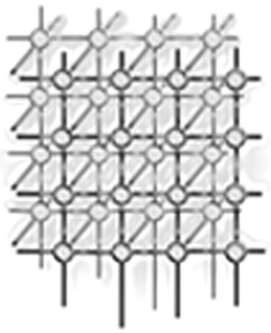
CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2002; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

Data Movement and Control Substrate for Parallel Adaptive Applications



Kevin Barker, Nikos Chrisochoides[†], Jeffrey Dobbelaere, Démian Nave, and Keshav Pingali

Computer Science, College of William and Mary, Williamsburg, VA 23185 US
Computer Science , Cornell University, Ithaca, NY 14853-3801 US
Computer Science and Engineering, University of Notre Dame South Bend, IN 46556 US.

SUMMARY

In this paper, we present the *Data Movement and Control Substrate* (DMCS), a library which implements low-latency one-sided communication primitives for use in parallel adaptive and irregular applications. DMCS is built on top of low-level, vendor-specific communication subsystems such as LAPI for IBM SP machines, as well as on widely available message-passing 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. Furthermore, DMCS is designed so that it can be easily ported and maintained by non-experts.

KEY WORDS: Message Passing, Runtime System, Parallel Adaptive Application, Mesh Generation.

1. Introduction

In this paper, we describe the design and implementation of a low-latency communication library for use in adaptive applications such as parallel, three-dimensional, unstructured mesh generation for crack propagation simulations [10].

The design and implementation of this library were motivated by three main concerns.

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

*Correspondence to: College of William and Mary, Williamsburg, VA 23185 US.
[†]E-mail: nikos@cs.wm.edu
Contract/grant sponsor: National Science Foundation,; contract/grant number: Career Award #CCR-0049086, ITR #ACI-0085969, RI # EIA-9972853, CISE Challenge #EIA-9726388, and ACI-9612959.

Received 11 March, 2001
Revised 11 September, 2001

- *Flexibility and ease of use*: We wanted to ensure that the library was useful for parallel adaptive and irregular numerical applications.
- *Portability*: We wanted to make DMCS as portable as possible to a wide variety of communication substrates.

Existing communication 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. Low-level Application Programming Interface (LAPI) [33] [23]), and those that sacrifice performance in favor of easing the burden placed on the application programmer (eg., MPI [27], and software Distributed Shared Memory (DSM) systems like Treadmarks [1]).

Both of these camps 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. For example, MPI addresses portability and ease-of-use issues successfully by providing an attractive interface for the parallel programmer, but it is not intended to be a target for runtime 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. Also, MPI does not supports a flexible RPC communication paradigm which simplifies the development of runtime systems for dynamic and unstructured applications. Finally, MPI does not address the issue of dynamic resource management. A new MPI standard known as MPI-2 [26] provides some basic one-sided functionality. Many MPI distributions [24, 36, 32] have implemented the bulk of the one-sided communication specified in the MPI-2 standard; however, none of them implements remote method invocation, an essential construct for writing many asynchronous parallel applications.

The DMCS effort grew out of a consortium known as *POrtable Run-time Systems* (PORTS) [30]. 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 —the MPI-2 standard was not yet defined, (ii) integration of multi-threading with communication, (iii) dynamic resource management, and (iv) performance measurement.

The PORTS group experimented with four different approaches and API's for the integration of communication with threads [31]: (i)a *thread-to-thread* communication approach supported by CHANT [22], (ii) a *Remote Service Request* communication paradigm like Active Messages, supported by NEXUS [21], (iii) *hybrid* communication, supported by TULIP [3], and (iv) *DMCS*, which was initially presented in [14]. Other systems with similar or even broader objectives have been developed [11, 5, 7, 29, 17].

CHANT implements thread-to-thread communication on top of portable message passing software layers such as p4 [9], PVM [4], and MPI [27]. 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 [20]. 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 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 [3]. In the runtime substrate, TULIP provides basic communication via global pointers and remote service requests. Threads are introduced in the pC++ language level.

DMCS attempts to combine efficiency, ease-of-use, and portability by using the following strategy.

1. *Performance*: DMCS is designed to provide one-sided 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.
2. *Single-threaded implementation*: To achieve low-latency, we decided to support a single-threaded communication paradigm. In [14] we presented an implementation that supported multi-threading, but for performance reasons, DMCS has to perform context-switching which is a operating system 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. *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.
4. *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.

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 [35]. The Cluster CoNTroller software system, developed at Cornell Theory Center and sold by MPI Softtech, is made up of secure resource management services and a deterministic heterogeneous scheduling algorithm. This system allows users to specify the administrator defined features that they require on compute nodes for their job.

It should be noted that DMCS is not designed to replace LAPI, MPI, or any other communication subsystem. DMCS is a thin, mid-level library that serves to isolate the

application, and therefore the application developer, from the details of the underlying communication substrate while allowing applications to take advantage of any specialized or high-performance communication features that may be present. DMCS leaves all communication decisions like multiple communication protocols available on SMP clusters up to the underlying messaging system. The goal of DMCS is to provide dynamic and unstructured applications with the single-sided communication operations that they need while making it as easy as possible to port applications across platforms. We currently have versions of DMCS built using Active Messages [20] and LAPI [33] communication subsystems on the IBM SP family of parallel machines, as well as using MPI [27] on clusters of workstations.

This paper describes the DMCS library and its performance. Section 2 describes in broad terms the applications that have driven the development of DMCS. First, we look at a parallel guaranteed-quality mesh generator [16], and then at multi-layer parallel runtime systems [2]. In Section 3, we describe the application programmer interface (API) and parallel execution model of DMCS, and compare it with the execution models of other parallel runtime systems. In Sections 4, we look at the architecture and the implementation of DMCS. In Section 5, we analyze the overhead that DMCS imposes over the underlying low-level communication subsystem. In Section 6, we discuss alternative API functions and implementations that we considered and explain why we rejected them. Finally, in Section 7 we summarize our conclusions and we briefly describe our plans for the next version of the DMCS system.

2. Motivating Applications

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: (i) three-dimensional Adaptive Mesh Generation, and (ii) a Multi-layer Runtime System (MRTS) designed to tolerate large-latency events and facilitate large-scale, out-of-core, adaptive applications. We describe why existing communication software and paradigms are often insufficient for such applications, and argue that what is needed is (i) a shared global address space and (ii) giving processors the ability to make remote service requests.

2.1. Adaptive Mesh Generation

Mesh generation is a basic building block for the numerical solution of partial differential equations (PDEs). One successful approach for guaranteed-quality adaptive unstructured mesh generation is *Delaunay triangulation* [34]. Delaunay triangulation refines unstructured meshes by adding new points on demand, thereby modifying an existing triangulation by means of purely local operations. The basic kernel for Delaunay algorithms is a four-step procedure that is often called the Bowyer-Watson (BW) kernel [8, 37]. 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.

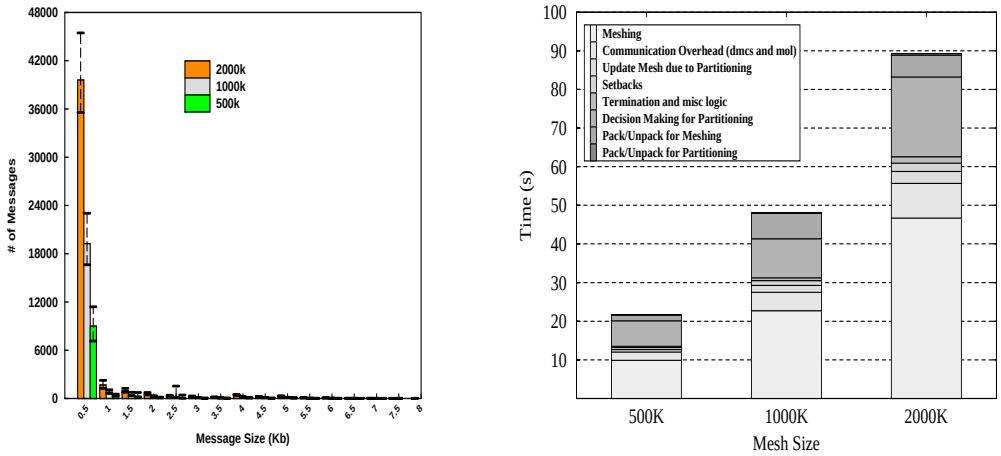


Figure 1. (a) Distribution of messages with respect to message size. (b) Breakdown of the total execution time of an adaptive parallel Delaunay triangulation using the BW kernel.

The parallel implementation of the BW algorithm for three-dimensional domains starts with an initial Delaunay tetrahedralization of a set of points. This tetrahedralization 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. It is the third step in the BW kernel that is the source of unpredictable computation and communication. This step 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. The number of elements in a cavity depends on the location of the newly inserted point, and on the existing elements themselves; furthermore, approximately 20% to 30% of the breadth-first searches touch non-local data elements. Synchronous communication deteriorates performance because it forces the computation to be executed almost sequentially, in phases. Moreover, it is difficult to use *any* two-sided communication protocol (communication in which explicit receives are required) because messages are sent with uncertain frequencies from uncertain sources.

Asynchronous remote procedure calls and one-sided communication primitives improve performance and simplify the logic of the code because they eliminate the problem of placing receives for unexpected data movement (one-sided communication primitives like *put* or *put_op* 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 a portion of the breadth-first search are found on a remote node, they can be stored in the memory of the process or thread that initiated the search, without having that process or thread wait or look for these elements.

Work-load imbalance is another problem 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. Specifically, some of newly-inserted points have to be removed because their corresponding cavities (i.e., elements that violate the Delaunay property due to a newly-inserted point) intersect and thus destroyed because the triangulation of intersecting cavities will lead to a non-conforming or non-Delaunay mesh.

n summary, adaptive applications require one-sided, non-blocking, and asynchronous data movement primitives such as *get/put*, *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 three-dimensional unstructured mesh generation is more than 90% of the overall communication. Furthermore, Figure 1(b) indicates that the the total time spent in message passing is about 15% of the total execution time; so optimizing this time is important.

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. While advances in network technology, such as Myrinet [6] and Giganet [19] have dramatically improved the network bandwidth available to the application, system software must still be able to effectively mask the latency associated with network communication. As a result of the growing gap between the latency associated with processor-to-memory communication and processor-to-disk communication, 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 disks is critical. To accommodate these application types, we have developed the *Multi-layer Runtime System*(MRTS) [2] 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 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 DMCS and the Mobile Object Layer [15] which provide single-sided communication in the context of 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 (MOL) 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 [15]. 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 5.2), and so low latency for small messages is crucial. Additionally, the MOL must be able to manipulate remote memory with *get/put* or *get-op/put-op* semantics that DMCS supports.

3. DMCS Application Program Interface and Architecture

We now present the DMCS API and architecture which was motivated by the considerations described in Section 2. A complete list of all functions with brief description can be found in Table 1. Details of these functions can be found in the DMCS web page [18].

3.1. DMCS API

The functionality provided by DMCS can be broken into three broad categories.

The first category 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 information such as the number of processors in the parallel machine, and the rank of the

calling processor†. 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.

The second category 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. 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.

The last category is concerned with *Remote Service Requests*. A *Remote Service Request* is similar to a remote procedure invocation, but with the 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. Removing unnecessary functionality (such as argument marshaling) allows DMCS to provide the lowest latency communication operations possible. As with remote memory manipulation functions, RSR's 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.

3.2. DMCS Architecture

In this section, we describe the internal architecture of DMCS, highlighting the features that simplify maintaining and porting DMCS to a wide variety of hardware and software platforms.

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

Table I. A brief description of the DMCS API

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 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
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

3.2.1. Parallel Execution Model

A key design decision we took was make the DMCS execution model single-threaded.

There are two primary reasons for this. First, adding threads to DMCS goes against the philosophical grain of the software. DMCS is designed to provide an efficient communication abstraction layer which provides dynamic and unstructured applications with the single-sided communication operations that they most need. We do not believe that support for multiple threads helps in achieving this goal. Furthermore, adding threads support to DMCS will most likely cause two unfavorable outcomes: increasing latency during message passing or forcing applications to conform to a more complicated runtime model. Forcing DMCS to be thread-safe adds cycles to the critical path of message passing due to the fact that we must lock internal data structures and provide mutual exclusion.

The second reason we have elected to provide only a single-threaded runtime model with DMCS is portability. There are two methods we could use to provide threads support with DMCS; we could either provide threads ourselves or we could provide an API which allows any third party threads package (such as *pthreads* [25][28]) to be used. Providing our own threads package severely hampers the portability of DMCS, due to the fact that the Operating System

and the processor architecture play a pivotal role in context switching between threads [‡]. Developing our own threads library would therefore come into conflict with our stated goal of designing DMCS in such a way that it can be ported to new platforms by non-experts. In a similar way, simply providing an API that allows any third party threads package to be used in conjunction with DMCS does not guarantee consistent application behavior across different platforms. It also forces anyone attempting to port DMCS to a new platform to port along two different "axes", namely message passing and threads, which only complicates the porting process.

In short, we have elected to leave threads out of our DMCS design because it does not aid in our goal of providing efficient single-sided communication operations and because it greatly complicates the task of porting DMCS. If, in the future, we determine that our target applications would benefit from the availability of multiple threads, we can either add thread support to DMCS via a separate utility module or through an entirely separate library. Because DMCS is single-threaded, all user-defined handlers must execute in the main application thread. This thread executes calls to *dmcs_poll*() to poll for the arrival of messages. Polling 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.

In contrast, 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. 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.

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 more efficient.

[‡]The importance of the Operating System can be seen from the fact that threads designed for the same processor often have different context switching semantics under various operating systems, such as Unix and Windows. Knowledge of the underlying processor architecture is necessary in order to save the registers and program state during a context switch.

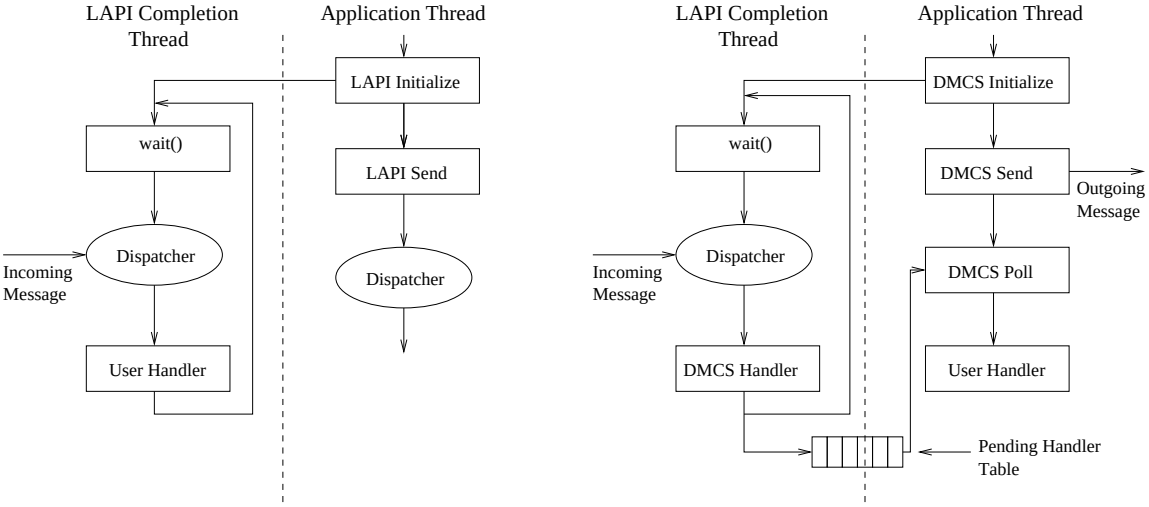


Figure 2. A closer look at the LAPI and DMCS parallel execution models

4. DMCS Implementation

As we will see in this section, 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. We achieve portability and maintainability by splitting DMCS into the Messaging Layer (the ML) and the Application Program Interface (the API) layers. The API layer is invariant across platforms and implementations. 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.

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 this section, we will look at these and other issues.

4.1. 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. In addition, LAPI executes

```
/* ----------------------------------------------------------------------
 * 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 3. DMCS RSR LAPI Completion Handler

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.

We deal with these issues as follows.

Single-Threaded Execution Model With LAPI

When a *Remote Service Request* message arrives at a node, a LAPI header handler and 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 3 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

```
/* * 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 4. Construction of a New Message Object

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.

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 4 shows the construction of a message for a DMCS *Remote Service Request* with a single argument, showing how sequence numbers are associated with messages.

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

[§] 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 ML layer of DMCS.

arrive out of order; in either case, processing stops till the next message in logical sequence is received.

4.2. DMCS Implementation Using 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. 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. 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. To make message reception as efficient as possible, DMCS makes use of a preallocated message pool. When 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. 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 4.3.

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 may occur if two nodes invoke synchronous DMCS methods at the same time. 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 extracting all pending 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 continues till the deadlock is resolved and execution continues as normal.

4.3. 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. On some parallel platforms, application and system programs reside at the same memory addresses on every processor. In that case, the sender of a message can simply use a function pointer to specify which handler should be invoked when that message is received. However, on other platforms, handlers can reside at different addresses on different processors. 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 indexes. 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.

4.4. 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 optimization will be incorporated into the next version of the DMCS implementation.

5. Performance Analysis

In this section, we measure the performance of DMCS operations, using microbenchmarks and complete adaptive applications. We use microbenchmarks to evaluate DMCS primitives on two different implementations: (i) DMCS implementation using LAPI, and (ii) DMCS implementation using MPI. The evaluation of the three complete adaptive applications is performed only on the MPI implementation because we did not have access to a large enough SP machine.

The DMCS/LAPI performance figures were collected using an IBM SP parallel machine with two 2-way 200 MHz PowerPC 604e processors with 256 MBytes of memory per processor.

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 Mb/s fast-Ethernet. The Solaris numbers were collected on a network of Sun Ultra 5 machines with 333MHz processors connected by a 100 Mb/s fast-ethernet network and with 256 megabytes of memory.

5.1. Microbenchmarks

To demonstrate how thin the DMCS layer is, we executed two of the most frequently used DMCS operations: *remote service request*, and *put_op*. It is apparent from Tables II and III that DMCS suffers 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 5.

Note that the DMCS overhead time is completely independent of message size. The DMCS overhead from Figure 5 are consistently 1 microsecond, despite sending an 8K message. This

¶The "send time" measured in the experiments refers to the amount of time spent in a method invocation before control is returned to the user-level program.

Table II. Send times (in microseconds) 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/s Fast Ethernet.

	DMCS Overhead	MPI Overhead	Total Time
dmcs_async_rsr0	1.000	7.000	10.000
dmcs_async_rsr1	1.000	10.000	13.000
dmcs_async_rsr2	1.000	11.000	14.000
dmcs_async_rsr3	1.000	9.000	12.000
dmcs_async_rsr4	1.000	10.000	12.000

Table III. Send times (in microseconds) 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/s Fast Ethernet.

	DMCS Overhead	MPI Overhead	Total Time
dmcs_async_rsr0	2.0	67.0	73.0
dmcs_async_rsr1	2.0	64.0	69.0
dmcs_async_rsr2	2.0	70.0	75.0
dmcs_async_rsr3	2.0	69.0	74.0
dmcs_async_rsr4	2.0	63.0	68.0

reflects the fact that DMCS uses no unnecessary memory allocation, deallocation, or copying. Also note that DMCS is not charged with the time for data transfer which allows the DMCS overhead to be fairly consistent no matter the amount of data being sent. Any discrepancy between numbers is simply network and processor noise created during the experiments. Numbers similar to those found on the Linux cluster can be seen in the experiments run on Solaris.

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 suffers little overhead while providing the power of a one-sided asynchronous paradigm.

Table IV 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 by user code.

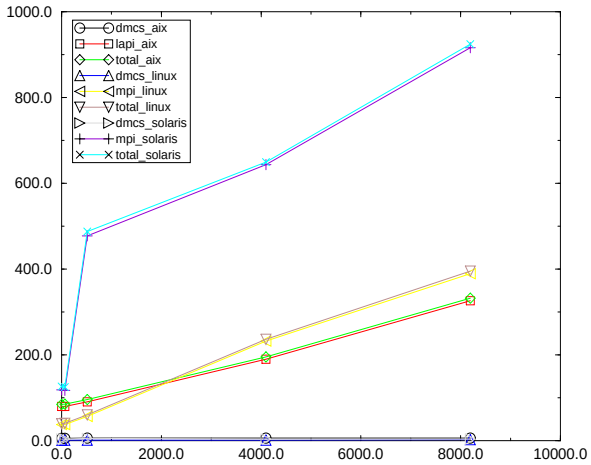


Figure 5. A plot of the Put-Op times. The graph contains plots of DMCS overhead, MPI/LAPI overhead, and total execution time. Tests were run on Sun Ultra 5 333Mhz machines running Solaris connected by 100 Mb/s Fast Ethernet, Intel PIII 1Ghz machines running Linux connected by 100 Mb/s Fast Ethernet, and an IBM SP parallel machine with 200MHz RS/6000 processors.

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 Figure 5, 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 5 depicts the performance of DMCS Put-op operations with message payloads of various sizes. Once 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 grows 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 because we could not measure the time taken by internal LAPI operations accurately. Overall, DMCS overhead adds between roughly 1% and 10% of the LAPI overhead we could measure; this is obviously an even smaller percentage of the total LAPI overhead.

Figure 6 depicts the performance of DMCS polling under LAPI and MPI. Part A provides the LAPI times; these are the totals and the averages for 10000 messages sent. The total LAPI time is the time spent inside LAPI poll (even if no message was received). Actual LAPI time is the time spent inside LAPI poll when there was actually a message there to receive. When messages arrive at the target processor during a LAPI_Probe() operation, two handlers are executed. The first is the header handler, and it is called when the first packet of a multi-packet message arrives. It provides LAPI with the address of a buffer to store the incoming message. The completion handler is called when all of the packets have arrived, and it builds a DML data structure containing the new message. The completion handler then inserts this data structure in the delayed table (which is the delay table insert time).

Figure 6(a) depicts the breakdown of a dmcs_poll() in the LAPI implementation. Inside of a dml_poll(), we check to see if there is anything in the delayed table (this is the delayed table check time). If there is, we get the first item. This is the delayed table removal time. From the data structure retrieved from the delayed table, we get the index of the handler. We use this index to look up the handler address in our handler table. This is the get handler index time. Then the user handler is immediately called.

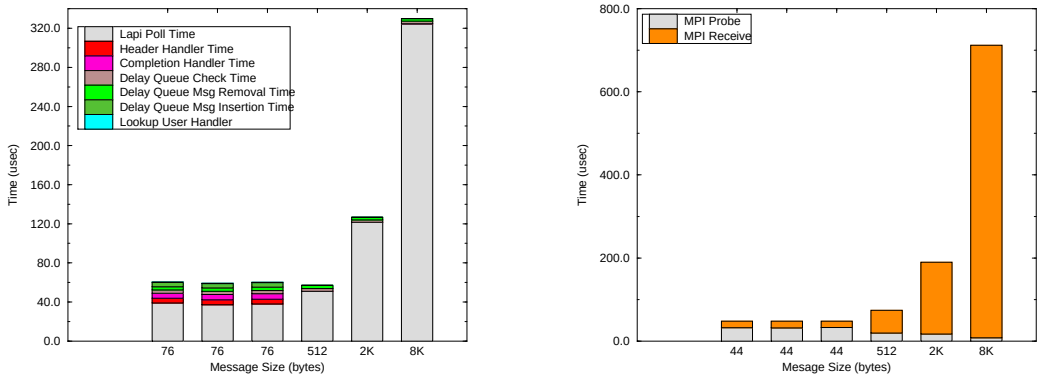
Figure 6(b) depicts the breakdown of the dmcs_poll() in the MPI implementation. It is immediately apparent that the MPI version has a much simpler version of the polling routine. Because MPI guarantees message ordering if a few rules are followed, there is no need to create delayed queues in which to put incoming messages. The dmcs_poll() therefore is implemented simply by a MPI_Probe() followed by an $MPI\_Recv()$ if there is message.

5.2. Adaptive Applications

In this subsection, we evaluate the DMCS overhead in the context of two complete applications and a netsort kernel. The first application is a 3-dimensional Parallel Guaranteed Quality Delaunay Triangulation [16] and the second application is a Multi-layer Runtime System [2] that is intended to build parallel, out-of-core adaptive mesh generation codes. The third application is a netsort kernel which consist of two parallel 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 to minimize load imbalance of the processors.

Table IV. Send times (in microseconds) 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.

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



(a) LAPI Implementation (b) MPI Implementation

Figure 6. Breakdown of time spent in dmcs_poll( ) in the (a) LAPI implementation and (b) MPI implementation. The bar-charts (from left to right) depict the time spend for receiving $dmcs\_async\_rsrX$ and $dmcs\_async\_rsrN$ with X $\in \{0, 1, 4\}$ and N $\in \{512, 2K, 8K\}$ bytes, respectively.

Guaranteed Quality Delaunay Triangulation (GQDT): The algorithm for this application was described in Section 2.1. 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. Cavities are constructed by a distributed breadth-first search algorithm over the mesh to locate every tetrahedron which contains the newly inserted point (say p) in its circumsphere and which violates the Delaunay property. The communication is *one-to-many* because the processor containing the newly inserted point

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

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

			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

p may send many messages to other processors containing tetrahedra violating the Delaunay condition; it is unpredictable because it is not known a priory where these tetrahedra are.

Table V shows the minimum, average, and maximum percent of distributed cavities, over 16 processors for a half, one, and two million elements. The last column depicts the average number of distributed cavities per processor. On average, each distributed cavity sends fourteen messages. Therefore, the overhead introduced by inefficient communication is substantial.

Table VI shows different mesh runs for 2, 4, 8 and 16 nodes (Sun Ultra 5 296Mhz machines running Solaris) connected by 100 Mb/s Fast Ethernet. The size of the meshes varies from one million tetrahedra to eight million tetrahedra. 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

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

Object Size	Total	DMCS Overhead
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

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.

Multi-Layer Runtime System: This runtime system was described in Section 2.2. We evaluate the impact of the DMCS overhead upon the MRTS [2]. 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 a Mobile Object (MO) into the local data buffer, and inserts a token referring to the MO into a parallel heap which is used for scheduling work at the Computing Engine layer. Next, the Assembler Module, which also executes on the Data Server processor, moves the data into the Computing Engine layer of the runtime system. In 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 VII 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 communication intensive kernels and are good tests of DMCS. 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

Table VIII. 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/s Fast Ethernet.

	Linux Cluster			*Solaris Cluster*		
Processors	Total	MPI Time	DMCS Overh.	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 IX. 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/s Fast Ethernet.

	Linux Cluster			*Solaris Cluster*		
Processors	Total	MPI Time	DMCS Overh	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

migrates the integers to new, random processors after each comparison, thereby continually redistributing the array. Tables VIII and IX 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.

6. Lessons from DMCS implementations

The DMCS API reached its current form after being used and critiqued for the past four years. It has also been influenced by the PORTS consortium meetings in the mid 1990s, the Generic Active Messages API [20], Tulip [3] and Nexus [21].

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*. 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. The first disadvantage is that 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. The second disadvantage is that 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 [15]. 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. In earlier versions of DMCS, 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 that transfer operation was complete. To use an acknowledgement variable, the application had to first request one using the routine *dmcs_newack()*, which would return a new acknowledgement variable. This could then be used as a handle to perform various operations. For example, *dmcs_testack()* checked if the acknowledgement variable had been set, and returned immediately.

Although the use of acknowledgement variables provided a very flexible method for signaling 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.

7. Conclusions and Future Work

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. We believe that DMCS 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 the correctness and performance of the applications substantially. We have found that some MPI implementations 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 the implementation of communication and execution model. In this case DMCS is used as a "buffer" to these subtle differences in various MPI implementations and it guarantees correctness and portability of large parallel codes.

The current DMCS version is not fault-tolerant and it does not allow dynamic resource allocation. The next version will support a multi-threaded communication model 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, and it will permit dynamic processor allocation; it will also use a fault-tolerant communication system.

Acknowledgments 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. Finally, the anonymous referees who with their suggestions helped to improve the presentation of this paper.

REFERENCES

1. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, . Yu, and W. Zwaenepoel TreadMarks: Shared Memory Computing on Networks of Workstations *IEEE Computer, Vol. 29, No. 2*, pp. 18-28, February 1996
2. Kevin Barker and Nikos Chrisochoides Multi-Layer Runtime System, To be submitted to *Concurrency Practice and Experience*, Spring 2002.
3. Pete Beckman and Dennis Gannon, Tulip: Parallel Run-time Support System for pC++, http://www.extreme.indiana.edu.
4. 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.
5. Angelos Bilas and Edward Felten, Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface, *Journal of Parallel and Distributed Computing*, Vol.40, No. 1, pp 138–146, 1997.
6. N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. IEEE Micro, 15(1):29–36, February 1995.
7. L. Bouge, J. Mehaut, and R. Namyst. MADELEINE: an efficient and portable communication interface for rpc-based multithreaded environments. *In Proc. of the 1998 Conf. on Parallel Architectures and Compilation Techniques, PACT'98*, pages 240–247, Paris, France, 1998.
8. A. Bowyer. Computing Dirichlet Tessellations. *The Computer Journal*, Vol. 24, No. 2, pp 162–166, 1981.
9. 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.
10. 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.
11. B. Chamberlain, S. Choi and L. Snyder, 1997. IRONMAN: An Architecture Independent Communication Interface for Parallel Computers, *University of Washington TR UW-CSE-97-04-04.*
12. Paul Chew, Nikos Chrisochoides, Guang Gao, Tony Ingrafea, Keshav Pingali, and Steve Vavasis, Crack Propagation on Teraflops Computers, unpublished manuscript, Cornell University 1997. NSF Proposal.
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, 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.
15. 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.
16. 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,* (Submitted, 2001)
17. David C. DiNucci. Cooperative Data Sharing: A Layered Approach to an Architecture-Independent Message-Passing Interface, *In Proceedings of the Second MPI Developer's Conference, Notre Dame*, July 1996, IEEE, pp. 58-65.
18. DMCS Homepage: http://www.cs.wm.edu/pes/software/dmcs/dmcs.html
19. Emulex Corporation http://www.emulex.com
20. 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.
21. Ian Foster, Carl Kesselman and Steven Tuecke, The NEXUS approach to integrating multithreading and communication, *Argonne National Laboratory, MCS-P494-0195.*
22. 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.
23. IBM Corp; Understanding and using the communication Lowlevel Application Programming Interface (LAPI). In IBM Parallel System Support Programs for AIX Administration Guide, GC233897 -04. 1997. (Available at http://ppdbooks.pok.ibm.com:80/cgibin /bookmgr/bookmgr.cmd/BOOKS/sspad230/9.1).
24. LAM Team, The University of Notre Dame, LAM/MPI Parallel Computing http://www.mpi.nd.edu/lam/
25. B. Lewis and D. J. Berg, Multithreaded Programming with Pthreads, Prentice Hall, 1998.
26. Ewing Lusk, et. al., Argonne National Laboratory, *MPI-2: Extensions to the Message-Passing Interface*
27. MPI Forum (1997), Message-Passing Interface Standard 1.0 and 2.0, http://www.mcs.anl.gov/mpi/index.html
28. B. Nichols, D. Buttlar and J. P. Farrell, Pthreads Programming, O'Reilly, 1996.
29. J. Nieplocha, B. Carpenter, ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems, *Proc. RTSPP IPPS/SDP'99*, 1999.
30. Portable Runtime System (PORTS) consortium, http://www.cs.uoregon.edu/research/paracomp/ports/
31. A Proposal for PORTS Level 1 Communication Routines, http://www.cs.uoregon.edu/research/paracomp/ports
32. RS6000 Group, International Buisness Machines, IBM Parallel Environment for AIX - MPI http://qpsf.edu.au/software/ppe.html
33. 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.
34. J. Shewchuk, Delaunay refinement mesh generation, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1997, Available as CMU Tech Report CMU-CS-97-137.
35. ClusterController AE Scheduling Software, MPI Software Technology, Inc. *http://www.mpi-softtech.com/products/cluster_controller/default.asp*
36. Sun MPI Group, Sun Microsystems, Sun HPC Cluster Tools 3.1 http://www.sun.com/software/hpc/overview.html
37. Watson, D., Computing the n-dimensional Delaunay tessellation with applications to Voronoi polytopes, *The Computer Journal*, Vol. 24, No. 2, pp 167–172, 1981.