A Load Balancing Framework for Adaptive and Asynchronous Applications

Kevin Barker, Andrey Chernikov, Nikos Chrisochoides, and Keshav Pingali

Abstract—This paper describes the design of a flexible load balancing framework and runtime software system for supporting the development of adaptive applications on distributed-memory parallel computers. The runtime system supports a global namespace, transparent object migration, automatic message forwarding and routing, and automatic load balancing. These features can be used at the discretion of the application developer in order to simplify program development and to eliminate complex bookkeeping associated with mobile data objects. An evaluation of this system in the context of a three-dimensional tetrahedral advancing front parallel mesh generator shows that overall runtime improvements of 15 percent compared to common stop-and-repartition load balancing methods, 30 percent compared to explicit intrusive load balancing methods, and 42 percent compared to no load balancing are possible on large processor configurations. At the same time, the overheads attributable to the runtime system are a fraction of 1 percent of the total runtime. The parallel advancing front method is a coarse-grained and highly adaptive application and therefore exercises all of the features of the runtime system.

Index Terms—Dynamic load balancing, adaptive and irregular applications, runtime support software, multithreading, message passing, parallel, distributed, and grid computing, scientific computing, parallel mesh generation.

1 Introduction

Application codes in computational science and engineering are concerned largely with simulating physical phenomena that are governed by partial differential equations (PDEs). A large proportion of these codes are *adaptive* in the sense that the field solution of the PDE drives changes to the geometry of the domain on which the PDE is defined. For example, crack growth in macroscopic structures under stress may be governed by the equations of elasticity and plasticity, which can be solved using the finite-element method. As the crack grows, the geometry of the problem obviously changes, so the domain needs to be remeshed locally after every crack growth step. For efficiency, the mesh generation and refinement must be done in parallel, but it is a very irregular process which, due to adaptivity, is difficult or impossible to efficiently load-balance statically.

By design, message-passing libraries such as MPI [33] and PVM [6] provide only point-to-point communication and global reduction operations. Consequently, the development of adaptive applications, such as three-dimensional (3D) finite-element mesh generation, on a bare-bones messaging layer can be a daunting task. In response to this need, the scientific computing community has developed application-specific runtime libraries and software systems [5], [16], [23], [27], [28], [38]. These systems are designed to support the development of parallel multiphase computations in which computationally intensive phases are separated by computations such as global error estimation that require global synchronization. Load-balancing is accomplished by dynamically repartitioning the data after the global synchronization phases [36]. Throughout this paper, we call this traditional way of load balancing the *stop-and-repartition* method.

In our experience, the stop-and-repartition approach is not well-suited for applications such as adaptive mesh generation and refinement because the synchronization overhead can overwhelm the benefits of improved load balance. This problem is exacerbated as the number of processors in the parallel system grows. In this paper, we describe the Implicit Load Balancing (ILB) component of the Parallel Runtime Environment for Multicomputer Applications (PREMA), which is an alternative approach to traditional load balancing libraries and it is based on

1. single-sided communication similar to that provided by Active Messages [35],
2. a global name space,
3. transparent object migration and automatic message forwarding for mobile objects, and
4. a framework which allows for the easy and efficient implementation of customized dynamic load balancing algorithms,

along with a suite of commonly used dynamic load balancing strategies such as *Diffusion* [14], *Gradient* [25], and *Multilist* Scheduling [37].

Mobile Schedulable Objects (Section 3.1) permit the application to be load-balanced dynamically without sacrificing locality of reference—when a computational task is migrated from one processor to another, the data it references can be moved along with it. There is no global synchronization phase; data is not repartitioned in bulk, but instead is migrated across processors in response to computational load balancing. We have several man-years of experience in using this system and our experience suggests that the programming abstractions provided by

- *K. Barker, A. Chernikov, and N. Chrisochoides are with the Computer Science Department, College of William and Mary, Williamsburg, VA 23185. E-mail: {kbarker1, ancher, nikos}@cs.wm.edu.*
- *K. Pingali is with the Computer Science Department, Cornell University, Ithaca, NY 14853-3801. E-mail: pingali@cs.cornell.edu.*

Manuscript received 9 Sept. 2002; revised 26 Mar. 2003; accepted 14 May 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 117306.

this system improve programmer productivity. For example, a first-year graduate student with little parallel programming experience was able to parallelize a sequential 3D mesh generator with about four months of part-time effort [10].

The rest of this paper is organized as follows: Section 2 discusses the software foundation upon which the ILB is constructed and the programming model. Section 3 discusses the load balancing (i.e., ILB) framework and its components. Section 4 discusses the parallel 3D advancing front mesh refinement program that we use to evaluate ILB performance. Section 5 then analyzes the results of our experiments. Section 6 puts our research in the context of work done in the community, while Section 7 concludes with some future plans.

2 Software Framework and Programming Model

The PREMA system is organized into layers according to the principle of *separation of concerns*. The Data Movement and Control Substrate (DMCS) [4] serves as the foundation and isolates the higher levels of the runtime system from the idiosyncrasies of the underlying hardware, operating systems, and low-level communication subsystems. As such, DMCS is the only software layer which needs to be ported when migrating the PREMA system from one platform to another. We have therefore designed DMCS with portability as a primary goal, and have successfully ported our software to Linux, Solaris, and Windows platforms, using LAM/MPI, MPICH, and MPIPro for low-level communication (for detailed performance data, see [11]). Also, it should be noted that, while we have used MPI as our low-level communication substrate, this is not required.

At the same time, DMCS ensures that a consistent programming model, built on single-sided communication and remote procedure invocation, is presented to the higher system and application software layers. DMCS's *remote service requests*, communication operations which invoke user-defined handler functions on target processors, form the basis for both data migration and computation invocation in the PREMA system. In addition, DMCS messages may be associated with tags (as with MPI [17]), allowing the application or runtime system to break incoming remote service requests into categories and receive messages according to the category to which it belongs. This ability is particularly useful when the runtime system must separate load balancing messages from application messages, as will be discussed later.

The Mobile Object Layer (MOL) [12] extends DMCS by providing a global namespace and the *mobile object* construct. Mobile objects are application-defined data objects and are not restricted to exist in contiguous memory. A mobile object may be referenced by any processor in the parallel system by using its associated *mobile pointer*, which is a system-wide unique identifier. The MOL's *message* operation extends the DMCS remote service request by allowing applications to send a message to invoke computation to the location of a mobile object, regardless of where it is in the parallel system. In this way, applications can deal directly with data objects without the tedious bookkeeping associated with maintaining up-to-date knowledge of each data object's current location.

The MOL combines message forwarding, an efficient distributed directory structure, and message sequence numbers to ensure that messages arrive in order at their target mobile objects, even if the target is in the process of migration (note that messages are only ordered with respect to their senders; two messages sent from two separate sources may arrive in arbitrary order). When a processor sends a message to a mobile object, it sends the message to the (possibly out-of-date) location given by its local directory. If this location turns out to be incorrect, the MOL forwards the message toward the real location.[1] Message forwarding will trigger an update of the directory on the processor on which the message originated; in this way, only those processors which show an explicit interest in the location of a mobile object are made aware of the object's current location, reducing directory update overheads.

The PREMA system encourages a data-centric programming model in which application data is broken into N chunks or work units, where $N \gg P$ and P is the number of available processors. This is known as *over-decomposition* [6]. The value chosen for N can have an impact on overall application performance; enough work units must be present for there to be available work for migration during load balancing.[2] A more detailed discussion of the trade offs between work unit size, number of work units, and processor count is presented in Section 5.

Parallel computation then consists of a series of operations on particular data objects, where data is accessed through PREMA's *message* operation regardless of whether the data is located on the local or a remote processor. In this model, a typical section of code which iterates over an array of method invocations on local objects will be replaced by a loop sending messages to possibly remote objects. The application therefore has a consistent mechanism to access data, be it local or remote, while the runtime system is able to bind pending computation to corresponding data so the two may be migrated together during load balancing.

3 Load Balancing Library

PREMA's ILB component library is built using the framework provided by DMCS and the MOL and provides automatic and transparent dynamic data (and, implicitly, computation) migration in response to perceived runtime workload imbalances. The ILB is designed not just as a single load balancing algorithm or family of algorithms, but as a framework which supports the rapid development and deployment of algorithms, allowing researchers to experiment without modification of existing application code. This supports our observation that there is not a single load

1. While it is proven that the message will make progress toward its destination, it is possible to devise a pathological case in which the mobile object will migrate ahead of the forwarded message. However, this points to an error in the application or load balancing algorithm and not to a flaw within the MOL.

2. Our results in this paper demonstrate that, when N is chosen to be as small as a factor of five greater than P, load balancing quality is excellent. However, the value of N should be determined with respect to other parameters like size and computation of work units, inter and intranode bandwidth, cache performance, message polling versus message interrupts, and scheduling policies. This is a difficult optimization problem with multiple constraints which we will be able to study using the software infrastructure we present in this paper.

balancing method which is optimal on all platforms for all problems. However, we have also implemented several of the more common load balancing methods, such as Diffusion [14], Work-Stealing [7], and a variation on Multilist [37] scheduling.

The ILB's architecture is designed to fulfill two objectives. First, we want to provide an evolutionary migration path for parallel applications written using the MOL. Specifically, for applications written using mobile objects and the MOL's messaging mechanism, making use of the ILB's functionality should involve minimal changes to existing application code. This means that the programming model and programming interface provided by the MOL and ILB should closely mirror one another.

Our second objective is to allow as much flexibility as possible in the range of load balancing policies implementable by the ILB library. To achieve this, we have isolated the application from the load balancer's decision making Scheduler module with a simple and flexible interface. Scheduler modules may be easily implemented and exchanged without propagating changes to application code, allowing for quick experimentation during development.

3.1 Schedulable Objects

For load balancing, it is necessary to extend the MOL's mobile object concept in order to associate application-defined data with its associated pending computation; migrating data from processor to processor will thereby implicitly migrate computation in order to balance workload. This coupling is implemented via Schedulable Objects.

In order to gather the information necessary to make accurate load balancing and scheduling decisions, the ILB interacts with Schedulable Objects through several call-back routines defined by the application and registered with the runtime system. Routines are used to gather load information, as well as calculate the difficulty in migrating an object. This information is used, along with a third call-back routine, to calculate an *affinity map*, which specifies the processor which would benefit most from the migration of a particular Schedulable Object. Through these call-back routines, the application is able to prioritize the factors that influence load balancing decisions, such as minimizing data migration, maximizing the smoothness of the workload distribution, or maintaining colocality of objects that may share data dependencies.

Because Schedulable Objects are application-defined and are not restricted to be of a certain size or exist in contiguous memory, it is impossible for the runtime system to know how they should be migrated from processor to processor. Therefore, three additional call-back routines are necessary: one to pack the Schedulable Object into a contiguous buffer prior to transport, one to unpack the Schedulable Object from a buffer, and one to return the size of the Schedulable Object in bytes.

3.2 Scheduling

The ILB library is designed so that applications can have the ability to quickly and easily create and adopt new load balancing strategies with minimal changes to existing application code. It is the Scheduler module that provides this flexibility. The Scheduler module encapsulates the load balancing decision making and data migration functionality into a single entity that is isolated from the rest of the PREMA system and user application by a well-defined simple interface. While its primary purpose is to schedule the execution and migration of Schedulable Objects during runtime, the exact policies used to make scheduling decisions are left to the individual Scheduler implementation.

In order to give some idea as to the breadth of scheduling policies that are implementable from within our framework, we have implemented several examples of well-known methods. These fall within three categories: Diffusion [14], Master-Worker [7], and Multilist [37]. Diffusion scheduling divides the processor pool into small, overlapping neighborhoods and begins when the local work load falls below a predefined watermark. The underloaded processor requests work levels from its neighbors and is then able to determine how many work units to request from each. Because neighborhoods overlap, work will eventually diffuse throughout the system. In addition, it is possible for neighborhoods to change once no neighbors are able to contribute work. An optimization is known as work-stealing and assigns a single neighbor to each processor. In this case, processors skip the data gathering phase.

Master-Worker scheduling policies begin with all work units located on a single "master" processor. This processor does not perform computation for the application, but acts as a server to send work units to worker processors. All other processors are workers, and request work from the master once the local workload falls below a predefined watermark.

Finally, we have implemented a Prioritized Multilist (PML) scheduler. Each of the P processors maintains P physical lists which contain the local work units sorted according to the values contained in the affinity maps dynamically calculated for that work unit (Section 3.1). In addition to the P physical lists, each processor maintains a priority list of P elements. The ith entry in processor p's priority list denotes the value of the pth entry in the affinity map of the Schedulable Object at the head of the pth physical list on processor i. In our implementation, we execute local work units until the available number falls below a predefined watermark. At this point, the priority list is consulted in order to determine which processor should act as the source for load transfer.

Several issues must be dealt with in order to efficiently implement these schedulers. For instance, the choice of the "low watermark" plays an important role in the performance of the algorithm, as does determining how many work units to migrate from one processor to another during each load balancing iteration. In addition, it is often difficult for developers to accurately predict appropriate values for watermarks as the optimal choice depends not only on application characteristics, but also on the system on which the application is executing. System characteristics such as network latency and bandwidth, as well as the number of available processors, can influence algorithm/watermark pairing.

3.3 Multithreading

Although the scheduling algorithms discussed so far are well known in the literature, we found several problems when attempting to adapt them to our specific target application types. The 3D advancing front mesh generation program we employ makes use of a relatively small number of coarse grained work units. In other words, $\frac{N}{P}$ (where N is the number of work units and P is the number of

processors) is often less than 10 and each work unit can take from several minutes to an hour to execute. Second, it is often the case, as it is in this instance, that an application being parallelized cannot be modified to include polling operations at strategic locations, either due to code complexity, licensing, or the fact that only precompiled libraries are made available. This is problematic because, as we have previously described, polling is necessary in order to receive and process both application messages and system-generated load balancing requests and information update messages. These factors together mean that it is often the case that load balancing requests and information are out-of-date by the time they are processed, leading the runtime system to make poor load balancing decisions.

We have found that a multithreaded approach nicely solves this problem. Our strategy is to spawn a "polling thread" whenever a long-running work unit is executing. This thread polls the network for load balancing messages at some predefined interval and allows each processor to maintain up-to-date information regarding system status, as well as satisfy any pending load balancing requests in a timely manner. Once the work unit has finished execution, the polling thread is killed and only a single application thread remains.

However, there are several requirements that must be met. First, the application code itself must remain single threaded and, therefore, the multiple threads must be confined to the runtime system only. As a result, any polling operations from within the polling thread must service only load balancer system messages and never application messages. Tagging the load balancer's messages with a system-reserved tag allows us to accomplish this. Second, care must be taken to ensure that the overhead associated with the polling thread does not dominate the overall runtime. This can lead to the case in which the load balancing is of good quality, but the overall runtime actually increases. We do this by adjusting the interval at which the polling thread "wakes up" and checks for messages. We have found that polling roughly once a second provides good quality for the load balancing without a negative impact on application performance.

4 Load Balancing Parallel Mesh Generation

Parallel mesh generation is an important adaptive application and a good candidate to demonstrate the effectiveness of PREMA's load balancing component. To this end, we have implemented a 3D parallel advancing front technique (PAFT) method presented in [29].

The key steps of the PAFT are: 1) Generate the dual graph of the submeshes and partition the graph into P subgraphs,[3] 2) load each subgraph in parallel into the available processors and create Schedulable Objects from each of its vertices (i.e., submeshes), and 3) apply an advancing front mesh generation algorithm [26] on every submesh. At the end of these steps, the mesh is ready for parallel finite analysis. Said et al. [29] present a similar approach; however, they use a centralized master/worker load balancing model.

3. This is a process known as over-decomposition, which was described in Section 2. For all of our experiments, we partition the mesh into $N = 640$ submeshes. We build the dual-graph of the submeshes and, after the partitioning of the dual-graph, we generated P subgraphs with N/P vertices per subgraph. Each submesh is a three-dimensional region of the original domain; the submeshes do not overlap and their union is equal to the original domain.

During the advancing front mesh generation phase, a dynamic load balancing algorithm is utilized to migrate work units in response to load imbalance caused by different levels of refinement in the geometry. In order to evaluate the effectiveness of the PREMA system, we have designed the PAFT mesh generation program to control data migration either *explicitly* or leave the decision making and data migration up to PREMA for *implicit* load balancing.

Explicit work stealing [7] begins with the PAFT mesher maintaining a queue of local submeshes pending refinement. During refinement, each processor performs three steps: local region refinement, load balancing, and polling the network. After refining local regions, a processor checks to see if its pending work load has fallen below some predetermined threshold. If so, the processor enters a *work-seeking* state in which it requests work from other processors in a round-robin fashion. The processor then polls the network for responses to these work requests. Iteration through these steps continues until there is no work left awaiting execution.

A second method which falls into the explicit load balancing category is to use a *stop-and-repartition* scheme using parallel Metis [30]. As with the other load balancing methods under consideration, load balancing begins when a processor's workload falls below a predetermined threshold.[4] At this point, all processors in the parallel system synchronize and exchange workload information. Metis' *LDiffusion* algorithm is then used to perform the decision making for migration work units in order to restore load balance.

Implicit load balancing places the load balancing decision making and data migration burden on the runtime system. The algorithms employed by the ILB load balancing module have been described in Section 3.2. After the initial distribution of submeshes and creation of Schedulable Objects, a single message is sent to each region invoking the mesh refinement stage of the algorithm.

5 Performance Evaluation

We begin with an examination of the effects of over-decomposition on application performance and on the overheads incurred by the runtime system. Three parameters play a role in this study: The number of work units created by the decomposition (N), the number of processors available (P), and the weights of the individual work units. We have developed a synthetic benchmark program which begins by dispersing work units to the available processors. Computation is then invoked via PREMA's messaging mechanism. Once computation involving a data object is complete, a notification is sent to the root processor; once all notifications have been received, the application terminates. Implicit load balancing is utilized during runtime when necessary.

Fig. 1a depicts the time spent inside the runtime system as both N and P vary, excluding the initialization and termination stages of the program.[5] In all processor configurations (ranging from 8 to 128 processors), PREMA overhead decreases as the number of work units (N) increases until a minimum is reached. After this point,

4. The thresholds used for all load balancing methods are identical.

5. We have chosen to exclude these stages because they are highly application dependent.

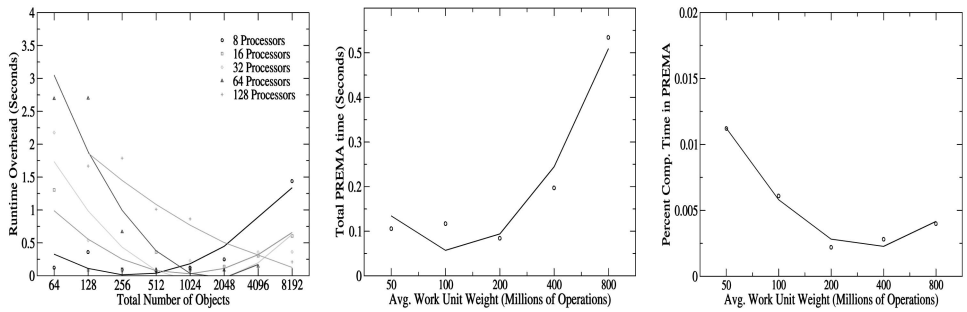

Fig. 1. Effects of varying processor count (P), work unit count (N), and work unit weight on PREMA overheads: (a) The runtime overhead in terms of seconds as the total number of work units increases on varying numbers of processors. (b) The total time spent in the PREMA system as the average weight of each work unit varies for 2,048 work units. (c) The total time spent in the PREMA system as a percentage of computation time as the average weight of each work unit varies for 2,048 work units.

PREMA overhead grows with N. This indicates there is a point at which further over-decomposition is actually detrimental to overall performance; in our study, optimal performance was achieved with roughly 32 work units initially allocated for each processor.

Fig. 1b and Fig. 1c show the results of varying the computational workload in each work unit (note that this in turn affects the overall computation performed by the program). Varying the work units from roughly 50 million operations to 800 million operations results in an increase in the amount of time spent within the runtime system by slightly less than half a second. However, the ratio between this time and the computation time actually decreases. This indicates that the ILB load balancing system is robust given changes in work unit sizes.[6]

We now evaluate PREMA's performance with the PAFT application in terms of three metrics: 1) overall application runtime, 2) the quality of the workload distribution (minimizing the standard deviation of mesh refinement times), and 3) overhead attributable to the runtime system itself.

Fig. 2a compares the overall runtimes of all load balancing methods (as well as no load balancing) on several processor configurations.[7] On 32 processors, the ILB's Diffusion Scheduler module provides an improvement of 43 percent over no load balancing, 12 percent over load balancing with stop-and-repartition methods, and roughly 13 percent over explicit load balancing. On 64 processors, these numbers are 39 percent, 9 percent, and 20 percent, while, on 128 processors, they are 42 percent, 15 percent, and 30 percent. In Fig. 2b, we compare the results of implicit load balancing with and without multithreading. Particularly in the cases of work stealing and diffusion, having multiple system threads can provide performance increases of over 40 percent. These numbers represent a significant overall performance increase over methods that are commonly in use today.

Our second metric is the quality of the workload distribution. In Fig. 2c, we see a processor-by-processor breakdown of the PAFT program's performance on a 128 processor system with no load balancing. Most of the computation is clustered within processors toward the "front" of the system (processors with IDs 0 through 31), leaving ample opportunity for effective load balancing. Fig. 2d provides the results for load balancing using a stop-and-repartition algorithm. Workload is distributed fairly evenly across the processors (mesh refinement time has a standard deviation of roughly 51, compared with 305 for no load balancing), however, from the figure, we can see that synchronization and partition computation causes a large amount of overhead (up to roughly 11 percent of the overall runtime) which can be avoided.

The last row of Fig. 2 shows the results of two explicit load balancing methods. Fig. 2e, the Master/Worker method, is the more successful of the two in terms of data distribution quality, balancing work load with a standard deviation of roughly 83. However, initialization costs impose a large penalty which hurts overall runtime. This can be avoided by using a "pipelining" method to read in the data objects, effectively overlapping I/O with computation on the worker nodes. However, judging from the workload distribution, there is still room for performance improvements. In addition, Master/Worker type algorithms suffer from another shortcoming: With iterative applications which may have several phases of mesh refinement, the algorithm will need to be "reset" at the beginning of each phase, meaning all data objects will have to be gathered on the Master processor.

Fig. 2f shows the results of load balancing with the explicit application-managed work stealing method. This method suffers from the same problems as the single-threaded implicit methods in that polling cannot occur during the execution of a work unit and, therefore, load balancing information arrives late and out-of-date. An underloaded processor may request work from a remote node that is currently executing a work unit and will not receive a reply until that work unit is finished. Consequently, most nodes spend a great deal of time idle. The ultimate result is that not many work units migrate to the underloaded nodes, as evidenced by a mesh refinement standard deviation of 246.

Fig. 3 contains processor-by-processor breakdowns for both single and multithreaded implicit load balancing methods implemented using PREMA's ILB framework. Having multiple threads in the runtime system clearly provides a performance benefit; with the diffusion method (row 2), overall runtime decreased by 41 percent compared to the single-threaded counterpart. Similar results can be seen in the case of work stealing (row 1). In addition, workload distribution quality is increased, compared with

6. We have used the Work Stealing Scheduler implementation for these tests; many details are dependent upon the scheduler implementation.

7. The test platform on which we conduct our experiments consists of 333MHz Ultra SPARC 2i machines, connected by Fast Ethernet and utilizing the LAM [24] implementation of MPI.

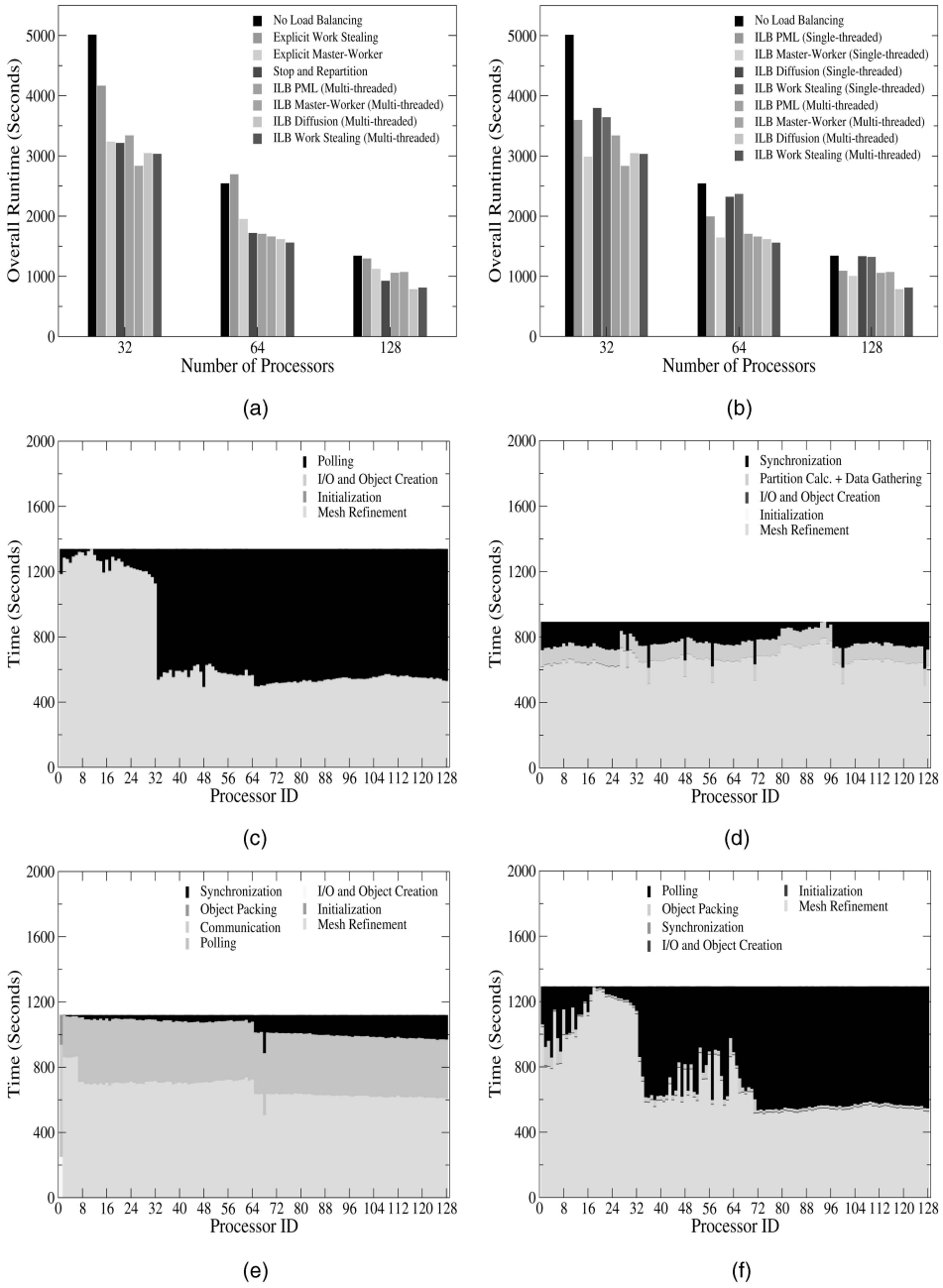

Fig. 2. Overall performance (in (a) and (b)) of 11 different load balancing methods implemented within PREMA using ILB and traditional libraries like Metis. (c) Breakdown data for no load balancing. (d) Stop-and-repartition load balancing. Two explicit methods: (e) Master/worker and (f) work-stealing on 128 processors.

both repartitioning and explicit load balancing methods. The standard deviation of mesh refinement time with the multithreaded implicit work stealing method is roughly 27, while, with diffusion, it drops to 25.

Note that less dramatic results are obtained with the multilist (PML) and the Master/Worker scheduling methods. In the case of the PML, this is most likely due to the large number of information and update messages needed. Because processors are not divided into small "neighborhoods," each processor receives load updates and may receive work requests from every processor in the system, potentially leading to a glut of system messages and performance degradation. In the case of the Master/Worker policy, workload is well balanced in the single-threaded case; the addition of multiple system threads does nothing to improve this. However, it does allow the Master processor to also act as a Worker.

Finally, we show that the overhead imposed by the runtime system is small and does not negatively impact overall application performance. Table 1 summarizes the overheads caused by PREMA according to several different categories. In all cases, overhead contributes significantly less than 1 percent to the overall runtime.

6 Related Work

Because of the irregular and adaptive nature of the applications we wish to optimize, we will restrict our

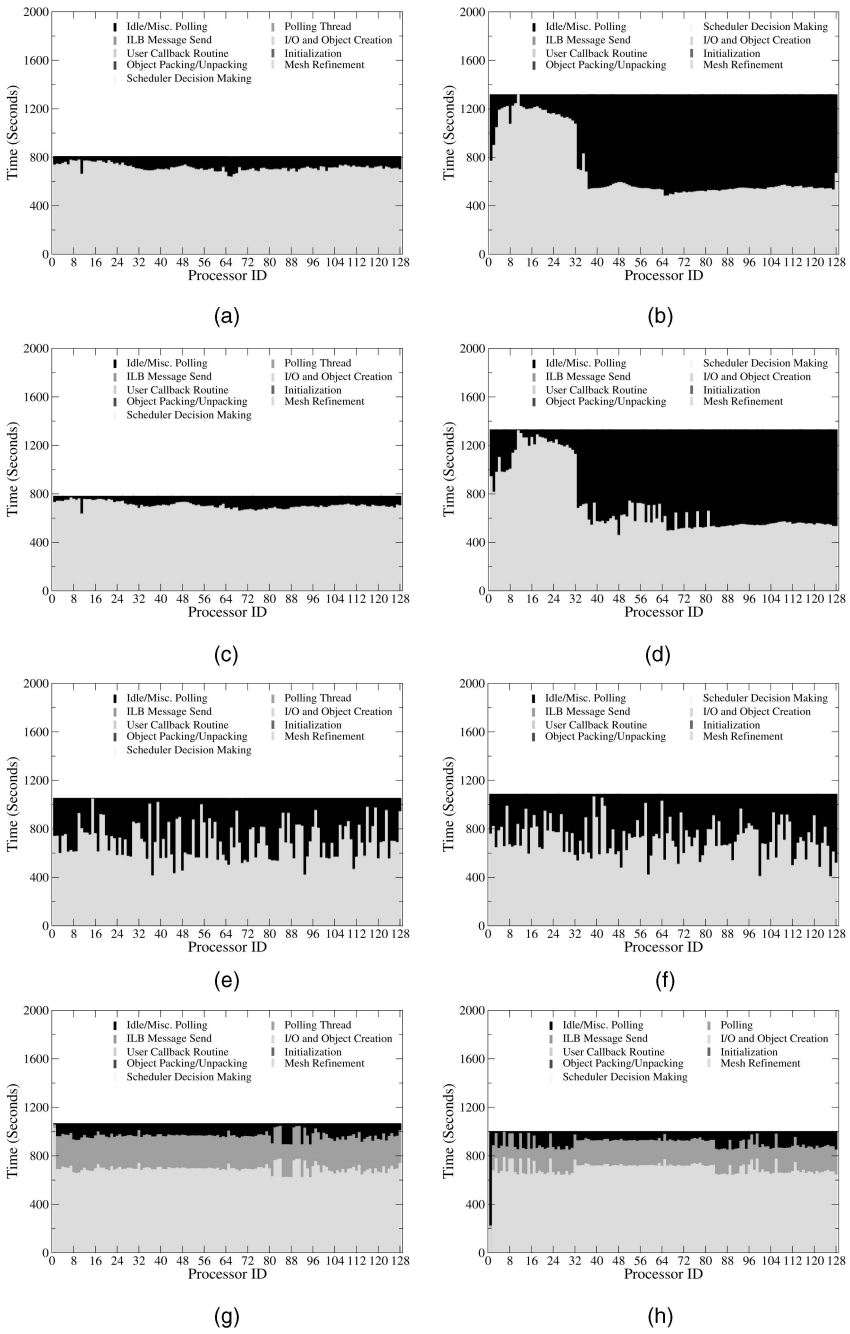

Fig. 3. Breakdown data for multithreaded (left) and single-threaded (right) implicit work stealing ((a) and (b)), diffusion ((c) and (d)), multilist ((e) and (f)), and master/worker methods ((g) and (h)) on 128 processors.

discussion in this section to those load balancing software projects which *dynamically* balance application workload. Since the goal of this research is not to discuss yet another load balancing strategy and its implementation, but instead to present a runtime software framework which can be used to quickly develop and easily evaluate customized load balancing algorithms, we will focus on this area in contemporary research.

In particular, we wish to distinguish our research using six criteria, shown in Table 2.

1. Support for data migration. Migrating processes or threads adds to the complexity of the runtime system and is often not portable. Migrating data and thereby implicitly migrating computation is a more portable and simple solution.
2. Support for explicit message passing. Message passing is a programming paradigm that developers are familiar with and the Active Messages [35] communication paradigm we use is a logical extension to that. Explicit message passing is also attractive because it does not hide parallelism from the developer.
3. Support for a global namespace. A global namespace is a prerequisite for automatic data migration; applications need the ability to reference data regardless of where it is in the parallel system.

TABLE 1
Overheads Imposed by Multithreaded Runtime System on 128 Processors

Load Balancing	Min/Max Refinement	Std Dev	PREMA Overhead					
			Poll Thr.	Packing	Msg. Send	Call-back	Sched.	Prcnt.
None	487/1332 sec	305	–	–	–	–	–	–
Stop & Repart.	499/786 sec	51	–	–	–	–	–	–
ILB W.S.	635/775 sec	27	0.1681 sec	0.0074 sec	0.0018 sec	0.0063 sec	0.0019 sec	0.00023%
ILB Diffusion	632/765 sec	25	0.2080 sec	0.0057 sec	0.0026 sec	0.0053 sec	0.0670 sec	0.00037%
ILB PML	411/1043 sec	147	0.9669 sec	0.0351 sec	0.0021 sec	0.0108 sec	1.0893 sec	0.00200%
ILB M.W.	622/775 sec	30	0.5377 sec	0.2227 sec	0.0111 sec	0.2115 sec	0.000004 sec	0.00092%

4. Single-threaded application model. Presenting the developer with a single-threaded programming model greatly reduces application code complexity and development effort.
5. Automatic load balancing. The runtime system should migrate data or computation transparently and without intervention from the application.
6. Customizable load balancing. It cannot be said that there is a "one size fits all" load balancing algorithm; different algorithms perform well in different circumstances. Therefore, users need the ability to easily develop and experiment with different application-specific load balancing strategies without the need to extensively modify their application code. This is in contrast to the strategy taken by libraries such as DRAMA [5], which provide several predefined load balancing or repartitioning algorithms but no facility to design and implement new strategies.

Several systems, such as Split-C [13], provide the basic support for a global namespace. However, Split-C is designed for applications with no data migration or else the user has to explicitly maintain the consistency of the namespace. An alternative approach can be found in systems such as Amber [9], Emerald [20], and COOL2 [1]. However, such systems often rely on a scheme of partitioning the virtual address space, with parallelism expressed through the features provided in a new programming language. This approach raises the cost associated with using the system, forcing developers to commit to a new language which may or may not be supported in the future. Also, while this functionality provides the needed infrastructure for load balancing, none of these systems assume responsibility for the decision making associated with automatic data migration.

Systems such as VDS [15] and Millipede [18] provide support for automatic dynamic load balancing through the dispersion and migration of computation threads. This is in contrast to the single-threaded programming model supported by PREMA. We believe that a single-threaded model not only eases the development of new application codes, but makes the porting of existing codes to the PREMA environment easy as well. However, because PREMA incorporates multiple threads in the runtime system, we are able to achieve the quality of load balancing available with multithreaded systems.

Systems such as the C Region Library (CRL) [19] implement a shared memory model of parallel computing. Parallelism is achieved through accesses to shared regions of virtual memory. The message passing paradigm we employ explicitly presents parallelism to the application. In addition, PREMA does not make use of copies of data

TABLE 2
Software Systems that Support Functionality Similar to the PREMA System

System Name	Data Migration	Message Passing	Global Name Space	Supports Single-Threaded Model	Auto. Load Balancing	Customizable Load Balancing
Orca [3]	•		•			
Split-C [13]			•	•		
Multipol [32]			•		•	
CRL [19]			•	•		
Cilk [7]					•	
Amber [9]	•		•			
Emerald [20]	•		•			
Treadmarks [2]			•	•		
COOL2 [1]	•		•			
CHAOS++ [8]	•		•	•		
VDS [15]	•				•	•
Millipede [18]	•		•		•	
Chare Kernel [31]	•	•		•		
CHARM++ [21]	•	•	•	•	•	•
DRAMA [5]	•			•	•	
PLUM [27]	•			•	•	
Zoltan [16]	•			•	•	•
JOVE [34]	•			•	•	
PREMA/ILB	•	•	•	•	•	•

objects, removing much of the complexity involved with data consistency and read/write locks.

Zoltan [16] and CHARM++ [21] are two systems with similar characteristics to PREMA. Zoltan provides graph-based partitioning algorithms and several geometric load balancing algorithms. Because of the synchronization required during load balancing, Zoltan behaves in much the same way as other stop-and-repartition libraries whose results are presented in this paper. CHARM++ is built on an underlying language which is a dialect of C++ and provides extensive dynamic load balancing strategies. However, the *pick-and-process* message loop guarantees that entry-point methods execute "sequentially and without interruption" [22]. This may lead to a situation in which coarse-grained work units may delay the reception of load balancing messages, negating their usefulness, as was seen with the single-threaded PREMA results presented in Figs. 3b and 3d.

7 Conclusions and Future Work

We have presented the Implicit Load Balancing (ILB) component of the PREMA runtime system and demonstrated its effectiveness in balancing the coarse-grained computation of a parallel 3D advancing front mesh refinement program. We have demonstrated performance improvements of 15 percent over traditional stop-and-repartition methods, 30 percent over intrusive explicit load balancing methods, and 42 percent over no load balancing on configurations of 128 processors. In addition, we have shown that the overhead caused by the PREMA runtime system is negligible, adding only a small fraction of 1 percent to the overall runtime.

Aside from these performance benefits, we have given an indication of the flexibility of the PREMA system by implementing several load balancing policies. The four policies shown here scratch the surface of the scheduling methods possible.

While the scheduling policies described here are adequate for use with the parallel advancing front mesh refinement program, our next goal is to extend these methods and the ILB framework in order to take into consideration object *affinities* when making load balancing decisions. It is often the case that dependencies will exist between data objects; colocating these objects proves beneficial to runtime performance. Knowledge of such dependencies must be incorporated into the scheduling policy in order to extend the class of applications which can derive maximum benefit from PREMA. PREMA will be the vehicle to investigate such scheduling policies.

Acknowledgments

The authors thank the anonymous referees who with their suggestions help to improve the presentation of this paper. This work was performed in part using computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the US National Science Foundation (NSF), and Virginia's Commonwealth Technology Research Fund. This work was supported by NSF grants: #EIA-0203974, Career Award #CCR-0049086, #ACI-0085969, and #EIA-9972853.

References

[1] P. Amaral, C. Jacquemot, P. Jensen, R. Lea, and A. Mirowski, "Transparent Object Migration in COOL2," *Position Papers of Proc. ECOOP '92 Workshop W2,* pp. 72-77, 1992.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared Memory Computing on Networks of Workstations," *Computer,* vol. 29, no. 2, pp. 18-28, Feb. 1996.

[3] H. Bal, M. Kaashoek, and A. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Trans. Software Eng.,* vol. 18, no. 3, pp. 190-205, Mar. 1992.

[4] K. Barker, N. Chrisochoides, J. Dobbelaere, and D. Nave, "Data Movement and Control Substrate for Parallel Adaptive Applications," *Concurrency Practice and Experience,* vol. 14, pp. 77-101, 2002.

[5] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw, "Dynamic Load-Balancing of Finite Element Applications with the Drama Library," *Applied Math. Modeling,* vol. 25, pp. 83-98, 2000.

[6] A. Belguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpore, "PVM: Experiences, Current Status, and Future Direction," *Supercomputing '93 Proc.,* pp. 765-766, 1993.

[7] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "CILK: An Efficient Multithreaded Runtime System," *Proc. Fifth Symp. Principles and Practice of Parallel Programming,* pp. 55-69, 1995.

[8] C. Chang, A. Sussman, and J. Saltz, *Parallel Programming Using C++,* chapter CHAOS++. MIT Press, 1996.

[9] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors," *Proc. 12th Symp. OS Principles (SOSP12),* pp. 147-158, Dec. 1989.

[10] A. Chernikov, N. Chrisochoides, and K. Barker, "Parallel Programming Environment for Mesh Generation," *Proc. Eighth Int'l Conf. Numerical Grid Generation in Computational Field Simulations,* 2002.

[11] N. Chrisochoides and K. Barker http://www.cs.wm.edu/pes/software/dmcs/dmcs.html, June 2003.

[12] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel, "Mobile Object Layer: A Runtime Substrate for Parallel Adaptive and Irregular Computations," *Advances in Eng. Software,* vol. 31, nos. 8-9, pp. 621-637, Aug. 2000.

[13] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumeta, T. von Eicken, and K. Yelick, "Parallel Programming in Split-C," *Proc. Supercomputing (SC '93),* pp. 262-273, 1993.

[14] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel and Distributed Computing,* vol. 7, no. 2, pp. 279-301, 1989.

[15] T. Decker, "Virtual Data Space—Load Balancing for Irregular Applications," *Parallel Computing,* vol. 26, pp. 1825-1860, 2000.

[16] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan, "Design of Dynamic Load-Balancing Tools for Parallel Applications," *Proc. Int'l Conf. Supercomputing,* May 2000.

[17] "MPI Forum Message Passing Interface Standard 1.0 and 2.0," http://www.mcs.anl.gov/mpi/index.html, 1997.

[18] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster, "Millipede: Easy Parallel Programming in Available Distributed Environments," *Software—Practice and Experience,* vol. 27, no. 8, pp. 929-965, Aug. 1997.

[19] K. Johnson, M. Kaashoek, and D. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," *Proc. 15th Symp. Operating Systems Principles (COSP15),* pp. 213-228, Dec. 1995.

[20] E. Jul, H. Levy, N. Hutchison, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Trans. Computer Systems,* vol. 6, no. 1, pp. 109-133, Feb. 1988.

[21] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93),* pp. 91-108, 1993.

[22] L. Kalé, B. Ramkumar, A. Sinha, and A. Gursoy, "The Charm Parallel Programming Language and System Part II—The Runtime System," Technical Report 95-03, Parallel Programming Laboratory, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, 1995.

[23] S. Kohn and S. Baden, "Parallel Software Abstractions for Structured Adaptive Mesh Methods," *J. Parallel and Distributed Computing,* vol. 61, no. 6, pp. 713-736, 2001.
[24] Univ. of Indiana at Bloomington LAM Team. "LAM/MPI Parallel Computing," http://www.lam-mpi.org/, 2003.
[25] F. Muniz and E. Zaluska, "Parallel Load Balancing: An Extension to the Gradient Model," *Parallel Computing,* vol. 21, pp. 287-301, 1995.
[26] J. Neto, P. Wawrzynek, M. Carvalh, L. Martha, and A. Ingraffea, "An Algorithm for Three-Dimensional Mesh Generation for Arbitrary Regions with Cracks," *Eng. with Computers,* vol. 17, pp. 75-91, 2001.
[27] L. Oliker and R. Biswas, "Plum: Parallel Load Balancing for Adaptive Unstructured Meshes," *J. Parallel and Distributed Computing,* vol. 52, no. 2, pp. 150-177, 1998.
[28] M. Parashar and J. Browne, "DAGH: A Data-Management Infrastructure for Parallel Adaptive Mesh Refinement Techniques," technical report, Dept. of Computer Science , Univ. of Texas at Austin, 1995.
[29] R. Said, N. Weatherill, K. Morgan, and N. Verhoeven, "Distributed Parallel Delaunay Mesh Generation," *Computer Methods in Applied Mechanics and Eng.,* vol. 177, pp. 109-125, 1999.
[30] K. Schloegel, G. Karypis, and V. Kumar, "Parallel Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes," Technical Report 97-014, Univ. of Minnesota, 1997.
[31] W. Shu and L. Kalé, "Chare Kernel—A Runtime Support System for Parallel Computations," *J. Parallel and Distributed Computing,* vol. 11, no. 3, pp. 198-211, 1991.
[32] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole and Radiosity," *J. Parallel and Distributed Computing,* vol. 27, pp. 118-141, 1995.
[33] M. Snir, S. Otto, S. Huss-Lederman, and D. Walker, *MPI the Complete Reference.* MIT Press 1996.
[34] A. Sohn and H. Simon, "JOVE: A Dynamic Load Balancing Framework for Adaptive Computations on an SP-2 Distributed-memory Multiprocessor," Technical Report 94-60, Dept. of Computer and Information Science, New Jersey Inst. of Technology, 1994.
[35] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Int'l Symp. Computer Architecture,* pp. 256-266, May 1992.
[36] C. Walshaw, M. Cross, and M. Everett, "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes," *J. Parallel and Distributed Computing,* vol. 47, pp. 102-108, 1997.
[37] I. Wu, "Multilist Scheduling: A New Parallel Programming Model," PhD thesis, School of Computer Science, Carnegie Mellon Univ., July 1993
[38] X. Yuan, C. Salisbury, D. Balsara, and R. Melhem, "Load Balancing Package on Distributed Memory Systems and Its Application Particle-Particle and Particle-Mesh (P3M) Methods," *Parallel Computing,* vol. 23, no. 10, pp. 1525-1544, 1997.

Kevin Barker received the BS degree in computer science (1997) from North Carolina State University and the MS degree in computer science (2001) from the University of Notre Dame. His Master's thesis was on runtime support software for adaptive applications on multilevel, nontraditional parallel computer architectures. In 2001, he began studying toward the PhD degree in the Computer Science Department at the College of William and Mary. His research interests include developing runtime support software for highly adaptive parallel numeric codes.

Andrey Chernikov received the BS (1999) and MS (2001) degrees in applied mathematics and computer science from the Kabardino-Balkarian State University, Nalchik, Russia. His Master's thesis was on molecular dynamics modeling of ionic systems. From 1997 to 2001, he worked at the Institute of Informatics of Russian Academy of Sciences on Geographic Information Systems. In 2001, he began studying toward the PhD degree in computer science at the College of William and Mary. His research areas include computational geometry and parallel computing with the focus on parallel mesh generation.

Nikos Chrisochoides received the BSc degree in math from Aristotle University, Thessaloniki Greece. He received the MSc degree in math and PhD degree in computer science from Purdue University, West Lafayette, Indiana. In 1997, he joined the faculty at the University of Notre Dame (Department of Computer Science and Engineering) as an assistant professor. In 2000, he joined the faculty of the Computer Science Department at the College of William and Mary as an Associate Professor . He has been a member of the JPL's HTMT Petaflop team responsible for large scale "out-of-core" scientific applications focusing on runtime issues like data transfer and synchronization patterns. He received a number of research fellowships and awards, among them the US National Science Foundation Career Award. His research in parallel and distributed computing, "green" computing, and parallel mesh generation is application driven.

Keshav Pingali received the BTech degree from the Indian Institute of Technology, Kanpur, India, in 1978. He graduated from the Massachusetts Institute of Technology in 1986 with the SM, EE, and ScD degrees. Since then, he has been a faculty member in the Computer Science Department at Cornell University, where he was promoted to full professor in 1999. His research is in the area of compilers for high-performance computers. He has won a number of best paper awards at conferences such as the International Conference on Supercomputing (ICS) and ASPLOS. His innovations in automatic locality enhancement for machines with deep memory hierarchies have been incorporated into several production compilers from leading computer manufacturers. He has also worked in functional languages and dataflow machines. At Cornell, he has won many teaching awards, including the Russell Teaching Award of the College of Arts and Sciences (1998) and the Ip-Lee Teaching Award of the College of Engineering (1997). In 2000, he was the N. Rama Rao Visiting Chaired Professor at the Indian Institute of Technology, Kanpur.