2004

Runtime support for load balancing of parallel adaptive and irregular applications

Kevin James Barker
College of William & Mary - Arts & Sciences

RUNTIME SUPPORT FOR LOAD BALANCING OF PARALLEL ADAPTIVE AND IRREGULAR APPLICATIONS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

Kevin James Barker

2004

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Kevin J. Barker

Approved, April 2004

Nikos Chrisochoides
Thesis Advisor

Adolfy Hoisie
Los Alamos National Laboratory

Phil Kearns

Dimitrios Nikolopoulos

Xiaodong Zhang

To my parents...

Table of Contents

ACKNOWLEDGMENTS

The work contained in this thesis was supported by grants from the National Science Foundation (NSF) Directorate for Computer and Information Science and Engineering (#CCISE-9726388), NSF Information Technology Research (#ACI-0085969), Career Award #CCR-0049086, Research Infrastructure #EIA-9972853 and Next Generation Software #EIA-0203974. Additional funding was provided by the University of Notre Dame Arthur J. Schmidt fellowship.

The work was performed [in part] using the computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the National Science Foundation, and the state of Virginia's Commonwealth Technology Research Fund. Particular appreciation goes to Tom Crockett, who was tireless in helping us understand the SciClone cluster computing resource, and never complained when we managed to crash it! Additional work was conducted on computational resources made possible by grants from International Business Machines (IBM) Shared University Research program.

The author would also like to thank Andrey Chernikov, Brian Holinka, and Démian Nave for their work in parallel mesh generation which provided not only the impetus for the design of the PREMA runtime system, but also the test applications and benchmarks for measuring its performance. The availability of "real-world" applications for performance measurements greatly strengthens the quality of this research.

Thanks also to Jeff Dobbelaere and Andriy Fedorov for their work, particularly in the low-level message passing components of the PREMA architecture. The author also wishes to thank Jeff Squyres for his help with all things MPI, and for providing critical assistance at many points during the course of this work. Additional appreciation goes to Houtan Bastani for assistance in porting the lower components of the PREMA runtime architecture to the Microsoft Windows/NT/2000 platform.

N-body simulation code was provided by the Engineering Research Center at the Mississippi State University. Gratitude goes to Mahadevan Balasubramanian for his help in this area. The three dimensional pipe model cited within this thesis arose out of work conducted by the ITR/ACS Adaptive Software Project, which is composed of researchers at Cornell University, Mississippi State University, the College of William and Mary, the University of Alabama, the Ohio State University, and Clark-Atlanta University.

Finally, the author extends sincere gratitude to his research advisor, Nikos Chrisochoides, who provided immeasurable direction and guidance (and funding!) during his tenure as a graduate student. The author also wishes to thank his committee members, whose insights and contributions added greatly to the quality of this work.

List of Tables

List of Figures

ABSTRACT

Applications critical to today's engineering research often must make use of the increased memory and processing power of a parallel machine. While advances in architecture design are leading to more and more powerful parallel systems, the software tools needed to realize their full potential are in a much less advanced state. In particular, efficient, robust, and high-performance runtime support software is critical in the area of dynamic load balancing. While the load balancing of loosely synchronous codes, such as field solvers, has been studied extensively for the past 15 years, there exists a class of problems, known as *asynchronous* and *highly adaptive*, for which the dynamic load balancing problem remains open. As we discuss, characteristics of this class of problems render compile-time or static analysis of little benefit, and complicate the dynamic load balancing task immensely.

We make two contributions to this area of research. The first is the design and development of a runtime software toolkit, known as the Parallel Runtime Environment for Multi-computer Applications, or PREMA, which provides interprocessor communication, a global namespace, a framework for the implementation of customized scheduling policies, and several such policies which are prevalent in the load balancing literature. The PREMA system is designed to support *coarse-grained* domain decompositions with the goals of portability, flexibility, and maintainability in mind, so that developers will quickly feel comfortable incorporating it into existing codes and developing new codes which make use of its functionality. We demonstrate that the programming model and implementation are efficient and lead to the development of robust and high-performance applications.

Our second contribution is in the area of performance modeling. In order to make the most effective use of the PREMA runtime software, certain parameters governing its execution must be set off-line. Optimal values for these parameters may be determined through repeated executions of the target application; however, this is not always possible, particularly in large-scale environments and long-running applications. We present an analytic model that allows the user to quickly and inexpensively predict application performance and fine-tune applications built on the PREMA platform.

RUNTIME SUPPORT FOR LOAD BALANCING OF PARALLEL ADAPTIVE AND IRREGULAR APPLICATIONS

Chapter 1

Introduction

While hardware advancements are leading to ever more powerful parallel computing platforms, the software to utilize these new systems lags further behind. The building blocks needed to implement adaptive and irregular applications are largely missing, and development of these components takes years and a great deal of expertise. An example of this is the lack of parallel software to efficiently handle the discretization (i.e. mesh generation) of 3D complex domains with rapidly changing geometry and/or topology. The complexity of efficient parallel codes for adaptive applications such as 3D unstructured mesh generation increases dramatically compared to the corresponding sequential code due to the dynamic, data-dependent, and irregular computation and communication requirements of the algorithms. This inherent complexity makes development using existing parallel programming paradigms, such as message passing, both time-consuming and error-prone, especially without the aid of parallel languages, software tools, and libraries.

Of critical importance is the issue of dynamic load balancing. Imbalance caused by either systemic issues (as are frequent in multi-user or heterogeneous environments) or algorithmic adaptivity can significantly reduce the efficiency of a parallel computation. Placing the monitoring and data/computation migration burden on the application can lead

to code which is error-prone, difficult to maintain, and potentially restricted to a particular execution environment. Furthermore, it is most likely not the case that an application developer is an expert in the field of load balancing; applications may therefore be written using a sub-optimal load balancing technique.

That is not to say that the research community has not begun to address this issue. The past two decades have seen advancements in the areas of interprocessor communication [75, 21], data partitioning and load balancing [64, 103], and modeling techniques [47]. However, the majority of this work is targeted for *synchronous* or *loosely synchronous* parallel applications. For a significant class of codes, called *asynchronous* and *highly adaptive*, the load balancing problem remains open.

The computation and communication patterns exhibited by such codes evolve as the application progresses (e.g., the fidelity of a mesh must increase around a crack boundary during Adaptive Mesh Refinement), rendering compile-time or static analysis of little benefit. In addition, asynchronous applications impose a large penalty on any synchronization introduced for the purpose of exchanging processor load information. These codes require a runtime toolkit which can provide efficient, high-performance, and scalable load balancing, yet is flexible enough to be useful to a variety of application types. In this thesis, we will describe our work in developing the Parallel Runtime Environment for Multi-computer Applications (PREMA). Our experiments will show that its design and implementation lead to reductions in application complexity and significant gains in parallel performance and efficiency.

1.1 Fundamental Issues In Parallel Computing

Software tools used to implement and maintain the building blocks of parallel adaptive codes can greatly ease the burden placed on parallel application developers. These tools should aid developers in addressing the following fundamental issues in parallel computing:

- *Domain partitioning*: To parallelize an application, the computation invoked by that application must be distributed among the available processing resources. This can be done either by decomposing the *iteration space* of a loop-based program, or by decomposing the *data domain*, as is the case, for example, with parallel mesh refinement.

- *Data locality*: While processors can quickly access the data stored in their local memories, the performance penalty for accessing non-local data is too large to be ignored. Furthermore, adaptive and irregular applications are not amenable to compile-time analysis, making a parallel runtime system with the ability to facilitate the exploitation of data locality imperative.

- *Communication and synchronization*: Codes running on MIMD parallel systems, such as Massively Parallel Processing (MPP) machines or Clusters of Workstations (COWs), are typically made up of independently executing processes that operate in concert, and therefore need to communicate during the runtime of the application. Also, synchronous or loosely synchronous applications often progress as a series of phases, at the end of which processors in the parallel system may need to synchronize in order to update data and begin the next phase. System software must provide

tools which developers can use to implement the unpredictable communication within a single phase, as well as basic tools to synchronize between phases when necessary.

- *Communication latency tolerance*: The large physical size of many parallel machines (and the clusters of machines often used for parallel computations) often implies large message passing latencies, the effects of which are exacerbated by increasing processor speeds in relation to network latencies. While single-processor performance has been accelerating according to Moore's Law for some time, the same cannot be said for network latencies. Technologies such as Myrinet [93, 29], Infiniband [7], and Giganet [55] have gone some way in attempting to close the gap between processor and network performance; however, they are by no means a complete solution. Furthermore, it does not appear that this problem will be solved by network technology in the foreseeable future. Once again, compile time analysis is of little help in hiding these latencies, due to the unpredictable and dynamic communication patterns present in adaptive applications such as unstructured mesh generation.

- *Global namespace*: Large-scale parallel applications typically create data structures whose elements are distributed among the processors' address spaces. Computation often requires the coordination of multiple data structure nodes, which, in a distributed address space, may mandate communication between several processors. Without system software support, the application will need to keep track of which data structure elements are located on which processors. In the presence of data migration, for instance due to dynamic load balancing, this bookkeeping can become very complex, leading to error prone application implementations. It is therefore critical for runtime

system software to relieve the application developer of this burden.

- *Load balancing*: Large-scale, high-performance parallel machines generally consist of many processing nodes which coordinate in a loosely synchronous fashion. In order to best utilize the available computing resources, it is important to avoid overloading some processors with pending work while leaving others idle. This is equivalent to stating that it is important to minimize the maximum runtime of any single processor, thus minimizing the runtime of the entire parallel application. The efficient implementation of adaptive applications which dynamically balance processor workloads requires a great deal of effort above what is required to implement static applications which execute in the presence of load imbalances. Therefore, efficient software tools that allow developers to quickly and easily incorporate load balancing into their applications are of critical importance.

Of these, load balancing is the principle concern of this thesis. However, it is important to recognize the relationships between issues during the design of any runtime toolkit. For instance, load balancing is often a multi-dimensional optimization problem in which not only processor cycles, but memory and network utilization as well as data dependencies between tasks must be considered.

1.2 Parallel Runtime Environment for Multi-computer Applications

The PREMA runtime toolkit is a middle-ware software library which provides a set of tools to the application via a concise and intuitive interface. It provides point-to-point com-

munication, a global namespace, and a preemptive load balancing framework targeted to support adaptive and asynchronous applications which employ a coarse-grained data decomposition in which the original domain is broken into N sub-domains, where N is greater than the number of available processors, P (known as *over-decomposition*). Computation is invoked on each sub-domain in a message-driven manner, using PREMA's message passing interface. This computation represents the load which composes the application; migrating a sub-domain implicitly migrates computation, allowing the runtime system to implement dynamic load balancing.

The PREMA runtime toolkit we describe is composed of several layers, which address the issues just discussed according to the principle of *separation of concerns*. Each software layer is assigned a specific task; application developers may then utilize only that functionality which they deem necessary. However, if more sophisticated runtime support is later required, it may be easily added with minimal changes to existing application code. Briefly, each runtime component may be described as follows.

1.2.1 Communication Infrastructure

We begin by laying the foundation of our runtime system hierarchy by presenting the *Data Movement and Control Substrate* (DMCS) [16] and the *Mobile Object Layer* (MOL) [50]. The DMCS software provides a lightweight and easily portable single-sided communication and data movement substrate. Applications making use of DMCS communicate between processors using the Active Messages [152] message passing paradigm, in which user-defined *handlers* are executed on target processors upon message arrival. DMCS provides *split-phase* operations which allow communication to potentially overlap with computation and

mitigate the effects of message latency.

The MOL extends this communication model by providing a global namespace and automatic message forwarding in the presence of object migration. The MOL is flexible enough to allow application developers to easily exploit data locality in their parallel codes. Specifically, the MOL provides the concept of a *mobile object*, which is any piece of application-defined data which is free to move throughout the parallel system under the control of the application or a higher-level system library. Applications are able to communicate directly with mobile objects via their unique *mobile pointers*, regardless of their present location, even if the target mobile object is presently in the process of migration.

By providing single-sided, Active Messages style communication primitives, data migration operations, and a global namespace, both DMCS and the MOL are designed to simplify the implementation and maintenance of software building blocks needed for parallel adaptive applications. We will show that the communication overhead involved in using DMCS and the MOL will not contribute significantly to the runtime of a parallel application.

1.2.2 Load Balancing

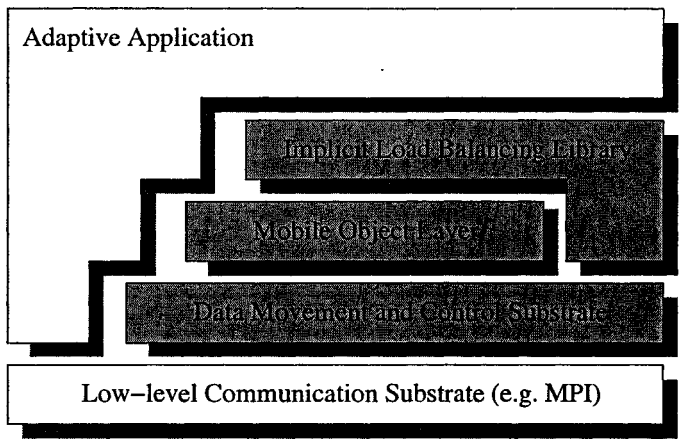
With the DMCS and MOL software layers in place, we present our load balancing framework. There are two strategies commonly used to integrate load balancing with parallel applications. The first is known as *explicit* load balancing, referring to the fact that the application is explicitly responsible for implementing the load balancing decision making and task migration. The advantages of this strategy lie in the fact that, because the load balancer is so closely tied to the application, it has access to accurate work load and data locality information, and is therefore able to make informed load balancing decisions. The

drawback to explicit load balancing methods is that the application must deal with functionality that is not strictly in the application domain. Programmers are forced to write load balancing code, about which they may not be familiar and are most likely not experts. This may greatly complicate the application code, making it difficult to maintain and understand, and increase the effort required to create efficient code.

The second load balancing strategy is to incorporate the services provided by a dedicated load balancing library into the application. This method is known as *implicit* load balancing, and has the advantage that application programmers no longer need to implement load balancing functionality, reducing the complexity of application code. In addition, because the load balancing library is implemented by domain experts, it can contain finely tuned and efficient load balancing algorithms. However, because the load balancing code has no knowledge of the application domain and must work with general programs, it is unable to make use of application-specific knowledge to migrate data with maximal efficiency. It is also true that implicit load balancers are often unable to obtain truly accurate processor load information and are therefore unable to gauge the load of the system processors with complete accuracy.

We present the *Implicit Load Balancing* (ILB) [15, 17] component of the PREMA architecture, which provides a framework for implicitly load balancing dynamic and adaptive applications. The ILB library takes the load balancing decision making and data migration away from the application, greatly reducing the code complexity and development time for load-balancing adaptive codes. Specifically, the ILB library was designed in accordance with the following goals:

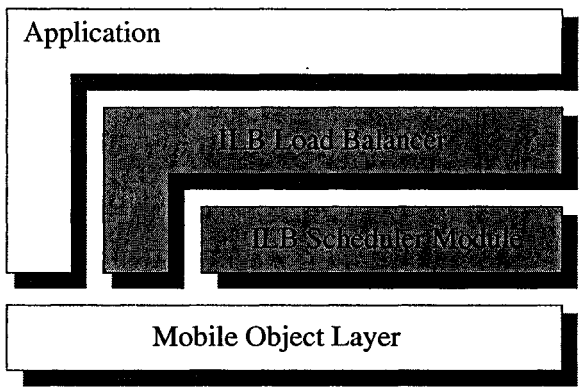
Figure 1.1: SOFTWARE RUNTIME SYSTEM ARCHITECTURE



- *High performance*: The ILB framework must allow for the implementation of high performance scheduling modules. In other words, scheduling algorithms which result in a nearly optimal work load distribution (given a particular initial task distribution) must be implementable. A corollary to this is that the schedulers that are possible within our framework must be able to significantly reduce the number of idle processor cycles in the parallel system.

- *Evolutionary approach to load balancing*: For applications written using the programming model provided by the MOL, it is a simple step to use the dynamic load balancing functionality provided by the ILB library. We can accomplish this by guaranteeing that the communication model, interface, and programming constructs (such as mobile objects and mobile pointers) provided by the MOL are mirrored in the ILB library. In the event of unacceptable runtime load imbalance, application developers may quickly and easily make use of the ILB library's implicit load balancing functionality without needing to rewrite significant portions of existing application code.

Figure 1.2: SCHEDULER MODULE "PLUG AND PLAY" ARCHITECTURE



- *Flexibility*: The ILB library actually provides a load balancing interface and framework which can be used in conjunction with a wide variety of scheduling algorithms. Figure 1.2 depicts the Scheduler module in the overall ILB architecture; by replacing the Scheduler, applications are able to customize the load balancing policy without needing to modify any application code. This allows application developers to quickly and easily experiment in order to find the most effective load balancing method.

- *Low overhead*: The overhead incurred by the ILB framework should be kept as low as possible. While an application may implement any desired scheduling algorithm, the Scheduler module should not have to conform to an interface that precludes high performance.

In this thesis, we will highlight the components of the load balancing framework which are the most important in achieving these goals. We will also subject the load balancing software to a series of tests using several benchmark programs and "real world" applications, including mesh generation codes and N-body simulators, under a variety of imbalanced conditions. Through these experiments, we will establish that the PREMA software achieves

its purpose of providing efficient and high-performance load balancing support for adaptive and asynchronous applications.

1.2.3 Analytic Modeling of Load Balancer Performance

In order to make the most effective use of the runtime software we demonstrate, certain runtime parameters must be set off-line which will govern the interaction between the application and the runtime system. Complicating the role of the application developer is the fact that these variables typically depend on characteristics of the specific application and hardware runtime environment. While optimal parameters can be determined through repeated experimental executions of the particular target application, such a procedure is time consuming, potentially expensive, and often prohibitive for large systems and applications. Therefore, we present a runtime analysis technique based on an analytic modeling method which allows the user to quickly predict application performance given certain assumptions. Such a tool allows developers to fine tune the performance of applications built using the PREMA software in the field.

1.3 Contributions of This Thesis

The first contribution we provide is the development of a programming model which is intuitive from a developer's standpoint and allows for the efficient construction of runtime tools. Through close contact with application developers within our research group, we have determined which operations are of the most benefit to adaptive and asynchronous applications, and have incorporated them into our toolkit. We also distinguish ourselves from other research projects by providing a single-threaded programming model based on

currently existing, widely used programming languages (C/C++), as we feel this removes many of the hurdles which stand in the way of developer adoption. We do this in such a way as to not preclude preemptive migration decision making, which is necessary for the efficient load balancing of coarse-grained tasks. Through a comparative analysis with other prevalent load balancing techniques, we demonstrate that PREMA is ideally suited for the asynchronous and adaptive applications we target. Finally, based on our literature review, we have determined that PREMA is the only runtime toolkit which provides this specific feature set.

Our second contribution is the development of the PREMA system itself. First, we will show that, using our runtime system, application developers are able to control the extent to which the runtime system manages their data migration patterns. Using only the basic functionality of the PREMA system allows application developers to remain in strict control over data locality and data migration. In the event that runtime analysis shows performance to be unacceptably hampered by dynamic load imbalance, developers may choose to incorporate load balancing, allowing data migration to be handled exclusively by the runtime system. Our evolutionary approach to runtime system design means that minimal changes to application code are necessary in order to make use of the more advanced functionality provided by the system.

A second characteristic of PREMA's implementation is its scheduling flexibility. PREMA incorporates not just a single load balancing algorithm or family of algorithms, but is instead a load balancing framework which allows developers to quickly modify existing scheduling policies or create new ones. These customized schedulers can be easily incorporated into the existing runtime framework. Because the application itself is isolated from the schedul-

ing policy (Figure 1.2), minimal modification of existing application code is required. In addition, PREMA provides several of the best-known scheduling policies available from the literature, which we will describe and examine in our Experimental Results chapter.

The final implementation characteristic is performance: we test the effectiveness of our load balancing mechanism with a variety of applications and demonstrate its performance relative to what is in the field today. We also prove its applicability to a variety of scientific codes. We are concerned about performance from two standpoints: minimizing the overall application runtime, and minimizing the performance impact caused by the costs incurred by the runtime system. In other words, the communication and load balancing operations provided by the PREMA runtime system must not come with a prohibitive cost. Such a cost may be a result of overheads within the operations themselves, or may stem from inflexibility or impracticality of the runtime system's programming model. We are concerned with *end-to-end* performance of the overall application, not simply the performance of individual operations taken in isolation. Therefore, we will endeavor to use "real-world" codes whenever possible to evaluate the efficiency of our runtime system software.

The contributions stemming from our work in the area of modeling dynamic load balancing performance are two-fold. First, we present an analytic modeling framework which can be used to study a wide variety of load balancing policies and implementations on an array of parallel machines ranging from Clusters of Workstations (COWs) to massively parallel MPPs. We then calibrate this model using parametric values corresponding to a particular cluster, and verify its accuracy through comparisons with actual run times of both a parallel benchmark [15] and a Parallel Constrained Delaunay Triangulation (PCDT) program. With this model established and verified, we provide our second contribution,

which is a parametric study focusing on two of the more important runtime parameters: level of *over-decomposition* and value of the preemptive *polling quantum*. We will explain both of these parameters in greater detail in the relevant Chapter.

Chapter 2

Related Work

We are interested in examining related research in each section of this project, from processor-to-processor communication through load balancing and any work done in the area of analytic and stochastic modeling of dynamic load balancing systems. Below, we will look at the current state of research in these areas.

2.1 Communication Infrastructure

The primary goals we have for the design of DMCS are (i) *high performance*, (ii) *flexibility* and *ease-of-use*, and (iii) *portability* (see Chapter 4, Section 4.1). Existing communication paradigms and systems tend to fall into one of two broad camps regarding these issues: those that are geared towards high performance at the expense of usability (e.g. the Low-level Application Programming Interface (LAPI) [84, 143]), and those that sacrifice performance in favor of easing the burden placed on the application programmer (e.g. software Distributed Shared Memory (DSM) systems such as Treadmarks [3]). Both approaches present difficulties to the application developer who requires a high degree of performance, maintainability, and portability, but does not possess the time nor the inclination to master the

complex intricacies of a large range of complex communication subsystems.

The Message Passing Interface (MPI) [75, 111, 149, 135, 110] has become the *de facto* standard for applications geared towards a distributed memory environment by successfully tackling the issues surrounding portability and ease-of-use, but for the types of applications we wish to study, MPI presents some conceptual difficulties. MPI does not support a flexible Remote Procedure Call (RPC) communication paradigm which would greatly simplify the development of runtime systems for dynamic and unstructured applications. In addition, MPI does not address the issue of dynamic resource management[1]. The *binary* communication protocol defined by the MPI standard is similar to that provided by the P4 [38] system. Applications match *send* operations with explicit *receive* operations on the target processor. This communication model presents a problem for adaptive applications, in which communication patterns cannot be known until runtime.

The Parallel Virtual Machine (PVM) [150, 119, 131], developed at Oak Ridge National Laboratory, is another popular message passing environment. Unlike DMCS, PVM is designed with heterogeneity in mind; it uses a simple send/receive library to control an arbitrary number of processes on possibly remote computers[2]. PVM supports both blocking and non-blocking sends, along with non-blocking receives. However, message latency using PVM can be on the order of milliseconds, due to the fact that a daemon running on each node is responsible for coordinating communication and process creation activities. In situations in which PVM can be implemented on top of an efficient low-latency communication package, as may be the case in an MPP environment, this latency may be reduced.

[1]The MPI-2 standard was not defined at the time DMCS was designed.

[2]Scalability in PVM 3.0 is actually limited to 4096 hosts and 262143 tasks per host [131]. However, it is also possible that the underlying operating system will impose additional limits

Madeleine [32, 8] is designed to support distributed, multi-threaded applications. In order to facilitate portability, Madeleine's architecture is similar to DMCS's two-tiered design, with platform-specific code isolated from the application. However, like MPI, Madeleine implements a binary communication protocol in which *send* operations must be matched with *receive* operations on remote nodes. In addition, Madeleine is designed to be used as a target for RPC-based multi-threaded environments such as Nexus and not used directly by user applications.

RPROC [157, 119] is an earlier system which, like PVM, is designed to send messages between heterogeneous computers. However, RPROC supports a single-sided Active Messages paradigm, making it similar to DMCS. RPROC assumes only an unreliable file transfer mechanism between processors, meaning that messages are delivered as a pair of transferred files. The first file transfer would contain the actual message, while the second was used as an acknowledgement mechanism. This file transfer mechanism was invisible to the user and can therefore be replaced by TCP/IP or another message passing mechanism in situations where such a mechanism is available.

The P4 macros [119, 38, 37] developed at Argonne National Laboratory was the first attempt to develop a true portability platform. Through the use of macros, P4 is able to avoid function call overhead when sending and receiving messages; however, all communication operations are blocking. P4 provides support for shared memory, message passing, and mixed systems such as loosely coupled shared memory clusters.

The PARMACS [40, 119] library developed from the work done with P4, and supports both synchronous and asynchronous *send* routines. PARMACS supports a binary communication protocol in which messages can be received either according to the sender's process

ID or by message tag. In addition, some support for collective operations are provided. A feature that distinguishes PARMACS from other systems is its extensive support for various application topologies. For instance, macros are provided to map processes onto rings and grids of two or three dimensions. PARMACS strongly influenced the topological aspects found later in MPI.

Express [73, 119] grew from work on the Crystalline Operating System at Caltech, and was later managed by the Parasoft company. The early goals for Express were to implement it on a wide range of parallel architectures and attempt to gain the best possible performance on each. This explicit goal of portability is shared by DMCS. Later, Parasoft emphasized usability and attempted to hide many of the details inherent in Express. This led to development of mapping and communication libraries for different processor topologies. Finally, Express moved beyond simple message passing and began to tackle problems such as parallel I/O and dynamic load balancing.

In addition to these systems, Zipcode [148], the Communication Kernel [128], and Panda [25] have been developed to facilitate the development of parallel languages and higher level parallel languages. Of these, only Panda provides single-sided communication primitives. However, Panda does not provide a single-threaded runtime environment, which is one of the design points of DMCS.

Another class of systems, including Horus [133, 134], Isis [26], and the Collective Communication Library (CCL) [11] are designed to provide efficient collective communication and reduction operations. Systems such as Linda [63], Cooperative Data Sharing (CDS) [68, 69], and Cooperative Shared Memory [89] are designed to effect communication through shared memory regions. Both classes contrast with the point-to-point, explicit message passing

design implemented by DMCS.

These, and other, parallel programming issues were addressed by the *POrtabile Runtime Systems* (PORTS) consortium [54], where different approaches to communication were identified. The notable communication models that arose from these discussions are: (i) a *thread-to-thread* communication approach, (ii) a *Remote Service Request* paradigm.

CHANT [85] implements thread-to-thread communication on top of portable message passing software layers such as p4, PVM [21], and MPI. The efficiency of this mechanism depends critically on the implementation of message polling. There are three common approaches to polling for messages: (i) individual threads poll until all outstanding receives have been completed, (ii) the thread scheduler polls before every context switch on behalf of all threads, and (iii) a dedicated thread polls for all registered receives. For portability, CHANT supports the first approach.

The Remote Service Request model has been implemented by a number of systems. NEXUS [77, 76] decouples the specification of the destination of communication from the specification of the thread of control that responds to it. Messages are handled by *message handlers* which are threads or routines registered by the user with the runtime system and are invoked upon message receipt. The handler possesses a pointer to a user-level buffer into which the contents of the message should be placed. Handler threads are scheduled in the same manner as computation threads. The RPC communication model is also incorporated into systems such as Peregrine [94], Illinois Fast Messages (FM) [129], and Active Messages [152, 117], and has been adopted by DMCS. For adaptive and unstructured applications, not needing to match *send* operations with explicit *receive* operations is a great advantage.

In addition, the *hybrid* communication model implemented by TULIP [20] essentially combines the thread-to-thread and RSR-driven communication paradigms. In the runtime substrate, TULIP provides basic communication via global pointers and remote service requests. Threads are introduced at the pC++ language level.

The design for DMCS was driven by the need for a runtime library for adaptive applications such as parallel adaptive mesh generation. However, DMCS was not designed to replace any of the systems mentioned, but instead exploits the low-latency constructs of the underlying communication subsystem and provides operations which handle the special requirements of adaptive applications. To achieve low-latency and to provide a programming model that is the most familiar to application developers, we have decided to support a single-threaded communication paradigm. Note that "single-threaded" in this case means a single communication thread and an arbitrary number of computation threads. In this way, developers may integrate any desired thread package into an application, latency associated with DMCS operations is minimized, and portability and maintainability issues (a primary consideration in the design of DMCS) are minimized.

2.2 Global Namespace and Object Migration

Our goals for the Mobile Object Layer (MOL) include (i) *to provide a global namespace*, (ii) *to preserve the Active Messages communication model*, and (iii) *to provide efficient message forwarding in the context of data migration.* There are several existing systems that share similar aspirations. Languages such as CC++ [44] and Split-C [57] have integrated global pointers at the language level and have shown that global pointers can be used

to build efficient distributed data structures. However, neither language provides direct support for object migration. Emerald [97] is a object-based language and system which provides language support for object mobility. In order to locate mobile objects, a scheme of forwarding addresses are used, which operate in a manner very similar to that employed by the MOL. However, while the Emerald system has decided to follow a route that has lead to the development of a new programming language, the MOL is simply a lightweight library, allowing applications to be written in an existing language such as C/C++.

Orca [10] is language for implementing coarse-grained, explicitly parallel applications on distributed systems. Shared data structures are encapsulated in objects called "shared data-objects", which may be migrated or replicated by the system without any intervention from the user. Instead of replicating data objects, the MOL employs an efficient message forwarding mechanism, which saves the runtime system the burden of maintaining object consistency.

CHAOS++ [45] is a portable, object-oriented runtime library designed to support the construction of dynamic distributed data structures. CHAOS++ defines *globally addressable objects* whose ownership is assigned to one processor. Shadow copies reside on other processors and are used to cache the contents of their remote counterparts, so that accesses are local. The contents of these shadow objects are updated through explicit calls to CHAOS++ routines, leaving the decision concerning when to synchronize objects up to the application. ABC++ [6] is another system which allows mobile objects to migrate away from their original "home nodes", but communication is required with the "home node" each time a message is sent to the object.

The Nexus [76, 77] approach is based on global pointers which act as endpoints for

communication. As with the MOL, Remote Service Requests specify a user handler function and a global pointer which acts as the target of the message. However, global pointers are not automatically updated as objects move. In addition, the MOL is designed to be much lighter weight, removing functionality such as authentication that is not applicable to our target runtime environment.

Because the MOL implements a global namespace in a distributed environment, it is helpful to briefly touch upon some of the research that has been done in the Distributed Shared Memory (DSM) arena. The C Region Library (CRL) [95] is an all-software DSM system designed for message-passing distributed computers. Applications share data through "regions", which are arbitrarily sized areas of memory defined by the application. Regions of memory are cached in local copies, which are kept consistent through a coherency protocol. Accesses to regions are grouped into "operations", and modifications to a region of memory are only visible to other processors after the end of a write operation. The Amber [46] system takes a slightly different approach, in that a distributed address space is achieved by partitioning the virtual address space. Each processor is forced to use disjoint regions of the address space for heap allocation of dynamic objects. In this way, Amber can determine which accesses are local and which will involve communication with remote nodes. One drawback to this approach is the sparse use of a very large virtual memory space. Amber programs are written using an object-oriented subset of C++ supplemented with primitives for managing concurrency and distribution.

COOL2 [2] is a distributed object-oriented computing system that extends the single address-space model of computing to a distributed environment. Objects in COOL2 are always manipulated through their local pointers, even if they are not local objects. COOL2

is therefore like Amber in that local object identifiers are extended to the network, but differs from Amber in that COOL2 uses virtual memory to maintain object consistency. Finally, Tempest [132] is an interface for shared memory whose goal is to provide the full range of shared-memory semantics in user-level software. When a processor accesses a shared page on a remote node, the reference invokes a user-level page-fault handler which allocates a new physical page, and maps it to the shared virtual address. The home processor ID for the data is looked up in a distributed table; this home node will perform any necessary coherency actions and transport the data to the faulting processor.

Finally, hardware systems like FLASH [109] integrate both message passing and global shared memory into a single architecture. The key feature of the FLASH architecture is the MAGIC programmable node controller which connects processor, memory, and network components at each node. MAGIC is an embedded processor that can be programmed to implement both cache coherence and message passing protocols.

2.3 Load Balancing

Load balancing methods can be divided into two classes: *static* methods and *dynamic* methods. The static methods [22, 30, 71, 145, 52, 53, 78] make use of *a priori* knowledge of the computation in order to partition the work-load into a number of chunks (or work-units) of roughly equal size. Static methods usually use generic graph partitioning libraries like Metis [141, 139, 140, 138], Chaco [88] and Jostle [153] to partition an initial computation graph into a predetermined number of subgraphs (which represent either sub-meshes [52, 53]

or sub-matrices [72, 39, 81, 136]) in such a way as to minimize size of separators[3] while they maintain an equal distribution of vertices (i.e., elements or non-zero matrix coefficients).

However, in the case of adaptive computations like adaptive PDE solvers with h-refinement and/or p-refinement, it is not possible to know beforehand the computational requirements, rendering the static load balancing methods ineffective. For these computations, in order to prevent the overall computation from becoming severely imbalanced, it is necessary to migrate computational load from overloaded processors to underloaded processors during the course of the application runtime. This is known as *dynamic* load balancing.

There are many dynamic load balancing methods; some are based on re-partitioning using the static load balancing methods we have mentioned earlier, while others are designed around application-specific methods like Recursive Coordinate Bisection (RCB) [22, 88], Unbalanced Recursive Bisection (URB) [88, 96], and Recursive Inertial Bisection (RIB) [88, 145]. These methods are often described as having a *global view* of application state, and divide the geometric problem space into pieces recursively through the use of cutting lines in two dimensions or planes in three dimensions. The URB and RIB methods are derivations of the RCB method in which the direction and location of the cut are altered according to particular criteria, such as evening the amount of work represented on either side of the cut or cutting perpendicular to the coordinate direction in which the sub-domain is the longest with the desire of minimizing the number of elements that lie along the cut edge.

Other dynamic load balancers avoid the potential bottleneck associated with a global

[3]In the case of mesh decomposition the communication between sub-meshes is a function of the number of interface elements or the number of edges cut between sub-meshes; minimizing the interface elements minimizes the communication between sub-meshes.

view and restrict themselves to a *local view* only. The most popular of these methods is *Diffusion* [56, 58, 60, 87, 88, 90, 91, 107, 108, 156, 155, 162, 163] which divides the processor pool into a set of overlapping load balancing domains. Diffusion load balancing methods come in two types, namely *Sender Initiated* (SI) or *Receiver Initiated* (RI). Each processor in a SI diffusion method acts independently to migrate excess workload to underloaded neighbor processors. Typically, each processor informs its neighbors of load levels throughout the execution of the parallel program. The profitability of load balancing is determined by computing the average load in the local domain; if the local processor's work load exceeds the average by a specified threshold, then it can relocate excess load to its neighbors. Receiver Initiated (RI) diffusion allows any processor to begin the load balancing algorithm once its workload drops below a threshold. At this point, the underloaded processor sends load requests to each overloaded neighbor for a fraction of the total load surplus.

Another popular method used for dynamic load balancing is the *Gradient Method* [13, 35, 41, 121, 156], in which underloaded processors propagate their state throughout the system, and overloaded processors send a portion of their workload to the nearest lightly loaded processor. The scheme is based on two thresholds: a processor's state is considered to be underloaded if the local load level is below a *Low Water Mark* and overloaded if it is above a *High Water Mark*. Any other state is considered optimal. In addition, a node's *proximity* is defined to be the distance to the nearest underloaded processor, and is propagated from a node to its neighbors, eventually potentially reaching every processor. The proximity map is used to perform task migration; overloaded processors send work to their neighbors with the lowest proximity (assuming that neighbor has a workload less than some maximum). The potential problem of multiple processors sending load to the same

underloaded processor can be avoided with a two-stage commit protocol.

In addition, there are several methods based on strategies that make use of neighbor sets that vary during runtime. At any stage of the load balancing procedure, a processor is in communication with only a small set of other processors, balancing a subset of the computation in each stage. Two such methods are the *Dimension Exchange Method* [160, 122, 156, 159] and the *Hierarchical Balancing Method* [156].

Finally, some dynamic load balancing methods are based on more general approaches, like the tree-based methods presented in [113], which make use of global system information like processor loads and communication costs to construct a spanning tree of the weighted processor graph. From the spanning tree, a tree walking algorithm is used to calculate global load balancing information, and a load transfer algorithm is invoked to balance the computational load among processors while minimizing communication cost.

The goal of our research is not to present yet another dynamic load balancing strategy and its implementation. Instead, our focus is to present a runtime software support system that can be used to develop and easily evaluate dynamic load balancing methods for parallel adaptive applications like mesh generation and refinement. Next, we overview software systems with similar objectives.

Many of the systems discussed in Section 2.2 are able to provide the basis for load balancing systems. For instance, ABC++ [6], Split-C [57], CC++ [44], Amber [46], Emerald [97], and COOL2 [2] provide the basic software support for object migration or global namespace. While these systems can provide the needed infrastructure for load balancing systems, none of them assume responsibility for the decision making associated with balancing runtime load. We therefore describe a class of systems that provide direct support

for dynamic load balancing and are therefore more related to the PREMA load balancing framework we present in this thesis.

Multi-list Scheduling [158] model is designed to greatly reduce the effort of developing dynamic task scheduling routines. Essentially, the Multi-list Scheduling model replaces a single, centralized task list with a distributed task list, where each processor's local lists contain only those tasks which are local. These local *physical lists* are then merged into a single *virtual list* on each node. The order of the tasks in the physical lists, and therefore the virtual lists, are dependent upon the priorities assigned to each task relative to each processor in the system. There are also several optimizations possible, including merging physical lists whose tasks are always in the same order, and maintaining only the head of the virtual list on each processor in order to reduce communication costs. While the Multi-list Scheduler is flexible in providing "work-pulling" load balancing algorithms, it does not easily provide for the implementation of "work-pushing" algorithms, such as SI Diffusion. Our load balancing framework encompasses Multi-list Scheduling, and, in fact, we will describe the design and performance of our *Prioritized Multi-list* Scheduler. In addition, the ILB framework allows for the implementation of "work-pushing" methods such as SI Diffusion.

Another such load balancing framework is incorporated into the *Virtual Data Space* (VDS) package [60]. The VDS system provides support for various application paradigms, such as fork-join, weighted tasks, and static DAGs (Directed Acyclic Graphs). VDS also supports various communication interfaces, such as PVM and MPI, and can therefore run on a wide variety of platforms and hardware architectures. VDS is designed to work with multi-threaded programs, spreading the computational load represented by threads among the computing resources. In contrast, the programming model supported by the ILB library

is single-threaded where application entities communicate via explicit message passing. The ILB library therefore migrates data and pending computational handlers instead of threads.

In [164], the authors describe a parametric model framework for the dynamic load balancing of unstructured applications in a distributed system. By tuning a Profitability Assessment Function (PAF), applications can vary the importance of communication costs and load imbalance in the load balancing decision making. This results in a spectrum of load balancing policies being available in a single load balancing framework. While tuning the PAF function allows the application to vary the level in which runtime imbalance and communication cost factor into load balancing decisions, other factors, such as affinity between data objects, cannot be accounted for as they can with the ILB library. In [125], the authors describe a framework for balancing "heterogeneous" tasks built on the Aroma [124] programming environment. The load balancer implements schedulers for several "homogeneous" task types[4]. Scheduling complex tasks of differing types requires the user to supply a global load balancer to coordinate the activities of the simpler, system implemented schedulers. In this way, simple homogeneous schedulers are composed to create complex load balancers that can schedule a wide variety of tasks. While our load balancing framework allows applications to develop their own scheduling policies for complex task types, this is not required. Applications may make use of a previously created scheduler, and only create a new one if a customized policy is desired. In addition, because the load balancing framework described by Nishikawa et.al. is built upon the Aroma programming environment, porting it to new platforms is a challenging task. This is in contrast to our modular

[4]Homogeneous tasks are tasks that all have the same runtime characteristics, such as number of dependencies.

runtime system design which places an emphasis on portability.

Functionality similar to what we present in the ILB library can be found in runtime systems such as Cilk and Multipol. Cilk [28] targets a more restricted class of computations: *strict* computations. The scheduling policy is fixed, and for a certain class of applications is provably efficient with respect to time, space and communication. In contrast to the Cilk runtime system, Multipol [147] provides more flexibility to the programmer. For example, the programmer is free to use customized schedulers to accommodate application-specific scheduling policies for better performance, and can also specify how much of a thread state needs to be saved. CHAOS++ [45] is a portable, object-oriented runtime library designed to support the construction of dynamic distributed data structures. CHAOS++ defines *globally addressable objects* whose ownership is assigned to one processor. Shadow copies reside on other processors and are used to cache the contents of their remote counterparts, so that accesses are local. The contents of these shadow objects are updated through explicit calls to CHAOS++ routines, leaving the decision concerning when to synchronize objects up to the application. However, in order to tolerate latency associated with dynamic data structures, Multipol and CHAOS++ allow for dynamic caching and replication. In order to avoid this complexity and overhead, and in order to prevent hiding the underlying message passing from the user, the ILB library does not allow for data replication.

Next we review only the directly related application-specific load balancing methods. The DRAMA runtime library [19] is comprised of various tools for the dynamic repartitioning of unstructured finite element applications. The DRAMA cost model, used for calculating the benefit of migrating data during load balancing, is based on hardware specific parameters and measurements taken via application code instrumentation. Another such

library, PLUM [27, 127], is an automatic and portable load balancing environment specifically created to handle unstructured grid applications. PLUM uses the dual of the initial mesh throughout the application for the purposes of calculating processor load, changing the weights associated with each element to reflect the current computational load due to mesh refinement or coarsening. PLUM then makes use of a general purpose partitioner (such as Metis) to balance the computational loads. PLUM also makes use of several metrics to model the cost of moving data during load balancing. *Zoltan* [66, 65, 64] is a dynamic load-balancing library that provides a call-back function interface that makes it easy to use with a wide variety of applications. Zoltan's interface provides graph-based partitioning algorithms through interfaces with ParMETIS [103] or Jostle and several geometric load balancing algorithms.

All of the application-specific *dynamic repartitioning* methods share some common characteristics, namely that processors involved in the parallel computation must synchronize at various stages in order to calculate the new mesh distribution. For large scale parallel machines, this can become a bottleneck in the computation as each processor must exchange information in order to construct a new global mesh. For example, load balancing libraries like DRAMA and PLUM libraries use the global view of the computational graph. Our ILB library does not repartition a global mesh, but instead migrates data objects with pending work only as needed, and potentially with only local workload knowledge, although the exact behavior of the load balancer is up to the implementation of an individual scheduler. In addition, the small interface to the ILB library is easy for application developers (especially those familiar with the MOL) to quickly gain familiarity with. This is in contrast to some of the larger libraries, such as Zoltan, whose APIs can be quite complex.

Another type of load balancer that comes under this category is the *Master/Worker* paradigm [86, 88]. In such a configuration, a centralized Master process[5] feeds work to some number of Workers processes. The Master/Worker approach can be attractive due to the simplicity in its implementation. However Master/Worker methods are generally only appropriate in instances where tasks can be performed independently and asynchronously by a single processor [88], making them inappropriate for mesh generation applications where affinity between regions in a mesh must be taken into consideration for optimal performance. Furthermore, the single Master process can become a bottleneck in instances where many Workers are requesting work at a rapid rate, making this scheme difficult to scale. Schedulers for the ILB library can be implemented with only a local workload view, eliminating the Master process bottleneck. While the ILB library allows implementations of the Master/Worker model, we also provide for more flexibility by allowing the runtime system to take into consideration data dependencies between objects.

A final system that deserves mentioning is the dynamic load balancing mechanism built into Charm++ [100, 98, 99, 101, 102]. We will delve more deeply into Charm++ in Chapter 7, Section 7.1, so we will defer the majority of our discussion until then. However, it is important at this point to distinguish between two types of Charm++ load balancers: *iterative* balancers and *seed-based* balancers. Iterative load balancers assume computation progresses in a series of phases, or in a loosely-synchronous progression. Load balancing is most beneficial if it occurs between phases. In addition, iterative load balancers assume load evolves in a gradual manner, without drastic load changes between phases. Seed-based load balancers operate asynchronously, and balance the creation of tasks. Once tasks are

[5]Either a single Master process or a series of Master processes may be used.

assigned to processors, load balancers that measure the state of the parallel system during runtime are responsible for subsequent task migration. Charm++ is also similar to PREMA, in that the runtime system is actually a framework, which may be extended through the creation or customization of scheduling policies.

While all these projects share some of the same goals as our research, our proposed work differs from these in some crucial respects. In contrast to systems whose object migration is built in at the language level, we have developed user-level libraries written entirely in ANSI C and C++. This means that applications using the ILB/MOL/DMCS library can be written in a widely available language, such as C or C++. We feel application developers will be much more likely to use a system if they are able to use a language they are already familiar with and do not need to rewrite some or all of any existing code. In addition, developers are also able to make use of the latest in compiler technology, which will be more readily available for non-specialized languages.

The ILB/MOL library also differs from many of these systems by not allowing copies of user data objects to exist. This eliminates the need for potentially costly and complex object coherency protocols. Additionally, unlike many DSM systems, the ILB and MOL libraries do not attempt to hide data location from the user by making mobile pointers look identical to local pointers. While such a strategy may ease the programming burden placed on the user, this benefit must be weighed against the difficulty in implementation. For instance, Amber and Emerald migrate stack frames as well as data objects; [97] describes a number of complications involved in moving stack frames, such as deciding which frames to move for a given object and dealing with callee-saves registers. Solutions to these problems are machine and operating system dependent, which makes porting these systems more

difficult.

Finally, we have decided to address fault-tolerance only at the application layer and have ignored issues such as authentication that do not apply to our target platform of MPPs and tightly coupled clusters of workstations. In such an environment, network security is handled by other systems such as Cluster CoNTroller [92]. The Cluster CoNTroller software system, developed by the Cornell Theory Center and sold by MPI Softtech, is made up of secure resource management services and a deterministic heterogeneous scheduling algorithm.

2.4 Parallel Application Modeling Techniques

Research into modeling methods for dynamic load balancing schemes tends to fall into four camps. The first of these requires parallel applications to conform to a well-understood parallel computing model, such as the Bulk Synchronous Processing (BSP) [47] model. The work by Nyland, et al. [126] is one such example, in which the BSP model is used to develop prototypes to estimate the differing costs of spatial decomposition algorithms in the context of Molecular Dynamics codes. However, the irregular applications in which we are interested do not typically conform to these parallel models, making other techniques necessary.

The second category involves simulation [1, 61, 161]. In [161], the authors use a representative workload generator to create test cases for the purposes of comparing the performance of different load balancing techniques. The authors of [1] use a large simulation data set to train a neural network which is, in turn, used to predict the performance of load balancing strategies under various system parameters (time to transfer a task, time to gather load balancing information, and time to exchange load information between nodes).

The third group of research uses queueing theory or Petri nets to perform performance analyses. The work presented in [33] uses queueing theory to develop a model to predict the beneficial impact of load balancing in a network of workstations given the impact of communication latency; if the benefit is above a given threshold, tasks are migrated to distribute the load. In [36], the authors are using a queueing model to predict the relative performance of both static, *a priori* and dynamic load balancing schemes on a class of pipelined computations. While this work is similar in some respects to what we describe here, it does not address the question of how to subdivide a given problem into tasks in order to obtain the maximum performance benefit (this is a primary question we wish to address). Mitzenmacher [120] examines a system in which arriving tasks choose from among some number of servers, based on the queue length at each location. In this work, new tasks arrive over time and the number of tasks is not fixed, distinguishing it from the analysis we undertake here. Esser [70] describes an object-oriented language based on time Petri nets which can be used to evaluate the design of heterogeneous systems made up of a variety of components with different performance characteristics. This language is then used to model and study a complex heterogeneous system made up of software, hardware, and mechanical components.

Such work may address topics in which we are also interested, such as the impact of communication latency on load balancing, comparison between static *a priori* and dynamic load balancing policies, and the assignment of tasks to processing elements. However, the computational requirements necessary for solving the potentially large systems of equations associated with the underlying Markovian processes make this approach less practical for the large-scale parametric studies we wish to undertake. In addition, as one of the future

goals of our research is to develop adaptive application steering through real-time, on-line modeling feedback, this approach is infeasible.

Therefore, as we are more interested in using analytic techniques to predict load balancing performance, we will focus our discussion on the fourth group, analytic modeling techniques. Research, such as the work in [80], is concerned with predicting the degree of imbalance remaining after a given load balancing cycle or iteration. Our load balancing philosophy is different, however; we are concerned more with minimizing idle cycles in the processor pool than maintaining an equal load distribution across nodes. Furthermore, we are interested in modeling the time to completion of all tasks, not the performance of a single load balancing iteration. Work done at Los Alamos National Laboratory [106, 104, 105] has developed analytic models similar to what we describe here for the purposes of predicting and evaluating application performance on newly installed parallel machines or for predicting end-to-end application performance [61]. While we follow similar techniques in developing our model, we tackle the challenging problem of modeling dynamic and adaptive load balancing.

While components of our research are present in the above mentioned work, we are unique in that we provide a mechanism for load balancing methodology analysis that may be performed in the field, allowing developers to calibrate the interaction between the application and the runtime system for maximal benefit. As the parameters in which we are primarily interested will not be constant for all applications and all parallel environments, such analysis is of critical importance.

Chapter 3

Terms and Definitions

Several terms are used in the remainder of this document and should be defined concretely before diving into the specifics of the PREMA design.

Asynchronous

In the most general sense, an *asynchronous* operation is one which will return control to the user program without needing to synchronize with a partner processor. From the user's point of view, asynchronous operations will have a lower latency than their synchronous counterparts, but possibly return before the operation has completed. As an example, an operation that sends a message asynchronously may return before that message has arrived at the target processor. Users that wish to know when an operation has completed may want to use synchronous operations, or may be forced to construct a secondary mechanism for determining whether or not an operation has completed.

Blocking

Blocking operations are asynchronous, but do not return until the data referenced by the operation is safe for modification. This may mean a copy into operating system

memory has taken place, or the data has been placed onto the network. However, blocking operations do not wait for actions to be completed by the target processor. Therefore, from the application's point of view, the latency associated with blocking operations will be less than the latency associated with synchronous operations.

Callback Routine

Callback routines are application defined and must be registered with the runtime system prior to making use of PREMA's load balancing capability. Callback routines are by the runtime system to asynchronously query the application about such matters as the load and priority of Schedulable Objects. They are also used to pack and unpack objects during migration. Schedulable Objects are covered in greater detail elsewhere in this thesis.

Handler

A handler is a user-defined function that is executed during a polling operation. The PREMA system operates in *polling* mode, meaning that arriving messages do not interrupt computation, but are instead queued[1] for execution during an application-posted polling operation. The prototype for handler messages depend on the type of message to which it corresponds, and these are covered in greater detail in the appropriate sections of this document.

Message

Messages have two denotations in this document. In the discussion concerning porting the DMCS runtime library, Message objects are data structures containing information

[1]This queueing may be the responsibility of the underlying communication substrate, or may be built into DMCS itself.

about the communication operation, such as what user handler to invoke on the target and what parameters should be passed to it. When discussing the MOL and load balancer libraries, messages are sent to user-defined data objects called *mobile objects*, and invoke user-specified handlers on receipt. Messages may be forwarded from processor to processor until they reach their target objects.

Message Tags

Message tags are integer values associated with messages that allow the recipient to classify them according to type. In this thesis, message tags are used to distinguish between system-generated load balancing messages and user-generated application messages.

Mobile Object

Mobile Objects are application-defined data objects which are free to migrate throughout the parallel system. Mobile Objects do not have to lie in contiguous memory. When using the MOL, mobile objects migrate under the influence of the application. Using the MOL *message* operation, applications may communicate with mobile objects and invoke user-defined handlers at their location; locating mobile objects is the responsibility of the runtime system, removing a great burden from the application. When using the load balancing library, mobile objects may migrate under the influence of the runtime system.

Mobile Pointer

Mobile Pointers are opaque handles that refer to mobile objects. Mobile pointers are system-wide unique identifiers; they reference target mobile objects for MOL *message*

operations. However, mobile pointers do not serve in the same capacity as a typical machine address. In other words, they cannot be dereferenced in a normal manner, but only through the use of the *mol_mobile_ptr_deref()* routine, which is described in this document.

Node

A node is a single process. The PREMA system treats the process as a single, indivisible unit, and therefore a process corresponds to only a single node. Also, by default, a single process is allocated to each processor (although the PREMA system does not require this), and therefore *node* and *processor* are used interchangeably in this document.

Nonblocking

Nonblocking operations are asynchronous and do not wait for the data to be copied to the network before returning. Therefore, data buffers referenced by nonblocking operations are not guaranteed to be safe when the operation returns, and modifying them may corrupt in-progress communication. The latency observed by the application may be lower for nonblocking operations than for blocking operations. Applications wishing to modify referenced data may choose to opt for blocking operations instead.

Plug-and-play

The plug-and-play load balancing architecture allows one Scheduler module to be easily substituted for another, allowing the user to customize the load balancing behavior to suit the requirements of a particular application. This is achieved by defining a specific interface to which all Schedulers must conform; this interface is designed to

be as concise and general as possible, so that the widest variety of Schedulers possible may be implemented. In addition, this interface isolates the Scheduler from the application, so that exchanging Scheduler modules will have minimal impact on existing application code.

Preemptive

In the context of this thesis, preemptive load balancing refers to the ability of the decision-making component of the load balancing software to interrupt the ongoing execution of an application task in order to process incoming load balancing requests and possible migrate tasks that have not yet begun computation.

Remote Service Request (RSR)

Remote Service Request operations invoke user-defined message handlers on remote processors. RSRs come in three forms: nonblocking, blocking, and synchronous. In addition, RSR operations may take several different types of parameters, which are discussed in more detail in Section B.2. The return type of messages handlers is always *void*, so a return value is not expected. In order to implement return types, the message handler would have to invoke a second communication operation in order to send a result back to the calling processor.

Request

An MOL Request is a communication operation in which a user-defined handler is invoked on a remote target processor, and is passed a user-specified data buffer as a parameter. MOL Requests are not forwarded from processor to processor.

Schedulable Object

A Schedulable Object is any user-defined mobile object that has been registered with PREMA's load balancing software. A Schedulable Object may or may not have pending computation which will act upon it. Once a message handler has arrived for a Schedulable Object, it, along with any parameter data, becomes bundled with the object so they migrate in concert during load balancing.

Scheduler Module

The Scheduler Module encompasses the decision-making and migration capabilities of PREMA's load balancing layer. The architecture of the runtime system is such that different Scheduler Modules may be swapped for one another in a "plug-and-play" architecture, and new Scheduler Modules may be created and used with the system. Some terms that are used synonymously are *Scheduler*, *Scheduler Implementation*, and *Scheduling Policy*.

Source Processor

Each operation deals with a pair of processes. The processor on which the operation originates is called the *source* processor, while the partner processor is referred to as the *target*.

Synchronous

Synchronous operations will not return until the initiating processor has synchronized in some manner with the target. From the user's point of view, a synchronous message passing operation will not return until that message has been received (or at least until the receive operation has begun) on the target node. Synchronous operations typically

incur a higher latency than their asynchronous counterparts, but applications may be guaranteed that data buffers used in the operation are available for reuse once the function has returned.

Target Processor

Each operation deals with a pair of processes. The processor on which the operation originates is called the *source* processor, while the partner processor is referred to as the *target*.

Task

A task is synonymous with a Schedulable Object; it is the unit of migration during load balancing. A task encompasses data, as well as any pending computation that acts on that data.

Work Unit

In this thesis, *work unit* is synonymous with *task*.

Chapter 4

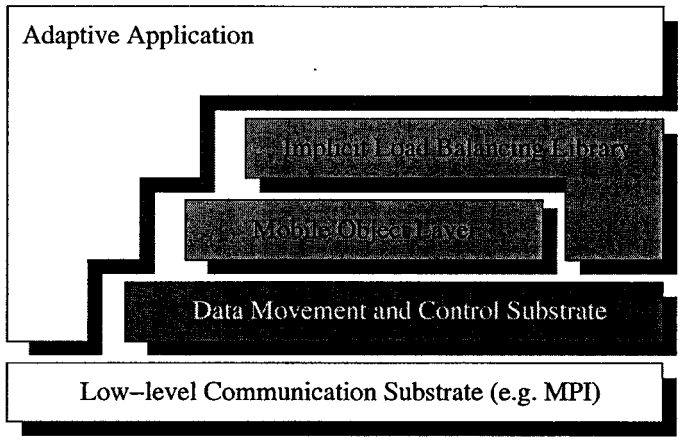
Load Balancing Foundations

The load balancing framework implemented by the PREMA runtime system has as its foundation two lower software layers which we have constructed. The first is the *Data Movement and Control Substrate* (DMCS) and provides basic Active Messages [152, 117] style interprocessor communication primitives. On top of this layer, we have constructed the *Mobile Object Layer* (MOL) which extends the functionality provided by DMCS by adding a global namespace and efficient object migration. In this Chapter, we will describe each of these software components in greater detail and examine how they are used to support load balancing.

4.1 Data Movement and Control Substrate

The *Data Movement and Control Substrate* (DMCS) [16] forms the communication infrastructure for our load balancing framework. DMCS is designed to isolate the upper layers of the runtime system and the application from the idiosyncracies of the underlying hardware, operating system, and low-level message passing infrastructure (Figure 4.1). DMCS is therefore the only software layer that must be ported in order to migrate the PREMA

Figure 4.1: PREMA ARCHITECTURE WITH DMCS HIGHLIGHTED



system from one platform to another.

4.1.1 Description and Applicability

DMCS is designed to serve as *middle-ware* in the communication software hierarchy, meaning that it is constructed using the data movement and message passing functionality provided by a lower-level communication substrate, such as MPI [75]. DMCS is therefore not designed to compete with or replace any low-level communication system, but instead isolates the application from details of the underlying communication substrate. In this way, DMCS has the ability to leverage specialized or high-performance communication features that may be present, such as those found in the LAPI [84, 143] communication package on IBM SP parallel machines or in VIA [43, 151] on clusters of PCs or workstations.

Our desire to create a lightweight communication substrate which would allow developers to implement high-performance and portable adaptive applications lead us to the following goals for the design and implementation of DMCS:

- *High-performance*: In particular, we wanted to provide access to low-latency commu-

nication operations that may be provided in the low-level communication substrate. DMCS is designed to be able to exploit such operations while being optimized to handle the special requirements of adaptive applications.

- *Flexibility and ease-of-use*: We wanted to ensure that DMCS would be useful for parallel adaptive and irregular numerical applications. Therefore, the DMCS API, which contains *Remote Service Requests* and *Remote Memory Manipulation* primitives, provides only those operations which are of the most benefit to adaptive application developers.

- *Portability*: We wanted to make DMCS as portable as possible to a wide variety of low-level communication substrates. DMCS is written entirely in ANSI C/C++ and is designed in a modular fashion on top of the DMCS Messaging Layer (DML), which provides the basic building blocks upon which DMCS is constructed. This reduces the complexity involved in porting DMCS to new platforms by making it necessary to implement only a small number of basic operations.

4.1.2 Operations and Programming Model

DMCS is designed in such a way as to target the requirements of adaptive and asynchronous parallel applications. As such, DMCS provides a single-sided communication interface using an Active Messages programming paradigm which has been tailored to fit the requirements of adaptive numerical computations such as unstructured mesh generation for crack propagation simulation on parallel computers [42]. Such a communication paradigm means that *sending* a message does not require an explicit *receive* operation to be posted on the target

processor. This distinguishes the programming model implemented by DMCS from that which is implemented by communication software supporting a *binary* communication protocol, such as the Message Passing Interface (MPI). Such a design leads to a significant reduction in the code complexity of adaptive applications; unpredictable communication patterns make matching message "sends" with "receives" impractical and unwieldy. The communication model supported by DMCS will yield applications that are developed more quickly and easily, and are easier to maintain in the future.

Communication between processors is provided by *Remote Service Requests*, which are received and processed on remote processors within *polling* operations. Remote Service Requests specify a user-defined *handler* routine to be executed upon processing at the target node. Handler routines may take user-defined parameters of arbitrary size, and may execute any arbitrary computation[1]. In this sense, the message passing mechanism implemented by DMCS can be thought of as being similar to Remote Procedure Invocation, except that DMCS message handlers do not return any values to their caller.

DMCS also provides operations to manipulate remote memory directly, using *put* and *get* operations. As with Remote Service Requests, remote memory manipulation messages require polling operations at the target node for processing; the effect of remote memory manipulation will not be seen by the target until a polling operation has been posted.

All messaging operations are available in *nonblocking*, *blocking*, and *synchronous* varieties, enabling the application developer to effectively overlap computation with communication where possible. Nonblocking operations return to the caller as soon as possible, and possibly before the outgoing parameter data buffer has been copied to the network.

[1]Message handler routines are not, however, allowed to post DMCS polling operations.

Although the latency observable by the user is as low as possible, care must be taken to not modify or delete the outgoing buffer prematurely. Blocking operations will not return until it is safe to modify the parameter buffer, and synchronous operations will not return until the receive operation has begun on the target processor.

In addition, DMCS messages may be associated with user-defined *tags*, that allow for the categorization of messages at the receiving node. Tags are critical to the implementation of PREMA's load balancing framework, as we will see in Chapter 5.

Finally, DMCS provides a single-threaded execution model[2] in which application-defined message handlers execute in the main application thread. This eases the burden placed on application developers, since applications using DMCS do not need to worry about thread safety issues such as controlling access to shared data structures. The single-threaded execution model can also provide significant benefits to application performance due to the fact that user handlers cannot preempt computation which is already in progress[3]. This is desirable because frequent context switching, even between threads, can have a detrimental impact on performance and may lead to an unnecessarily large number of page faults and cache misses.

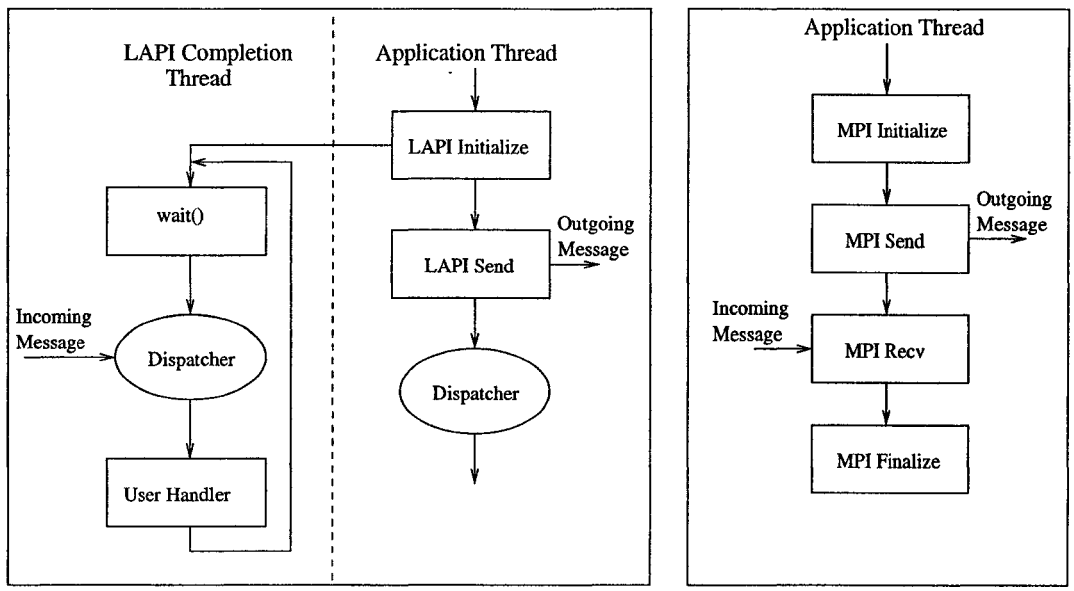
4.1.3 Implementation Summary

We have implemented two versions of DMCS, one built on top of the LAPI [84, 143] communication substrate for the IBM SP family of parallel machines, and another built for clusters

[2] "Single-threaded" in this case means a single communication thread with an arbitrary number of computation threads.

[3] Note that we are making a distinction between the programming model provided by DMCS and the preemptive extension provided by the load balancing framework. We will elaborate on this further in the appropriate chapter.

Figure 4.2: EXECUTION MODELS FOR LAPI AND MPI



LAPI Execution Model MPI Execution Model

of workstations using MPI [75] for communication. Because both implementations support the same API and programming model, several construction details must be resolved. We provide here only a brief discussion of these topics; for a more complete treatment, please refer to [16].

The execution model of DMCS differs significantly from that of LAPI (Figure 4.2). Of critical importance is that the LAPI execution model mandates that user-defined handlers execute inside of a a LAPI *completion thread*, which violates the DMCS single-threaded execution model. In addition, LAPI message handlers execute by default in *interrupt* mode, meaning that user handlers execute as soon as they arrive at the target processor. On the other hand, DMCS handlers must execute only from within polling operations posted by the user.

A second issue is that LAPI, like many other low-level message passing systems, does

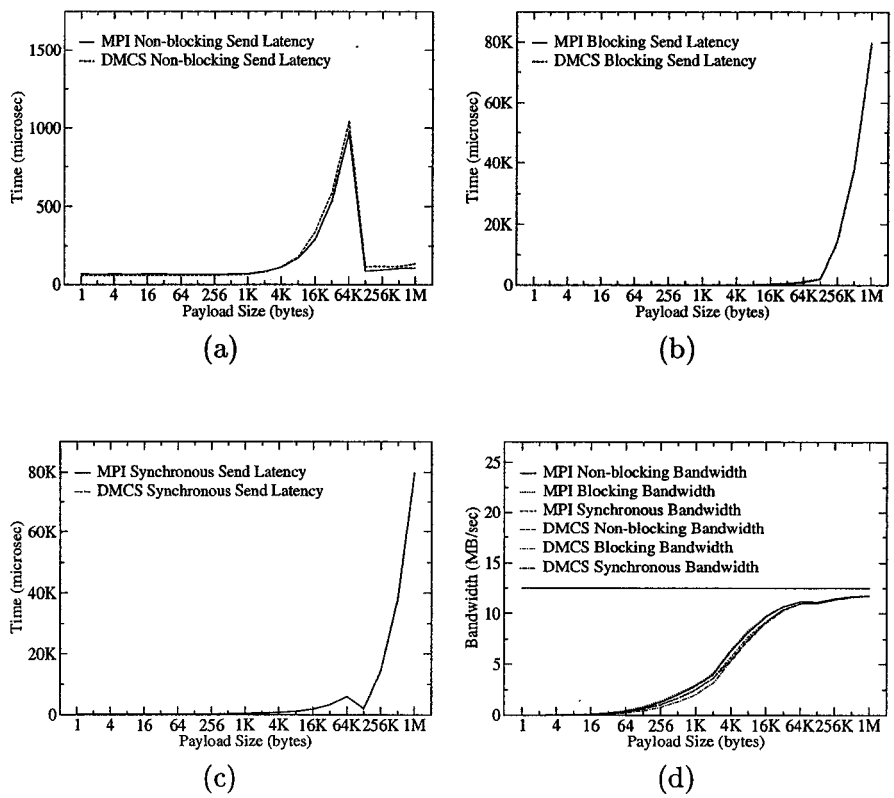
not guarantee message ordering by default. However, message ordering is critical for the correctness of many applications, and therefore must be provided by DMCS[4] through the use of sequence numbers and delay queues; when a message arrives out of order, it is delayed until its turn for execution.

Implementing DMCS using the MPI message passing standard presents its own set of challenges. MPI already provides a single-threaded execution model, and therefore maps well to the single-threaded model mandated by DMCS. Also, DMCS can leave message ordering to MPI; with MPI, messages are guaranteed to arrive in the order in which they were sent provided certain criteria are met. However, MPI specifies a *binary* communication protocol in which explicit *receive* operations must be posted to match with the *send* operation of a remote node. In the context of the adaptive and unstructured nature of the applications which we target, this binary protocol can lead to unnecessarily complex and difficult to maintain application codes.

Additionally, because of the single-threaded execution model provided by MPI and existence of synchronous versions of DMCS message passing operations, the possibility of deadlock must be addressed. DMCS's single-threaded execution model implies that deadlock may occur when two processors send synchronous messages to one another simultaneously. However, through a careful use of timeouts, deadlock can be avoided. Again, the purpose of this discussion is simply to provide some insight into the nature of the challenges that must be overcome in implementing DMCS on various platforms. For a more complete discussion,

[4]Message ordering is an example of functionality that is necessary, but also platform-specific. Some low-level communication software may provide message ordering (such as MPI), and in such cases it should not be provided by DMCS. Therefore, message ordering is an example of functionality that is found in the DML layer of DMCS.

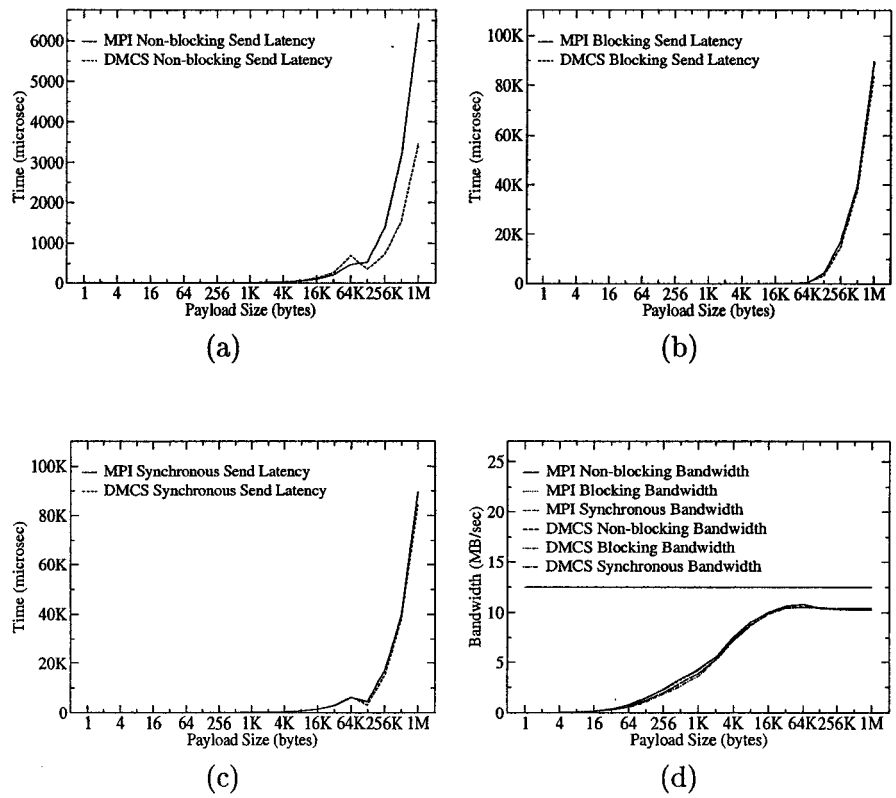
Figure 4.3: DMCS PING-PONG MEASUREMENTS ON SOLARIS CLUSTER



Ping-pong measurements on Solaris cluster using MPI for communication over 100Mb/s fast ethernet: non-blocking latency (a), blocking latency (b), synchronous latency (c), and bandwidth for all three varieties (d).

refer to [16]. It should be noted that the existence of such issues in some sense validates the existence of DMCS itself; by handling these circumstances in the system level, we are freeing the application developer from worrying about them. By writing applications to the consistent programming model provided by DMCS, applications are able to push such portability issues down to the system level, leading to more rapid code development and the creation of more portable software.

Figure 4.4: DMCS PING-PONG MEASUREMENTS ON LINUX CLUSTER



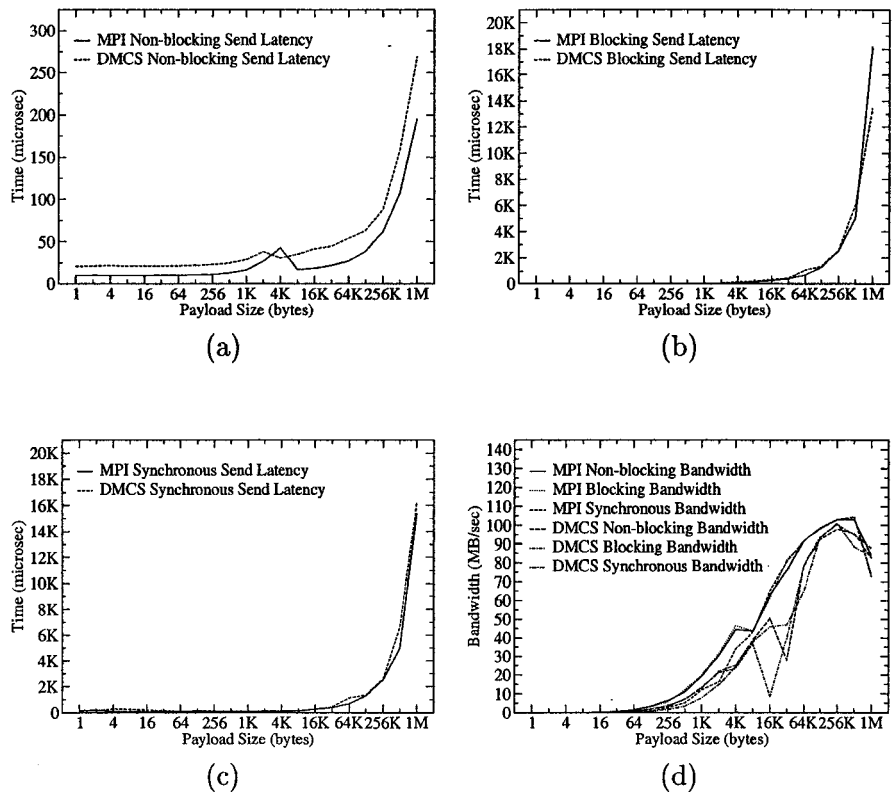
Ping-pong measurements on Linux cluster using MPI for communication over 100Mb/s fast ethernet: non-blocking latency (a), blocking latency (b), synchronous latency (c), and bandwidth for all three varieties (d).

4.1.4 Performance Summary

The performance of the DMCS software can be analyzed both in terms of performance for individual communication operations, and in terms of the aggregate performance for an entire application. Because DMCS is constructed using lower-level communication software, in each case it is necessary to view the performance of DMCS in terms of the overhead added to the lower-level substrate.

Figure 4.3 presents both latency and bandwidth results for a ping-pong micro-benchmark

Figure 4.5: DMCS PING-PONG MEASUREMENTS ON WINDOWS CLUSTER



Ping-pong measurements on Windows cluster using MPI built on top of VIA for communication over 100Mb/s fast ethernet: non-blocking latency (a), blocking latency (b), synchronous latency (c), and bandwidth for all three varieties (d).

running on a cluster of workstations running the Solaris operating system with a 650 MHz UltraSPARC II processor and using the LAM/MPI [111, 75] implementation for communication, and connected using 100 Mb fast ethernet. In these cases, DMCS adds a roughly fixed amount of overhead that is independent of the message size, and varies between 0% and 10% for message sizes ranging from one byte to one megabyte. We see that we obtain similar results for the same experiment running on a cluster of 1 GHz Pentium III workstations operating with the Linux operating system and connected with fast ethernet (Figure 4.4). It should be noted in Figure 4.4(a) that the latency observable with DMCS

is actually *lower* than with MPI alone for large sized messages. While the exact cause is unknown, we feel it can be traced to the message-passing architecture of DMCS. For large sized messages, the message *header* is sent separately from the payload, or *body*. Using two messages, the latency observable to the application is lower than if a single, larger message were sent. It should be noted that this does slightly impact the maximum bandwidth available to the application.

However, when performing the same experiments on a cluster of 1GHz Pentium III workstations running on the Windows operating system and using an implementation of MPI for message passing that is able to take advantage of the VIA interface architecture [43, 151], we see that, in the non-blocking case (Figure 4.5(a)), the constant amount of overhead attributable to the DMCS software is a larger percentage of the overall message passing latency. This is due to the extremely low latency associated with message passing operations due to VIA.

This performance penalty is offset, however, by the portability that DMCS provides to the application developer. Even in the case of applications written using MPI, certain decisions left by the MPI standard to the MPI implementation developer may render applications non-portable from one platform to another. We have experienced this first-hand while porting our software from one implementation of MPI to another. However, with DMCS, this problem is eliminated; DMCS behaves identically across all platforms.

In order to evaluate the effectiveness of DMCS in the context of an adaptive application, we have implemented a 3D Guaranteed Quality Delaunay mesh refinement program[5] (mesh

[5] For more information concerning the 3D guaranteed quality mesh refinement program, please refer to [16] and the Chapter on performance evaluation contained in this thesis.

Table 4.1: DMCS OVERHEAD ON PARALLEL 3D MESH GENERATION

Number of Tets	DMCS Overhead (sec)			MPI Overhead (sec)			Total Time (sec)	Avg. DMCS Overhead (pcnt)
	Min.	Avg.	Max.	Min.	Avg.	Max.		
2M	.0665	.1060	.1312	28.44	30.75	34.02	71	**.1493%**
4M	.0974	.1642	.2066	44.45	49.23	57.23	128	**.1283%**
8M	.1670	.2662	.3768	76.02	82.78	91.59	240	**.1109%**

Parallel 3D mesh generation data gathered for 16 Sun Ultra5 296 MHz processors connected with Fast Ethernet, running the Solaris operating system, and using LAM/MPI for low-level message passing. On average each distributed cavity (approx. 30000 per processor for 8M tets) sends 14 messages.

generation and refinement is a basic building block for the numerical solutions of partial differential equations (PDEs)). One successful approach to generating unstructured meshes of guaranteed quality is *Delaunay triangulation* [144], which refines a given mesh by adding new points on demand using the four-step Bowyer-Watson kernel [34, 154]:

1. *Point creation* creates a new point by using an appropriate spatial distribution technique,

2. *Point location* identifies an element containing this new point,

3. *Cavity computation* removes existing elements that violate the Delaunay property, and

4. *Element creation* builds new triangles or tetrahedra by connecting the new point with old points such that the new cavity satisfies certain geometric properties.

It is this third step that introduces unpredictable computation and communication patterns into the algorithm. The following computation is required: *given a point p and an element e, search among all elements adjacent to e and identify those that violate the Delaunay property.* This search is typically done in a breadth-first order, and approximately 20% to 30% of all searches will access non-local data, requiring communication between

processors. Non-blocking Remote Service Requests are ideally suited for this purpose, as synchronous and blocking communication can seriously degrade performance. In addition, the unpredictable pattern of communication renders binary communication protocols ineffective.

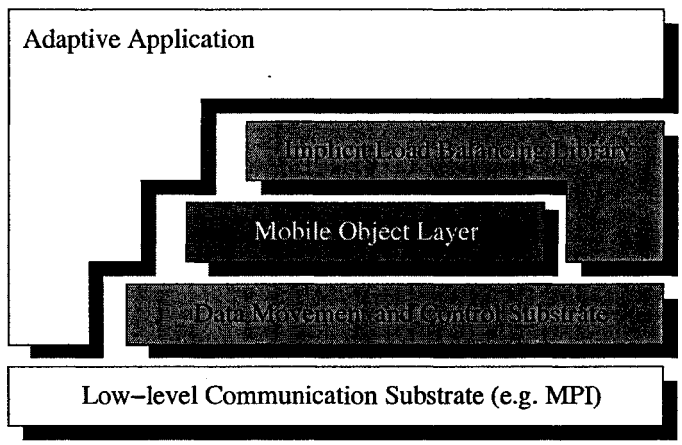
Table 4.1 depicts the performance and DMCS overheads during mesh refinement. In the case of eight million tetrahedra, roughly 30000 cavities per processor (using 16 processors) must access non-local data during the course of the applications runtime. On average, each distributed cavity is responsible for sending 14 messages, leading to roughly 6.72 million messages in total. For these experiments, we use an implementation of DMCS built using MPI for lower-level message passing. The data presented is in terms of the amount of time spent in the MPI message passing layer, as well as the overhead spent within DMCS (excluding MPI message passing). It is therefore possible to determine the overhead attributable to DMCS above and beyond what is incurred by the lower software layers. The results indicate that DMCS overhead accounts for only 0.10% to 0.15% of overall runtime.

We have demonstrated that DMCS provides valuable functionality to adaptive and irregular application developers. In addition, the low overheads incurred by the DMCS software will not significantly impact overall application performance.

4.2 Mobile Object Layer

The *Mobile Object Layer* (MOL) [50] is a runtime substrate we have developed which extends the programming model provided by DMCS by providing a global namespace and automatic

Figure 4.6: PREMA ARCHITECTURE WITH MOL HIGHLIGHTED



message forwarding in the presence of object migration. The load balancing layer of the PREMA runtime system has three primary responsibilities: decision making (determining when and where to migrate data and computation), data migration, and communication in the presence of data migration. The first and second of these fall under the jurisdiction of the higher layers of the runtime system and will be covered in greater detail in later Chapters. The third, communication in the presence of data migration, is the responsibility of the MOL.

4.2.1 Description and Applicability

The MOL is a lightweight software library which facilitates the automatic data movement which is a prerequisite for runtime system provided dynamic load balancing. This is done by providing a global namespace, and a consistent mechanism for interaction with application-defined data objects in the presence of object migration. To this end, the MOL addresses the following three issues:

- *Provide a global namespace*: The MOL provides a global namespace, meaning that application-defined data objects are referenced through handles known as *mobile pointers* which are valid from anywhere in the parallel system. In addition, a mechanism for interacting with distributed data is provided which preserves the communication model implemented by the DMCS (Section 4.1) layer of the PREMA runtime system.

- *Preservation of the communication model*: The MOL is constructed using the DMCS substrate for communication and data migration. While the MOL layer extends DMCS through the addition of a global data namespace, it must preserve the single-sided communication paradigm provided by DMCS. The MOL must also preserve the single-threaded execution model in which application message handlers are executed only from within *polling* operations invoked by the communication thread.

- *Provide an efficient software implementation*: In terms of functionality, the MOL is a significant extension of DMCS; however, the additional runtime overhead must be minimal. Specifically, the location of migratable data objects must be quick and efficient.

The MOL is well-suited to forming the foundation of our load balancing framework. The addition of a global namespace allows the load balancer to migrate data objects (and implicitly, as we will see, computation) according to detected load imbalance without requiring complex and time-consuming routines and data structures for maintaining current mobile object locations. This greatly reduces the complexity and development effort of any higher-level load balancing software.

4.2.2 Operations and Programming Model

The foundations of the PREMA system's programming model have previously been described in Section 4.1.2. The MOL builds upon this framework by adding a global namespace and a mechanism for interacting with distributed data. Below, we will briefly describe these extensions; for a more in depth treatment, refer to [50].

A global namespace is provided by the *mobile object* and *mobile pointer* concepts. Mobile objects are any application-defined data objects, and are not restricted to lie in contiguous memory. Mobile objects have the ability to migrate from processor to processor as a single unit; a single mobile object is not distributed and will always exist in the memory space of a single processor. Mobile pointers are the handles utilized by the application to refer to a mobile object. Mobile pointers are always valid, regardless of the location of the referenced object, and may themselves be passed between processors as messages.

Interaction with mobile objects is done via an extension of the *Remote Service Request* concept described in Section 4.1.2. We call this extension an MOL *message*, which is a communication from a processor to a mobile object, regardless of the location of that target object. In other words, instead of addressing a message to a processor, as is typically done in message passing systems, the message will be addressed to the mobile pointer which refers to the target mobile object. The MOL will be responsible for routing the message to the target object in an efficient manner.

As with communication operations provided by DMCS, *polling* operations must be posted on the target processor in order for a message to be received and processed. MOL messages specify user-defined handler routines to be executed during processing; message

handlers are provided with a pointer to the target mobile object as a parameter, allowing the handler to manipulate the mobile object if necessary.

These operations suggest a programming model which is both message-driven and data-driven. Applications begin by decomposing their data domain into chunks which we will refer to as *sub-domains*. Subdomains are then allocated to the available processors, and are registered with the MOL as mobile objects. The creation of mobile objects will provide the application with the correct mobile pointers, which may be exchanged among processors if necessary. Computation then begins by sending messages to the appropriate mobile objects. The handlers invoked by the messages will encapsulate the computation to be performed on each sub-domain, and may themselves generate new messages. Once the sub-domains are established and the initial computation messages are sent, the application will typically enter a polling loop, which will receive and process any incoming messages until termination is detected, and the program exits.

Although the number of sub-domains is entirely application dependent, sub-domains themselves are typically "coarse-grained". This implies a fairly large computation to communication ratio, which will have an impact on the design of the load balancing layer of the runtime system, which we shall examine later.

4.2.3 Programming Model Example: Distributed Tree

In order to make the programming model just described more concrete, and as a way to demonstrate the effectiveness of the Mobile Object Layer, we present a simple example. Below, we will take a sequential code snippet which performs an operation over the elements contained in a tree data structure, and extend it to demonstrate how the procedure can be

Figure 4.7: SERIAL IMPLEMENTATION OF COMPUTATION OVER A TREE

```
1   struct TreeNode {
2     struct TreeNode* left;
3     struct TreeNode* right;
4     int              data;
5     // ... Any other data members ...
6
7     void do_work(int param) {
8       // ... do some computation here involving 'data' ...
9       if (left != NULL) {
10        left->do_work(data);
11      }
12      if (right != NULL) {
13        right->do_work(data);
14      }
15    }
16  };
17
18
19  // ---------- main() ----------
20  int main(int argc, char* argv[]) {
21    TreeNode* root = create_tree();  // Build the tree
22    root->do_work(0);                // Begin computation at root
23    return 0;
24  }
```

parallelized using the MOL.

Figure 4.7 contains a simple C++ definition of a `TreeNode` data structure, which contains a single `do_work()` method. This method will perform some calculation involving an integer value received from the caller, and then invoke the same method on both the left and right children, if they exist, passing the integer data value as a parameter. In order to parallelize this process, we want to make it possible for the nodes of the tree to be distributed among the available processors.

By converting the class to make use of the MOL, we must replace the local pointers between tree nodes with mobile pointers, and we must replace the direct method invocations with MOL message operations. Figure 4.8 contains the resulting code. In lines 2 and 3, we see the local pointers replaced with mobile pointers (`mol_mobile_ptr_t`). In addition, the

Figure 4.8: MODIFICATION OF TREENODE STRUCTURE TO USE THE MOL

```
1   struct TreeNode {
2     mol_mobile_ptr_t left;
3     mol_mobile_ptr_t right;
4     int              data;
5     // ... Any other data members ...
6
7     void do_work(int param) {
8       // ... do some computation here involving 'data' ...
9       if (!mol_mobile_ptr_is_null(left)) {
10        mol_message(left, do_work_handler, &data, sizeof(int), NULL);
11      }
12      if (!mol_mobile_ptr_is_null(right)) {
13        mol_message(right, do_work_handler, &data, sizeof(int), NULL);
14      }
15    }
16  };
```

previous method invocations have been replaced with MOL messages in lines 10 and 13. As mobile pointers are valid regardless of the location of the referenced data, this method allows nodes in the distributed tree to migrate among the available processors. This ability is useful, for instance, in the case of dynamic load balancing. We will reexamine this example in the context of load balancing in Section 5.3.

Finally, we must create a message handler to be executed when arriving messages are processed. Handlers must exist at a static location, and therefore cannot be typical class methods. The way to handle this restriction is to declare a handler function outside of the class definition which can then invoke a public method of the class. The definition of this handler routine begins in Figure 4.9 on line 1. The parameter list for each MOL message handler is identical, and contains the processor which originated the message (in case a reply is necessary), the mobile pointer and local pointer which refer to the target mobile object, a user-defined parameter data buffer and its size in bytes, and finally a single machine-word sized argument (which is passed NULL in this example).

Figure 4.9: IMPLEMENTATION OF PARALLEL COMPUTATION USING THE MOL

```
1   void do_work_handler(int src, mol_mobile_ptr_t mp, void* obj_data,
2                        void* user_data, int user_size, void* arg)
3   {
4     TreeNode* node = (TreeNode*)obj_data;
5     int* param_data = (int*)user_data;
6     node->do_work(*param_data);
7   }
8
9
10  // ---------- main() ----------
11  int main(int argc, char* argv[]) {
12    mol_msg_handler_t msg_handlers[] = { do_work_handler };
13    mol_init(argc, argv);
14    mol_register_msg_handlers(msg_handlers, 1);
15    mol_barrier();
16
17    // Each processor will build the local part of the tree, and then
18    // combine the pieces to form a single global tree before returning
19    mol_mobile_ptr_t root_mp = create_tree();
20
21    if (mol_my_proc() == 0) {
22      int init_data = 0;
23      mol_message(root_mp, do_work_handler, &init_data, sizeof(int), NULL);
24    }
25
26    // Make sure all work is complete before exiting ...
27    while (!done) { mol_poll(); }
28    return 0;
29  }
```

In the `main()` routine, which begins on line 10 of Figure 4.9, we must initialize the runtime system and register any message handlers that may be invoked by the program. The barrier operation on line 15 guarantees that all processors have finished the initialization procedure before creating the distributed tree data structure.

As an optimization, the function `mol_mobile_ptr_deref()` may be used to check if a mobile object resides on the local processor before sending a message to it (Figure 4.10). This routine will return NULL if the object referenced by the given mobile pointer is not local, and a local pointer to the object if it is. In this way, the local mobile object may be accessed directly, without invoking the MOL's message passing mechanism.

Figure 4.10: OPTIMIZING CODE WITH MOBILE POINTER DEREFERENCING

```
1   if (!mol_is_mobile_ptr_null(left)) {
2     TreeNode* node = (TreeNode*)mol_mobile_ptr_deref(left);
3     if (node != NULL) {
4       node->do_work(data);
5     } else {
6       mol_message(left, do_work_handler, &data, sizeof(int), NULL);
7     }
8   }
```

The final version of the tree code we present makes use of the MOL's object migration functionality (Figure 4.11). It is important to note that the object migration mechanism will later form the foundation of the load balancing layer of the PREMA runtime system. We begin by extending the basic `TreeNode` data structure by adding a mobile pointer to the local data object, and a pointer to a buffer which will be created and filled in during the uninstall process (lines 5 and 6). Note that these data fields are not required, but are often very convenient. We also add a constructor which will take a packed data buffer containing the migrating object's state, and will recreate a `TreeNode` class instance (line 9).

Uninstalling the `TreeNode` object and migrating it to a new processor is encapsulated by the `migrate_node()` routine (lines 13 through 24). Uninstalling the object requires creating a buffer to hold the MOL's *move info* bookkeeping information, and then invoking the `mol_uninstall_obj()` routine which will notify the runtime system that the object is migrating, and will fill in the *move info* buffer. Note that the MOL provides the macro `MOL_MOVE_INFO_SIZE` to indicate the size of the required *move info* buffer, in bytes. The purpose of the *move info* buffer is to pass sequence number information between processors during object migration; sequence numbers are described more fully in Section 4.2.4.1 and 4.2.4.2.

Figure 4.11: OBJECT MIGRATION USING THE MOL

```
1   struct TreeNode {
2     mol_mobile_ptr_t left;
3     mol_mobile_ptr_t right;
4     int              data;
5     unsigned char*   move_info;  // Necessary for object migration
6     mol_mobile_ptr_t self;       // Mobile ptr to myself for convenience
7     // ... Any other data members ...
8
9     TreeNode(void* packed_data) { ... }
10    void do_work() { ... }
11  };
12
13  void migrate_node(TreeNode* node, int proc) {
14    // Uninstall the object
15    node->move_info = new unsigned char[MOL_MOVE_INFO_SIZE];
16    mol_uninstall_obj(node->self, proc, node->move_info);
17    // Pack the tree node into a contiguous buffer and send to new proc
18    int bytes = 0;
19    unsigned char* buffer = pack_tree_node(node, &bytes);
20    dmcs_block_rsrN(proc, migrate_handler, buffer, bytes);
21    // Free the local object and buffer
22    delete [] buffer;
23    delete node;
24  }
25
26  void migrate_handler(int src, void* buffer, int size, void* arg) {
27    // Create the new queue node object from the data buffer and install it
28    TreeNode* node = new TreeNode(buffer);
29    mol_install_obj(node->self, node, node->move_info, src);
30  }
```

Once the mobile object has been uninstalled, it is packed into a contiguous buffer (line 19) using an application-defined packing routine (not shown), and then migrated to the new processor using the *Remove Service Request* functionality provided by the DMCS layer of the runtime system (Chapter 4, Section 4.1.2). However, any mechanism available may be used to transfer the packed object buffer from one processor to another. Once it is safe to do so, the packed object buffer, as well as the object itself, is deallocated.

On the target processor, the `migrate_handler()` message handler (lines 26 through 30) will unpack the data buffer and install the object on the new processor. The *move info* buffer filled in by the uninstall process is passed to the install operation, ensuring future

messages to the mobile object will be routed correctly.

4.2.4 Implementation Summary

The fundamental building blocks provided by the MOL are *mobile objects*, which are any application-defined data objects that have the freedom to migrate from processor to processor, and *mobile pointers*, which are system-wide unique identifiers used to refer to mobile objects. The runtime system places no restrictions on the size of mobile objects, nor does it require that mobile objects exist only in contiguous memory. The mobile pointers used to refer to these mobile objects are valid on any processor in the parallel system and serve as the means for the interaction with the application data necessary to effect the application's computation. Because no "shadow-copies" of mobile objects exist, this interaction typically takes the form of messages passed from ongoing computation to target mobile objects. This indicates that the primary concerns of the MOL's implementation are:

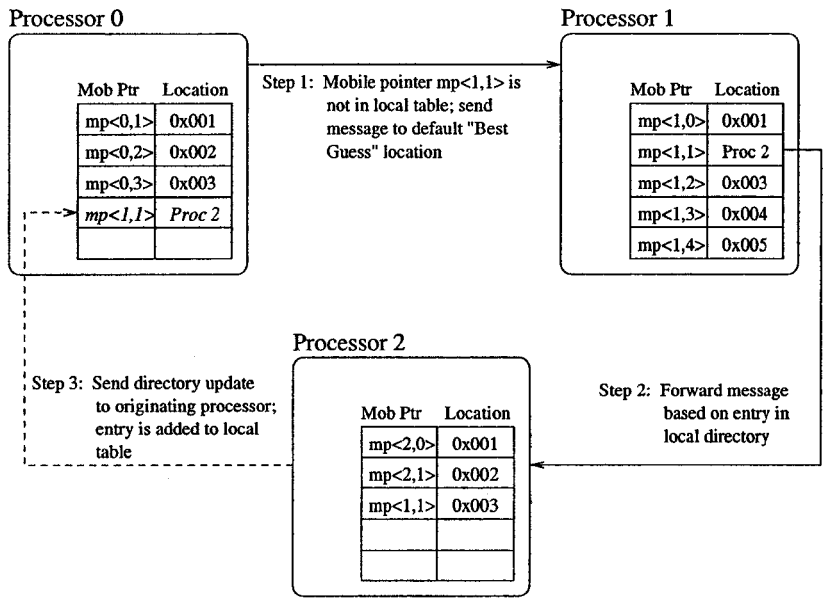
1. efficient object location and migration,
2. the ability to guarantee message ordering between processors and mobile objects, and
3. low overhead message passing.

We will discuss the first two of these considerations here, and defer our evaluation of the efficiency of the MOL's message passing functionality to Section 4.2.5.

4.2.4.1 Mobile Object Location Using Distributed Directories

The efficiency of object location and migration is a direct result of the distributed directory data structure which the MOL uses to locate mobile objects. Under this mechanism, each

Figure 4.12: MOL FORWARDING AND DIRECTORY UPDATE MECHANISM



processor maintains a local directory which contains the "best-guess" locations for the mobile objects that are known to the processor, as well as a sequence number indicating how up-to-date the directory entry is. Such a system minimizes the communication necessary for object creation and migration; only the processor on which a mobile object is created updates its local directory, and migration of an object involves modification of the two processors directly involved in the migration (the source and the target). It should also be noted that mobile objects are not associated with a "home processor" which would be notified of an object's migration, further reducing communication[6].

The mobile object location protocol proceeds as follows (Figure 4.12). Whenever a processor wants to send a message to a mobile object identified by a particular mobile pointer, it first must look that mobile pointer up in its local directory. There are three

[6]The tradeoff in this particular instance is that, although communication is reduced during object migration, message forwards may be necessary in order to locate a mobile object. This is discussed in greater detail below.

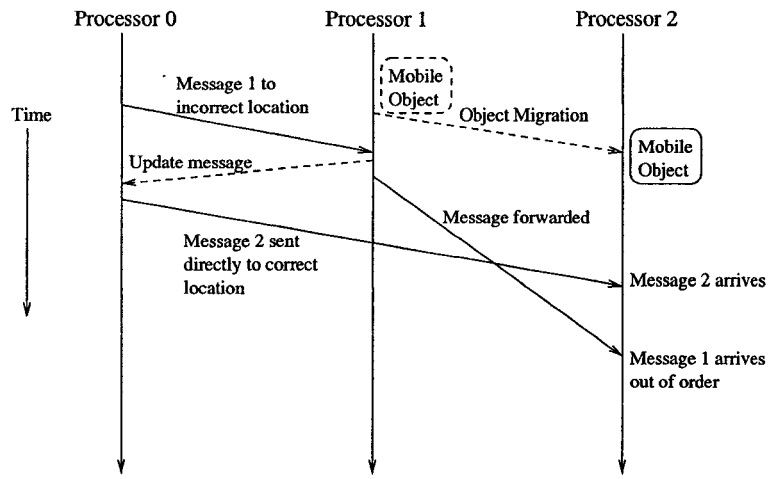
possible results of this lookup. First, the object may reside on the local processor, in which case the message may be handled locally. Second, there may be an entry in the directory indicating that the object resides, or at least used to reside, on a particular remote processor. In this case, the MOL message is sent to the processor indicated by the directory entry. If the mobile object no longer resides at that processor, the message will be subsequently forwarded. The third possible result is that the directory may have no entry for the mobile pointer, in which case the processor on which the mobile object was created serves as the default "best-guess" location[7]. This third case is depicted graphically in Figure 4.12.

In order to minimize the cost of object migration, the MOL *lazily* updates distributed directory entries. When the user migrates a mobile object, only the source and target processors are aware of the change and update their directory entries. Other processors' directories are updated only when they send a message to the object's old location and the message gets forwarded. Again, this is shown in Figure 4.12. In this example, the directory on Processor 0 is updated only once the forwarded message to *mp<1,1>* arrives on Processor 2. Once a processor receives an update for an object, all subsequent messages to the object will go directly to the object's new location. This amortizes the cost of directory updates and message forwards over the number of messages sent to a particular mobile object.

Each mobile object has a movement sequence number associated with it that is incremented each time the object moves. The MOL does not assume any special ordering in the network (such as FIFO or causal ordering), and allows the network to delay or slow down messages for arbitrary lengths of time. It is therefore possible for a processor to receive

[7]Each mobile pointer is a *<processor, sequence number>* pair. The processor on which the mobile object is created is the first element of this pair.

Figure 4.13: OUT OF ORDER MESSAGE ARRIVAL DUE TO OBJECT MIGRATION



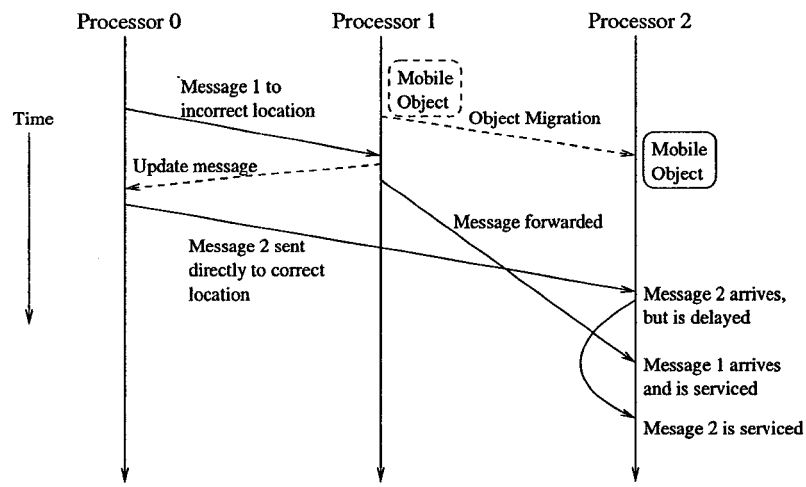
object location updates out of order. To prevent an older update from overwriting a newer one, each update is tagged with the movement sequence number of the object that the update represents. The processor receiving the updates is then able to discard those that are out of date.

4.2.4.2 Message Ordering in the Presence of Object Migration

Although the DMCS layer of the runtime system guarantees message ordering between processors, this is not sufficient for preserving message order for MOL messages. Figure 4.13 demonstrates this. Processor 0 sends Message 1 to a "best-guess" location for a particular mobile object that turns out to be incorrect. Once Processor 0 receives the update message notifying it of the correct location, Message 2 is sent. Because Message 2 is sent directly to Processor 2 (the correct location for the mobile object in this Figure), it is able to pass Message 1 in the network. Consequently, the two messages arrive in the incorrect order.

To correct this problem, each processor maintains two sets of sequence numbers. An *outgoing* sequence number is maintained on each processor for every mobile pointer with

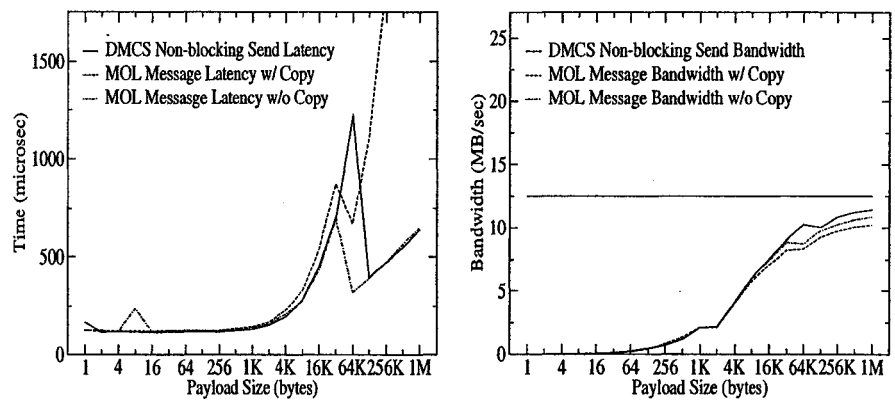
Figure 4.14: MECHANISM TO PRESERVE MESSAGE ORDERING



which that processor communicates. When a processor sends a message to a mobile object, it includes the outgoing sequence number for the associated mobile pointer. This sequence number is then incremented after each outgoing message.

A table of *incoming* sequence numbers contain the next expected sequence number for each <*source processor, mobile pointer*> pair. Arriving messages have the outgoing sequence number contained in the message header checked against the expected incoming sequence number contained in the table. If the two match, then the message may be handled and the expected sequence number contained in the incoming table is incremented. If the arriving message contains a sequence number that has already been seen, then the message may be dropped as it is a repeat. If the sequence number contained in the message is greater than the next expected value, the message must be delayed until the missing messages have arrived and have been serviced. This strategy is shown in Figure 4.14.

Figure 4.15: MOL PING-PONG MEASUREMENTS ON SOLARIS CLUSTER



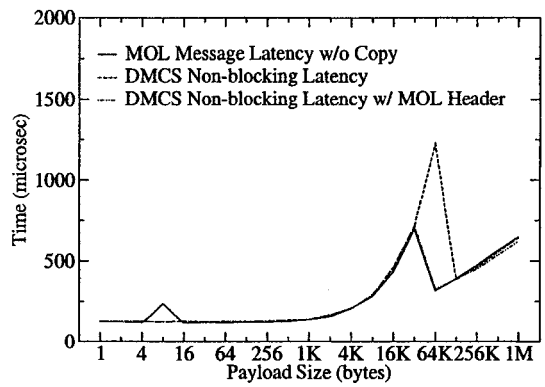
(a) (b)

Ping-pong measurements for MOL messages on Solaris cluster over 100Mb/s fast ethernet: latency measurements (a) and bandwidth measurements (b). DMCS performance is provided for comparison.

4.2.5 Performance Summary

We begin by examining the latency and bandwidth afforded by the MOL's message passing operations relative to the lower levels of the runtime system. Because the MOL is built using the operations provided by the DMCS software, the performance possible using the MOL is bounded by the performance of DMCS. Figure 4.15 (a) contains latency data for the MOL executing on a network of workstations equipped with the Solaris operating system and 650 MHz UltraSPARC II processors. Low-level communication is provided by the LAM [111] implementation of MPI [75], over 100 Mbit ethernet. We have provided two MOL *message* operations for comparison. The decrease in observable latency between 32 Kbyte and 64 Kbyte message sizes is attributable to the LAM/MPI handshaking message passing protocol for large-sized messages. In such a case, only the sending of the *envelope* is begun before the operation returns the caller. As this message is very small, a decrease in latency can be

Figure 4.16: PING-PONG LATENCY MEASUREMENTS FOR DMCS AND MOL



Ping-pong latency measurements for DMCS non-blocking send, DMCS non-blocking send with an extra `MOL_HANDLER_DATA_SIZE` bytes, and MOL send without message copy.

seen.

In the first case, the message passing operation is required to make a copy of the parameter argument buffer, in order to construct a single outgoing buffer with the requisite MOL header information. Beginning at a payload size of roughly 4K bytes, we can see that the memory copy required to build the outgoing message begins to drastically affect the latency associated with the operation. As the message size grows, this time becomes more prevalent.

In the second case, the application provides a buffer to the operation that contains the payload and has `MOL_HANDLER_DATA_SIZE` bytes free at the beginning of the buffer. This allows the runtime system to write the necessary message header information directly into the user-supplied buffer, removing the need to copy the parameter data buffer. With this option, we can see that the latency associated with the MOL more closely mirrors that of DMCS.

However, at a message payload size of 64K bytes, we see that message passing latency is

considerably lower with the MOL than with DMCS. The reason behind this is that the 52 bytes[8] required for the MOL message header information increases the payload size so that LAM's *rendezvous* protocol is used, instead of the *eager* protocol used with smaller messages (the eager protocol is used with the DMCS message of size 64K bytes) [14]. DMCS uses non-blocking `MPI_Isend()` operations to send messages, which are prohibited from blocking by the MPI standard definition. By using the eager protocol, LAM will attempt to send as much of the message as possible before returning, which may include all of the data. In contrast, the rendezvous protocols will only attempt to send the *envelope* of the message, which will incur a much lower latency. It is important to note that neither the MOL nor DMCS are tied to LAM implementation of MPI, or to MPI at all. Since this behavior is an artifact of LAM, it may not be present with other low-level communication substrates.

In order to ensure this behavior was not an artifact of our MOL implementation, we conducted an experiment in which we measure the latency of our DMCS messaging operation, but increased the size of the outgoing payload buffer by `MOL_HANDLER_DATA_SIZE` bytes. The latency observed was nearly identical to that of the MOL messaging operation (Figure 4.16).

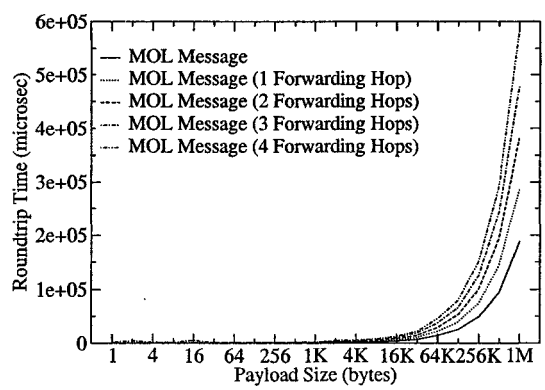
In addition to considering the overall latency and bandwidth of the key MOL operations, it is important to evaluate the costs of certain critical components of the software architecture. Table 4.2 contains the costs for operations such as forwarding messages, looking up mobile objects in the distributed directory data structure, and installing and uninstalling mobile objects. The MOL's implementation is seen to be efficient, particularly in the loca-

[8]The constant `MOL_HANDLER_DATA_SIZE` is equal to 52. This is the number of bytes required for the MOL's message header.

Table 4.2: COSTS ASSOCIATED WITH MESSAGE PASSING

Operation	Time (sec)
Directory lookup	1.8023×10^{-6}
Directory update	5.6740×10^{-6}
System message handler	2.3726×10^{-5}
Message forwarding	1.8725×10^{-4}
Installing mobile object	9.1274×10^{-6}
Uninstalling mobile object	1.1509×10^{-4}

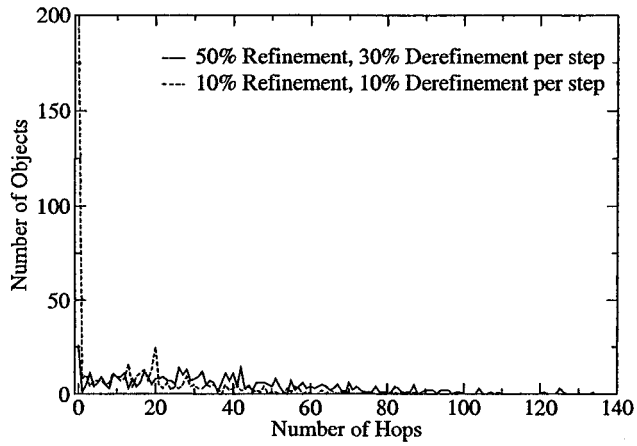
Figure 4.17: ROUND-TRIP TIMES FOR MOL MESSAGES WITH HOPS



tion and migration of mobile objects.

Figure 4.17 contains the round-trip times for MOL messages that are subjected to forwarding hops. The required time scales linearly with the number of forwarding hops that are necessary. This makes sense, as a forwarding hop can be thought of as a re-sending of the original message. Two things should be noted, however. The first is that the maximal number of hops seen by a typical application is quite low. This is due to the fact that, for efficiency reasons, applications tend to create mobile objects in a distributed manner, and objects are primarily migrated due to load balancing. The load balancers we describe in this thesis do not tend to move objects multiple times, which greatly reduces the number of forwarding hops necessary. Second, once a processor is made aware of a mobile

Figure 4.18: MIGRATION HISTOGRAM FOR MULTI-STEP PCDT SIMULATION

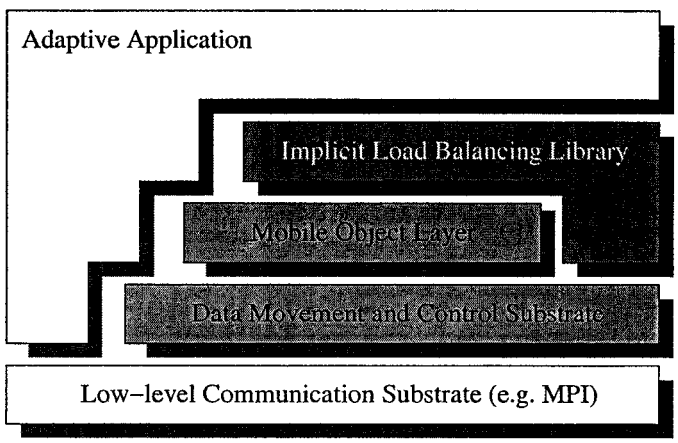


object's new location, subsequent messages can be routed directly. This is in accordance with the *lazy* directory updating scheme we have described. The result is that the vast majority of messages are able to reach their intended targets without being forwarded at all. This observation is reinforced with Figure 4.18, which depicts the number of message hops necessary for two executions of a multiple time-step Parallel Constrained Delaunay Triangulation (PCDT) mesh refinement simulator. The total number of mobile objects in this experiment is 512.

Chapter 5

Load Balancing Framework

Figure 5.1: PREMA ARCHITECTURE WITH ILB HIGHLIGHTED



Developers of parallel applications whose performance suffers from load imbalance have typically been faced with limited choices. On the one hand, they may take advantage of load balancing software systems, but these often involve adapting software to complex programming interfaces or forcing applications to conform to a programming model understood by the load balancer. Making use of such systems is rarely intuitive and may lead to time consuming and complex development efforts. This leaves application developers to exercise their second option, which is to incorporate load balancing algorithms directly into their codes. Although such *explicit* load balancing strategies have the advantage of a close

collaboration between the application and load balancer, quite a large development effort is necessary in order to implement them, and the complexity of application code may be greatly increased. Due to the large amount of work involved in implementing these load balancing algorithms, rarely are the most optimal strategies used. Developers may therefore find themselves rigidly locked into a particular load balancing policy when more efficient methods exist.

In response to these issues, we have developed the Implicit Load Balancing (ILB) framework library. The ILB component of PREMA is built using the tools and programming model provided by DMCS (Section 4.1) and the MOL (Section 4.2), and provides automatic and transparent 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 applications, allowing researchers to experiment with load balancing policies without needing to extensively modify existing codes. This supports our observation that there is not a single load balancing method which is optimal on all platforms for all problems. However, in addition to the framework itself, we have also implemented several of the more common load balancing methods, such as Diffusion [58], Work-Stealing [28], and a variation on Multi-list [158] scheduling.

The architecture of the ILB is designed to fulfill three primary objectives. The first is to provide high performance and low overhead load balancing for the adaptive, irregular, and asynchronous applications in which we are interested in studying. In order for load balancing tools to be effective, these application characteristics must be incorporated into the design of the runtime system.

The 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 using a simple and flexible architecture and interface. Scheduler modules may be easily implemented and exchanged without propagating changes to the application code, allowing for quick experimentation during development.

Finally, 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 message passing mechanism, making use of the ILB's load balancing functionality should involve minimal changes to existing application code. The benefit for the developer is that applications may be implemented quickly without needing to regard load balancing at all; however, if load imbalance becomes a performance issue that must be addressed, the ILB may be easily incorporated. The requirement placed on the runtime system is that the programming model and interface provided by the ILB should closely parallel that which is provided by the MOL.

In the remainder of this Chapter, we will examine more closely the programming model, operations, and architecture of the ILB framework.

5.1 Programming Model

We have already described in Chapter 4 the programming models defined by the lower software layers within the PREMA runtime framework: the Data Movement and Control Substrate (DMCS) and the Mobile Object Layer (MOL). The Implicit Load Balancing (ILB)

layer of the runtime system builds on the tool set supplied by this software foundation. Although some of the discussion here will mirror what has come before, it is important to provide a complete description of the programming model provided by the PREMA runtime system in a single location.

As with the MOL, the basis of the ILB's programming model is a global namespace provided by system-wide unique *mobile pointers* which refer application defined *mobile objects.* Each mobile object is created from a single *sub-domain*, resulting from the partitioning of the application's data domain. As an example, applications such as parallel mesh generation and refinement and N-body codes typically create sub-domains by geometrically partitioning the data domain being manipulated.

As with the MOL, messages are used to invoke computation on mobile objects. Messages bind computation to data, creating the units of work, or tasks, which represent the cumulative load invoked by an application at a given point in time. The load balancing library's messaging mechanism is built upon the functionality provided by the MOL; therefore message routing and object location management are provided by the lower software layers.

At the time of creation, mobile objects are registered with the load balancing layer of the PREMA runtime system in order to create *Schedulable Objects.* Schedulable Objects bind computation to the mobile data on which it operates and serve as the entities that are migrated during dynamic load balancing. Migrating data therefore implicitly migrates computation.

The number of Schedulable Objects created by an application typically is much greater than the number of available processors in the parallel system; this process is known as

over-decomposition and gives the load balancer greater flexibility in its data migration decisions. It is often the case that a greater level of over-decomposition will lead to a more well-balanced final load distribution, and therefore a more optimal runtime, but this is not always the case. Application developers must factor in the cost of performing the decomposition, the (small) overheads that are incurred by the runtime system on a per-task basis, and the increased amount of intertask communication traffic. In this dissertation, we also describe modeling techniques that can be used to evaluate these tradeoffs, aiding developers in making decisions concerning the number of Schedulable Objects to create.

Once the necessary Schedulable Objects have been created and the initial messages have been sent to them in order to invoke computation, *polling* operations must be posted in order to receive and process arriving messages. Message handlers arriving for local tasks will be executed at this time, and Schedulable Objects may be migrated to balance dynamic workload. In addition, new messages may be generated from within message handlers, leading to the creation of dynamic load which may itself require balancing.

Applications may devise their own mechanism to discover termination, or may rely on an implementation of a termination detection algorithm [67] built into the runtime system.

5.2 Operations and Application Interaction

The mechanism by which the application interacts with the Implicit Load Balancing layer of the PREMA runtime system can be broken into two components: the Application Programming Interface (API) and a series of *callback* routines. In general, the API is used by the application to provide information to the runtime system, while callback routines are

Table 5.1: ILB ENVIRONMENT OPERATIONS

System Initialization and Shutdown	
void **ilb_init***(int argc, char* argv[], ilb_configurator_t* config)*	Initialize the ILB system
void **ilb_shutdown***()*	Shuts down the ILB system
System Configuration	
void **ilb_set_configurator_field***(ilb_configurator_t* config, int field, int value)*	Sets a field in the configurator; see below for complete description
Toggle Application Phase	
void **ilb_phase_set***(int phase)*	Toggles load balancing on and off
Querying the Environment	
int **ilb_my_proc***()*	Returns the processor id of the caller
int **ilb_num_procs***()*	Returns the number of processors in the system

used by the runtime system to asynchronously request information from the application. We will delve more deeply into each component in the following subsections.

5.2.1 Application Programming Interface

The Application Programming Interface (API) defines the set of routines which the application uses to interact with the runtime system. The routines described in this Section may be invoked directly by the application. We will give a brief description of each routine, as well as the parameters required and any return values.

Environment Operations

Table 5.1 contains the operations to manipulate and query the ILB library's runtime environment. These routines are responsible for initializing and shutting down the runtime system, and querying the runtime system for information.

ilb_init()

Parameters:

Table 5.2: HANDLER AND CALLBACK ROUTINE REGISTRATION

Message Handler Registration	
void **ilb_register_msg_handlers**(*ilb_msg_handler_t handlers[], int size)*	Register ILB Message handlers with no associated names
void **ilb_register_named_msg_handlers**(*ilb_msg_handler_t handlers[], char* names[] int size)*	Register ILB Message handlers with associated names
Callback Routine Registration	
void **ilb_register_size_functions**(*ilb_size_func_t funcs[], int size)*	Register size calculation callback routines
void **ilb_register_pack_functions**(*ilb_pack_func_t funcs[], int size)*	Register object packing callback routines
void **ilb_register_unpack_functions**(*ilb_unpack_func_t funcs[], int size)*	Register object unpacking callback routines
void **ilb_register_load_functions**(*ilb_load_func_t funcs[], int size)*	Register load calculation callback routines
void **ilb_register_gran_functions**(*ilb_gran_func_t funcs[], int size)*	Register granularity calculation callback routines
void **ilb_register_prio_functions**(*ilb_prio_func_t funcs[], int size)*	Register priority calculation callback routines

1. **int argc**: The number of command line parameters. Passed to *main()* as the first parameter.

2. **char* argv[]**: The command line parameters given to the program. Passed to *main()* as the second parameter.

3. **ilb_configurator_t* config**: <u>*OPTIONAL*</u> This parameter may be left out of the *ilb_init()* call. The configurator object is used to configure the ILB library, and then is passed directly to the MOL during initialization (see Section C.2.1). There are three fields in the configurator object using the *ilb_set_configurator_field()* routine:

   - ILB_CONFIGURATOR_POOL_SIZE : The size of the buffer entries controlled by the ILB's memory manager

   - ILB_CONFIGURATOR_POOL_ENTRIES : The number of buffers man-

Table 5.3: SCHEDULER AND MOBILE OBJECT REGISTRATION

Scheduler Registration	
void **ilb_register_scheduler(** *ilb_scheduler_t* scheduler)*	Register scheduler module with the runtime system
Mobile Object Registration	
void **ilb_register_mobile_obj(** *mol_mobile_ptr_t mptr, ilb_pack_func_t pack_func, ilb_unpack_func_t unpack_func, ilb_size_func_t size_func, vector<size_t> priority_list, size_t granularity_val, size_t load_val)*	Register a mobile object with statically assigned priority, granularity, and load
void **ilb_register_mobile_obj(** *mol_mobile_ptr_t mptr, ilb_pack_func_t pack_func, ilb_unpack_func_t unpack_func, ilb_size_func_t size_func, vector<mol_mobile_ptr_t> dependencies, ilb_prio_func_t priority_func, ilb_gran_func_t granularity_func, ilb_load_func_t load_func)*	Register a mobile object with dynamically calculated priority, granularity, and load

aged by the ILB's memory manager

- ILB_CONFIGURATOR_DIR_UPDATE_PROTOCOL : The value of this field is not used by the ILB library, but is passed transparently to the MOL layer during initialization.

Returns: None

Description:

This function is responsible for initializing the PREMA runtime system and must be the first operation called (except for the *ilb_set_configurator_field()* operation described below). This routine will also initialize the lower layers of the PREMA system (DMCS and MOL), so their initialization routines are no longer called by the application. This operation is collective, meaning that all processors must

call *ilb_init()* at the same time. After the initialization routine returns, other PREMA routines may be invoked.

NOTE: The ILB library contains the notion of application *phases*. Immediately after initialization completes, the application is assumed to be in a startup phase in which no load balancing should occur. In order to move into a load balancing phase in which data migration will take place, the application will need to make use of the *ilb_phase_set()* routine described below.

ilb_shutdown()

Parameters: None

Returns: None

Description:

This operation is the final ILB call made by any application. The ILB is, in turn, responsible for shutting down the lower layers of the PREMA runtime system. This is a collective operation and must be invoked by all processors at the same time. **NOTE:** In the case in which runtime profiling information is gathered (Refer to *Compiling and Installing the PREMA Libraries*), calling *mol_shutdown()* will result in profiling files being generated for the ILB and lower PREMA software layers.

ilb_set_configurator_field()

Parameters:

1. **ilb_configurator_t* configurator**: Pointer to a preallocated configurator object.
2. **int field**: Which field of the configurator object to set. These fields are specified in the description of the *ilb_init()* operation.
3. **int value**: The value to set the configurator field to. Because the ILB's configurator object is analogous to the MOL's configurator, refer to Section C.2.1 for more details.

Returns: None

Description:

Sets a single field of the ILB configurator object. Configurator objects may be passed as an optional third parameter to the *ilb_init()* operation in order to configure components of the ILB library, such as the size of each entry and the number of entries in the preallocated message pool. Not all fields of the configurator need to be set; default values are provided for each of the fields. If no configurator object is passed to *ilb_init()* default values are used that are good in most cases.

ilb_phase_set()

Parameters:

1. **int phase**: This value indicates to the runtime system which phase the application is currently in. The available phases are:
   - *ILB_LOAD_BALANCE_NO*: This indicates that calls to the ILB's polling operations should not result in data migration. This is often useful if

the application has an initial "startup" phase in which message passing is necessary, but load balancing is unwanted. All applications begin in this phase.

- *ILB_LOAD_BALANCE_YES*: This indicates that load balancing is now desired.

Returns: None

Description:

It is often necessary for applications to initially set up data structures and neighbor-relationships among its mobile objects. During this "startup" phase, message passing is often necessary, but load balancing is unwanted. Once this startup phase has completed, the application's computation phase begins, and load balancing is necessary. The location and extent of these phases are application defined and unknown to the runtime system. To accommodate this, the ILB library has the concept of "phases". Initially, all applications are in the startup phase (once *ilb_init()* is invoked). The *ilb_phase_set()* routine is used to toggle between phases. This is a collective operation, and involves a synchronization point.

ilb_my_proc()

Parameters: None

Returns: Integer; caller's processor ID

Description:

Returns a value between 0 and $N-1$ where N is the number of processors in

the parallel system. This number is the processor ID of the calling processor. Although the exact numbering of processors depends on the lowest-level communication substrate (such as MPI), it is generally assumed that the processor IDs begin at zero and proceed sequentially.

ilb_num_procs()

Parameters: None

Returns: Integer; number of processors in parallel system

Description:

Returns the number of processors in the parallel system, N.

Registering Scheduler Module With the ILB

Preparing a scheduling module for use with PREMA's load balancing functionality is a two-stage process. First, the scheduler must be instantiated using the constructor for the specific scheduler class. The second phase is to register a pointer to the new scheduling policy object with the runtime system. The following routine is provided for that purpose.

ilb_register_scheduler()

Parameters:

1. **ilb_scheduler_t* scheduler**: Pointer to an instance of the scheduling policy class. This class may be a user-defined class, but all scheduler types are derived from the *ilb_scheduler_t* type.

Returns: None

Description:

This routine is used to register an instance of the scheduling policy class with the PREMA runtime system. This must be done on each processor before any load balancing can take place.

Registering Message Handlers With the ILB

Although the message handlers for ILB messages are of the same type as message handlers for MOL messages, they must be registered separately. However, it is perfectly acceptable for the same handler to be used for both ILB and MOL messages, if the application so desires. All ILB handlers must be registered on each processor in the same order. Typically, handler registration takes place immediately after ILB initialization.

ilb_register_msg_handlers()

Parameters:

1. **ilb_msg_handler_t handlers**[]: An array containing pointers to the user-defined ILB message handlers to be registered.
2. **int size**: The length of the handlers array.

Returns: None

Description:

Applications make use of this routine to register ILB message handlers with the runtime system. All message handlers must be registered before they may be invoked via message passing.

ilb_register_named_msg_handlers()

Parameters:

1. **ilb_msg_handler_t handlers**[]: An array containing pointers to the user-defined ILB message handlers to be registered.
2. **char* names**[]: An array containing names for the handlers; this array must be of the same length as the handlers array.
3. **int size**: The length of the handlers array.

Returns: None

Description:

This routine will register user-defined ILB message handlers with the runtime system. Furthermore, a name will be associated with each handler. This is helpful at times for debugging, but this routine is not often used.

Registering Callback Routines With the ILB

As with application-defined message handlers, callback routines must be registered with the ILB system prior to use. The following routines are used for this purpose.

ilb_register_size_functions()

Parameters:

1. **ilb_size_func_t funcs**[]: An array containing pointers to the size calculation callback routines to be registered.
2. **int count**: The number of entries in the function array.

Returns: None

Description:

Applications make use of this routine to register size calculation callback routines with the runtime system. This must be done prior to use.

ilb_register_pack_functions()

Parameters:

1. **ilb_pack_func_t funcs**[]: An array containing pointers to the packing callback routines to be registered.
2. **int count**: The number of entries in the function array.

Returns: None

Description:

Applications make use of this routine to register object packing callback routines with the runtime system. This must be done prior to use.

ilb_regster_unpack_functions()

Parameters:

1. **ilb_unpack_func_t funcs**[]: An array containing pointers to the unpacking callback routines to be registered.
2. **int count**: The number of entries in the function array.

Returns: None

Description:

Applications make use of this routine to register object unpacking callback routines with the runtime system. This must be done prior to use.

ilb_register_load_functions()

Parameters:

1. **ilb_load_func_t funcs[]**: An array containing pointers to the load calculation callback routines to be registered.
2. **int count:** The number of entries in the function array.

Returns: None

Description:

Applications make use of this routine to register load calculation callback routines with the runtime system. This must be done prior to use.

ilb_register_gran_functions()

Parameters:

1. **ilb_gran_func_t funcs[]**: An array containing pointers to the granularity calculation callback routines to be registered.
2. **int count:** The number of entries in the function array.

Returns: None

Description:

Applications make use of this routine to register granularity calculation callback routines with the runtime system. This must be done prior to use.

ilb_register_prio_functions()

Parameters:

1. **ilb_prio_func_t funcs[]**: An array containing pointers to the priority calculation callback routines to be registered.
2. **int count:** The number of entries in the function array.

Returns: None

Description:

Applications make use of this routine to register priority calculation callback routines with the runtime system. This must be done prior to use.

Registering Mobile Pointers With the ILB

As with the Mobile Object Layer, the ILB expresses parallelism in terms of *mobile objects.* Creating a mobile object and registering it with the runtime system for the purposes of load balancing is a two-step process. In the first step, a mobile object is created using the *mol_create_mobile_ptr()* routine described in Section C.2. The return value from this operation is a *mobile pointer*, which is a system-wide unique identifier used to reference the mobile object. This mobile pointer is then registered with the ILB layer of the runtime system, making it available for load balancing, using the following operation. This routine is overloaded; each variant is described below.

ilb_register_mobile_obj()

Parameters:

1. **mol_mobile_ptr_t mp**: Mobile pointer returned by a call to *mol_create_mobile_ptr()*. This must refer to a local data object.

2. **ilb_pack_func_t pack_func**: Pointer to a user-defined function responsible for packing a mobile object into a contiguous buffer for transport. This routine must be previously registered with the ILB.

3. **ilb_unpack_func_t unpack_func**: Pointer to a user-defined function responsible for unpacking and reconstructing the mobile object. This routine must be previously registered with the ILB.

4. **ilb_size_func_t size_func**: Pointer to a user-defined function responsible for calculating the size (in bytes) of a mobile object. This routine must be previously registered with the ILB.

5. **vector<size_t> priority**: This is a priority vector, with an entry for each processor in the parallel system. The higher the value associated with a particular processor, the stronger the "affinity" the mobile object has for residing on that particular processor. When load balancing occurs, objects have a greater probability of migrating to processors with a greater priority value.

6. **size_t granularity**: The granularity value is an index describing the difficulty associated with migration; the higher the value, the greater the difficulty.

7. **size_t load**: This is an index representing the load associated with a particular mobile object; the higher the value, the greater the associated load.

Returns: None

Description:

Table 5.4: ILB COMMUNICATION AND SYNCHRONIZATION

void **ilb_message***(mol_mobile_ptr_t mptr, ilb_msg_handler_t handler, void* user_data int user_size, void* arg)*	Sends a message to a specific mobile object
void **ilb_poll***()*	Receives and processes pending PREMA messages
void **ilb_barrier***()*	Blocks execution until all processors enter the barrier

Application-defined mobile objects must be registered with the Load Balancing system software. This informs the runtime system that the load associated with this particular mobile object should be considered during load balancing. The mobile pointer in question must also be local to the calling processor. In addition to the mobile pointer, user-defined callback routines are provided to calculate runtime information.

ILB Message Passing Operations

Computation using the PREMA runtime system progresses in a message-driven manner. This means that computation results from the receipt and processing of application-defined messages. As with the MOL layer discussed previously, messages may be directed to any application-defined mobile object; the runtime system is responsible for routing messages. Messages may contain an arbitrary parameter data buffer, and will invoke a user-defined *handler* routine. In order for an arriving message to be processed and the specified handler executed, a *polling* operation must be posted by the application. Polling operations are described in greater detail later.

ilb_message()

Parameters:

1. **mol_mobile_ptr_t mp**: This is the mobile pointer that refers to the object that is to be the target of the message.
2. **ilb_msg_handler_t handler**: This is the user-defined handler that is to be invoked in response to this message. Handlers must be registered with the runtime system prior to invocation.
3. **void* user_data**: This is a pointer to the user-defined parameter buffer. May be NULL.
4. **int user_size**: This is the size of the user-defined parameter buffer, in bytes. Zero for a NULL buffer.
5. **void* arg**: This is a single machine-word sized argument, and will be passed to the user handler. Note that this is not a pointer to an argument. May be NULL.

Returns: None

Description:

This routine is used to send a message to a specific mobile object. The runtime system will route the message to the correct destination in the presence of dynamic object migration. Note that the data buffer is available for reuse as soon as this operation returns.

ILB Polling Operations

The PREMA runtime system provides a single-threaded programming model, meaning that newly arrived messages will not preempt executing computation. There must therefore be some mechanism the application may use to explicitly check the network for pending

messages. This mechanism is the polling operation.

ilb_poll()

Parameters: None

Returns: None

Description:

The polling operation polls the network for incoming messages and executes any user handlers referenced by the messages. In addition, the ILB polling operation includes an *mol_poll()* call, so MOL messages and DMCS messages will be handled within an ILB poll. Because the ILB presents a single-threaded model, the poll call should be made by the main application thread only.

ILB Synchronization Operations

Like the MOL, the ILB provides a simple synchronization operation which can be used to synchronize all processors. The algorithm is a simple fan-in, fan-out algorithm which ensures that all processors enter the barrier before any are allowed to leave. What distinguishes the ILB barrier variant from the barrier operations found in the MOL and DMCS is that the barrier is implemented using an internal *ilb_poll()*. This means that load balancing and data migration may result due to a barrier.

ilb_barrier()

Parameters: None

Returns: None

Description:

Blocks execution until all processors have entered the barrier. Pending PREMA message handlers may be executed within a barrier, and Schedulable Objects may be migrated due to perceived load imbalance.

5.2.2 Callback Routines

In order to gather runtime information necessary for effective load balancing, the ILB library must have a mechanism for asynchronously querying the application. For this purpose, the user must supply six types of *callback* routines, which are described below.

Object Size Calculation Routine

This routine takes a mobile pointer (Section 4.2) as a parameter, and should return an unsigned integer value (the size of the referenced mobile object, in bytes). This information is necessary for packing the user's data object prior to migration during load balancing.

Type Name: **ilb_size_func_t**

Parameters:

1. **mol_mobile_ptr_t mp**: This is the mobile pointer which refers to the user-defined mobile object in question. It is guaranteed that this object will be local, and so can be dereferenced with the *mol_mobile_ptr_deref()* routine.

Returns: size_t; size of the data object in bytes

Object Packing Routine

This routine packs a user-defined data object into a contiguous buffer prior to migration. The buffer used for packing in preallocated and managed by the system. At the end of

the packing routine, the user is responsible for deallocating any memory occupied by the mobile data object and ensuring that application data structures are maintained and remain consistent.

Type Name: **ilb_pack_func_t**

Parameters:

1. **void* object**: This is a pointer to the application-defined mobile object.
2. **void* buffer**: This is a pointer to the system-managed buffer into which the object should be packed.
3. **int tgt**: This is the processor to which the object will be migrating, in case this information is necessary to update the application's data structures.

Returns: size_t; the number of bytes packed into the buffer. This figure should match that which would have been returned from the Size Calculation Routine.

Object Unpacking Routine

This routine performs the inverse of the packing routine, reconstructing an application data object from a packed buffer. The routine is responsible for allocating any memory necessary for the object. Note that the object must be removed from the buffer; at the end of this routine, the contents of the system-managed buffer cannot be guaranteed.

Type Name: **ilb_unpack_func_t**

Parameters:

1. **void* buffer**: This is the buffer which contains the packed object.
2. **size_t size**: This is the number of bytes contained within the buffer.

Returns: void*; a pointer to the newly reconstructed data object.

Load Calculation Routine

This routine is used to calculate the pending load represented by a data object. The data object will have one or more newly arrived message handlers awaiting execution; this routine is given one of these handlers and returns the load represented by this handler for this particular data object. The runtime system is then able to determine the cumulative load pending for this object. The handler is able to compare the handler pointer passed as a parameter with the addresses of the possible handlers in order to determine which this is.

Type Name: **ilb_load_func_t**

Parameters:

1. **mol_mobile_ptr_t mp:** This is the mobile pointer which references the application data object. It is guaranteed that this data object will be local, so the mobile pointer may be dereferenced with the *mol_mobile_ptr_deref()* routine.
2. **ilb_msg_handler_t handler:** This is a handler that is currently pending for this data object. This routine will calculate the load represented by this pending handler; the application can compare this handler pointer to the addresses of possible handlers in order to determine which it is.

Returns: size_t; the index representing the pending load. Larger indices indicate a greater pending load.

Granularity Calculation Routine

This routine is used to calculate the "granularity" of a mobile object, which is an

index describing the difficulty associated with migrating that object to a remote processor. Objects that are more difficult to migrate, due to possibly the size of the object or the effort required in updating application data structures which contain that object, are associated with a larger granularity index.

Type Name: **ilb_gran_func_t**

Parameters:

1. **mol_mobile_ptr_t mp**: This is the mobile pointer which references the application data object. It is guaranteed that this data object will be local, so the mobile pointer may be dereferenced with the *mol_mobile_ptr_deref()* routine.

Returns: size_t; the index representing the granularity. Larger indices indicate greater difficulty in migrated the associated object.

Priority Calculation Routine

This routine is used to calculate a priority vector, with an entry for each processor in the parallel system. Higher values indicate a greater priority, and thus a greater "affinity" for that data object to reside on the associated processor. Applications can use the priority vector in conjunction with the scheduling algorithm to influence the migration and scheduling sequence.

Type Name: **ilb_prio_func_t**

Parameters:

1. **size_t granularity**: This is the value returned by the granularity calculation routine.
2. **size_t load**: This is the value returned by the load calculation routine.
3. **vector<size_t> dependencies**: This is a vector with an entry for each processor in the parallel system. The value of each entry indicates the number of "dependencies" located on the associated processor. Dependencies are mobile objects declared to have an affinity with the current object (affinities are declared when the mobile objects are registered with the ILB). Note that the dependencies vector will be impacted by the directory update protocol used in the MOL layer of the runtime system; the vector is filled in with the "best guess" for the locations of the neighboring mobile objects.

Returns: vector<size_t>; this is a vector with an entry for each processor. The value in each entry is the priority of the mobile object with respect to the associated processor; higher values indicate higher priorities.

5.3 Programming Model Example: Distributed Tree

One of our stated goals for the design and implementation of the Implicit Load Balancing library was to provide an *evolutionary* migration path for code created using the Mobile Object Layer (Section 4.2). Following this path, code developed without the capability for dynamic load balancing can quickly incorporate load balancing functionality without a severe impact to existing code. The best way to illustrate this is to revisit the example developed in Section 4.2.3. We will then extend this code to incorporate ILB functionality.

We begin by reviewing the code sample given in Figures 4.8 and 4.9, and repeated here in Figures 5.2 and 5.3. In this example, the mechanism is in place to distribute the tree data structure among the available processors. However, application code must intervene in order to reassign nodes. If the computation performed by the `do_work()` method is unpredictable, differing numbers of tree nodes are initially assigned to each processor, or systemic variances intervene, workload imbalance may result, which must be detected and corrected by the application itself.

What is needed is runtime system software that will automatically detect and correct this dynamic load imbalance. This is the job of the ILB. Figure 5.4 contains the modified `TreeNode` data structure and `do_work_handler()` message handler. Comparing these with Figure 5.2 and Figure 5.3, we can see that very little has changed. The only difference comes on lines 10 and 13, in which the `mol_message()` routine has been replaced by a call to `ilb_message()`. This signifies to the runtime system that the computation invoked as a result of the message is subject to load balancing. In other words, the target mobile object and pending computation (message handlers) may be migrated to an underloaded processor prior to handler execution.

In order to minimize the impact of incorporating dynamic load balancing, message handlers carry over unchanged from Figure 5.3. In addition, the parameter order and values of the message invocation operations are identical. However, the code contained in Figure 5.5 must be added. There are six routines, summarized as follows (all of these *callback* routine types are described in greater detail in Section 5.2.2):

- *Object packing routine (line 1):* This routine is used to prepare a `TreeNode` object for migration during load balancing. It must pack the object into a contiguous buffer.

Figure 5.2: TREENODE STRUCTURE WITH MOBILE POINTERS

```
1   struct TreeNode {
2     mol_mobile_ptr_t left;
3     mol_mobile_ptr_t right;
4     int              data;
5     // ... Any other data members ...
6
7     void do_work(int param) {
8       // ... do some computation here involving 'data' ...
9       if (!mol_mobile_ptr_is_null(left)) {
10        mol_message(left, do_work_handler, &data, sizeof(int), NULL);
11      }
12      if (!mol_mobile_ptr_is_null(right)) {
13        mol_message(right, do_work_handler, &data, sizeof(int), NULL);
14      }
15    }
16  };
```

Because the *TreeNode* type exists in contiguous memory, this amounts to a simple memory copy.

- *Object unpacking routine (line 8):* This routine unpacks an object after transport, and modifies any necessary local application data structures to insert this new object into the global computation.

- *Object size calculation routine (line 14):* This routine returns the size of the reference object, in bytes.

- *Priority calculation routine (line 18):* This routine returns a vector with a priority value for each processor in the parallel system. In this example, we mandate that Schedulable Objects have no preference as to the processor on which they execute; therefore the entries in the priority vector are equivalent.

- *Load calculation routine (line 23):* This routine returns an integer index describing the current load imposed on the mobile object by a particular message handler. In this

Figure 5.3: PARALLEL COMPUTATION USING MOBILE OBJECT LAYER

```
1   void do_work_handler(int src, mol_mobile_ptr_t mp, void* obj_data,
2                        void* user_data, int user_size, void* arg)
3   {
4     TreeNode* node = (TreeNode*)obj_data;
5     int* param_data = (int*)user_data;
6     node->do_work(*param_data);
7   }
8
9
10  // ---------- main() ----------
11  int main(int argc, char* argv[]) {
12    mol_msg_handler_t msg_handlers[] = { do_work_handler };
13    mol_init(argc, argv);
14    mol_register_msg_handlers(msg_handlers, 1);
15    mol_barrier();
16
17    // Each processor will build the local part of the tree, and then
18    // combine the pieces to form a single global tree before returning
19    mol_mobile_ptr_t root_mp = create_tree();
20
21    if (mol_my_proc() == 0) {
22      int init_data = 0;
23      mol_message(root_mp, do_work_handler, &init_data, sizeof(int), NULL);
24    }
25
26    // Make sure all work is complete before exiting ...
27    while (!done) { mol_poll(); }
28    return 0;
29  }
```

example, we stipulate that each handler imposes an equal load on the Schedulable Object. More complex applications may wish to distinguish between handlers; the pointer to the handler function can be used for this purpose.

- *Granularity calculation routine (line 27):* This routine returns an integer index describing the difficulty associated with migrating the mobile object. We dictate that each Schedulable Object is equally difficult to migrate.

Finally, Figure 5.6 contains the `main()` routine for the load balanced example. It progresses in essentially the same pattern as the `main()` routine in Figure 5.3. Lines 2–8 create the arrays of message handlers and callback routines used by the example. Line 10 replaces

Figure 5.4: TREENODE STRUCTURE AND ILB MESSAGE HANDLER

```
1   struct TreeNode {
2     mol_mobile_ptr_t left;
3     mol_mobile_ptr_t right;
4     int              data;
5     // ... Any other data members ...
6
7     void do_work(int param) {
8       // ... do some computation here involving 'data' ...
9       if (!mol_mobile_ptr_is_null(left)) {
10        ilb_message(left, do_work_handler, &data, sizeof(int), NULL);
11      }
12      if (!mol_mobile_ptr_is_null(right)) {
13        ilb_message(right, do_work_handler, &data, sizeof(int), NULL);
14      }
15    }
16  };
17
18
19  void do_work_handler(int src, mol_mobile_ptr_t mp, void* obj_data,
20                       void* user_data, int user_size, void* arg)
21  {
22    TreeNode* node = (TreeNode*)obj_data;
23    int* param_data = (int*)user_data;
24    node->do_work(*param_data);
25  }
```

the previous call to `mol_init()` with a call to `ilb_init()`, while lines 11–17 register the message handlers and callback routines.

Lines 19 and 20 create the *Scheduler* module and register the Scheduler with the runtime system. Note that the Scheduler may be of a predefined or user-defined type. For more information on the Scheduler interface and predefined Scheduler types, refer to Section 5.4.2. Once the Scheduler is registered, initialization is complete.

Line 23 invokes a routine that creates the global tree data structure. We omit this routine in the code samples for the sake of brevity. Tree creation can proceed in the same manner as before, with the exception that each mobile object must be registered with the runtime system using the `ilb_register_mobile_ptr()` routine. This registration signifies to the runtime system that this mobile object is a candidate for migration during load

Figure 5.5: CODE FOR OBJECT TRANSPORT AND PRIORITY CALCULATION

```
1   size_t pack_func(void* object, void* buffer, int tgt) {
3     memcpy(buffer, object, sizeof(TreeNode));
4     delete (TreeNode)object;
5     return sizeof(TreeNode);
6   }
7
8   void* unpack_func(void* buffer, size_t size) {
9     TreeNode* t = new TreeNode;
10    memcpy(t, buffer, size);
11    return (void*)t;
12  }
13
14  size_t size_func(void* object) {
15    return sizeof(TreeNode);
16  }
17
18  vector<size_t> prio_func(size_t gran_val, size_t load_val,
19                           vector<size_t> dependencies) {
20    return vector<size_t>(mol_num_procs(), 1);
21  }
22
23  size_t load_func(mol_mobile_ptr_t mp, ilb_msg_handler_t handler) {
24    return 1;
25  }
26
27  size_t gran_func(mol_mobile_ptr_t mp) {
28    return 1;
29  }
```

balancing.

The rest of the `main()` routine is unchanged, save for replacing `mol_message()` with `ilb_message()` on line 27, and replacing `mol_poll()` with `ilb_poll()` on line 31.

5.4 Load Balancing Framework Architecture

The architecture and implementation of the ILB load balancing framework arose from a need to provide efficient dynamic load balancing support for asynchronous and adaptive parallel applications. Adaptivity in the application level imposes several consequences on the runtime system. The first is that computational complexity will vary unpredictably

Figure 5.6: HANDLER REGISTRATION AND SYSTEM INITIALIZATION

```
1   int main(int argc, char* argv[]) {
2     ilb_msg_handler_t msg_handlers[] = { do_work_handler };
3     ilb_pack_func_t pack_funcs[] = { pack_func };
4     ilb_unpack_func_t unpack_funcs[] = { unpack_func };
5     ilb_size_func_t size_funcs[] = { size_func };
6     ilb_prio_func_t prio_funcs[] = { prio_func };
7     ilb_load_func_t load_funcs[] = { load_func };
8     ilb_gran_func_t gran_funcs[] = { gran_func };
9
10    ilb_init(argc, argv);
11    ilb_register_msg_handlers(msg_handlers, 1);
12    ilb_register_pack_functions(pack_funcs, 1);
13    ilb_register_unpack_functions(unpack_funcs, 1);
14    ilb_register_size_functions(size_funcs, 1);
15    ilb_register_prio_functions(prio_funcs, 1);
16    ilb_register_load_functions(load_funcs, 1);
17    ilb_register_gran_functions(gran_funcs, 1);
18
19    MySchedulerType* sched = new MySchedulerType();
20    ilb_register_scheduler(sched);
21
22    // Each processor will build the local part of the tree ...
23    mol_mobile_ptr_t root_mp = create_tree();
24
25    if (mol_my_proc() == 0) {
26      int init_data = 0;
27      ilb_message(root_mp, do_work_handler, &init_data, sizeof(int), NULL);
28    }
29
30    // Make sure all work is done before exiting ...
31    while (!done) { ilb_poll(); }
32
33    return 0;
34  }
```

around areas of interest in the data domain. For instance, as the tip of a growing crack in a solid enters a sub-domain used in adaptive mesh refinement, the computational complexity associated with further refinement of that sub-domain will rise rapidly. Furthermore, as this computational complexity varies unpredictably, static *a priori* partitionings will often lead to underutilization of processing resources. In other words, compile time analysis alone is insufficient.

Second, asynchronous codes lack inherent synchronization points that can be used to ex-

change processor workload information. Any synchronization points added for the purposes of load balancing will negatively impact performance, often severely. The cost incurred by synchronization is proportional to the disparity between the execution times of tasks currently running on each processor. The same unpredictable behavior that leads to poor *a priori* static partitionings also contributes to the cost of global processor synchronization.

Finally, it is our experience that most such codes are computationally bound, meaning the tasks that compose the application have a small "surface-to-volume" ratio. This fact impacts the rate at which processors will be able to service network messages during task execution; the computation kernels which make up these tasks often do not poll for messages, meaning that a mechanism must be provided to process high priority load balancing requests in a timely manner.

In order to create a flexible load balancing architecture well-suited to improving the performance of imbalanced dynamic, adaptive, and asynchronous applications, it is instructive to decompose the load balancing process into its constituent parts. By examining how each phase in the load balancing process is impacted by adaptivity in the application, we can ascertain the characteristics the runtime software must possess in order to be effective. Once these characteristics are known, we can develop an architecture which both preserves the runtime model already provided by the DMCS and MOL layers of the runtime system and allows for maximal load balancing efficiency.

The first phase in the load balancing process is *gathering local processor load information.* With this information, the runtime system can devise a picture of the load distribution among the processing elements at any given time. There are two methods by which this information may be gathered. The first is through application-supplied hints, which rely

on the ability of the application itself to predict the computational weight of each task. However, in the context of adaptive applications, this is often difficult to do with any degree of certainty. The second approach is based on runtime instrumentation, and often makes the assumption that task performance in the near future is based on that of the recent past. Referring to our example of crack tip growth, however, we can see that this is not always the case. With unpredictably and rapidly evolving workload patterns, both methods are impractical. Application requirements therefore dictate that the load balancing architecture reduce its reliance on accurate future workload prediction.

Once local load information is gathered, the second phase is the *dissemination of workload information to a set of neighboring processors.* This set of neighbors may or may not include all other processors in the parallel system. Again, there are two methods for information dispersal. The first is synchronous, meaning that all processors in the neighborhood must synchronize, and all information is exchanged at once. This implies that computation stops while load balancing is in progress. As we have mentioned, the codes in which we are interested do not have inherent synchronization points, so this process will incur some amount of overhead. Conversely, asynchronous techniques allow computation to progress while processor workload is exchanged. However, if this information is not acted upon quickly, load balancing decisions may be made based on out-of-date information. What is required from the runtime system is therefore an asynchronous information exchange mechanism which allows for rapid decision making based on up-to-date information.

The third and final step in the load balancing process is *decision making and task migration.* This phase can be initiated explicitly by the user through an interface routine from within the runtime library. This method has the advantage of not interrupting well

tuned computation kernels, but delays in decision making combined with asynchronous information dissemination can lead directly to poor load balancing and unnecessary idle cycles. Implicitly (or *preemptively*) initiated load balancing can reduce the delay associated with workload information processing. However, any overhead caused by the decision making procedure will have a negative impact on application performance. Therefore, it is necessary that the load balancing runtime system provide a preemptive decision making mechanism capable of balancing the tradeoff between information processing response time and overhead.

With these requirements established, we will now present the architecture of the load balancing component of the PREMA runtime system. The primary elements of the architecture are Schedulable Objects, which bind computation to data and serve as the units of migration during load balancing, the Scheduler interface, which defines the operations that each scheduling policy must provide in order to be used in conjunction with the load balancing framework, and the preemptive decision making capability. In the remainder of this Section, we will explore each of these pieces in greater detail.

5.4.1 Schedulable Objects

Schedulable Objects encapsulate application-defined data objects as well as any pending work that has arrived for that object in the form of application message handlers. In this way, Schedulable Objects extend the idea of mobile objects found in the Mobile Object Layer by providing the runtime system with information necessary to make load balancing migration decisions, and functionality needed to enact those decisions. Schedulable Objects communicate information and provide services through the asynchronous *callback* routines

defined in Chapter 5, Section 5.2.2. These callback routines allow the runtime system to asynchronously query Schedulable Objects during the load balancing process.

In order for the load balancer to make informed migration decisions, Schedulable Objects must be able to provide information describing the processor workload they represent, and their affinity for execution on each processor in the parallel system. There are two methods used to provide this information. The first is to assign a static *load value* and *priority map* to each Schedulable Object at creation time. The load value is an unsigned integer index which describes the load associated with that object; higher values indicate a greater load. The priority map contains an index for each processor in the parallel system; Schedulable Objects that need to migrate tend to move to processors for which they have higher priority indices. Because the load value and priority map are static, they can be obtained with very little overhead. At the same time, however, they are ill-suited to reflect the runtime changes commonly found in adaptive applications, such as dynamically changing workloads and data-dependence relationships between Schedulable Objects.

To express these dynamic characteristics and relationships, we have designed and implemented a method for assigning dynamic load values and priority maps to each Schedulable Object during runtime. Application-defined callback routines are assigned to each object at creation time and are invoked by the runtime system prior to object migration. The priority callback routine can take into account dynamic changes in the load associated with the Schedulable Object, the cost of migrating the object, and the location of objects which share data dependencies. Again, however, there is a tradeoff between functionality and speed; while dynamic runtime information allows the runtime system to more accurately consider evolving dependency patterns among application data objects, a greater amount

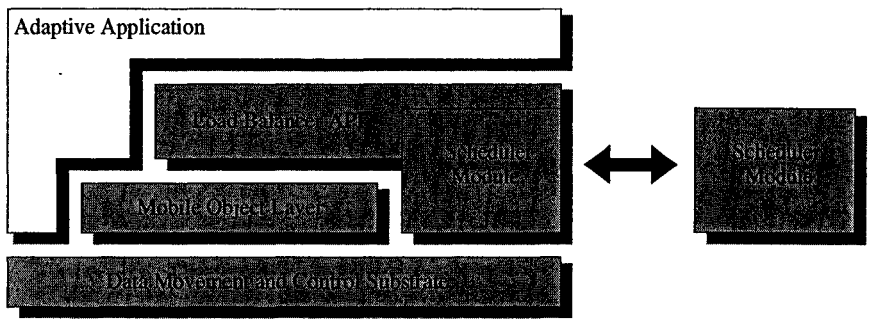
of overhead is involved in their calculation. This overhead is typically quite small, and we have found that this mechanism is quite useful in load balancing adaptive applications.

Once the runtime system has determine that Schedulable Object migration is necessary in order to balance the runtime load, the first step is to pack an object into a contiguous buffer that may be sent across the network in a single message. Because the structure of each object is known only to the application, *packing* and *unpacking* callback routines must be provided. These routines are associated with each Schedulable Object at the time of creation, and are responsible for packing the object into a buffer provided and managed by the runtime system, and rebuilding any necessary data structures once the buffer has arrived at the target processor. If application data objects are simple objects that already exist in contiguous memory, this process amounts to only a memory copy to and from the runtime system managed memory buffers. In order to ensure that the system managed buffers are of adequate size, a callback routine is also necessary that calculates the size of a given Schedulable Object.

5.4.2 Scheduler Module

A design goal we set forth for the ILB library is for applications to have the ability to quickly and easily adopt new load balancing strategies with minimal changes to existing source 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 object that is isolated from the rest of the load balancing infrastructure by a well-defined and simple interface. While the primary purpose of the Scheduler is to schedule the execution and migration of Schedulable Objects during runtime, the exact policies to make

Figure 5.7: SCHEDULER MODULE WITHIN THE PREMA ARCHITECTURE



these decisions are left to the individual Scheduler implementation.

We will begin by examining how the Scheduler module fits into the architecture of the runtime system. We will then examine the interface to which each Scheduler implementation must conform. We will show that this interface allows for a wide variety of Scheduler designs and implementations. To give some idea as to the breadth of scheduling policies that are implementable within our framework, we will describe the Scheduler modules that we have already developed.

The Scheduler modules we have implemented fall into four categories: *Diffusion* model Schedulers, *Gradient* model Schedulers, *Prioritized Multi-list* model Schedulers, and *Master-Worker* model Schedulers. We will provide descriptions of their architectures here, while we will defer a discussion of their performance merits until Chapter 7.

5.4.2.1 Scheduler Interface

From Figure 5.7, we can see that the Scheduler itself is isolated from the application layer by the ILB interface. This allows us to create a "plug-and-play" architecture, wherein the Scheduler may be modified or replace with minimal impact to the application itself, and no impact to the remainder of the runtime system. In order to accomplish this, we define a

Table 5.5: INTERFACE FOR THE ILB SCHEDULER MODULE

Constructor	Create and initialize the Scheduler
Destructor	Destroy Scheduler and clean up memory
*ilb_schedulable_t** **get_next**()	Returns a Schedulable Object with pending work
void **activate**(*ilb_schedulable_t** *sched*)	Notify Scheduler of a newly created or arrived Schedulable Object
void **notify**(*ilb_schedulable_t** *sched*)	Notify Scheduler of a change to a Schedulable Object
bool **empty**()	Return true if Scheduler has no remaining pending work

concise interface (Table 5.5) which must be supported by all Scheduler implementations. We have endeavored to make this interface as general as possible while still allowing for high-performance Scheduler implementations. Below, we will describe the Scheduler module's interface in greater detail.

Any Scheduler module implementing a specific scheduling policy must be implemented as a C++ class which inherits its interface from the *ilb_scheduler_t* class. Although all methods declared in the interface class must be implemented, each Scheduler module is free to implement other methods to be used internally. However, these methods will not be visible to the rest of the runtime system, or to the application itself.

The first method that must be implemented is the constructor for the Scheduler class. The constructor may take any number of parameters, which can be used to initialize any internal data members. For instance, a Scheduler may elect to have any thresholds required for load balancing, or the number of neighbors in the load balancing neighborhood, set via the constructor. In addition, the constructor for the Scheduler class must ensure that all data structures, such as work pools and task queues, that are needed are initialized and ready for use. The constructor is called by the application code; at its conclusion the Scheduler module must be ready to be registered with the PREMA library and used for

Schedulable Object execution and load balancing.

Conversely, the Scheduler must also implement a destructor which is responsible for cleaning up any dynamically allocated buffers or data structures. The Scheduler's destructor method is called by the runtime system itself during system shutdown. At this point, no further load balancing or Schedulable Object execution is possible.

In addition, four methods of the Scheduler module have specified names, parameter lists, and return types. The first of these is the *get_next()* method, which returns the next Schedulable Object with pending work for execution. This method is called exclusively from within the load balancer's *polling* routine, and returns a pointer to an object of type *ilb_schedulable_t*[1]. The Scheduler object examines its internal data structures (such as task pools) to determine if any Schedulable Objects are available for execution, and returns a pointer to one that satisfies the criteria and load balancing policy implemented by the particular Scheduler. For instance, the Diffusion Schedulers will look to its work pool for Schedulable Objects. If any are present, one is removed and its address is returned to the caller.

In the case of a Scheduling policy in which work is *pulled* from an overload processor to an underloaded one, the *get_next()* routine is where load balancing begins. Here, the Scheduler can determine when the processor load level drops below a specified threshold, and can therefore begin the information dissemination procedure, requesting the availability of tasks from neighboring processors. For Scheduler policies that *push* work from overloaded processors, the *get_next()* routine will typically return a pointer to a Schedulable Object,

[1]Note that previously in this document we have referred to Schedulable Objects, whereas now we refer to objects of type *ilb_schedulable_t*. These are equivalent.

or NULL if there are none available.

The *activate()* method is used to communicate the existence of new Schedulable Objects to the Scheduler module. This method is invoked either when a new Schedulable Object is created, or when a Schedulable Object arrives at a processor as a result of migration. Note that newly created Schedulable Objects often have no pending work associated with them, and therefore may not be of interest to the load balancing scheduling policy. Migrating Schedulable Objects often will have pending work, but this is not necessarily the case. The *activate()* method is called exclusively by the runtime system and never by the application; invocation arises from within the Schedulable Object constructor or from within the Schedulable Object installation method.

The Scheduler may elect to examine the length of the Schedulable Object's pending handler queue (using the *handler_queue_size()* method), and use this information to make a decision as to what to do with the Schedulable Object. Schedulers that calculate local load based solely on the number of objects with pending work may elect to do nothing in the *activate()* method if the Schedulable Object in question currently has no pending message handlers. However, if there is work associated with the object, as is often the case during migration, the Scheduler may elect to insert it in a work pool data structure. This decision is left entirely to the scheduling policy implemented by the particular Scheduler.

Schedulable Objects that have already been activated use the *notify()* method to inform the Scheduler of a change in the workload associated with the object. The Scheduler may elect to use this information to adjust the execution order of local Schedulable Objects, or determine which objects are the best candidates for migration. For instance, Schedulers that maintain load information based only on Schedulable Objects that currently have pending

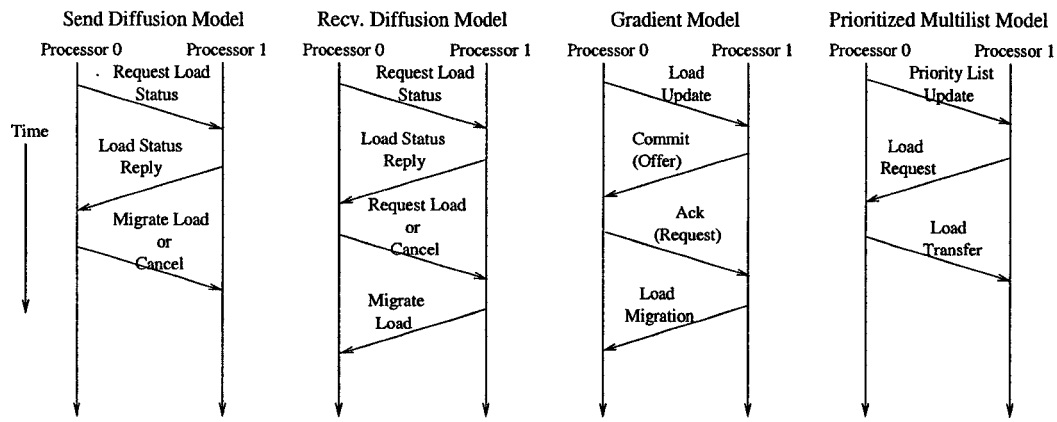
message handlers use the *notify()* method to insert the Schedulable Object in any task pool data structures. Again, the exact behavior of the notification method is entirely dependent upon the load balancing policy implemented by the Scheduler.

The notification method is called exclusively by the runtime system from one of two places. The first is from the load balancing library's system-defined message handler that is executed upon arrival of application messages. This system handler is responsible for enqueueing newly arrived application handlers in the pending handler queue of the target Schedulable Object. This will cause the load represented by the Schedulable Object to increase; notifying the Scheduler may cause the Schedulable Object to move up in the execution order. The second location from which the *notify()* method is invoked is from within the *activate()* method previously discussed. If the activation method determines that pending work is already associated with the Schedulable Object (as is often the case when the object has migrated due to load imbalance), it may make use of the notification method.

The final method mandated by the Scheduler interface is the *empty()* method, which returns the boolean value *true* if there are no local Schedulable Objects with pending work, and *false* otherwise. This method is used exclusively by the termination detection algorithm built into the load balancing runtime system.

It is possible to implement a wide variety of scheduling policies using the interface we have defined. We have attempted to place the minimum number of constraints possible by keeping the interface concise. In order to give some idea of the breadth of policies that can fit into the framework, we have implemented several of the more common load balancing methods (Figure 5.8). In the remainder of this Subsection, we will describe the Schedulers

Figure 5.8: SCHEDULER COMMUNICATION MODELS



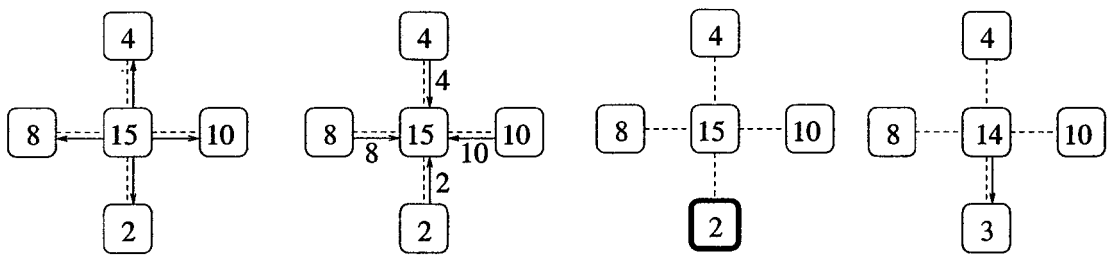
we have implemented in greater detail.

5.4.2.2 Diffusion Model Schedulers

The first Scheduler modules we have implemented are of the *Diffusion* [58] type. During the course of this research, we have experimented with both *Sender-Initiated Diffusion* (SID) and *Receiver-Initiated Diffusion* (RID) scheduling policies. However, we have discovered that RID Schedulers are much more effective at load balancing adaptive and irregular applications, and are also much simpler for application developers to use. Therefore, we will give a description of both scheduler types, but will focus primarily on the RID variety. The performance data contained in this thesis will also be from RID Scheduler modules whenever diffusion schedulers are used.

Sender-Initiated Diffusion Schedulers maintain a task pool for local Schedulable Objects that currently have pending message handlers awaiting execution. The Schedulable Objects are sorted within each task pool according to their estimated load value. Local Schedulable Objects scheduled for execution are taken from the head of the task pool, as are Schedulable

Figure 5.9: SENDER-INITIATED DIFFUSION LOAD BALANCING



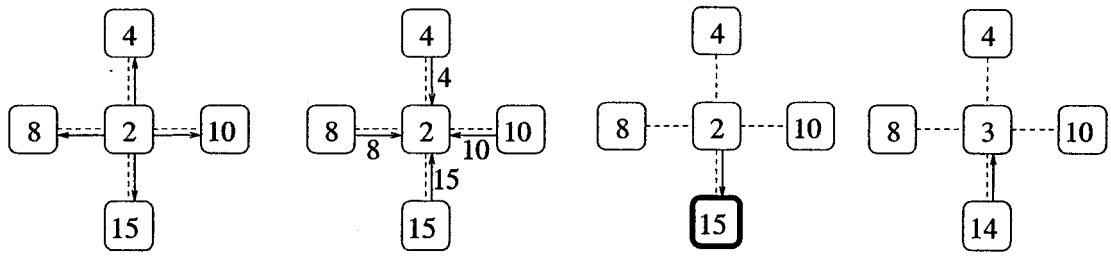
(A) Request load levels (B) Reply with local level (C) Find underloaded (D) Migrate task

Objects that are selected for migration. In this way, the Scheduler will attempt to schedule "heavy" tasks first.

Load balancing for the SID Scheduler begins when an application-defined message arrives for a Schedulable Object. If the target object had no previous pending work, then the load represented by the new message handler is added to the aggregate processor load, possibly raising the work level above the application-specified *high water mark* threshold. A primary issue in using SID Schedulers is the difficulty in selecting an appropriate watermark; setting the mark too low will result in unnecessary task migration while setting it too high will prevent load balancing from beginning soon enough, reducing its effectiveness. An alternative approach is for the Scheduler to "ping" the processor's neighbors periodically, thereby maintaining a picture of the neighborhood load distribution that is updated after some period of time. There is a tradeoff between the accuracy of each Scheduler's view of the neighborhood load and the overhead due to communication. We however, have not implemented such a scheduler.

Figure 5.9 depicts the SID Scheduler's load balancing algorithm. The overloaded processor begins by requesting the current load levels of its immediate neighbors. Once the Scheduler has the current neighbor loads, it is able to determine which neighbor is the

Figure 5.10: RECEIVER-INITIATED DIFFUSION LOAD BALANCING



(A) Request load levels (B) Reply with local level (C) Find overloaded (D) Migrate task

least loaded; this neighbor will be the recipient of migrated tasks. The processor will also determine whether to migrate a single task, or multiple tasks so that each processor will have an equal load after task migration. We have found that, because tasks tend to be coarse-grained and have an unpredictable execution time, it is most effective to migrate a single task at a time. The overhead of task migration tends to be very small as compared to task execution time, so multiple migrations tends to not be a factor. In addition, the unpredictability of task run times often means that tasks migrated to balance the load will have to be migrated again in the future.

Once the least loaded processor in the neighborhood is known, local Schedulable Objects are uninstalled and migrated to the underloaded neighbor. In addition to the previously mentioned issue of determining an appropriate high water mark, the SID scheduling policies have the disadvantage that they place the burden of load balancing computation on the processors that are overloaded. We will see that Receiver-Initiated Diffusion policies reverse this, thereby distributing the application and load balancing computation among the available processors.

The RID Scheduler differs from its SID counterpart in that load balancing is initiated when the local work level falls *below* a particular threshold. The local load is checked

therefore not when new application messages arrive, but when pending message handlers for Schedulable Objects are executed. While it may seem that picking a suitable *low water mark* is no simpler than picking the high water mark necessary for the SID Schedulers, it is often the case that a threshold of zero can be used; load balancing will therefore not start until the local tasks are completely exhausted. In some cases, the performance afforded by this strategy can be improved by starting the load balancing procedure when the last task begins execution; in this way, computation and load balancing communication can be overlapped, yielding a reduction in overall runtime.

Figure 5.10 depicts the RID Scheduler's load balancing algorithm. The first two phases are identical to the SID Scheduler; the processor initiating load balancing must query its immediate neighbors for their local loads. Once all of the neighbors have replied, the underloaded processor determines which neighbor has the greatest level of current work. A message is then sent to this overloaded node requesting a Schedulable Object to be migrated.

A strategy that can be employed to improve the performance of both SID and RID Schedulers is to allow processors that initiate load balancing to select new neighbors to replace those that are unable to accept work (in the SID case) or contribute work (in the RID case). This will prevent processors that are unable to participate in load balancing from being queried over and over, and will also allow processors to find those that are most beneficial, leading to more efficient task migration.

By studying the communication required to migrate a single task (Figure 5.10), we have implemented an optimization that we term *Work-Stealing*[2]. A primary goal of the

[2]The term "Work-Stealing" has existed in the literature for some time, and is most often used in conjunction with the Diffusion policies we have described here. However, in this thesis, the term is used exlusively to refer to the Diffusion optimization we present. It should also be noted that, like the other Scheduler im-

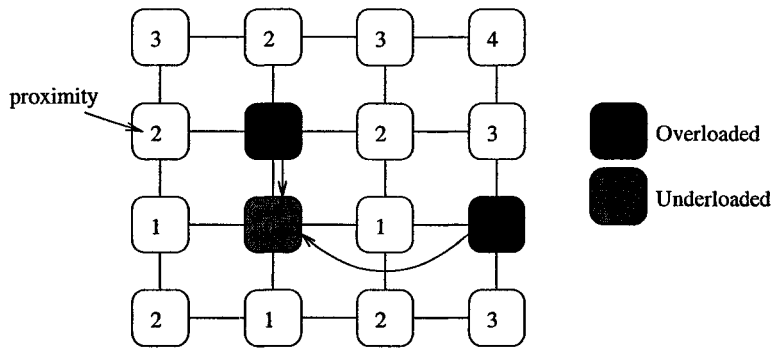
Work-Stealing Scheduler is the elimination the information gathering phases (phases (A) and (B) in Figure 5.10) which can reduce the efficiency of the load balancing algorithm. Each processor is paired with a single neighbor, and load balancing consists of a single request-response mechanism. Underloaded processors request Schedulable Objects from their neighbors, who respond with a task if one is available, or a negative acknowledgement (NACK). If an underloaded processor receives a NACK, then a new neighbor is selected and the process begins again.

The Work-Stealing scheduling policy represents one end point of a continuum that plots communication complexity against load balancing efficiency. By reducing the neighborhood size, each load balancing cycle can be performed less expensively. However, the amount of system state that is available to each processor is also lower, and this can lead to a reduction in load balancing efficiency. In addition, because communication to several processors can often be overlapped, it often takes more time to query N processors in a Work-Stealing environment than is required to query each neighbor in a neighborhood of size N. However, particularly for small processor configurations (less than 64 processors), the experimental evidence we are able to accumulate indicates that simpler communication schemes, such as those in a Work-Stealing Scheduler are able to out-perform more sophisticated but expensive schemes. We will explore this issue in more detail in subsequent Chapters of this thesis.

---

plementations we describe, we do not claim to have originally developed the Work-Stealing policy; instead we wish to indicate the breadth of the load balancing policies that are implementable within our framework. The design and development of the framework is one of the contributions of this thesis.

Figure 5.11: GRADIENT MODEL LOAD BALANCING



5.4.2.3 Gradient Model Scheduler

The *Gradient Model* (GM) Scheduler [13] uses two water marks to delineate the three states in which a processor may exist: a processor may be underloaded, overloaded, or optimally loaded. Like the Diffusion Model Schedulers, the GM Scheduler groups processors into overlapping neighborhoods of a fixed size. These neighborhoods may take into account physical groupings creating by the network topology, or may be arbitrary in the case of a flat network. Load balancing begins once a processor drops below the *low water mark* threshold. Therefore, as with the Receiver-Initiated Diffusion Scheduler, the local load is calculated each time the pending handlers for a Schedulable Object complete execution (within the *get_next()* Scheduler method, described in Section 5.4.2.1).

The GM Scheduler on each processor contains a *proximity* value that denotes the "distance" to the nearest underloaded processor, along with the processor ID of that underloaded node. This concept can be particularly beneficial to those calculations that benefit from migrating computation in such a way as to preserve a notion of data locality. Once a processor P_u becomes underloaded, it begins the process of propagating its own ID throughout the parallel system by sending a load status message to its immediate neighbors. This

load status message contains processor P_u's ID as well as a proximity value; initially the proximity value is zero.

Upon receipt of a load update message, a processor P_o first checks to see if it is able to satisfy the migration request. This is possible only if P_o's local load value is above the prescribed *high water mark.* If this is not the case, the load status message is propagated through the new neighborhood according to:

$$\text{proximity}(p) = \min_i(\text{proximity}(n_i)) + 1$$

As processors propagate the load status message, they maintain the proximity and ID of the closest underloaded processor. In the case that a processor becomes overloaded, it will attempt to migrate some of its work to this node. Note that the spread of load status messages can be halted if they arrive at processors that are already aware of an underloaded node with a lower proximity

While load status messages propagate through the parallel system through message passing between immediate neighbors, load migration may take place between any two processors. This is in contrast to the Gradient methods described in [35, 41, 121, 156] in which tasks are also passed only between neighbors. Figure 5.11 depicts a 16 processor array in which the lightly shaded processor is underloaded while the two darkly shaded processors are overloaded. The numbers in each processor box represent the proximity to the nearest underloaded processor.

Migrating Schedulable Objects to underloaded nodes is a two step process. First, an overloaded processor P_o sends a message to the nearest underloaded processor P_u containing the number of Schedulable Objects it is able to migrate, if multiple Schedulable Objects

are to be transferred in a single load balancing iteration. If only single tasks are to be migrated, this message will simply specify that a Schedulable Object is available. This is known as the *commit* phase of the migration protocol. By splitting task migration into two phases, the Scheduler can avoid the situation in which an underloaded processor becomes overloaded due to simultaneous task migrations from several sources.

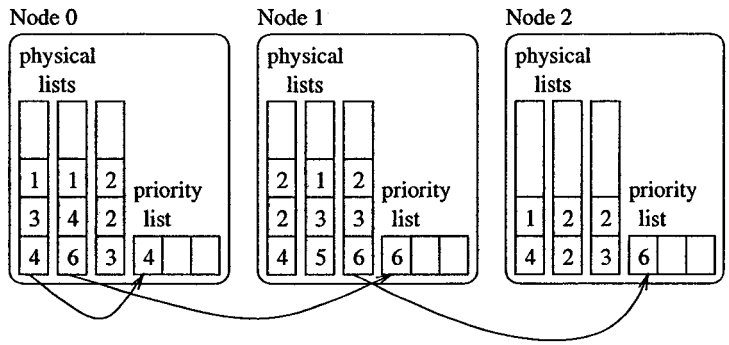
Processor P_u will respond to a single overloaded processor with the number of tasks it is requesting (this is one in the case where single tasks are migrating at any one time). This number may be less than or equal to the number of tasks offered. For instance, if another processor has beaten processor P_o with an offer, P_u may respond to P_o with a request for zero Schedulable Objects. Processor P_u will always take into consideration the number of tasks pledged to it when calculating its local load.

The second stage of load transfer is the *migration* stage. When P_o receives a reply from P_u requesting some number of tasks, P_o uninstalls that number of Schedulable Objects and packs them into a contiguous buffer. The requested tasks are therefore sent in a single message. Figure 5.8 depicts the overall communication pattern graphically.

5.4.2.4 Prioritized Multi-list Scheduler

The *Prioritized Multi-list* (PML) Scheduler we have implemented is based on the work contained in [158]. Like the Gradient Model Scheduler, any two processors may act as source and sink for load transfer. However, the PML Scheduler does away with the concept of localized neighborhoods. Each of the P processors maintains P *physical lists* which contain the local Schedulable Objects for which pending work exists. The ith physical list is sorted using the ith entry in each Schedulable Object's priority vector (Section 5.4.1) as

Figure 5.12: LOCAL PRIORITY VECTORS IN THE PML ARCHITECTURE



a key. This distinguishes the PML Scheduler from the others that we have implemented, because the other Schedulers make use of only a single local task list, the Schedulable Objects contained within the task list of processor p are sorted using the pth entry of the priority vector.

In addition to the P physical lists, each processor maintains a *priority list* of length P. Entry i in processor p's priority list denotes the priority of the Schedulable Object at the head of the pth physical list on processor i, as well as the processor on which the Schedulable Object is located (processor i). Examining this priority list for the greatest value tells the Scheduler where it may find the Schedulable Object with the highest priority; this is the Schedulable Object that should execute next[3]. Figure 5.12 shows the physical list and priority list architecture. The numbers in the physical list entries refer to the priority of the Schedulable Object located there.

Once processor i receives a migration request from processor j, it will remove the Schedulable Object at the head of the jth local physical list and transfer it to processor j. Note that due to the aging of load and priority information, the object at the head of the physical

[3]In order to preclude the need to scan the priority list in a linear fashion, it is implemented as a sorted list, so that the head of the list always denotes the processor with the highest priority Schedulable Object.

may not have the priority that processor j assumes it to have. For instance, another processor may have already executed the message handlers pending for the requested Schedulable Object. In this case, the PML Scheduler removes whatever object happens to be at the head of the physical list and packages it for transport. While this may violate the property that Schedulable Objects should be executed in the order that their priorities dictate, this solution was deemed to be more efficient than sending back a negative acknowledgement message containing the new priority of the head of the physical list. Figure 5.8 shows this communication pattern. In the case that no Schedulable Objects are available for a requesting processor, a negative acknowledgement is returned. When no local objects are available, a message is sent to each processor so that the appropriate entries in the priority lists may be invalidated.

5.4.2.5 Master-Worker Scheduler

The *Master-Worker* Scheduler differs from the others we have described in that all Schedulable Objects are initially allocated to a centralized Master processor, which is responsible for doling out tasks to the Workers. With PREMA's preemptive mechanism, it is possible for the Master processor to also act as a Worker. Without preemption, it is necessary for the Master processor to solely monitor the network for incoming migration requests; otherwise the idle cycles incurred by the Worker nodes waiting for task migration will have a drastic impact on application performance.

At application initiation, all Schedulable Objects are located on the Master processor. This may have implications for applications which are large enough so that the physical memory available is not adequate to hold all tasks. As application messages arrive for

the Schedulable Objects, they are enqueued in a task pool which contains all pending computation for the application. Worker processors request tasks from the Master, which are then allocated in chunks of a fixed number of tasks. The size of the migration chunks represents a tradeoff; small chunk sizes will lead to a more even load distribution but will incur a higher communication cost. Typically, due to the coarse granularity of tasks, a single Schedulable Object is used to satisfy each migration request.

Master-Worker load balancing schemes will generate optimal load balancing for a given initial ordering of tasks in the Master processor's task pool. However, they suffer from several short comings. First is that the Master processor will tend to act as a bottleneck on computation as the number of Worker processors increases. We will explore this issue in greater detail elsewhere in this thesis. Another potential problem is that, for iterative applications, the system state will need to be "reset" at the beginning of each iteration. In the worst case, this will involve migrating all Schedulable Objects back to the Master processor. Finally, as task granularity becomes more fine, the migration chunk size must be used to limit the load balancing communication overhead.

Several strategies exist for dealing with each of these issues. For instance, in large systems a hierarchical Master processor network can be used to alleviate the communication traffic on a single Master. Also, methods exist, such as those described in [12, 74], for allowing tasks to be initially distributed among the workers. The PREMA load balancing framework we have described can easily accommodate both optimizations, although we have not implemented them.

5.4.3 Preemption Mechanism

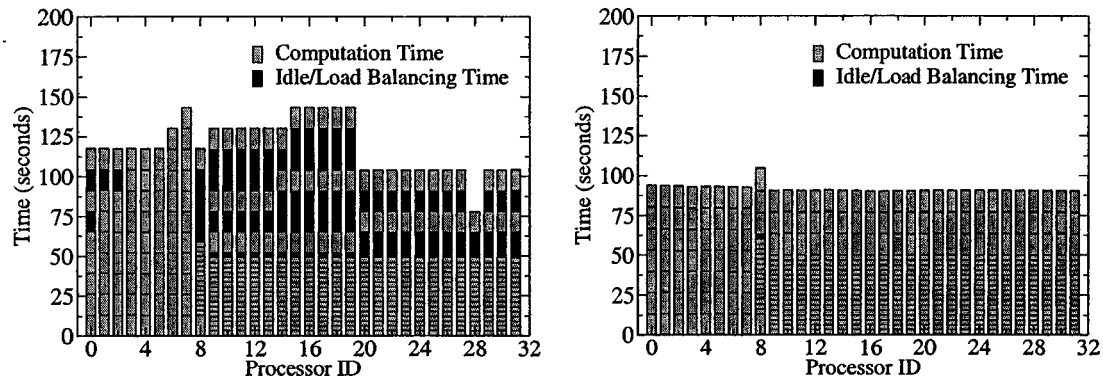
Although the scheduling algorithms discussed so far are well known in the literature, we discovered several problems when attempting to adapt them to our specific target application types. For instance, the 3D parallel advancing front mesh generation program we wish to study makes use of a relatively small number of coarse grained work units[4]. Typically, the number of tasks is less than ten times the number of processors, and each task can take from several minutes to over an hour to execute.

Second, it is often the case 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 pre-compiled 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 request 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.

As an example to illustrate this point, we provide Figure 5.13, which graphically shows the runtime performance of a micro-benchmark code executing on 32 processors. The benchmark is composed of 512 tasks which are classified as either "heavy" or "light," depending upon their required execution time. One quarter of the tasks are heavy and require approximately 4.25 times the computation time of light tasks. In Figure 5.13, the gray color represents task execution time, while the black represents idle time or load balancing time.

[4]The codes used to evaluate the performance of the PREMA runtime library are described more fully in the performance evaluation Chapter of this thesis.

Figure 5.13: IMPACT OF PREEMPTION ON LOAD BALANCING PERFORMANCE



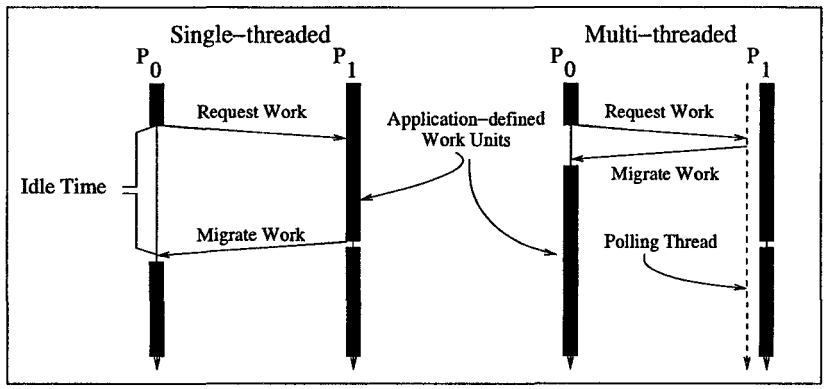
(A) Without preemption (B) With preemption

We can see the number of idle cycles increases dramatically without preemption, leading to a poorer quality of load distribution and a longer total execution time.

We have developed a multi-threaded preemptive approach which nicely solves this problem. Our strategy is to spawn a *polling thread* during system initialization, which will periodically awaken according to a specified polling *quantum*. Upon awakening, the polling thread will examine the network for any load balancing messages that have arrived during the previous quantum. This allows each processor to maintain up-to-date information regarding system or neighborhood status, as well as satisfy any pending load balancing requests in a timely manner.

Figure 5.14 provides an intuitive example that explains the benefit of this approach. As processor P_0 exhausts its local task supply, it requests work from its neighbor P_1. Note that this method is the *Work Stealing* load balancing strategy described in Section 5.4.2.2. This request arrives during the execution of a task. Without preemption, processor P_1 must complete the currently executing task before it will be able to process and respond to the

Figure 5.14: HOW PREEMPTION REDUCES LOAD BALANCING RESPONSE TIME



request. In the meantime, processor P_0 may be sitting idle. In the case in which no tasks are available for migration, P_0 will have to repeat this process with a new neighbor, leading to a large number of wasted cycles. With preemption, the idle time is bounded by the choice of the polling quantum.

The polling quantum represents a tradeoff between timely message response and overhead added to the ongoing computation. For very small quantum, the overhead imposed by the polling thread can dominate the runtime. Overly large quanta, however, will prevent optimal load balancing. We explore this issue in greater detail in Chapter 6 of this thesis.

Note that, without preemption, the idle time may be reduced by beginning the load balancing process while there is still a buffer of local work awaiting execution. However, we have found that, in practice, this solution is not always satisfactory. Setting the load balancing threshold at too high of a value will lead to unnecessary task migration, and may often exacerbate an already poor load distribution. Conversely, setting the threshold too low will incur a large number of idle cycles. With a small number of coarse-grained tasks per processor, there is often no middle ground.

Chapter 6

Modeling Dynamic Load Balancing

Previously in this thesis, we have described the PREMA runtime model and software implementation, which is adept at supporting asynchronous and dynamic parallel computations. However, several questions have been left unanswered to this point, namely how system software parameters and platform performance characteristics affect load balancing performance and overall application runtime. In particular, we are interested in studying the impact on performance caused by the following three elements:

- *Degree of application decomposition.* The computation contained within an application is discretized into a series of *tasks*, which may or may not communicate among themselves. Generating a greater number of tasks will most often lead to a more even potential load distribution. However, this comes with a cost, typically in the form of increased communication requirements. The analytic model we develop will allow the developer to quickly analyze this tradeoff.

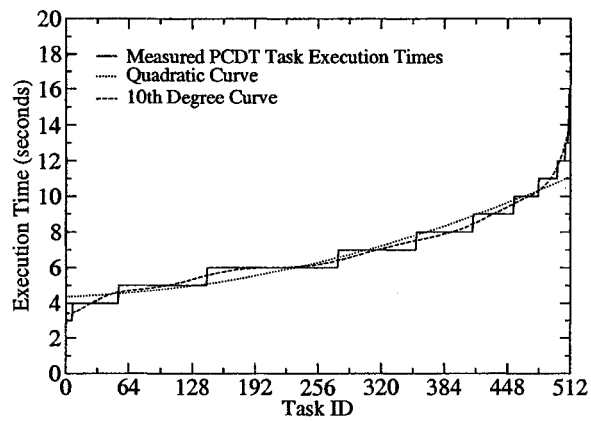
- *Runtime system software parameters.* The PREMA system we have described provides a great deal of flexibility to the application developer in terms of load balancing policies and implementations. However, it is often difficult to intuitively understand

the impact caused by particular values for the systems parameters. Two parameters in which we are most interested are the length of the preemptive polling thread's *quantum*, and the number of neighbors in a load balancing neighborhood. Both variables represent a tradeoff; small quantum values may lead to rapid replies to load balancing requests, but will incur larger amounts of overhead attributable to the polling thread. Similarly, increasing the size of the load balancing neighborhood may lead to fewer necessary load balancing iterations, at a greater cost per cycle. What is especially unclear is how these parameters interact.

- *Performance characteristics of the parallel platform.* In particular, we wish to examine how characteristics such as message passing latency and bandwidth can impact overall application performance and load balancing quality.

However, accurately modeling a generalized task distribution is a very difficult analytic problem. We will therefore begin our discussion in Section 6.1 with an overview of our work in developing analytic techniques which are able to accurately estimate the runtime of applications with a particular class of task weight distributions. This work involves the derivation of a "bi-modal" *approximation function* which can be used to model an application's task execution. In the simplest case, (see Section 6.1), such applications are composed of tasks that can be described using a single average weight estimation and a maximum deviation which dictates how much individual tasks are allowed to vary from that average. This class encompasses those applications whose task execution times are contained within an exponential distribution around a fixed mean, or describe a linear function.

Figure 6.1: TASK EXECUTION TIMES FOR PCDT



There are, however, applications whose task weight distributions cannot be described in this manner. An example is Parallel Constrained Delaunay Triangulation (PCDT), whose refinement times are related to the square (in two dimensions) of the area bound placed on the sub-domain (Figure 6.1). While, from an analytic point of view it is desirous to retain a bi-modal approximation function, our method for deriving it must be refined. In Section 6.2, we extend our mechanism used to create the approximation function in order to account for this more general class of applications.

Once an appropriate bi-modal approximation function is derived, the techniques described in Section 6.3 can be used to estimate an application's runtime in the presence of dynamic load balancing. We then conclude this Chapter by verifying the accuracy of our model's predictions using both benchmark programs and a PCDT application, and conducting a parametric study to evaluate the impact of the previously mentioned runtime parameters on load balancing performance.

6.1 Modeling Simple Task Distributions

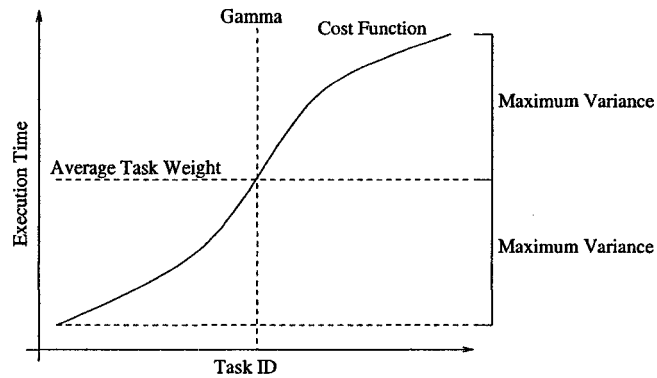
Accurately modeling a generalized task distribution analytically is a very challenging problem. Therefore, we begin by simplifying the problem; we will define a class of applications characterised by a task distribution in which task weights may not vary from the average by more than a fixed percentage. By sorting the tasks so that their execution times are monotonically increasing, we can define the parameter *Gamma* (Γ) which divides the task pool into two classes. In this Section, for simplicity, we have selected an approximation technique which dictates that Γ is equal to half the number of tasks (Figure 6.2).

We then approximate the *cost function* of the original task weights ($task\_weight = f(task\_id)$) using a "bi-modal" *approximation function* that defines two classes of equally weighted tasks, separated at point Γ. Note that it is not necessary to have a precisely defined cost function, as this would imply *a priori* knowledge of task execution times. As our goal is to estimate optimal values for such parameters as preemption quantum and task granularity, coarse estimations often suffice.

We will model the heavier tasks using a class which we will term *Alpha* (α), which will be composed of tasks with equal weight T_{α_task}. The lighter tasks will be approximated with a group we call *Beta* (β), which is made up of tasks with equal weight T_{β_task}.

The average task weight, along with the maximum variance from that weight are supplied as input to the model (Figure 6.3). With this information, the model is able to calculate upper and lower bounds on the predicted program execution time using a specified number of processors. The lower bound is computed under the assumption that all tasks are identical

Figure 6.2: BI-MODAL DIVISION OF TASK EXECUTION TIMES

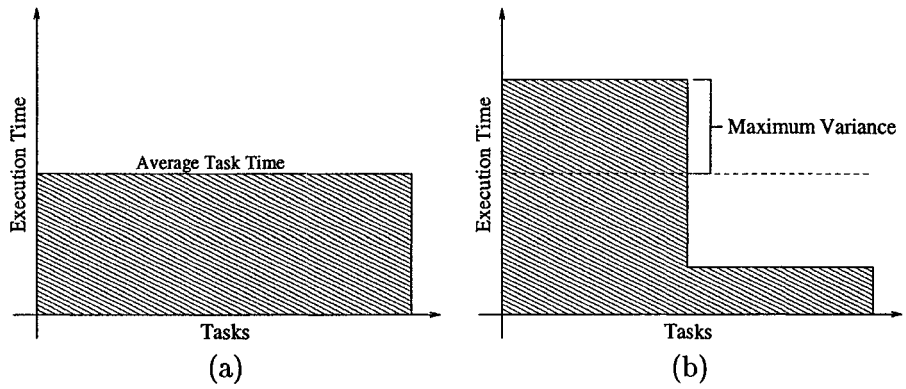


to the specified average task weight. Assigning an equal number of tasks to each processor[1] will yield an equal distribution of computation across all processors. Similarly, the upper bound can be computed by assigning half of the tasks to the α class, and giving them a computation weight equal to the average plus the maximum variance. The other half are assigned to the β class and given a weight of the average minus the maximum variance. In essence, we have approximated the original problem by constructing a new one composed of only two types of tasks. It is this simplification which allows us to tackle the modeling of dynamic load balancing analytically.

In Figure 6.4, we demonstrate that this modeling method works well for those applications whose task execution times can be modeled with this bi-modal approximation function. We have created a benchmark program which can be parameterized with the number of tasks, the average task weight, and the maximum allowable variance from this average. In this particular example, the number of tasks is given on the x-axis. The benchmark contains an aggregate of roughly 5000 total seconds of computation, and a maximum task

[1]Our models assume that an equal number of tasks are assigned to each processor at the start of the application.

Figure 6.3: SIMPLIFIED TASK DISTRIBUTION

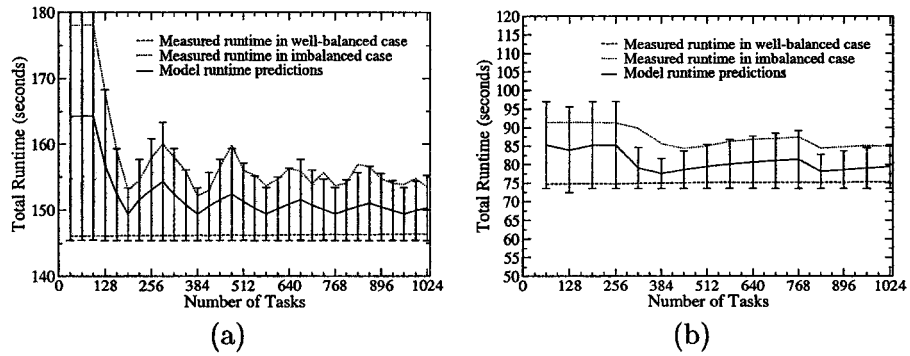


The minimum and maximum potential program execution times are represented by even initial task distributions (a) and task distributions with the maximum allowable variance (b).

variance of 25%. On a 32 processor cluster composed of 650 MHz Sun UltraSPARC IIe processors, our model was able to predict the benchmark runtime in both the balanced and imbalanced cases with a maximum error of approximately 3.5% (Figure 6.4(a)). The same experiment performed on a 64 processor cluster composed of 333 MHz Sun UltraSPARC IIi processors yielded similar results (Figure 6.4(b)), with a maximum error of 11%. However, as the number of tasks increased, the error on 32 processors dropped to 2%, while on 64 processors the error was reduced to 6%. This is acceptable, as we are typically interested in studying the performance of large-scale problems.

However, the aproximation method we have described is not without problems. Most importantly, separating tasks into two equally sized α and β classes, whose weights are separated from an average task weight by a maximum allowable variance, does not suffice for more generalized task distributions. As an example, the task execution times for a sample execution of our Parallel Constrained Delaunay Triangulation (PCDT) program are shown in Figure 6.1. In this case, the average task execution time is 7 seconds, with

Figure 6.4: VERIFICATION OF SIMPLE ANALYTIC MODEL



Actual benchmark execution times in both imbalanced and balanced cases, compared against model predictions on both 32 processors (a) and 64 processors (b). In each case, measured run times are provided for the benchmark for both balanced and imbalanced executions, an average prediction generated by the model, and error bars to indicate the maximum possible variance from the average.

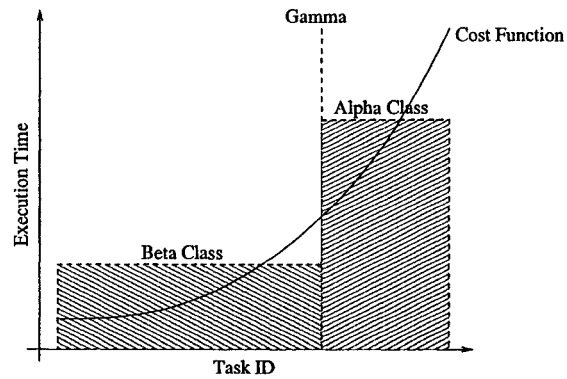
a maximum variance of 9 seconds. This indicates that the weights of the β tasks are -2 seconds! This situation arises in any case in which the cost function describing task weights is of a degree greater than linear. It is therefore clear that our model must be extended to deal with the generalized case; this work is covered in the next Section.

6.2 Modeling General Task Distributions

In this Section, we wish to retain the bi-modal (two-class) approximation of task distribution, due to its analytic tractability, but, in order to handle more general task distributions, redefine the policy used to divide tasks into the two equal classes. Therefore, we are left with the following question: *How can we map general task distributions to the two-class formulation which our analytic model can handle?*

We seek to define a bi-modal step function which approximates the execution times of

Figure 6.5: BI-MODAL DIVISION OF GENERAL TASK EXECUTION TIMES



a set of tasks (defined by a *cost function*) by partitioning it into two subsets which are *not necessarily of equal size*, using the parameter *Gamma* (Γ) as the separation point. This step function provides an approximation of the original problem formatted in such a way that it may be tackled by our analytic modeling technique. However, we wish to eliminate the restriction that each task class must contain an equal number of tasks. We also remove the stipulation that required the α class to contain tasks whose weights were equal to the global task average plus a maximum variance, and the β class to contain those tasks whose weights were equal to the global average minus the maximum variance. In our generalized model, the weights of the α tasks will be independent of the weights of the β tasks.

We are able to define a unique approximation function using the following two criteria:

1. The area under the curve defined by the step function must be equal to the area under the original cost function. In our interpretation, this amounts to stating that the aggregate amount of computation invoked by the tasks in the original cost function must be equal to the amount of computation invoked by the approximation model.

2. The computational complexity of the tasks contained within the α and β approxi-

mation classes must "accurately" reflect the weights of the tasks in the original cost function (we define this more precisely shortly). This ensures that we accurately approximate the time at which load balancing will begin and the amount of computation that will be migrated during each load balancing operation.

Our search for a unique approximation function entails searching for the point which separates the α class from the β class, which we have termed *Gamma* (Γ) (Figure 6.5). For a given selection of Γ, there are unique values for the task weights in each of the approximation classes (T_{α_task} and T_{β_task}) that satisfy Equation 6.1, Equation 6.2, and Equation 6.3, in which T_i is the computational weight of (or time required by) task i and N is the number of tasks. Hence, these equations also satisfy the first criterion defined above.

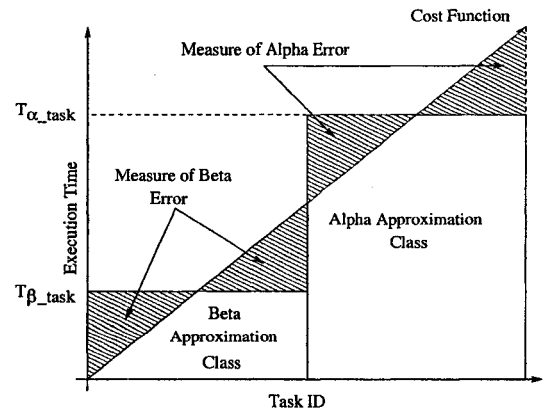
$$Work_{Total} = \sum_{i=1}^{N} T_i = (\Gamma \times T_{\beta\_task}) + ((N - \Gamma) \times T_{\alpha\_task}) \tag{6.1}$$

$$Work_{\beta} = \sum_{i=1}^{\Gamma} T_i = \Gamma \times T_{\beta\_task} \tag{6.2}$$

$$Work_{\alpha} = \sum_{i=\Gamma+1}^{N} T_i = (N - \Gamma) \times T_{\alpha\_task} \tag{6.3}$$

The selection of Γ, from the N possible choices, is determined from our second criterion. We select the unique Γ which minimizes the sum of $Error_{\alpha}$ and $Error_{\beta}$, defined by Equations 6.4 and 6.5 (in the case in which all tasks are of equal weight, Γ will not be unique; however this case will not require any load balancing and so is not considered further). We interpret the definition of $Error_{\alpha}$ to be a measure of the accuracy (as in a least-square approximation [79]) in which our selection for the weight of the α tasks (T_{α_task}) represents the original cost function. This is depicted graphically by the shaded regions of Figure 6.6.

Figure 6.6: MEASURE OF ERROR IN BI-MODAL CLASS APPROXIMATION



The same interpretation holds for $Error_\beta$.

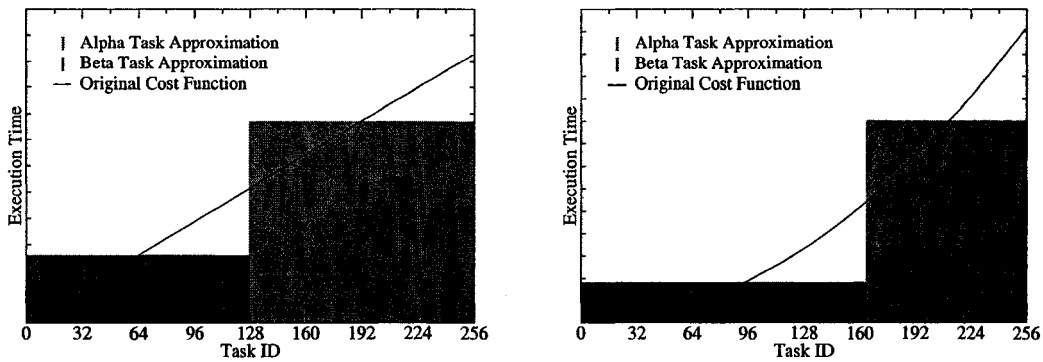
$$Error_\alpha = \sum_{i=\Gamma+1}^{N} (T_{\alpha\_task} - T_i)^2 \tag{6.4}$$

$$Error_\beta = \sum_{i=1}^{\Gamma} (T_{\beta\_task} - T_i)^2 \tag{6.5}$$

Figure 6.7(a) depicts the approximation function's assignment of tasks into α and β classes for a linear initial cost function. In this case, Γ is selected so that half of the tasks belong to each group, indicating that this approximation method reduces to that which was described in Section 6.1 for "simple" task distributions. In addition, the computational weight assigned to the "approximation" tasks contained within each class is equal to the average weight of the actual tasks within the group. However, for a cost function which describes a quadratic task distribution (Figure 6.7(b)), Γ is shifted toward the heavier tasks, with the α and β classes adjusting accordingly.

We defer our discussion of the accuracy of this generalized method until Section 6.4, after we have provided more concrete details concerning our modeling technique.

Figure 6.7: GAMMA FOR LINEAR AND QUADRATIC TASK DISTRIBUTIONS



(a) Linear task distribution (b) Quadratic task distribution

6.3 Analytic Model for Diffusion Load Balancing

We wish to model the execution time of the slowest processor (which we term the *dominating* processor) in the parallel system, as this will determine the overall run time of the parallel application. As there are an infinite number of task execution time distributions, we make the problem tractable by dividing the tasks into two classes, *Alpha* (α) and *Beta* (β). The α tasks are "heavy" tasks and require more computation time than the "lighter" β tasks. We have described our method for creating this bi-modal approximation function in the previous Section.

The processors initially assigned β tasks will finish their computation first, at which point they are termed *underloaded* and will begin the process of requesting tasks from other processors in the parallel system. Each processor's preemptive *polling thread* will awaken periodically (according to the specified preemption *quantum*), interrupt the ongoing computation, and check the network for any such load balancing requests. Upon processing a request, if sufficient tasks are currently in the local work pool, a task will be uninstalled and

migrated to the requesting node. Note that all processors assigned β tasks will exhaust their work at the same time; only processors initially assigned α tasks will possibly have any work to contribute. This equates with the Diffusion scheduler type we have previously described in Chapter 5, Section 5.4.2, which we have found to be the most generally applicable. In addition, the model we present can be trivially extended to encompass the Work-stealing method. We leave modeling the remaining methods as future work.

Once the requisite task partitioning information is defined, Equation 6.6 is used to predict application execution time, and is itself comprised of several terms. The remainder of this section will be used to discuss the derivation of each of these.

$$T_{total} = T_{work} + T_{thread} + T^{app}_{comm} + T^{lb}_{comm} + T^{lb}_{migr} + T^{lb}_{decision-making} - T_{overlap} \tag{6.6}$$

6.3.1 Computation Component

The T_{work} term of Equation 6.6 encompasses the amount of time attributable to task execution. As there is an imbalance of work caused by the discrepancy between α task and β task execution times, dynamic load balancing is possible and we must take this into account. As a simplifying assumption, we state that $\frac{N}{P}$ tasks are assigned to each processor, where N is the total number of tasks and P is the number of available processors ($P \ll N$). We also name processors by the type of task initially assigned to them, with P_α and P_β denoting those processors assigned α and β tasks, respectively. We define N_{P_α} and N_{P_β} to be the number of P_α and P_β processors.

Note that N must be divisible by P in the remainder of this discussion. Furthermore, we stipulate that each task class is grouped together. This is not an unrealistic assumption, as it is often the case that geometric domains in close proximity will be assigned to the

same processor in order to maximize locality. Neighboring regions also tend to be affected by adaptivity in an ongoing computation.

$$T_\alpha = \frac{N}{P} \times T_{\alpha\_task} \tag{6.7}$$

$$T_\beta = \frac{N}{P} \times T_{\beta\_task} \tag{6.8}$$

We know that load balancing will begin once the β tasks have completed, which we model using Equation 6.8. Similarly, Equation 6.7 defines the time required to complete all α tasks, barring migration. Once all β tasks have completed, suitable α tasks must be located; this gives rise to two possible cases. In the best case, a P_β processor will locate an α task in only a single attempt. In the worst case, the P_β processor will probe every other P_β node before a task suitable for migration is located (due to the unpredictable nature of adaptivity, neither the runtime system nor the application knows in advance the location of α tasks). For simplicity, we use the term T_{locate} to describe both instances in the remainder of this discussion, although we define the amount of time required for each attempt in Section 6.3.4.

$$T_\Delta = T_\alpha - (T_\beta + T_{locate}) \tag{6.9}$$

The time between the completion of β tasks and the completion of α tasks is termed T_Δ, and is defined by Equation 6.9. However, as we do not migrate currently executing tasks, only tasks that have not yet begun execution are candidates for load balancing. Knowing the size of each α task, from T_Δ and T_{α_task} we can determine how many α tasks on each P_α processor are actually available for transfer. We term this quantity $M_\alpha = \lfloor \frac{T_\Delta}{T_{\alpha\_task}} \rfloor$. Note that because there are upper and lower bounds on T_Δ (due to the upper and lower bounds

placed on T_{locate}), there are subsequently upper and lower bounds on M_α, which we denote as M_{α_upper} and M_{α_lower}. The upper bound on M_α will result in *more* task migrations, and subsequently a shorter total run time.

To calculate the lower bound of T_{work}, we begin by determining the number of load balancing iterations that will be possible before all tasks are consumed. If each processor in the system consumes a single task per iteration, then the number of tasks consumed per round per P_α processor is given by $C_\alpha = \frac{N_{P_\beta}}{N_{P_\alpha}} + 1$, and the number of possible load balancing rounds is given by $N_{rounds} = \lfloor \frac{M_{\alpha\_upper}}{C_\alpha} \rfloor$. The number of tasks that each P_α processor will donate to migration can then be expressed as $Migr_\alpha = N_{rounds} \times \lfloor \frac{N_{P_\beta}}{N_{P_\alpha}} \rfloor$. Therefore, the lower bound on the time required for task computation on P_α processors is given by:

$$T_{\alpha\_work} = \left(\frac{N}{P} \times \alpha_{minimum} \right) - (Migr_\alpha \times T_{\alpha\_task}) \tag{6.10}$$

where $\alpha_{minimum}$ denotes the minimum task weight of a task in the original cost function approximated by the α class.

We calculate the number of tasks that will be received by each underloaded P_β processor as $Migr_\beta = N_{rounds} \times \lceil \frac{N_{P_\alpha}}{N_{P_\beta}} \rceil$. The computation time required for task execution on β processors is given by:

$$T_{\beta\_work} = \left(\frac{N}{P} \times \beta_{minimum} \right) + (Migr_\beta \times T_{\alpha\_task}) \tag{6.11}$$

where $\beta_{minimum}$ denotes the minimum task weight of a task in the original cost function approximated by the β class.

Because execution time will always be dictated by the slowest processor, we compare T_{α_work} (Equation 6.10) with T_{β_work} (Equation 6.11); the greater value is determined to be

the lower bound on task execution time. Note that in both equations we take an optimistic view on the amount of work that is executed before load balancing begins.

Calculating an upper bound on task execution time is done in a similar manner. Note that using M_{α_lower} will lead to a fewer number of load balancing iterations, and therefore fewer task migrations. A more pessimistic estimation of the weight of computation executed before load balancing begins yields the following equations:

$$T_{\alpha\_work} = \left(\frac{N}{P} \times \alpha_{maximum}\right) - (Migr_{\alpha} \times T_{\alpha\_task}) \tag{6.12}$$

$$T_{\beta\_work} = \left(\frac{N}{P} \times \beta_{maximum}\right) + (Migr_{\beta} \times T_{\alpha\_task}) \tag{6.13}$$

where $\alpha_{maximum}$ and $\beta_{maximum}$ denote the maximum original task weight in each approximation class.

Once again, the greater value is selected as the upper bound on task execution time. Although we evaluate Equation 6.6 for both the upper and lower bounds, in order to simplify our remaining discussion, we will simply refer to T_{work} as the task computation component.

6.3.2 Preemptive Polling Thread Component

Because the preemptive polling thread periodically awakens to interrupt ongoing computation in order to check the network for load balancing messages, it is viewed as adding a fixed percentage of overhead to each task. As input to the model, we are given $T_{quantum}$, which represents the *quantum* or period after which the polling thread will awaken. An additional input is T_{poll}, which is the amount of time required to complete a single polling

operation, and is independent of $T_{quantum}$. With this information we can derive:

$$T_{thread} = \frac{T_{work}}{T_{quantum}} \times (T_{poll} + (2 \times T_{ctx})) \tag{6.14}$$

where T_{ctx} is the time required for a thread context switch.

6.3.3 Application Communication Component

We assume that each task will send a fixed number of messages of a fixed size. Both the number of messages and the message size are input as parameters to the model. We also model message passing (for both the application and the runtime system) as a fixed startup cost ($T_{startup}$) plus a fixed cost per byte of the message (T_{per_byte}). Again, both $T_{startup}$ and T_{per_byte} are parameters to the model. We then define:

$$T_{msg} = T_{startup} + (T_{per\_byte} \times message\_size) \tag{6.15}$$

and the message passing cost per task (T_{msg_task}) as $T_{msg} \times N_{msg}$, where N_{msg} is the number of messages sent by a single task. Note that, although we do not overlap consecutive communication operations, it would be trivial to do so. We then derive:

$$T_{comm}^{app} = \left(\frac{N}{P} - Migr_{\alpha}\right) \times T_{msg\_task} \tag{6.16}$$

$$T_{comm}^{app} = \left(\frac{N}{P} + Migr_{\beta}\right) \times T_{msg\_task} \tag{6.17}$$

Equation 6.16 is used in the case than a P_{α} processor dominates, while Equation 6.17 is used when a P_{β} processor dictates the overall run time.

6.3.4 Load Balancing Communication Component

We focus on developing an analytic modeling technique for the *Diffusion* scheduling strategy discussed previously in Chapter 5, Section 5.4.2.2. This strategy groups processors into overlapping neighborhoods of a size of at least two. When a processor becomes underloaded, it sends an information request to the other processors in the neighborhood, inquiring as to the availability of migratable tasks. However, this procedure is performed only by underloaded, and therefore P_β, processors. Note that we are not yet concerned with the costs of task migration (which involve both P_α and P_β processors), but only with information gathering. In addition, because it is impossible to accurately predict the number of unsuccessful load balancing attempts, we will assume the the number of attempts is equal to the number of migratable tasks, which serves as an optimistic lower bound.

Therefore, if the dominating processor is a P_β processor:

$$T_{comm}^{lb} = \left(T_{req} + \frac{T_{quantum}}{2} + T_{proc\_reply}\right) \times Migr_\beta \tag{6.18}$$

$$T_{req} = (N_{neighbor} \times T_{startup}) + (T_{per\_byte} \times message\_size) \tag{6.19}$$

In Equation 6.18, the term T_{proc_reply} refers to the amount of time the load balancer requires to process a reply to an information request, and is an input parameter to the model. In addition, $\frac{T_{quantum}}{2}$ represents the expected elapsed time on the receiver to process and reply to the request. Since it is possible that the receiver is currently processing a task, this term is the expected amount of time before the preemptive polling thread will awaken and process the request.

Note that Equation 6.19 indicates that the load balancer can overlap communication to

the neighbor set. This equation is trivial to alter if this is not the case.

In the case in which a P_α node is the dominating processor, no load balancing information gathering is required, so we assume this component contributes no time to the predicted application run time.

6.3.5 Load Balancing Migration Component

The T^{lb}_{migr} term of Equation 6.6 represents the time required for task migration. This can be broken into two cases. In the case in which a P_α processor dominates application run time, the cost of uninstalling and packing a task for migration must be considered. This cost is given by Equation 6.20. Conversely, P_β processors must unpack and install migrated tasks, and this cost is defined by Equation 6.21.

$$T^{lb}_{migr} = Migr_\alpha \times [T_{uninstall} + T_{pack} + T_{startup} + (T_{per\_byte} \times task\_size)] \quad (6.20)$$

$$T^{lb}_{migr} = Migr_\beta \times (T_{unpack} + T_{install}) \quad (6.21)$$

The times for $T_{uninstall}$, $T_{install}$, T_{pack}, and T_{unpack} are measured quantities and are provided as input to the model, along with the size of each task in bytes, $task\_size$.

6.3.6 Migration Decision Making Component

The $T^{lb}_{decision\_making}$ term of Equation 6.6 represents the time required for the load balancing scheduling software to select a partner processor once it has received replies to all information request messages. It is from this partner processor that the underloaded node will request a task. There is no equation to define this term as it is a measured quantity and

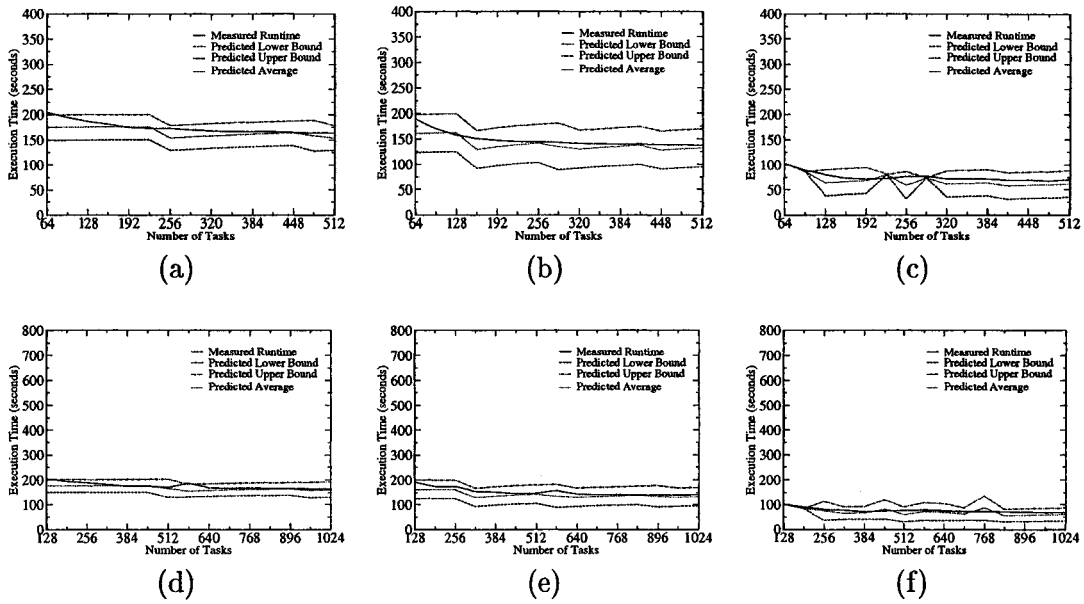
an input to the model. Substituting one load balancing policy for another will impact this value, as will the $N_{neighbor}$ constant used in Equation 6.19.

6.3.7 Accounting for Overlap Between Components

On several parallel architectures, it is possible to overlap communication between processors with ongoing computation. In such an environment, the $T_{overlap}$ term of Equation 6.6 must be non-zero in order to produce as accurate an approximation as possible. This will adjust the model for those instances in which, for example, the communication required for data migration (T^{lb}_{migr}), inter-task communication (T^{app}_{comm}), or load balancing communication (T^{lb}_{comm}) can be overlapped with task computation (T_{work}). The extent to which computation and communication may overlap, along with the number of communication operations, will determine the value of $T_{overlap}$.

It should be noted that the experimental studies and results contained within this thesis were conducted on a platform which does not allow for overlapping communication operations or overlapping communication with computation. Although the Message Passing Interface (MPI) used to construct the Data Movement and Control Substrate component of the PREMA system provides the capability for *immediate* message sends, with only a single processor per compute node (and no processing capability on the network interface card itself), ongoing computation must be suspended in order to complete the sending operation.

Figure 6.8: BENCHMARK VERIFICATION OF MODEL PREDICTIONS



Comparison between measured benchmark run times and model predictions for 32 (row 1) and 64 (row 2) processors.

6.4 Verifying the Analytic Model

With our analytic model defined, the next step is to ascertain its predictive ability. We verify the accuracy of the model using both a benchmark program and an implementation of a 2D Parallel Constrained Delaunay Triangulation (PCDT) mesh refinement algorithm. We utilize the benchmark program in order to accurately control the initial imbalance, while the PCDT code gives some insight into the behavior of the model in a more "real-world" scenario.

Using our benchmark program, we present results from three test cases. In the first, task execution times vary linearly from a minimum value T_{min} to a maximum of T_{max}, where T_{max} is double T_{min} (*linear-2* test). In the second test, the maximum task execution time is a factor of four times the minimum (*linear-4* test), which leads to a more severe

initial imbalance. In the third test, 25% of the tasks are rated as "heavy" tasks, which require double the computation time of the remaining 75% "light-weight" tasks (*step* test). In all three tests, tasks are allocated so that tasks of similar weight are assigned to the same processor.

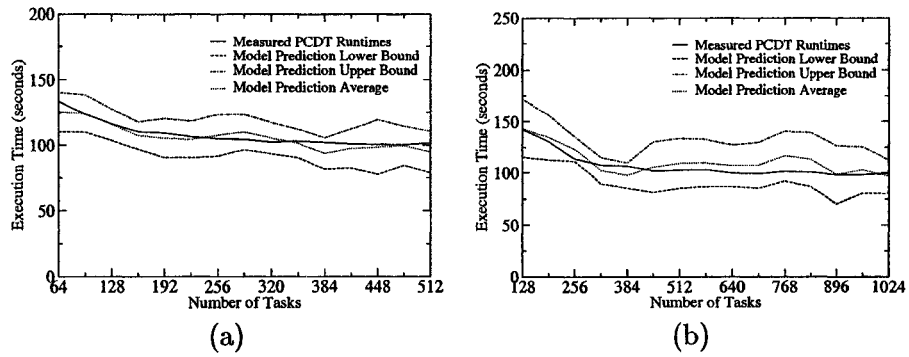
Figure 6.8 contains the results of these tests on both 32 and 64 homogeneous processors. In each test, we vary the granularity of the task decomposition by varying the number of tasks allocated to each processor from 2 to 16. Each graph displays the measured program execution time, along with an upper, lower, and average prediction generated by the analytic model.

In Figure 6.8(a), which contains the results from the *linear-2* test, the expected runtime (prediction average) and the measured execution time differ by an average of only 2.9%, while this figure increases slightly to 4.4% for the *linear-4* test shown in Figure 6.8(b). For the *step* test on 32 processors (Figure 6.8(c)), the average error is roughly 11%. However, this increase can be explained by the shorter total execution time, and the existence of a couple of outlying points. However, for longer execution times, our model becomes more accurate; this is desirable as our modeling tool is most beneficial to long-running, and therefore more expensive, programs.

Figures 6.8(d), 6.8(e), and 6.8(f) contain results from the same tests executed on 64 processors. The average errors are roughly 4% for the *linear-2* and *linear-4* tests, and roughly 10% for the *step* test.

While these results give us some measure of confidence in our model's accuracy, further tests are required before it may be applied to more typical codes. The PCDT application is more challenging for two reasons. First, the load imbalance is generated by a "heavy-tailed"

Figure 6.9: PCDT VERIFICATION OF MODEL PREDICTIONS



Comparison between measured PCDT execution times and predictions generated by the analytic model on 32 (a) and 64 (b) processors.

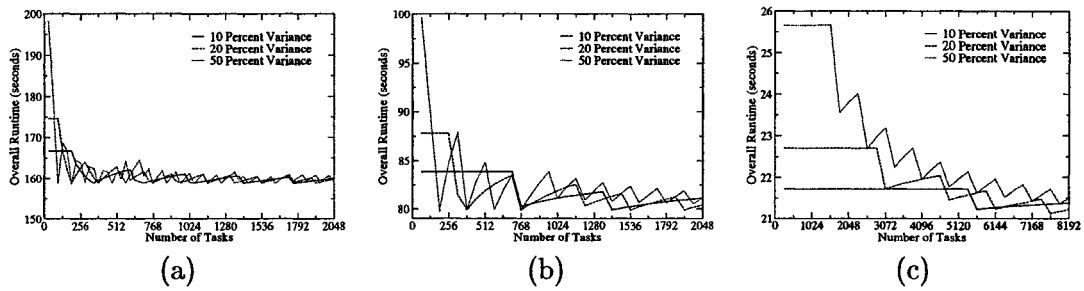
distribution of task execution times (Figure 6.1), not by a linear or stair-step distribution. Secondly, communication between tasks during runtime must be captured (for a more complete description of the PCDT algorithm, refer to Chapter 7).

Figure 6.9 indicates the effectiveness of our model at predicting PCDT performance on 32 and 64 processors. The average error for 32 processors is 3.2%, while on 64 processors this figure increases slightly to 6%. As with our benchmark program, these small error values indicate that our modeling technique is robust enough to capture the added complexities of the PCDT application.

6.5 Parametric Studies

With the accuracy of our modeling technique established, we next want to study the impact variations in certain runtime parameters have on application performance. The specific variables in which we are interested include the preemption quantum, number of processors, task granularity, load balancing neighborhood size, and communication latency. We also

Figure 6.10: BI-MODAL TIMES: VARIANCE VS. DECOMPOSITION



Predicted benchmark execution times for three levels of task variance under the influence of various degrees of over-decomposition on 32 processors (a), 64 processors (b), and 256 processors (c).

want to model applications that conform to both the simpler stair-step imbalance pattern without inter-task communication, and the more complex linear load distribution pattern with application communication. We therefore break this discussion into two components.

6.5.1 Bi-modal Imbalance

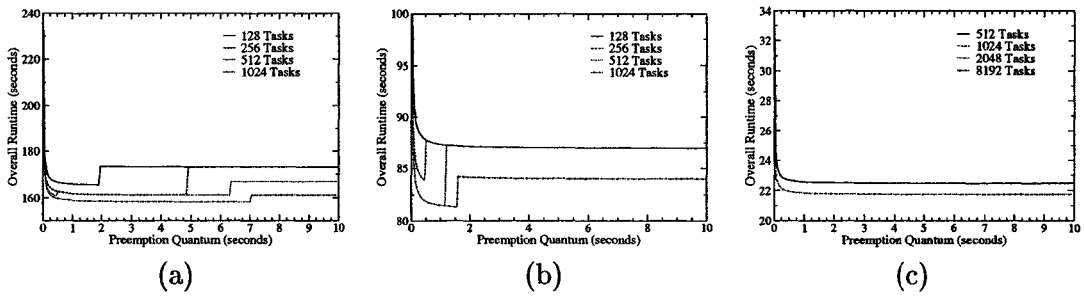
The applications we model here are composed of two types of tasks: heavy and light. Heavy tasks make up 50% of the task count, and the *variance*, or difference in execution time between heavy and light tasks is configurable at runtime.

The first question we wish to ask is how the number of tasks, or the level of *over-decomposition*, affects the overall runtime. Intuitively, we would expect that a greater number of tasks will give the load balancer a greater degree of flexibility in terms of task migration, and we can see from Figure 6.10[2] that this is indeed the case.

In all cases, we notice a certain dampening periodic behavior as the number of tasks increases. This reflects the ability of the load balancing algorithm to evenly distribute the

[2]Please note that the values of the y-axes for the graphs vary.

Figure 6.11: BI-MODAL TIMES: DECOMPOSITION VS. PREEMPTION QUANTA

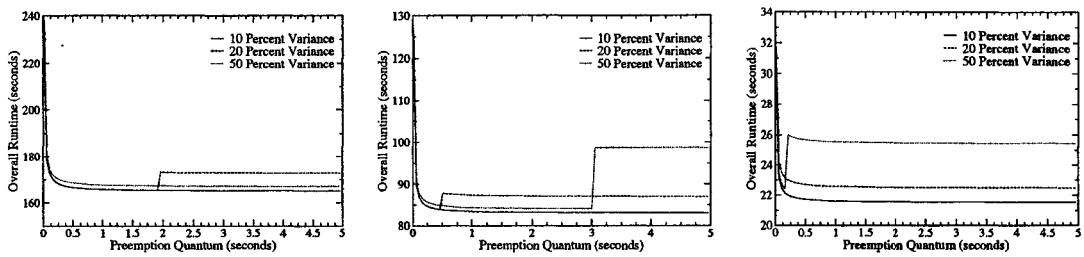


Predicted benchmark execution times for three levels of over-decomposition under the influence of various preemption quanta on 32 processors (a), 64 processors (b), and 256 processors (c).

workload represented by the available tasks; once a task has begun execution it cannot be migrated. Periodically, the task decomposition creates a situation in which the smoothest possible load distribution creates a workload difference between processors of nearly an entire task, creating a peak in the total runtime. Further over-decomposition eliminates this effect by allowing "part" of the original task to migrate. The dampening effect is caused by the decreasing size of this "peak" task, as the task size is inversely proportional to the number of tasks.

We next wish to examine the effect of the preemption *quantum* on the overall runtime. We can see from Figure 6.11 that there is often a range of quantum values that will lead to a local minimum in terms of total execution time. If the quantum is too small, the cost of frequent thread context switches and network polling operations will dominate the computation time. On the other hand, if the quantum is too large, polling will not happen frequently enough and load balancing performance will suffer due to an increased delay in responses to load balancing queries. We also see that increasing the number of tasks allows the load balancing algorithm to be more tolerant of large quanta; this is due again to the

Figure 6.12: BI-MODAL TIMES: VARIANCE VS. PREEMPTION QUANTA



(a) (b) (c)

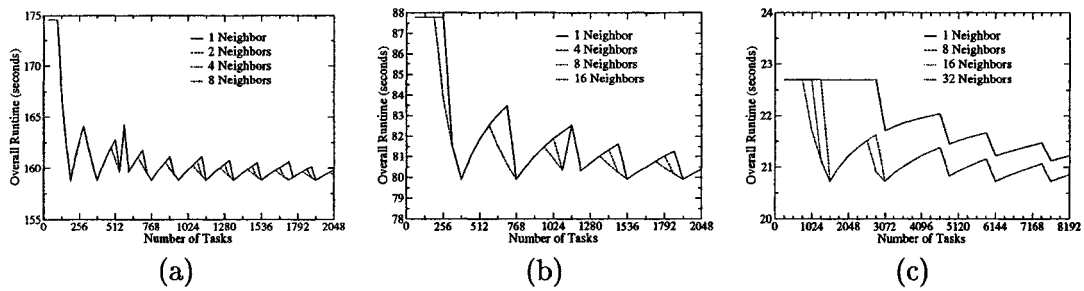
Predicted benchmark execution times for three levels of task variance under the influence of various preemption quanta on 32 processors (a), 64 processors (b), and 256 processors (c).

increased flexibility afforded by the finer granularity of the task decomposition.

Figure 6.12 allows us to study the effect the preemption quantum has as the amount of variance between α and β tasks (and therefore the amount of initial imbalance) changes. Again, we see that certain values of the quantum will lead to a minimum in overall runtime. However, in this case we see that as the variance increases, the load balancer becomes more sensitive to long quanta. In the case of 256 processors and a 50% task variance, we can see that the quantum that leads to a minimal predicted runtime is quite small.

Finally, we wish to examine how the load balancing neighborhood size will impact load balancing efficiency (Figure 6.13). In all previous experiments, each processor was paired with a single partner. If a partner was unable to satisfy a request for task migration, the underloaded processor would pick a new partner and issue a new request. The amount of time required for a request/reply transaction is, in large part, dictated by the preemption quantum; as the number of processors grows, it may not be possible for an underloaded node to issue a request to every processor. As a result, some tasks that are available for load balancing may not be migrated, resulting in a load imbalance. By increasing the number

Figure 6.13: BI-MODAL TIMES: NEIGHBORHOOD SIZE VS. DECOMPOSITION



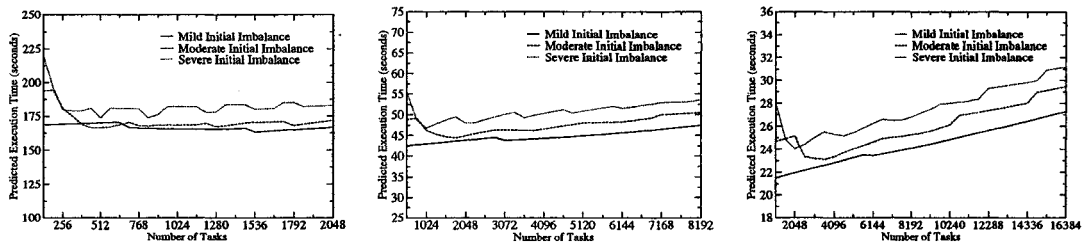
Predicted benchmark execution times for four neighborhood sizes under the influence of various degrees of over-decomposition on 32 processors (a), 64 processors (b), and 256 processors (c).

of processors that belong to each neighborhood, this problem may be overcome. Doing so represents a tradeoff between the cost of each load balancing iteration and the number of iterations necessary to locate tasks for migration. The benefit of a larger neighborhood is particularly apparent in the case of Figure 6.13(c). On 256 processors, it often not possible to locate all available work when each processor is paired with a single neighbor. For large processor configurations, increasing the neighborhood size will lead to performance improvements, while on smaller numbers of nodes small neighborhoods will suffice.

6.5.2 Linear Imbalance

For the second set of experiments with our analytic model, we choose to study a more complex type of application. Instead of a "stair-step" load distribution as we saw before, task execution times vary linearly over one of three ranges. *Mild* imbalance varies task execution times over a range in which the heaviest tasks require roughly 20% more time than the lightest ones. With a *moderate* imbalance, heavy tasks are roughly twice as costly as the lightest ones, while *severe* imbalance increases this range to a factor of four.

Figure 6.14: LINEAR TIMES: IMBALANCE VS. DECOMPOSITION



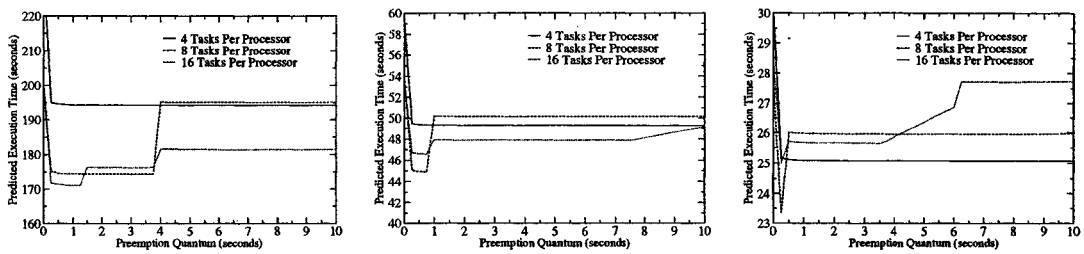
(a) (b) (c)

Predicted benchmark execution times for three levels of initial imbalance under the influence of various degrees of over-decomposition on 64 processors (a), 256 processors (b), and 512 processors (c).

Secondly, tasks now communicate during runtime and this must be captured by the model. We study an application in which each task has four neighbors with whom a single message is exchanged during runtime (the amount of message traffic can be trivially reconfigured, however). We also assume that the computation invoked as a result of each message is constant, as is the size of each message. Note that this is similar to the execution model of a Parallel Constrained Delaunay Triangulation (PCDT) program, which we study in greater detail elsewhere.

As with the previous benchmark, the first point we wish to understand concerns the impact over-decomposition has on overall execution time. As was the case earlier, an increase in the number of tasks that compose an application (and subsequent decrease in task granularity) will afford the load balancer a greater degree of task migration flexibility. However, this freedom is in tension with the greater amount of required inter-task communication. From Figure 6.14, we can see that this tension will eventually penalize greater levels of over-decomposition, particularly in the case of a mild initial imbalance. In this case, tasks must be fine-grained before any migration is possible; however, by this point, the cost due

Figure 6.15: LINEAR TIMES: DECOMPOSITION VS. PREEMPTION QUANTA



(a) (b) (c)

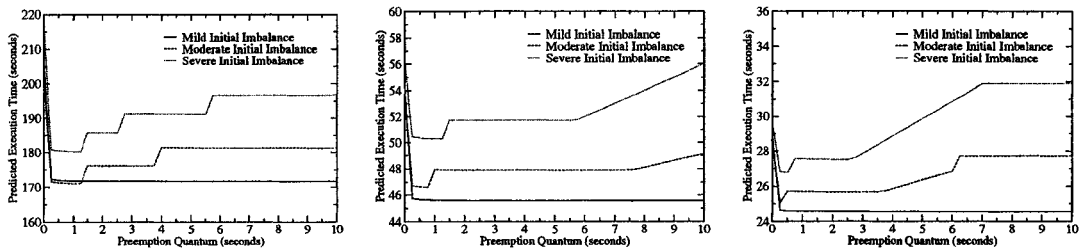
Predicted benchmark execution times for three levels of over-decomposition under the influence of various preemption quanta on 64 processors (a), 256 processors (b), and 512 processors (c).

to computation invoked by inter-task communication will outweigh the benefits gained by improved load distribution. This effect is also seen in the cases of moderate and severe initial imbalance after a local minimum in total execution time is reached. Load balancing is initially effective at reducing the runtime via task redistribution; however, the benefits are overcome as task granularity decreases.

We next examine the effect of the preemption quantum on the overall runtime. As was the case with our previous benchmark, we expect that a quantum value which is too small will lead to excessive thread context switching and network polling, while a quantum which is too large will prevent effective load balancing due to the lengthly delay between request and response messages. From Figure 6.15, we can see that this is indeed the case. What is interesting is that the range of optimal quantum values decreases in size as the number of processors grows. In addition, we again note that decreasing the granularity of the tasks allows the load balancer to be more tolerant of excessively large quantum values.

Figure 6.16 depicts the results of repeating this experiment while varying the level of initial imbalance. We again obtain similar results. Regardless of the initial level of processor

Figure 6.16: LINEAR TIMES: IMBALANCE VS. PREEMPTION QUANTA



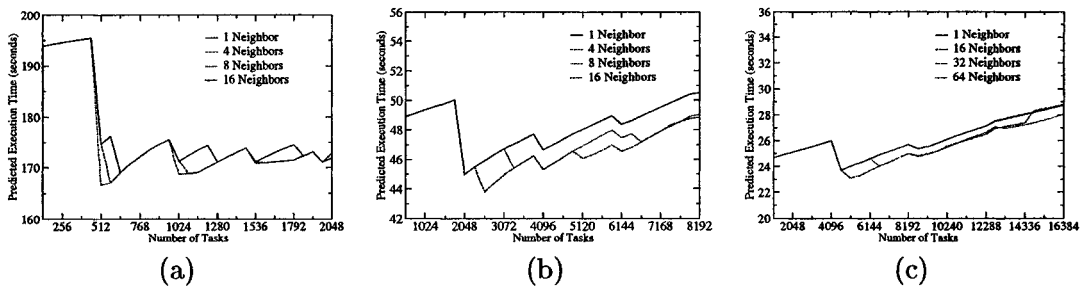
(a) (b) (c)

Predicted benchmark execution times for three levels of initial imbalance under the influence of various preemption quanta on 64 processors (a), 256 processors (b), and 512 processors (c).

imbalance, a range of preemption quantum values exist which will lead to an optimal total execution time. What is critical to note with both benchmarks, is that regardless of the task granularity of initial level of workload imbalance, the optimal quantum values are equivalent. This indicates that certain quantum values will be the most beneficial, regardless of application characteristics.

Lastly, we examine the effect of neighborhood size on load balancing efficiency (Figure 6.17). We can see that for smaller processor configurations (Figure 6.17(a)), neighborhood size does not have a great impact, provided the preemption quantum is not too large (in these experiments, the preemption quantum is 0.5 seconds). This indicates that it is possible for the load balancer to probe all processors in a relatively small amount of time. As the number of processors increases, this is not always feasible. We can see from Figure 6.17(b) that increasing the neighborhood size to 8 or 16 processors can cause a significant performance improvement. As the number of processors increases further (Figure 6.17(c)), the neighborhood size must likewise grow.

Figure 6.17: LINEAR TIMES: NEIGHBORHOOD SIZE VS. DECOMPOSITION



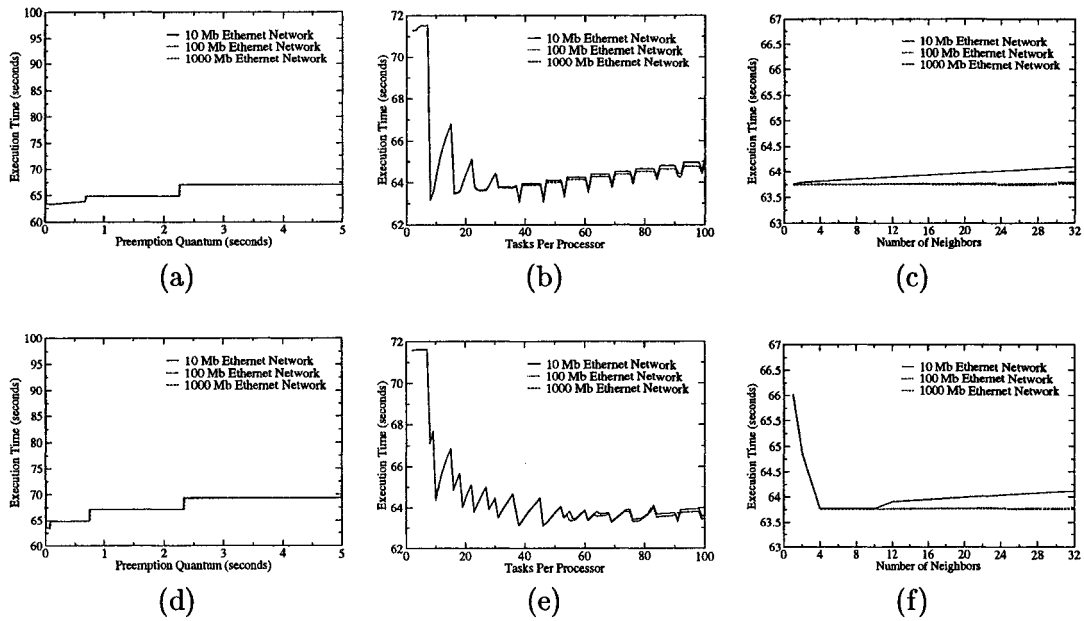
Predicted benchmark execution times for four neighborhood sizes under the influence of various degrees of over-decomposition on 64 processors (a), 256 processors (b), and 512 processors (c).

6.5.3 Impact of Communication Latency

A final set of experiments explores the impact networking technology has on benchmark execution time and load balancing efficiency. We begin with a simple ping-pong program which allows us to gather performance data for 10 Mb, 100 Mb, and 1000 Mb ethernet networking. We then calibrate our model using this data. It should be noted, however, that our analytic model is capable of only an approximation of actual network performance. We model message passing as a startup cost, plus a fixed cost for each byte of payload data. In reality, the latency figures we observed with our ping-pong test were quadratic, not linear. In any case, our linear approximation does not impair the accuracy of the model's predictions.

In Figure 6.18(a) and 6.18(d), we examine the effect of varying the preemption quantum on each type of network. The communication latency's impact in this case is negligible, as the application performance is nearly identical in all three cases, on both small (32 processors) and large (256 processors) parallel configurations. In this particular example,

Figure 6.18: COMMUNICATION LATENCY'S EFFECT ON LOAD BALANCING



Modeling the effect of communication latency on 32 (top row) and 256 (bottom row) processors. Results are shown for varying the polling quantum (column 1), varying the task decomposition (column 2), and varying the number of load balancing neighbors (column 3). Tests are for a linear load distribution with a minimum of 40 seconds and a maximum of 80 seconds of computation per processor.

each processor is initially allocated 16 tasks; decreasing the task granularity (increasing the number of tasks) may result in poorer load balancing performance as the network latency increases. However, we are most interested in improving the performance of coarse decompositions, and in such a case network latency seems to play little role.

Figure 6.18(b) and 6.18(e) contain the predicted application execution time as the task granularity varies, from two tasks per processor to 100. In this case, we can see that, as the task granularity decreases, there is a noticeable penalty for low network performance. For both large and small processor configurations, the predicted run times begin to diverge at a task decomposition of roughly 40 tasks per processor. This observation is in agreement with

our earlier prediction; network latency becomes a factor as the task granularity decreases.

Finally, we study the interaction between network latency and load balancing neighborhood size (Figure 6.18(c) and 6.18(f)). With a parallel configuration of 32 processors, experiments indicate an increase in predicted execution time on 10 Mb ethernet networks, while the higher performance networks remain relatively constant. This is due to the increased cost of each load balancing interaction: querying neighbors and waiting for replies requires more time as network latency increases. However, it can be seen that this impact is nominal; an total runtime increase of less than half a second is predicted. On larger processor configurations, increasing the neighborhood size will initially lead to a lower execution time, due to the resulting improved load distribution. However, once a local minimum is reached, we see the same phenomenon as before. The increased cost associated with each load balancing iteration will lead to a slight increase in total execution time.

Chapter 7

Performance Evaluation

Previously in this document we have described the PREMA approach to load balancing which allows adaptive parallel applications to easily migrate from an explicit message passing execution model, to message passing within a global namespace combined with support for explicit application-controlled load balancing, and finally to system-managed load balancing in which data and computation migration is handled exclusively by the runtime system. We have described the *Data Movement and Control Substrate* (DMCS) (Chapter 4) which forms the foundation of our runtime environment and provides single-sided message passing and remote memory manipulation primitives. Chapter 4 also describe the *Mobile Object Layer* (MOL), which provides a global namespace in the presence of data migration and which forms the message forwarding infrastructure of our load balancing framework. Chapter 5 looked at the *Implicit Load Balancing* (ILB) framework, which provides implicit and dynamic load balancing for adaptive and asynchronous applications.

With this software framework established, what remains is to examine its performance in the context of our target applications. The results of these experiments are contained within this Chapter. As this research is focused on the development of load balancing technology for asynchronous parallel applications, we will focus our experiments in this direction. Our

study is broken into four sections, with each section describing a set of experiments designed to highlight one or more particular aspects of load balancing performance.

The first set of tests examines load balancing performance using a micro-benchmark program developed within our research group. Although a synthetic benchmark program may seem to be a trivial example, there are several important reasons why we begin our discussion here. We will enumerate these in the appropriate section. From there, our second set of experiments looks at a 3D Parallel Advancing Front Technique (PAFT) mesh generation and refinement program. This program is computation bound and very limited in its use of runtime message passing, and therefore serves as an important introductory application. Our third set of tests involve a 2D Parallel Constrained Delaunay Triangulation (PCDT) mesh generation application. This code is not as computationally intense as the PAFT, and makes use of interprocessor message passing during computation. From the load balancer's point of view, these characteristics make the PCDT program more challenging than the PAFT. Next, we examine a Fast Multi-pole Algorithm (FMA) N-body program. As this code was developed prior to PREMA and outside our research group, this will allow us to determine the ease of integrating PREMA into previously existing applications, as well as determine PREMA's applicability to non-meshing codes. In addition, we will examine PREMA's load balancing performance in comparison to a tightly integrated and provably optimal load balancing strategy which was incorporated into the software by its original developer. Up to this point, we have been concerned with load balancing performance of asynchronous applications, as it is this class that originally motivated our research. For the final Section of this Chapter, we will provide some insight into PREMA's performance on loosely-synchronous applications. Our results indicate that PREMA is a robust toolkit

whose applicability extends beyond our original intentions.

7.1 Synthetic Micro-benchmark

We evaluate two aspects of the PREMA system software: the runtime execution model and the software implementation itself. For the first part of this analysis, we will compare the performance of the PREMA library with an example from the family of repartitioning tools (Metis) as well as a representative of the loosely synchronous class of dynamic load balancers (Charm++'s iterative load balancers). We will demonstrate that these load balancing paradigms are not well suited for asynchronous and adaptive applications.

Second, we will compare PREMA's performance with another member of the asynchronous class of load balancers, namely the seed-based balancers implemented by the Charm++ library. We will show that PREMA's design leads to more efficient implementations, which in some cases can have a dramatic effect on reducing overall runtime.

To carry out this analysis, we have implemented a benchmark program which simulates a simple class of asynchronous applications. The application is composed of a number of tasks (either 4, 8, or 16 times the number of processors) which are rated as either "heavy" or "light". Heavy tasks require twice the amount of computation as lighter ones. We begin with either 10% or 25% of the tasks rated as heavy, with heavy tasks clustered together. It is important to note that the amount of computation required by tasks cannot be predicted in advance, making static partitioning algorithms ineffective. In addition, no communication is required between tasks during runtime.

7.1.1 Representative Load Balancing Systems

We wish to evaluate the effectiveness and efficiency of both PREMA's programming model and implementation. To do this, we have chosen three load balancing methods that will serve as a basis for comparison. These methods are widely available and are used by many scientific computing projects within academia and industry.

7.1.1.1 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 (sub-domains). 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 sub-domain surface-to-volume ratio, cut edge weights, or data redistribution costs.

Repartitioning tools are incorporated into such projects as Jostle [153], DRAMA [19], Zoltan [66], and Metis [103]. 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 [140], 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

$$|E_{cut}| + \alpha\,|V_{move}| \tag{7.1}$$

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 [140] 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 [141], 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 7.1) is then computed, with the best option chosen. Finally, a multilevel refinement algorithm is used [139] while minimizing Equation 7.1.

Explicit repartitioning such as these have two disadvantages for adaptive asynchronous applications such as parallel mesh generation and refinement. The first is the global synchronization that is necessary in order to exchange current load information between processors and build the data structures needed for the repartitioning routines. The cost of synchronization can grow both with the number of processors, and with the variance in the

computational work load present; processors with less computationally intense data partitions may reach global synchronization points long before more loaded processors, leading to wasted processor cycles. A second disadvantage stems from the difficulty in predicting future work loads for highly adaptive applications. It is difficult to predict the computational weight associated with particular data sub-domains, meaning that processors after repartitioning may still not be balanced. In order to combat this problem, multiple repartitionings are often necessary, forcing the costs associated with synchronization to be paid multiple times.

7.1.1.2 Iterative Load Balancing in 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++ [100, 98, 99, 101, 102] 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 [23], 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

[1] According to [23], local synchronization barriers are used in all load balancing modules provided in the Charm++ distribution, but are not necessary in custom-designed modules.

the entry point to execute upon reception. Load balancing is achieved by mapping and remapping chares to available processors. An assumption, known as the *principle of persistent computation and communication structure* [24], 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 [23]. 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 sub-domain) 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 [102], 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 we present validates this point.

7.1.1.3 Seed-based Load Balancing in Charm++

Seed load balancing involves the migration of chare creation messages, or "seeds", to create a balance of work across the available processors [130, 31]. Seed-based balancers load balance chares only at creation time; once the chare has been created, the seed-based balancer will not induce subsequent migration. However, the measurement-based balancers we have previously described will perform the task of moving chares during execution in order to achieve dynamic load balancing.

Several variations of seed-based balancers have been implemented and studied within Charm++:

1. *Random:* This strategy will place the seeds randomly among the processors at creation time and performs no seed migration afterward. By default, this is the seed-based balancer used.

2. *Neighbor:* This strategy imposes a "virtual topology" on the processors, and load exchange happens between neighbors only. Overloaded processors initiate load balancing, pushing work to its underloaded neighbors. By default, the topology imposed

on the processors in a 2D mesh, although this can be changed at runtime to one of several other topologies, such as a torus, ring, dense graph, or 3D variants.

3. *Spray:* This strategy imposes a spanning tree organization on the available processors. A global average load value can then be computing using a global reduction communication operation. The load balancer uses this average load information to compute how seeds should be migrated during load balancing.

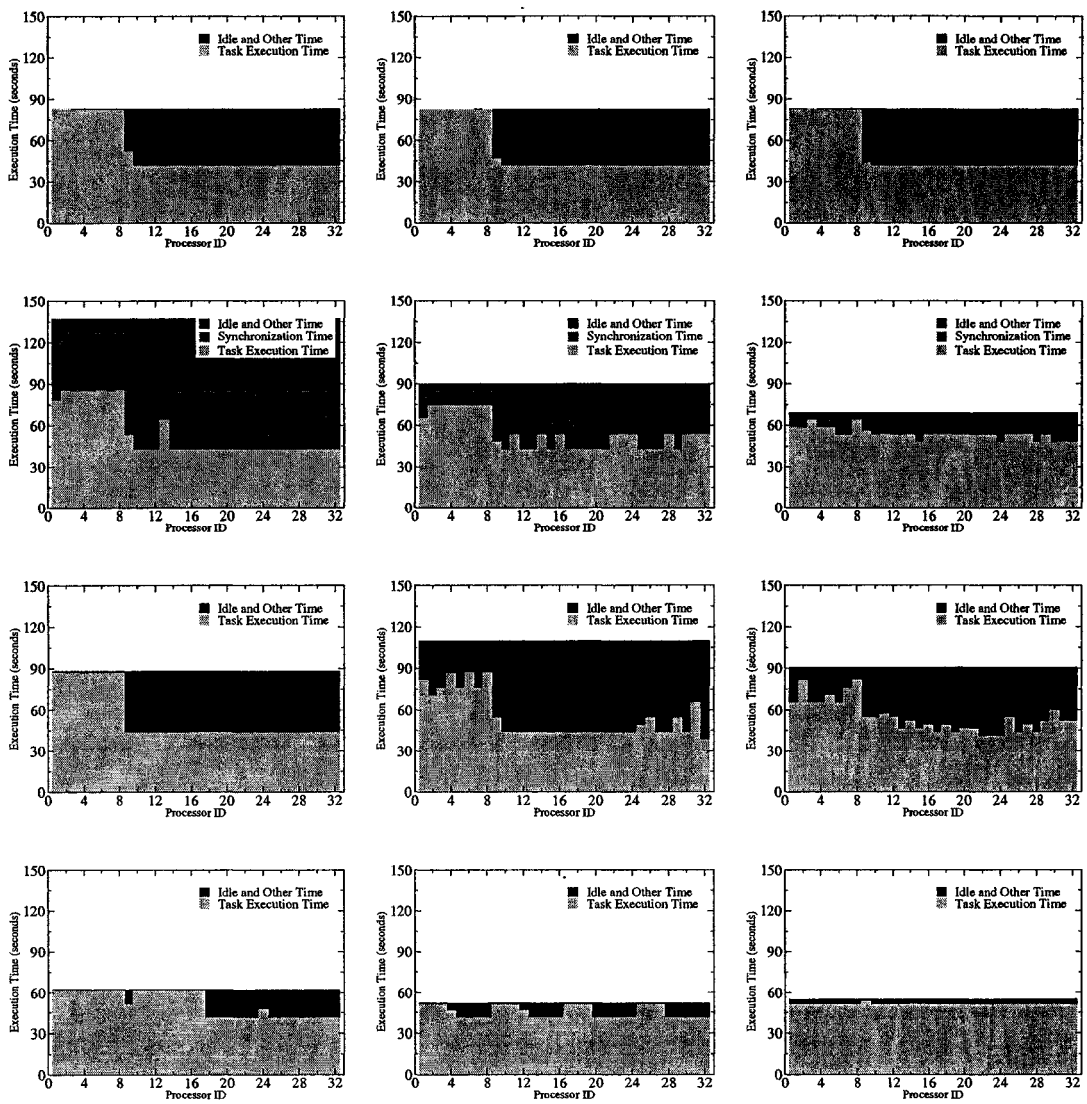
Other load balancing strategies may be implemented using the provided seed load balancer interface.

By default, the random load balancer is always used. For our experiments, however, we make use of the neighbor policy combined with the 3D torus and 2D mesh connection topologies.

7.1.2 Experimental Results

We break the discussion of our experimental results into three parts. First, we make use of both stop-and-repartition and loosely synchronous load balancing tools in order to evaluate the appropriateness of PREMA's execution model. With these experiments, we are able to determine the efficiency of the *design* of the runtime system. We then evaluate the efficiency of its *implementation* using performance data from another asynchronous load balancing tool. Finally, we demonstrate the flexibility of PREMA's load balancing framework by examining the performance of several Scheduler types.

Figure 7.1: BENCHMARK RUN TIMES ON 32 PROCESSORS; 25% HEAVY TASKS



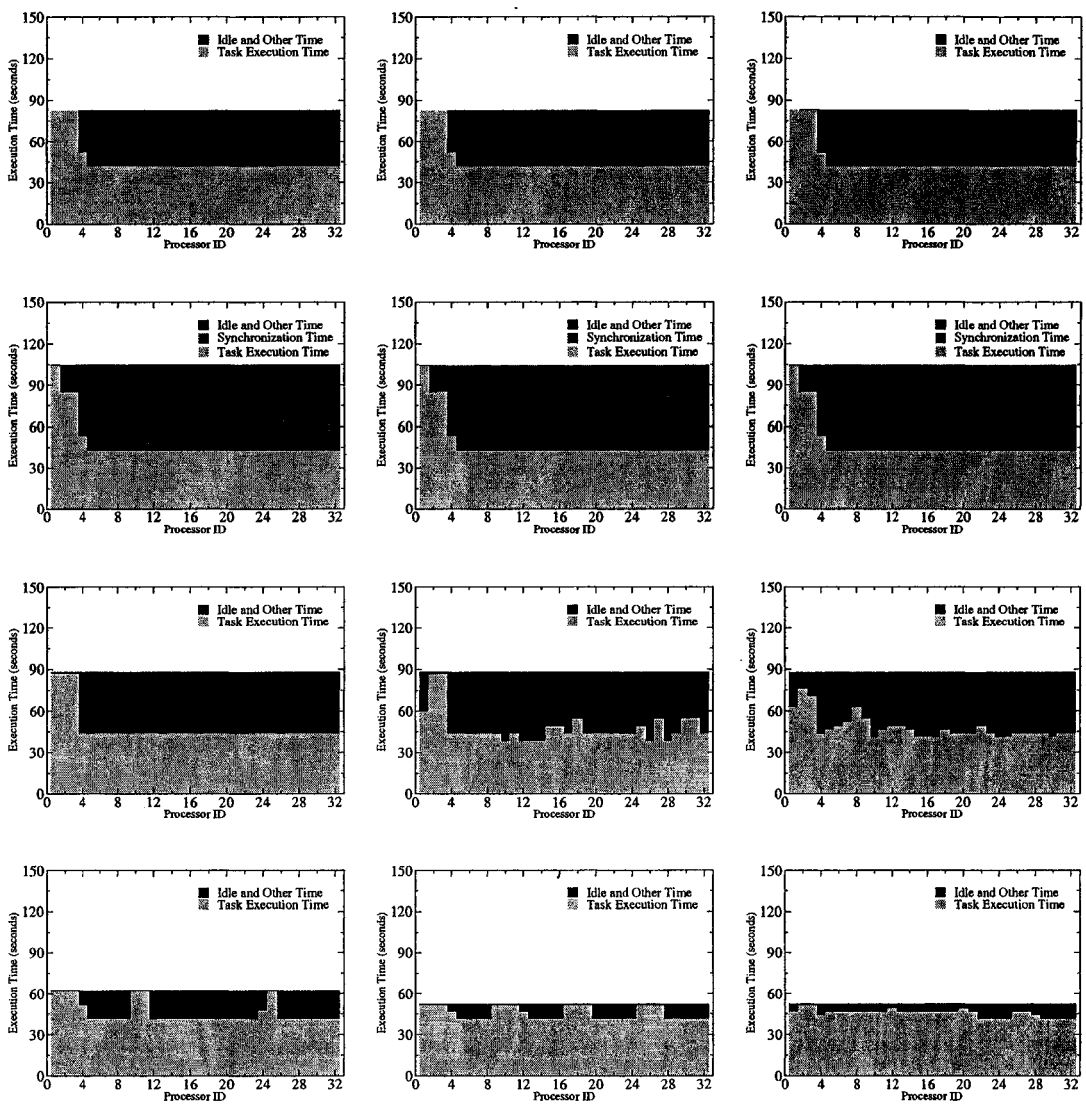
Overall benchmark run times on 32 processors for no load balancing (row 1), load balancing using the Metis repartitioning tool (row 2), load balancing using a Charm++ iterative balancer (row 3), and load balancing using PREMA's Diffusion Scheduler (row 4) with 4 tasks (column 1), 8 tasks (column 2), and 16 tasks (column 3) initially assigned to each processor. 25% of the tasks are rated as "heavy".

7.1.2.1 Evaluation of the PREMA Execution Model

Figure 7.1 contains the overall run times on a 32 processor cluster for our benchmark program with 25% of the tasks rated as heavy. The first row contains execution data for runs with no load balancing at all, with 4 tasks (column 1), 8 tasks (column 2), and 16 tasks (column 3) initially allocated to each processor. We expect these graphs to be identical; the total amount of computation executed by the program is the same, regardless of the task decomposition. Due to the discrepancy between the execution times of "heavy" and "light" tasks, processors 1 through 8 require roughly twice the computation time of the remaining processors. The large area of black in these graphs indicates wasted cycles due to idle processors.

The second row of Figure 7.1 contains the performance of the same benchmark when Metis is used for periodic task repartitioning. Once a processor drops below a predefined threshold, it will request a barrier from all other processors for the purpose of load information exchange and the calculation of new task assignments. If a satisfactory amount of work remains to be completed, a new partitioning is calculated and tasks are subsequently migrated. From Figure 7.1 (row2, column 3), in which there are initially 16 tasks allocated to each processor, we can see that this scheme works reasonably well. However, for several processors the synchronization overhead can grow as high as 24% of the total runtime. However, as the task decomposition becomes more coarse, the costs due to synchronization grow substantially. In Figure 7.1 (row 2, column 1), in which there are only 4 tasks per processor, synchronization costs average roughly 60% of the total runtime.

Figure 7.2: BENCHMARK RUN TIMES ON 32 PROCESSORS; 10% HEAVY TASKS

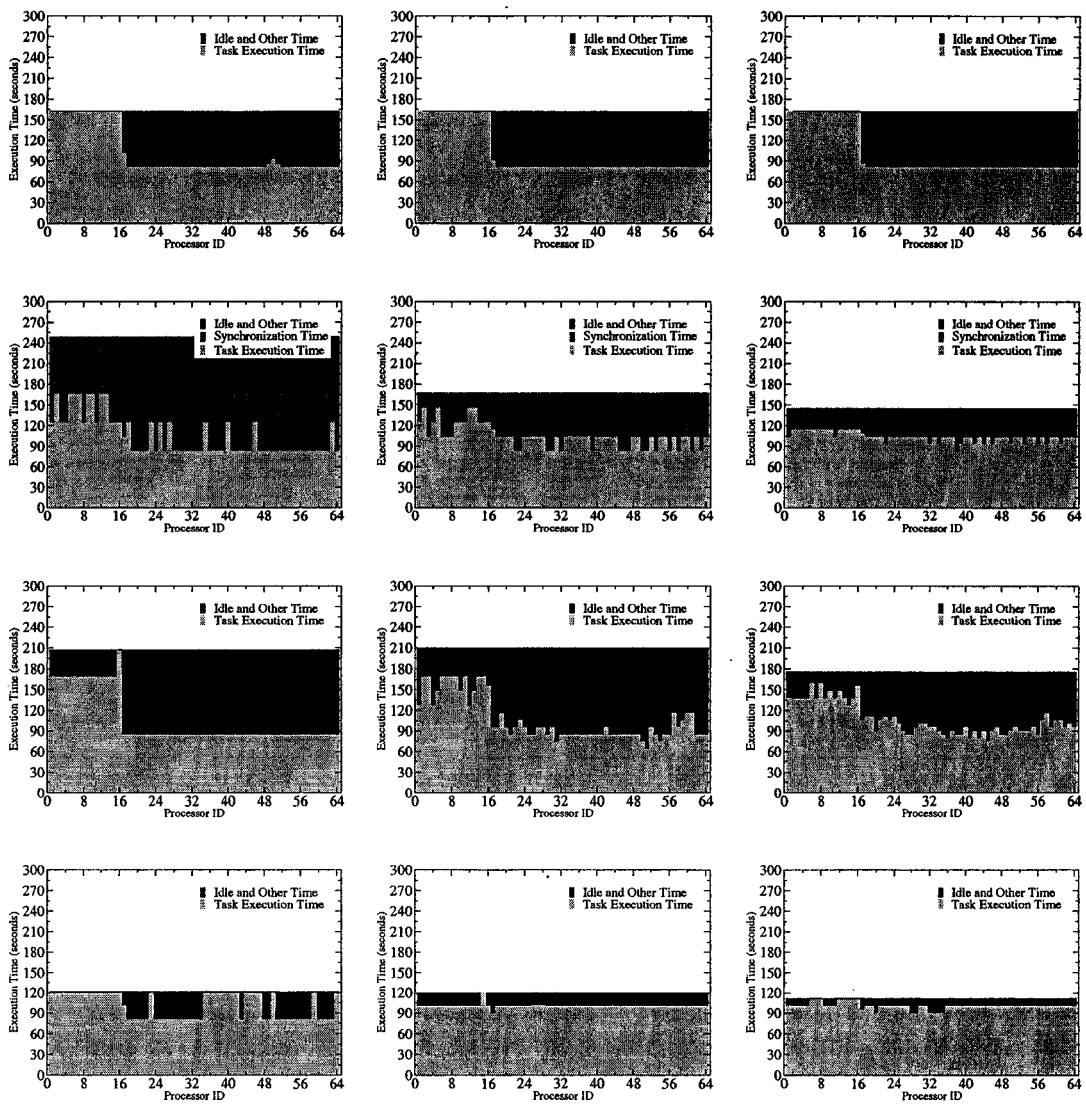


Overall benchmark run times on 32 processors for no load balancing (row 1), load balancing using the Metis repartitioning tool (row 2), load balancing using a Charm++ iterative balancer (row 3), and load balancing using PREMA's Diffusion Scheduler (row 4) with 4 tasks (column 1), 8 tasks (column 2), and 16 tasks (column 3) initially assigned to each processor. 10% of the tasks are rated as "heavy".

The third row of Figure 7.1 depicts the performance of a benchmark which makes use of the iterative algorithms implemented in Charm++ to dynamically balance the workload. Load balancing takes place at *synchronization points*; once a predefined number of tasks have completed, the runtime system will evaluate the load distribution and potentially migrate tasks. We display the results of using four synchronization points, as this seems to be the best balance between quality of load distribution and load balancing overhead. As with Metis, we can see that the quality of the load distribution increases inversely with task granularity. However, we can also see that load balancing and synchronization overhead overcome any benefits gained by load balancing. This is not a fault of the Charm++ implementation; asynchronous applications carry a large penalty for processor synchronization.

PREMA's load balancing framework (Figure 7.1 (row 4)) contains a preemptive *polling thread*, which awakens after each *quantum* of time elapses in order to process and respond to pending requests from neighboring processors. By eliminating synchronization points, the quality of the resulting load distribution, as well as the overall runtime, can be greatly improved. For a coarse task decomposition (column 1), PREMA provides a performance improvement of 25% over no load balancing, 54% over repartitioning with Metis, and a 30% improvement over incremental load balancing using Charm++. As the task granularity decreases (column 3), PREMA is able to provide similar results. PREMA demonstrates a 33% improvement over no load balancing, a 20% improvement over load balancing using Metis, and a 39% improvement over the iterative techniques implemented in Charm++.

Figure 7.3: BENCHMARK RUN TIMES ON 64 PROCESSORS; 25% HEAVY TASKS



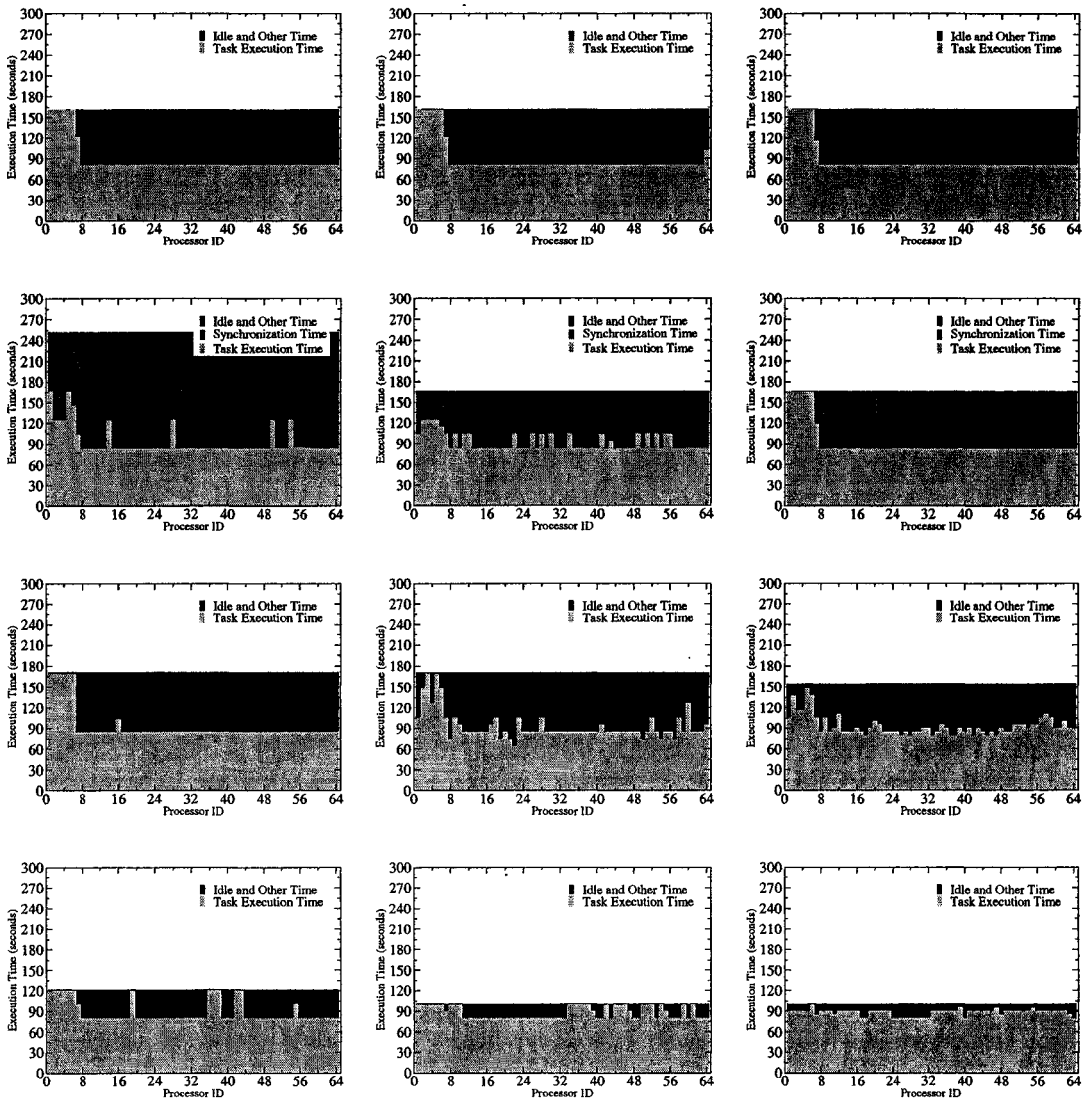
Overall benchmark run times on 64 processors for no load balancing (row 1), load balancing using the Metis repartitioning tool (row 2), load balancing using a Charm++ iterative balancer (row 3), and load balancing using PREMA's Diffusion Scheduler (row 4) with 4 tasks (column 1), 8 tasks (column 2), and 16 tasks (column 3) initially assigned to each processor. 25% of the tasks are rated as "heavy".

Figure 7.2 repeats this same experiment with only 10% of the initial tasks rated as "heavy". While overall, the relative performance between load balancing methods is similar, it is worth noting the poor performance of the stop-and-repartition methods contained in row 2. Regardless of the task granularity, Metis is unable to effectively migrate tasks in order to improve runtime.

Figure 7.3 contains performance results for the benchmark executing on 64 processors with 25% of the tasks rated as "heavy" (the problem size has scaled with the number of processors). As on the smaller processor configurations, the synchronization points mandated by both the stop-and-repartition (row 2) and incremental load balancing schemes (row 3) are detrimental to overall performance. For coarse-grained task decomposition (column 1), PREMA provides a 25% improvement over no load balancing, a 50% improvement over stop-and-repartition balancing, and a 41% improvement over incremental load balancing. For a finer-grained decomposition (column 3), PREMA leads to a 31% improvement over no load balancing, a 23% improvement over repartitioning using Metis, and a 36% improvement over incremental load balancing using Charm++.

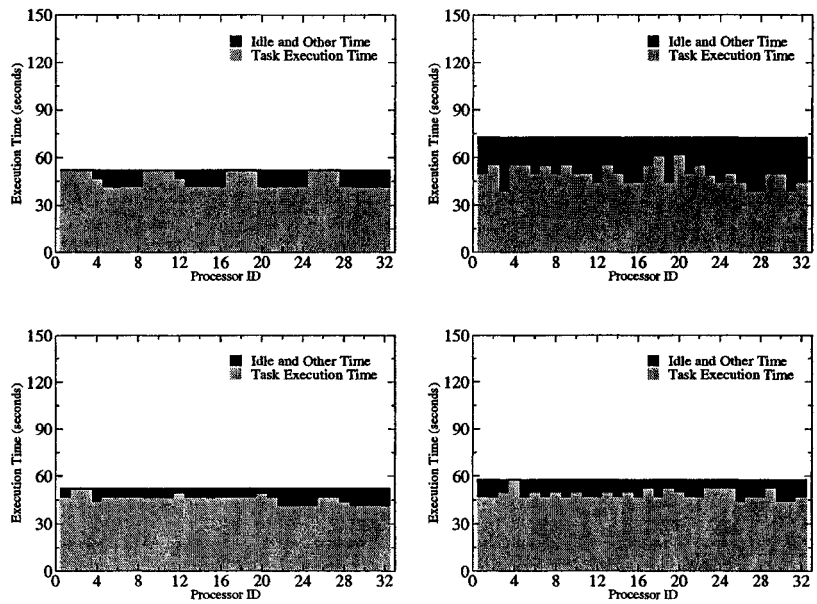
Figure 7.4 contains the performance results of this same experiment executed on 64 processors with the "heavy" tasks making up only 10% of the total population. Results in this case are very similar. For a coarse-grained task decomposition (column 1), PREMA provides a runtime improvement of 25% over no load balancing, 52% over repartitioning using Metis, and 28% over a loosely-synchronous load balancing method. For a finer-grained decomposition (column 3), PREMA provides an improvement of 38% over no load balancing, 39% over repartitioning using Metis, and 35% over iterative load balancing using Charm++.

Figure 7.4: BENCHMARK RUN TIMES ON 64 PROCESSORS; 10% HEAVY TASKS



Overall benchmark run times on 64 processors for no load balancing (row 1), load balancing using the Metis repartitioning tool (row 2), load balancing using a Charm++ iterative balancer (row 3), and load balancing using PREMA's Diffusion Scheduler (row 4) with 4 tasks (column 1), 8 tasks (column 2), and 16 tasks (column 3) initially assigned to each processor. 10% of the tasks are rated as "heavy".

Figure 7.5: PREMA VS. SEED-BASED LOAD BALANCERS; 32 PROCESSORS



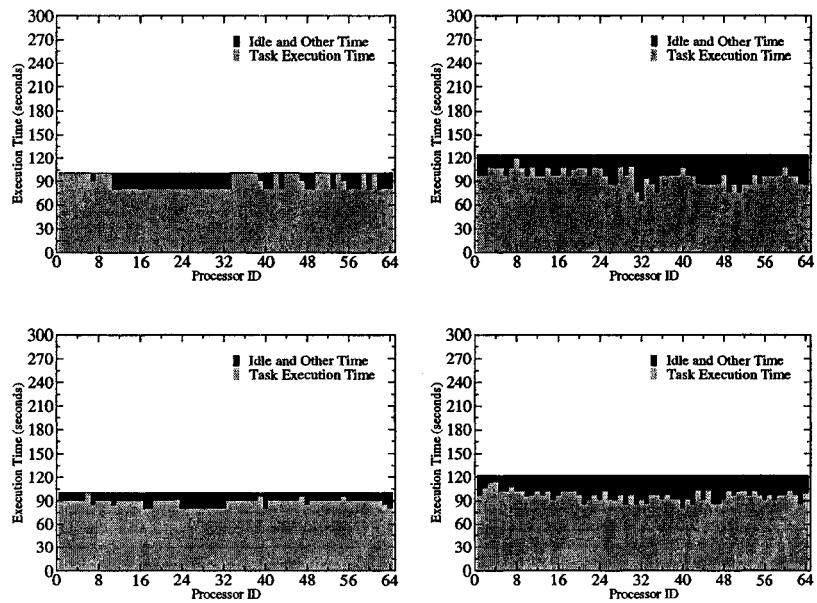
Overall benchmark run times for PREMA (column 1) and seed-based load balancers implemented by Charm++ (column 2) on 32 processors. In row 1, 8 tasks are allocated to each processor, while in row 2, 16 tasks are initially allocated to each node.

7.1.2.2 Implementation Efficiency

We use the same benchmark program to evaluate the efficiency of PREMA's implementation, with an initial imbalance of 25%. However, in this case we use the *seeded* load balancers provided as part of the Charm++ distribution as a means for comparison. Like PREMA, Charm++'s seeded balancers are asynchronous, eliminating the global synchronization points that plagued the iterative balancers.

Figure 7.5 contains the results of our experiments, executed on 32 processors. Comparing PREMA (row 1) and Charm++ (row 2) for 16 finer-grained tasks initially allocated to each processor (column 1) indicates that PREMA offers a performance advantage of roughly 9%. However, as the task granularity increases, the performance advantage of

Figure 7.6: PREMA VS. SEED-BASED LOAD BALANCERS; 64 PROCESSORS



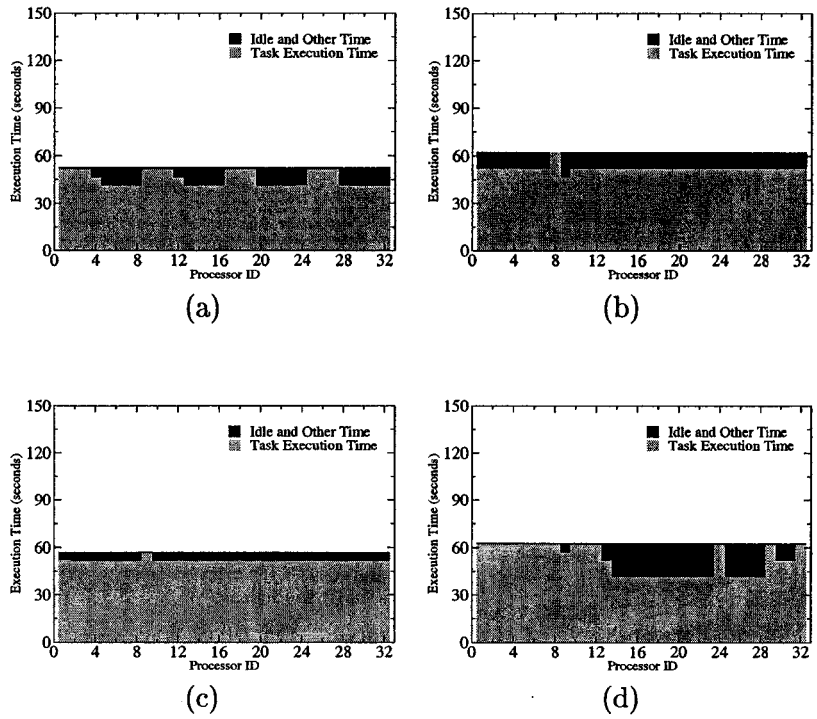
Overall benchmark run times for PREMA (column 1) and seed-based load balancers implemented by Charm++ (column 2) on 64 processors. In row 1, 8 tasks are allocated to each processor, while in row 2, 16 tasks are initially allocated to each node.

PREMA becomes more evident. With the application decomposed into half the number of tasks (8 tasks initially allocated to each processor), the runtime improvement afforded by PREMA increases to 27% (Figure 7.5, column 2). As the number of processors grows to 64 (Figure 7.6), PREMA offers roughly an 18% performance advantage in both cases.

7.1.2.3 Framework Flexibility

The final component of this set of experiments is to ascertain the efficiency of the PREMA load balancing framework itself. We can do this by examining a variety of Scheduling Module implementations; a truly useful framework will allow for the efficient implementation of multiple Schedulers, not just a single type. We once again make use of the benchmark

Figure 7.7: PREMA SCHEDULER ANALYSIS; 32 NODES, 8 TASKS PER NODE

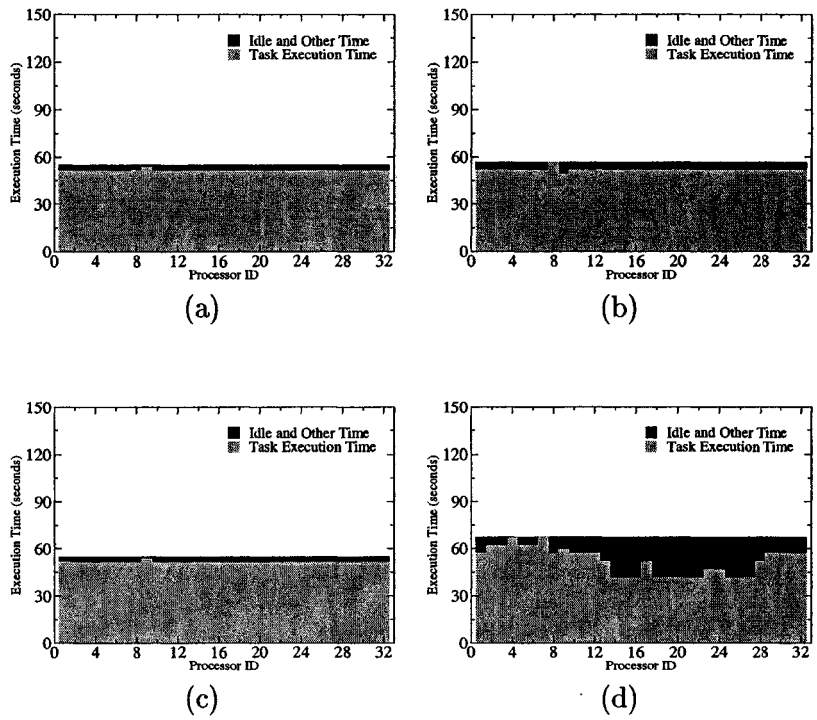


Benchmark run times for PREMA's Diffusion (a), Work-stealing (b), Multi-list (c), and Gradient (d) Schedulers on 32 processors with 25% initial imbalance and 8 tasks initially allocated to each processor.

program that we have described previously. In addition, we will use four Scheduler types that we have covered in this thesis, and which are well-known in the dynamic scheduling literature: Diffusion, Work-stealing, Multi-list, and Gradient.

In Figure 7.7, we present the results of our experiments on 32 processors in which 8 tasks are initially allocated to each processor. The load balancing method which leads to the lowest total execution time is the Diffusion method, although this is not the method which results in the highest quality final load distribution. In this case, the tradeoff between the increased complexity of a single load balancing iteration and the reduction in the number of iterations necessary to find work results in a gain in overall performance. In this case,

Figure 7.8: PREMA SCHEDULER ANALYSIS; 32 NODES, 16 TASKS PER NODE

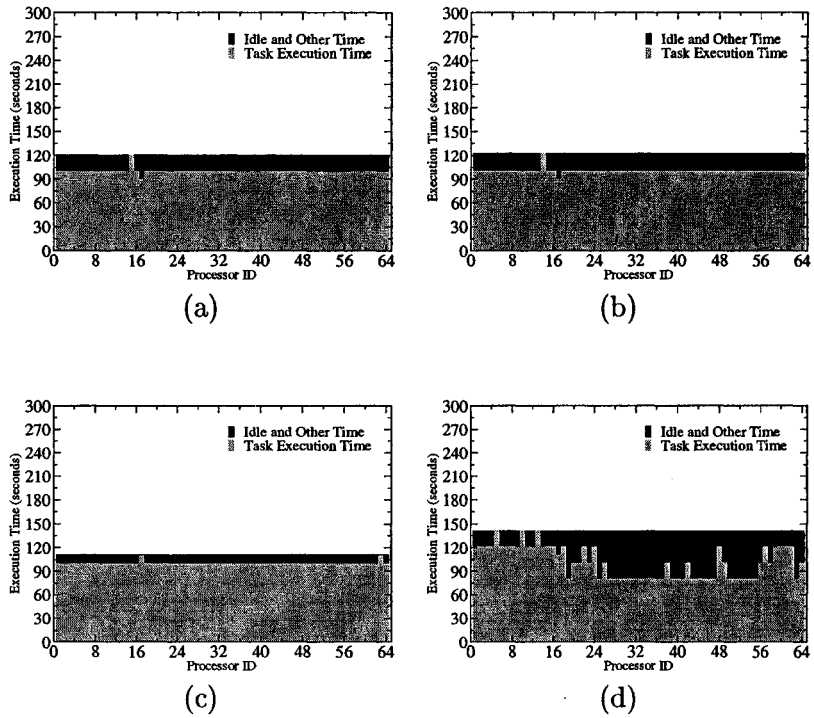


Benchmark run times for PREMA's Diffusion (a), Work-stealing (b), Multi-list (c), and Gradient (d) Schedulers on 32 processors with 25% initial imbalance and 16 tasks initially allocated to each processor.

the PREMA Diffusion Scheduler results in a performance improvement of 37% over no load balancing (shown in Figure 7.1, row 1, center), while gains of 24%, 32%, and 24% are achieved with the Work-stealing, Multi-list, and Gradient Schedulers, respectively.

Similar results can be seen both with a finer task granularity (Figure 7.8), as well as when the number of processors increases (Figure 7.9 and Figure 7.10). This indicates that a variety of Scheduler implementations result in a high quality of load distribution and a low overhead. We are therefore able to comfortably state that the PREMA load balancing framework is not restrictive in the types of scheduling policies it allows, freeing developers to experiment with load balancing algorithms and implementations.

Figure 7.9: PREMA SCHEDULER ANALYSIS; 64 NODES, 8 TASKS PER NODE



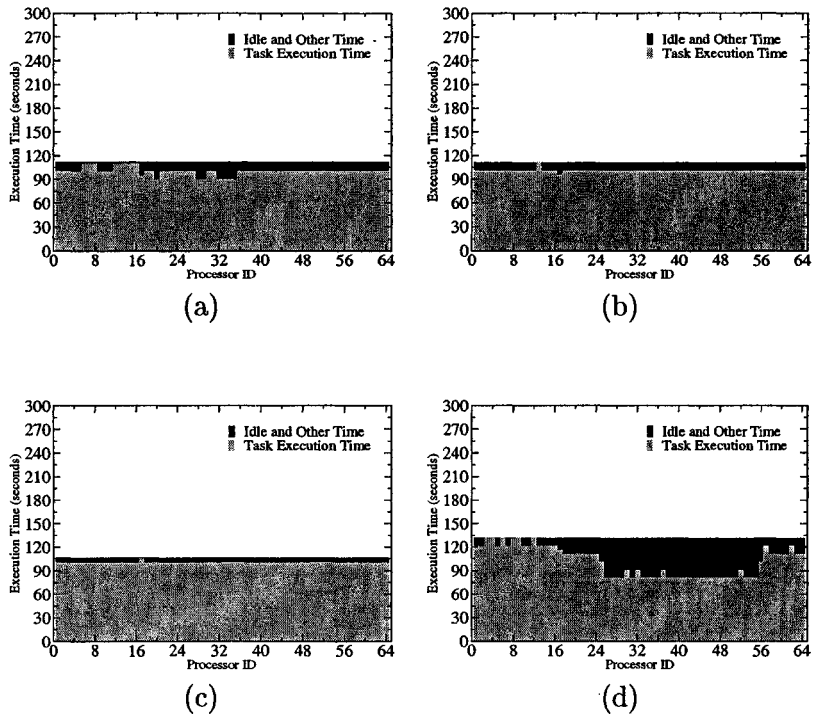
Benchmark run times for PREMA's Diffusion (a), Work-stealing (b), Multi-list (c), and Gradient (d) Schedulers on 64 processors with 25% initial imbalance and 8 tasks initially allocated to each processor.

7.2 Parallel Advancing Front Mesh Refinement

Parallel mesh generation and refinement is an important adaptive application and a good candidate to demonstrate the effectiveness of PREMA's load balancing infrastructure. To this end, we have implemented a 3D Parallel Advancing Front Technique (PAFT) [137] mesh refinement program, which is presented in [48].

A critical point that is worth mentioning is that the software used to perform the sequential mesh refinement is provided by a third party [123]; the PREMA framework allows us to use this software without modification and still achieve excellent speedup and scalability.

Figure 7.10: PREMA SCHEDULER ANALYSIS; 64 NODES, 16 TASKS PER NODE



Benchmark run times for PREMA's Diffusion (a), Work-stealing (b), Multi-list (c), and Gradient (d) Schedulers on 64 processors with 25% initial imbalance and 16 tasks initially allocated to each processor.

A crucial element to this success is the preemptive decision making capability incorporated into PREMA, and we will subsequently demonstrate its importance in achieving acceptable parallel performance.

7.2.1 Parallel Advancing Front Skeleton

Three dimensional *advancing front* mesh refinement methods [114, 115, 116, 9, 118] begin by dividing a coarse mesh into triangular faces, which form the initial *front*. Tetrahedra are then generated starting at these boundary faces, and work toward the center of the region being meshed. The inner surface of these elements collectively form the *advancing*

front [144].

The domain decomposition approach we employ was initially proposed by Lohner in [116], and extended for handling adaptivity and load balancing in [137]. However, we do not use the concept of interior and interface regions to uncouple the sub-domains as in [116]. Moreover, instead of the master/worker model used in [137], we implement a decentralized scheduling of parallel computation.

The key steps in the parallel advancing front method are:

1. Decompose the initial volume grid into N ($N \gg P$, where P is the number of available processors) sub-domains (i.e., apply *over-decomposition*).

2. Generate the dual graph of the subdivision and partition it into P sets of sub-domains.

3. Find the representation of each sub-domain as a set of triangular faces and orient the faces.

4. Load each set of sub-domains in parallel into the corresponding processor and create a Schedulable Object (Chapter 5, Section 5.4.1) for each of the sub-domains.

5. Apply mesh generation subroutine [123] on each processor for every local Schedulable Object (i.e., sub-domain), while executing a dynamic load balancing algorithm in between, so that some of the Schedulable Objects are moved from one processor to another in the case of imbalance due to different levels of refinement in the geometry.

6. Glue the adjacent sub-domains on each processor in parallel.

At the end of the execution of the above steps, the mesh is ready for parallel finite element analysis. The PAFT represents one of the simpler application types which are still consid-

ered interesting; once Schedulable Objects have been created, no further communication is necessary until time to glue the global mesh together.

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 to the PREMA for *implicit* load balancing. Explicit work stealing begins with the PAFT mesher maintaining a queue of local sub-meshes pending refinement. During refinement, each processor performs three steps: local region refinement, load balancing, and polling the network. After refining local regions, a processor determines whether or not its pending work load has fallen below a 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 [138]. As with the other load balancing methods under consideration, load balancing begins when a processor's workload falls below a predetermined threshold[2]. 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 of work units in order to restore load balance.

Implicit load balancing places the decision making and data migration burden on the runtime system. The pre-defined scheduling algorithms implemented by PREMA have been described previously. After the initial distribution of sub-meshes and creation of

[2]The thresholds used for all load balancing experiments in this Section are identical.

Schedulable Objects, a single message is sent to each region invoking the mesh refinement stage of the algorithm. Each processor then polls for incoming messages; polling will invoke the refinement computation as well as the migration of Schedulable Objects when necessary.

7.2.2 Effects of Domain Decomposition

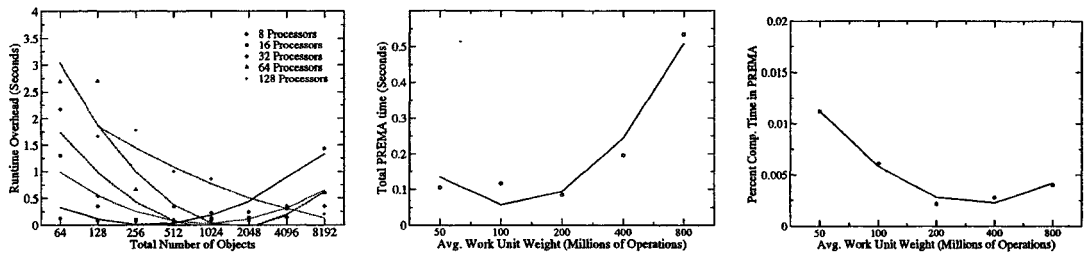
We begin with an examination of the effects of over-decomposition on application performance and on the overheads incurred by the runtime system. Three parameters play a role in this study: the number of work units created by the decomposition (N), the number of processors available (P), and the weights 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.

Figure 7.11(a) depicts the time spent inside the runtime system as both N and P vary, excluding the initialization and termination stages of the program[3]. 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, 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.

Figure 7.11(b) and Figure 7.11(c) show the results of varying the computational work-

[3]We have chosen to exclude these stages because they are highly application dependent.

Figure 7.11: EFFECTS OF OVERDECOMPOSITION ON PAFT PERFORMANCE



(a) (b) (c)

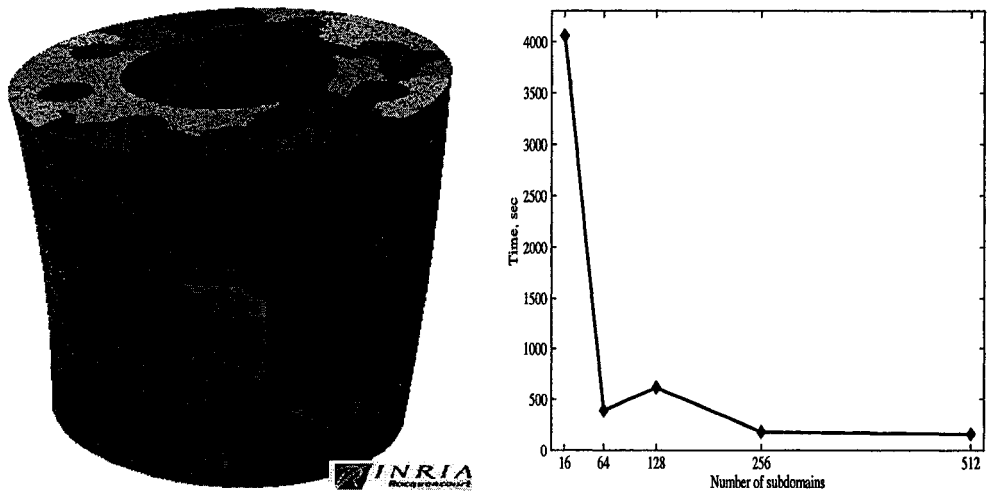
Effects of varying processor count (P), work unit count (N), and work unit weight on PREMA overheads: the runtime overhead in terms of seconds as the total number of work units increases on varying numbers of processors (a), the total time spent in the PREMA system as the average weight of each work unit varies, for 2048 work units (b), and the total time spent in the PREMA system as a percentage of computation time as the average weight of each work unit varies, for 2048 work units (c).

load 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[4].

Figure 7.12 depicts the the reduction in total execution time per processor for a pipe geometry (left), viewed by Medit from the Institut National de Recherche en Informatique en Automatique (INRIA) [59]. The reduction in total execution time is due to better memory utilization and nonlinear computational complexity of the sequential mesh generator [48]. As the number of sub-domains increases and the final mesh size remains fixed, the working set per processor is decreasing. The improvement in the utilization of virtual memory

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

Figure 7.12: EFFECTS OF OVERDECOMPOSITION ON PIPE MODEL



(a) (b)

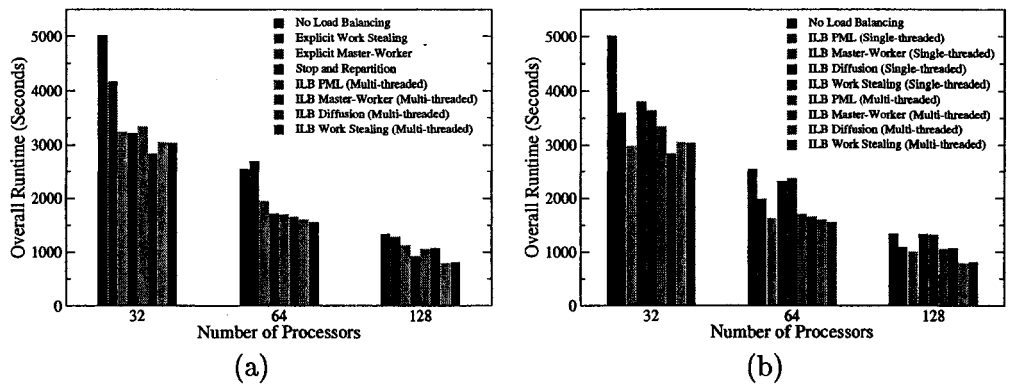
overcomes the slight overhead introduced by the runtime system per mobile object.

7.2.3 Experimental Results

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.

Figure 7.13(a) compares the overall execution times of the PAFT program using all load balancing methods (as well as no load balancing) on several processor configurations. The test platform on which we conduct our experiments consists of 333 MHz Ultra SPARC IIi machines, connected by Fast Ethernet and utilizing the LAM [111] implementation of the Message Passing Interface (MPI) [75]. On 32 processors, PREMA's Diffusion Scheduler module provides an improvement of 42% over no load balancing, 12% over load balancing

Figure 7.13: PERFORMANCE OF IMPLICIT AND EXPLICIT LOAD BALANCING



Runtime performance of explicit, implicit, and stop-and-repartitioning load balancing (left), as well as single-threaded and multi-threaded preemptive load balancing (right).

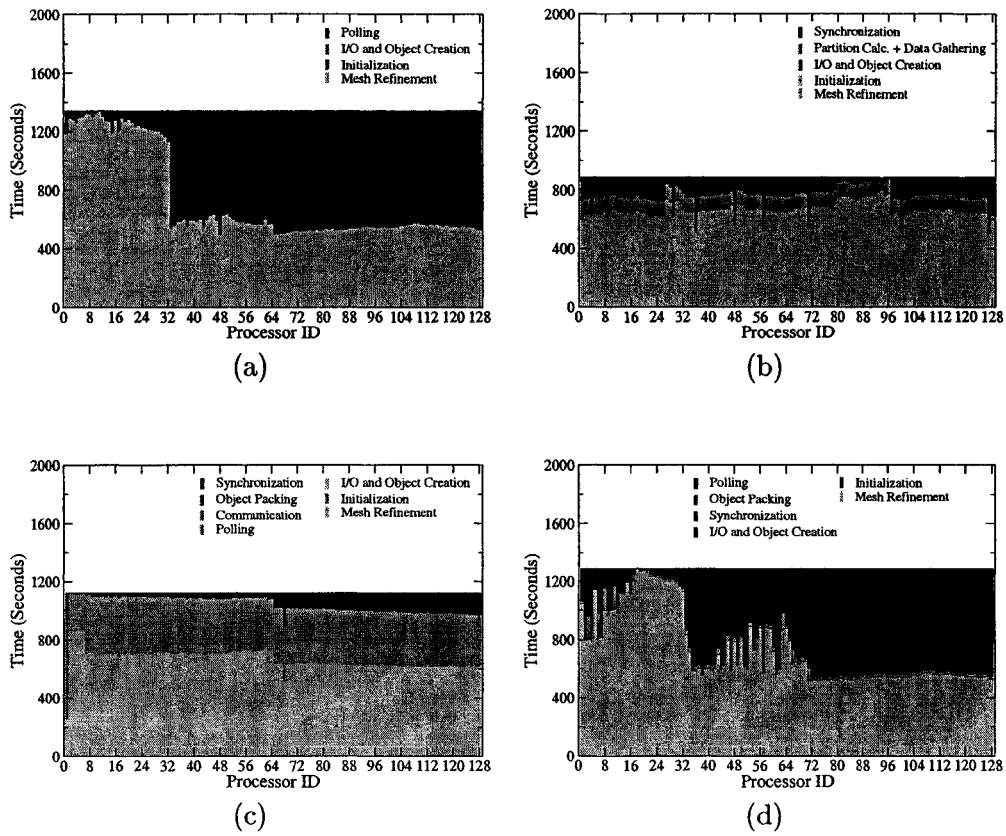
with stop-and-repartition methods, and roughly 13% over explicit load balancing. On 64 processors, these numbers are 39%, 9%, and 20%, while on 128 processors they are 42%, 15%, and 30%[5].

In Figure 7.13(b), we compare the results of implicit load balancing with and without multi-threaded preemption (the preemptive component of our runtime framework was described in Chapter 5, Section 5.4.3). Particularly in the cases of Work-stealing and Diffusion, preemptive load balancing decision making can provide a reduction in total run time of over 40%. 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 resulting from load balancing. In Figure 7.14(a), 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

[5]Note that the 128 node cluster used in these experiments is heterogeneous and made up of processors whose clock speeds range from 333 MHz to 650 MHz.

Figure 7.14: BREAKDOWN DATA FOR SEVERAL LOAD BALANCING METHODS



Breakdown data for no load balancing (a), stop-and-repartition load balancing (b), and two explicit load balancing methods: master/worker (c), and work-stealing (d) on 128 processors.

clustered within processors toward the "front" of the system (processors with IDs 0 through 31), leaving ample opportunity for effective load balancing. Figure 7.14(b) provides 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 which can be avoided (up to roughly 11% of the overall runtime).

The last row of Figure 7.14 shows the results of two explicit load balancing methods. Figure 7.14(c), the master/worker method, is the more successful of the two in terms of computation distribution quality, balancing the work load with a standard deviation of roughly 83. However, initialization costs impose a large penalty. 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.

Figure 7.14(d) contains the results of load balancing with the explicit application-managed work-stealing method. This method suffers due to the fact that polling cannot occur during the execution of a task, preventing messages containing load balancing information and requests from being processed in a timely manner. Consequently, underloaded nodes tend to spend a great deal of time idle. The ultimate result is that too few tasks migrate to alleviate workload imbalance, as evidenced by a mesh refinement standard deviation of 246.

Figure 7.15 contains processor-by-processor breakdowns for both preemptive and non-preemptive implicit load balancing methods implemented using PREMA's load balancing framework. Providing a preemption mechanism within the runtime system clearly provides a performance benefit; with the Diffusion method (row 2), overall execution time decreased by 41% compared to the non-preemptive counterpart. Similar results can be seen in the case of Work-stealing (row 1). In addition, workload distribution quality is increased, compared

Table 7.1: RUNTIME SYSTEM OVERHEADS ON 128 PROCESSORS

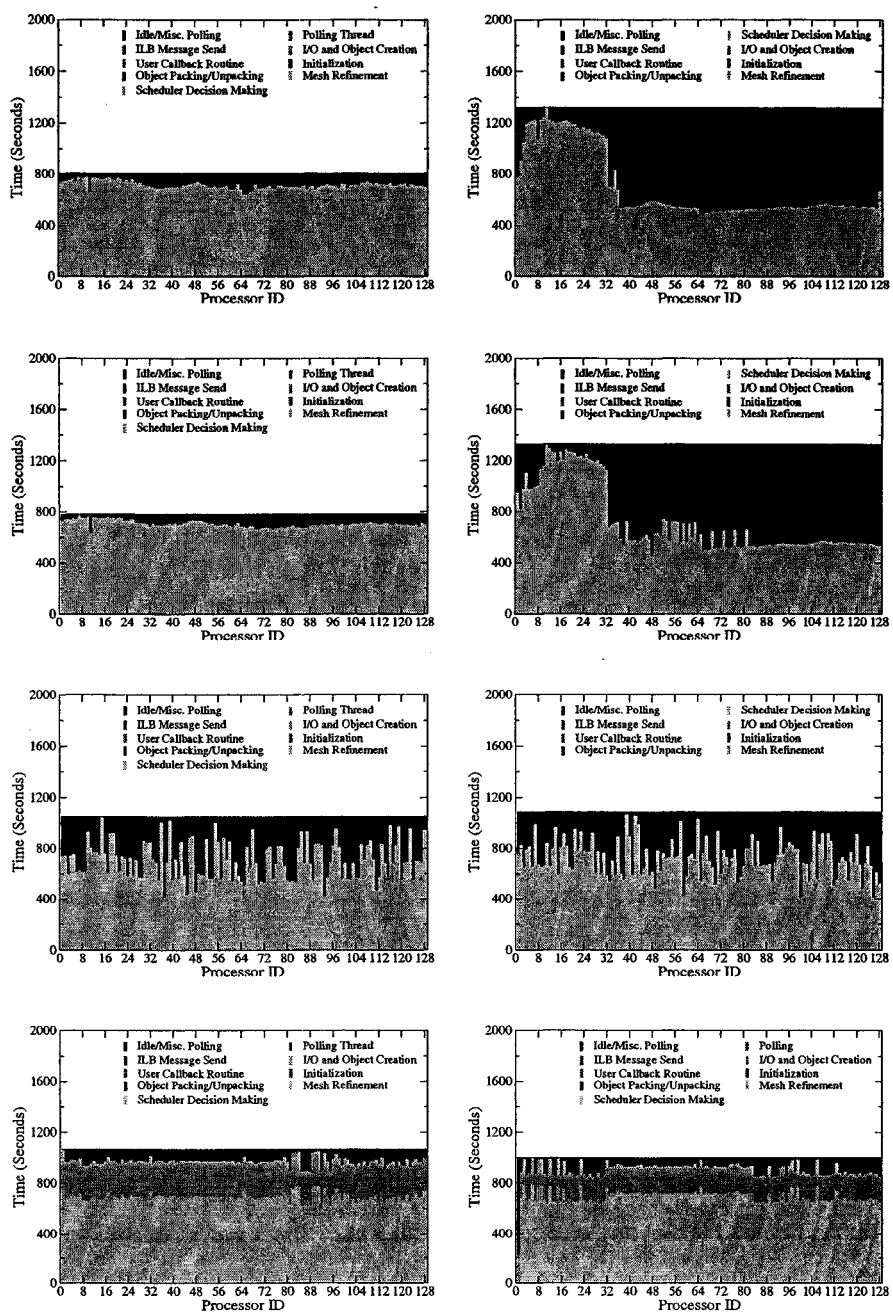
Load Balancing	Min/Max Refinement	PREMA Overhead				
		Poll Thr.	Msg. Send	Call-back	Sched.	Prcnt.
None	487/1332 sec	–	–	–	–	–
Stop & Repart.	499/786 sec	–	–	–	–	–
ILB W.S.	635/775 sec	0.1681 sec	0.0018 sec	0.0137 sec	0.0019 sec	0.023%
ILB Diffusion	632/765 sec	0.2080 sec	0.0026 sec	0.0110 sec	0.0670 sec	0.037%
ILB PML	411/1043 sec	0.9669 sec	0.0021 sec	0.0459 sec	1.0893 sec	0.200%
ILB M.W.	622/775 sec	0.5377 sec	0.0111 sec	0.4329 sec	0.000004 sec	0.092%

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

Note that less dramatic results are obtained with the Prioritized Multi-list (PML) and 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 subsequent performance degradation. In the case of the Master/Worker policy, workload is well-balanced in the non-preemptive case; the addition of a preemption mechanism does nothing to improve this. However, preemption does allow the Master processor to itself act as a Worker, which was not possible before.

Finally, we show that the overhead imposed by the runtime system is small and does not negatively impact overall application performance. Table 7.1 summarizes the overheads caused by PREMA according to several different categories. In all cases, overhead contributes significantly less than 1% to the overall runtime.

Figure 7.15: PREEMPTIVE VS. NON-PREEMPTIVE LOAD BALANCING



Breakdown data for preemptive (left) and non-preemptive (right) implicit Work-stealing (row 1), Diffusion (row 2), Prioritized Multi-list (row 3), and Master/Worker methods (row 4) on 128 processors.

7.3 Parallel Constrained Delaunay Triangulation

Our next series of experiments involves using the load balancing facilities provided by PREMA to balance the workload created by a 2D Parallel Constrained Delaunay Triangulation (PCDT) [49, 51] program. As we will demonstrate, this code presents a set of unique challenges for the runtime system that we have not seen until this point. In this Section, we will provide a brief introduction for the purposes of establishing how this problem is unique. We will then describe our experiments and present our performance results.

7.3.1 Parallel Constrained Delaunay Skeleton

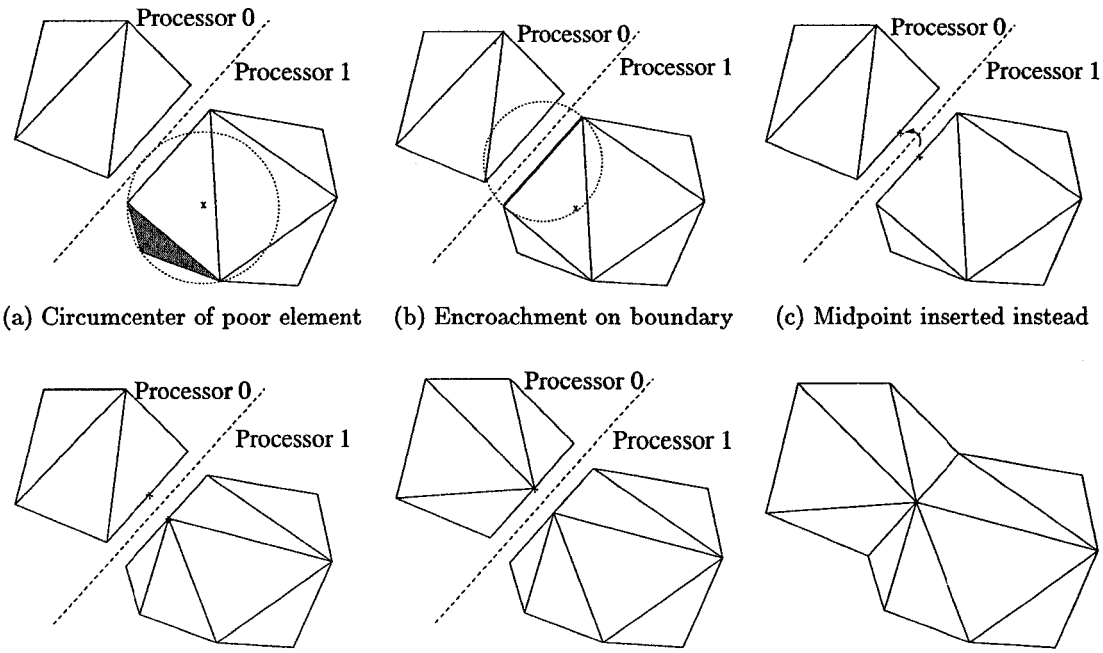
The *Delaunay Triangulation* D of a set of vertices V is defined as follows [62]:

- Any circle in the plane is said to be empty if it contains no vertex of V in its interior. However, vertices are permitted on the circle.
- An edge from vertex u to vertex v is in D if and only if there exists an empty circle that passes through u and v.

The Delaunay triangulation is very popular in the mesh generation and engineering communities due to the resulting element quality; the Delaunay triangulation of a vertex set maximizes the minimum angle among all possible triangulations [144].

The mesh *refinement* process begins with a coarse triangulation whose elements do not conform to specified restrictions on element quality. This coarse mesh is refined by adding new points and modifying the existing triangulation through strictly local operations, in a procedure that is often referred to as the Bowyer-Watson (BW) kernel [51]:

Figure 7.16: PARALLEL CONSTRAINED DELAUNAY TRIANGULATION



(a) Circumcenter of poor element (b) Encroachment on boundary (c) Midpoint inserted instead

(d) Cavity retriangulation (e) Cavity retriangulation (f) Consistent retriangulation

Point creation: A new point is inserted into the vertex set using an appropriate spatial distribution technique.

Point location: A triangle which contains this new point is identified and located.

Cavity computation: Existing triangles which interact with this new point in such a way as to violate the Delaunay property are removed.

Element creation: New triangles are built by properly connecting the newly inserted point with the old points, so that the resulting triangles are satisfactory.

However, in a parallel implementation of this kernel, two points cannot be inserted concurrently if their corresponding cavities overlap. We therefore introduce a set of *constrained* edges which define boundaries between sub-domains. The resulting triangulation is as close

as possible to a Delaunay triangulation, given that these constrained edges must be present in the final triangulation. The resulting modified BW kernel is shown in Figure 7.16.

Once a poor element is located (Figure 7.16(a)), it can be determined whether or not that element's *circumcircle* encroaches on a constrained boundary edge (Figure 7.16(b)). If so, we insert the midpoint of that constrained edge into the vertex set V (normally, we would insert the center of the circumcircle). However, because the constrained edge is shared by two processors, the remote processor must be notified of this edge split (Figure 7.16(d)) and the remote cavity must therefore be re-triangulated. As a result, the mesh is able to remain globally consistent (Figure 7.16(f)).

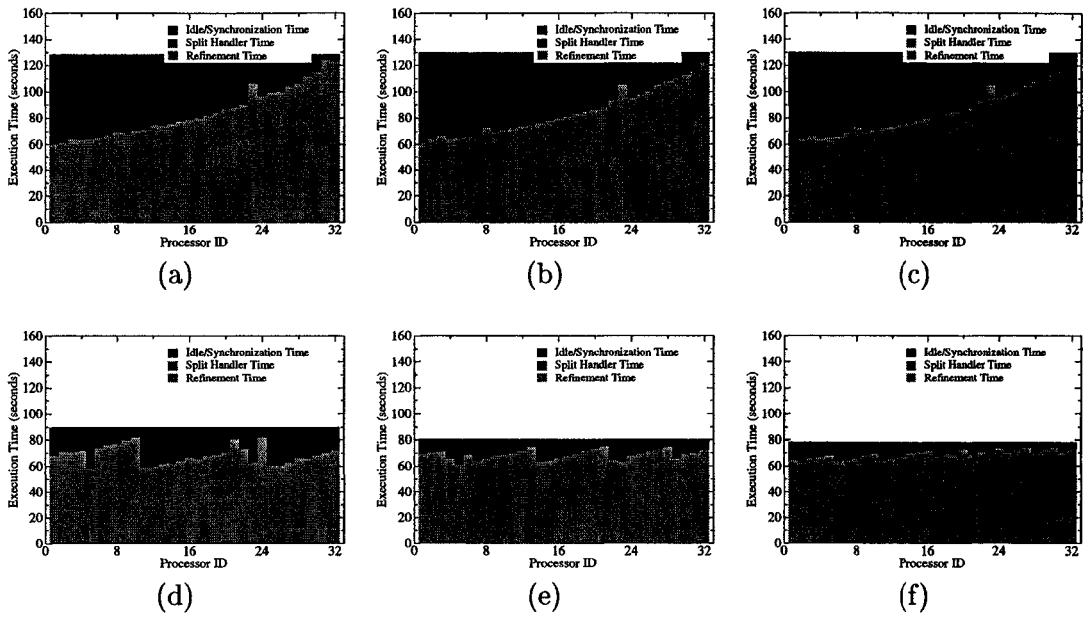
This application introduces a new component that was not present in our previous experiments, namely communication between tasks during the refinement process. In addition, 2D triangulation is not as compute intensive as the 3D tetrahedralization that we studied in the previous Section, allowing us to explore PREMA's impact on finer-grained computations.

7.3.2 Experimental Results on Homogenous Clusters

We begin with experiments performed on a homogeneous 32 processor cluster. We are able to achieve a load imbalance by varying the *area bound* across the sub-domains. Refinement will take place until the areas of the elements within the sub-domain are less than the specified area bound. This allows us to simulate a greater level of refinement for areas of interest within the geometry.

The top row of Figure 7.17 depicts the performance with no load balancing. These three graphs are nearly identical, indicating that the total computation performed by the

Figure 7.17: PCDT PERFORMANCE ON 32 PROCESSORS

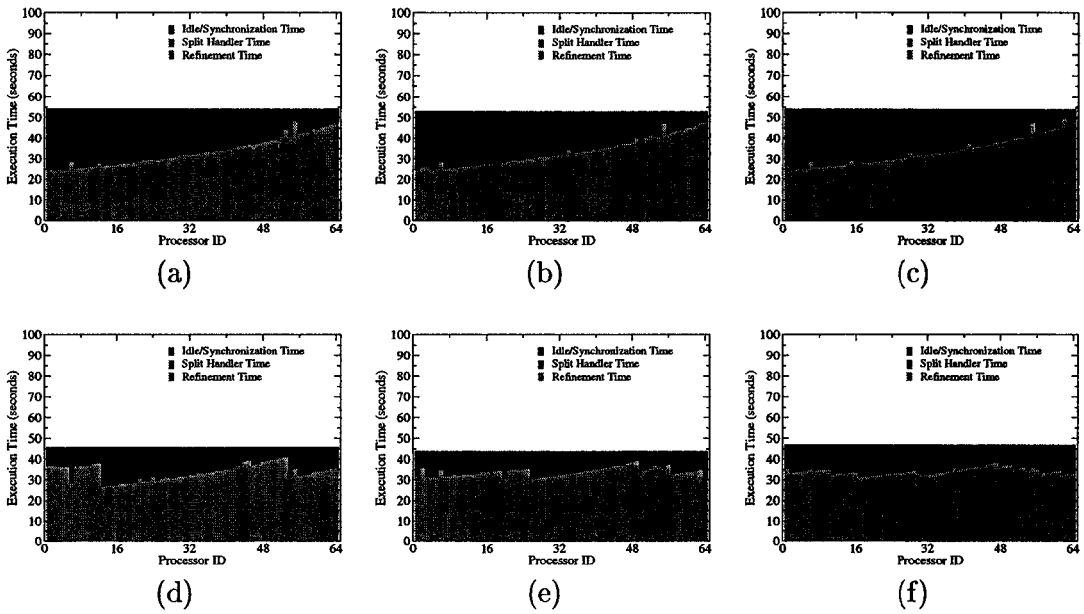


PCDT performance with no load balancing on 32 processors with either 4 (a), 8 (b), or 16 (c) sub-domains per processor, as well as with work-stealing load balancing with either 4 (d), 8 (e), or 16 (f) subdoamins per processor.

application is independent of the level of over-decomposition. Note, however, that this is not strictly true; increasing the number of sub-domains will increase the amount of message traffic necessary to communicate split information on boundary edges. However, the computation resulting from this traffic is quite small as compared to the refinement procedure.

The bottom row of Figure 7.17 contains performance data from the same experiments, but using PREMA's *Work-stealing* load balancing policy. With a fairly coarse domain decomposition (only 4 sub-domains initially allocated to each processor), PREMA is able to provide a 30% performance increase, reducing total execution time from 128 seconds to roughly 89 seconds. As the sub-domain granularity becomes finer and the load balancer has more freedom in terms of task migration, performance improvements measure as high

Figure 7.18: PCDT PERFORMANCE ON 64 PROCESSORS

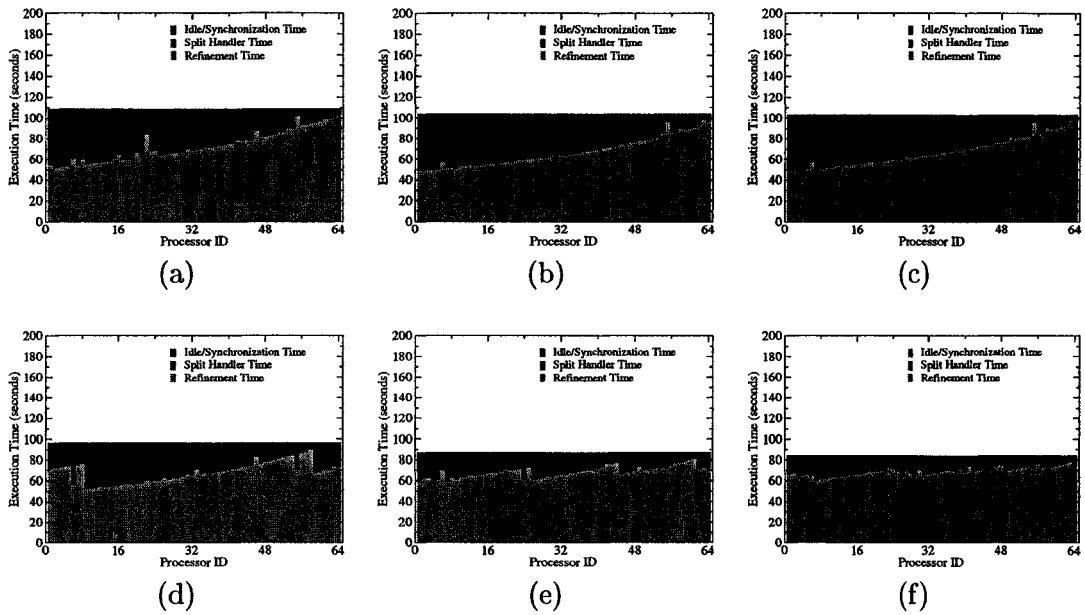


PCDT performance with no load balancing on 64 processors with either 4 (a), 8 (b), or 16 (c) sub-domains per processor, as well as with work-stealing load balancing with either 4 (d), 8 (e), or 16 (f) subdoamins per processor.

as 42% (Figure 7.17(c) vs. Figure 7.17(f)).

Our next set of experiments involved meshing the same sized problem on 64 processors. These results are shown in Figure 7.18. In this case, the overall performance improvements were not as dramatic, with a reduction in overall execution time of only roughly 15% in each case. We also notice a larger synchronization cost as the number of processors increases. This can be traced to the termination detection algorithm within the application itself, which employs a global reduction mechanism to ensure that all split messages have been received and processed. At issue is the possibility of split operations themselves generating more message traffic. Termination guarantees that any such oscillation has completed before the application is allowed to exit. Experimentally, we have shown that up to 4 or 5 such cycles are possible.

Figure 7.19: SCALED PCDT PERFORMANCE ON 64 PROCESSORS

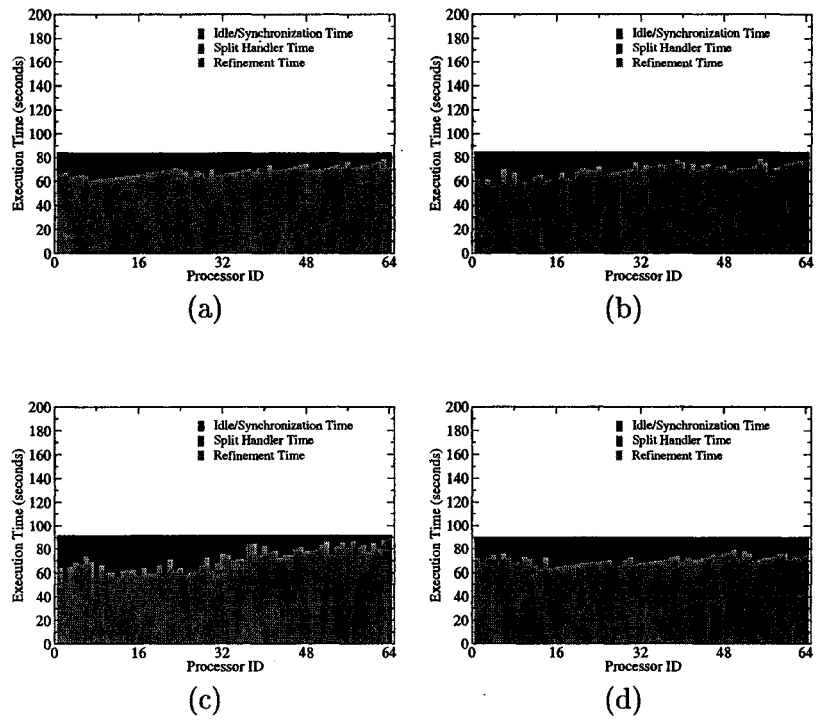


PCDT performance for a scaled problem size with no load balancing on 64 processors with either 4 (a), 8 (b), or 16 (c) sub-domains per processor, as well as with work-stealing load balancing with either 4 (d), 8 (e), or 16 (f) subdoamins per processor.

We were then lead to wonder if the problem size would affect the synchronization overhead. To determine this, we scaled the problem size so that the number of elements in the final mesh is twice that of our earlier experiments. These results are shown in Figure 7.19. The synchronization overhead is shown to be independent of the problem size, and scales only with the number of processors. Moreover, the maximum performance increase due to dynamic load balancing grows to roughly 20%.

In Figure 7.20, we present performance data for four scheduling policies implemented with the PREMA runtime framework. The problem size is the same as is shown in Figure 7.19, in which the number of mesh elements has scaled with the number of processors. The total number of sub-domains in this case is 1024, or a factor of 16 times the number of available processors. In each case, we can see that PREMA is successful in improving

Figure 7.20: PCDT PERFORMANCE WITH SEVERAL PREMA SCHEDULERS



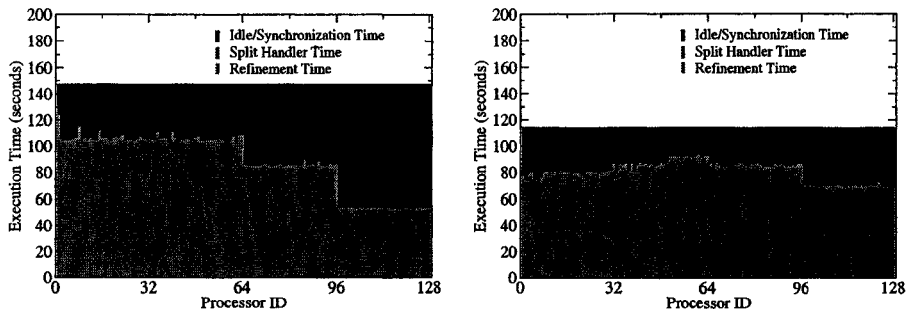
PCDT performance for a scaled problem size with work-stealing (a), diffusion (b), gradient (c), and multilist (d) load balancing on 64 processors with 16 sub-domains initially allocated to each node.

the total execution time versus runs in which dynamic load balancing is not utilized. Runtime decreases by 20% with the Work-stealing scheduler, 18% with the Diffusion scheduler, 12% with the Gradient scheduler, and 14% with the Multilist scheduler. These figures reinforce our assertion that the PREMA framework is flexible enough to allow for efficient implementations of a wide variety of scheduler types.

7.3.3 Experimental Results on Heterogeneous Clusters

As a final experiment with the PCDT program, we expanded the execution environment to include 128 heterogeneous processors. Specifically, the runtime environment was made

Figure 7.21: PCDT PERFORMANCE ON HETEROGENEOUS CLUSTER



PCDT performance on a 128 processor heterogeneous cluster both without (left) and with (right) dynamic load balancing.

up of 64 333 MHz UltraSPARC IIi processors equipped with 256 MB of physical memory, 32 360 MHz UltraSPARC II processors equipped with 256 MB of physical memory, and 32 650 MHz UltraSPARC IIe processors equipped with 1 GB of physical memory. All nodes were connected via a 100 Mb fast ethernet network. We then refined a new mesh where the imbalance is caused by *systemic differences only.* In other words, no load imbalance was incurred by the application itself; each sub-domain was refined to an equal extent.

Our results are shown in Figure 7.21. On the left, we can clearly see that the varying capabilities of the computing hardware leads to a significant overall imbalance. In fact, the time required for refinement varies among the nodes by roughly a factor of two, which corresponds directly to the differences in processor clock speeds. However, with load balancing managed by the PREMA runtime system, the refinement is spread much more evenly among the available nodes, which leads to a reduction in overall run time by 23%.

This result is significant, in that it demonstrates the effectiveness of the PREMA system in situations in which application-perceived load imbalance is caused by outside factors, such as other users in a shared environment, or, as in this case, a heterogeneous environment.

As Cluster of Workstations (COW) parallel environments are becoming more and more prevalent due to their cost-effectiveness, software such as PREMA can lead to reduced application run times and therefore better resource utilization.

7.4 Fast Multi-pole N-body Simulation

The goals for this phase of our experimental analysis are three-fold. First, we want to establish the effectiveness of the PREMA programming model and runtime system implementation when used in conjunction with non-meshing codes. As mesh generation is the application which has driven our development work and is of the most interest to our research group, it is only natural that we have concentrated our efforts in that direction. However, the PREMA system is designed to be of use to a wide variety of computational scientists. It is therefore important to determine its usefulness in a wider context.

Our second goal is to determine the ease of integrating the PREMA system with previously existing codes or codes that have been developed outside of our local research group. The codes we have previously discussed were developed in a symbiotic environment in which the application developer had ready access to the runtime system designer and implementor. It is important to determine weather or not this environment had any impact on our perceptions toward the runtime system. In other words, did the proximity between application and system developers bias our view that it is easy to integrate scientific codes with PREMA?

Finally, our third criterion is to compare the efficiency of PREMA's load balancing mechanism with a previously created and tightly integrated load balancing strategy that is

provably optimal for a given initial cell distribution [12, 74]. The PREMA approach cleanly separates the application domain from load balancing, freeing the application developer from worrying about load balancing details. However, this convenience is for naught if it carries a stiff performance penalty. These sets of experiments will allow us to evaluate this tradeoff.

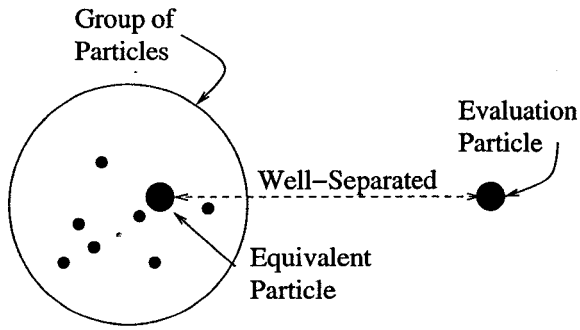
We begin with a brief discussion of the N-body problem itself, along with an overview of the Fast Multi-pole Algorithm. We will then examine the initial load balancing strategy which is bundled with the application. From this, we will devise a method for integrating the application with PREMA, and conclude with our experimental results.

7.4.1 N-body and Fast Multi-pole Algorithm Background

N-body simulations are useful in many areas of science, such as astrophysics, fluid mechanics, and molecular dynamics. In general, the problem considers the interaction of N particles or bodies and computes the forces they exert on one another, calculating how these forces influence their respective locations over some number of time-steps. Once the new positions are calculated, the process can move to the next time-step.

The simplest N-body algorithm computes all pairwise interactions. The disadvantage of such an approach is the resulting $O(N^2)$ complexity per time-step. In order to make large simulations computationally tractable, more efficient algorithms have been developed, such as continuum and hierarchical methods [4, 5, 18, 82]. Hierarchical N-body methods use trees (quad-trees in two dimensions, and oct-trees in three dimensions) to decompose the physical domain space into units known as "cells", and thereby can reduce the time complexity to either $O(N \log N)$ [18] or $O(N)$ [5, 82] per time-step. This can be achieved by computing

Figure 7.22: APPROXIMATING INTERACTIONS BETWEEN DISTANT PARTICLES



the interaction forces exactly only for those particles that are determined to be "near" the particle in question; forces contributed by more distant particles are approximated.

The *Fast Multi-pole Algorithm* (FMA) computes the potential forces among N particles in a time proportional to N. The leaf-level nodes of the corresponding domain decomposition tree contain particles, while coarser levels contain the field effects of the particles found in the rooted subtree. The algorithm makes two passes over the tree. During the upward pass, the summary of field effects of particles in the subtrees are propagated up the tree, while during the downward pass the local expansions and direct particle interactions are computed.

In order to reduce the computational effort associated with computing direct particle-to-particle interactions, the FMA clusters particles and computes the interactions between clusters that are sufficiently far away[6] (Figure 7.22) using *multi-pole expansions*, which represent the potential field effect of a number of particles as an infinite series. Any level of desired accuracy can be obtained by truncating the infinite series after a sufficient number of terms.

[6]Although there are precise definitions for "far away" and "well-separated", for the purposes of this thesis the intuitive definitions will suffice. Interested readers may consult the previously cited works.

The FMA has also been shown to be fairly well suited for parallelization, and, with appropriate partitioning and scheduling, can execute with very little interprocessor communication [83, 112, 142, 146].

7.4.2 Available Parallelism in the FMA

The unit of parallel granularity in the FMA is the *cell*, and there are three types of cell dependencies: nearest neighbors, parent/child relationships, and interaction lists (the interaction list for a cell x at some level is the set of cells that are children of the nearest neighbors of x and that are well-separated from cell x). Compared to other hierarchical N-body algorithms, such as the Barnes-Hut algorithm [18] which requires distant data access for every particle, the FMA data access is relatively local.

Greengard [83] has shown that some phases of the FMA, such as the multi-pole expansion computations at the leaf-level in the upward pass and the local expansion and the sum of the direct far-field computations at the leaf-level of the downward pass, are well-suited for parallelization and can execute with a complexity of $O(\frac{N}{P})$. Other phases may involve interprocessor communication, such as when parent and child nodes are not assigned to the same processor, or during the computation of interaction lists which involves cells located on different processors. In addition, direct interaction computations with particles in nearest neighbor cells may involve interprocessor communication in the cases in which the computation crosses processor boundaries. Finally, barrier synchronization is needed between the upward and downward passes.

The FMA can, however, lead to load imbalance that can only be corrected dynamically. There are two principle contributing factors. The first is due to unequal particle distribution;

i.e., an unequal number of particles in each cell. This will increase the number of direct particle interactions that must be calculated for certain regions in the domain space. This problem may also be exacerbated by the movement of particles as the simulation progresses. Another factor leading to runtime imbalance is the difference in computation required for particles near the domain boundary versus interior particles (interior particles tend to have more direct particle interactions). Due to these factors (and to systemic variances, which can also have a part in runtime load imbalance), a dynamic load balancing strategy must be employed in order to achieve maximal efficiency.

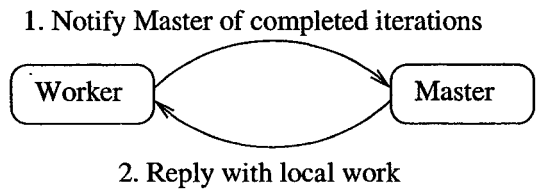
7.4.3 Explicit and Integrated Master/Worker Load Balancing

As we have mentioned, the parallel FMA N-body simulation code we are using for our experiments came bundled with a tightly-integrated, explicit load balancing mechanism based on a modification of the Master/Worker load balancing method we have previously described. We will now provide an overview of this method, as it will be subsequently used in our experimental analysis. We will begin our discussion assuming that the application has broken the data domain into a number of "cells". Cells contain particles, and interact with some subset of other cells (called an *Interaction List*), which are specified using a MAC (Multi-pole Acceptability Criterion) macro.

The N cells are initially distributed to the P available processors in a static manner; each processor is allocated $\frac{N}{P}$ cells. Each processor then partitions its local cells into *batches*, according to the load balancing method used. For the purposes of this load balancing discussion, the batch creation scheme is irrelevant. However, the relevant methods include:

- **Static-Sized Chunking**: Static chunking allocates cells to processors in fixed-size

Figure 7.23: MASTER-WORKER INTERACTION: CASE 1

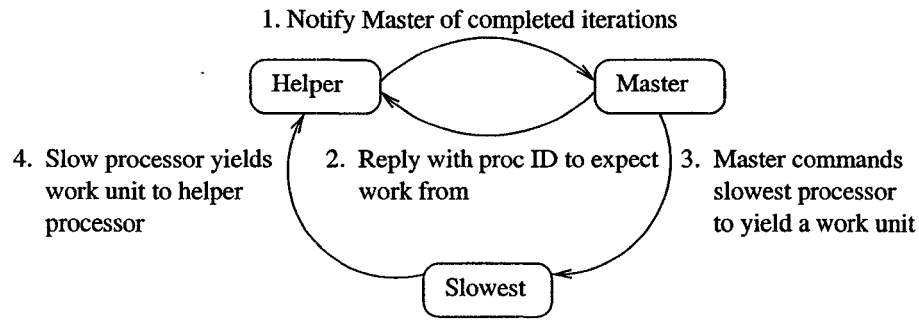


batches. When the size of each batch is a single cell, this is referred to as *Guided Self-Scheduling.* The size of the batches represents a trade-off between load balancing and overhead; for small batches, load balancing is often good but overhead is often high. The high overhead is attributable to the number of scheduling events that must take place. At the other extreme, scheduling cells in batches of size $\frac{N}{P}$ leads to very low overhead, but will often result in load imbalance.

- **Dynamic-Sized Chunking (Factoring)**: The factoring method of batch creation aims to reduce the number of scheduling events while still maintaining satisfactory load balance. To do this, batches are dynamically sized, with large batches being scheduled early and small batches scheduled later in order to smooth out the load imbalance. It has been shown that a good size for the batches is to contain half of the remaining work in each batch; therefore each batch is half the size of the previously scheduled batch, up to some minimum batch size.

Once the cells are partitioned among the processors, and each processor has partitioned its local cells into batches, the batches must be scheduled for execution. One processor is designated as the *Master* processor and is responsible for maintaining accurate counts of how many batches and cells have been executed by each processor. Each time a processor wishes to execute some work, it must communicate with the Master processor requesting

Figure 7.24: MASTER-WORKER INTERACTION: CASE 2



pending work to be done. This request will contain the number of cells that have been completed, so that the Master may update its counters and thereby keep an accurate and up-to-date picture of the system. For instance, at the beginning of program execution, each worker processor will send a message to the Master stating that it has completed no work. The Master may then reply in one of the following ways.

- **Case 1: Worker still has local work**: Because the Master is aware of how cells have been allocated at the beginning of the program, and maintains an up-to-date counter specifying how much work each processor has completed, it is able to know whether or not the worker processor still has local work remaining. In this case, the Master will reply that the worker node should schedule the next local batch for execution. This interaction looks like the communication pattern contained in Figure 7.23.

- **Case 2: Worker has run out of local work, but work still remains in the system**: One consequence of notifying the Master processor each time a batch is scheduled is that the Master is aware how much work remains on each processor and is therefore able to determine which processor is the slowest (has the most work remaining). This slow processor is also the one that would benefit the most from

off-loading some of its remaining work. In this case, four messages are required:

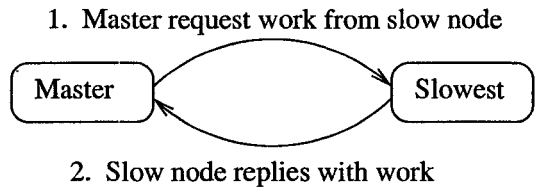
1. Message from the worker processor to the Master notifying it that it has finished a batch and requests more work to do. However, there is no local work remaining. This worker processor we will call the Helper.
2. Message from the Master back to the Helper informing it that there is no local work available. Work will therefore be arriving from the slowest processor; the Helper processor therefore knows from where work will be arriving and how to post the *receive* operation to receive it.
3. Message from the Master to the Slow processor instructing it to yield the next batch and send it to the Helper processor.
4. Message from the slow node to the Helper node containing the batch of work.

This interaction follows the communication pattern shown in Figure 7.24.

- **Case 3: The Master itself has run out of work, but work still remains in the system**: The Master processor is initially allocated cells to execute; once these cells are finished the Master tries to pull work to itself for execution. Again, because the Master keeps track of the amount of work executed by each processor, it knows which processor has the greatest amount of work remaining and selects this processor as the load balancing processor. The Master requests a batch of cells from this slow node, and the node replies with cells for execution. The communication pattern is illustrated in Figure 7.25.

- **Case 4: Worker has run out of local work and no work remains in the**

Figure 7.25: MASTER-WORKER INTERACTION: CASE 3



system: In this case, the Master replies with an *exit* message, signalling the worker that its portion of the application is finished. Once the Master has sent $P - 1$ such exit messages, it may exit itself.

There are several reasons why such a load balancing scheme may not perform optimally. First is that, for each and every batch scheduled, synchronization is required between the worker processor and the Master processor. In the case of load balancing, in which cells must migrate to a helper processor, this synchronization may extend to three processors.

This problem is compounded by the fact that, without preemptive decision making, messages may arrive at a processor some period of time before they are processed. This is because messages may arrive (particularly at the Master) while the recipient is processing a work unit. The newly arrived message will therefore be delayed, and will result in idle cycles on the worker processor. In the case of load balancing (Case 2 above), this may happen twice: once when the worker sends the initial request to the Master, and again when the Master processor sends the yield command to the slowest node. While it is the case that the FMA code has been designed in such a way so that periodic polling operations are inserted into the work units, this is not the case in all applications (nor is it always possible). In addition, over-polling can become a problem as well. It is important to note that the burden of deciding the optimal polling frequency has been placed on the application

developer, forcing him or her to make a decision about which they may know very little.

By removing the synchronization with the Master processor and by providing preemptive decision making, we can reduce the idle time spent by the worker processors and improve load balancing performance.

7.4.4 Load Balancing using the PREMA Library

We had several guidelines to bear in mind as we adapted the existing N-body code to use the PREMA library. First, we wanted to impact the existing code as little as possible. The FMA can be quite complex, and it is an area in which we are not experts. We therefore did not want to undertake the task of modifying existing data structures or tinkering with the FMA itself. Second, we wanted to reuse as much existing code as possible. For instance, routines were already in place to pack and unpack data structures for migration during load balancing. Finally, we obviously wanted to remove all traces of the Master-Worker scheduling algorithm that was already in place and replace it with PREMA scheduling policies. This task was simplified by the fact that the original load balancing code was isolated into a module that was separate from the FMA itself.

Four things must be done in order to adapt the FMA N-body code to make use of the PREMA dynamic load balancing functionality:

1. **Create mobile Schedulable Objects.** Schedulable Objects are application-defined data objects which represent the decomposition of the overall data domain. Load balancing is achieved via the migration of Schedulable Objects and their associated computation.

2. **Invoke computation using messages to objects.** The PREMA runtime system binds computation to data in a message-driven manner. Messages sent to data objects invoke computation in the form of application-defined message handler routines. Messages bound to Schedulable Objects migrate with their targets in order to effect load balancing.

3. **Create callback routines invoked by the runtime system.** The runtime system must have a mechanism to asynchronously obtain information concerning the application's data objects and pending computation. This is achieved through the use of several callback functions provided by the application.

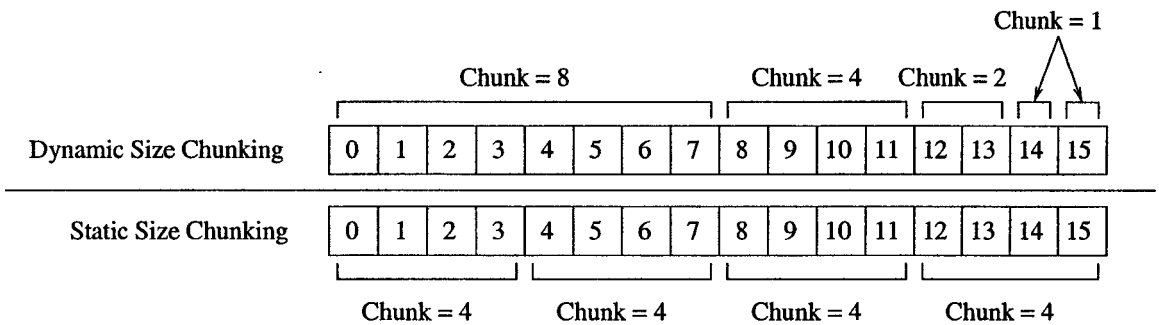
4. **Select a load balancing scheduling policy.** The PREMA system does not provide a single load balancing algorithm, but is instead a framework that allows load balancing scheduling policies to be quickly and easily substituted in a "plug-and-play" fashion. The application may select from one of the supplied policies, or may choose to implement a custom policy.

Below, we will cover each step in more detail and describe how each relates to the specific FMA N-body simulation code.

The N-body application code naturally partitions the global data domain into discrete units called "cells" which act as scheduling units. Therefore, cells make a natural candidate for Schedulable Objects. However, cells are typically scheduled in "chunks", which contain one or more cells. We have decided to make each chunk of cells a single Schedulable Object.

The size of each cell, the particles contained within each cell, and the cells to be contained within each chunk are determined at the beginning of a time-step. Equal numbers of cells

Figure 7.26: POSSIBLE CHUNKING SCHEMES FOR 16 CELLS



are distributed to each processor, and are then grouped into chunks according to either a Static-Sized or Dynamic-Sized chunking policy (Section 7.4.3 and Figure 7.26). Each chunk is registered with the PREMA system as a mobile Schedulable Object. From this point onward, it is the responsibility of the runtime system to migrate objects in order to restore load balance; the application no longer has any control over cell migration.

In order to minimize the impact to existing code, we did not modify the cell data structure in order to accommodate the mobile object concept. Instead, we created a "wrapper" data structure that defines a chunk. This data structure contains the cell IDs of the first and last cells in the chunk (the cells contained within a chunk always have contiguous cell IDs), and a PREMA mobile pointer pointing to itself. This is for convenience only and is not required by the runtime system. Finally, the chunk contains a flag which denotes which FMA work routine to execute; the original FMA code has one work routine for locally created work and a separate routine for chunks which have been migrated. In order to maximize code reuse we chose to stick with this mechanism. Our chunk data structures interface well with these existing computation routines, as each takes as parameters the first and last cell IDs for the cells whose forces are to be computed. Our chunk data structure is shown in Figure 7.27.

Figure 7.27: CHUNK DATA STRUCTURE WRAPPER

```
struct prema_chunk_t {
  int first_cell_id;            // ID of first cell in the chunk
  int last_cell_id;             // ID of last cell in the chunk
  mol_mobile_ptr_t self;        // PREMA mobile pointer to myself
  int work_flag;                // Should we execute FMA's work() routine
                                // or Others_Work() routine?
};
```

Once the application has created chunks and registered them with the runtime system as PREMA Mobile Objects, computation must be invoked on each chunk. We use a message-driven mechanism to bind computation to data; as data migrates during load balancing, computation is implicitly relocated smoothing out the computational workload. Messages are passed not only between processors, but from processors to application-defined data objects (mobile objects); it is the responsibility of the PREMA system to route messages to objects, even though objects have the ability to migrate between processors. An efficient distributed directory data structure combined with message forwarding allows message routine to take place with minimum overhead, sparing the application from the complex bookkeeping that is associated with mobile objects [50].

After a chunk mobile object is created, a message is sent to it using PREMA's *message* operation. Message processing at the target takes place only during PREMA *polling* operations and involves invoking an application-defined handler routine. This handler is provided with a pointer to the chunk data structure, from which the first and last cell IDs contained within that chunk can be obtained. The chunk also contains a flag which indicates whether this chunk was created locally or has been migrated due to load balancing. The original FMA code distinguishes between these two cases, and therefore the PREMA adaptation does as well. The appropriate FMA supplied work routine is then executed.

The third task is to create the six callback routines that are application-defined but invoked by the runtime system. These routines provide information to the runtime system that is then used to provide priorities to individual objects for scheduling purposes and to prepare data objects for migration. The following are the routines that must be provided:

- **Packing chunks into buffers:** For packing data objects, we are able to make use of a packing routine that was already contained in the FMA code. The packing callback function takes as parameters a pointer to the object to be packed, a pointer to a system-managed buffer, and the target processor id to which the chunk is to be migrated. The first things packed into the buffer are the cell IDs of the first and last cells contained within the chunk. After this, each cell must be serialized and packed into the buffer. Once the packing is complete, the chunk data structure may be deallocated. The number of bytes packed into the buffer is then returned[7].

- **Unpacking chunks from buffers:** Unpacking a chunk is simply the reverse procedure of packing it. Again, we can make use of code which already existed within the FMA. After the cells have been unpacked and a new chunk object created, we set the *work_flag* field to indicate that this object has been migrated. This routine returns a pointer to the newly created chunk object.

- **Determining the chunk size:** This routine returns the size of a chunk object in bytes. The original FMA code has a similar routine which returns 40000 bytes, regardless of the actual size of the chunk; this seems to be a upper bound on the

[7]The original FMA code always stated that the size of the packed buffer was 40000 bytes, regardless of the actual size. This seems to be an upper limit on the size of a chunk. We have maintained this policy in our adaptation.

potential size of a chunk. Larger chunk sizes will cause the FMA code to fail. For consistencies sake, we will continue this policy in our adaptation, although in no way is it required. PREMA allows mobile objects to be of any size.

- **Determining the load of a chunk:** The load imposed by a particular chunk is dependent upon where the contained cells lie in the domain space (on the boundary or in the interior) as well as the number of direct particle interactions contained within the cells. However, for the purposes of this adaptation it is sufficient to state that the load imposed by a chunk is proportional to the number of cells contained within that chunk. Therefore, we state that the load of a chunk is the number of cells within that chunk.

- **Determining the granularity of a chunk:** The granularity of a chunk is an integer index which indicates the difficulty in migrating that chunk. Although the size of chunks may vary (depending on the number of cells and particles contained within that chunk), we elect to state that all chunks are equal difficult to migrate. Therefore, we always return 1.

- **Determining the priority of a chunk:** The priority of a chunk is a vector with an entry for each processor in the parallel system. Each entry is an integer, which higher values indicated a greater "affinity" that chunk has for the associated processor. The application can steer the data migration pattern, for instance by assigning a higher value to the local processor in order to discourage migration, or assigning equal values to each processor in order to attempt to achieve the smoothest possible load distribution. The load and granularity values for the chunk are passed as parameters

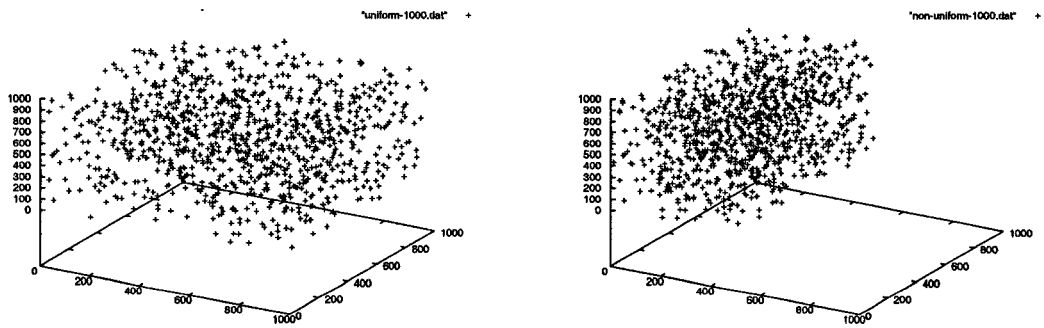
to this routine and can be used to calculate the priorities. Because we want as smooth a load balance as possible, we set each entry in the priority vector to be the load value calculated for this chunk. This will guarantee that larger chunks will execute first, which is what is necessary for fractiling [74] scheduling.

The PREMA runtime library does not provide simply a load balancing algorithm, but is a framework in which many conceivable algorithms may be developed and experimented with quickly. Moving from one method to another involves minimal impact to existing source code. Several load balancing policies come packaged with the PREMA system and are available for immediate use, such as diffusion, work-stealing, and Multi-list [158] methods.

Some experimentation may be necessary in order to determine the optimal load balancing algorithm. We have elected to use a diffusion method, in which the available processor pool is grouped into overlapping *neighborhoods*. When a processor detects that it is underloaded, it first sends a request to each neighbor asking its current work load. Once a response is received for each neighbor, the underloaded processor selects the neighbor with the heaviest load, and requests work to be migrated. If certain neighbors reply that they have no work to contributed, the underloaded processor may replace them with new neighbors for the next round of load balancing.

The number of neighbors can be varied at the beginning of the application run, and represents a tradeoff between overhead and a more complete system image. Although load balancing takes place asynchronously and does not require synchronization within a neighborhood, the overhead grows with the neighborhood size. However, greater number of neighbors tends to lead toward more even work load distribution, due to the fact that a more complete system image is available to the underloaded processors. In our experiments with

Figure 7.28: UNIFORM (A) AND NON-UNIFORM (B) POINT DISTRIBUTION



(a) (b)

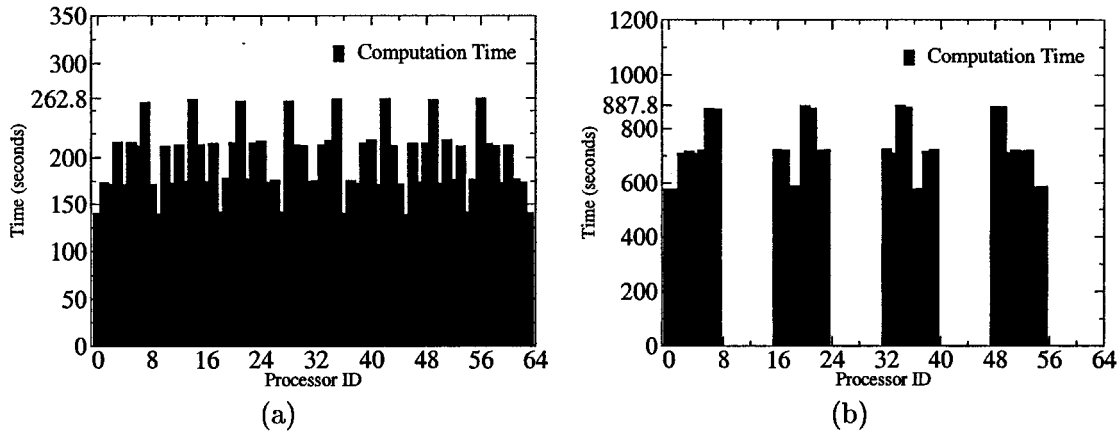
this N-body code, we have found that eight neighbors tends to represent a good tradeoff, although this number is somewhat application dependent.

7.4.5 Experimental Results

We will examine the performance of the PREMA load balancing library using the N-body code we have previously described and which was provided by Mississippi State University, and which implements the Fast Multi-pole Algorithm. All of our experiments are performed on 64 nodes of 650 MHz Sun IIe machines with 1 GB of memory and connected using Fast Ethernet (100 Mbit). Communication is performed using LAM/MPI v.7.0. All simulations contained 1 million points and a cubic spatial domain with each dimension of length 1000, and progressed through a single time-step.

All experiments begin with one of two initial point distributions. Uniform distribution randomly distributes the particles throughout the spacial domain (1000 particles scattered with a uniform distribution are shown in Figure 7.28(a)). Non-uniform particle distribution

Figure 7.29: N-BODY PERFORMANCE WITHOUT LOAD BALANCING



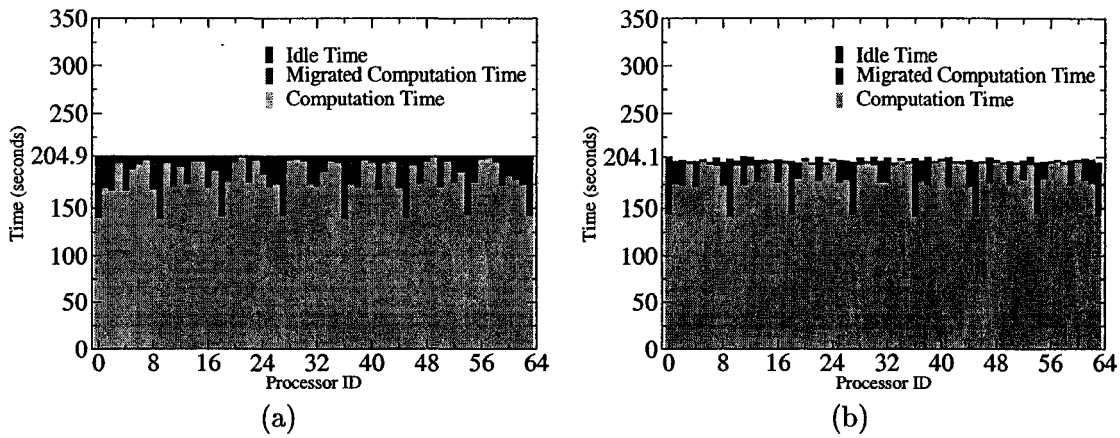
N-body execution times for a single time-step without load balancing for uniform (a) and non-uniform (b) distributions of 1 million particles.

restricts the assignment of points to half of the cube, as shown in Figure 7.28(b). The same number of particles are used in both distributions.

In order to obtain a baseline for performance comparisons, we ran the simulation with no load balancing (Figure 7.29). Even with a uniform particle distribution (Figure 7.29(a)), the simulation is not well load balanced. This is due to the differing numbers of particle interactions for particles in the interior versus those near the boundary. Figure 7.29(b) contains the results for the simulation with a non-uniform distribution. Half of the processors are unused, due to the fact that cells are created by recursively geometrically dividing the spacial domain, regardless of the locations of particles within the domain. Therefore, half of the cells are empty. Cells are then distributed to processors such that each processor begins with an equal number of cells. Due to the method of cell allocation, half of the processors receive empty cells and therefore have no interactions to compute.

Figure 7.30 contains the results for load balancing using both the PREMA runtime li-

Figure 7.30: UNIFORM DISTRIBUTION WITH FIXED CHUNK SIZE 1



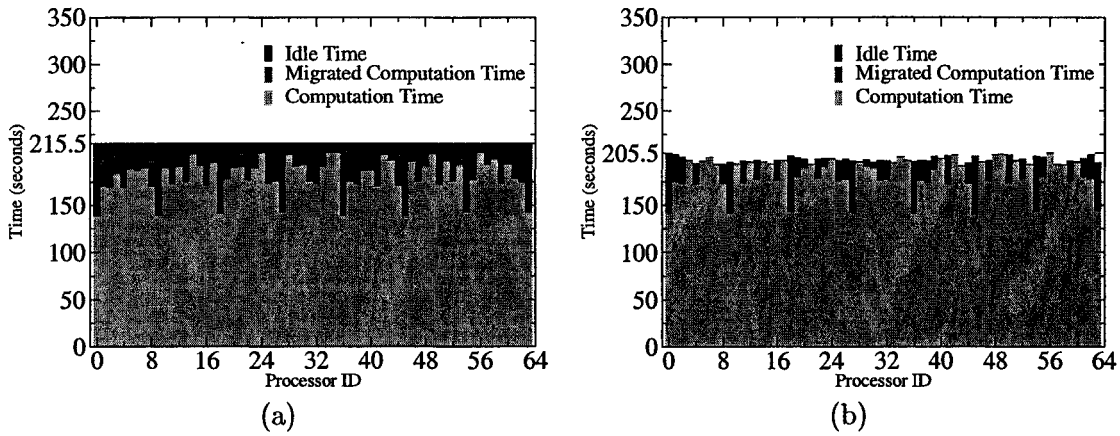
N-body execution time for a single time-step using PREMA (a) and Master/Worker (b) load balancing for uniformly distributed 1 million particles and a static chunking scheme of chunk size 1.

brary and the Master/Worker algorithm originally incorporated into the N-body code (and hereafter referred to as the "Master/Worker" scheme)[8]. In both cases a static chunking scheme was employed in which chunks were made up of single cells. While such a chunking method increases the number of scheduling events, the work load can be more finely distributed, which leads to a more even load distribution. Performance is roughly equal with each load balancing scheme, with overall runtime varying by less than one second.

Another metric that is commonly used to examine the load balancing efficiency is a ratio between the maximum processor computation time to the average time; the closer this ratio is to one, the more even the work distribution. In the case of PREMA, this ratio for the above experiment is 1.020, while for the Master/Worker scheme this ratio is 1.024. However, this figure can be somewhat deceiving in this case. Overheads attributable to the

[8]In all cases, the execution times of the PREMA experiments are equal across all processors, while this is not the case with the Master/Worker method. This is an artifact of the application itself; a "work loop" in which all computation is performed is timed. With PREMA, this loop does not terminate until all processors have completed, leading to equal execution times across all nodes.

Figure 7.31: UNIFORM DISTRIBUTION WITH FIXED CHUNK SIZE 6

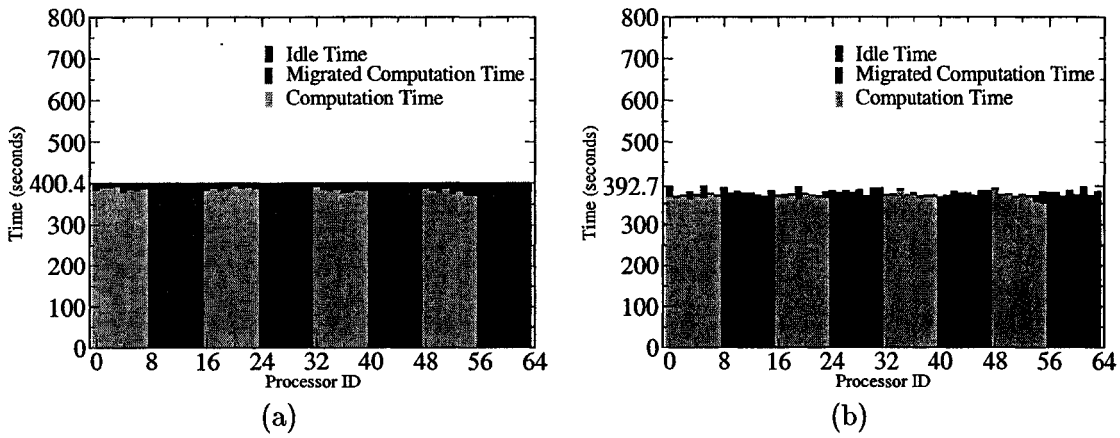


N-body execution time for a single time-step using PREMA (a) and Master/Worker (b) load balancing for uniformly distributed 1 million particles and a static chunking scheme of chunk size 6.

runtime system can give the appearance of a smooth workload. What is important to note is that the work distribution achievable with PREMA can very closely mirror that of the Master/Worker method, which is shown to be optimal.

Figure 7.31 contains the results for the same experiment using a chunking scheme where six cells are allocated to each chunk (there is a single chunk per processor which contains 4 cells; 64 cells are allocated to each processor initially). We can see that the runtime increases slightly for both load balancing methods, although PREMA suffers more. This is due to the fact that the Master/Worker method benefits from knowledge of a global system state. The Master processor is able to determine the optimal processor to donate work to underloaded nodes, ensuring at all times an optimal workload distribution. The PREMA scheduler used in this test makes use of a diffusion algorithm which maintains only local state information. Optimal workload distribution is not possible in this case. However, three points should be stressed here:

Figure 7.32: NON-UNIFORM DISTRIBUTION WITH FIXED CHUNK SIZE 1



N-body execution time for a single time-step using PREMA (a) and Master/Worker (b) load balancing for non-uniformly distributed 1 million particles and a static chunking scheme of chunk size 1.

1. given sufficient workload, the PREMA results presented here are nearly identical to the Master/Worker method,

2. because the PREMA diffusion scheduler presented here makes use of only local information, it is theoretically more scalable than the Master/Worker method[9],

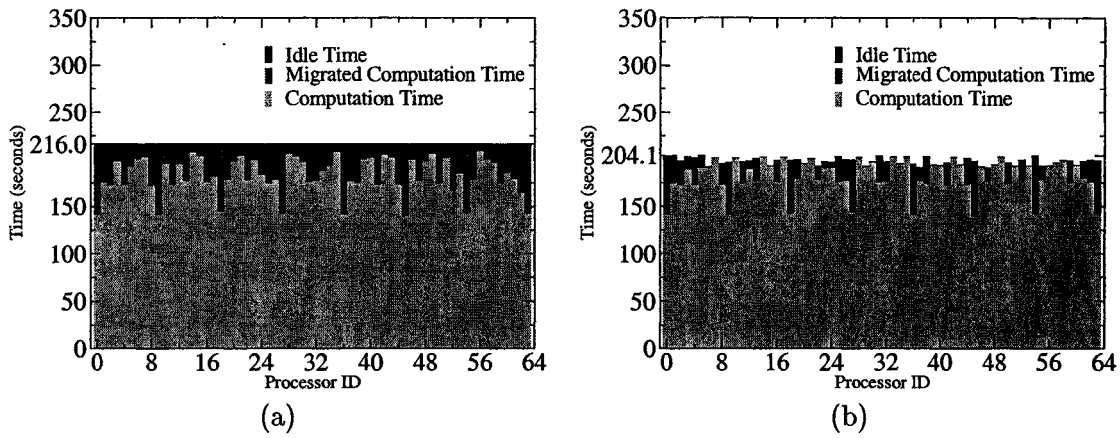
3. and nothing prevents a developer from implementing the Master/Worker method within the PREMA framework[10].

Figure 7.32 shows the results for load balancing a non-uniform particle distribution. Simulation times vary from one run to the next; however, PREMA overall run times varied from being nearly identical to their Master/Worker counterparts to being slower by 1%-2%. This runtime variation can lead to some interesting results, such as what is shown in

[9]We would have liked to demonstrate this fact, but the Master/Worker method crashed for 128 processors, and the processor count must be a power of two.

[10]Although this implementation should be quite trivial, we have not done it.

Figure 7.33: UNIFORM DISTRIBUTION WITH DYNAMIC CHUNK SIZE



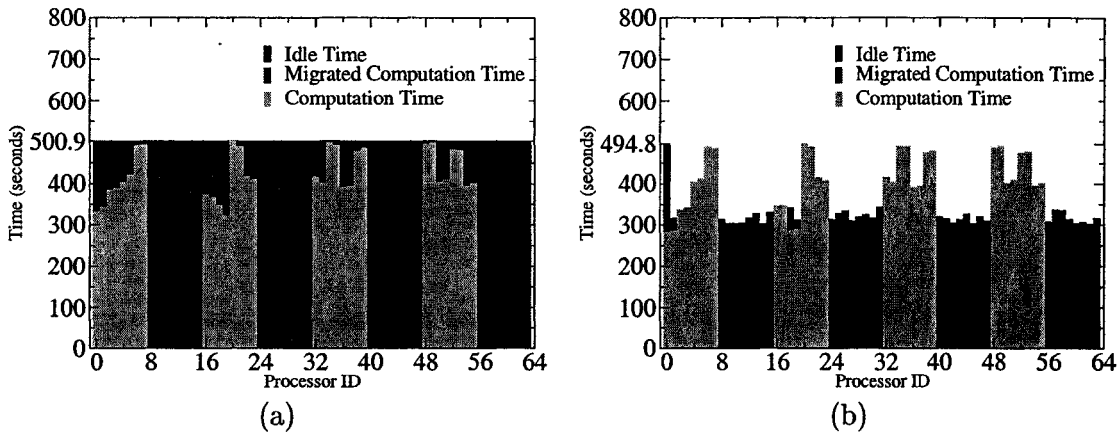
N-body execution time for a single time-step using PREMA (a) and Master/Worker (b) load balancing for uniformly distributed 1 million particles and a dynamic chunking scheme.

Figure 7.32. Here, the load balancing quality demonstrated by PREMA is higher (the ratio between the maximum and average computation times is 1.037, while for the Master/Worker method it is 1.076), while the overall runtime is also higher! Again, PREMA benefits from having a finer-grained workload decomposition, but is theoretically more scalable due to the fact that this particular Scheduler module makes use of only local neighborhood information.

A dynamic chunking scheme creates chunks of different sizes on each processor; each chunk contains half of the remaining cells allocated to that processor. Larger chunks are scheduled at the beginning of the execution, while smaller ones are left to the end to smooth out any load imbalance.

Figure 7.33 contains the execution times for both PREMA (Figure 7.33(a)) and the Master/Worker method (Figure 7.33(b)). The overall execution times are within 3%, and the ratios between the maximum and average computation times are also similar (1.061 for PREMA and 1.032 for the Master/Worker method). Also of note is that the overall

Figure 7.34: NON-UNIFORM DISTRIBUTION WITH DYNAMIC CHUNK SIZE



N-body execution time for a single time-step using PREMA (a) and Master/Worker (b) load balancing for non-uniformly distributed 1 million particles and a dynamic chunking scheme.

execution times displayed in Figure 7.33 are very similar to those in Figure 7.31. This indicates that a performance improvement could be obtained through a finer decomposition of cells.

However, Figure 7.34 contains data from the same experiment but with a non-uniform initial particle distribution. In this case, runtime suffers greatly. This is due to two primary factors. First, having the same number of particles in half the space dramatically increases the number of direct particle interactions, increasing the computational "weight" of each cell. Second, only half of the cells have particles in them, meaning only half of the cells have work to do. These factors together make the workload much more coarse and a smooth distribution impossible.

Even so, the overall runtime achieved using the PREMA library is within 1% of the time achieved using the Master/Worker method. As we have seen in some other experiments, the

Table 7.2: SUMMARY OF OVERHEADS ATTRIBUTABLE TO PREMA

Overhead Type	Uniform Distribution	Non-uniform Distribution
Polling Thread	0.2477 sec	0.5810 sec
Message Sends	0.0392 sec	0.0194 sec
PREMA Message Handlers	0.0214 sec	0.0066 sec
Determining Neighbors	0.0004 sec	0.0010 sec
Determining Data to Migrate	0.0000 sec	0.0001 sec
Scheduler Handlers	0.1554 sec	0.5249 sec
Callback Routines	0.0197 sec	0.0609 sec
Total Overhead	0.4838 sec	1.1937 sec
Total Runtime	204.9 sec	400.4 sec
Overhead Percent	≈0.24%	≈0.29%

total amount of work performed by the PREMA version is greater than the work performed by the Master/Worker method, although the reason for this is not known. While it is true that a second thread is employed by PREMA, this does not account for the difference as the total execution time of the thread is a fraction of a single second on each processor. In addition, the same cells are allocated to each processor in each case, and the same cells are grouped into chunks. It is even often the case that the same chunks are migrated during load balancing in each case.

In addition to demonstrating the quality of the resulting load balancing and the reduction in overall runtime, it is important to show that the PREMA runtime system does not contribute significant overhead which may adversely affect application performance. Table 7.2 contains the overheads attributable to the PREMA system for two different application runs. The column marked "Uniform Distribution" corresponds to an initially uniform particle distribution (whose runtime breakdown is given in Figure 7.30(a)), while the "Non-uniform Distribution" column corresponds to a run with an initial particle imbalance (Figure 7.32(a)).

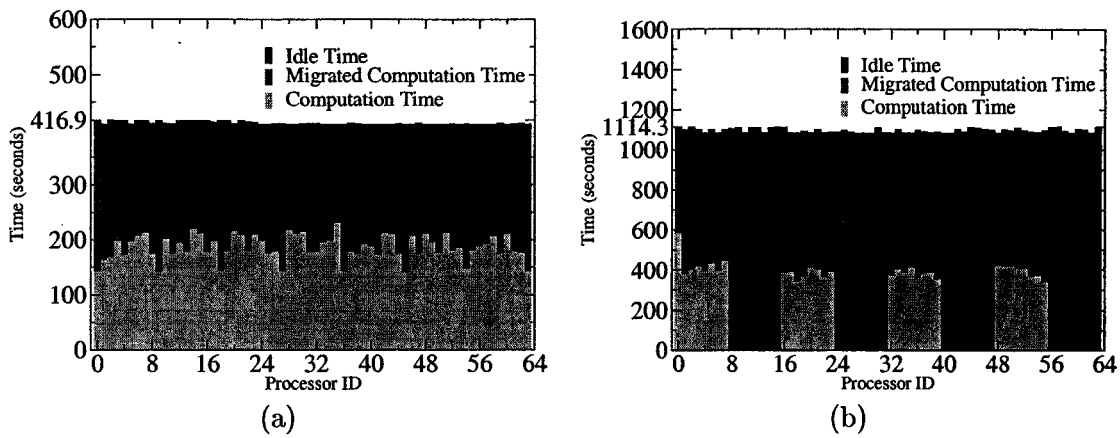
Overhead attributable to the PREMA system can be divided into several categories. The first is the overhead attributable to the second, polling thread which periodically interrupts executing user work units to check the network for load balance requests and updates. We can see that in the case of non-uniform distribution, the polling thread takes a greater amount of time than with a uniform initial distribution. This is due to the fact that the non-empty work units have a greater number of particles, must therefore compute a larger number of inter-particle interactions, and therefore take longer to compute. This gives the polling thread more opportunity to "wake up" and therefore incur greater overhead.

Correspondingly, the second category ("Message Sends" and "PREMA Message Handlers") incurs a smaller overhead in the initially imbalanced case. There are fewer cells that require work, and therefore fewer messages must be sent. This second category measures the overhead incurred by the runtime system in sending and receiving messages.

The third category measures the overhead caused by the Scheduler module itself. It should be noted that this overhead is influenced by the design and implementation of the Scheduler, and will therefore vary from one Scheduler to the next. The components here are "Determining Neighbors", "Determining Data to Migrate", and "Scheduler Handlers". The overhead corresponds to the number of load balancing invocations and items migrated, and is therefore higher in the initially imbalanced case.

The final category is "Callback Routines", which measures the time spent in priority calculation, and the packing and unpacking of migratable objects. These times are again influenced by the number of work units that are migrated. In addition, the packing and unpacking routines are influenced by the number of particles in the migrating cell, which is greater in the initially imbalanced case.

Figure 7.35: MASTER/WORKER LOAD BALANCING WITHOUT PREEMPTION



N-body execution time for a single time-step using the Master/Worker method with no polling during work units for uniform (a) and non-uniform (b) particle distributions.

While the overheads are greater in the imbalanced case, the runtime percentage attributable to overhead is roughly equal for uniform and non-uniform particle distributions. Furthermore, we can see that the PREMA overhead accounts for far less than 1% of the total runtime, demonstrating that PREMA can be used to effectively balance dynamic runtime load without concern that it will dominate application performance.

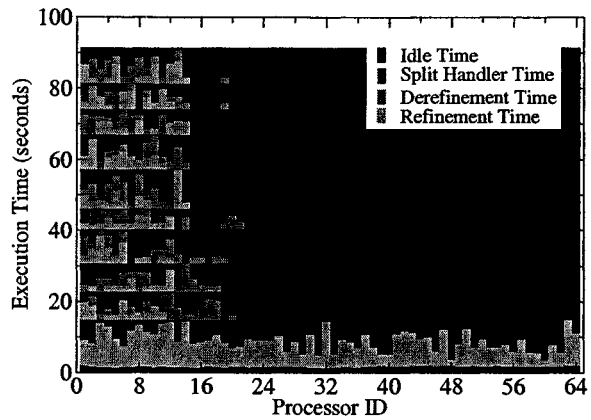
Finally, we want to graphically demonstrate the importance of preemptive decision making to load balancing success. The Master/Worker method is dependent upon periodic polling operations placed within the computational routines. PREMA, on the other hand, removes this necessity by incorporating a preemptive thread which awakes periodically to check for load balancing requests and updates. The frequency at which this thread awakes represents a tradeoff between timely message processing and additional overhead. However, this approach can be advantageous in the cases in which third party software is used during work routines for which source code is not available.

Figure 7.35 depicts the results of the Master/Worker load balancing method when no polling takes place during work routines for uniform (Figure 7.35(a)) and non-uniform (Figure 7.35(b)) initial particle distributions. Both instances suffer dramatically from two problems. The first is the great increase in processor idle time, due to the fact that processors must wait for messages from the Master processor before beginning computation on any chunk. If the request arrives at the Master during a local computation phase, the Master must complete its work (which, in this case may take over a minute) before responding. This time is charged as idle time to the Worker processor. The second problem is that a load balancing request from the Master must suffer this delay once, while a work request from any other processor must suffer twice (once while the request awaits processing at the Master, and again when the load balancing request must wait at the slowest processor). This leads to a work "spike" at the Master and a poor distribution of work.

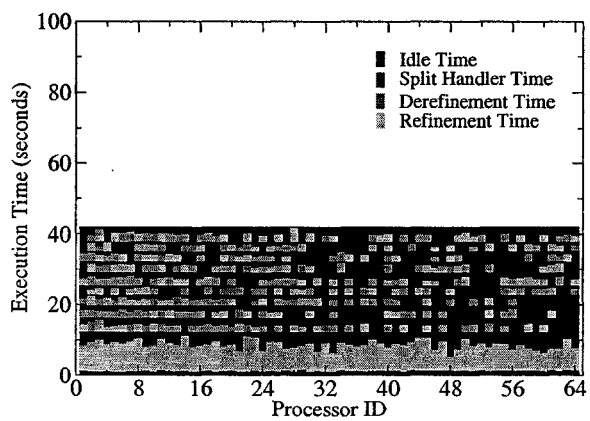
7.5 Loosely Synchronous Benchmark

We conclude our performance evaluation discussion with a look at PREMA's applicability to the class of loosely synchronous parallel applications. To aid in this endeavor, we have constructed a benchmark program which simulates the execution of our previously mentioned Parallel Constrained Delaunay Triangulation (PCDT) code over a series of successive time-steps. As input to the benchmark, we can specify the percentage of sub-domains to *refine* in each time-step, the percentage to *de-refine* or coarsen, and the volume of message traffic containing segment split information. Note that in the first time-step, all sub-domains are refined. The initial geometry is broken into 512 sub-domains in all cases.

Figure 7.36: LOOSELY SYNCHRONOUS TEST; 10 STEPS, 10% REFINEMENT



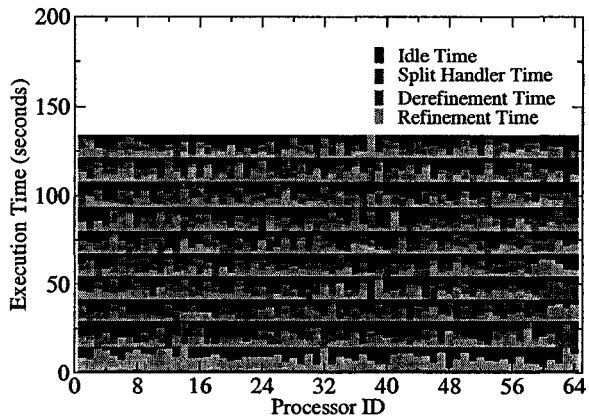
(a) No load balancing



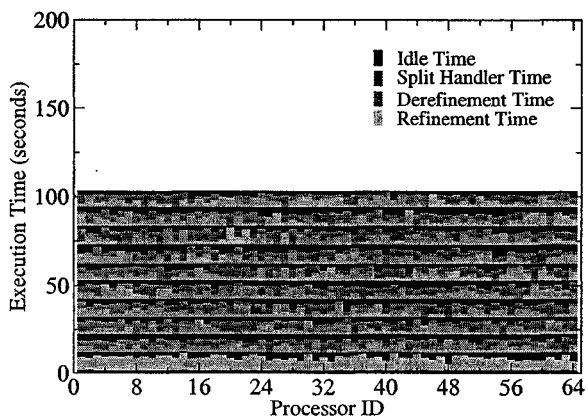
(b) PREMA load balancing

Figure 7.36 contains the results of two experiments carried out on 64 processors. In the top row, 10% of the sub-domains are refined, and 10% are de-refined in each time-step after the first. In Figure 7.36(a), we can see that the majority of the cycles are wasted; once past the initial time-step, the majority of processors have no work, which results in an unnecessarily long total execution time. Using PREMA's dynamic load balancing capability, we are able to substantially reduce the total runtime by distributing work with each time-

Figure 7.37: LOOSELY SYNCHRONOUS TEST; 10 STEPS, 50% REFINEMENT



(a) No load balancing



(b) PREMA load balancing

step (Figure 7.36(b)). We can still see that some processors remain idle within each time-step; however, this is due to the lack of sufficient computation to keep each processor busy. With only 20% of the sub-domains requiring either refinement or coarsening, there are not enough tasks to allocate work to each processor. In this case, using the PREMA library for dynamic load balancing results in a 54% reduction in overall runtime.

Figure 7.37 contains the results of a similar experiment. However, in this case 50%

of the sub-domains require refinement, and 50% of the sub-domains require de-refinement in each time-step. We can see from Figure 7.37(a) that, while none of the processors are completely idle in any time-step, the work is not distributed evenly and there are a large number of idle cycles that can be eliminated. Using the dynamic load balancing capabilities provided by the runtime system results in a reduction in total execution time of 23%.

Repeating the same experiments with 20 time-steps instead of 10 gives similar results. PREMA provides a runtime improvement of 57% in the case of 10% refinement (reducing runtime from 180 to 103 seconds), and 24% in the case of 50% refinement (reducing runtime from 267 to 204 seconds).

Chapter 8

Conclusions and Future Work

In this thesis, we have described two significant contributions:

1. the design and development of the PREMA runtime system, which is mid-level system software to support the dynamic load balancing of adaptive, asynchronous, and irregular parallel applications; and

2. the development of analytic modeling techniques which allow developers to study the effects of load balancer parameter settings in an inexpensive, off-line environment.

Our experimental results and data from comparisons with several prevelant load balancing tools and techniques have shown that PREMA can provide a significant performance improvement for parallel applications whose dynamic load imbalance is caused by either application or systemic factors. Our runtime system design is distiguished from other tools in the field by the flexibility it affords developers; PREMA provides not just a single load balancing policy, or a family of policies, but a framework in which a wide variety of scheduler types may be implemented. We have given some insight into the breadth of possible policies by implementing several schedulers, and have demonstrated their efficiency on several challenging problems.

Our second contribution lies in the area of performance modeling. We have developed a technique based on the derivation of a bi-modal approximation function, which allows users to accurately approximate general task execution patterns using two distinct class types. In addition, we have highlighted a number of runtime parameters, including level of *over-decomposition* and preemption, or polling, *quantum*, which are of critical importance to load balancing quality and overall execution time. Using the analytic model we have developed, we are able to quantitatively study the impact of these parameters on application performance. In particular, our model may be used to answer the nagging question concerning the appropriate level of over-decomposition. Before a problem domain is partitioned, which can be a significant problem in and of itself, the performance achievable from such a partitioning can be accurately predicted. This gives developers the ability to make educated and informed decisions, potentially shortening this costly application phase.

With the PREMA toolkit, users create applications using an explicit-message passing paradigm, which is already very familiar to most developers. In addition, applications may be created in C/C++, a language which is already the platform of choice for many third-party numerical kernels and for which sophisticated compiler support already exists. Such familiarity ensures that developers will be able to quickly get their bearings and begin writing quality parallel code.

The architecture of the runtime system allows developers to utilize only that functionality which they deem appropriate. For instance, applications may be developed using only the provided global namespace and message passing abilities. Once this initial development stage is complete, if workload imbalance is deemed to be harmful, dynamic load balancing can be quickly incorporated without requiring significant portions of code to be rewritten.

In fact, this can often be done simply by including the proper header file, with no further modification to the code at all! We feel this approach will allow applications experts to quickly feel at ease with PREMA; functionality which is not necessary does not have to be incorporated.

We have also striven to keep the interfaces of our software libraries as compact as possible. This can often lead to a tradeoff; it is often the case that small interfaces lead to inflexible software. However, we have had the opportunity to work in close coordination with our "first wave" of application developers, and through this interaction we have been able to ensure that PREMA includes those operations which are determined to be the most beneficial for adaptive and asynchronous codes.

PREMA's explicitly parallel message-passing paradigm is a common one throughout the scientific computing community, and this can allow our runtime system to be quickly integrated into existing codes. We have demonstrated this ability with a challenging N-body simulation code, which a single student was able to integrate with our system software in under two days. Furthermore, the performance achievable with PREMA's load balancing capability lead to significant reductions in overall runtime versus no load balancing, and was comparable to a provably optimal, tightly integrated load balancing mechanism that was already in place within the application.

This project highlights two advantages of the PREMA system, from the application developer's point of view. By separating load balancing from the application itself, developers are able to concentrate on their domain of expertise, leaving the system management and task scheduling to a third-party software library. This leads to a reduction in code complexity and simplifies subsequent code maintenance.

Our future goals for this line of research again focus on two areas. We first plan to incorporate the lessons we have learned in system architecture to improve the performance of the PREMA implementation. Two areas worth mentioning at this time are efficiency across runtime layer boundaries (e.g., between DMCS and the MOL), as well as message passing performance for large numbers of small messages. Many of the codes we have studied are relatively coarse-grained in their decompositions. We are interested in exploring PREMA's applicability to more fine-grained application types, where communication between components is much more frequent. Of critical importance is developing a message passing architecture in which the more sporadic load balancing request and update messages do not become lost in a flood of application messages, hampering load balancing efficiency.

A second line of future research involves integrating our analytic modeling capabilities with the runtime itself, in order to use the model's predictive abilities to dynamically "steer" the application in the presence of dynamic resource requirements. This involves adapting our off-line model to an on-line environment, and embedding it into the runtime system itself. The model will act as a component in a feedback loop, which will monitor the application's status and performance and use this information to adjust any parameters made available by the runtime system. Such a capability will be particularly useful for long-running programs which are composed of phases with very different resource requirements, or in multi-user environments. Additionally, this will remove some setup burden from the developer.

At any rate, the work contained within this thesis is useful to computational scientists today. We have demonstrated the scalability, efficiency, and applicability of our software design and implementation. We have furthermore armed the developer with an analytic

toolkit so that they may achieve the maximum performance from the runtime software. We hope these tools will continue to mature and impact future scientific computing projects.

Appendix A

Compiling and Installing the PREMA Libraries

The PREMA runtime system is composed of several component libraries, which may be compiled all together or separately. Only those libraries which provide the functionality actually used by the application need to be linked in order to form the executable. In other words, if the application uses only the functionality provided by the DMCS library, only *libdmcs.a* must be linked in order to form the executable program.

The first step in preparing to compile and install the libraries is to edit the *config-defs* file in the *prema/* directory. Several macros need to be set to accurately reflect the current directory structure and compilation environment:

- *TOPDIR*: This macro should be set to the directory in which the PREMA source code can be found. The *dmcs/*, *mol/*, and *ilb/* directories should be found here.

- *CXX*: This defines the C++ compiler that should be used to compile the source code. Unlike earlier versions, a C++ compiler must be used to compile all layers of the PREMA library.

- *SYS_FLAGS*: This macro is used to tell the library what type of system it is being compiled on. This is often necessary, for instance, in determining what header files are present. An example is that header files such as *sys/time.h*, which are found on Unix systems, are not found on Windows platforms. This macro is used to control the conditional compilation. Some common definitions to be included are:
  - -D__GCC_296__: Specifies that the g++ v. 2.96 compiler should be used for compilation.
  - -D__SOLARIS_CC__: Specifies that the Sun Microsystems C++ compiler should be used for compilation.
  - -D__WIN_32_CC__: Specifies that the Windows C++ compiler should be used for compilation.
  - -D__LONG_LATENCY_NETWORK__: This macro should be defined when compiling the PREMA system for a platform in which Ethernet is used within the communications layer. This will specify the "cross-over" point in which DMCS will attempt to send large messages in separate pieces, as well as the timeout values used in deadlock detection.
  - -D__SHORT_LATENCY_NETWORK__: This macro should be defined when compiling PREMA for a platform in which a dedicated, high-speed network is available for communication.
  - -D__64_BIT_ARCH__: This macro should be defined when compiling the PREMA system for a 64-bit architecture. By default, compilation is for a 32-bit architecture.

- *DML_DIR*: This tells the system which version of the DML should be compiled into the DMCS library. The DML contains low-level, core functionality upon which the DMCS library is built. A separate DML code base is used for each low-level communication platform, such as MPI. A more complete description of the DML is provided in the DMCS portability discussion later in this document.

- *BUILD_LIB*: This macro defines the commands that are used to combine the compiled object files into a single, statically linked library. This changes from system to system.

- *PREMA_FLAGS*: This macro defines the flags that are to be passed to the library during compilation. They may include:

  - -DPREMA_PROFILING_ON
  - -DPREMA_LOGGING_ON
  - -DPREMA_DEBUG_ON
  - -DPREMA_REENTRANT -mt
  - -DPREMA_USE_POSIX_THREADS
  - -D_REENTRANT
  - -D_POSIX_THREAD_SEMANTICS
  - -DPREMA_USE_SOLARIS_THREADS
  - -DMOL_OUTPUT_USER_HANDLERS

  In addition, one of the following *must* be defined:

  - -DMOL_USE_HASH_DIRECTORY

– -DMOL_USE_MAP_DIRECTORY

With profiling turned on, runtime statistics are gathered and written to files when the runtime system is shutdown. Files in this case take the form of *dmcs-profiling.proc<proc-id>* and *mol-profiling.proc<proc-id>*. Turning on logging will cause descriptive statements to be written to a file as the program progresses. In this case, files are of the form *prema-log.<proc-id>*. All files are written to the current working directory.

Turning on debugging simply compiles more "sanity checking" code into the PREMA libraries. However, a slight performance penalty must be suffered for this extra level of protection.

Defining one of the threads macros will allow the load balancing component of PREMA to make use of preemptive, multi-threaded load balancing capability.

Defining *MOL_OUTPUT_USER_HANDLERS* will cause the name of user-defined handler routines to be written to a file as the handler is executed. The file will have the form *mol-user-handler.<proc-id>* and will be written in the working directory. In order for a name to be associated with a user handler, the handler must be registered using the *mol_register_named_req_handlers()* and *mol_register_named_msg_handlers()* routines covered in Section C.2.

Finally, either *MOL_USE_HASH_DIRECTORY* or *MOL_USE_MAP_DIRECTORY* must be defined. This macro determines what type of data structure will be used for the MOL's internal directory. As such, it probably should not be changed from the default value without good reason.

- *INSTALL_DIR*: This macro specifies the location where the PREMA header files

and libraries should be installed. If this directory does not exist, it will be created (assuming its parent directory exists). The user must have the appropriate permissions to create files in this directory. After compilation, there will be *lib/*, *include/*, and *man/* subdirectories, where the libraries, header files, and online man pages will be installed.

- *DML_MACROS*: This macro is where any flags that must be passed to the DML files during compilation are specified. The DML layer of the runtime system is described further in the portability discussion concerning DMCS.

Appendix B

Data Movement and Control Substrate

The Data Movement and Control Substrate is a thin software runtime layer which provides a point-to-point communication API encompassing only those operations that provide the most benefit to asynchronous and irregular parallel applications. Such applications are characterized as containing communication and computation patterns which cannot be predicted at compile time, and are therefore not amenable to compile time static analysis.

DMCS is designed to be easily portable, thereby isolating the application and higher-level libraries from the specifics of the uderlying communication layer and platform. Applications written using DMCS can therefore quickly and easily be moved to new target platforms without needing to alter application code.

DMCS is lightweight and provides a concise API to the user. This is so that developers can quickly become familiar and comfortable with DMCS and will be inclined to use it for a wide variety of applications. Keeping the code size as small as possible will also decrease the effort required to maintain the library.

DMCS is not designed to compete with or replace any lower-level communication li-

Table B.1: DMCS USER HANDLER PROTOTYPES

Remote Service Request Handlers	
*void (**dmcs_rsr0_handler_t*)(int proc)*	Remote Service Request with no parameters
*void (**dmcs_rsr1_handler_t*)(int proc, dmcs_arg_t arg1)*	Remote Service Request with 1 machine-word-size parameter
*void (**dmcs_rsr2_handler_t*)(int proc, dmcs_arg_t arg1, dmcs_arg_t arg2)*	Remote Service Request with 2 machine-word-size parameters
*void (**dmcs_rsr3_handler_t*)(int proc, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3)*	Remote Service Request with 3 machine-word-size parameters
*void (**dmcs_rsr4_handler_t*)(int proc, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, dmcs_arg_t arg4)*	Remote Service Request with 4 machine-word-size parameters
*void (**dmcs_rsrN_handler_t*)(int proc, dmcs_pointer_t buffer, size_t size)*	Remote Service Request with variable-sized parameter buffer
Remote Memory Manipulation Handlers	
*void (**dmcs_mem_op_handler_t*)(int proc, dmcs_pointer_t loc_addr, size_t size, dmcs_arg_t arg1)*	Handler executed during *get_op* and *put_op* operations

braries. For example, DMCS does not claim to be a replacement for MPI, but instead isolates the application from the specific communication syntax and semantics provided by MPI. DMCS provides a consistent programming model to the user across a wide variety of parallel platforms.

B.1 User-defined Handlers and Prototypes

User-defined handlers must conform to the prototypes given in Table B.1. These handlers make use of the following types:

- *dmcs_arg_t*: This type is a single machine-word sized argument. Typically, this is the size of a pointer, so for instance on a 32-bit machine, a dmcs_arg_t argument can be any 32 bits of data.

- *dmcs_pointer_t*: This is a typical pointer used to refer to local data.

Each handler, regardless of the type, is supplied with the processor id of the calling processor as the first argument. This is useful, for exapmle, when the handler wishes to return a reply to its caller.

Because DMCS is not able to make any assumptions concerning the location of user handlers in memory on each processor, handlers must be registered with the runtime system during initialization. The routines provided for this purpose are discussed in Section B.2.

B.2 Operations Provided

With the programming model firmly established, we can now examine the operations that DMCS provides to the user. These can be broken into five categories: Environment functions, Remote Memory Manipulation functions, Remote Service Request functions, Polling functions, and Synchronization Operations.

B.2.1 Environment Operations

Table B.2 contains the operations in the Environment section of the DMCS API. These functions are responsible for initializing and shutting down the runtime system, along with determining certain runtime information.

dmcs_init()

Parameters:

1. **int argc**: The number of command line arguments given to the executable from the command line and subsequently passed to the *main()* routine

Table B.2: DMCS ENVIRONMENT OPERATIONS

System Initialization and Shutdown	
void **dmcs_init***(int argc, char* argv[])*	Initialize the DMCS system
void **dmcs_shutdown***()*	Shuts down the DMCS system
Profiling Information Printing	
void **dmcs_profiling_dump_info***(char* path)*	Outputs profiling information
Querying the Environment	
int **dmcs_my_proc***()*	Returns the processor id of the caller
int **dmcs_num_procs***()*	Returns the number of processors in the system
Handler Registration	
void **dmcs_register_rsr0_handlers***(dmcs_rsr0_handler_t handlers[], int size)*	Register user handlers that take no arguments
void **dmcs_register_rsr1_handlers***(dmcs_rsr1_handler_t handlers[], int size)*	Register user handlers that take 1 argument
void **dmcs_register_rsr2_handlers***(dmcs_rsr2_handler_t handlers[], int size)*	Register user handlers that take 2 arguments
void **dmcs_register_rsr3_handlers***(dmcs_rsr3_handler_t handlers[], int size)*	Register user handlers that take 3 arguments
void **dmcs_register_rsr4_handlers***(dmcs_rsr4_handler_t handlers[], int size)*	Register user handlers that take 4 arguments
void **dmcs_register_rsrN_handlers***(dmcs_rsrN_handler_t handlers[], int size)*	Register user handlers that take variable sized argument buffers
void **dmcs_register_mem_op_handlers***(dmcs_mem_op_handler_t handlers[], int size)*	Register user handlers used for remote memory reads and writes

2. **char* argv[]**: The command line arguments given to the executable from the command line and subsequently passed to the *main()* routine

Returns: void

Description:

This function is responsible for initializing the runtime system and must be the first DMCS operation called. DMCS is responsible for initializing any underlying communication substrate, so, for instance, the application should no longer call *MPI_Init()* or similar initialization function. Parameters to *dmcs_init()* include the *argc* and *argv* parameters given to the *main()* routine, and no return value is given. This operation is collective, so all processors must call *dmcs_init()* at the

same time.

dmcs_shutdown()

Parameters: None

Returns: None

Description:

This operation is the final DMCS call made by any application. As with initialization, DMCS is responsible for shutting down any lower level communication systems, so such calls should not be present in the application. No parameters are given to the shutdown routine, and no return type is expected. As with initialization, *dmcs_shutdown()* is a collective operation and must be called by each processor.

dmcs_my_proc()

Parameters: None

Returns: Integer; the ID of the calling processor

Description:

This routine is used to determine the ID of the calling processor. Processors are numbered according to the method used by the underlying communication system. Typically, this is a number between zero and $P-1$, where P is the total number of processors.

dmcs_num_procs()

Parameters: None

Returns: Integer; the number of processes in the parallel system

Description:

Returns the number of processes in the parallel system.

B.2.2 Handler Registration Operations

Table B.2 contains the routines that are used to register application-defined DMCS message handlers with the runtime system. All message handlers must be registered with the runtime system before they may be invoked as the result of a message from a remote processor. Handler registration must be performed by each processor, with every processor registering handlers in the same order. Typically this is done immediately after initializing the system.

dmcs_register_<*handler-type*>_handlers()

Parameters:

1. **dmcs_<*handler-type*>_handler_t handlers[]**: This is an array of the handlers to register with DMCS. These arrays must conform to the correct prototype, given in Table B.1.
2. **int size**: The number of handlers in the array

Returns: None

Description:

These routines are used to register user-defined handlers with the runtime system. This is necessary due to the fact that DMCS is unable to make any assumption regarding the locations of function code in memory on remote processors. Therefore, instead of referring to remote handlers by absolute address, another level of

Table B.3: ALLOCATING AND FREEING REMOTE MEMORY

*void** **dmcs_malloc***(int proc, size_t size)*	Allocate memory on a remote processor
void **dmcs_free***(int proc, void* addr)*	Free memory on a remote processor

indirection is provided via handler table indices. There are handler registration operations provided for each type of user handler. These routines take an array of handlers and the length of the array as pointers. Handler registration must be performed on each processor at the same time, typically immediately after the *dmcs_init()* call is made. The valid handler types include:

- *rsr0*: Remote Service Request that takes zero machine-word sized arguments.
- *rsr1*: Remote Service Request that takes one machine-word sized argument.
- *rsr2*: Remote Service Request that takes two machine-word sized arguments.
- *rsr3*: Remote Service Request that takes three machine-word sized arguments.
- *rsr4*: Remote Service Request that takes four machine-word sized arguments.
- *rsrN*: Remote Service Request that takes a variable length buffer as a parameter.
- *mem-op*: Handler used for a *get-op* or *put-op* operation.

B.2.3 Remote Memory Manipulation Operations

Tables B.3, B.4, and B.5 contain the operations used to manipulate remote memory. This includes allocation and deallocation, as well as reads and writes. There are two types of operations which can be described as *asynchronous* and *synchronous*. Broadly speaking,

Table B.4: DMCS OPERATIONS TO READ REMOTE MEMORY

void **dmcs_sync_get***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, int tag)*	Read data from a remote processor
void **dmcs_async_get***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, int tag)*	Read data from a remote processor
void **dmcs_sync_get_op***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, dmcs_mem_op_handler_t loc_handler, dmcs_arg_t handler_arg, int tag)*	Read data from a remote processor and execute a handler on the local node
void **dmcs_async_get_op***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, dmcs_mem_op_handler_t loc_handler dmcs_arg_t loc_arg dmcs_mem_op_handler_t rem_handler dmcs_arg_t rem_arg dmcs_arg_t rem_arg, int tag)*	Read data from a remote processor and execute a handler on both the local and remote node

synchronous operations return only when a condition has taken place on the remote processor, while asynchronous operations are under no such restriction. While synchronous operations will incur a greater latency from the application's point of view, developers are guaranteed that data buffers are free to be altered or deallocated once the operation returns. For asynchronous operations, a secondary mechanism must be provided to signal the user that data buffers are safe to handle.

dmcs_malloc()

Parameters:

1. **int proc**: The procesor on which the memory should be allocated.
2. **size_t size**: The number of bytes to allocate.

*Returns: void**; address of allocated memory

Table B.5: DMCS OPERATIONS TO WRITE TO REMOTE MEMORY

void **dmcs_sync_put***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, int tag)*	Write data to a remote processor
void **dmcs_block_put***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, int tag)*	Write data to a remote processor
void **dmcs_noblock_put***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, dmcs_status_t* status, int tag)*	Write data to a remote processor
void **dmcs_sync_put_op***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, dmcs_mem_op_handler_t rem_handler, dmcs_arg_t handler_arg, int tag)*	Write data to a remote processor and execute a handler on the remote node
void **dmcs_block_put_op***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, dmcs_mem_op_handler_t rem_handler, dmcs_arg_t handler_arg, int tag)*	Write data to a remote processor and execute a handler on the remote node
void **dmcs_noblock_put_op***(int proc, size_t size, dmcs_pointer_t rem_addr, dmcs_pointer_t loc_addr, dmcs_mem_op_handler_t rem_handler dmcs_arg_t rem_arg, dmcs_mem_op_handler_t loc_handler, dmcs_arg_t loc_arg, dmcs_status_t* status, int tag)*	Write data to a remote processor and execute a handler on both the local and remote node

Description:

This function is used to allocate memory on a remote processor. It works by sending a request to the specified processor and waiting for a reply. This function is *synchronous* and will not return until a reply is heard back from the target node. The pointer returned is valid only on the remote processor.

dmcs_free()

Parameters:

1. **int proc**: The processor on which the memory was previously allocated.
2. **void* addr**: The address of the previously allocated memory.

Returns: None

Description:

This function is used to free a block of memory that was previously allocated using the *dmcs_malloc()* operation. This operation is *non-blocking* and *asynchronous*, meaning that it will return at its earliest opportunity and will not synchronize with the remote processor.

dmcs_sync_get()

Parameters:

1. **int proc**: The processor from which to retrieve memory.
2. **size_t size**: The number of bytes to retrieve.
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to read from.
4. **dmcs_pointer_t loc_addr**: The address on the local processor to write to.
5. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This function will retrieve memory of a specified size from a remote processor, and copy that memory into the provided local buffer. It will not return to the user until the buffer is filled. This routine assumes that the local buffer provided is of sufficient size to hold the incoming data. The data on the remote node is unaltered. The application has the reponsibility to allocate and free local and remote memory buffers.

dmcs_async_get()

Parameters:

1. **int proc**: The processor from which to retrieve memory.
2. **size_t size**: The number of bytes to retrieve.
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to read from.
4. **dmcs_pointer_t loc_addr**: The address on the local processor to write to.
5. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This function will retrieve memory of a specified size from a remote processor, and will copy that memory into the provided local buffer. It will return to the user at the earliest opportunity, and will not wait for the data to arrive from the remote node. If the user wishes to be notified of the data arrival, it is suggested that one of the *get_op* operations be used instead.

dmcs_sync_get_op()

Parameters:

1. **int proc**: The processor from which to retrieve memory.
2. **size_t size**: The number of bytes to retrieve.
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to read from.
4. **dmcs_pointer_t loc_addr**: The address on the local processor to write to.
5. **dmcs_mem_op_handler_t loc_handler**: A pointer to a user-defined handler function to execute on the local processor once the data has arrived at the caller.
6. **dmcs_arg_t handler_arg**: A single machine-word sized argument to pass to the remote handler
7. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation

discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This function invokes a two-step operation designed to copy memory from the target processor to the source processor. In the first step, a request is sent to a target processor asking for a memory block of a specified size and at a specified location to be returned to the calling processor. The target processor replies with the requested memory; once this reply arrives at the originating processor, a user-defined handler is executed[1]. The handler is supplied with the local address where the memory was copied, the size of the memory block, and a single machine-word sized argument. Because this operation is synchronous, it will not return to the user until the *original* message to the target processor has arrived. Note, however, that this routine will return to the user before the data has been returned from the target. It is therefore necessary for the user application to post a polling operation in order to receive the data.

dmcs_async_get_op()

Parameters:

1. **int proc**: The processor from which to retrieve memory.
2. **size_t size**: The number of bytes to retrieve.

[1] Actually, the handler is not executed until a *polling* operation is posted.

3. **dmcs_pointer_t rem_addr**: The address on the remote processor to read from.
4. **dmcs_pointer_t loc_addr**: The address on the local processor to write to.
5. **dmcs_mem_op_handler_t loc_handler**: A pointer to a user-defined handler function to execute on the local processor once the data has arrived at the target.
6. **dmcs_arg_t loc_arg**: A single machine-word sized argument to pass to the local handler.
7. **dmcs_mem_op_handler_t rem_handler**: A pointer to a user-defined handler function to execute on the remote processor once the data has arrived at the target.
8. **dmcs_arg_t rem_arg**: A single machine-word sized argument to pass to the remote handler.
9. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG.

Returns: None

Description:

Like the *dmcs_sync_get_op()* operation, this function is designed to copy memory from the target processor to the source. However, there are some semantic

differences. First, the operation will return as soon as possible; the originating request to the target processor is asynchronous. Second, once the reply from the target has arrived at the source node, another message is sent to the target which will execute a user-defined *remote handler*. This handler may be used, for instance, to deallocate the referenced memory on the target node.

dmcs_sync_put()

Parameters:

1. **int proc**: The processor to which to write data.
2. **size_t size**: The number of bytes to write.
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to write the data.
4. **dmcs_pointer_t loc_addr**: The address on the local processor from which to read the data.
5. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This function will write a specified memory buffer into a specified location on

the target processor. It will not return until the receive operation has begun on the target node. In addition, this operation assumes that the target buffer has been allocated and is of sufficient size to hold the incoming data. The memory buffer on the source processor is unaltered.

dmcs_noblock_put()

Parameters:

1. **int proc**: The processor to which to write data.
2. **size_t size**: The number of bytes to write.
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to which to write the data.
4. **dmcs_pointer_t loc_addr**: The address on the local processor from which to read the data.
5. **dmcs_status_t* status**: The status variable can be used to determine when it is safe to modify the parameter data buffer. The operations available to test the status object are described in this document. If a status object is not required, the constant *DMCS_STATUS_IGNORE* may be used.
6. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This function will write a specified memory buffer into a specified location on the target processor. It will return to the user at the earliest opportunity and will not wait for the data to be copied to the network. Therefore, the user should not modify the source buffer; if the application needs to know when it is safe to access the buffer but still wants the low latency associated with asynchronous operations, it is suggested that the asynchronous *put_op* operations be used instead. The target memory buffer is assumed to be of sufficient size to contain the incoming data. The memory buffer on the source processor is unchanged.

dmcs_block_put()

Parameters:

1. **int proc**: The processor to which to write data.
2. **size_t size**: The number of bytes to write.
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to which to write the data.
4. **dmcs_pointer_t loc_addr**: The address on the local processor from which to read the data.
5. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a

certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This function will write a specified memory buffer into a specified location on the target processor. It will return to the user once it is safe for the outgoing data buffer to be modified. It is assumed that the target memory buffer is of sufficient size to contain the incoming data. In addition, the memory buffer on the source processor is unchanged by this operation.

dmcs_sync_put_op()

Parameters:

1. **int proc**: The processor to which to write data
2. **size_t size**: The number of bytes to write
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to write the data
4. **dmcs_pointer_t loc_addr**: The address on the local processor from which to read the data
5. **dmcs_mem_op_handler_t rem_handler**: A pointer to a user-defined handler function to execute on the remote processor once the data has been written to the target

6. **dmcs_arg_t handler_arg**: A single machine-word sized argument to pass to the local handler
7. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This function will write a specified memory buffer into a specified location on the target processor. It will return to the user only after the request has arrived at the target processor. In addition, a user-defined handler function will execute on the target and may be used to possibly inform the processor of the arrival of the data. It is assumed that the target memory buffer is of sufficient size to contain the incoming data. The source memory buffer is unchanged by this operation.

dmcs_noblock_put_op()

Parameters:

1. **int proc**: The processor to which to write data
2. **size_t size**: The number of bytes to write
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to which to write the data

4. **dmcs_pointer_t loc_addr**: The address on the local processor from which to read the data

5. **dmcs_mem_op_handler_t rem_handler**: A pointer to a user-defined handler function to execute on the remote processor once the data has been written to the target

6. **dmcs_arg_t rem_arg**: A single machine-word sized argument to pass to the remote handler

7. **dmcs_mem_op_handler_t loc_handler**: A pointer to a user-defined handler function to execute on the local processor once the data has been written to the target

8. **dmcs_arg_t loc_arg**: A single machine-word sized argument to pass to the local handler

9. **dmcs_status_t* status**: The status variable can be used to determine when it is safe to modify the parameter data buffer. The operations available to test the status object are described in this document. If a status object is not required, the constant *DMCS_STATUS_IGNORE* may be used.

10. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

Like the *dmcs_sync_put_op()* operation, this function will copy memory from the source processor to the specified target. However, unlike its synchronous counterpart, this operation returns to the user at the earliest possible point. Therefore, the application must not modify the parameter buffer until it is safe to do so. Once the message has arrived at the target, and before the remote handler is executed, a reply is sent from the target back to the source to execute a user-specified handler. This handler may inform the source processor that it is now safe to modify the local memory buffer. Additionally, the DMCS status object may be used to inform the sender that the message has been sent to the target.

dmcs_block_put_op()

Parameters:

1. **int proc**: The processor to which to write data
2. **size_t size**: The number of bytes to write
3. **dmcs_pointer_t rem_addr**: The address on the remote processor to write the data
4. **dmcs_pointer_t loc_addr**: The address on the local processor from which to read the data
5. **dmcs_mem_op_handler_t rem_handler**: A pointer to a user-defined handler function to execute on the remote processor once the data has

been written to the target

6. **dmcs_arg_t handler_arg**: A single machine-word sized argument to pass to the local handler

7. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

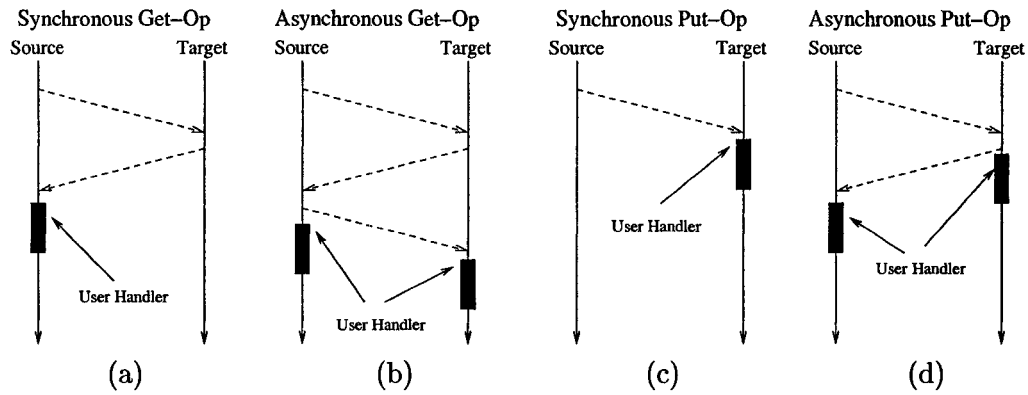
Returns: None

Description:

This function will write a specified memory buffer into a specified location on the target processor. It will return to the user once it is safe to modify the outgoing data buffer. In addition, a user-defined handler function will execute on the target and may be used to possibly inform the processor of the arrival of the data. It is assumed that the target memory buffer is of sufficient size to contain the incoming data. The source memory buffer is unchanged by this operation.

Figure B.1 depicts the communication pattern between source and target processors for the *put_op* and *get_op* type operations. In each diagram, time moves downward, and messages between processors are shown as dashed arrows. Black rectangles represent user-specified handlers. Note that the *synchronous* `get-op` operation (Figure B.1(a)) does not

Figure B.1: DMCS GET-OP AND PUT-OP OPERATIONS



block until the data is returned to the initiator; instead a user-specified handler is executed when the data arrives on the calling node. This handler may be used to signal the process that the requested data has arrived. Likewise with the *synchronous* `put-op` operation (Figure B.1(c)). The *asynchronous* `put-op` operation (Figure B.1(d)) will invoke user-specified handlers on both the source and target nodes; these handlers typically notify the process that the data has migrated.

B.2.4 Remote Service Request Operations

Tables B.6, B.7, and B.8 contain the operations used to send Remote Service Requests (RSRs) between processors. Operations are provided which send a fixed number of machine-word sized arguments (zero through four), as well as an application-defined buffer of any size. It is the user's responsibility to marshall data into contiguous memory prior to transmission. Any buffers required for parameter marshalling on the sending processor fall under the responsibility of the user; the system will not deallocate these buffers.

In addition, the target processor must make sure to place incoming data into safe loca-

Table B.6: DMCS BLOCKING REMOTE SERVICE REQUEST OPERATIONS

void **dmcs_block_rsr0***(int proc, dmcs_rsr0_handler_t handler, int tag)*	Send a blocking RSR with no arguments
void **dmcs_block_rsr1***(int proc, dmcs_rsr1_handler_t handler, dmcs_arg_t arg1, int tag)*	Send a blocking RSR with one machine-word sized argument
void **dmcs_block_rsr2***(int proc, dmcs_rsr2_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, int tag)*	Send a blocking RSR with two machine-word sized arguments
void **dmcs_block_rsr3***(int proc, dmcs_rsr3_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, int tag)*	Send a blocking RSR with three machine-word sized arguments
void **dmcs_block_rsr4***(int proc, dmcs_rsr4_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, dmcs_arg_t arg4, int tag)*	Send a blocking RSR with four machine-word sized arguments
void **dmcs_block_rsrN***(int proc, dmcs_rsrN_handler_t handler, dmcs_pointer_t buffer, size_t size, int tag)*	Send a blocking RSR with variable sized argument buffer

tions; memory buffers given to message handlers are not guaranteed to be permanent. For instance, the underlying communication layer may elect to reuse memory buffers to store incoming message data. Because these buffers will likely be reused in the future, data left in them is not guaranteed to be permanent. It is left to the application to move incoming data to safe locations. With all message types, *polling* operations must be posted on the target processors; once a message has arrived and a *polling* operation has been posted, the specified application-defined handler function will be executed with the specified arguments as parameters.

The semantics of the various RSR types are discussed below. The operations come in three types. *Nonblocking* operations return to the user before the associated data has been copied to the network. While the latency observable to the application is the lowest in

Table B.7: DMCS NONBLOCKING REMOTE SERVICE REQUEST OPERATIONS

void **dmcs_noblock_rsr0***(int proc, dmcs_rsr0_handler_t handler, int tag)*	Send a nonblocking RSR with no arguments
void **dmcs_noblock_rsr1***(int proc, dmcs_rsr1_handler_t handler, dmcs_arg_t arg1, int tag)*	Send a nonblocking RSR with one machine-word sized argument
void **dmcs_noblock_rsr2***(int proc, dmcs_rsr2_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, int tag)*	Send a nonblocking RSR with two machine-word sized arguments
void **dmcs_noblock_rsr3***(int proc, dmcs_rsr3_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, int tag)*	Send a nonblocking RSR with three machine-word sized arguments
void **dmcs_noblock_rsr4***(int proc, dmcs_rsr4_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, dmcs_arg_t arg4, int tag)*	Send a nonblocking RSR with four machine-word sized arguments
void **dmcs_noblock_rsrN***(int proc, dmcs_rsrN_handler_t handler, dmcs_pointer_t buffer, size_t size, dmcs_status_t* status, int tag)*	Send a nonblocking RSR with variable sized argument buffer

this case, users must be careful not to modify data buffers until the data has been safely transferred[2]. *Blocking* operations do not return to the user until the outgoing message has been copied to the network. For large messages in particular, the latency incurred by blocking operations can be significantly higher than with their nonblocking counterparts. However, once the operation has returned, users can safely modify any parameter data buffers. The final operation type is *synchronous.* Synchronous operations do not return until the data has been received at the target processor, and therefore have the highest observable latency of the three types. The three blocking types are specified with the strings "noblock", "block", and "sync".

dmcs_<*block-type*>**_rsr0()**

[2]However, with the provided *status* parameter, users are able to associate DMCS status objects with nonblocking communication operations, which can be checked in order to determine operation completeness.

Table B.8: DMCS SYNCHRONOUS REMOTE SERVICE REQUEST OPERATIONS

void **dmcs_sync_rsr0***(int proc, dmcs_rsr0_handler_t handler, int tag)*	Send a synchronous RSR with no arguments
void **dmcs_sync_rsr1***(int proc, dmcs_rsr1_handler_t handler, dmcs_arg_t arg1, int tag)*	Send a synchronous RSR with one machine-word sized argument
void **dmcs_sync_rsr2***(int proc, dmcs_rsr2_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, int tag)*	Send a synchronous RSR with two machine-word sized arguments
void **dmcs_sync_rsr3***(int proc, dmcs_rsr3_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, int tag)*	Send a synchronous RSR with three machine-word sized arguments
void **dmcs_sync_rsr4***(int proc, dmcs_rsr4_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, dmcs_arg_t arg4, int tag)*	Send a synchronous RSR with four machine-word sized arguments
void **dmcs_block_rsrN***(int proc, dmcs_rsrN_handler_t handler, dmcs_pointer_t buffer, size_t size, int tag)*	Send a synchronous RSR with variable sized argument buffer

Parameters:

1. **int proc**: The processor on which the specified handler is to execute
2. **dmcs_rsr0_handler_t handler**: A pointer to a user-defined handler function which is to execute on the remote processor; no parameters are passed to the handler
3. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This type of Remote Service Request takes no arguments, other than the handler to execute on the remote processor. The handler executed on the remote processor must conform to the prototype given in Table B.1.

dmcs_<*block-type*>_rsr1()

Parameters:

1. **int proc**: The processor on which the specified handler is to execute
2. **dmcs_rsr1_handler_t handler**: A pointer to a user-defined handler function which is to execute on the remote processor; a single machine-word sized argument is passed to this handler
3. **dmcs_arg_t arg**: A single machine-word sized argument to be passed to the handler
4. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This type of Remote Serivce Request takes a single machine-word size argument.

The handler executed on the remote processor must conform to the prototype given in Table B.1.

dmcs_<*block-type*>_rsr2()

Parameters:

1. **int proc**: The processor on which the specified handler is to execute
2. **dmcs_rsr2_handler_t handler**: A pointer to a user-defined handler function which is to execute on the remote processor; two machine-word sized arguments are passed to this handler
3. **dmcs_arg_t arg1**: A single machine-word sized argument to be passed to the handler
4. **dmcs_arg_t arg2**: A single machine-word sized argument to be passed to the handler
5. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This type of Remote Serivce Request takes two machine-word size arguments.

The handler executed on the remote processor must conform to the prototype given in Table B.1.

dmcs_<*block-type*>_rsr3()

Parameters:

1. **int proc**: The processor on which the specified handler is to execute
2. **dmcs_rsr3_handler_t handler**: A pointer to a user-defined handler function which is to execute on the remote processor; three machine-word sized arguments are passed to this handler
3. **dmcs_arg_t arg1**: A single machine-word sized argument to be passed to the handler
4. **dmcs_arg_t arg2**: A single machine-word sized argument to be passed to the handler
5. **dmcs_arg_t arg3**: A single machine-word sized argument to be passed to the handler
6. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This type of Remote Serivce Request takes three machine-word size arguments. The handler executed on the remote processor must conform to the prototype given in Table B.1.

dmcs_<*block-type*>_rsr4()

Parameters:

1. **int proc**: The processor on which the specified handler is to execute
2. **dmcs_rsr4_handler_t handler**: A pointer to a user-defined handler function which is to execute on the remote processor; four machine-word sized arguments are passed to this handler
3. **dmcs_arg_t arg1**: A single machine-word sized argument to be passed to the handler
4. **dmcs_arg_t arg2**: A single machine-word sized argument to be passed to the handler
5. **dmcs_arg_t arg3**: A single machine-word sized argument to be passed to the handler
6. **dmcs_arg_t arg4**: A single machine-word sized argument to be passed to the handler
7. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation

discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This type of Remote Serivce Request takes four machine-word size arguments. The handler executed on the remote processor must conform to the prototype given in Table B.1.

dmcs_<*block-type*>_rsrN()

Parameters:

1. **int proc**: The processor on which the specified handler is to execute
2. **dmcs_rsrN_handler_t handler**: A pointer to a user-defined handler function which is to execute on the remote processor; a single parameter buffer of a user-specified size is passed as an argument to this handler
3. **dmcs_pointer_t buffer**: A pointer to the parameter data buffer
4. **int size**: The size of the parameter data buffer, in bytes
5. **dmcs_status_t* status**: The status variable can be used to determine when it is safe to modify the parameter data buffer. The operations available to test the status object are described in this document. If a status object is not required, the constant *DMCS_STATUS_IGNORE* may be used. **NOTE**: This is a parameter for the *non-blocking* RSR only!
6. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to

Table B.9: DMCS BLOCKING BROADCAST OPERATIONS

void **dmcs_block_broadcast_rsr0**(*dmcs_rsr0_handler_t handler, int tag)*	Invoke blocking RSR with no arguments
void **dmcs_block_broadcast_rsr1**(*dmcs_rsr1_handler_t handler, dmcs_arg_t arg, int tag)*	Invoke blocking RSR with a single machine-word sized argument
void **dmcs_block_broadcast_rsr2**(*dmcs_rsr2_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, int tag)*	Invoke blocking RSR with two machine-word sized arguments
void **dmcs_block_broadcast_rsr3**(*dmcs_rsr3_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, int tag)*	Invoke blocking RSR with three machine-word sized arguments
void **dmcs_block_broadcast_rsr4**(*dmcs_rsr4_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, dmcs_arg_t arg4, int tag)*	Invoke blocking RSR with four machine-word sized arguments
void **dmcs_block_broadcast_rsrN**(*dmcs_rsrN_handler_t handler, dmcs_pointer_t buffer, int size, int tag)*	Invoke blocking RSR with variable sized argument buffer

differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This type of Remote Service Request takes a single user-defined parameter buffer as an argument. The handler executed on the remote processor must conform to the prototype given in Table B.1.

Table B.10: DMCS NON-BLOCKING BROADCAST OPERATIONS

void **dmcs_noblock_broadcast_rsr0(** *dmcs_rsr0_handler_t handler, int tag)*	Invoke nonblocking RSR with no arguments
void **dmcs_noblock_broadcast_rsr1(** *dmcs_rsr1_handler_t handler, dmcs_arg_t arg, int tag)*	Invoke nonblocking RSR with a single machine-word sized argument
void **dmcs_noblock_broadcast_rsr2(** *dmcs_rsr2_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, int tag)*	Invoke nonblocking RSR with two machine-word sized arguments
void **dmcs_noblock_broadcast_rsr3(** *dmcs_rsr3_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, int tag)*	Invoke nonblocking RSR with three machine-word sized arguments
void **dmcs_noblock_broadcast_rsr4(** *dmcs_rsr4_handler_t handler, dmcs_arg_t arg1, dmcs_arg_t arg2, dmcs_arg_t arg3, dmcs_arg_t arg4, int tag)*	Invoke nonblocking RSR with four machine-word sized arguments
void **dmcs_noblock_broadcast_rsrN(** *dmcs_rsrN_handler_t handler, dmcs_pointer_t buffer, int size, dmcs_status_t* status, int tag)*	Invoke nonblocking RSR with variable sized argument buffer

B.2.5 Broadcast Operations

Broadcast operations are used to invoke user-defined handlers in all processes *except* for the caller. The blocking and nonblocking semantics are identical as for Remote Service Request invocations. Blocking operations return once it is safe to modify any user-defined parameter buffers, while nonblocking operations will return immediately, possibly before the outgoing message has actually been placed on the network.

In addition, the user should notice that there are no synchronous broadcasting operations. In order to make the broadcast operation as efficient as possible, it is possible that the messages are sent in a tree configuration, with the processors in the parallel system serving as the nodes in the tree. For large numbers of processors, this would lead to very high latencies associated with synchronous operations. For this reason, we have elected not

to provide synchronous broadcast operations.

There is a second ramification to a tree-structure broadcast implementation. That is polling operations are necessary, not only to receive incoming messages, but to know to forward messages for lower stages in the call tree. However, this behavior is not mandated by DMCS and is left to the individual implementation.

The DMCS broadcast API is given in Tables B.9 and B.10. Each broadcast operation is discussed in more detail below, where *block-type* denotes either "block" or "noblock".

dmcs_<*block-type*>_broadcast_rsr0()

Parameters:

1. **dmcs_rsr0_handler_t handler**: A pointer to a user-defined handler function which is to execute on all processors except the caller; no parameters other than the caller processor id is passed to this handler.
2. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Descrption:

This operation is used to invoke a user-defined handler on all processors except for the caller. The broadcast operation takes as a parameter the handler ot invoke

on each node. The user-defined handler must conform to the handler prototypes given in Table B.1. The user handler must be registered with the runtime system prior to its execution, using the DMCS handler registration routines described previously.

dmcs_<*block-type*>_broadcast_rsr1()

Parameters:

1. **dmcs_rsr1_handler_t handler**: A pointer to a user-defined handler function which is to execute on all processors except the caller. As parameters, this handler must accept the source id of the caller (integer) and a single machine-word sized argument (dmcs_arg_t).
2. **dmcs_arg_t arg**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
3. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This operation is used to invoke a user-defined handler on all processors except for the caller. The user handler invoked by the broadcast operation takes a single

machine-word sized argument as a parameter and must conform to the handler prototypes given in Table B.1. In addition, the user handler must be registered with the runtime system prior to its execution, using the DMCS handler registration routines described previously.

dmcs_<*block-type*>_broadcast_rsr2()

Parameters:

1. **dmcs_rsr2_handler_t handler**: A pointer to a user-defined handler function which is to execute on all nodes except the caller. As parameters, this handler will take the source id of the caller (integer), and two machine-word sized arguments (dmcs_arg_t).
2. **dmcs_arg_t arg1**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
3. **dmcs_arg_t arg2**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
4. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This operation is used to invoke a user-defined handler on all processors except for the caller. The user handler invoked by the broadcast operation takes two machine-word sized arguments as parameters and must conform to the handler prototypes given in Table B.1. In addition, the user handler must be registered with the runtime system prior to its execution, using the DMCS handler registration routines described previously.

dmcs_<*block-type*>_broadcast_rsr3()

Parameters:

1. **dmcs_rsr3_handler_t handler**: A pointer to a user-defined handler function which is to execute on all nodes except the caller. As parameters, this handler will take the source id of the caller (integer), and three machine-word sized arguments (dmcs_arg_t).
2. **dmcs_arg_t arg1**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
3. **dmcs_arg_t arg2**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
4. **dmcs_arg_t arg3**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
5. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a

certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This operation is used to invoke a user-defined handler on all processors except for the caller. The user handler invoked by the broadcast operation takes three machine-word sized arguments as parameters and must conform to the handler prototypes given in Table B.1. In addition, the user handler must be registered with the runtime system prior to its execution, using the DMCS handler registration routines described previously.

dmcs_<*block-type*>_broadcast_rsr4()

Parameters:

1. **dmcs_rsr4_handler_t handler**: A pointer to a user-defined handler function which is to execute on all nodes except the caller. As parameters, this handler will take the source id of the caller (integer), and four machine-word sized arguments (dmcs_arg_t).
2. **dmcs_arg_t arg1**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
3. **dmcs_arg_t arg2**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.

4. **dmcs_arg_t arg3**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
5. **dmcs_arg_t arg4**: A single machine-word sized argument that is passed as a parameter to the user-defined handler on each node.
6. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This operation is used to invoke a user-defined handler on all processors except for the caller. The user handler invoked by the broadcast operation takes four machine-word sized arguments as parameters and must conform to the handler prototypes given in Table B.1. In addition, the user handler must be registered with the runtime system prior to its execution, using the DMCS handler registration routines described previously.

dmcs_<*block-type*>_broadcast_rsrN()

Parameters:

1. **dmcs_rsrN_handler_t handler**: A pointer to a user-defined handler function which is to execute on all nodes except the caller. As parameters, this

handler will take the source id of the caller (integer), a user-defined parameter buffer of arbitrary length (dmcs_pointer_t), and the length of the parameter buffer in bytes (dmcs_arg_t).

2. **dmcs_pointer_t buffer**: A user-defined parameter data buffer of arbitrary length.
3. **int size**: The size of the parameter data buffer, in bytes.
4. **dmcs_status_t* status**: The status variable can be used to determine when it is safe to modify the parameter data buffer. The operations available to test the status object are described in this document. If a status object is not required, the constant *DMCS_STATUS_IGNORE* may be used. **NOTE**: This is a parameter for the *non-blocking* broadcast only!
5. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_DEFAULT_TAG. Tags should be positive integral values.

Returns: None

Description:

This operation is used to invoke a user-defined handler on all processors except for the caller. The user handler invoked by the broadcast operation takes an arbitrarily long user-defined parameter buffer as a parameter, and must conform

Table B.11: DMCS POLLING AND SYNCHRONIZATION

Polling Operation	
void **dmcs_poll()**	Polls the network for pending messages
Synchronization Operation	
void **dmcs_barrier()**	Blocks a node until all nodes enter

to the handler prototypes given in Table B.1. In addition, the user handler must be registered with the runtime system prior to its execution, using the DMCS handler registration routines described previously.

B.2.6 Polling Operations

Because DMCS is single-threaded, interrupts are not able to notify the application of new message arrivals. Therefore, the application must poll the network to check for any pending messages and to execute their associated handlers. The *dmcs_poll()* operation in Table B.11 is provided for this purpose.

How often the application chooses to post polling operations represents a tradeoff. Frequent polling will ensure the timely delivery of messages; however polling overhead may begin to take its toll on application performance. For this reason, applications must decide carefully when and where to post polling operations.

Polling will first check the network for any pending messages. For messages that have arrived since the previous poll, their associated user and system handlers will be executed in the order in which the messages have arrived and within the polling thread. This means that application developers writing single-threaded applications will be guaranteed that all application code, including message handlers, will execute within the same thread, eliminating the need for critical sections and mutex locks.

It should be noted that polling operations are necessary not only for RSR messages, but also for *get*, *put*, *get_op*, and *put_op* operations, even when no application handler function is executed.

dmcs_poll()

Parameters:

1. **int tag**: This is a user-specified tag field. Because all communication must be received using polling operations, the tag field can allow the poller to differentiate between message types, and only handle those types with a certain tag. A greater description is provided in the *dmcs_poll()* operation discussion. By default, this field has the value DMCS_ANY_TAG, which indicates that a message of any tag will be received. Tags should be positive integral values.

Returns: None

Description:

Polls the network for any newly arrived messages, and executes any user-defined handlers or other functionality associated with the messages. Note that the user may supply an optional *tag* parameter, which can be used to differentiate among incoming message tags. Posting a polling operation with a specific tag will ensure that only messages with matching tags will be handled. However, all messages of any type are received, and those with nonmatching tags are buffered for later handling. This buffering may impact memory utilization, and applications should take care to not exhaust available memory.

B.2.7 Synchronization Operations

The *dmcs_barrier()* operation provides a simple fan-in, fan-out barrier algorithm which will ensure that all processors enter the barrier before any processor is allowed to leave. There are several words of warning when using the barrier, however. First is that the *dmcs_barrier()* operation is designed to synchronize *all* processes. If only a subset of processes which to synchronize, the application will need to provide this functionality. However, this is relatively easy to build using the communication operations provided by DMCS[3].

Second, DMCS is unable to distinguish between one barrier call and another. Therefore, it is not necessary that all processors enter the *same* barrier, only that all processors enter *some* barrier. A corollary to this statement is that two barriers placed closely together will often lead to unpredictable behavior and should be avoided.

Finally, this barrier operation is not particularly high performance. Users desiring a more performance and a greater degree of scalability may wish to implement their own synchronization functionality.

dmcs_barrier()

Parameters: None

Returns: None

Description:

Blocks processes entering the barrier until all processes have entered the barrier.

[3]In fact, the *dmcs_barrier()* operation itself is built using DMCS communication operations.

Figure B.2: PING-PONG CODE USING DMCS

```
1   #include "dmcs.h"
2
3   bool ping_pong_ack = false;
4   bool finished = false;
5   int  num_messages = 1000;
6   int  payload_sizes[] = {16, 32, 64, 128, 256, 512, 1024};
7   int  num_payload_sizes = 7;
8
9   void finish_handler(int src_proc) { finished = true; }
10
11  void rsrN_reply(int src_proc, dmcs_pointer_t buffer, size_t size) {
12    ping_pong_ack = true;
13  }
14
15  void rsrN_handler(int src_proc, dmcs_pointer_t buffer, size_t size) {
16    dmcs_block_rsrN(src_proc, rsrN_reply, buffer, size);
17  }
18
19  int main(int argc, char* argv[]) {
20    dmcs_rsr0_handler_t rsr0_handlers[] = { finish_handler };
21    dmcs_rsrN_handler_t rsrN_handlers[] = { rsrN_handler, rsrN_reply };
22
23    dmcs_init(argc, argv);
24    int my_proc = dmcs_my_proc();
25    int num_proc = dmcs_num_procs();
26    dmcs_register_rsr0_handlers(rsr0_handlers, 1);
27    dmcs_register_rsrN_handlers(rsrN_handlers, 2);
28
29    if (my_proc == 0) {
30      for (int i = 0; i < num_payload_sizes; ++i) {
31        ping_pong_ack = false;
32        char* buffer = new char[payload_size[i]];
33        for (int j = 0; j < num_messages; ++j) {
34          dmcs_block_rsrN(1, rsrN_handler, buffer, payload_size[i]);
35          while (!ping_pong_ack) { dmcs_poll(); }
36          ping_pong_ack = false;
37        }
38        delete [] buffer;
39      }
40      for (int i = 0; i < num_procs; ++i) { dmcs_noblock_rsr0(i, finish_handler); }
41    }
42
43    while (!finished) { dmcs_poll(); }
44    dmcs_shutdown();
45    return 0;
46  }
```

B.3 Example DMCS Code

In this Section, we will take a look at some sample code snippets that illustrate in concrete terms how to use the DMCS library. The program shown in Figure B.2 is a simple ping-pong program that sends some number of DMCS *RSRN* messages from processor 0 to processor 1 and waits for responses back. This program needs only two processors to run; any other processors present will have no work to do. The header file *dmcs.h* is included on line 1; this is the only header file that needs to be included to make use of the DMCS functionality. Lines 3-7 declare some global variables that are used throughout the program, such as the

number of ping-pong messges that will be sent, and the size of the message payloads. The *done_flag* variable is used to signal when a single ping-pong message has completed the round-trip journey, and the *finished* flag is used to signal when the program is done.

Next, lines 9-17 declare the three handlers that will be used by this program. The *finish_handler()* routine is used to signal program completion and forms part of a motif that is used frequently in DMCS programs. Often, only a single processor is aware that a program has completed and is ready to exit. Many times this is the result of all data being gathered to a single processor for output, or some condition has occurred of which only one processor is aware. In this case, only processor 0 will know that all ping-pong messages have arrived. This processor will then send a shutdown message to all others, so all processors will be able to call *dmcs_shutdown()* at the same time. The other two handlers form the two ends of the ping-pong cycle: processor 0 will send a message to node 1 and invoke the *rsrN_handler()* routine, which will respond with a message to invoke the *rsrN_reply()* handler.

Line 19 begins the *main()* routine for the program. In lines 20 and 21, arrays are declared which contain the handlers we wish to register. There is a separate array for each handler type; in this case we have only *RSR0* and *RSRN* handlers. Line 23 calls *dmcs_init()*, passing the *argc* and *argv* parameters. We are then able to determine the local processor's rank and the number of processors in the parallel system. Lines 26 and 27 then register the handlers with the runtime system.

At line 29, the processors diverge, with processor 0 entering the conditional statement, and processor 1 proceeding directly to the *while-loop* on line 43. Processor 0 simply loops over all of the payload sizes and the number of messages and sends the first half of a

ping-pong to processor 1. Processor 0 then waits for the response in a *while-loop* on line 35. Because processor 1 is polling on line 43, it will receive the message and execute the *rsrN_handler()* routine, which was specified in the message. This routine will send a reply back to the caller, which processor 0 will receive in the polling loop on line 35. The *rsrN_reply()* routine will set the *done_flag* to false, which will break processor 0 out of the *while-loop* on line 35 and allow the process to begin again.

Once all ping-pong cycles have completed, processor 0 will send the completion message to all processors in the *for-loop* on line 40. All processors will receive this message within the polling operation on line 43, which will end the *while-loop* and allow each processor to shutdown DMCS and exit.

B.4 Porting DMCS

In order to make the DMCS runtime layer as easily portable as possible, we have isolated a small set of operations upon which all DMCS operations may be constructed. This layer, known as the DMCS Messaging Layer (DML), is completely isolated from the application and higher layers of the runtime system by the DMCS API software layer. Porting DMCS to a new communication platform involves providing only this small set of DML operations.

Each DML implementation should be kept in a separate subdirectory of the *dmcs/* directory (for instance, the DML/MPI implementation is found in *dmcs/dml/dml_mpi/*). Selecting which DML implementation to use is done using the *DML_DIR* macro provided in the *prema/config-defs* configuration file.

Each DML module must provide a Makefile that is responsible for compiling all DML

Table B.12: DML ENVIRONMENT OPERATIONS

System Initialization and Shutdown	
void **dml_init***(int argc, char* argv[])*	Initialize the DMCS system
void **dml_shutdown***()*	Shuts down the DMCS system
Profiling Information Printing	
void **dml_profiling_dump_info***(FILE* fp)*	Outputs profiling information
Querying the Environment	
int **dml_my_proc***()*	Returns the processor id of the caller
int **dml_num_procs***()*	Returns the number of processors in the system

source files. However, no library should be built from these source files and the resulting object files should be left in the *src* directory under the DML directory. The object files will be combined with the object files from the DMCS API layer into a single library.

Below, we will describe the operations that must be provided by the DML runtime layer, as well as the data structures that are used to convey information between the DMCS operations and the DML operations.

B.4.1 DML Operations

There are certain operations that must be provided in the DML layer; we will discuss those operations next. We will supplement this discussion with some example code snippets illustrating the operations' implementation using MPI. This may provide some insight to developers for how they may be implemented using other low-level communication substrates.

B.4.1.1 DML Environment Operations

Table B.12 contains the operations in the Ennironment section of the DML API. The operations closely mirror the operations found in Table B.2, which the exception that operations

used to register user-defined handlers are not present.

dml_init()

Parameters:

1. **int argc**: The number of command line arguments given to the executable from the command line and subsequently passed to the *main()* routine
2. **char* argv[]**: The command line arguments given to the executable from the command line and subsequently passed to the *main()* routine

Returns: void

Description:

This function is responsible for initializing the lower-level communication substrate, as well as setting any global variables that are needed by the DML runtime layer. In addition, it is up to the DML to decide whether or not user handlers must be registered; if so, then the system handlers *dmcs_free_remote_handler()*, *dmcs_malloc_remote_handler()*, and *dmcs_malloc_local_handler()* must be registered with the runtime system. This can be accomplished by inserting the handlers into the *_dmcs_handler_table* handler table. The *dml_init()* routine is only called from within the *dmcs_init()* routine, and never by the user application. The code snippet shown in Figure B.3 gives the MPI implementation of the *dml_init()* routine.

dml_shutdown()

Parameters: None

Figure B.3: MPI IMPLEMENTATION OF DML INITIALIZATION

```
1  void dml_init(int argc, char* argv[]) {
2    MPI_Init(&argc, &argv);
3    MPI_Comm_rank(MPI_COMM_WORLD, &_dml_my_proc);
4    MPI_Comm_size(MPI_COMM_WORLD, &_dml_num_procs);
5
6    // Create the data structures needed internally by the DML
7    _dml_request_avail_pool = new dml_request_avail_pool_t();
8    _dml_request_pending_pool = new dml_request_pending_pool_t();
9    _dml_pending_msg_map = new dml_pending_msg_map_t();
10
11   // Register the DMCS system handlers
12   _dmcs_handler_table->insert((void*)dmcs_free_remote_handler);
13   _dmcs_handler_table->insert((void*)dmcs_malloc_remote_handler);
14   _dmcs_handler_table->insert((void*)dmcs_malloc_local_handler);
15 }
```

Returns: None

Description:

This routine is used to shut down the low-level communication substrate. In addition, any data structures created by the DML must be cleaned up. This routine is called only by the DMCS shutdown procedure, and never by the user application.

dml_profiling_dump_info()

Parameters:

1. **FILE* fp**: A file pointer to a previously opened file; profiling information should be written to this file

Returns: None

Description:

The DML runtime layer is free to determine what system states should be recorded during profiling. The purpose of this routine is to output any gathered

runtime information. This routine is called only by the *dmcs_profiling_dump_info()* routine, and never by the user application.

dml_my_proc()

Parameters: None

Returns: Integer; the ID of the calling processor

Description:

This routine is used to determine the ID of the calling process. Processes are numbered according to the method used by the underlying communication system, but typically these IDs are in the range of 0 to $P - 1$, where P is the number of processes in the system. For small routines such as this, it may be advantageous for the implementation to make use of the C++ *inline* capability. Note that this routine is only called by the *dmcs_my_proc()* routine, and never by the user application.

dml_num_procs()

Parameters: None

Returns: Integer; the number of processes in the parallel system

Description:

Returns the number of processes in the parallel system. For small routines such as this, it may be advantageous for the implementation to make us of the C++ *inline* capability. Note that this routine is called only by the *dmcs_num_procs()* routine, and never by the user application.

Table B.13: DML MESSAGE SEND OPERATIONS

Send Operations - Without Handlers	
void **dml_block_send**(*int tgt, dmcs_message_t* msg*)	Send a DMCS Message object with blocking semantics
void **dml_noblock_send**(*int tgt, dmcs_message_t* msg*)	Send a DMCS Message object with nonblocking semantics
void **dml_sync_send**(*int tgt, dmcs_message_t* msg*)	Send a DMCS Message object with synchronous semantics
Send Operations - With Handlers	
void **dml_block_send**(*int tgt, dmcs_message_t* msg, void* rem_handler, void* loc_handler*)	Send a DMCS Message object with blocking semantics; user handlers specified
void **dml_noblock_send**(*int tgt, dmcs_message_t* msg, void* rem_handler, void* loc_handler*)	Send a DMCS Message object with nonblocking semantics; user handlers specified
void **dml_sync_send**(*int tgt, dmcs_message_t* msg, void* rem_handler, void* loc_handler*)	Send a DMCS Message object with synchronous semantics; user handlers specified

B.4.1.2 DML Send Operations

Table B.13 contains the operations used by DMCS to send all types of messages to remote processors. The DML send operations come in two types. The first is used to send simply a DMCS Message object, and is used for the *dmcs_sync_put()*, *dmcs_async_put()*, *dmcs_sync_get()*, and *dmcs_async_get()* operations; these do not invoke user-defined handlers on either the local or remote processors. The second type of send operation takes local and remote user-defined handler pointers as parameters (NULL is specified for any handler that is not applicable).

DMCS Message objects are allocated and filled in by the DMCS-level routines used to send messages, such as Remote Service Request operations. DMCS Message object deallocation is dependent upon the blocking/nonblocking/synchronous semantics of the operation. For blocking and synchronous operations, the DMCS Message object may be deallocated at the conclusion of the *send* operation. A code snippet shown with the *dmcs_sync_send()*

operation below shows how this deallocation must be performed. For nonblocking operations, deallocating the Message object from within the *send* operation is not possible. See below for a more detailed description of what must be done in this case.

Note that the example code below will make use of fields of the *dmcs_message_t* objects; a more complete description of this datatype can be found elsewhere in this document.

dml_block_send()

Parameters:

1. **int tgt**: The process ID to which the outgoing message should be sent
2. **dmcs_message_t* msg**: Pointer to a DMCS Message object that contains outgoing message data (the Message object is allocated and filled in by the DMCS API-layer routines)

Returns: None

Description:

This routine is used to send a DMCS Message object to a target processor using blocking semantics. This routine is used to send a Message to a target process that does not require the invocation of either a local or remote user-defined handler, such as with a DMCS *get* or *put* operation. This routine should not return until the Message object is placed onto the network; modifying the Message object once the routine has returned should not affect the communication operation. The DMCS Message object is allocated and filled in by the DMCS-level API operations. Note that this routine is called only by the DMCS operations, and never by the user application.

dml_noblock_send()

Parameters:

1. **int tgt**: The process ID to which the outgoing message should be sent
2. **dmcs_message_t* msg**: Pointer to a DMCS Message object that contains outgoing message data (the Message object is allocated and filled in by the DMCS API-layer routines)

Returns: None

Description:

This routine is used to send a DMCS Message object to a target processor using nonblocking semantics. This routine is used to send a Message to a target process that does not require the invocation of either a local or remote user-defined handler, such as with a DMCS *get* or *put* operation. This routine may return before the communication operation is complete, and the user should not modify the DMCS Message object until it can be guaranteed that it is safe to do so. The DMCS Message object is allocated and filled in by the DMCS-level API operations. Note that this routine is called only by the DMCS operations, and never by the user application.

Because this operation is nonblocking, the implementation must provide a method to determine when it is safe to deallocate the DMCS Message object. The code shown in Figure B.4 shows the nonblocking send operation in the MPI implementation of the DML.

dml_sync_send()

Parameters:

1. **int tgt**: The process ID to which the outgoing message should be sent
2. **dmcs_message_t* msg**: Pointer to a DMCS Message object that contains outgoing message data (the Message object is allocated and filled in by the DMCS API-layer routines)

Returns: None

Description:

This routine is used to send a DMCS Message object to a target processor using synchronous semantics. This routine is used to send a Message to a target process that does not require the invocation of either a local or remote user-defined handler, such as with a DMCS *get* or *put* operation. This routine will not return until the message has been received on the target processor. The DMCS Message object is allocated and filled in by the DMCS-level API operations. Note that this routine is called only by the DMCS operations, and never by the user application. In Figure B.5, we show a code snippet that illustrates the implementation of this operation using MPI. In it, the DMCS Message object is deallocated once the operation is finished; this same strategy is applicable to the DML send operations with blocking semantics.

dml_block_send()

Parameters:

1. **int tgt**: The process ID to which the outgoing message should be sent

2. *dmcs_message_t* msg*: Pointer to a DMCS Message object that contains outgoing message data (the Message object is allocated and filled in by the DMCS API-layer routines)
3. *void* rem_handler*: A pointer to a user-defined handler function to be executed on the remote processor
4. *void* loc_handler*: A pointer to a user-defined handler function to be executed on the local processor

Returns: None

Description:

This routine is used to send a DMCS Message object to a target processor using blocking semantics. The DML layer has the option of whether or not to use user-handler tables, so the handler pointers passed to this routine may be looked up in the DMCS handler table, or inserted into the Message object as-is. More information concerning the DMCS handler tables as well as the DMCS Message object fields can be found elsewhere in this document.

dml_nonblock_send()

Parameters:

1. **int tgt**: The process ID to which the outgoing message should be sent
2. *dmcs_message_t* msg*: Pointer to a DMCS Message object that contains outgoing message data (the Message object is allocated and filled in by the DMCS API-layer routines)

3. *void* rem_handler*: A pointer to a user-defined handler function to be executed on the remote processor
4. *void* loc_handler*: A pointer to a user-defined handler function to be executed on the local processor

Returns: None

Description:

This routine is used to send a DMCS Message object to a target processor using nonblocking semantics. The DML layer has the option of whether or not to use user-handler tables, so the handler pointers passed to this routine may be looked up in the DMCS handler table, or inserted into the Message object as-is. More information concerning the DMCS handler tables as well as the DMCS Message object fields can be found elsewhere in this document.

dml_sync_send()

Parameters:

1. **int tgt**: The process ID to which the outgoing message should be sent
2. *dmcs_message_t* msg*: Pointer to a DMCS Message object that contains outgoing message data (the Message object is allocated and filled in by the DMCS API-layer routines)
3. *void* rem_handler*: A pointer to a user-defined handler function to be executed on the remote processor
4. *void* loc_handler*: A pointer to a user-defined handler function to be executed on the local processor

Figure B.4: MPI IMPLEMENTATION OF DML NONBLOCKING SEND

```
1  void dml_noblock_send(int tgt, dmcs_message_t* msg) {
2    // For the message header, we get the MPI Request object from the pool of
3    // preallocated reqeuests.  We will check (and free) the completion of
4    // these communication operations in the dml_poll() operation.
5    MPI_Request* msg_send_req = _dml_request_avail_pool->remove();
6    MPI_Request buf_send_req;
7    int tag = 0;  // Arbitrary
8
9    MPI_Isend(msg, sizeof(dmcs_message_t), MPI_BYTE, tgt, tag,
10             MPI_COMM_WORLD, msg_send_req);
11   if (msg->loc_ptr != NULL) {
12     MPI_Isend(msg->loc_ptr, msg->size, MPI_BYTE, tgt, tag,
13               MPI_COMM_WORLD, &buf_send_req);
14   }
15
16   // Insert the header's MPI Request object into the pending request pool,
17   // and insert the request/message pair into the pending message map.
18   // These data structures are emptied in the DML polling operation.
19   _dml_request_pending_pool->insert(msg_send_req);
20   _dml_pending_msg_map->insert(msg_send_req, msg);
21 }
```

Returns: None

Description:

This routine is used to send a DMCS Message object to a target processor using synchronous semantics. The DML layer has the option of whether or not to use user-handler tables, so the handler pointers passed to this routine may be looked up in the DMCS handler table, or inserted into the Message object as-is. More information concerning the DMCS handler tables as well as the DMCS Message object fields can be found elsewhere in this document.

B.4.1.3 DML Broadcast Operations

DML broadcast operations trasmit DMCS Message objects and user data buffers to all nodes in the parallel system except the caller. If the underlying communication substrate provides

Figure B.5: MPI IMPLEMENTATION OF DML SYNCHRONOUS SEND

```
1  dmcs_sync_send(int tgt, dmcs_message_t* msg) {
2    MPI_Request msg_send_req;
3    int tag = 0;  // Arbitrary
4
5    // If the payload buffer is not NULL, then we will first send the header
6    // information using nonblocking semantics, and then send the payload
7    // data using synchronous semantics.  Otherwise, we will send the header
8    // information using synchronous MPI operations.
9    if (msg->loc_ptr != NULL) {
10     MPI_Isend(msg, sizeof(dmcs_message_t), MPI_BYTE, tgt, tag,
11               MPI_COMM_WORLD, &msg_send_req);
12     MPI_Ssend(msg->loc_ptr, msg->size, MPI_BYTE, tgt, tag, MPI_COMM_WORLD);
13     MPI_Request_free(&msg_send_req);
14   } else {
15     MPI_Ssend(msg, sizeof(dmcs_message_t), MPI_BYTE, tgt, tag,
16               MPI_COMM_WORLD);
17   }
18
19   // Since this is a synchronous operation, we can safely return the DMCS
20   // Message object to the pool of available objects
21   _dmcs_message_pool->insert(msg);
22 }
```

Table B.14: DML BROADCAST OPERATIONS

Broadcast Operations	
void **dml_block_broadcast**(*dmcs_message_t* msg, void* handler)*	Broadcast a DMCS message with blocking semantics
void **dml_noblock_broadcast**(*dmcs_message_t* msg, void* handler)*	Broadcast a DMCS message with nonblocking semantics

efficient broadcast operations conforming to the semanitics mandated by DMCS, then the implementor may feel free to use them. However, in the MPI version of DMCS, broadcasts are built using only point-to-point communication operations, and messages travel to all nodes in a tree-like fashion. Child nodes are dynamically calculated in such a way that each messaage arrives at each node exactly one time.

The broadcast operations that must be provided by the DML runtime layer are shown in Table B.14. It should be noted that there is no synchronous broadcast available.

dml_block_broadcast()

Parameters:

1. **dmcs_message_t* msg**: DMCS Message object, filled in by the DMCS broadcast operation.
2. **void* handler**: Pointer to the user-defined handler to execute, cast as a generic pointer.

Returns: None

Description:

This routine is used to invoke the user defined handler on all nodes in the parallel system, except the caller. Blocking semantics mean that the routine will not return until it is safe to modify the outgoing data buffer. However, the latency observable by the application will most likely be higher with this routine than with its nonblocking counterpart.

dml_noblock_broadcast()

Parameters:

1. **dmcs_message_t* msg**: DMCS Message object, filled in by the DMCS broadcast operation.
2. **void* handler**: Pointer to the user-defined handler to execute, cast as a generic pointer.

Returns: None

Table B.15: DML POLLING AND SYNCHRONIZATION OPERATIONS

Synchronization Operations	
void **dml_poll()**	Polls the network for pending messages
Polling Operations	
void **dml_barrier()**	Blocks until all processors enter barrier

Description:

This routine is used to invoke the user defined handler on all nodes in the parallel system, except the caller. Nonblocking semantics mean that the routine may possibly return before it is safe to modify the outgoing data buffer. The caller must take measures to ensure that the data buffer will not be deallocated or modified until it is safe to do so. However, the latency observable by the application will most likely be lower with this routine than with its blocking counterpart.

B.4.1.4 DML Polling Operation

The purpose of the DML polling operation (Table B.15) is to encapsulate the system-specific functionality that is required to poll for messages. Note that the *dml_poll()* routine is called only by *dmcs_poll()*, and never by the user application.

dml_poll()

Parameters: None

Returns: None

Description:

The DML polling operation encapsulates any low-level system-specific functionality needed to poll the network for incoming messages. Incomming messages

contain tags that state the type of arriving message (more information concerning the DMCS Message object is found later in this document); these messages should be properly handled in the DML runtime layer.

NOTE: The DMCS polling operation simply calls the DML poll routine, so all polling activity must be handled in the DML runtime layer.

B.4.1.5 DML Synchronization Operation

Table B.15 contains the DML synchronization operation. While it would be possible to build a barrier operation using the RSR functionality provided by DMCS, we have chosen to place the barrier within the DML runtime layer in order to take advantage of any higher performance barrier functionality that may be provided by the lower communication substrate.

dml_barrier()

Parameters: None

Returns: None

Description:

Blocks processes entering the barrier until all processes have entered the barrier.

B.4.2 DML Data Structures

Several data structures are used to transfer information between the DMCS API routines and the underlying DML operations upon which they are constructed. In this Section, we

Figure B.6: DMCS MESSAGE DATA STRUCTURE

```
1  struct dmcs_message_t {
2    // The process that sent the message
3    int src;
4
5    // The type of message:  RSR, PUT, GET, PUT_OP, GET_OP.  The tag field
6    // is used for specific message types:  RSR1, RSR2, RSR3...
7    int tag;
8    int type;
9
10   // Array to hold the arguments available for RSR operations
11   dmcs_arg_t arg[4];
12
13   // Handler table index of the local and remote handlers.  The local
14   // handler always executes on the originating (src) processor, while
15   // the remote handler executes on the target node.
16   int        loc_handler_idx; // May also be a function address
17   dmcs_arg_t loc_handler_arg;
18   int        rem_handler_idx; // May also be a function address
19   dmcs_arg_t rem_handler_arg;
20
21   // The pointers to the local and remote data buffers used in the Get-op
22   // and Put-op operations
23   dmcs_pointer_t rem_ptr;
24   dmcs_pointer_t loc_ptr;
25
26   // The size of the data buffer to transfer; used for both RSRN and Put-op
27   // and Get-op operations
28   int size;
29
30   // The sync flag address is used to signal a processor that a particular
31   // synchronous operation has completed.
32   bool* sync_flag_addr;
33 };
```

will describe these data structures and how they may be used to implement DML-level routines.

B.4.2.1 dmcs_message_t

The *dmcs_message_t* data structure (Figure B.6) contains information about a particular DMCS message, such as the sender of the message and the message type (RSR, put, get, etc.).

The *dmcs_message_t* object is filled in by the DMCS-level message send operations, such

Figure B.7: A PORTION OF THE DML POLLING IMPLEMENTATION

```
while (probe_flag) {
  probe_flag = false;

  // Probe each processor for incoming messages
  for (int i = 0; i < num_procs; ++i) {
    dmcs_message_t* msg_ptr = dml_probe(i);
    if (msg_ptr != NULL) {
      probe_flag = true;
      switch (msg_ptr->tag) {
        case DMCS_RSR: dml_rsr_handler(msg_ptr); break;
        case DMCS_PUT: dml_put_handler(msg_ptr); break;
        case DMCS_GET: dml_get_handler(msg_ptr); break;
        case DMCS_PUT_OP: dml_put_op_handler(msg_ptr); break;
        case DMCS_GET_OP: dml_get_op_handler(msg_ptr); break;
      } // end switch
    } // end if
  } // end for
} // end while
```

as *dmcs_async_rsrN()*. By examining the contents of particular fields, DML routines are able to determine how to properly handle outgoing and incoming messages. For instance, DML *send* operations look at the *loc_ptr* field to determine the location of any user-defined parameter data buffers, as may be present with DMCS *put* and *rsr* operations. Examining the *size* field gives the number of bytes to transfer to the remote processor. At the target processor, the DML polling operation is able to determine how to handle the incoming message via the *tag* and *type* fields.

As a concrete example, the code in Figure B.7 illustrates part of the polling functionality found in the MPI implementation of the DML (in a somewhat shorthand notation). The code examines the incoming DMCS message object's *tag* field to determine what type of message is arriving. A separate function is called to handle each type of message.

In this code example, the *dml_probe()* routine is used to check for an arriving message from a single processor. If a message is present, a new DMCS Message object is allocated

Table B.16: DMCS MESSAGE POOL INTERFACE METHODS

*dmcs_message_t** **remove**()	Allocate a DMCS Message object from the pool
void **insert**(*dmcs_messaage_t** *msg_ptr*)	Return a DMCS Message object to the pool for future reuse

from the DMCS Message object pool (discussed below) and is filled in with the arriving data. If no incoming message is present, NULL is returned. Once the message is handled, the DMCS Message object is returned to the available Message object pool.

B.4.2.2 dmcs_message_pool_t

In order to eliminate the dynamic allocation and deallocation of DMCS Message object during runtime, a pool of available Message objects is maintained by the runtime system. When creating and sending an outgoing message, the DMCS API-level routines will request Message objects from this pool and fill them in before passing them to the DML-level *send* operations. It is then up to the DML routines to deallocate the Message objects and return them to the preallocated pool. On the target node, it is up to the DML layer to entirely manage the allocation and deallocation of DMCS Message objects.

Inside the *dmcs_init()* routine, a global DMCS Message pool object is created. The *_dmcs_message_pool* object has two public methods, one for inserting objects into the pool (deallocation) and one for removing items from the pool (allocation). The *remove()* method will return a pointer to a DMCS Message object (call it *msg_ptr*), while *insert(msg_ptr)* will return the DMCS Message object to the pool, making it available for future reuse.

Note that in order to access the global DMCS Message pool and its methods, the header files *dmcs_globals.h* and *dmcs_message_pool.h* need to be included.

Table B.17: DMCS HANDLER TABLE INTERFACE METHODS

void **insert***(void* handler)*	Insert a user-defined handler into the table
int **lookup_idx***(void* handler)*	Given a handler pointer, return its table index
*void** **lookup_handler***(int idx)*	Given a handler index, return its address
void **remove***(int idx)*	Remove a user-defined handler from the table

B.4.2.3 dmcs_handler_table_t

The third data structure we will discuss is the DMCS handler table, which is used to register user-defined handlers prior to their invocation. Many parallel environments, such as loosely-coupled networks of workstations, are not able to guarantee that routines are mapped to corresponding addresses on each node. For these environments, user-defined handlers are associated with small integer indices, which are then passed between processors instead of routine addresses. However, some environments may be able to pass memory address of functions safely between processors, and will therefore not need to make use of the handler tables. Whether or not to make use of the handler tables is therefore a decision left to the DML layer of the runtime system.

Table B.17 defines the interface to the DMCS handler table. A global handler table (*_dmcs_handler_table*) is created inside the *dmcs_init()* routine. In addition, operations are provided by the DMCS API-level to register user-defined handlers with the handler table. However, in the DML *send* operations, user handlers may need to be looked up, thus requiring the use of the handler table. In such cases, transforming table indices to handler addresses will be necessary on the target node. Note that in order to access the global handler table and its methods, the header files *dmcs_globals.h* and *dmcs_header_table.h* must be included.

Appendix C

Mobile Object Layer

The Mobile Object Layer is a light-weight toolkit which provides a global namespace and the mechanisms necessary for object migration. The MOL provides a global namespace through the concept of *mobile pointers* which refer to application-defined data objects regardless of where they are in the parallel system. The MOL provides an efficient migration mechanism which allow these *mobile objects* to migrate under the control of either the application or higher-level runtime libraries; this migration mechanism makes use of a distributed data structure and automatic message forwarding to efficiently migrate data and ensure that processors are able to communicate with application data objects. This mechanism means that the MOL frees application developers from the tedious bookkeeping of maintaining up-to-date locations for all data objects in the application domain, greatly reducing the complexity of parallel adaptive applications.

The MOL extends the communication model provided by DMCS [16] by providing operations for communication between processors and mobile objects, not just between processors. This allows applications to invoke computation at the site of data, without needing to know explicitly where that data lies; it is up to the MOL to ensure that messages arrive at their intended targets.

Table C.1: MOL USER HANDLER PROTOTYPES

MOL Request Handler Prototype	
*void (***mol_req_handler_t**)(int proc, void* data, int size, void* arg)*	MOL Request handler prototype
MOL Message Handler Prototype	
*void (***mol_msg_handler_t**)(int proc, mol_mobile_ptr_t mptr, void* obj_data, void* user_data, int size, void* arg)*	MOL Message handler prototype

The MOL is built entirely using the communication operations provided by DMCS, meaning that the MOL is immediately portable to any platform for which a DMCS implementation exists. The MOL is also written in ANSI-C++, further easing the burden of porting the runtime system to new platforms.

The MOL is not intended to provide a fully Distributed Shared Memory programming model. By this, it is meant that processors may not access distributed data using semantics which are inherent to sequential programming. Processors access data through *messages*, which will be described more fully later.

In addition, the MOL does not allow copies of data to exist. This allows the MOL to avoid the issue of maintaining object consistency and lowers the cost associated with the runtime system. This does not prevent the application from explicitly creating copies of data objects, but the coherency concerns are left to the application.

C.1 User Handler Prototypes

The Mobile Object Layer provides two basic types of communication operations: requests and messages. These operations are designed to execute a user-defined handler at the target. These handlers must conform to specific prototypes, which are given in Table C.1.

C.1.1 Request Handlers

MOL Request handlers execute within a *mol_poll()* operation, are sent from one processor to another, and are described by the following:

Type Name: **mol_req_handler_t**

Parameters:

1. **int proc**: This is the processor that sent the request, and is often useful for sending replies back to the caller from within the handler.
2. **void* data**: This is the user-specified parameter data buffer. This buffer may be of any size, and may possibly be NULL.
3. **int size**: This argument is the size of the parameter data buffer, in bytes.
4. **void* arg**: This argument is the single machine-word sized argument that was specified as the final requred parameter in the *mol_request()* operation. Again, this value can be NULL.

Returns: None

C.1.2 Message Handlers

MOL Message handlers also execute within a *mol_poll()* operation. However, these are sent from a processor to a mobile object whose location does not need to be known to the caller. The message will be forwarded if necessary in order to reach its intended target. MOL Message handlers are described by the following:

Type Name: **mol_msg_handler_t**

Parameters:

1. **int proc**: This is the processor that sent the message, and is often useful for sending replies back to the caller from within the handler.
2. **mol_mobile_ptr_t mptr**: This is the mobile pointer that is the target of the message.
3. **void* obj_data**: This is a pointer to the local mobile object. Because MOL message handlers always execute on the processor on which the target mobile object is located, we know that a local pointer to the data object is available. This is provided so that the handler will have access to the mobile object.
4. **void* user_data**: This is the user-specified parameter data buffer. This buffer may be of any size, and may possibly be NULL.
5. **int size**: This argument is the size of the parameter data buffer, in bytes.
6. **void* arg**: This argument is the single machine-word sized argument that was specified as the final required parameter in the *mol_message()* operation. Again, this value may be NULL.

Returns: None

C.2 Operations Provided

Now that we have provided a brief introduction to the Mobile Object Layer and its programming model, we can delve into the specifics of the operations that are provided. These operations can be divided five categories: Environment functions, Mobile Pointer functions, Communication functions, Polling functions, and Synchronization functions.

Table C.2: MOL ENVIRONMENT OPERATIONS

System Initialization and Shutdown	
void **mol_init***(int argc, char* argv[], mol_configurator_t* config)*	Initialize the MOL system
void **mol_shutdown***()*	Shuts down the MOL system
System Configuration	
void **mol_set_configurator_field***(mol_configurator_t* config, int field, int value)*	Sets a field in the configurator; see below for comlete description
Querying the Environment	
int **mol_my_proc***()*	Returns the processor id of the caller
int **mol_num_procs***()*	Returns the number of processors in the system
Handler Registration	
void **mol_register_req_handlers***(mol_req_handler_t handlers[], int size)*	Register MOL Request handlers with no associated names
void **mol_register_named_req_handlers***(mol_req_handler_t handlers[], char* names[] int size)*	Register MOL Request handlers with associated names
void **mol_register_msg_handlers***(mol_msg_handler_t handlers[], int size)*	Register MOL Message handlers with no associated names
void **mol_register_named_msg_handlers***(mol_msg_handler_t handlers[], char* names[] int size)*	Register MOL Message handlers with associated names

C.2.1 Environment Operations

Table C.2 contains the operations in the Environment section of the MOL API. These functions are responsible for initializing and shutting down the runtime system, along with determining certain runtime information.

mol_init()

Parameters:

1. **int argc**: The number of command line parameters. Passed to *main()* as the first parameter.
2. **char* argv[]**: The command line parameters given to the program. Passed to *main()* as the second parameter.

3. **mol_configurator_t* config**: *OPTIONAL* This parameter may be left out of the *mol_init()* call. There are three fields in the configurator object using the *mol_set_configurator_field()* routine:

   - MOL_CONFIGURATOR_POOL_SIZE : The size of the buffer entries controlled by the MOL's memory manager
   - MOL_CONFIGURATOR_POOL_ENTRIES : The number of buffers managed by the MOL's memory manager
   - MOL_CONFIGURATOR_DIR_UPDATE_PROTOCOL : The directory update protocol to use. By default, the MOL uses a lazy directory update protocol (MOL_LAZY_FORWARDING). However, the MOL is designed to allow alternative protocols to be easily incorporated into the architecture[1].

Returns: None

Description:

This function is responsible for initializing the runtime system and must be the first MOL operation called (except for the *mol_set_configurator_field()* operation described below). The MOL is responsible for initializing the underlying DMCS layer, so a *dmcs_init()* call is no longer necessary in the application code. This operation is collective, meaning that all processors must call *mol_init()* at the same time. After this function returns, other MOL operations may be invoked.

mol_shutdown()

[1]Descriptions of how to create and incorporate directory update protocols is beyond the scope of this document.

Parameters: None

Returns: None

Description:

This operation is the final MOL call made by any application. As with initialization, the MOL is responsible for shutting down any lower level communication systems, so such calls should not be present in the application. In addition, the shutdown routine is a collective operation and must be called by each processor at the same time.

NOTE: In the case in which runtime profiling information is gathered (Refer to *Compiling and Installing the PREMA Libraries*), calling *mol_shutdown()* will result in profiling files being generated for the MOL and lower PREMA software layers.

mol_set_configurator_field()

Parameters:

1. **mol_configurator_t* configurator**: Pointer to a preallocated configurator object.
2. **int field**: Which field of the configurator object to set. These fields are specified in the description of the *mol_init()* operation.
3. **int value**: The value to set the configurator field to. The pool size field is set to an integer value which specifies the size of the pool buffers in bytes. The number of pool entries field is an integer value which specifies the number of buffers managed for each *source-target* processor pair. The directory

Figure C.1: EXAMPLE USING MOL CONFIGURATOR OBJECT

```
1   int main(int argc, char* argv[]) {
2     mol_configurator_t config;
3
4     mol_set_configurator_field(&config, MOL_CONFIGURATOR_POOL_SIZE, 1024);
5     mol_set_configurator_field(&config, MOL_CONFIGURATOR_POOL_ENTRIES, 256);
6     mol_init(argc, argv, &config);
7
8     ...
9
10    return 0;
11  }
```

update protocol field is set to an integer flag which will specify which protocol to use. Any protocols provided by the MOL have constants defined in *mol_constants.h*; however, the user is free to create new directory upate protocols.

Returns: None

Description:

Sets a single field of an MOL configurator object. Configurator objects may be passed as an optional third parameter to the *mol_init()* operation in order to configure the Mobile Object Layer. Not all fields of the configurator object need to be set; default values are provided for each of the fields. The code in Figure C.1 is an example of how to use configurator objects:

mol_my_proc()

Parameters: None

Returns: Integer processor id

Description:

Returns a value between 0 and $N-1$ where N is the number of processors in the parallel system. Although the exact numbering of processors depends on the lowest-level communication substrate (such as MPI), it is generally assumed that the processor ids begin at zero and proceed sequentially.

mol_num_procs()

Parameters: None

Returns: Integer number of processors

Description:

Returns the number of processors in the parallel system, N. Although the numbering of processors in the parallel system is dependent upon the lowest-level communication substrate, processors are typically numbered from zero to $N-1$.

C.2.2 Handler Registration Operations

Table C.2 contains the routines that are used to register application-defined MOL message and request handlers with the runtime system. All message handlers must be registered with the runtime system before they may be invoked as the result of a message from a remote processor. Handler registration must be performed by each processor, with every processor registering handlers in the same order. Typically this is done immediately after initializing the system.

mol_register_req_handlers()

Parameters:

1. **mol_req_handler_t handlers[]**: An array of MOL request handlers to be registered with the runtime system. The prototype for the request handlers is discussed in Section C.1.
2. **int size**: The number of handlers in the handlers array.

Returns: None

Description

This function is used to register MOL Request handlers with the runtime system. Because the MOL is unable to make the assumption that functions lie at the same address on each processor in the parallel system, requests are sent between processors specifying a table index which processors use to locate user-defined handlers. With this function, no names are associated with the handlers; associating names with handlers can be useful for application debugging with the *MOL_OUTPUT_USER_HANDLERS* flag provided during compilation.

mol_register_named_req_handlers()

Parameters:

1. **mol_req_handler_t handlers[]**: An array of MOL request handlers to be registered with the runtime system. The prototype for the request handlers is discussed in Section C.1.
2. **char* names[]**: An array of strings used to identify the handlers. These names may be the names of the handlers, or any other string the application developer can use for identification purposes.

3. **int size**: The number of entries in each of the handlers array and names array. Note that these two arrays must be of the same size.

Returns: None

Description:

This function is used to register MOL Request handlers with the runtime system. Because the MOL is unable to make the assumption that functions lie at the same address on each processor in the parallel system, requests are sent between processors specifying a table index which processors use to locate user-defined handlers. With this function, the application is able to associate names with each of the request handlers. These names may be useful during application debugging, provided the *MOL_OUTPUT_USER_HANDLERS* flag is specified on the compile line when compiling the MOL. The name associated with each handler will be output to *stderr* when it is entered and when it exits during the application run.

mol_register_msg_handlers()

Parameters:

1. **mol_msg_handler_t handlers[]**: An array of MOL message handlers to be registered with the runtime system. The prototype for the message handlers is discussed in Section C.1.
2. **int size**: The number of handlers in the handlers array.

Returns: None

Description:

This function is used to register MOL Message handlers with the runtime system. Because the MOL is unable to make the assumption that functions lie at the same address on each processor in the parallel system, messages are sent between processors specifying a table index which processors use to locate user-defined handlers. With this function, no names are associated with the handlers; associating names with handlers can be useful for application debugging with the *MOL_OUTPUT_USER_HANDLERS* flag provided during compilation.

mol_register_named_msg_handlers()

Parameters:

1. **mol_msg_handler_t handlers[]**: An array of MOL message handlers to be registered with the runtime system. The prototype for the request handlers is discussed in Section C.1.
2. **char* names[]**: An array of strings used to identify the handlers. These names may be the names of the handlers, or any other string the application developer can use for identification purposes.
3. **int size**: The number of entries in each of the handlers array and names array. Note that these two arrays must be of the same size.

Returns: None

Description:

This function is used to register MOL Message handlers with the runtime system. Because the MOL is unable to make the assumption that functions lie at the

Table C.3: MOL MOBILE POINTER OPERATIONS

Mobile Object Creation	
mol_mobile_ptr_t **mol_create_mobile_ptr(** *void* object)*	Create a mobile pointer given a local pointer
Mobile Pointer Dereferencing	
*void** **mol_mobile_ptr_deref(** *mol_mobile_ptr_t mptr)*	Return the address of the data referenced by the mobile pointer (if local)
Miscellaneous Mobile Pointer Operations	
int **mol_mobile_ptr_locate(** *mol_mobile_ptr_t mptr)*	Returns the processor on which a mobile object is located (Please read the description of this operation for a complete discussion)
int **mol_num_local_objs()**	Returns the number of mobile objects on the local processor

same address on each processor in the parallel system, messages are sent between processors specifying a table index which processors use to locate user-defined handlers. With this function, the application is able to associate names with each of the message handlers. These names may be useful during application debugging, provided the *MOL_OUTPUT_USER_HANDLERS* flag is specified on the compile line when compiling the MOL. The name assocated with each handler will be output to *stderr* when it is entered and when it exits during the application run.

C.2.3 Mobile Pointer Operations

The MOL mobile pointer is a system-wide unique identifier that refers to any application-defined data object. The data object is not restricted to lie in contiguous memory. The operations discussed below are used to create and dereference mobile objects.

mol_create_mobile_ptr()

Parameters:

1. **void* object**: Pointer to local data object to which the mobile pointer will refer

Returns: mol_mobile_ptr_t Mobile pointer which refers to the data object

Description:

This operation is used to create a mobile pointer which refers to an application-specified data object. The application data object must be local to the processor calling the routine. Once this operation returns, the mobile pointer is known only to the calling processor. However, mobile pointers may be distributed to other processors using communication routines provided by the MOL, DMCS, or lower communication substrate.

mol_mobile_ptr_deref()

Parameters:

1. **mol_mobile_ptr_t mptr**: The mobile pointer to dereference

Returns: void* address of data object. This is a valid address for local objects, and NULL for remote objects

Description:

This operation is used to "dereference" a mobile pointer. For mobile pointers that refer to local data objects, this routine will return the local address of the data object. For mobile pointers which refer to remote data objects, this routine will return NULL. This operation is often used for optimizations; the application can first check to see if the data object is local, and if so can operate on it directly.

mol_mobile_ptr_locate()

Parameters:

1. **mol_mobile_ptr_t mptr**: The mobile pointer for the object to locate

Returns: Integer processor id

Description:

This routine is used to locate a mobile object. However, it must be noted that the MOL uses a distributed directory structure to maintain mobile object locations. Using some directory update protocols (and by default), local directories are updated only lazily and are allowed to be out of date. In order to make this operation as efficient as possible, mobile objects are looked up on the local directory only, so the value returned to the user may reflect the out of date status of the local directory.

mol_num_local_objs()

Parameters: None

Returns: Integer value; the number of local objects

Description:

This routine returns the number of mobile objects that are currently located on the calling processor.

C.2.4 Communication Operations

The MOL provides two types of communication operations: *requests* and *messages*. Applications send requests from one processor to another, while messages are sent to application-defined mobile objects. These operations are shown in Table C.4.

Table C.4: MOL COMMUNICATION OPERATIONS

MOL Request Operations	
void **mol_request***(int proc, mol_req_handler_t handler, void* data, int size, void* arg, mol_status_t* status)*	Sends a request from one processor to another
void **mol_request_no_copy***(int proc, mol_req_handler_t handler, void* data, int size, void* arg, mol_status_t* status)*	Sends a request from one processor to another without copying the data on the sender node
MOL Message Operations	
void **mol_message***(mol_mobile_ptr_t mptr, mol_msg_handler_t handler, void* data, int size, void* arg, mol_status_t* status)*	Sends a message to a mobile object
void **mol_message_no_copy***(mol_mobile_ptr_t mptr, mol_msg_handler_t handler, void* data, int size, void* arg, mol_status_t* status)*	Sends a message to a mobile object without copying the data on the sender node
MOL Status Object Operations	
int **mol_status_test***(mol_status_t* status)*	Tests for associated operation's status
void **mol_status_wait***(mol_status_t* status)*	Waits for associated operation's completion
void **mol_status_reset***(mol_status_t* status)*	Resets the status object for subsequent reuse

Figure C.2: USING THE MOL STATUS OBJECT

```
1   mol_status_t status;  // Object is initialized to the correct state
2   mol_request_no_copy(tgt_proc, user_handler, data_buffer, size_of_buffer, NULL,
3                       &status);
4   while (mol_status_test(&status) != MOL_STATUS_COMPLETE) {
5      // Do some work here; do not modify data_buffer!
6   }
7   mol_status_reset(&status);  // status now available for subsequent operations
```

MOL communication operations make use of a *status* object, which allows applications to monitor the state of ongoing communication operations. The status object is particularly useful for requests and messages sent using the *no_copy* option; for larger messages, split-phase communication is used to lower the latency visible to the application. However, the application must have a mechanism for determining when it is safe to modify the data buffer used in the communication operation. Status variables provide this mechanism. In the code snippet shown in Figure C.2, we demonstrate the use of the status object.

mol_request()

Parameters:

1. **int proc**: The processor to which the request will be sent
2. **mol_req_handler_t handler**: The user handler to invoke on the target processor
3. **void* data**: Parameter data buffer; this buffer will be provided to the user handler
4. **int size**: The size of the parameter data buffer, in bytes
5. **void* arg**: This is a single machine-word sized argument, and will be passed to the user handler
6. **mol_status_t* status**: *OPTIONAL* This parameter may be left out of the *mol_request()* call.

Returns: None

Description:

Sends a request to a remote processor. The data buffer is copied in order to construct an outgoing message and is therefore available for modification as soon as this operation returns. However, for large data sizes, this copy can have an adverse impact on latency and overall performance.

mol_request_no_copy()

Parameters:

1. **int proc**: The processor to which the request will be sent

2. **mol_req_handler_t handler**: The user handler to invoke on the target processor
3. **void* data**: Parameter data buffer; this buffer will be provided to the user handler
4. **int size**: The size of the parameter data buffer, in bytes
5. **void* arg**: This is a single machine-word sized argument, and will be passed to the user handler
6. **mol_status_t* status**: *OPTIONAL* This parameter may be left out of the *mol_request()* call.

Returns: None

Description:

Sends a request to a remote processor. The data buffer is not copied in order to construct the outgoing message; therefore the application should be sure to provide the optional status parameter. Only when testing the status object returns *MOL_STATUS_COMPLETE* can the application know that it is safe to modify the data buffer. In addition, the user-supplied data buffer must have an empty region at the beginning of size *MOL_HANDLER_DATA_SIZE* bytes, which the MOL uses to store data that must be sent with the request. Any data occupying this region will be overwritten before the request is sent.

mol_message()

Parameters:

1. **mol_mobile_ptr_t mptr**: The application-defined mobile object to which this data message should be sent
2. **mol_msg_handler_t handler**: The user handler to invoke on the target processor
3. **void* data**: Parameter data buffer; this buffer will be provided to the user handler
4. **int size**: The size of the parameter data buffer, in bytes
5. **void* arg**: This is a single machine-word sized argument, and will be passed to the user handler
6. **mol_status_t* status**: *OPTIONAL* This parameter may be left out of the *mol_message()* call.

Returns: None

Description:

Sends a message to a user-defined mobile object which may be anywhere in the parallel system, and which may be in transit as the message is being delivered. The data buffer is copied in order to construct an outgoing message and is therefore available for modification as soon as this operation returns. However, for large data sizes, this copy can have an adverse impact on latency and overall performance.

mol_message_no_copy()

Parameters:

1. **mol_mobile_ptr_t mptr**: The application-defined mobile object to which this data message should be sent
2. **mol_msg_handler_t handler**: The user handler to invoke on the target processor
3. **void* data**: Parameter data buffer; this buffer will be provided to the user handler
4. **int size**: The size of the parameter data buffer, in bytes
5. **void* arg**: This is a single machine-word sized argument, and will be passed to the user handler
6. **mol_status_t* status**: *OPTIONAL* This parameter may be left out of the *mol_message()* call.

Returns: None

Description:

Sends a message to a user-defined mobile object which may be anywhere in the parallel syste, and which may be in transit as the message is being delivered. The data buffer is not copied in constructing the outgoing message; therefore the application should be sure to provide the optional status parameter. Only when testing the status object returns *MOL_STATUS_COMPLETE* can the application know that it is safe to modify the data buffer. In addition, the user-supplied data buffer must contain an empty region at its beginning of size *MOL_HANDLER_DATA_SIZE* bytes, which the MOL uses to store data that must be sent along with the message. Any data occupying this region will be

overwritten before the message is sent.

mol_status_test()

Parameters:

1. **mol_status_t* status**: Pointer to a status object

Returns: Integer; one of three possible values

1. MOL_STATUS_UNINITIALIZED: The status object has not yet been associated with a communication operation
2. MOL_STATUS_IN_PROGRESS: The communication operation associated with the status object has begun, but has not yet completed
3. MOL_STATUS_COMPLETE: The communication operation assocated with the status object has completed, and any data buffers are now safe to access or modify

Description:

This routine is used to test the status of a communication operation. The MOL makes use of asynchronous and split-phase communication operations which return to the user before the operation has completed. This lowers the latency associated with communication visible to the application. However, it is not safe to modify data buffers used in communication until the operation has completed. The status variable is used to signal when operations have finshed. This routine returns immediately, allowing applications to proceed with other work.

Note:

In order for a processor to be notified of an operation's completion, that processor

must be issueing either DMCS or MOL polling operations.

mol_status_wait()

Parameters:

1. **mol_status_t* status**: Pointer to a status object

Returns: None

Description:

This routine does not return until the communication operation associated with the status object has completed. Once this operation returns, the application may safely modify parameter data buffers.

Note:

This operation executes DMCS polling operations, and therefore any application DMCS messages may be serviced during the *mol_status_wait()* operation.

mol_status_reset()

Parameters:

1. **mol_status_t* status**: Pointer to a status object

Returns: None

Description:

This routine is used to prepare a status object that has already been used in a communication operation for subsequent reuse. The result is that the state of the status object is returned to *MOL_STATUS_UNINITIALIZED.*

Table C.5: MOL OBJECT MIGRATION OPERATIONS

Installing and Uninstalling Operations	
void **mol_install_obj***(mol_mobile_ptr_t mp, void* data, mol_move_info_t* move_info, int src)*	Installs a mobile object after migration
*mol_move_info_t** **mol_uninstall_obj***(mol_mobile_ptr_t mp, int tgt)*	Uninstalls a mobile object prior to migration
void **mol_uninstall_obj***(mol_mobile_ptr_t mp, int tgt, void* buffer)*	Uninstalls a mobile object prior to migration and uses *buffer* to store *move info* information
Convenience Operations (DEPRICATED!!!)	
void **mol_move_obj***(mol_mobile_ptr_t mp, int size, int tgt)*	Sends an object to a remote processor and installs it

C.2.5 Object Migration Operations

The MOL provides the mechanism, but not the policy, for object migration. That means that applications or higher level libraries may migrate objects explicitly, but the MOL itself is not going to initiate any object migrations. It is up to the application or higher level library to determine which mobile objects are candidates for migration and to where those objects should be moved.

Table C.5 contains the operations used to migrate mobile objects.

mol_install_obj()

Parameters:

1. **mol_mobile_ptr_t mp**: The mobile pointer which refers to the object which is being installed
2. **void* data**: A pointer to the local data which makes up the mobile object
3. **mol_move_info_t* move_info**: An opaque object which contains information used to update the local directory structure
4. **int src**: The processor from which the mobile object is migrating

Returns: None

Description:

This operation is used to install a mobile object once that object is moved to a new processor. This is not necessary when the object is first created. This routine is passed an pointer of type *mol_move_info_t**, which is obtained when the object is uninstalled from the original processor. This buffer contains information used to update the MOL's internal tables, guaranteeing that messages will find their intended targets and messages sent from a particular processor will arrive in order. (The MOL guarantees message ordering for messages sharing a common source, but makes no guarantees about the ordering of messages sent from different sources.) Once this function is called, subsequent messages intended for this data object will be routed correctly.

mol_uninstall_obj()

Parameters:

1. **mol_mobile_ptr_t mp**: The mobile pointer which refers to the object which is being uninstalled; note that the mobile object must be local to the calling processor
2. **int tgt**: The processor to which the mobile object is being transferred

Returns: mol_move_info_t*; needed to update the directory on the remote processor. This is a pointer to a buffer of size *MOL_MOVE_INFO_SIZE.*

Description:

This operation is used to uninstall the mobile object before migrating it to a

new processor. This routine returns a pointer of type *mol_move_info_t**, which points to a buffer that must be passed, along with the object data, to the target processor for reinstallation. This buffer is of size *MOL_MOVE_INFO_SIZE*. It should also be noted that this routine does not deallocate the memory occupied by the mobile object; that is the reponsibility of the user or higher-level library.

mol_uninstall_obj()

Parameters:

1. **mol_mobile_ptr_t mp**: The mobile pointer which refers to the object which is being uninstalled; note that the mobile object must be local to the calling processor
2. **int tgt**: The processor to which the mobile object is being transferred
3. **void* buffer**: A user-defined buffer used to contain the *move info* buffer. This buffer must be at least of size *MOL_MOVE_INFO_SIZE* bytes.

Returns: None

Description:

This operation is used to uninstall the mobile object before migrating it to a new processor. This routine fills in a buffer with information used to update the receiving processor's internal tables used to enforce message ordering and provide correct message routing. This routine is provided in order to avoid the dynamic memory allocation that would otherwise be necessary during object migration. The buffer passed to this routine must be sent, along with the mobile object data, to the new processor during object migration.

mol_move_obj()

NOTE: This routine has been depricated and should no longer be used; it will cause a memory leak (or worse). The better way to migrate mobile objects is to provide routines which are responsible for allocating and deallocating the memory occupied by the objects, calling the *mol_uninstall_obj()* and *mol_install_obj()* routines, and transferring the data. DMCS or another low-layer communication substrate may be used for the transferral.

Parameters:

1. **mol_mobile_ptr_t mp**: The mobile pointer which refers to the object which is being moved
2. **int size**: The size of the mobile object in bytes
3. **int tgt**: The processor to which the mobile object is being transferred

Returns: None

Description:

This is a convenience operation, and only works in a limited set of cases. The mobile object being transferred must be a single, contiguous chunk of memory. In addition, the target processor must be polling in order to receive the incoming data. This operation is simply a wrapper around the *mol_uninstall_obj()* and the *mol_install_obj()* routines, and frees the user from having to deal with *mol_move_info_t* objects.

Table C.6: MOL POLLING AND SYNCHRONIZATION OPERATIONS

MOL Polling Operation	
void **mol_poll()**	Polls the network for pending messages
MOL Synchronization Operation	
void **mol_barrier()**	Blocks until all nodes enter the barrier

C.2.6 Polling Operations

There are two methods that runtime systems use to notify applications of the arrivals of messages from the network: interrupts and explicit application polling. In the first method, arriving messages asynchronously signal the application, at which time interrupt handlers are executed in order to deal with the new arrival. The second method has applications post explicit polling operations to retrieve any messages that may have arrived since the last poll.

While interrupts ensure the timely delivery of messages, they can lead to poor application performance. For instance, many interrupts require a user-kernel level boundary crossing. In addition, interrupts can lead to poor cache utilization due to the fact that application code, which may fill the cache with needed data, may be interrupted, potentially flushing the cache.

Polling avoids these pitfalls. However, in order to ensure timely delivery of messages, polling operations must be placed at the appropriate intervals. Polling infrequently may lead to network congestion with some messages being dropped. Polling too frequently may cause polling overheads to build up and harm overall application performance. Applications must therefore be careful concerning where and how often *mol_poll()* is called. Turning on the MOL's profiling capability may provide information which will assist application developers

in making this decision.

Table C.6 contains the polling operations provided by the MOL.

mol_poll()

Parameters: None

Returns: None

Description:

The polling operation polls the network for any incoming communications, and executes any user handlers. The MOL's polling operation contains a DMCS polling operation, so any arriving DMCS messages will also have their associated user handlers executed. Therefore, applications do not need to post both *mol_poll()* and *dmcs_poll()* calls. Because the MOL is single-threaded, the *mol_poll()* call should be made only from within the main application thread. This means that all user handlers will execute within a single thread.

C.2.7 Synchronization Operations

The MOL provides only a simple barrier operation to synchronize processes. The algorithm is a simple fan-in, fan-out algorithm which ensures that all processes enter the barrier before any are allowed to leave. As with DMCS, there are some words of warning for applications using the MOL barrier.

First is that the MOL barrier operation synchronizes *all* processes. If applications wish to synchronize only a subset of the available processes, then separate synchronization operations will need to be constructed. Second is that the MOL is unable to distinguish

between different barrier calls in the application code. Therefore, it is not necessary that all processes enter the same barrier, only that all processes enter *some* barrier. A result of this is that barriers placed close together in the application code will often lead to unpredictable behavior.

mol_barrier()

Parameters: None

Returns: None

Description:

Blocks until all processes have entered the barrier.

C.3 Example Code

The code snippet in Figure C.3 shows a simple singly-linked queue node data structure, with a single integer value field. In addition, the node data structure has a single method defined which takes as a parameter the partial sum of the values in the queue nodes, adds the local value to the partial sum, and calls the method on the next node, passing the new partial sum. The last node in the queue prints out the total result.

This program begins by creating a ten element queue, where each node contains a random integer value. The *create_queue()* function returns a pointer to the first node in the queue. The program then calculates the sum of all of the values contained in the queue, by calling the *calculate_sum()* method on the first node in the queue, and seeding the partial sum value with 0.

Figure C.3: SEQUENTIAL CODE FOR A SINGLY-LINKED QUEUE

```
1  struct queue_node_t {
2    int value;
3    queue_node_t* next;
4
5    queue_node_t(int v) : value(v) { }
6
7    void calculate_sum(int partial_sum) {
8      int new_sum = partial_sum + value;
9      if (next == NULL) {
10       cout << "Total sum is " << new_sum << endl;
11     } else {
12       next->calculate_sum(new_sum);
13     }
14   }
15 };
16
17 queue_node_t* create_queue(int num_elems) {
18   queue_node_t* node = new queue_node_t(rand());
19   queue_node_t* prev_node = node;
20   for (int i = 1; i < num_elems; ++i) {
21     queue_node_t* new_node = new queue_node_t(rand());
22     prev_node->next = new_node;
23     prev_node = new_node;
24   }
25   return node;
26 }
27
28 int main(int argc, char* argv[]) {
29   quee_node_t* queue_head;
30
31   queue_node_t* queue_head = create_queue(10);
32   queue_head->calculate_sum(0);
33
34   return 0;
35 }
```

Each node in the queue will add its local value to the partial sum and pass that value along until the last node, which will print out the total sum of the nodes in the queue. The next step is to examine how we would implement this simple program using the MOL. The code to do this is given in Figure C.4.

The code changes necessary begin with the *queue_node_t* structure itself. In the sequential example, line 3 indicates that the nodes in the queue are linked together with local

Figure C.4: PARALLEL CODE FOR A SINGLY-LINKED QUEUE

```
1  bool glob_finished = false;
2
3  struct queue_node_t {
4    int value;  mol_mobile_ptr_t next;
5    queue_node(int v) : value(v) { }
6    void calculate_sum(int partial_sum) {
7     int new_sum = partial_sum + value;
8     if (next == NULL) {
9       cout << "Total sum is " << new_sum << endl;
10      for (int i = 0; i < mol_num_procs(), ++i) { mol_request(i, finish_handler, NULL, 0, NULL); }
11    } else {
12      mol_message(next, calculate_sum_handler, &new_sum, sizeof(int), NULL);
13    }
14   }
15 };
16
17 void calculate_sum_handler(int src, mol_mobile_ptr_t mptr, void* obj_data,
18                             void* user_data, int size, void* arg) {
19   queue_node_t* q_node = (queue_node_t*)obj_data;
20   q_node->calculate_sum(*(int*)user_data);
21 }
22
23 void finish_handler(int src, void* data, int size, void* arg) { glob_finished = true; }
24
25 mol_mobile_ptr_t create_queue(int num_elems) {
26   queue_node_t* node = new queue_node_t(rand());
27   mol_mobile_ptr_t queue_head = mol_create_mob_ptr(node);
28   queue_node_t*  prev_node = node;
29   for (int i = 1; i < num_elems; ++i) {
30     queue_node_t* new_node = new queue_node_t(rand());
31     mol_mobile_ptr_t new_mob_ptr = mol_create_mob_ptr(new_node);
32     prev_node->next = new_mob_ptr;
33     prev_node = new_node;
34   }
35   return queue_head;
36 }
37
38 int main(int argc, char* argv[]) {
39   mol_mobile_ptr_t queue_head; int zero = 0;
40   mol_msg_handler_t msg_handlers[] = { calculate_sum_handler };
41   mol_init(argc, argv); mol_register_msg_handlers(msg_handlers, 1);
42   if (mol_my_proc() == 0) {
43     mol_mobile_ptr_t queue_head = create_queue(10);
44     mol_message(queue_head, calculate_sum_handler, &zero, sizeof(int), NULL);
45   }
46   while (!glob_finished) { mol_poll(); }
47   mol_shutdown();
48   return 0;
49 }
```

pointers. In a distributed data structure, this method is no longer valid. Instead, the MOL provides mobile pointers to link queue nodes to one another. Line 4 of Figure C.4 demonstrates this; a mobile pointer replaces the local pointer to point to the next element in the queue. This alteration has ramifications on the *calculate_sum()* method. In line 12 of the sequential example, we see that, because all nodes of the queue are in same memory space, a simple method invocation is all that is necessary to invoke the *calculate_sum()* method on the next queue node. When using the MOL, this is no longer the case, and the method invocation is replaced in line 14 with a *mol_message()* call. Also, because a single function

Figure C.5: OPTIMIZING COMMUNICATION USING OBJECT LOCALITY

```
1  queue_node_t* node = (queue_node_t*)mol_mobile_ptr_deref(next);
2  if (node == NULL) {
3    mol_message(next, calculate_sum_handler, &new_sum, sizeof(int), NULL);
4  } else {
5    node->calculate_sum(new_sum);
6  }
```

call stack does not exist in the second case, we must devise a method to signal all processors that the end of the queue has been reached and the program is finished. Line 10 loops over all processors in the parallel system and sends a request signalling the end of the program. The request handler simply sets a global flag to *true*. On line 46 of the program, the while-loop will wait for this condition while polling the network for any incoming messages or requests.

The function used to construct the queue data structure (beginning on line 17 of the sequential example and line 25 of the parallel example) also needs to change. The *mol_create_mob_ptr()* routine is used to make mobile objects of each queue node. This routine will return a mobile pointer to the newly created mobile object. This mobile pointer is then stored in the queue node structure.

As a brief aside, there is a simple optimization which may prove beneficial in a variety of instances, and that is to check to see if the mobile object which is the target of an *mol_message()* operation is actually not local. This applies specifically to line 17 in Figure C.4. We can replace this line of code with the code given in Figure C.5.

In this code, we used the *mol_mobile_ptr_deref()* function to determine whether or not a mobile object is local or remote; a non-NULL pointer will be returned only if the object is local. We can then use this pointer to call the desired class method directly. Only when

Figure C.6: MIGRATING MOBILE OBJECTS USING THE MOL

```
1  struct handler_struct_t {
2    mol_mobile_ptr_t mp;
3    mol_move_info_t move_info;
4  };
5
6  void receive_obj_handler(int src, dmcs_pointer_t data, dmcs_arg_t size) {
7    header_struct_t* header = (header_struct_t*)data;
8    int obj_size = size - sizeof(header_struct_t);
9    void* obj_data = malloc(obj_size);
10   memcpy(obj_data, (char*)data + sizeof(header_struct_t), obj_size);
11   mol_install_obj(header->mp, obj_data, header->move_info);
12 }
13
14 void move_obj(mol_mobile_ptr_t mp, int size, int tgt) {
15   void* obj_data = mol_mobile_ptr_deref(mp);
16   mol_move_info_t move_info = mol_uninstall_obj(mp, tgt);
17   void* buffer = malloc(size + sizeof(header_struct_t);
18   header_struct_t* header = (header_struct_t*)buffer;
19   header->mp = mp;
20   header->move_info = move_info;
21   memcpy((char*)buffer + sizeof(handler_struct_t), obj_data, size);
22   dmcs_block_rsrN(tgt, receive_obj_handler, buffer,
23                   size + sizeof(handler_struct_t));
24   free(buffer);
25 }
```

the object is actually on a remote processor do we need to go through the *mol_message()* mechanism.

From lines 53-54 in Figure C.4, we can see that only processor 0 is going to be responsible for creating the parallel queue. Therefore all queue nodes are going to exist only on that processor. To be truely distributed, there must be some way to move nodes from one processor to another. The code snippet given in Figure C.6 shows how to do this. First, we declare a simple data structure that will contain the information necessary during object migration. This consists of the mobile pointer for that object, as well as a field of type *mol_move_info_t*, which is an opaque data object used to transport information necessary for updating the distributed directory structure.

The processor on which the queue node is originally located calls the *move_obj()* routine, which begins on line 14. This routine makes use of the *mol_uninstall_obj()* function (line 16) to uninstall the mobile object and obtain a *mol_move_info_t* object. A single buffer is then constructed which contains the header information and the user-defined mobile object. Note that if the mobile object does not exist in contiguous memory, then there must be some method to pack it prior to transport. This packed buffer is then sent to the target processor using a DMCS communication operation, however, any communication operation from DMCS, the MOL, or the lower-level communication transport may be used.

Once the data arrives at the target, the *receive_obj_handler()* routine is executed (line 6). This handler will obtain the mobile pointer and the move info structure from the header at the beginning of the buffer. Note that the mobile object data must be copied into a safe permanent location; the data buffer supplied to the handler is in system memory and will not be valid once that handler returns. Finally, in line 11, the *mol_install_obj()* routine is used to install the mobile object on the new processor. Now, messages will be routed correctly to the mobile object.

Bibliography

[1] I. Ahmad, A. Ghafoor, K. Mehrotra, and C. Mohan. Performance modeling of load balancing algorithms using neural networks. *Concurrency; Practice and Experience*, 6(5):393–409, 1994.

[2] P. Amaral, C. Jacquemot, P. Jensen, R. Lea, and A. Mirowski. Transparent object migration in COOL2. In *Position Papers of the ECOOP '92 Workshop W2*, Y. Berbers and P. Dickman, editors, pages 72–77, 1992.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] C. Anderson. An implementation of the fast multipole method without multipoles. *SIAM Jour. Scientific Computing*, 13(4):923–947, July 1992.

[5] A. Appel. An efficient program for many-body simulation. *SIAM Jour. Scientific Stat. Computing*, 6, 1985.

[6] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Eigler, and G. Gao. ABC++: Concurrency by inheritance in C++. *IBM Systems Journal*, 34(1):120–137, 1995.

[7] Infiniband Trade Association. Infiniband home page. http://www.infinibandta.org. Accessed June 30, 2000.

[8] A. Aumage, L. Bougé, A. Denis, J. Méhaut, G. Mercier, R. Namyst, and L. Prylli. Madeleine ii: A portable and efficient communication library for high-performance cluster computing. Technical Report Research Report No. 2000-26, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 2000.

[9] T. Baker. Automatic mesh generation for complex three-dimensional regions using a constrained delaunay triangulation. *Engineering with Computers*, 5:161–175, 1989.

[10] H. Bal, M. Frans Kaashoek, and A. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[11] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C. Ho, S. Kipnis, and M. Snir. CCL: A portable and tunable collective communication library for

scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, 1995.

[12] I. BANICESCU. *Load Balancing and Data Locality in the Parallelizatoin of the Fast Multipole Algorithm.* PhD thesis, Polytechnic University, Brooklyn, New York, January 1996.

[13] A. BARAK AND A. SHILOH. A distributed load-balancing policy for a multicomputer. *Software Practice and Experience*, 15(9):901–913, September 1985.

[14] K. BARKER. Personal conversation with Jeffrey M. Squyres, 2004. Jeff Squyres' clarification of the cross-over point between LAM's immediate eager and rendevous protocols, and how each protocol impacts the latency observable to the application.

[15] K. BARKER AND N. CHRISOCHOIDES. An evaluation of a framework for the dynamic load balancing of highly adaptive and irregular applications. In *Proceedings of the IEEE/ACM SC'03*, 2003.

[16] 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.

[17] K. BARKER, N. CHRISOCHOIDES, AND K. PINGALI. A load balancing framework for adaptive and asynchronous applications. *IEEE Transactions on Parallel and Distributed Computing*, 15(2):77–101, February 2004.

[18] J. BARNES AND P. HUT. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324:446–449, 1986.

[19] A. BASERMANN, J. CLINCKEMAILLIE, T. COUPEZ, J. FINGBERG, H. DIGONNET, R. DUCLOUX, J.-M. 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 Modelling*, 25:83–98, 2000.

[20] P. BECKMAN AND D. GANNON. Tulip: Parallel runtime support system for pC++. http://www.extreme.indiana.edu.

[21] A. BELGUELIN, J. DONGARRA, A. GEIST, R. MANCHEK, S. OTTO, AND J. WALPORE. PVM: Experiences, current status, and future direction. In *Supercomputing '93 Proceedings*, pages 765–766, 1993.

[22] M. BERGER AND S. BOKHARI. A partitioning strategy for non-uniform problems on multicomputers. *IEEE Transactions on Computers*, 36:570–580, 1987.

[23] 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.

[24] 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.

[25] R. BHOEDJANG, T. RUHL, R. HOFMAN, K. LANGENDOEN, H. BAL, AND F. KAASHOEK. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, September 1993.

[26] K. BIRMAN, R. COOPER, T. A. JOSEPH, K. MARZULLO, M. MAKPANGOU, K. KANE, F. SCHMUCK, AND M. WOOD. The ISIS system manual, September 1990. Dept. of Computer Science, Cornell University.

[27] R. BISWAS, S. DAS, D. HARVEY, AND L. OLIKER. Parallel dynamic load balancing strategies for adaptive irregular applications. *Applied Mathematical Modelling*, 25:109–122, 2000.

[28] R. BLUMOFE, C. JOERG, B. KUSZMAUL, C. LEISERSON, K. RANDALL, AND Y. ZHOU. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 55–69, 1995.

[29] N. BODEN, D. COHEN, R. FELDERMAN, A. KULAWIK, C. SEITZ, J. SEIZOVIC, AND W. SU. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[30] S. BOKHARI AND D. MAVRIPLIS. The tera multithreaded architecture and unstructured meshes, 1998. ICASE, NASA Langley Research Center, NASA/CR-1998-208953.

[31] J. BOOTH. Balancing priorities and load for state space search on large parallel machines. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[32] L. BOUGE, J. MEHAUT, AND R. NAMYST. Madeleine: An efficient and portable communication interface for RPC-based multithreaded environments. In *Proceedings of the 1998 Conference on Parallel Architectures and Compilation Techniques, PACT '98*, pages 240–247, 1998.

[33] J. BOURGEOIS AND F. SPIES. Performance prediction of distributed applications running on network of workstations. In *Proc. of PDPTA'99*, volume 2, pages 672–678, June 1999.

[34] A. BOWYER. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[35] M. BOZYIGIT. History-driven dynamic load balancing for recurring applications on networks of workstations. *The Journal of Systems and Software*, 51:61–72, 2000.

[36] A. BRUNSTROM AND R. SIMHA. Dynamic versus static load balancing in a pipeline computation. *International Journal of Modeling and Simulation*, 17(4):317–327, 1997.

[37] R. BUTLER AND E. LUSK. User's guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992.

[38] R. BUTLER AND E. LUSK. The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1995.

[39] X. C. CAI AND Y. SAAD. Overlapping domain decomposition algorithms for general sparse matrices. Technical Report umsi-93-027, Army High Performance Computing Research Center, 1993.

[40] R. CALKIN, R. HEMPEL, H.-C. HOPPE, AND P. WYPIOR. Portabile programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, 1994.

[41] L. CAMPOS AND I. SCHERSON. Rate of change load balancing in distributed and parallel systems. *Parallel Computing*, 26:1213–1230, 2000.

[42] B. CARTER, C. CHEN, L. CHEW, N. CHRISOCHOIDES, G. GAO, G. HERBER, A. INGRAFFEA, R. KRAUSE, C. MYERS, D. NAVE, K. PINGALI, P. STODGHILL, S. VAVASIS, AND P. WAWRZYNEK. *Lecture Notes in Computer Science 1800*, chapter Parallel FEM Simulation of Crack Propogation – Challenges, Status. Springer-Verlag, 2000.

[43] NATIONAL ENERGY RESEARCH SCIENTIFIC COMPUTING CENTER. M-VIA: High performance modular via for linux. http://www.nersc.gov/research/FTG/via.

[44] K. CHANDY AND C. KESSELMAN. *Research Directions in Concurrent Object-Oriented Programming*, chapter CC++: A Declarative Concurrent Object-Oriented Programming Notation. MIT Press, 1993.

[45] C. CHANG, A. SUSSMAN, AND J. SALTZ. *Parallel Programming Using C++*, chapter CHAOS++. MIT Press, 1996.

[46] J. CHASE, F. AMADOR, E. LAZOWSKA, H. LEVY, AND R. LITTLEFIELD. The amber system: Parallel programming on a network of multiprocessors. In *The 12th Annual ACM Symposium on Operating System Principles (SOSP12)*, pages 147–158, December 1989.

[47] T. CHEATHAM, A. FAHMY, D. STEFANESCU, AND L. VALIANT. Bulk synchronous parallel computing – a paradigm for transportable software. In *Proc. of the 28th Annual Hawaii Conference on System Sciences*, volume II. IEEE Computer Society Press, January 1995.

[48] A. CHERNIKOV, N. CHRISOCHOIDES, AND K. BARKER. Parallel programming environment for mesh generation. Submitted to 8th International Conference on Numerical Grig Generation in Computational Field Simulations; Waikiki Beach, Honolulu, Hawaii, USA, June 2-6 2002.

[49] L. CHEW. Constrained delaunay triangulations. *Algorihmica*, 4:97–108, 1989.

[50] 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.

[51] N. CHRISOCHOIDES, P. CHEW, AND F. SUKUP. Parallel constrained delaunay meshing. In *1997 ASME/ASCE/SES Special Symposium on Trends in Unstructured Mesh Generation*, pages 89–96, June 1997.

[52] N. CHRISOCHOIDES, E. HOUSTIS, AND J. RICE. Mapping algorithms and software environment for data parallel iterative PDE solvers. *Special Issue of the Journal of Parallel and Distributed Computing on Data-Parallel Algorithms and Programming*, 21(1):75–95, April 1994.

[53] N. CHRISOCHOIDES, N. MANSOUR, AND G. FOX. Performance evaluation of data mapping algorithms for parallel single-phase iterative PDE solvers. In *IEEE Proceedings of the Scalable High Performance Computing Conference*, pages 764–772, Knoxville, Tennessee, May 23-25 1994.

[54] PORTABLE RUNTIME SYSTEMS (PORTS) CONSORTIUM. http://www.cs.uoregon.edu/research/paracomp/ports/. Accessed June 30, 2000.

[55] EMULEX CORPORATION. Emulex homepage. http://www.emulex.com. Accessed June 30, 2000.

[56] A. CORRADI, L. LEONARDI, AND F. ZAMBONELLI. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1):22–31, January-March 1999.

[57] D. CULLER, A. DUSSEAU, S. GOLDSTEIN, A. KRISHNAMURTHY, S. LUMETA, T. VON EICKEN, AND K. YELICK. Parallel programming in split-c. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[58] G. CYBENKO. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.

[59] INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE EN AUTOMATIQUE. Site web de l'inria. http://www.inria.fr/. Accessed March 29, 2004.

[60] T. DECKER. Virtual data space - load balancing for irregular applications. *Parallel Computing*, 26:1825–1860, 2000.

[61] E. DEELMAN, A. DUBE, A. HOISIE, Y. LUO, R. OLIVER, D. SUNDARAM-STUKEL, AND H. WASSERMAN. Poems: End-to-end performance design of large parallel adaptive compuational systems. In *Proc. of the First International Workshop on Software Performance*, pages 18–30, October 1998.

[62] B. DELAUNAY. Sur la sphere vide. *Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7:793–800, 1934.

[63] A. DESHPANDE AND M. SCHULTZ. Efficient parallel programming with linda. In *Supercomputing '92*, pages 238–244, 1992.

[64] K. DEVINE, B. HENDRICKSON, E. BOMAN, M. ST. JOHN, AND C. VAUGHAN. Zoltan: A dynamic load-balancing library for parallel applications: Developer's guide. Technical Report SAND99-1376, Sandia National Laboratories, Albuquerque, NM, 1999.

[65] K. DEVINE, B. HENDRICKSON, E. BOMAN, M. ST. JOHN, AND C. VAUGHAN. Zoltan: A dynamic load-balancing library for parallel applications: User's guide. Technical Report SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 1999.

[66] K. DEVINE, B. HENDRICKSON, E. BOMAN, M. ST. JOHN, AND C. VAUGHAN. Design of dynamic load-balancing tools for parallel applications. In *Proceedings of the International Conference on Supercomputing*, Santa Fe, May 2000.

[67] E. DIJKSTRA, W. SEIJEN, AND A. VAN GASTEREN. Derivation of a termination detection algorithm for a distributed computation. *Information Processing Letters*, 15(5):217–219, 1983.

[68] D. DINUCCI. Cooperative data sharing: An architecutre-independent interface for implementing parallel cfd applications. Technical Report M/S T27A-2, NASA Ames Research Center.

[69] D. DINUCCI. Cooperative data sharing: A layered approach to an architecture-independent message-passing interface. In *Proceedings of the Second MPI Developers Conference, Notre Dame*, pages 58–65, July 1996.

[70] R. ESSER, J. JANNECK, AND M. NAEDELE. Applying an object-oriented petri net language to heterogeneous systems design. In *Proc. of the Workshop on Petri Nets in Systems Engineering*, September 1997.

[71] C. FARHAT. On the mapping of massively parallel processors onto finite element graphs. *Computers and Structures*, 32(2):347 – 353, 1989.

[72] W. FERNG, K. WU, S. PETITON, AND Y. SAAD. Sparse matrix computation on massively parallel computers. Technical Report umsi-92-084, Army High Performance Computing Research Center, 1992.

[73] J. FLOWER AND A. KOLAWA. Express is not just a message passing system. *Parallel Computing*, 20(4):597–614, 1994.

[74] L. FLYNN AND S. HUMMEL. The mathematical foundations of the factoring scheduling method. Technical Report RC18462, IBM Research Report, Oct. 1992.

[75] MPI FORUM. Message passing interface standard 1.0 and 2.0. http://www.mpi-forum.org, 1997. Accessed Nov 11, 2003.

[76] I. FOSTER, C. KESSELMAN, AND S. TUECKE. Nexus: Runtime support for task parallel programming languages. Technical Report FKT94, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne Il., 1994.

[77] I. FOSTER, C. KESSLEMAN, AND S. TUECKE. The nexus approach to integrating multithreading and communication. Technical Report MCS-P494-0195, Argonne National Laboratory, 1994.

[78] G. Fox, R. Williams, and P. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc. San Francisco, CA., 1994.

[79] W. Gautschi. *Numerical Analysis: An Introduction.* Birkhäuser, Boston, 1997.

[80] B. Ghosh, F. Leighton, B. Maggs, S. Muthukrishnan, C. Plaxton, R. Rajaraman, A. Richa, R. Tarjan, and D. Zuckerman. Tight analyses of two local load balancing algorithms. In *Proc. of the 27th Annual ACM Symposium on Theory of Computing*, pages 548–558, May 1995.

[81] T. Goehring and Y. Saad. Heuristic algorithms for automatic graph partitioning. Technical Report umsi-94-29, University of Minnesota Supercomputer Institute, 1994.

[82] L. Greengard. The rapid evaluation of potential fields in particle systems. Technical Report YALEU/DCS/RR-533, ACM Distinguished Dissertation Series, Vol. 14, Cambridge, MA, April 1997.

[83] L. Greengard and W. Gropp. A parallel version of the fast multipole method. *Computers Mathematical Applications*, 20(7):63–71, 1990.

[84] IBM RS6000 Group. *IBM Parallel Systems Support Programs for AIX: Administration Guide.* IBM RS6000 Group.

[85] M. Haines, D. Cronk, and P. Mehrotra. On the design of chant: A talking threads package. In *Proceedings of Supercomputing '94*, pages 350–359, Washington, D.C., November 1994. Also appears as ICASE Technical Report 94-25.

[86] M. Hamdi and C. Lee. Dynamic load-balancing of image processing applications on clusters of workstations. *Parallel Computing*, 22:1477–1492, 1997.

[87] H. Heiss and M. Schmitz. Decentralized dynamic load balancing: The particles approach. *Information Sciences*, 84:115–128, 1995.

[88] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184:485–500, 2000.

[89] M. Hill, J. Larus, S. Reinhardt, and D. Wood. Cooperative shared memory: Software and hardware support for scalable multiprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, New York, NY, 1992. ACM Press.

[90] Y. Hu and R. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25:417–444, 1999.

[91] C. Hui and S. Chanson. Flexible and extensible load balancing. *Software-Practice and Experience*, 27(11):1283–1306, November 1997.

[92] MPI Software Technology Inc. Clustercontroller scheduling software. http://www.mpi-softtech.com/products/cluster_controller/default.asp.

[93] MYRICOM INC. Myricom homepage. http://www.myri.com. Accessed June 30, 2000.

[94] D. JOHNSON AND W. ZWAENEPOEL. The peregrine high-performance RPC system. *Software – Practice and Experience*, 23(2):201–221, 1993.

[95] K. JOHNSON, M. KAASHOEK, AND D. WALLACH. CRL: High-performance all-software distributed shared memory. In *15th Annual Symposium on Operating Systems Principles (COSP15)*, pages 213–228, December 1995.

[96] M. JONES AND P. PLASSMANN. Computational results for parallel unstructured mesh computations. *Computing Systems in Engineering*, 5:297–309, 1994.

[97] E. JUL, H. LEVY, N. HUTCHISON, AND A. BLACK. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[98] L. KALÉ. Parallel programming with charm: An overview, July 1993. Technical Report PPL-TR-93-8, University of Illinois, Urbana-Champaign, Department of Computer Science.

[99] L. KALÉ, M. BHANDARKAR, AND R. BRUNNER. Run-time support for adaptive load balancing. In *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun, Mexico*, J. Rolim, editor, volume 1800, pages 1152–1159, March 2000.

[100] L. KALÉ AND S. KRISHNAN. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 91–108, 1993.

[101] L. KALÉ, B. RAMKUMAR, A. SINHA, AND A. GURSOY. The charm parallel programming language and system: Part I – description of language features, 1995. Technical Report 95–2, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign.

[102] L. KALÉ, B. RAMKUMAR, A. SINHA, AND V. SALETORE. The CHARM parallel programming language and system: Part II – the runtime system. *IEEE Transactions on Parallel And Distributed Systems*, 1994.

[103] G. KARYPIS AND V. KUMAR. ParMETIS: Parallel graph paritioning and sparse matrix ordering library. Technical Report 97-060, Dept. of Computer Science, Univ. of Minnesota, http://www-users.cs.umn.edu/ karypis/metis/parmetis/main.shtml, 1997.

[104] D. KERBYSON, A. HOISIE, AND H. WASSERMAN. Modeling the performance of large-scale systems (keynote paper). In *UK Performance Engineering Workshop*, July 2003.

[105] D. KERBYSON, A. HOISIE, AND H. WASSERMAN. Use of predictive performance modeling during large-scale system installation. *Parallel Processing Letters*, 2003.

[106] D. KERBYSON, A. HOISIE, AND H. WASSERMAN. *Verifying Large-Scale System Performance During Installation Using Modeling.* Kulwer, September 2003.

[107] G. KOHRING. Dynamic load balancing for parallelized particle simulations on mimd computers. *Parallel Computing*, 21:683–693, 1995.

[108] V. KUMAR, A. GRAMA, AND N. VEMPATY. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Systems*, 22:60–79, 1994.

[109] J. KUSKIN, D. OFELT, M. HEINRICH, J. HEINLEIN, R. SIMONI, K. GHARACHORLOO, J. CHAPIN, D. NAKAHIRA, J. BAXTER, M. HOROWITZ, A. GUPTA, AND J. HENNESSY. The stanford flash multiprocessor. In *Proceeings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[110] ARGONNE NATIONAL LABORATORY. Mpich - a portable implementation of mpi. http://www-unix.mcs.anl.gov/mpi/mpich/.

[111] BLOOMINGTON LAM TEAM, INDIANA UNIVERSITY. Lam/mpi parallel computing. http://www.lam-mpi.org/. Accessed Nov. 11, 2003, Last Modified Sept. 23, 2003.

[112] J. LEATHRUM. *Parallelization of the Fast Multipole Algorithm: Algorithm and Architecture Design.* PhD thesis, Duke University, 1992.

[113] C. LIAO AND Y. CHUNG. Tree-based parallel load balancing methods for solution-adaptive finite element graphs on distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 10(4), April 1999.

[114] R. LÓHNER. Generation of three-dimensional unstructured grids by the advancing front method. In *26th AIAA Aerospace Sciences Meeting*, Reno, Nevada, 1988.

[115] R. LÓHNER, J. CAMBEROS, AND M. MARSHAL. Parallel unstructured grid generation. In *Unstructured Scientific Computation on Scalable Multiprocessors*, Piyush Mehrotra and Joel Saltz, editors, pages 31–64. MIT Press, 1990.

[116] R. LÓHNER, J. CAMBEROS, AND M. MARSHAL. *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 31–64. MIT Press, 1990.

[117] A. MAINWARING AND D. CULLER. Active Messages API and Communication Subsystem Organization. Technical report, University of California at Berkeley, 1999.

[118] D. MAVRIPLIS. An advancing front delaunay triangulation algorithm designed for robustness. Technical Report 92-49, ICASE, October 1992.

[119] O. MCBRYAN. An overview of message passing environments. *Parallel Computing*, 20(4):417–443, 1994.

[120] M. MITZENMACHER. On the analysis of randomized load balancing schemes. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 292–301, 1997.

[121] F. MUNIZ AND E. ZALUSKA. Parallel load balancing: An extension to the gradient model. *Parallel Computing*, 21:287–301, 1995.

[122] K. NAM, J. SEO, S. LEE, AND J. KIM. Syncrhonous load balancing in hypercube multicomputers with faulty nodes. *Journal of Parallel and Distributed Computing*, 58:26–43, 1999.

[123] J. NETO, P. WAWRZYNEK, M. CARVALHO, L. MARTHA, AND A. INGRAFFEA. An algorithm for three-dimensional mesh generation for arbitrary regions with cracks. *Engineering with Computers*, 17:75–91, 2001.

[124] H. NISHIKAWA AND P. STEENKISTE. Aroma: Language support for distributed objects. In *6th International Parallel Processing Symposium*, pages 686 – 690, March 1992.

[125] H. NISHIKAWA AND P. STEENKISTE. A general architecture for load balancing in a distributed-memory environment. In *International Conference on Distributed Computing Systems*, pages 47–54, 1993.

[126] L. NYLAND, J. PRINS, R. H. YUN, J. HERMANS, H. KUM, AND L. WANG. Modeling dynamic load balancing in molecular dynamics to achieve scalable parallel execution. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 356–365, 1998.

[127] L. OLIKER AND R. BISWAS. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distribued Computing*, 52(2):150–177, 1998.

[128] B. OVEREINDER, J. VESSEUR, F. VAN DER LINDEN, AND P. SLOOT. A communication kernel for parallel programming support on a massively parallel processor system. In *Proceedings of the Workshop on Parallel Programming and Computation (ZEUS '95) and the 4th Nordic Transputer Conference (NTUG '95)*, Peter Fritzon and Leif Finmo, editors, pages 259–266, Amsterdam, 1995. IOS Press.

[129] S. PAKIN, M. LAURIA, AND A. CHEN. High performance messaging on workstations: Illinois fast messages (FM) for myrinet. In *Supercomputing '95*, San Diego, CA, December 1995.

[130] UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN PARALLEL PROGRAMMING LABORATORY. Response to paper by nikos chrisochoides and kevin barker at sc'03. http://charm.cs.uiuc.edu/research/ldbal/nikos/nikos.html, 2003. Accessed March 19, 2004.

[131] I. PARSONS. Evaluation of distributed communication systems. In *CASCON '93, IBM Toronto*, pages 956–970 vol. 2, 1993.

[132] S. REINHARD, J. LARUS, AND D. WOOD. Tempest and typhoon: User-level shared memory. In *Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA '94)*, pages 325 – 337, 1994.

[133] R. VAN RENESSE, K. BIRMAN, R. COOPER, B. GLADE, AND P. STEPHENSON. The horus system, 1993. Reliable Distributed Computing with the ISIS toolkit, pages 133-147. IEEE Computer Society Press, Los Alamitos, CA.

[134] R. VAN RENESSE, K. BIRMAN, B. GLADE, K. GUO, M. HAYDEN, T. HICKEY, D. MALKI, A. VAYSBURD, AND W. VOGELS. Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76–83, April 1996.

[135] INTERNATIONAL BUSINESS MACHINES RS6000 GROUP. IBM parallel environment for AIX - MPI. http://qpsf.edu.au/software/ppe.html.

[136] Y. SAAD. Data structures and algorithms for domain decomposition and distributed sparse matrix computations. Technical Report umsi-95-014, Dept. of Computer Science, University of Minnesota, 1995.

[137] R. SAID, N. WEATHERILL, K. MORGAN, AND N. VERHOEVEN. Distributed parallel delaunay mesh generation. *Comp. Methods Appl. Mech. Engrg.*, 177:109–125, 1999.

[138] K. SCHLOEGEL, G. KARYPIS, AND V. KUMAR. Parallel multilevel diffusion schemes for repartitioning of adaptive meshes. Technical Report 97-014, University of Minnesota, 1997.

[139] K. SCHLOEGEL, G. KARYPIS, AND V. KUMAR. Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes. Technical Report 98-034, Department of Computer Science and Engineering, Unversity of Minnesota, http://www-users.cs.umn.edu/~karypis/publications/partitioning.html, 1998.

[140] K. SCHLOEGEL, G. KARYPIS, AND V. KUMAR. A unified algorithm for load-balancing adpative scientific simulations. Technical Report TR 00-033, Department of Computer Science and Engineering, University of Minnesota, http://www-users.cs.umn.edu/~karypis/publications/partitioning.html, 2000.

[141] K. SCHLOEGEL, G. KARYPIS, V. KUMAR, R. BISWAS, AND L. OLIKER. A performance study of diffusive vs. remapped load-balancing schemes. Technical Report 98-018, Department of Computer Science and Engineering, Unversity of Minnesota, http://www-users.cs.umn.edu/~karypis/publications/partitioning.html, 1998.

[142] K. SCHMIDT AND M. LEE. Implementing the fast multipole method in parallel. *Journal of Statistical Physics*, 63:1120, 1991.

[143] G. SHAH, J. NIEPLOCHA, J. MIRZA, C. KIM, R. HARRISON, R .K. GOVINDARAJU, K. GILDEA, P. DINICOLA, AND C. BENDER. Performance and experience with lapi: A new high-performance communication library for the IBM rs/6000 sp. In *Proceedings of the International Parallel Processing Symposium, IPPS '98*, pages 260 – 266, 1998.

[144] J. SHEWCHUK. *Delaunay Refinement Mesh Generation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213, May 1997.

[145] H. SIMON. Partitioning of unstructured problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*, 1991. Pergamon, New York.

[146] J. SINGH. *Parallel Hierarchical N-body Methods and their Implementations for Multiprocessors.* PhD thesis, Stanford University, 1993.

[147] 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. *Journal of Parallel and Distributed Computing*, 27:118–141, 1995.

[148] A. SKJELLUM, S. SMITH, N. DOSS, A. LEUNG, AND M. MORARI. The design and evolution of zipcode. *Parallel Computing*, 20(4):565–596, 1994.

[149] SUN MICROSYSTEMS SUN MPI GROUP. Sun hpc clustertools 3.1. http://www.sun.com/software/hpc/overview.html.

[150] V. SUNDERAM. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–340, 1990.

[151] VIA. The virtual interface architecture specification, version 1.0. http://www.viarch.org. Accessed December, 1997.

[152] T. VON EICKEN, D. CULLER, S. GOLDSTEIN, AND K. SCHAUSER. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.

[153] C. WALSHAW, M. CROSS, AND M. EVERETT. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47:102–108, 1997.

[154] D. WATSON. Computing the n-dimensional delaunay tessellation with applications to voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[155] J. WATTS AND S. TAYLOR. A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):235–267, March 1998.

[156] M. WILLEBEEK-LEMAIR AND A. REEVES. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[157] A. Wouk, editor. *New Computing Environments: Microcomputers in Large-Scale Scientific Computing*, chapter Using Supercomputers as Attached Procesors. SIAM, Philadelphia, 1987.

[158] I. WU. *Multilist Scheduling: A New Parallel Programming Model.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213, July 1993.

[159] M. WU AND W. SHU. Dde: A modified dimension exchange method for load balancing k-ary n-cubes. *Journal of Parallel and Distributed Computing*, 44:88–96, 1997.

[160] C. XU AND F. LAU. The generalized dimension exchange method for load balancing in k-ary n-cubes and variants. *Journal of Parallel and Distributed Computing*, 24:72–85, 1995.

[161] C. Xu, B. Monien, R. Lüling, and F. Lau. An analytical comparison of nearest neighbor algorithms for load balancing in parallel computers. In *Proc. of 9th International Parallel Processing Symposium*, pages 472–479, 1995.

[162] J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Parallel and Distributed Computing*, 18:1–13, 1993.

[163] F. Zambonelli. Exploiting biased load information in direct-neighbour load balancing policies. *Parallel Computing*, 25:745–766, 1999.

[164] T. Znati and R. Melhem. A uniform framework for dynamic load balancing strategies in distributed processing systems. *Journal of Parallel and Distributed Computing*, 23(2):246–255, 1994.

VITA

Kevin James Barker

Kevin Barker received his B.S. degree in Computer Science from North Carolina State University in May, 1997. He received his M.S. degree in Computer Science from the University of Notre Dame in January, 2001. His Master's thesis described the development of a prototype runtime system for the support of adaptive and asynchronous applications on non-traditional, multi-layer parallel architectures. His current research interests include runtime load balancing support and modeling techniques for adaptive and irregular parallel applications, and real-time application steering using embedded feedback.