An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications[*][†]

Kevin J. Barker and Nikos P. Chrisochoides
Dept. of Computer Science
College of William and Mary
Williamsburg, VA 23185

Abstract

We present an evaluation of a flexible framework and runtime software system for the dynamic load balancing of asynchronous and highly adaptive and irregular applications. These applications, which include parallel unstructured and adaptive mesh refinement, serve as building blocks for a large class of scientific applications. Extensive study has lead to the development of solutions to the dynamic load balancing problem for loosely synchronous and computation intensive programs; however, these methods are not suitable for asynchronous and highly adaptive applications. We evaluate a new software framework which includes support for an Active Messages style communication mechanism, global name space, transparent object migration, and preemptive decision making. Our results from both a 3-dimensional parallel advancing front mesh generation program, as well as a synthetic microbenchmark, indicate that this new framework out-performs two existing general-purpose, well-known, and widely used software systems for the dynamic load balancing of adpative and irregular parallel applications.

1 Introduction

The dynamic load balancing of loosely synchronous and computationally intense adaptive applications (such as field solvers) has been studied extensively in the past 15 years, and some very good methods and software tools [12, 16, 3, 8, 15, 17, 11] have been developed to assist the computational scientist. These methods and software systems are designed to support the development of parallel multi-phase computations in which computationally-intensive phases are separated by computations, such as global error estimation, which require global synchronization. Load balancing is accomplished by dynamically repartitioning the data after the global synchronization phases [11, 22]. However, there exists a class of problems, known as *asynchronous* and *highly adaptive*, for which the dynamic load balancing problem remains open.

Asynchronous and highly adaptive applications are defined by several characteristics. The first is that no global synchronization points are inherent to the application. Barriers used to exchange processor workload information for the purposes of load balancing must therefore be artificially introduced. A second characteristic is that the computational weights associated with individual work units may vary drastically throughout the execution (i.e., from one iteration to the next) of the application. This reflects the adaptivity of the application. For instance, the computational weights associated with "interesting" areas in the data domain will be much larger than for "uninteresting" regions. Furthermore, as the computation progresses, it is impossible to predict which regions may become more or less "interesting". These characteristics imply

[*]This work was supported by NSF Career Award #CCR-0049086, ITR #ACI-0085969, Research Infrastructure #EIA-9972853, and NGS #EIA-0203974.

[†]Permission to make digitalor hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on services or to redistribute to lists, requires prior specific permission and/or fee.
SC'03 November 15-21, 2003, Phoenix, Arizona, USA. Copyright 2003 ACM 1-58113-695-1/03/0011 ... $5.00

that the computational characteristics of the application cannot be predicted *a priori*, complicating the load balancing task immensely.

An example of such an application is Parallel Adaptive Mesh Refinement (PAMR) for multi-scale applications, such as those used to study crack growth through a macroscopic structure under stress. Such a system is subject to the laws of elasticity and plasticity which can be solved using the finite element method. However, crack growth forces the geometry of the domain to change, which, in turn, necessitates localized re-meshing. Adaptivity arises, for instance, when the tip of the advancing crack crosses from one subdomain into another (say from subdomain D_i into D_j). It is unknown in advance when or where the crack growth will take place, as well as the subdomains that will be affected. The computational complexity associated with subdomain D_j will increase dramatically due to the greater level of mesh refinement necessary to capture the advancing crack tip. This adaptivity makes the mesh refinement process highly irregular, meaning incremental load balancing will be of little benefit.

Existing load balancing methods found in then literature and in publicly available software are not suitable for asynchronous and highly adaptive applications for the following three reasons:

- *Large penalty for global synchronization.* Load balancing schemes built on a loosely synchronous application model rely on global synchronization points for exchanging processor load information. For applications built using this model, this is often not a problem, since computationally intensive workloads can amortize this overhead. However, for asynchronous applications such as PAMR, which are often data intensive, synchronization points inserted solely for the purpose of load balancing can be detrimental to overall performance.

- *Difficulty in predicting future work loads.* For instance, in the crack growth example, it is difficult to predict which subdomains are going to be impacted by the advancing crack, making the computational effort associated with those domains difficult to predict.

- *Heavy workloads may delay message processing.* This means that status update and migration request messages associated with asynchronous load balancers may not arrive or be processed in a timely manner. This is especially true for message passing libraries which rely on *polling* or *receive* operations.

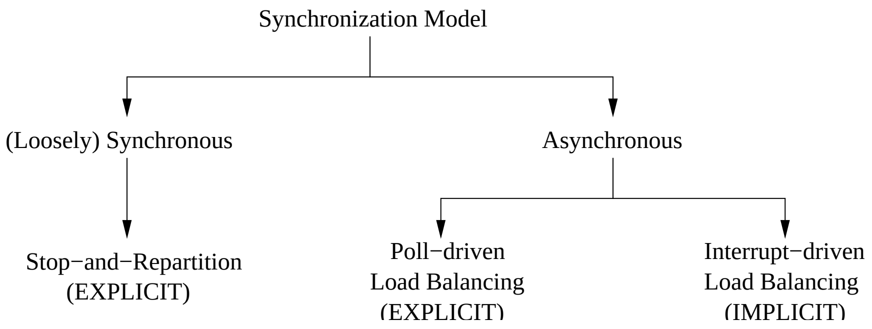
The contribution of this paper is an evaluation of a low-overhead and general purpose runtime system for the efficient implementation of asynchronous and highly adaptive irregular parallel scientific computing applications.

The rest of the paper is organized as follows: in Section 2 we overview the state-of-the-art dynamic load balancing methods and systems, as well as enumerate several of the tradeoffs that system designers must make. In Section 3, we focus on two general purpose and widely used software systems for the dynamic load balancing of parallel scientific computations. In Section 4, we present our framework and software system. In Section 5, we examine the performance of these three systems in the context of a synthetic benchmark program which, due to its flexibility, enables us to examine the effects of factors such as initial load imbalance and ranges of computational weights present in the work units. Finally, we present our conclusions and future work in Section 6.

2 Load Balancing State-of-the-Art

In order to provide an overview of the current state-of-the-art in load balancing technology, it is important to partition the design space along several axes. This can be done by dividing the load balancing process into its three primary steps: (1) information gathering and dissemination, (2) decision making, and (3) data and/or computation migration. Of these, the first two play a critical role in the load balancing of highly adaptive applications.

From the application's perspective, we can view these axes in terms of a *syncrhonization model.* For instance, *loosely synchronous* applications are composed of computationally intense phases separated by phases which require global synchronization, making them well suited for load balancing schemes which employ synchronous information gathering and decision making. This point of view gives us the following classification:



Synchronization Model
(Loosely) Synchronous
Asynchronous
Stop−and−Repartition (EXPLICIT)
Poll−driven Load Balancing (EXPLICIT)
Interrupt−driven Load Balancing (IMPLICIT)

Info. Dissemination	Explicit	Implicit
Prog. Supplied Hints	Zoltan [8], DRAMA [3], METIS [11], ParMETIS [17], PREMA + Explicit Polling	PREMA + Interrupts
Profiling	Charm++ [9]	

Figure 1: Using synchronization as a criterion for system classification

- **(Loosely) Synchronous vs. Asynchronous**: Synchronous load balancing methods and tools must gather load information from all processors in order to reconstruct the global system state before any load balancing decisions can be made. Software known as *repartitioning* tools fall into this category. Once a graph describing the application state is constructed, it is partitioned in order to equally distribute the work load. While such load balancers are often able to achieve load balance of high quality, the cost associated with synchronization and information exchange grows with both the number of processors in the parallel system, and variance between the most and least computationally intense work units; processors with less computationally intense data partitions may reach global synchronization points long before more loaded processors leading to wasted processor cycles. In contrast, asynchronous methods require communication with only a small fixed-size "neighborhood" of processors. Load balancing occurs only within neighborhoods, meaning that unaffected processors may continue with useful application work. However, load balance across the entire system is often of lower quality, although work has been done to improve this [14].

- **Programmer-supplied Hints vs. Runtime Instrumentation**: Load balancers must be able to predict future load in order to make intelligent migration decisions. One method for doing this is for the programmer to provide hints about the weight of pending computation. However, in an adaptive application, this can be difficult to do with any accuracy. A second option is to make the assumption that future performance will be related to what has been seen in the past. In this case, the runtime system (or compiler, in the case of a specialized language) can insert profiling code into the program to gather runtime statistics which can be used to predict future load. Unfortunately, the computational weight associated with given data objects or methods can vary dramatically in an adaptive application (for instance, a growing crack tip may cross into a new subdomain). In addition, some methods or routines may only execute once, making profiling of little value.

- **Explicitly Initiated Load Balancing vs. Preemptive Load Balancing**: Because the load balancer's functionality exists on the same physical processor set as the user's application, load balancing will necessarily steal processor cycles away from the application's computation. Therefore, a mechanism must exist for determining when and how often to stop executing the application's code and begin load balancing. This can occur in one of two ways: either explicitly by the application, or implicitly without the application's knowledge or control. Explicit load balancing has the advantage that well-tuned application routines will not be interrupted. This can be beneficial, for instance, when high-performance caching algorithms are utilized. However, a disadvantage to this approach can be that load balancer invocation can be delayed during computationally intense periods, leading to poor load balancing performance. As an alternative, implicit load balancing will periodically check for pending balancer messages, guaranteeing that they are processed quickly. However, well-tuned application code must be interrupted, potentially impacting routine performance negatively.

Load balancing systems each must make decisions regarding these tradeoffs. Figure 1 highlights several systems, which will discuss in greater detail next.

3 Representative Load Balancing Systems

Below we examine two widely-used load balancing software packages in the context of the classification framework described in Section 2. This analysis will allow us to determine which characteristics are of the most benefit when load balancing asynchronous and irregular applications.

3.1 ParMETIS

Repartitioning tools are the most frequently used dynamic load balancing methods found in the scientific computing literature. These methods make use of *a priori* knowledge of the computation in order to partition the workload (or problem domain, in the case of mesh refinement) into a user-specified number of chunks (subdomains). Some methods use graph partitioning algorithms to divide an initial graph into equally weighted subgraphs. Other methods are more application-specific, and may choose to optimize certain criteria, such as subdomain surface-to-volume ratio, cut edge weights, or data redistribution costs.

Repartitioning tools can be found incorporated into such projects as Jostle [22], DRAMA [3], Zoltan [8], and Metis [11]. For the comparisons presented in this paper, we have chosen to use Metis as a representative for this class of tools, due to the fact that Metis is widely used and often serves as a basis for other software systems.

Two common methods exist for creating a new partitioning for an already distributed mesh that has become load imbalanced due to mesh refinement and coarsening: *scratch-remap* schemes create an entirely new partition and tend to more evenly distribute load, while *diffusive* schemes attempt to tweak the existing partition to achieve better load balance, often minimizing data migration costs. Metis' *ParMETIS_V3_Adaptive-Repart()* routine makes use of a Unified Repartitioning Algorithm [19], which combines the characteristics of both scratch-remap and diffusive schemes.

A parameter known as the Relative Cost Factor (α) is application-defined and describes the relative costs required for performing interprocessor communication during parallel processing and performing data redistribution associated with load balancing. This gives rise to the minimization function

$$|E_{cut}| + \alpha\,|V_{move}| \tag{1}$$

where $|E_{cut}|$ is the edge-cut of the partitioning, and $|V_{move}|$ is the total cost of data redistribution.

Repartitioning progresses in three stages. First, the graph is coarsened using a local variant of heavy-edge matching [19] that is shown to be effective at helping to minimize both the number of edge-cuts and data redistribution costs. In addition, this algorithm is scalable to a large number of processors. The second step is to create an initial partition. Because the most beneficial method depends on the particular problem instance [20], as well as the value chosen for the Relative Cost Factor (α), the initial partition is created twice (once using a scratch-remap method, and once using a diffusive method). The cost function (Equation 1) is then computed, with the best option chosen. Finally, a multilevel refinement algorithm is used [18] while minimizing Equation 1.

This type of explicit repartitioning suffers from the global synchronization and inaccurate workload prediction problems discussed previously. In order to combat these problems, multiple repartitionings are often necessary, forcing the costs associated with synchronization to be paid multiple times.

3.2 Charm++

In many cases, applications (e.g. simulations) are organized as a series of discrete time steps. In such cases, it is often beneficial to perform load balancing at strategic locations, rather than at arbitrary points during the computation. Charm++ [9] provides a runtime framework in which load balancing policies may be "plugged into" an application in a modular fashion. With each module provided in the Charm++ distribution, the load balancing methods are implemented using a global barrier [4], making them well suited for loosely synchronous computations[1].

Charm++ presents a programming model in which the application data domain is divided into a number of *chunks*, with the number of chunks being much greater than the available number of physical processors. Each chunk is represented as a *chare* object, whose interface is defined by *entry point* methods. Messages invoke computation by specifying the entry point to execute upon reception. Load balancing is achieved by mapping and re-mapping chares to available processors. An assumption, known as the *principle of persistent computation and communication structure* [5], is made which states that changes to the computation and communication structure of an application happen slowly or infrequently.

Two components make up the Charm++ load balancing framework: the specific load balancing policy or strategy and a distributed load balancing database constructed through runtime monitoring of the application. The load balancing module makes use of the information contained within the database (possibly gathering it at a central location, if necessary) to determine what chares should migrate in order to balance the runtime load.

Because creating an optimal load distribution is an NP-hard problem that involves optimizing for both interprocessor communication and load distribution, several heuristic approaches are provided [4]. The simplest are *Greedy Strategies*, which sort both chare workloads and processor load levels in order to assign the heaviest free chare to the processor with the lightest current load. Such a strategy may result in a large amount of data migration. *Refinement Strategies* aim to minimize the number of chare migrations while improving load balance. For each overloaded processor only, heavy objects are migrated to underloaded processors until the load falls below a threshold, which is defined as a percentage of the average processor workload. Finally, Charm++ provides *Metis-based Strategies*, which make use of Metis graph partitioning capabilities described earlier.

For highly adaptive asynchronous applications, such as Parallel Adaptive Mesh Generation and Refinement, there are several problems with this runtime model. First, each chare (mesh subdomain) is refined only once during each mesh refinement iteration, making the runtime data gathering method ineffective at predicting future load. Second, Charm++'s *pick-and-process* loop [10], which executes on each processor and selects messages destined for local chares for execution, ensures that the entry point methods specified by the messages execute atomically. Large, coarse-grained entry point methods may delay the subsequent processing of messages, potentially delaying load balancing status update or request messages for some time, hampering load balancing performance. The performance data of Figures 3 and 4 validate this point.

4 PREMA

The Parallel Runtime Environment for Multicomputer Applications (PREMA) is a runtime library based on a design philosophy which includes:

- single-sided communication [2] similar to what is provided by Active Messages [21];
- a global namespace which assigns a unique identifier to application-defined *mobile objects* [6];
- transparent object migration and automatic message forwarding for mobile objects [6];
- a framework which allows for the easy and efficient implementation of customized dynamic load balancing algorithms [1];
- and a suite of commonly used dynamic load balancing strategies, such as *Diffusion* [7] and *Multi-list Scheduling* [23].

The PREMA runtime model is similar to that of Charm++ in that the application's data domain is first decomposed into some number of subdomains, which is greater than the number of available physical

[1]According to [4], local synchronization barriers are used in all load balancing modules provided in the Charm++ distribution, but are not necessary in custom-designed modules.

```
    // ---- Sequential Example ----
1   class tree_node_t {
2    public:
3     void do_work() {
4       if (left_child != NULL) { left_child->do_work(); }
5       if (right_child != NULL) { right_child->do_work(); }
6       // ... do more work here for local node ...
7     }
8    private:
9     tree_node_t* left_child;
10    tree_node_t* right_child;
11  };

    // ---- Example using PREMA's ILB load balancing framework ----
1   void do_work_handler(int src, mol_mobile_ptr_t* mp, void* obj_data,
2                        void* user_data, int size, void* arg)
3   {
4     tree_node_t* node = (tree_node_t*)obj_data;
5     node->do_work();
6   }
7
8   class tree_node_t {
9    public:
10    void do_work() {
11      if (!mol_mobile_ptr_is_null(left_child)) {
12        ilb_message(left_child, do_work_handler, NULL, 0, NULL);
13      }
14      if (!mol_mobile_ptr_is_null(right_child)) {
15        ilb_message(left_cihld, do_work_handler, NULL, 0, NULL);
16      }
17    }
18   private:
19    mol_mobile_ptr_t left_child;
20    mol_mobile_ptr_t right_child;
21  };
```

Figure 2: Example to perform some task over the nodes in a tree data structure. At the top is the sequential version of the code, while below is the code using the load balancing framework provided by PREMA's ILB component.

processors. Each subdomain is then registered with the PREMA system as a mobile object and assigned a unique *mobile pointer* [6]. Computation proceeds using a message-driven mechanism; messages target mobile objects and specify application-defined *handler* routines to be executed. Polling operations are used to receive and process application messages. As mobile objects migrate during runtime due to load balancing, the PREMA system uses a message forwarding scheme to guarantee that messages arrive at their targets and that message ordering is preserved.

The mobile pointer construct, combined with a consistent data access mechanism (the *message*), allows PREMA to facilitate a straight-forward migration from sequential code to its parallel, load balanced counterpart. The top portion of Figure 2 contains a sequential code sample which performs some task over the elements contained within a tree data structure. To convert this code to make use of the PREMA runtime system, local pointers between tree nodes are replaced by mobile pointers, and direct data accesses using pointer dereferences are replaced with accesses through messages (lines 12 and 15 in the bottom portion of the figure). These messages will invoke the application-defined *do_work_handler()* routine at their targets, which can then access the appropriate tree node methods. This strategy is independent of the locations of the individual tree nodes, allowing the runtime system to migrate data in accordance to the particular load balancing strategy.

PREMA's load balancing framework [1] is designed to allow as much flexibility as possible in the range of load balancing policies that are implementable. In order to distinguish PREMA from the software tools that require global synchronization points, we will focus this discussion on the *Work Stealing* scheduling policy which is provided with the PREMA distribution. Processors are paired with a single neighbor, with whom mobile objects may be exchanged during load balancing. Once a processor determines that it is underloaded (by comparing the local load level to an application-defined "water-mark"), a request message is sent to the partner processor who determines if any work is available to contribute. If so, some number of mobile objects[2] are uninstalled and migrated to the requesting processor. If no work is available, a negative acknowledgement is returned, at which point the requesting processor may choose another partner.

The PREMA library allows load balancing to be initiated either explicitly or implicitly. The application may explicitly hand control to the load balancer by posting a polling operation, which will check for incoming application messages, schedule the next work unit for execution, evaluate the current local work level, and process any system-generated load balancing messages. Alternatively, the runtime system may preempt the application at periodic intervals and perform load balancing functions. Note that, even in the case of preemptive load balancing, it is still necessary for the application to poll for its own messages.

4.1 Explicit Load Balancing

Explicit load balancing requires the application program to explicity hand control to the load balancing algorithm. This is done with the *polling* operation, which is required for the reception and processing of both application- and system-generated (load balancing) messages.

In addition to the delay often suffered by load balancing information and request messages due to the inability to preempt an already executing work unit is the difficulty in choosing an appropriate water-mark for triggering the initiation of load balancing. The threshold should be carefully selected in order to give the application a "cushion" of pending work while load balancing proceeds. Due to adaptivity, however, it is difficult to predict the computational weights of pending handler routines, which, in turn, makes predicting a suitable low water mark difficult. This can mean that applications either run out of work, incurring idle processor cycles, or begin load balancing before it is necessary, leading to mobile object "thrashing".

4.2 Implicit Load Balancing

PREMA is able to function in an implicit load balancing mode, in which the load balancer may preemptively take control of the processor. This occurs during computationally-intense work units, when a *polling thread* is spawned which will awake at predefined intervals to check for arriving load balancing information or request messages. From the application's point of view, load balancing messages arrive asynchronously at these points.

However, load balancing messages that are processed preemptively in no way affect the execution of the application. Work units that have begun execution are not migrated, and they must complete their execution before subsequent work units are scheduled. Also, the single-threaded programming model implemented by PREMA is preserved. By associating tags with messages, the runtime system is able to separate system-generated messages from those generated by the application, leaving application messages to be handled during application-posted polling operations. This means applications do not need to be concerned with making themselves thread-safe.

Such a preemptive scheme provides two primary benefits. First is that load balancing messages can be guaranteed to be received in a timely manner, ensuring that load balancing decisions will be based on up-to-date information. The second is that the importance of the value of the low water mark is deemphasized. Load balancing begins when the underloaded processor begins work on its last local work unit, regardless of that work units size. In the event that the work unit finishes before new units arrive, the number of wasted processor cycles is minimized. It is possible to use a platform-dependent threshold whose value can be determined at runtime to eliminate even these cases. We will implement this optimization in our future work.

[2]For particularly coarse-grained objects, a single mobile object may be migrated during load balancing; for finer-grained computations, multiple mobile objects may be migrated.

5 Performance Evaluation

Both a synthetic benchmark program and a 3-dimensional parallel advancing front mesh generator are used to evaluate the performance of each load balancing system. The benchmark program allows us to compare the performance of the three load balancers we have selected under a variety of different program characteristics, while the mesh generator allows us to see the performance of the PREMA system in a more "real-world" scenario.

The benchmark program makes use of the following algorithm:

1. Command-line parameters are parsed to determine the number of work units, the minimum and maximum computational weight for each unit, and the initial imbalance percentage.

2. The work units are created and distributed to the available processors. For PREMA and ParMETIS, this involves creating the appropriate number of data structures which represent the data subdomains, and, in the case of PREMA, registering them with the runtime system as mobile objects. With Charm++, a 1-dimensional chare array of the appropriate length is created. In this case, the runtime system will initially distribute the array elements to the available processors. This phase of the application is not considered in the final timing results.

3. Computation is assigned to each work unit. In the case of PREMA this involves each processor sending messages to local work units which will invoke a computation handler. With Charm++, these messages are sent using the chare array's proxy object. Finally, as ParMETIS does not provide any message passing functionality, the computation routines are invoked directly by each processor on local work units. The amount of computation assigned to each subdomain is dependent upon that subdomain's global index and the initial imbalance percentage.

4. Finally, control is handed to the runtime system and the load balancer.

There is no communication between work units, and work units are able to execute in any order. Note that the objective of this paper is to study the effectiveness and efficiency of load balancing framework for highly adaptive applications. Collocation of objects for the purpose of improving data locality is primarily the responsibility of the load balancing policy itself, and not of the framework itself. Therefore this study is outside the scope of this paper.

We vary two parameters: the initial imbalance percentage and the difference in computational weights between the heavy and lightly loaded work units. Note that because the imbalance is measured as a percentage of the total number of work units that are weighted heavily, different imbalance percentages mean a different total amount of computation performed by the application. Therefore, we must restrict our analysis to comparing the performance of different load balancing methods on the same problem, and avoid discussions of the performance of a particular method on two problems with differing initial characteristics. The data shown represent two values for each parameter. The initial imbalance is either 50% or 10%, and the heavy work units have a computational weight that is either double that of the lighter work units, or 20% heavier. In addition, in order to more accurately simulate the conditions found in a highly adaptive applications, those load balancing methods which rely on application-supplied hints to predict future work loads are often intentionally fed inaccurate information. This is due to the difficulty adaptive applications have in predicting the work loads of pending work units.

Because of the differences in each load balancing method, load balancing proceeds differently for each test case. PREMA provides for completely asynchronous load balancing, while ParMETIS and Charm++ require synchronization points in order for load balancing to begin. The number of synchronization points can have an impact both on the overall quality of the load balancing, as well as on the overhead attributable to the runtime system. In the case of the ParMETIS tests, load balancing does not begin until a particular processor's work load falls below a threshold[3]. At this point, the "root" processor is notified, and the load balancing procedure begins. The root processor is kept aware of which work units have completed as the program progresses, and is therefore able to make a determination of whether or not there is enough

[3]The threshold used in the ParMETIS tests is the same as what is used in the PREMA single-threaded and multi-threaded test cases. Charm++, because of its runtime performance profiling, does not rely on water-marks.

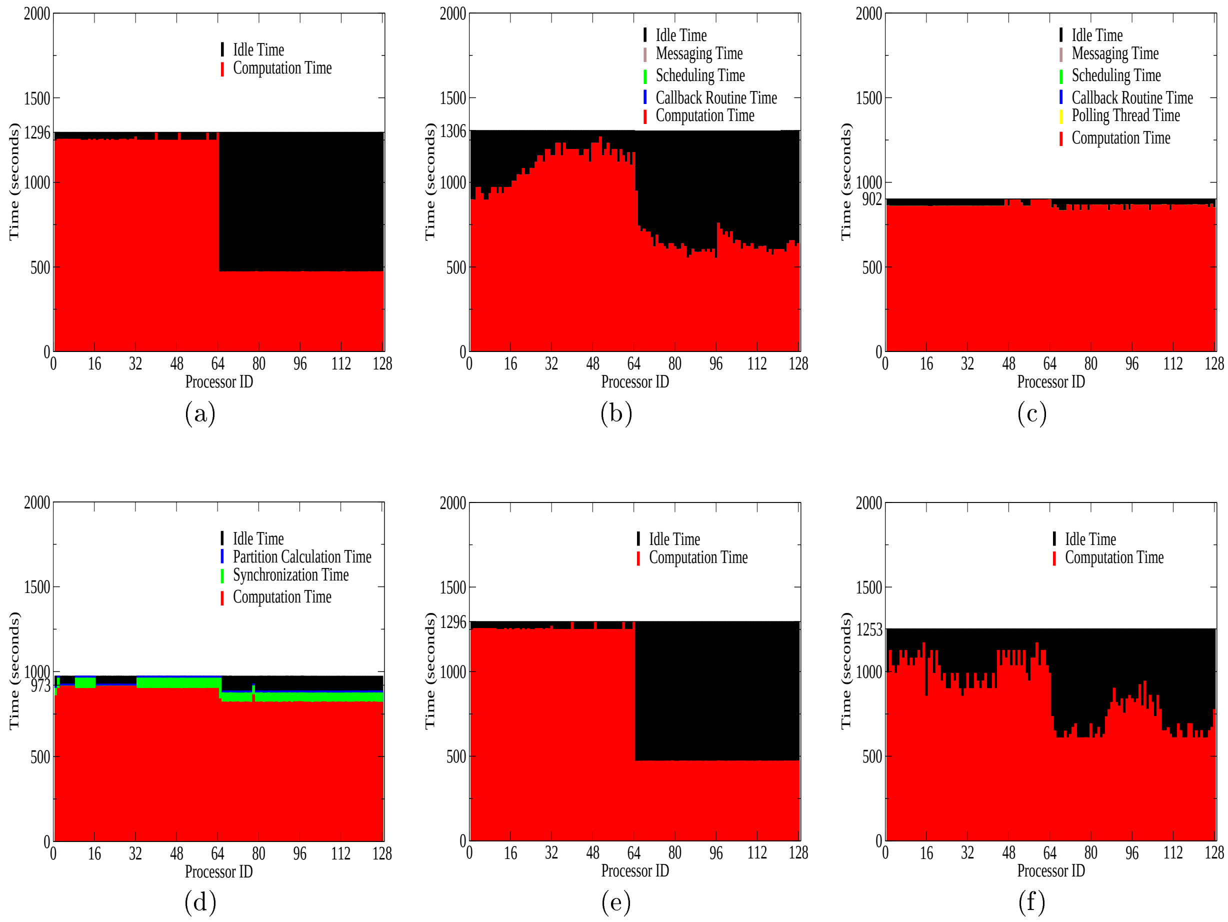


Figure 3: Benchmark performance for 50% initial imbalance, with the computational weight of heavy work units double that of lighter ones. (a) No Load Balancing, (b) PREMA with explicit load balancing, (c) PREMA with implicit load balancing, (d) ParMETIS, (e) Charm++ with no synchronization, and (f) Charm++ with 4 synchronization points.

outstanding work in the system to warrant load balancing. If so, the root node sends a message to all processors asking them to exchange work load information. Once this all-to-all communication phase is complete, each processor makes use of the ParMETIS library to calculate a new partition (re-partitioning), and then migrates work units accordingly.

Charm++ provides no provision for such a water-mark scheme, so load balancing must be performed incrementally. We must first decide how many load balancing iterations we want during the course of the application (I). This determines the size of the chare array which will contain our work units ($\frac{N}{I}$). The program begins by sending a message to each entry in the chare array to invoke a work unit. After the work unit completes, it checks to see how many times it has executed. If this number is less than I, then the *AtSync()* routine is used to signal a load balancing point has been reached. Once all entries in the chare array have reached the *AtSync()* operation, load balancing begins.

We evaluate the performance of each load balancing scheme in terms of two criteria: (1) its success in minimizing the overall runtime, and (2) the quality of the load distribution after load balancing. From the application's viewpoint, it is typically the overall runtime which is of the greatest interest. However, the quality of the load balancing will give some indication as to the success of the particular method's ability to contend with various symptoms of adaptivity, such as dramatic increases in work load in a localized area of the data domain. In addition, load balancing quality can be of interest to any stages that come later in the execution chain.

Figure 3 contains the performance breakdowns for the benchmark program with 50% of the work units comprised of tasks which require approximately 500 Mflops of computation (referred to as computationally "heavy"), and the remaining 50% composed of work units which require approximately 250 Mflops[4]. PREMA with preemptive message arrival is the overall performance winner, although ParMETIS also does well. Two factors account for ParMETIS' performance. First, the heavier work units are an integral factor of two times the weight of the lighter units, implying that when an even number of work units finish execution (on an individual processor), a heavier work unit is also finishing. This helps to reduce the synchronization overhead. *However, it should be noted that this is due to the construction of the particular benchmark test; other tests can be constructed in which this is not the case and which lead to higher synchronization costs.* Second, with 50% of the work units rated as "heavy", there is a large amount of pending work awaiting execution once load balancing begins. This allows ParMETIS to create a high quality partitioning.

PREMA with preemptive message arrival represents an overall runtime savings of 30% over no load balancing at all, and 7.3% over load balancing using ParMETIS, the next quickest total runtime. In addition, PREMA using implicit load balancing outperformed PREMA with explicit load balancing and Charm++ with no synchronization points by roughly 30%.

With highly adaptive applications, it is often the case that workload levels will "spike" in a relatively localized portion of the overall data domain. To simulate this, we ran our benchmark program with only 10% of the work units rated as "heavy", with the remaining 90% being "light". Figure 4 depicts the results of this test. Again, PREMA with implicit load balancing is the winner in terms of overall runtime. However, in this case ParMETIS did not perform so well, due to the fact that less work remained for execution by the time processors began to cross the underloaded threshold. In this case, ParMETIS was unable to calculate an effective partitioning, and mandated that work units remain on the processors on which they were originally assigned.

Using the standard deviation of the computation times across each processor, we can examine the quality of the load distribution after load balancing. Again, the most successful method is PREMA with preemptive message arrivals, with a standard deviation of just over 10. Charm++ and PREMA with explicit load balancing, which are based on load balancing messages arriving at specified points of operation, performed comparably with standard deviations of 128 and 100, respectively.

While dealing with drastic imbalance "spikes" is important for a load balancer when faced with highly adaptive applications, many applications suffer from less severe forms of imbalance. Figures 5 and 6 repeat the tests shown in Figures 3 and 4, but are different in that the work units rated as computationally "heavy" are only 20% heavier than the rest. Overall runtime and the quality of workload distribution are important in this case, just as before. However, because less load migration is possible, the overhead attributable to the runtime system becomes an important factor.

In the case of load balancing methods that rely on global synchronization points being used to balance asynchronous applications, the cost of the synchronization must be taken into effect. This cost, taken over all processors, relates to the number of processors in the parallel system, the imbalance percentage, and the difference in computational weight between "heavy" work units and lighter ones. For the case of ParMETIS, in Figure 5(d) this comes out to 7.4% of the useful computation time, while in Figure 4(d) this figure swells to 29.9%.

Because PREMA combined with preemptive load balancing message processing does not require even localized synchronization, this figure is much lower. Furthermore, the overhead attributable to the PREMA system remains constant, regardless of the characteristics of the application. For the same two tests, (Figures 5(c) and 4(c)) PREMA overhead works out to 0.045% and 0.029% of the useful computation time.

In addition to the benchmark data presented here, we have used our PREMA framework combined with preemptive load balancing to study a 3-dimensional tetrahedral parallel advancing front mesh generator. While we have not yet had the opportunity to study this application in the context of Charm++, we have demonstrated a 15% overall runtime improvement over stop and repartition techniques, and a 42% improvement over no load balancing [1]. At the same time, the overheads attributable to the PREMA runtime system are again well under 1% of the total runtime.

[4]The platform on which all experiments were executed consists of 128 333MHz Ultra Sparc 2i machines, connected by Fast Ethernet and utilizing the LAM [13] implementation of MPI.

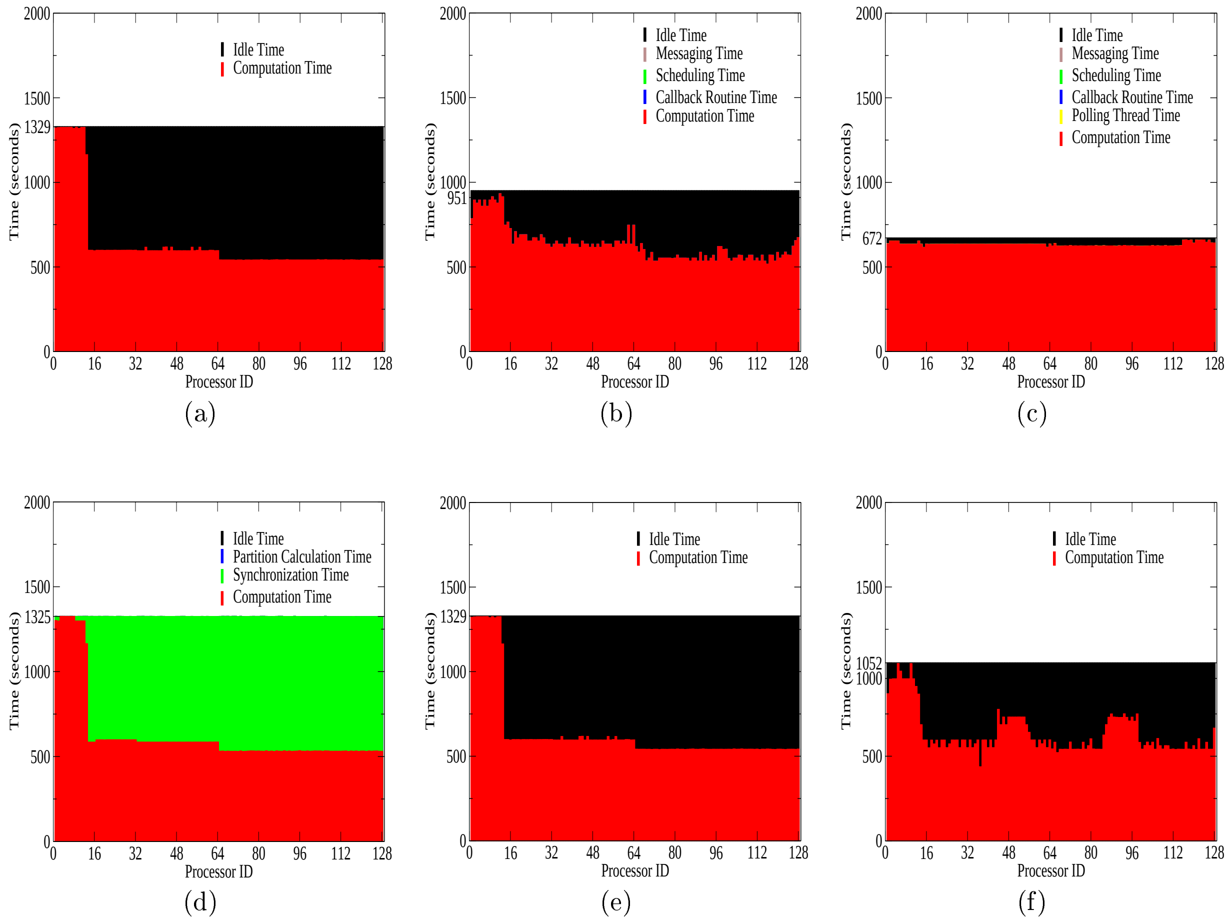


Figure 4: Benchmark performance for 10% initial imbalance, with the computational weight of heavy work units double that of lighter ones. (a) No Load Balancing, (b) PREMA with explicit load balancing, (c) PREMA with implicit load balancing, (d) ParMETIS, (e) Charm++ with no synchronization, and (f) Charm++ with 4 synchronization points.

6 Conclusions and Future Work

We have presented a runtime software system for implementing asynchronous and highly adaptive and irregular applications on distributed memory platforms. Our framework and its implementation are based on the flexible active messages communication paradigm, global name space, transparent object migration, automatic message forwarding for ease of use, and preemptive information dissemination and decision making for effective load balancing. Our performance data from a synthetic benchmark program and a 3-dimensional parallel mesh generation application suggest that our approach is effective in terms of minimizing idle cycles due to work load imbalances and efficient in terms of the overhead introduced during work load balancing for asynchronous and highly adaptive applications. The major contribution of this work is the implementation of an efficient runtime library that provides support for the efficient implementation of adaptive and irregular applications without relying on explicit and global synchronization for dynamic load balancing; to the best of our knowledge it is the only runtime system within the scientific community that effectively solves the dynamic load balancing problem for asynchronous and highly adaptive and irregular applications.

Our next goal is to extend this work so that we can present a unified method for solving the load balancing problem for end-to-end applications that consist of both asynchronous, highly adaptive computation phases, such as parallel mesh refinement, and loosely synchronous computation phases such as parallel sparse iterative field solvers. In this context, we plan to investigate within our existing framework the development of load

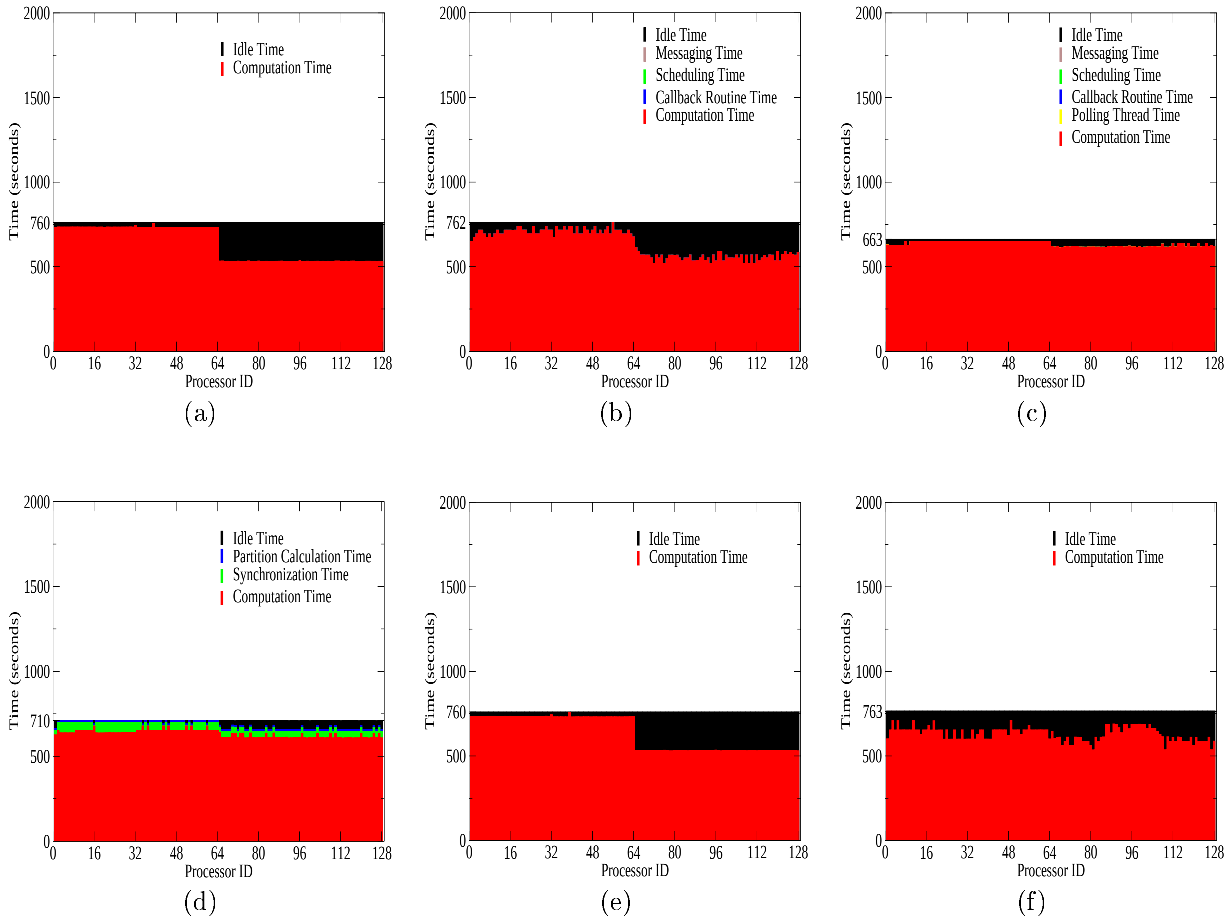


Figure 5: Benchmark performance for 50% initial imbalance, with the computational weight of heavy work units 20% higher than that of lighter ones. (a) No Load Balancing, (b) PREMA with explicit load balancing, (c) PREMA with implicit load balancing, (d) ParMETIS, (e) Charm++ with no synchronization, and (f) Charm++ with 4 synchronization points.

balancing policies that maximize data-locality while minimizing node work load imbalances.

References

[1] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. Architecture and evaluation of a load balancing framework for adaptive and asynchronous applications. To appear in IEEE TPDS, 2003.

[2] K. Barker, N. Chrisochoides, J. Dobbelaere, and D. Nave. Data movement and control substrate for parallel adaptive applications. *Concurrency Practice and Experience*, 14:77–101, 2002.

[3] 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 Mathematical Modeling*, 25:83–98, 2000.

[4] M. Bhandarkar and R. Brunner. Run-time support for adaptive load balancing. In *Proc. of 4th Workshop on Runtime Systems for Parallel Programming*, Cancun, Mexico, March 2000.

[5] M. Bhandarkar, L. Kalé, E. Sturler, and J. Hoeflinger. Object-based adaptive load balancing for mpi programs. Technical Report 00-03, Univ. of Illinois at Urbana-Champaign, 2000.

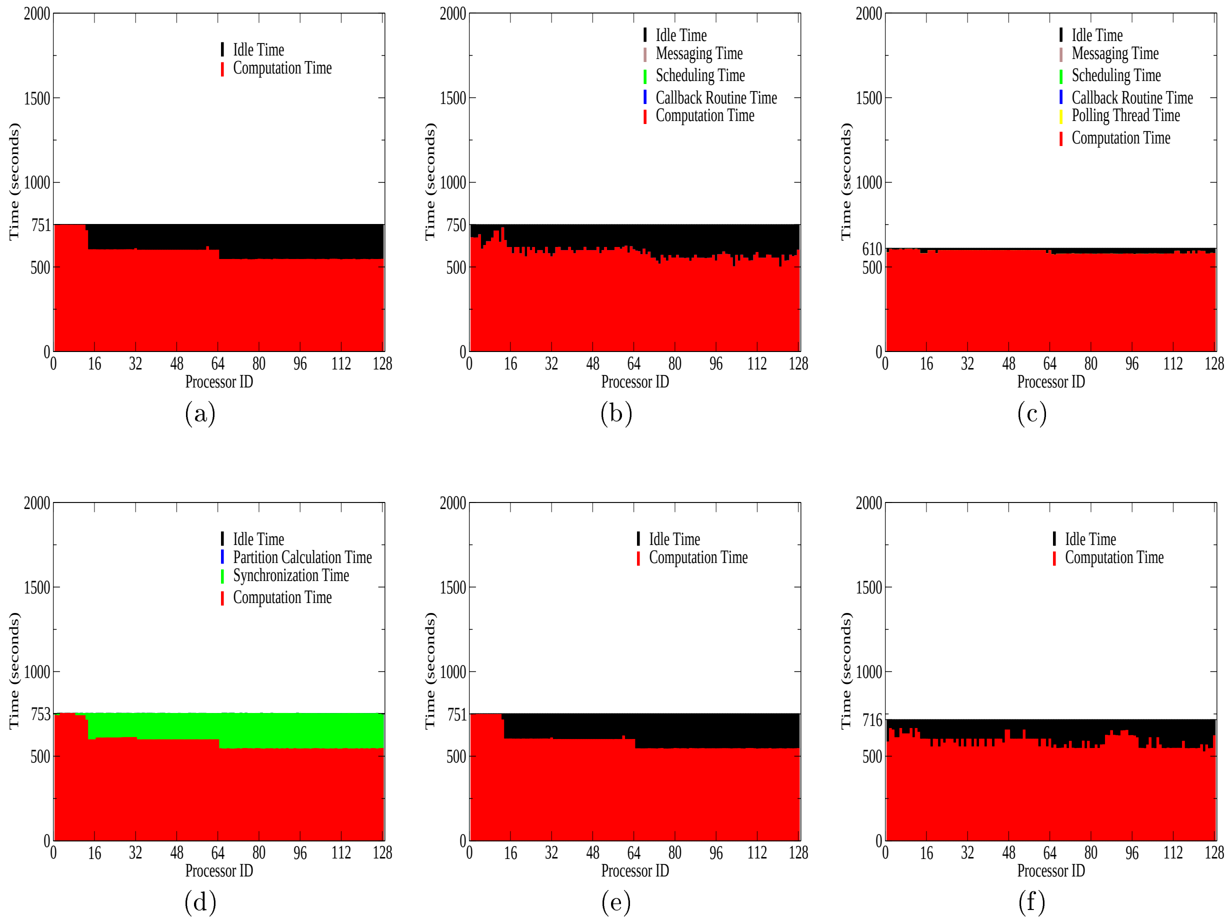


Figure 6: Benchmark performance for 10% initial imbalance, with the computational weight of heavy work units 20% higher than that of lighter ones. (a) No Load Balancing, (b) PREMA with explicit load balancing, (c) PREMA with implicit load balancing, (d) ParMETIS, (e) Charm++ with no synchronization, and (f) Charm++ with 4 synchronization points.

[6] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. Mobile object layer: A runtime substrate for parallel adaptive and irregular computations. *Advances in Engineering Software*, 31(8-9):621–637, August 2000.

[7] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Par. and Dist. Comp.*, 7(2):279–301, 1989.

[8] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Design of dynamic load-balancing tools for parallel applications. In *Proc. of the Int. Conf. on Supercomputing*, Santa Fe, May 2000.

[9] L. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA '93*, pages 91–108, 1993.

[10] L. Kalé, B. Ramkumar, A. Sinha, and A. Gursoy. The charm parallel programming language and system part ii – the runtime system. Tech. Rep. 95–03, Par. Prog. Lab., Dept. of Comp. Sci. Univ. of Illinois, Urbana-Champaign, 1995.

[11] G. Karypis and V. Kumar. Metis: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Dept. of Comp. Sci, Univ. of Minnesota, 1995.

[12] S. Kohn and S. Baden. Parallel software abstractions for structured adaptive mesh methods. *Journal of Par. and Dist. Comp.*, 61(6):713–736, 2001.

[13] The University of Notre Dame LAM Team. Lam/mpi parallel computing. http://www.mpi.nd.edu/lam/.

[14] E. Luque, A. Ripoll, A. Cortes, and T. Margalef. A distributed diffusion method for dynamic load balancing on parallel computers. In IEEE CS Press, editor, *Proc. of EUROMICRO Workshop on Parallel and Distributed Processing*, San Remo, Italy, January 1995.

[15] L. Oliker and R. Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Par. and Dist. Comp.*, 52(2):150–177, 1998.

[16] M. Parashar and J. Browne. Dagh: A data-management infrastructure for parallel adaptive mesh refinement techniques. Technical report, Dept. of Comp. Sci., Univ. of Texas at Austin, 1995.

[17] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel diffusion schemes for repartitioning of adaptive meshes. Technical Report 97-014, Univ. of Minnesota, 1997.

[18] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes. Technical Report 98-034, Dept. of Comp. Sci and Engr, Univ. of Minnesota, 1998.

[19] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. of the Intl. Conf. on Supercomputing*, 2000.

[20] K. Schloegel, G. Karypis, V. Kumar, R. Biswas, and L. Oliker. A performance study of diffusive vs. remapped load balancing schemes. In *ISCA 11th Intl. Conf. on Parallel and Distributed Computing Systems*, pages 59–66, 1998.

[21] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Int. Symp. on Comp. Arch.*, pages 256–266. ACM Press, May 1992.

[22] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Par. and Dist. Comp.*, 47:102–108, 1997.

[23] I. Wu. *Multilist Scheduling: A New Parallel Programming Model.* PhD thesis, School of Comp. Sci., Carnegie Mellon University, Pittsburg, PA 15213, July 1993.