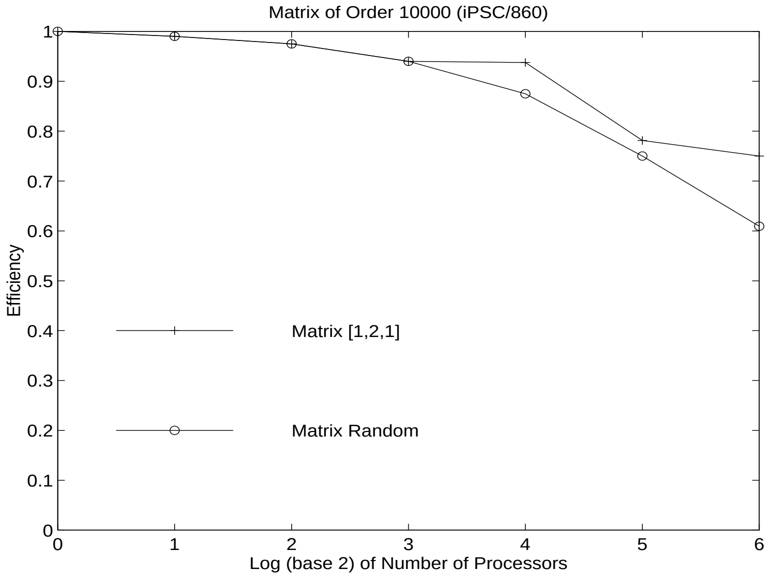

FIG. 12. *Efficiencies achieved with PMESC for the* $[1, 2, 1]$ *and random matrices of order 10000 using the sender-initiated approach on the iPSC/860.*

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 obtained on the iPSC/860.

Observe that these 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 as most eigenvalues are concentrated in a few processors while the rest of the processors 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.

5.2.2. 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 that

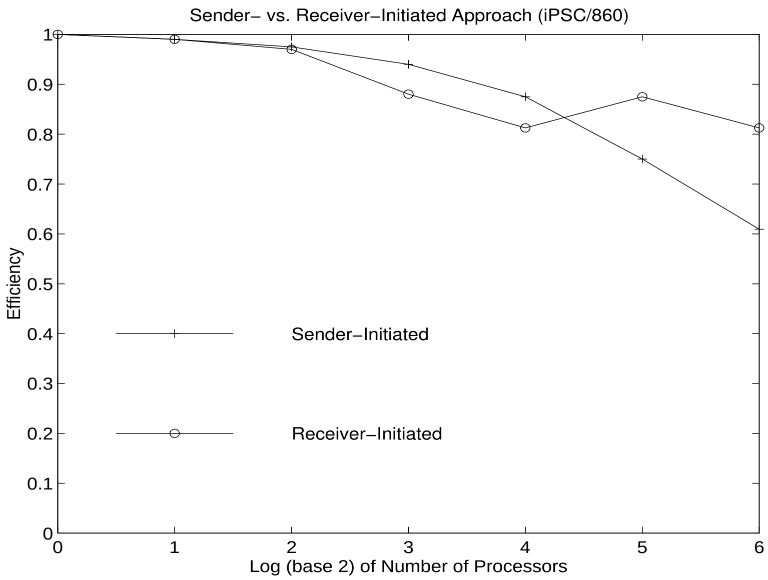

FIG. 13. *Efficiencies achieved with PMESC for the random matrix of order 10000 using the sender- and receiver-initiated approaches on the iPSC/860.*

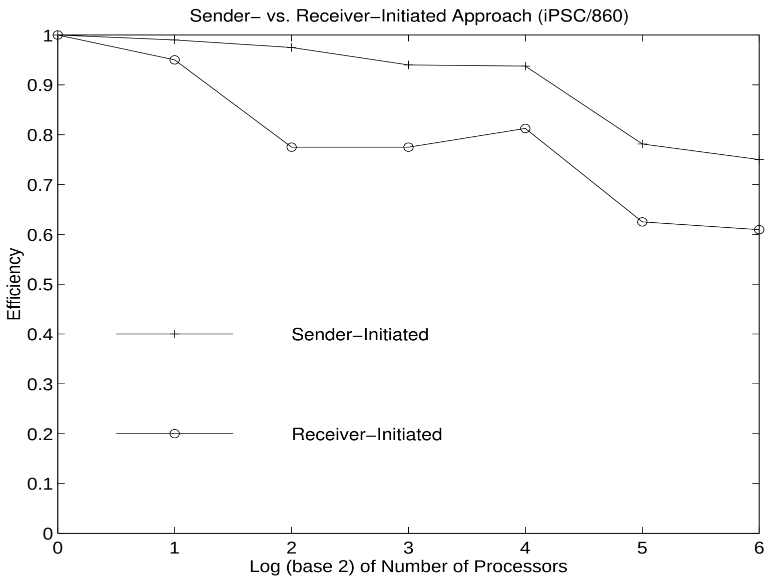

FIG. 14. *Efficiencies achieved for the* $[1, 2, 1]$ *matrix of order 10,000 using the sender- and receiver-initiated approaches on the iPSC/860.*

represent the lower and upper bounds of the queue length (`l_b` and `U_b` in Figure 3.) To illustrate the tremendous impact of this simple change in the performance, we perform an experiment that consists of varying the queue length upper bound in the bisection example. Figure 15 shows the different distributions of work obtained when

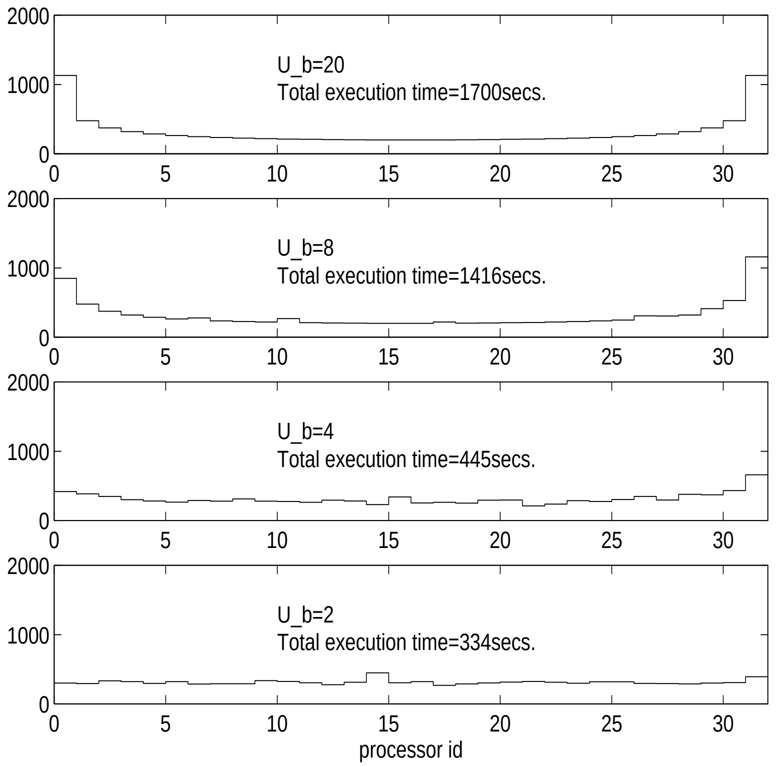

FIG. 15. *Eigenvalues computed by each of the 32 processors using different values of the upper bound limit (U_b) when applying bisection to the* $[1, 2, 1]$ *matrix of order 10,000 (on the SP1.)*

applying bisection to the matrix $[1, 2, 1]$ of order 10,000 using 32 processors on the IBM SP1 with upper bound values between 20 and 2. The value 20 corresponds to the static approach in which no load redistribution occurs. Decreasing values of that parameter result in more frequent calls to the load sharing routine. Thus, the lower the upper bound is, the better balanced the workload despite the higher the overhead.

The important lesson of this test is that the user can expect an improvement over the static time when using the dynamic approach for any choice of queue length. In this case, best performance is attained by the most dynamic program.

5.2.3. Controlling granularity. The grain size of a problem is defined as the average computation time 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 according to previous experience and results. For instance, in the parallel bisection example, we

```
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;
        }
  }
}
```

FIG. 16. *Pseudo-code for the ParBisection function with larger grained tasks*

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-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.

6. Related Work . Although there has been a great deal of effort directed at providing software for irregular data-parallel computations [1, 8, 10, 24, 46, 30, 34, 20, 45], up to this point, only a few tools have been developed specifically to address irregular task-parallel ones. Among them, Charm [28, 32] and Charm++ [33] are the closest to PMESC. Charm and Charm++ are parallel programming systems that supports explicitly parallel C- and C++-based languages, respectively, 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 and Charm++ allow 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 and Charm++ the explicit definition of the tasks and the choices of strategies to use for some programming issues. However, Charm and Charm++ do not provide complete control to users. Rather, they leave 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.

PMESC addresses a SPMD model of irregular task-parallelism in which a large problem can be decomposed into coarse grained subproblems that can be computed in parallel. Other systems support a Multiple-Program, Multiple-Data (MPMD) task-parallel approach that allows programs to dynamically create multiple threads. Examples of such systems include Multipol [9, 52] and Nexus [23]. Multipol is a library of distributed data structures that supports multiple threads per processor for hiding communication latency. Multipol threads are small, lightweight computations that are executed atomically. Multipol threads are invoked either locally or remotely by another processor. Thus, unlike PMESC, Multipol allows the computation rather than the data to migrate in order to keep the load balanced. Multipol provides a global memory model through the use of shared data structures. However, these virtually shared structures are physically distributed among the processors and the user is responsible for their distribution.

Nexus is another runtime system that supports multiple threads of control, dynamic processor acquisition, and a global memory model. Communication and synchronization are between threads, rather than processors. Nexus is designed to support heterogeneous computing and it is built on top of standard thread packages.

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 only 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] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CMU-CS-95-137, School of Computer Science, Carnegie Mellon University, 1995.

[2] 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.

[3] A. Baker, S. Crivelli, and E.R. Jessup. Asynchronous mechanisms for terminating task-parallel problems on distributed-memory computers. Paper in preparation, 1999.

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

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

[6] 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.

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

[8] R. Buttler and E. Lusk. User's guide to the P4 parallel programming system. http://www-fp.mcs.anl.gov/ lusk/p4/index.html, August 14, 1998.

[9] S. Chakrabarti, E. Deprit, E.-J. Im, J. Jones, A. Krishnamurthy, C.-P. Wen, and K. Yelick. Multipol: a distributed data structure library. Technical Report CSD-95-979, Department of Computer Science, University of California at Berkeley, 1995.

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

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

[12] S. Crivelli, R. Byrd, E. Eskow, R. Schnabel, R. Yu, T. Phillips, and T. Head-Gordon. A global optimization strategy for predicting protein tertiary structure: α-helical proteins. Submitted to Proteins: Structure, Function, and Genetics, 1998.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[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] *Express C User's Guide, Version 3.0*. Parasoft Corporation, Pasadena, California, 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] Message Passing Interface Forum. MPI2: A message passing interface standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[23] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Par. and Distr. Computing*, 37:70, 1996.

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

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

[26] A.Y. Grama, V. Kumar, and Panos Pardalos. *Parallel Processing of Discrete Optimization Problems: A Survey*. John Wiley and Sons, 1993.

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

[28] A. Gursoy and V. Kale. *Charm 4.3 Programming Language Manual*. Department of Computer Science, University of Illinois, Urbana-Champaign, 1994.

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

[30] R. Hempel, H.-C Hoppe, U. Keller, and W. Krotz. *PARMACS V6.0 Specification, Pallas*

Internal Report, November 1993.

[31] 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.

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

[33] V. Kale et al. *The Charm++ (4.5) Programming Language Manual*. Department of Computer Science, University of Illinois, Urbana-Champaign, 1999.

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

[35] 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.

[36] 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.

[37] 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.

[38] C.M. Pancake. A collaborative effort in parallel tool design. In J. Dongarra, editor, *Environments and Tools for Parallel Scientific Computing*. SIAM, 1995.

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

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

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

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

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

[44] 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.

[45] 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.

[46] Barry Smith and William Gropp. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. *Scientific Programming*, 5:329–336, 1996.

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

[48] 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.

[49] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI The Complete Reference*. MIT Press, 1996.

[50] The Parallel Tools Consortium. http://www.ptools.org, 1999.

[51] 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.

[52] C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, and K. Yelick. Runtime support for portable distributed data structures. In *Proceedings of the Workshop on languages, compilers and runtime systems for scalable computers*. Kluwer Academic Publishers, May 1995.

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