Data Movement and Control Substrate for Parallel Adaptive Applications*

Jeffrey Dobbelaere, Kevin Barker, Nikos Chrisochoides, and Démian Nave
Computer Science Department
College of William & Mary
Williamsburg, VA 23185

Keshav Pingali
Computer Science Department
Cornell University
Ithaca, NY 14853-3801

Abstract

In this paper, we present the *Data Movement and Control Substrate* (DMCS), that implements one-sided low-latency communication primitives for parallel adaptive and irregular computations. DMCS is built on top of low-level, vendor-specific communication subsystems such as LAPI for IBM SP machines and widely available libraries like MPI for clusters of workstations and PCs. DMCS adds a small overhead to the communication operations provided by the lower communication system. In return DMCS provides a flexible and easy to understand application program interface for one-sided communication operations including *put-ops* and *remote procedure calls*. Furthermore, DMCS is designed so that it can be easily ported and maintained by non-experts.

1 Introduction

In this paper, we describe an application-driven design and implementation of a low-latency communication library for use in adaptive applications such as 3D unstructured mesh generation for crack propagation simulations on parallel computers [1]. We had two main goals in designing and implementing this library.

- *High-performance*: In particular, we wanted to provide low-latency communication operations.

*This work was supported by the NSF Career Award CCR-9876179, and by NSF grants CISE Challenge # EIA-9726388, Research Infrastructure # EIA-9972853, # ITR-0085969, # ACI-9612959, and IBM's Shared University Research Program.

- *Flexibility and ease of use*: We wanted to ensure that the library was useful for parallel adaptive and irregular numerical applications.

Existing communication paradigms and communication systems tend to fall into one of two broad camps regarding these two issues: those that are geared towards high performance at the expense of usability (e.g. Low-level Application Programming Interface (LAPI) [2]), and those that sacrifice performance in favor of easing the burden placed on the application programmer (eg., MPI [20], and software Distributed Shared Memory (DSM) systems like Treadmarks [3]). Both approaches present serious difficulties to the application developer who demands maximal performance and a high degree of maintainability but who does not possess the time or the desire to master the intricacies of complex communication systems.

MPI addresses portability and ease-of-use issues successfully by providing an attractive interface for the parallel programmer. However, it is not intended to be a target for run-time support systems software needed by compilers and problem solving environments since these systems require a very efficient (and perhaps inevitably, less friendly) communication substrate like LAPI. MPI also does not address issues like dynamic resource management, and concurrency at the uniprocessor level (threads).

These issues were addressed by a consortium known as *POrtable Run-time Systems* (PORTS) [4], and more recently, by the GRID community. PORTS consisted of research universities, national laboratories, and computer vendors interested in advancing research for software communication substrates that provide support for compilers and advanced tools like parallel debuggers for current and next generation supercomputers. Some of the goals of the PORTS group were the definition of standard applications programming interfaces (API's) for (i) one-sided communication*, (ii) integration of multi-threading with communication, (iii) dynamic resource management, and (iv) performance measurement.

In particular, the PORTS group experimented with four different approaches and API's for the integration of communication with threads [5]: (i)a *thread-to-thread* communication approach supported by CHANT [6], (ii) a *Remote Service Request* communication paradigm like Active Messages, supported by NEXUS [7], (iii) *hybrid* communication, supported by TULIP [8], and (iv) *DMCS*, which was initially presented in [9].

CHANT implements thread-to-thread communication on top of portable message passing software layers such as p4 [10], PVM [11], and MPI [20]. 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, called the *message thread*, polls for all registered receives. For portability, CHANT supports the first approach.

NEXUS decouples the specification of the destination of communication from the specification of the thread of control that responds to it. NEXUS supports the *Remote Service Request* (RSR) communication paradigm based on a remote procedure call mechanism, like Active Messages [12]. Messages are handled by *message handlers*; each message handler is a thread registered by the user or by the multi-threaded system, and invoked upon receipt of

*MPI-2 standard was not yet defined.

the message. The handler possesses a pointer to a user-level buffer into which the user wishes the message contents to be placed. Handler threads are scheduled in the same manner as computation threads.

TULIP's *hybrid* approach is essentially a combination of thread-to-thread and RSR-driven communication paradigm [8]. In the runtime substrate, TULIP provides basic communication via global pointers and remote service requests. Threads are introduced in the pC++ language level.

The development of DMCS was driven by the need for a runtime library for adaptive applications such as parallel adaptive mesh generation. DMCS attempts to combine efficiency, ease of use, and portability by using the following strategy:

1. *Performance*: DMCS is designed to provide unilateral communication primitives. These primitives exploit the low-latency constructs of the underlying communication subsystem and are optimized to handle the special requirements of adaptive applications.

   To achieve low-latency, we decided to support a single-threaded communication paradigm. In [9] we presented an implementation that supported multi-threading, but for performance reasons DMCS has to perform context-switching which is a hardware dependent operation and thus impairs the portability and maintainability of the system. The alternative was to extend DMCS's API to support any thread package. This approach increases the latency of the DMCS primitives as we show in Section 5.3.

   After four years of continuous use and experimentation with DMCS, and the development of four different adaptive and irregular applications, we decided to uncouple multi-threading from data movement. There are several advantages to this approach. The user is free to choose his own package, the latency of DMCS calls is minimized, and portability and maintainability issues are simplified. Furthermore, the integration of data movement with multi-threading can be easily accomplished outside DMCS. The disadvantage is that for SMP nodes, the user is forced to use a single communication thread. Our approach is to force communication through the main application thread; details are given in Section 5. This approach simplifies programming because the users do not need to worry about thread safety issues such as locking common data structures. To support object/data migration, we built on top of DMCS a software system called the Mobile Object Layer [16] which implements a global name space in the context of object/data migration. In this way the application suffers the additional 10% to 14% latency only if it is necessary.

2. *Maintainability and Portability*: DMCS is written entirely in ANSI C, and is designed in a modular fashion on top of a lower communication substrate. This reduces the amount of code that needs to be ported to new systems since only the lowest layers must be ported. Existing implementations on both LAPI and MPI provide examples of porting DMCS to different platforms.

3. *Flexibility and Ease-of-Use*: A simple and intuitive API that interoperates with widely used systems like MPI makes DMCS easy and useful tool to developers of parallel adaptive applications.

Finally, we decided to address fault-tolerance only at the application level and ignored authentication because we target MPPs and tightly coupled cluster of workstations where the network security is handled by other systems like Cluster CoNTroler.

The rest of this paper is organized as follows. Sections 2 describes in broad terms the applications that have driven the development of DMCS. First, we look at a parallel guaranteed-quality mesh generator [14], and then at multi-layer parallel runtime systems [15]. In Section 3.1, we examine the parallel execution model of DMCS, and compare it with the execution models of other parallel runtime systems. In Sections 3 and 5, we look at the architecture and the implementation of DMCS. In Section 4, we describe the application programmer interface (API) for DMCS. In Section 6, we analyze the overhead that DMCS imposes over the underlying low-level communication subsystem. Finally, in Section 8 we summarize our conclusions and we briefly describe our plans for the next version of the DMCS system.

2 Application Descriptions

The development of DMCS can be best understood by examining some of the applications in which it is used. In this section, we look at two such applications: 3D Adaptive Mesh Generation, and the Multi-layer Runtime System which is designed to tolerate large latency events and facilitate large scale, out-of-core, adaptive applications. We will describe why existing communication software and paradigms are often insufficient for such applications, and what operations and properties are necessary.

2.1 Adaptive Mesh Generation

Mesh generation is a basic building block for the discretization of partial differential equations (PDEs) and the generation of discrete linear systems of algebraic equations. A very successful approach for guaranteed-quality adaptive unstructured mesh generation is *Delaunay triangulation*. This algorithm generates unstructured meshes by adding new points on demand, thereby modifying the existing triangulation by means of purely local operations. The basic kernel for Delaunay algorithms is a four-step procedure [17] that is often called the Bowyer-Watson (BW) kernel [18, 19]. The first step, *point creation*, creates a new point by using an appropriate spatial distribution technique. The second step, *point location*, identifies an element containing this new point. The third step, *cavity computation*, removes existing elements that violate the Delaunay property. Finally, the fourth step, *element creation*, builds new triangles or tetrahedra by connecting the new point with old points such that the resulting triangulation satisfies certain geometric properties.

The parallel implementation of the BW algorithm for 3D domains starts with an initial Delaunay tetrahedralization of a set of points which is over-decomposed into $N \gg P$ subdomains (or *regions*), where P is the number of processors. Regions are assigned to processors in a way that maximizes data locality; each processor is responsible for managing multiple regions. The third step in the BW kernel is the source of unpredictable computation and communication because the number of elements and regions that participate in any given cavity varies. The number of elements in a cavity depends on the location of the newly inserted point, the elements themselves, and the partitioning of the existing elements. Synchronous communication deteriorates performance because it forces the computation to be executed almost sequentially, in phases. Moreover, it is difficult to use *any* binary communication protocol (in which explicit receives are required) because messages are sent with uncertain frequencies from uncertain sources.

By eliminating the problem of placing receives for unexpected data movement, asynchronous remote procedure calls and one-sided communication primitives improve performance and simplify the logic of the code. The third step of mesh generation requires the following computation: *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 usually done in a breadth-first order. Approximately 20% to 30% of the breadth-first searches touch non-local data elements. In this case, a remote procedure call with two to three arguments simplifies the complexity of the code substantially. Similarly, one-sided communication primitives like *put* or *put_op* are helpful to perform a remote write (or remote write plus a simple operation like *gather* or *scatter*) on remote memory without the participation of the application on the target side. For example, once all elements (if any) from the breadth-first search are found on a remote node, they must be stored in the proper memory location of the process or thread that made the request without having the application wait or look for them.

In addition, work-load imbalance is a source of problems whose solution requires flexible, one-sided, non-blocking, and asynchronous data movement primitives. Imbalance can occur due to refinement, remeshing, and setbacks. Mesh refinement takes place because of large variability in the error of the solution. For applications such as crack propagation, remeshing is required to handle changes in the topology and geometry of the mesh. Setbacks in the progress of the algorithm (in certain regions) occur because of concurrency. Concurrency is very useful, not only to exploit parallelism, but also to tolerate communication and synchronization latencies. However, concurrency can create many problems. Newly inserted points and newly created elements sometimes may have to be released (not used, even though they have been computed) because they violate certain rules required for the correctness and the quality of the mesh. The computation for creating and selecting these points and elements must be performed again in the future. This type of setback introduces additional sources of load imbalance which is exacerbated by the variable and unpredictable computation and communication patterns in each region of the mesh.

In summary, the communication requirements for adaptive applications require one-sided, non-blocking, and asynchronous data movement primitives like *get/put* and *get_op/put_op* and *remote procedure calls*. In addition, the latency of small size (half kilobyte) data movement primitives is very critical for the performance of adaptive applications. Figure 1(a) shows that the communication traffic due to small size messages for 3D unstructured mesh generation is more than 90% of the overall communication, and Figure 1(b) indicates that the the total time spent in message passing is about 15% of the total execution time. The time spent in other computation activities is also shown in Figure 1(b) for comparison; more detailed data is presented in Section 6.3.

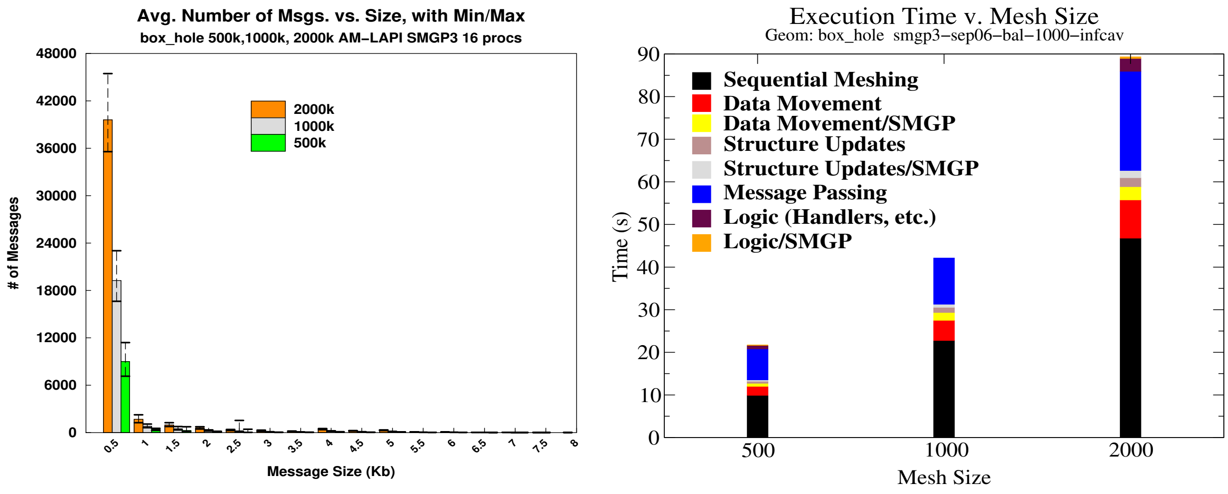

Figure 1: (a) Distribution of messages with respect to message size. (b) Breakdown of the total execution time of an adaptive application, parallel guarantee quality Delaunay triangulation.

2.2 Multi-layer Runtime System

Modern runtime systems for parallel computers provide another example of the need for efficient one-sided, non-blocking, and asynchronous communication. A major concern in these systems is that performance is becoming bound by large latency events such as disk reads and communication between processors because advances in network and disk technology have not kept pace with advances in processor performance. As a result, processors are wasting more and more cycles waiting for communication and I/O. This problem is only exacerbated by the types of applications that typically make use of parallel architectures. In particular, *out-of-core* applications must manipulate much more data than can fit in the combined memories of all of the processors in the parallel system or cluster. For these applications, masking the latency associated with reads from the disks is critical. Similarly, as we have seen in Section 2.1, applications that make use of *dynamic* and *unstructured* communication patterns depend on efficient communication libraries for good performance. To accommodate these application types, we developed the *Multi-layer Runtime System* [15] on top of DMCS.

The MRTS divides the hardware into two (or more) layers, with the lower layer acting as a data server for the upper layer which acts as as a computing engine. It is possible with such an approach to reserve faster processors for the upper layer, keeping slower processors or processors with larger amounts of memory for the lower layer (such a configuration may arise naturally when organizations choose to upgrade clusters or parallel machines with newer hardware, but still wish to make use of the older machines). The MRTS allows applications to create *percolation objects* which are nothing more than application-defined pieces of data with user-defined handlers. For example, a 3D mesh generator may define tetrahedra or mesh subregions to be percolation objects which have a user-defined handler for mesh refinement. As work becomes available for a percolation object (for example, from refining a mesh subregion), the percolation object migrates from the lower layer into the upper layer where the work is actually performed. Once this work is completed, the percolation object will migrate back into the lower layers, and will possibly be retired to disk. In this way, running the application results in a continuous migration of objects from the lower layer to the upper layer and back again.

Communication substrates that rely on binary communication semantics (such as MPI) are ill-suited for implementing such a system. Because of the unstructured nature of the application, percolation objects have no way to know where the communication is going to take place. Refinement in a particular mesh subregion may trigger changes in a neighboring subregion at any time, as described in Section 2.1. Building such an adaptive system on top of a communication substrate like MPI would place much of the communication burden on the application programmer, and would greatly increase the complexity of the code.

For these reasons, the MRTS makes use of the Mobile Object Layer [16] which provides single-sided communication with data migration. Application-defined data, called Mobile Objects, migrate from processor to processor in the parallel system in any application-defined manner. The Mobile Object Layer makes use of a distributed directory protocol in which messages are sent to processors where Mobile Objects are believed to reside, and then forwarded if this location turns out to be incorrect [16]. This directory protocol is heavily dependent upon remote procedure invocation, or sending a message to a remote processor which specifies a handler to be invoked upon message receipt. Typically, these messages are very small (refer to Section 6.3), and so low latency for small messages is crucial. Additionally, the Mobile Object Layer must be able to manipulate remote memory with *get/put* or *get-op/put-op* semantics that DMCS supports.

3 DMCS Architecture

DMCS is designed to meet two requirements in addition to the performance and application specific requirements described earlier: those of *maintainability* and *portability*. DMCS should not only deliver the lowest latency communication operations to the user possible, but should do so without resorting to constructs that would be difficult to implement on any given platform. In this section, we describe the architecture of DMCS, highlighting the features that simplify maintaining and porting DMCS to a wide variety of hardware and software platforms. We currently have versions of DMCS built using Active Messages [12] and LAPI [2] communication subsystems on the IBM SP family of parallel machines, as well as using MPI [20] on clusters of workstations.

3.1 Parallel Execution Model

First, we discuss the parallel execution model provided by LAPI, which runs on the IBM SP family of parallel machines, and MPI, which is a common message passing software package which can run on both dedicated parallel machines and clusters of workstations. Then, we contrast these with the DMCS execution model.

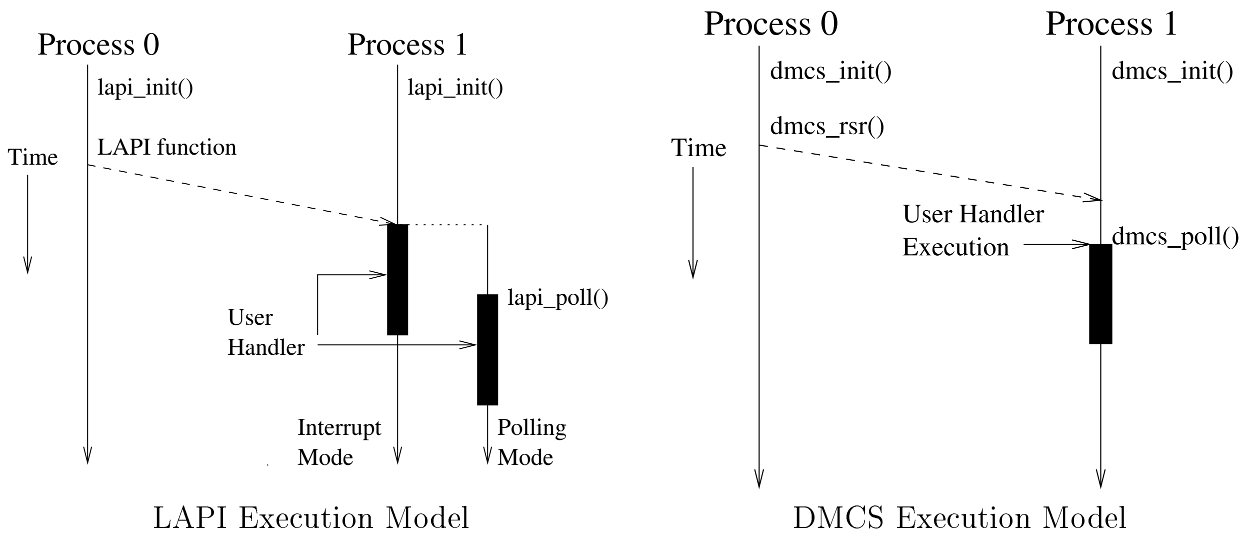

Figure 2: LAPI and DMCS Parallel Execution Models

The LAPI execution model involves two threads: (i) a user application thread, and (ii) a LAPI completion handler thread or completion thread that constantly polls the network for incoming messages and executes any user-defined handlers referenced by those messages. In contrast, the DMCS execution model, forces all user-defined handlers to execute in the main application thread. Also, by default, LAPI executes in *interrupt* mode. This means that as messages arrive, the main application thread is interrupted so that incoming messages may be handled. This contrasts with DMCS, which operates in *polling* mode in which messages are handled only during a *poll* operation.

In contrast to LAPI, the MPI execution model is inherently single-threaded. In this sense, it is closer to the execution model provided by DMCS. However, there are significant differences between the two. Most notably, MPI-1 provides only a binary communication protocol, which means that *send* operations originating at one node must be paired with explicit *receive* operation on the target node. MPI does provide several variations on this basic execution model, including non-blocking immediate send operations, and synchronous send operations, but these variations do not significantly alter the basic communication model of MPI. For many types of applications, particularly those that involve bulk data transfers, this is an acceptable communication model. For other application types, such as those that make use of dynamic runtime load balancing or unstructured communication patterns, a binary communication protocol is inappropriate. For these applications, the single-sided communication operations provided by DMCS are much more efficient.

DMCS provides a single-threaded execution model[†] and a one-sided communication paradigm. DMCS is itself single threaded and therefore only a single application thread may make use of DMCS during program execution. This single-threaded nature distinguishes DMCS from low-level communication software like LAPI and eases the burden placed on

[†]This means a single communication thread and many computation threads.

the application developer. Because user handlers execute in the main application thread, applications using DMCS do not need to worry about thread safety issues for application data structures.

The single-threaded execution model can also provide significant benefits in application performance. Because user handlers can only execute when a *poll* operation is executed by the application thread, they cannot interrupt computation. This is desirable because frequent context switching from taking interrupts can have a detrimental impact on performance, and may also thrash memory, causing an unnecessarily large number of page faults. By allowing user handlers to execute only when *poll* operations are executed, we can avoid such behavior.

DMCS also provides a single-sided communication paradigm, meaning that sending a message to a remote node does not require an explicit operation to receive the message on the target node. Applications making use of dynamic or unstructured communication patterns benefit greatly from this feature, as described before.

3.2 The Layered Approach to DMCS Architecture

We achieve portability and maintainability by splitting DMCS into the Messaging Layer (the ML) and the Application Program Interface (the API) layers. This layered approach reduces both the effort required to port DMCS to new platforms and the complexity presented to the application developer. The API layer is invariant across platforms and implementations, while the ML layer contains code that will need to be modified to port DMCS to platforms that support different communication paradigms and/or low-level communication subsystems. For example, as we will see in Section 5, the challenges in porting the DMCS communication primitives to a low-level system like LAPI, which supports the Active Messages communication paradigm, are completely different from the challenges found in porting DMCS to MPI, which implements a binary communication protocol.

The ML contains all hardware or software communication specific code, and its purpose is to isolate code that needs to be modified for portability reasons. The functionality provided by the ML closely parallels that which is provided by DMCS. This allows any optimizations that can be provided by the low-level communication subsystem to be propagated to the API layer, and ultimately to the application. For example, Active Messages provides message passing calls which take up to five machine word sized arguments. An implementation of the ML built on top of Active Messages would be able to efficiently make use of this functionality, while a ML that required argument marshaling would not. Because of the close parallels between DMCS and the ML, the ML is the largest layer in DMCS in terms of the number of lines of code.

The API provides the function set that applications use to interact with DMCS and subsequently with communication hardware. It also contains all of the source code that remains invariant across all platforms. In other words, code is included in the API layer if and only if it is completely independent from both the hardware and the low-level communication software. Most API functions have close parallels in the ML layer. Two exceptions to this rule are the *dmcs_malloc()* and the *dmcs_free()* functions. They can be built entirely from functions provided by the ML, and therefore have no direct counterparts in the ML code. Because of this design, the API layer is very thin, with many functions containing only a single line or are defined as macros, further lowering latency.

3.3 Threads in the DMCS Architecture

Threads are absent from the DMCS architecture. Adding threads to the DMCS architecture can be done in one of two ways: (i) threads themselves can be implemented in DMCS, or (ii) the API of DMCS can be extended to include the API of some third party thread package.

The first alternative would greatly impair, if not destroy, the portability of DMCS. Implementations of preemptive threads are heavily dependent upon the hardware architecture of the underlying processor. This is because of the need to save the state of a running thread when switching to a new thread, and the fact that the states of running threads are denoted by the register values of the processor. These register values contain such things as the stack pointer and the program counter, and vary widely from one processor architecture to another.

The second option, merely extending the API of DMCS, would allow any thread package to be used with DMCS. However, for now, we have opted against extending DMCS in this way for mainly philosophical reasons. DMCS is designed to implement efficient message passing; extending DMCS to provide threaded functionality does not aid it in its message passing role. Incorporating threads into DMCS would also have ramifications in the message passing performance. For example, allowing message passing from a threaded handler would necessitate that DMCS become thread-safe, which would add to the latency of message passing operations. This is in direct contrast with our previously stated goals. An alternative would be to enforce a policy stating that no message passing operations may be initiated from within a threaded handler.

Furthermore, because the development of DMCS is driven by the needs of target applications, we have to ask if applications would necessarily benefit from having threads incorporated into DMCS. At this point, it seems that applications would not benefit substantially. However, if this were to change, it may be possible to create a DMCS utility package that would incorporate an implementation of threads.

4 DMCS Application Program Interface

The functionality provided by DMCS can be broken into three broad categories. The first group is made up of *Environment* functions, and these are used to initialize and shutdown DMCS in an orderly fashion, as well as to query the DMCS environment for certain information, such as the number of processors in the parallel machine, or the rank of the calling processor[‡]. The second group of functions are *Remote Memory Manipulation* functions. Included in this group are *dmcs_malloc()* and *dmcs_free()*, which allow a process to allocate and later free memory on a remote node. Also included in this group are remote *read* and *write* operations, which allow processes to manipulate remote memory using read and write semantics. The third group is the *Remote Service Request* group. A *Remote Service Request*

[‡]DMCS ranks start at 0 and continue with consecutive integers up to the number of processes in the parallel system minus one.

is similar to a remote procedure invocation, but with an added restriction that a *Remote Service Request* cannot return any value. There are several RSR calls, each of which takes a different number of arguments. This allows DMCS to optimize the communication, possibly avoiding argument marshaling if the underlying communication layer makes it possible to do so. For example, DMCS built on top of Active Messages can take advantage of Active Messages functionality and avoid argument marshaling for up to four machine word size parameters. Again, removing unnecessary functionality (such as argument marshaling) allows DMCS to provide the lowest latency communication operations possible.

Environment manipulation functions like *dmcs_init()* and *dmcs_shutdown()* are responsible for the orderly startup and shutdown of the DMCS environment. Routines like *dmcs_num_procs()* and *dmcs_my_proc()* are used to query the environment for particular information such as the number of processes and process rank in the parallel system. The handler registration function *dmcs_register_handlers()* also falls into the category of environmental functions. Details of these and other DMCS API functions can be found in the DMCS web page [23]. A complete list of all functions with brief description can be found in Table 1.

Data movement functions provide remote *read* and *write* operations on a parallel system. DMCS provides two basic function types: *Get* and *Put* functions (*dmcs_get()* and *dmcs_put()*) which correspond to reads and writes, respectively. DMCS extends these basic function types with the concept of *Get-and-op* and *Put-and-op* functions, *dmcs_get_op()* and *dmcs_put_op()*, which allow users to specify operations to take place on the target nodes after a particular read or write has completed. Furthermore, DMCS provides both synchronous and asynchronous data movement functions, the default being asynchronous communication. The synchronous alternatives use the same names with the addition of *sync*. For example, the default *dmcs_put_op()* function becomes *dmcs_sync_put_op()* in its synchronous form.

Control functions, or Remote Service Requests, can be viewed as remote function invocations with the added restriction that remote functions are unable to return any value. As with remote memory manipulation functions, RSRs come in synchronous and asynchronous forms. The synchronous version will return only after the message has been received at the target node, but possibly before the user handler executes on the target. The asynchronous operation will return immediately, possibly before the message has been sent. For optimization purposes we have implemented RSRs with between zero and four (*dmcs_rsr0()* to *dmcs_rsr4()*) arguments for the user-defined handler. A version for arbitrary size data, *dmcs_rsrN()*, is available, but requires marshaling of the data (the arguments) into a contiguous memory buffer. Because of this, there is a higher amount of latency associated with this function.

5 DMCS Implementation

We will look at two implementations of DMCS, one built on top of LAPI for the IBM SP family of parallel machines, and another built for clusters of workstations using MPI for communication. Because both implementations must support the same API, several interesting construction details needed to be resolved. In the following subsections, we will

DMCS Environmental Functions	
dmcs_init	initialize DMCS
dmcs_shutdown	shutdown DMCS
dmcs_my_proc	the relative process ID of the calling process
dmcs_num_procs	the number of running processes
dmcs_register_rsr$X$_handlers	registers rsr$X$ type handlers where $X \in \{0, 1, 2, 3, 4, N\}$
DMCS Remote Service Request Functions	
dmcs_async_rsr$X$	RSR with an X argument handler where $X \in \{0, 1, 2, 3, 4\}$
dmcs_async_rsrN	RSR with a handler taking a variable size buffer
DMCS Remote Memory Manipulation Functions	
dmcs_malloc	allocate memory on a remote processor
dmcs_free	frees memory on a remote processor
dmcs_async_put	copy a data buffer to a remote processor
dmcs_async_put_op	copy a data buffer to a remote processor; the remote processor then returns with a dmcs_async_rsr1
dmcs_async_get	retrieve a data buffer from a remote processor
dmcs_async_get_op	requests a dmcs_async_put_op be executed on a remote processor

Table 1: A brief description of the DMCS API

look at these and other issues.

5.1 DMCS Implementation Using LAPI

The execution model of DMCS differs significantly from that of LAPI, so there are a number of challenges in implementing DMCS on top of this substrate. For example, the LAPI execution model mandates that user-defined handlers execute inside a LAPI completion handler thread, while DMCS handlers must execute in the main application thread. Another crucial difference concerns the time when user handlers must execute. LAPI executes in *interrupt* mode by default, meaning that user handlers execute as soon as messages arrive at the target node (a LAPI thread executes in the background, interrupting the application thread when a message arrives). On the other hand, DMCS handlers must only execute when the application performs a *polling* operation. Resolving these differences in the execution models is the primary challenge in implementing DMCS on top of LAPI.

5.1.1 Single-Threaded Execution Model With LAPI

In mapping DMCS to LAPI, the two challenges of (i) implementing the single-threaded model mandated by DMCS, and (ii) forcing all user-defined handlers to execute only during a *polling* operation are both addressed through the use of *delay tables* containing handlers.

When a *Remote Service Request* message arrives at a node, a LAPI header handler and

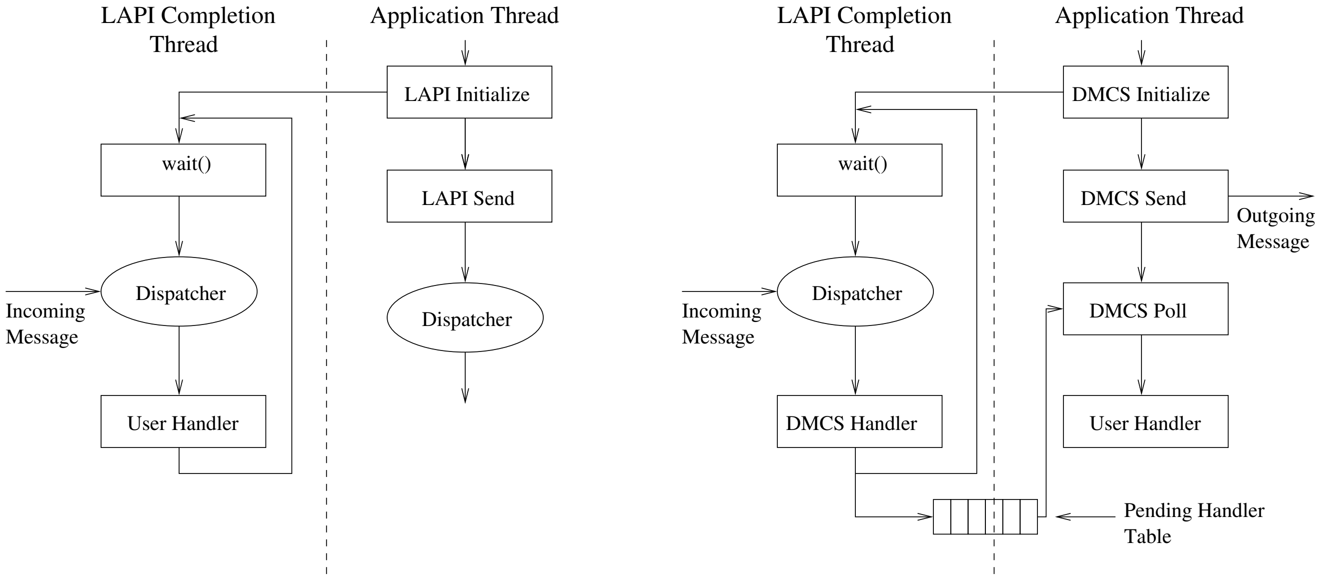

Figure 3: A closer look at the LAPI and DMCS parallel execution models

```
/* ---------------------------------------------------------------------
 * Function:  dml_rsrX_complhndlr()
 * Returns:   void
 * Description:
 *   LAPI completion handler.  Because dmcs is single threaded, we
 *   simply enqueue messages in a delay table so they can be dequeued
 *   and handled during a poll() from the main thread.
 * ---------------------------------------------------------------------
 */
void
dml_rsrX_complhndlr(lapi_handle_t *pHndl, void *pParam)
{
  dml_message_t *pMsg = (dml_message_t*)pParam;
  dml_delay_table_insert(pMsg->nSrc, pMsg->nSeq_num, (void*)pMsg);
}
```

Figure 4: DMCS RSR LAPI Completion Handler

```
/* * Construct a new message object */ pMessage->nType = RSRX_TYPE;
pMessage->nSrc = dml_my_proc(); pMessage->nTgt = tgt;
pMessage->nSeq_num = dml_get_sequence_number(tgt);
pMessage->nAsync_flag = DML_SYNC; pMessage->nRemote_handler_idx =
dml_lookup_handler(handler); pMessage->pCompletion_handler =
dml_rsrX_complhndlr; pMessage->nArgs[0] = nArg1;
```

Figure 5: Construction of a New Message Object

completion handler execute. The completion handler, running in the LAPI completion handler thread, simply inserts a data structure describing the user handler and any parameters into a delay table, instead of calling the user handler directly. This method has a couple of advantages. First, this insertion operation is very quick, thus freeing the LAPI completion thread to service other requests and preventing the network from backing up during times of peak traffic. Second, it allows DMCS to provide the necessary single-threaded execution model. The *polling* operation, which executes in the main thread, simply empties the delay table, executing any handlers that may be pending. With this design, only the delay table itself needs to be thread safe, and user applications do not need to worry about thread safety issues such as locking common data structures.

Figure 4 illustrates a completion handler for a *Remote Service Request*. The completion handler is passed a pointer to a message object from the header handler. This message object is inserted into the delay table using the source node identifier and the message sequence number as a key. The delay table is a hash table in order to make lookups as speedy as possible. When the application executes a *poll* operation, the delay table is flushed and any pending handlers are executed in the order specified by DMCS's message ordering strategy.

The same solution is used to handle *put-op* and *get-op* messages.

5.1.2 Message Ordering Strategy

LAPI, like many other low-level messaging systems, does not guarantee message ordering by default. Unfortunately, message ordering is crucial for the correctness of many applications, and therefore must be provided by DMCS[§].

Message ordering is provided via sequence numbers, which are appended to a message at transmission time and then checked upon receipt. Each processor maintains a list of sequence numbers, one for each other processor in the system. When a message is sent, the current sequence number corresponding to the target processor is included in the message, and that number is then incremented. Figure 5 shows the construction of a message for a DMCS *Remote Service Request* with a single argument, showing how sequence numbers are associated with messages.

[§] Message ordering is an example of functionality that is necessary, but is also platform specific. In other words, some low-level communication software may provide message ordering, and in such cases it should not be provided by DMCS. Such redundant functionality only adds to the overhead incurred by the runtime system. Therefore, message ordering functionality belongs in the platform-specific DML layer of DMCS.

Maintaining message ordering upon arrival is handled by the delay table mechanism. As messages arrive, they are inserted into the delay tables using their source node and sequence number as a key. The insertion algorithm is designed so that gaps will be left in the table if messages arrive out of order. In other words, if message n arrives before message $n-1$, there will be a gap left in the table where message $n-1$ should reside. When a *polling* operation is executed, the messages in the delay table are processed, beginning with the message with the next expected sequence number for each processor in turn. Messages are processed until a message with the next expected sequence number is not present in the delay table. This happens when either all messages that have arrived have been processed or when messages arrive out of order; in either case, processing stops till the next message in logical sequence is received.

5.2 DMCS Implementation Using MPI

Currently, MPI is the most widely distributed message passing system due to its portability and accessibility. These attributes were in fact the ideas behind the design and creation of MPI. However it is not always possible for generic MPI implementations to provide the performance of vendor tuned communication systems. On a system such as the IBM SP, MPI is built on top of the LAPI low-level communications system, and can therefore take advantage of the SP's high speed communication switch. MPI is also implemented for clusters of workstations, using TCP/IP to communicate between nodes. On such systems, performance may be subservient to portability; MPI implementations must be designed to run on significantly different platforms. In order to port applications to a wide variety of platforms, many of which were not specifically targeted by the application designers, we need an implementation of DMCS that is able to run on many systems, and yet adheres to the standards already imposed on DMCS. For this task, MPI was used for underlying message passing. By keeping the DMCS layer thin, it is possible to offer performance nearly equal to that of MPI, while providing a superior API and substrate with which the application developer may create adaptive and unstructured applications.

5.3 Single-Threaded Execution Model With MPI

MPI inherently provides a single-threaded execution model and therefore maps well to the single-threaded model provided by DMCS. There is no need for the enqueueing of messages to guarantee ordering and single-threaded execution like there is in the LAPI implementation of DMCS. The ordering of messages is left to the MPI layer of the system; with MPI, messages are guaranteed to be received in the order that they are sent so long as certain criteria are met. However, because MPI implements a binary communication protocol, an explicit *receive* must be posted to match the *send* request of a remote node. Because of the binary nature of MPI, the *receives* must contain certain information about the incoming message in order for that message to be received. This information includes the source of the message, the size of the message, and the MPI tag associated with the message. With DMCS, we do not know in advance the information of messages that must be posted, making it difficult to receive messages in order of arrival. Fortunately, MPI offers two probing functions (*MPI_Probe()*

and *MPI_Iprobe()*) that check the network for any incoming messages ready to be received by the polling node. If there are multiple messages available, the probe will return the information of the message that arrived earliest. DMCS is then able to receive that message and handle it appropriately. In accordance with the standards imposed on DMCS, the reception of messages takes place in the *dmcs_poll()* function call. The only exception to this rule occurs when endeavoring to avoid deadlock, and this is explained in greater detail in the next section.

5.3.1 Message Reception and Deadlock Avoidance

Efficient message reception is critical to maintaining high performance in the runtime system. For example, dynamically allocating memory to hold incoming messages can lead to unacceptable performance and wasted memory resources. DMCS, in order to make message reception as efficient as possible, makes use of a preallocated message pool. As a message arrives, a preallocated message object will store its contents, and will be used by DMCS to invoke the user-specified handler. This method requires all messages to be received in the same way, and therefore the encoding and decoding of messages must be very specific to ensure messages are handled in the proper manner. With the help of C macros, the type of message is encoded in the message. Upon reception of a message, this flag is extracted and examined in order to determine which DMCS level handler should be called to properly execute the message intent.

This is the encoding strategy employed by DMCS for all message types except for *Remote Service Requests*. Since MPI was created to run on many systems, memory mapping across distributed memory machines is not guaranteed. In order to execute a user-level handler on a remote node, that handler must be registered as a DMCS handler. This is described further in Section 5.3.2.

Because DMCS offers synchronous versions of all of its operations, the possibility of deadlock needed to be taken into account during implementation. Since DMCS is a single-threaded system, and the only time a DMCS message of any kind can be received is during a *dmcs_poll()*, deadlock can occur when two nodes send each other synchronous methods simultaneously. In this case, each node will be waiting for the signal that the message is received on the remote node, but, unfortunately, if both sides are waiting, then neither side is able to send the signal. To avoid this possibility of deadlock, a second polling function (one that is invisible to the programmer) had to be created. This polling function operates almost identically to *dmcs_poll()*, except that it receives only a single message at a time instead of ridding the network of all possible messages. In synchronous operations, there is a loop that waits for notification that the sent message has been received on the remote node. If this loop executes for a predetermined number of cycles without receiving this notification, the new polling function is called to determine if a message is on the network that may be causing the deadlock. If there is such a message, it is received and handled in the intended manner. Control is then returned to the synchronous operation to determine if the deadlock has been eliminated. This process cycles until the deadlock is resolved and execution continues as normal.

5.3.2 Handler Registration

Because DMCS is designed to run on a large variety of hardware and software platforms, it must make as few assumptions as possible about the underlying operating environment. One assumption that does not always hold true is the congruent mapping of parallel processes to memory on nodes in a parallel machine. On some platforms, a single process is assigned to each node in the parallel system, and it can be assumed that each process occupies the same memory addresses. Furthermore, corresponding data structures and corresponding functions also occupy the same addresses on each node. In other cases, processes and data are allocated to whatever memory may be available on a processor.

Such behavior has direct bearing on any parallel runtime system. For example, in the first case, function pointers can be passed between nodes with no modification. In the second case, however, function pointers passed between nodes are invalid and useless. In such an environment, special care must be taken when referencing functions and data on remote nodes.

The obvious solution to this problem is to use a level of indirection. User handlers must be registered at the time of DMCS initialization. The collective *dmcs_register_handlers()* operation creates an internal handler table which associates user handlers with small integer indices. In all DMCS operations that specify a user handler (such as a *Remote Service Request*), a translation takes place before the message is actually sent. The function pointer specified in the function call is converted to the handler index, which is then converted back to a function pointer on the remote node.

5.4 Optimizing with Preallocated Message Pools

Providing low latency communication operations and a suitably high level of performance often means minimizing the amount of dynamic memory allocation in the critical path of sending a message. To reduce the amount of dynamic memory allocation, DMCS makes use of preallocated message pools to send and receive messages. These message pools are allocated at system startup time, and provide message buffers to processes that wish to send or receive messages.

When a process wishes to send a message, it simply dequeues the head of the outgoing message pool. The outgoing message pool contains unused message buffers, which are contiguous regions of memory ready to be filled in with valid outgoing message field values. Once the message has been sent and the memory on the sending node is free to be modified, the message buffer is returned to the outgoing message queue, ready to be used for a subsequent message.

A similar strategy handles incoming messages. When a message arrives from the network, a preallocated message buffer is taken from an incoming message pool to store the message. Once the message is handled, the preallocated message buffer is returned to the message pool, to be used again by some future message.

Because the entries in the message pool are of a fixed size, they cannot store variable sized data, such as the data for a *Put* operation or for an *RSR* operation with more than four arguments. Storage to store this data needs to be allocated during the runtime of the program, but the responsibility for allocating the memory falls on a different party in each case. For a *Put* operation, the responsibility for making sure there is storage available falls on the application, while in the case of the *RSR* message, it is the responsibility of the runtime system to allocate the memory. If, during the course of execution, it can be determined that the dynamic memory allocation required to handle the variable sized buffer for the *RSR* message is hampering performance, the application can replace *dmcs_rsrN()* calls with *dmcs_put_op()* calls. This will allow the application to preallocate storage for the *Put* operation, which will copy the arguments to a known location, and then run a handler.

Another solution is for DMCS to provide preallocated message pools of user-specified sizes to store the incoming *RSR* argument buffers. This is an optimization that should be incorporated into DMCS, and it is discussed briefly in Section 8 of this paper.

6 Performance Analysis

In Section 3 we defined an execution model that simplifies message passing for parallel adaptive applications —i.e., satisfies the communication requirements we identified in Section 2; and at the same time permits a slim and clean design and implementation of the mid-level communication library, DMCS. In Sections 4 and 5 we presented an easy to understand software design and implementation which is easy to be maintained by non-experts. The only aspect of the communication system that remains to be examined is the overhead of the individual DMCS communication operations. The performance of DMCS operations can be gauged by examining the overhead of individual primitives within microbenchmarks and complete adaptive applications.

We evaluate the performance of DMCS primitives by looking into two microbenchmarks, for the evaluation of the most frequently used primitives: *Remote Service Request* and *Put_Op*. In the adaptive mesh generation, approximately 67% of messages are *remote service requests* and about 33% of messages are *Put_Op*. Also, we evaluate the success of the DMCS system from applications perspective. We use three communication adaptive applications which are communication intensive: a 3-dimensional Guarantee Quality adaptive Delaunay Tetrahedralization, dynamic Network Sort (netsort) kernel, and a Multi-Layer Runtime System [15]. We perform the evaluation of DMCS primitives on two different implementations: (i) DMCS implementation using LAPI, a low-latency communication substrate for SP machines and (ii) DMCS implementation using MPI which is a high-latency communication library for cluster of workstations and PCs. The evaluation of the three complete adaptive applications is performed only on the MPI implementation because of limited number of nodes[¶] on the SP machine.

6.1 Experimental Set-up

The DMCS/MPI performance figures were collected using two systems. The Linux numbers were collected from a network running 1GHz Pentium III machines with 128 megabytes and connected by 100 Mbit fast-Ethernet. The Solaris numbers were collected on a network

[¶]Our SP machine has only two 2-way PowerPC 604e 200 MHz nodes.

	DMCS Overhead	MPI Overhead	Total Time
dmcs_async_rsr0	1.000e-6	7.000e-6	1.000e-5
dmcs_async_rsr1	1.000e-6	1.000e-5	1.300e-5
dmcs_async_rsr2	1.000e-6	1.100e-5	1.400e-5
dmcs_async_rsr3	1.000e-6	9.000e-6	1.200e-5
dmcs_async_rsr4	1.000e-6	1.000e-5	1.200e-5

Table 2: Send times (in seconds) for DMCS RSRs. These include the DMCS overhead, the MPI (LAM) overhead, and the total time to execute the call. Tests were run on Intel PIII 1GHz machines running Linux connected by 100 Mb Fast Ethernet.

of Sun Ultra 5 machines with 333MHz processors connected by a 100 Mbit fast-ethernet network and with 256 megabytes of memory.

The DMCS/LAPI performance figures were collected using an IBM SP parallel machine with 200 MHz RS/6000 processors. Each node has 256 megabytes of memory and is connected with a SP-switch.

6.2 Microbenchmarks

This subsection describes the performance of two DMCS implementations for two of the most frequently used DMCS operations: *remote service request*, and *put_op*. We look at both the MPI implementation and the LAPI implementation, and we are able to show that DMCS does not add significant overhead in either case.

To demonstrate just how thin the DMCS layer is, we have run experiments on a few members of the API on two separate platforms. It is apparent from the Tables 2 to 5 that DMCS offers very little overhead with respect to the total execution time as well as the MPI overhead. This can also be observed in the graphs of Figure 6 and Figure 7. What is even more interesting is that the DMCS overhead time is completely independent of message size. The DMCS overhead from Table 3 are consistently $1e-6$ seconds, despite sending an 8K message. This implies and accurately reflects that DMCS uses no unnecessary memory allocation, deallocation, or copying. Similar numbers can be seen on the experiments run on Solaris, and both imply that as the message size increases, the percentage of total execution time spent in the DMCS layer decreases. For a one byte message on the Linux cluster, the DMCS overhead for total execution time is approximately 2%. Similarly, for an 8k message, the percentage drops to approximately 0.5% of the total execution time. It is apparent from these tables and graphs, that DMCS offers the smallest overhead possible while providing the strength of a one-sided asynchronous paradigm.

Table 6 depicts the send times for DMCS RSR operations with a fixed number of machine-word sized parameters and Put-op times for different sized message payloads. The total time is broken into several categories: the DMCS overhead, which contains the time spent in DMCS code; the LAPI overhead, which contains the time spent executing LAPI polls, handlers, and other operations; and the total time spent in the DMCS operation as perceived

	Size (bytes)	DMCS Overhead	MPI Overhead	Total Time
dmcs_async_put_op	1	1.000e-6	3.700e-5	3.900e-5
dmcs_async_put_op	64	1.000e-6	3.700e-5	4.000e-5
dmcs_async_put_op	512	1.000e-6	5.600e-5	6.000e-5
dmcs_async_put_op	4096	1.000e-6	2.320e-4	2.360e-4
dmcs_async_put_op	8192	2.000e-6	3.890e-4	3.950e-4

Table 3: Send times (in seconds) for DMCS Put-Ops. These include the DMCS overhead, the MPI (LAM) overhead, and the total time to execute the call. Tests were run on Intel PIII 1Ghz machines running Linux connected by 100 Mb Fast Ethernet.

	DMCS Overhead	MPI Overhead	Total Time
dmcs_async_rsr0	2.000e-6	6.700e-5	7.300e-5
dmcs_async_rsr1	2.000e-6	6.400e-5	6.900e-5
dmcs_async_rsr2	2.000e-6	7.000e-5	7.500e-5
dmcs_async_rsr3	2.000e-6	6.900e-5	7.400e-4
dmcs_async_rsr4	2.000e-6	6.300e-5	6.800e-5

Table 4: Send times (in seconds) for DMCS RSRs. These include the DMCS overhead, the MPI (LAM) overhead, and the total time to execute the call. Tests were run on Sun Ultra 5 333Mhz machines running Solaris connected by 100 Mb Fast Ethernet.

	Size (bytes)	DMCS Overhead	MPI Overhead	Total Time
dmcs_async_put_op	1	3.000e-6	1.180e-4	1.260e-4
dmcs_async_put_op	64	3.000e-6	1.170e-4	1.250e-4
dmcs_async_put_op	512	5.000e-6	4.770e-4	4.870e-4
dmcs_async_put_op	4096	3.000e-6	6.430e-4	6.490e-4
dmcs_async_put_op	8192	3.000e-6	9.160e-4	9.240e-4

Table 5: Send times (in seconds) for DMCS Put-Ops. These include the DMCS overhead, the MPI (LAM) overhead, and the total time to execute the call. Tests were run on Sun Ultra 5 333Mhz machines running Solaris connected by 100 Mb Fast Ethernet.

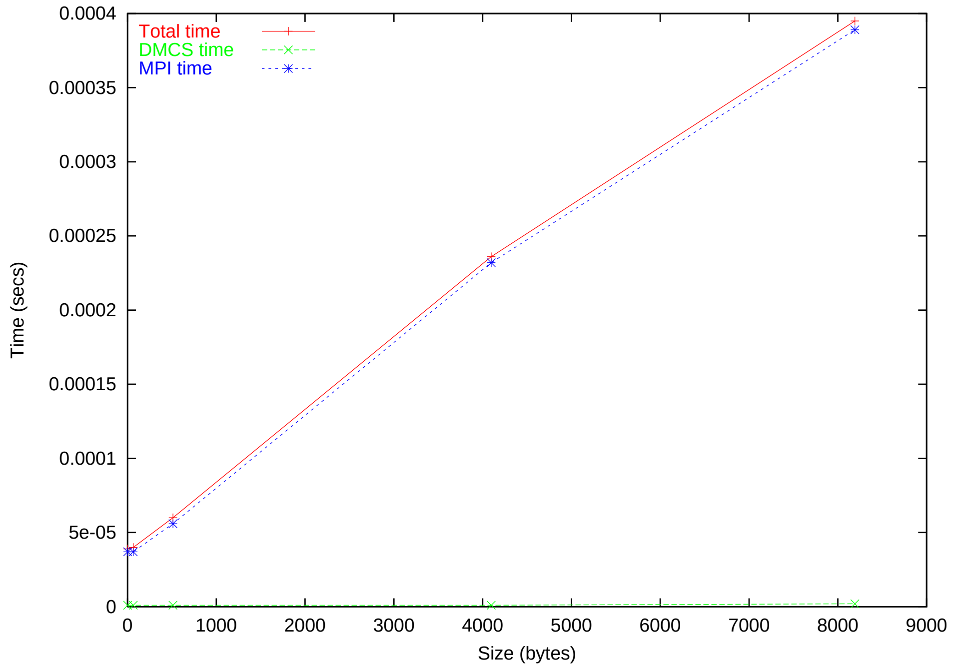

Figure 6: A plot of the Put-Op times. The graph contains plots of DMCS overhead, MPI overhead, and total execution time. Tests were run on Intel PIII 1Ghz machines running Linux connected by 100 Mb Fast Ethernet.

by user code. However, there are several things that must be noted when examining these numbers. First is the difficulty in measuring LAPI time. Specifically, the time spent moving the user data from the Network Interface (NIC) into the kernel, and finally into user space cannot be measured without access to the LAPI implementation code. Because we do not have such access, we measured LAPI calls by simply wrapping timers around them. While this serves as only an approximation, it still allows us to view trends in the timings. Secondly, the DMCS overhead plus the LAPI overhead does not equal the total user time. This is due to several factors which we are not able to measure, including thread context switch time, kernel-level polling time, and the time to run the LAPI dispatcher. Every LAPI call attempts to make progress on any pending messages by running the dispatcher function, either in the user's main thread on the LAPI completion handler thread. This function does not execute instantaneously, and therefore adds time to the perceived user time.

In Table 7, we can see that the DMCS overhead time remains fairly constant for each operation. This is due to the fact that no copies of parameters must be made. Also, we see that the LAPI overhead and the total execution time are constant, due to the fact that each operation simply fills in a system structure, which is the same size no matter the number of parameters sent. Importantly, we can see that the DMCS overhead is in the range of 10% of the LAPI overhead we could measure; using the total LAPI overhead, including context switching that takes place in the LAPI layer, the actual DMCS should be substantially smaller, for each operation.

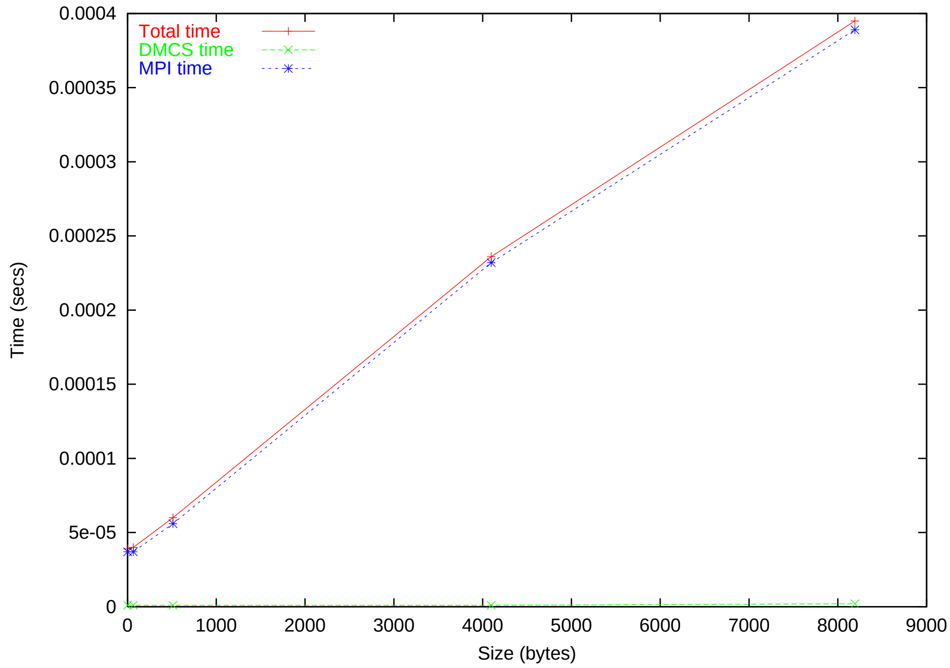

Figure 7: A plot of the Put-Op times. The graph contains plots of DMCS overhead, MPI overhead, and total execution time. Tests were run on Sun Ultra 5 333Mhz machines running Solaris connected by 100 Mb Fast Ethernet.

Table 7 depicts the performance of DMCS Put-op operations with message payloads of various sizes. Again, the DMCS overhead remains fairly constant as the message size increases. This once again demonstrates the fact that there are no message copies within DMCS, allowing it to propagate the performance of the underlying communication substrate to the user. As the message size increases, we can see that the LAPI overhead grow, along with the total user-level time. As before, the DMCS overhead added to the LAPI overhead does not equal the total user-level time, due to unmeasurable operations within LAPI, and thread context switch time between the user thread and the LAPI completion handler thread. Overall, DMCS overhead adds between roughly 1% and 10% of the LAPI overhead we could measure; using the total LAPI overhead, including context switching that takes place in the LAPI layer, the actual DMCS should be adding a substantially smaller percentage to the LAPI overhead.

6.3 Adaptive Applications

In this subsection we evaluate the DMCS overhead in the context of two complete applications and a netsort kernel we described in Section 2. The first application is a 3-dimensional Parallel Guaranteed Quality Delaunay Triangulation [14] and the second application is a Multi-layer Runtime System [15] that is intended to build parallel, out-of-core adaptive mesh generation codes. The third application is a netsort kernel which consist of two par-

	DMCS Overhead	LAPI Overhead	Total Time
dmcs_async_rsr0	2.103e-6	34.024e-6	67.963e-6
dmcs_async_rsr1	2.465e-6	35.590e-6	74.554e-6
dmcs_async_rsr2	4.126e-6	38.565e-6	60.837e-6
dmcs_async_rsr3	2.480e-6	24.968e-6	67.191e-6
dmcs_async_rsr4	2.483e-6	22.706e-6	70.022e-6

Table 6: Send times (in seconds) for DMCS RSRs. These include the DMCS overhead, the LAPI overhead, and the total time to execute the call. Tests were run on an IBM SP parallel machine with 200MHz RS/6000 processors.

	Size (bytes)	DMCS Overhead	LAPI Overhead	Total Time (secs)
dmcs_async_put_op	1	4.321e-6	45.018e-6	102.321e-6
dmcs_async_put_op	64	3.264e-6	36.350e-6	126.644e-6
dmcs_async_put_op	512	3.535e-6	25.721e-6	123.515e-6
dmcs_async_put_op	4096	4.607e-6	128.725e-6	232.500e-6
dmcs_async_put_op	8192	5.050e-6	329.448e-6	372.904e-6

Table 7: Send times (in seconds) for DMCS Put-Ops. These include DMCS overhead, LAPI overhead, and the total time to execute the call. Tests were run on an IBM SP parallel machine with 200MHz RS/6000 processors.

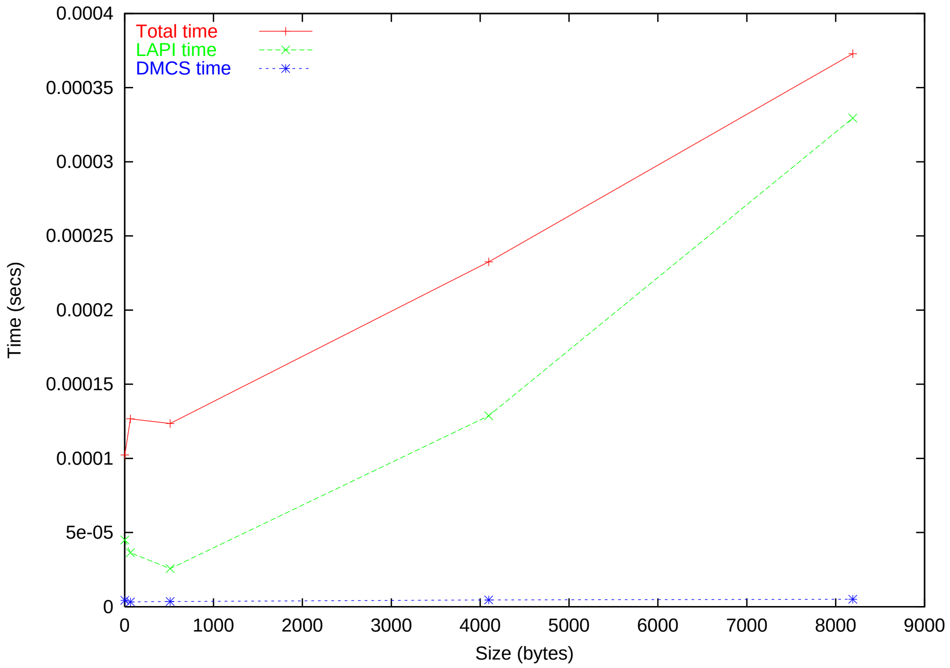

Figure 8: A plot of the Put-Op times. The graph contains plots of DMCS overhead, LAPI overhead, and total execution time. Tests were run on an IBM SP parallel machine with 200MHz RS/6000 processors.

allel network sort routines using two different scenarios: *static netsort* and *mobile netsort* where parts of the data to be sorted move around randomly. Data might have to move in different processors in order to minimize load imbalance of the processors.

Guaranteed Quality Delaunay Triangulation (GQDT): The performance of the 3-dimensional Parallel GQDT depends heavily upon efficient communication for the computation of Delaunay cavities which contain tetrahedra owned by more than one processor. *Distributed* cavities are constructed by a distributed breadth-first search algorithm over the mesh to locate violated tetrahedra, which contain a newly inserted point, p, in their circumspheres. The communication is *one-to-many*, in that the processor containing the newly inserted point p may, in order to compute the cavity around this point, sends many messages to other processors containing subdomains which own tetrahedra violated by the newly inserted point. In section 2.1 we have seen that the communication required for computing the distributed cavities is variable and unpredictable so the standard message passing techniques do not apply.

Table 8 shows the minimum, average, and maximum percent of distributed cavities, over 16 processors for for a half, one, and two million elements. The last column depicts the average number of distributed cavities per processor. This table shows that even if each distributed cavity required only few messages to its neighbor processors, the overhead of an inefficient message passing system would be significant. On average, each distributed cavity sends fourteen messages.

Size	Min. (%)	Avg. (%)	Max. (%)	Avg. #
0.5M Elements	12	19	26	1720
1M Elements	12	19	31	3705
2M Elements	15	18	22	7415

Table 8: Percent of distributed cavities and the average number of distributed cavities, per processor.

			DMCS Overhead			*MPI Overhead*		
P	# Tets	Time	MIN	AVE	MAX	MIN	AVE	MAX
2	1M	162	.1023	.1028	.1034	20.53	20.63	20.73
4	1M	95	.0848	.0960	.1127	22.13	22.60	23.40
4	2M	185	.1428	.1542	.1769	37.45	38.50	40.18
8	1M	61	.6790	.0874	.1150	19.70	21.66	24.39
8	2M	111	.1024	.1351	.1850	32.00	35.82	40.90
8	4M	208	.1430	.2055	.2886	49.71	57.38	67.16
16	1M	40	.3785	.6863	.8575	19.32	27.04	29.89
16	2M	71	.0665	.1060	.1312	28.44	30.75	34.02
16	4M	128	.0974	.1642	.2066	44.45	49.23	57.23
16	8M	240	.1670	.2662	.3768	76.02	82.78	91.59

Table 9: Performance data from parallel adaptive mesh generation on two to sixteen processors, generating one to eight million tets. Total execution time with the DMCS and MPI overheads are shown. All tests run on Sun Ultra 5 296Mhz nodes running Solaris connected by 100 Mb Fast Ethernet.

Object Size	Total	DMCS Overh
32 Bytes	0.6	3.1e-02
512 Bytes	1.1	6.2e-02
8192 Bytes	1.3	7.2e-02
32768 Bytes	1.8	8.9e-02
65536 Bytes	2.0	9.6e-02

Table 10: Percolation time and overhead for a single object. Tests run on Sun Ultra 5 333Mhz machines running Solaris connected by 100 Mb Fast Ethernet.

Table 9 depicts different mesh runs for 2, 4, 8 and 16 nodes (Sun Ultra 5 296Mhz machines running Solaris) connected by 100 Mb Fast Ethernet. The size of the meshes varies from one million tetrahedra (Tets) to eight million Tets. The total execution time is measured in seconds. The DMCS overhead as well as the MPI overhead are measured in seconds and the minimum (MIN), average (AVE), and maximum (MAX) overhead per run are listed. The DMCS latency over the MPI overhead on average varies from 0.5% (one million elements generated in two nodes) to 2.5% (one million elements generated in 16 nodes). The maximum DMCS overhead over the total execution time is between less than 0.1% and 1.7%. Also, it is apparent from these data that the communication overhead is a clear source of imbalance. This issue has been studied in [22].

Multi-Layer Runtime System: We evaluate the impact of the DMCS overhead upon the MRTS. In this test, we migrate a single object (it can be a subregion or a block of a large matrix) through the entire percolation cycle, with the runtime system executing on two processors contained within two different machines. The percolation cycle begins with reading the data object from disk, and injecting it into the cycle. The Initiator Module writes the data parcel into the local "Hot" data buffer, and inserts a token referring to the parcel into the parallel heap. Next, the Assembler Module, which also executes on the Data Server processor, moves the data into the Computing Engine layer of the runtime system. This involves moving the object from the "Hot" data buffer into a local buffer on the Computing Engine, and moving the token which refers to it from the parallel heap into an appropriate upward moving parallel queue. The third stage, the Scheduler, is responsible for executing the computation pending for the percolating data. This is the only stage of the cycle which executes on the Computing Engine processors. Finally, the Terminator is responsible for retiring the data which has just finished percolating and writing it back to disk. This finishes a single cycle through the MRTS system. We examine the time required to complete the cycle for objects of three different sizes. The handlers executed for each data parcel does nothing, and therefore does not contribute to the overall runtime. Table 10 shows that the DMCS overhead is about 0.2% of the total time it takes to percolate a single object.

Network Sort: We have implemented two versions of a parallel network sorting algorithm, one which migrates objects after each stage of the sorting algorithm (*mobile netsort*) and one which does not (*static netsort*). Both the static and mobile netsort routines are are

Processors	*Linux Cluster* Total	MPI Time	DMCS Overh.	*Solaris Cluster* Total	MPI Time	DMCS Overh.
2 Procs	4.010	3.225	0.141e-1	9.451	7.228	0.287e-1
4 Procs	5.716	3.312	0.191e-1	25.920	13.222	0.432e-1
8 Procs	65.756	41.744	0.384	62.871	43.717	0.411e-1

Table 11: Static netsort times in secs for a Linux and Solaris cluster of workstations using MPI (LAM). Tests run on Intel PIII 1Ghz machines running Linux and Sun Ultra 5 296 Mhz machines running Solaris. Both cluster use 100 Mb Fast Ethernet.

Processors	*Linux Cluster* Total	MPI Time	DMCS Overh	*Solaris Cluster* Total	MPI Time	DMCS Overh
2 Procs	21.209	4.377	0.222e-1	176.405	23.167	0.592e-1
4 Procs	19.161	4.361	0.263e-1	160.671	20.302	0.459e-1
8 Procs	22.989	7.158	0.220e-1	159.220	24.562	0.549e-1

Table 12: Mobile netsort times in secs for a Linux and Solaris cluster of workstations using MPI (LAM). Tests run on Intel PIII 1Ghz machines running Linux and Sun Ultra 5 296Mhz machines running Solaris. Both clusters use 100 Mb Fast Ethernet.

communication intensive kernels and very good test to stretch the DMCS implementation to its limits. The static netsort implementation begins by creating a number of integers that we wish to sort, and randomly assigning them to processors in the parallel system. We then move through a series of steps, where each integer is compared with its "neighbor" and exchanged if necessary, moving the integers with lower values toward one end of the array and integers with higher values toward the other. The aspect that makes this application parallel lies in the fact that the array exists across all processors, and an integers neighbor may lie on another processor. By the end of this process, the integers in the array are in sorted order. The second implementation uses this same algorithm, but migrates the integers to new, random processors after each comparison, thereby continually redistributing the array. Tables 11 and 12 depict the MPI and DMCS overheads which is varies from 0.04% to 0.9%. The DMCS and MPI overheads are higher on the Solaris cluster because the nodes are much slower.

7 Application-driven Evolution of DMCS

The API for DMCS reached its current form by being used and critiqued routinely for the past four years in the context of two adaptive applications and one parallel runtime system: Structured Adaptive Mesh Refinement, Unstructured Adaptive Mesh Generation and Refinement [14], and a parallel runtime system for multi-layer parallel architectures [16]. The DMCS API has also been influenced by the PORTS consortium meetings in the mid

1990s, the Generic Active Messages API [12], Tulip [8] and Nexus [7] the last of which are resulted from the efforts of the PORTS consortium meetings.

In contrast with the current API, DMCS originally provided an API based on the concept of the *global pointer*. A global pointer was defined by a local pointer and a DMCS *context* pair. The DMCS context was a unique integer identifier that was assigned at initialization time to each DMCS processor or context. Remote data were then accessed through a global pointer. This approach is elegant and familiar to the application programmer, but has certain disadvantages for adaptive applications. First, the access of remote data depended on the ability to determine unique context numbers. In cases where dynamic resource management is required, this approach needed major revisions. Second, for adaptive applications, the global pointers need to be maintained by the application. This is due to data migration and dynamic load balancing. This left us with two choices: either let the application maintain global pointers, which would add complexity to the application, or augment DMCS to maintain global pointers, which would increase DMCS's complexity and thus complicate its maintenance. For this reason, we separated data movement and control from global pointer functionality and developed a new layer on top of DMCS called the *Mobile Object Layer* (MOL) which supports global pointers in the context of data migration [16]. Application developers can choose to use DMCS without having to use the MOL.

Asynchronous and non-blocking message passing can be much more efficient than synchronous communication. However, asynchronous communication can lead to subtle race conditions. Early versions of DMCS used acknowledgement variables to enable applications to find out about the status of completion for data transfers. Acknowledgement variables could have three states: *cleared*, *set*, and *uninitialized*. To use an acknowledgement variable, the application had to first request one using the routine *dmcs_newack()*, which would return an unused acknowledgement variable. This could then be used as a handle to perform various operations. For example, *dmcs_testack()* checks if the acknowledgement variable has been set or not, and return immediately. On the other hand, *dmcs_waitack()* would wait until the variable in question has been set before returning. It was also possible to clear an acknowledgement variable using *dmcs_clearack()*. Finally, it was possible to anticipate the use of an acknowledgement variable in future data transfers using the *dmcs_anticipateack()* function. This last routine had an important role in one-sided data transfer operations.

Data movement routines like *get* and *put* were asynchronous and used acknowledgement variables to determine the state of data transfers. For example, a *get* operation transferring data from a source specified by a global pointer to a destination specified by a local pointer would *set* an acknowledgement variable when the transfer operation was complete. In a similar manner, a *put* operation used to transfer data from a location referenced by a local pointer to one referenced by a global pointer would have three acknowledgement variables associated with it. A *local_ack* is set when the local data buffer can be reused by the application program. A *remote_ack* is set on the processor that initiated the *put* operation to indicate that the put operation on the remote processor is complete. Finally, a *remote_remote_ack* is set on the remote processor to signal that the *put* operation has completed. For this remote processor communication to work correctly, the remote node must first anticipate the *put* operation by calling *dmcs_anticipateack()* on the acknowledgement variable specified as *remote_remote_ack*.

Although the use of acknowledgement variables provided a very flexible method for signalling the completion of data transfer operations, it ultimately proved to be somewhat confusing to application developers. The lessons we learned from this implementation of DMCS allowed us to develop the current API, which defines simple and clear semantics. Currently, instead of requiring the user to request and test acknowledgement variables, we explicitly provide synchronous and asynchronous versions of the API functions. Such a method has proven to be easier to understand and use correctly by application developers, while not reducing the functionality provided by DMCS. Also, with the addition of *Put-and-op* and *Get-and-op*, the semantics of the earlier versions can be retained, but in a much more concise and easily understood manner.

8 Conclusions

We have described the design and implementation of a Data Movement and Control Substrate for network-based, homogeneous communication within a single multiprocessor or tightly couple workstations and PCs. DMCS implements a one-sided communication API for message passing. The DMCS system serves three objectives: (i) it isolates large-scale and expensive parallel applications from vendor-specific communication subsystems at the cost of small overhead, less than 3% when MPI is used as an underline communication layer and less than 10% when LAPI is used, (ii) it is flexible for adaptive applications that require many low-latency small messages, and (iii) finally, DMCS is easy to understand and port by non-experts. Subsequently can be integrated in large-scale environments and be part of codes that expected to have along life-time.

Our recent experience from porting a parallel mesh generator that was developed on a Unix cluster to a Windows 2000 cluster suggests that even if MPI is used as an underline communication layer portability is not automatic. Different MPI implementations incorporate different optimizations [‖] that impact that correctness and performance of the applications substantially. In this case DMCS is used as a "buffer" to these subtle differences in various MPI implementations and it guarantees first correctness and second seamless portability of very large parallel codes.

The current DMCS version supports a single-threaded communication model, it is not a fault-tolerant and does not allow dynamic resource allocation. The next version will support multi-threaded communication model in order to allow integration of adaptive simulations with visualization and other I/O devices. It will be ported on top of VIA for PCs and Windows 2000 in a way that dynamic processor allocation will be allowed and it will be fault-tolerant communication system.

[‖] We have found that commercial implementations of MPI make assumptions that improve MPI performance but might lead to programming that increases application complexity. Our experience suggests that DMCS is the best layer to absorb all complexity that relates to communication and execution model.

9 Acknowledgements

Many colleagues and experts during the last five years contributed in this work and we are thankful to all. Induprakas Kodukula for his valuable contributions during the first implementation of the PORTS API. Pete Beckman, Ian Foster, Dennis Gannon, Matthew Haines, L. V. Kale, Carl Kesselman, Piyush Mehrotra, and Steve Tuecke for very productive and alive discussions in the mid 90's on the PORTS API and implementation issues. Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken for very helpful insight for the implementation of Active Messages on the SP-2 machine. Finally, IBM's Research Program and Marc Snir for helping Prof. Chrisochoides in his effort to acquire a small but extremely useful for this project SP machine.

References

[1] Bruce Carter, Chuin-Shan Chen, L. Paul Chew, Nikos Chrisochoides, Guang R. Gao, Gerd Heber, Antony R. Ingraffea, Roland Krause, Chris Myers, Démian Nave, Keshav Pingali, Paul Stodghill, Stephen Vavasis, Paul A. Wawrzynek. Parallel FEM Simulation of Crack Propagation – Challenges, Status, *Lecture Notes in Computer Science 1800, pp. 443-449, Springer-Verlag* 2000.

[2] DiNicola P, Gildea K, Govindaraju R, Mirza J, Shah G; LAPI Architecture Definition: Low Level API Draft, *IBM Confidential Report*, December 1996.

[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, Vol. 29, No. 2*, pp. 18-28, February 1996

[4] Portable Runtime System (PORTS) consortium, http://www.cs.uoregon.edu/research/paracomp/ports/

[5] A Proposal for PORTS Level 1 Communication Routines, http://www.cs.uoregon.edu/research/paracomp/ports

[6] Matthew Haines, David Cronk, and Piyush Mehrotra, On the design of Chant : A talking threads package, *NASA CR-194903 ICASE Report No. 94-25*, Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23681-0001, April 1994.

[7] Ian Foster, Carl Kesselman and Steven Tuecke, The NEXUS approach to integrating multithreading and communication, *Argonne National Laboratory, MCS-P494-0195.*

[8] Pete Beckman and Dennis Gannon, Tulip: Parallel Run-time Support System for pC++, http://www.extreme.indiana.edu.

[9] Nikos Chrisochoides, Induprakas Kodukula, and Keshav Pingali Data Movement and Control Substrate for parallel scientific computing, *Workshop on Communication and Architectural Support for Network-based Parallel Computing*, February 1997.

[10] Ralph M. Butler, and Ewing L. Lusk, *User's Guide to p4 Parallel Programming System* Oct 1992, Mathematics and Computer Science division, Argonne National Laboratory.

[11] A. Belguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpore, PVM: Experiences, current status and future direction. *Supercomputing'93 Proceedings*, pp 765–6.

[12] Thorsten von Eicken, Davin E. Culler, Seth Cooper Goldstein, and Klaus Erik Schauser, Active Messages: a mechanism for integrated communication and computation *Proceedings of the 19th International Symposium on Computer Architecture, ACM Press*, May 1992.

[13] Chichao Chang, Grzegorz Czajkowski, Chris Hawblitzell and Thorsten von Eicken, Low-latency communication on the IBM risc system/6000 SP. *Supercomputing '96 Proceedings.*

[14] Nikos Chrisochoides and Démian Nave, Parallel guaranteed-quality h-refinement and mesh generation *p and hp Finite Element Methods: International Journal for Numerical Methods in Engineering,* To be submitted spring 2001

[15] Kevin Barker and Nikos Chrisochoides Multi-Layer Runtime System, To be submitted to *Concurrency Practice and Experience*, Spring 2001.

[16] Nikos Chrisochoides, Kevin Barker, Démian Nave, and Chris Hawblitzel Mobile Object Layer: A Runtime Substrate for Parallel Adaptive and Irregular Computations. *Advances in Engineering Software, Vol 31 (8-9), pp. 621-637,* August, 2000.

[17] F. P. Preparata, I. M. Shamos. Computational Geometry, An Introdu ction, 1985.

[18] A. Bowyer. Computing Dirichlet Tessellations. *The Computer Journal*, Vol. 24, No. 2, pp 162–166, 1981.

[19] Watson, D., Computing the n-dimensional Delaunay tessellation with applications to Voronoi polytopes, *The Computer Journal*, Vol. 24, No. 2, pp 167–172, 1981.

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

[21] Paul Chew, Nikos Chrisochoides, Guang Gao, Tony Ingrafea, Keshav Pingali, and Steve Vavasis, Crack Propagation on Teraflop Computers, unpublished manuscript, Cornell University 1997. NSF Proposal.

[22] Nikos Chrisochoides, Nashat Mansour, and Geoffrey Fox, Comparison of optimization heuristics for the data distribution problem. *Concurrency Practice and Experience, Vol 9(5)*, May, 1997.

[23] DMCS Homepage: http://www.cs.wm.edu/∼jdobbela/dmcs/