

One-Sided Communication Over MPI-1*

Jeffrey Dobbelaere and Nikos Chrisochoides
Computer Science Department
College of William & Mary
Williamsburg, VA 23185
{jdobbel, nikos}@cs.wm.edu

February 1, 2001

Abstract

MPI-1 has been accepted as the standard parallel messaging substrate. However, as parallel applications become more complex and communication patterns more difficult to predict, there is a dire need for one-sided communication that allows asynchronous message passing. In this paper we present a one-sided communication system that provides users with the interface to send asynchronous messages whenever the need arises. On the receiving side, it allows the user to receive messages without knowing the intent or size of those messages. Portability can be inherited by building this system on top of MPI in ANSI C. By providing only the bare minimum in terms of API, the user has to suffer minimal overhead. The bottom line is that with minimal addition of code, a portable, one-sided, asynchronous parallel runtime system can be created that greatly enhances the user's environment.

*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.

1 Introduction

The current state of parallel programming is largely dependent on a binary (send/receive) protocol. MPI has been accepted as the standard in parallel computing systems, and although others exist, none is as widely used nor as readily available as MPI. MPI-1 has proven itself extremely useful in Bulk Synchronous Protocol (BSP) programming when the user has *a priori* knowledge of communication scheduling as well as message sizes. However, as parallel applications become more complex communication patterns become increasingly difficult to predict. In such cases MPI falls short as a messaging substrate. The need for a one-sided message passing protocol has been well-known for many years. Some of these issues were initially addressed by the *P*ortable *R*un-*T*ime *S*ystems (PORTS) consortium [1]. Among others, one of the missions of PORTS was to define a standard application programmer interface (API) for one-sided communication.¹ During development, several different approaches were taken towards the one-sided API. The first is the *thread-to-thread* communication paradigm which is supported by CHANT [13]. The second is the *remote service request* (RSR) communication approach supported by libraries such as NEXUS [9]. The third approach is *hybrid* communication which is a combination of the two prior paradigms. This approach is supported by the TULIP [2] project. Our project is focused on the needs of the applications we develop, such as guaranteed quality 3-dimensional Delaunay triangulation and takes on the RSR technique to supply these needs.

As previously stated, the PORTS consortium set out to define a one-sided API prior to MPI-2 standard. In 1997, a new MPI standard known as MPI-2 [14] was agreed upon which would provide some basic one-sided functionality. Many MPI distributions [17, 12, 10] have implemented the bulk of the one sided communication discussed in the MPI-2 standard, however, for the most part, the computational world still relies on MPI-1 as the standard communication substrate because of its portability and reliability. Even as computational code moves towards MPI-2, there is still the matter of remote method invocations, an essential part of many asynchronous parallel applications. Basically, we do not propose a replacement for the MPI-2 standard, but rather a different approach to one sided communication that is built on MPI-1. In this paper, we offer a solution that satisfies the one-sided, portable, and easy-to-use criteria. We provide a system as portable as MPI-1, that allows the user to communicate completely asynchronously between computational nodes. The system provides three distinct groups in terms of application programmer interface: common environmental calls, remote memory manipulation, and remote method invocation. The first group contains those calls familiar to all parallel programmers that initialize and shutdown the runtime system as well as offer information about the processors themselves. The second group allows the user to read remote data as well as write local data to a remote machine. It also allows the user to allocate a free remote memory. This is extremely useful in the final group of API calls which allows the user to invoke methods on remote computational nodes. By creating specific calls for methods receiving between zero and four arguments, we can optimize the

¹The MPI-2 standard was not yet defined.

message passing performance. An API call for methods needing arbitrary size argument lists is also demonstrated. The system we developed that demonstrates the above listed characteristics is called Data Movement and Control Substrate (DMCS). This system was initially presented in [5] and built on Active Messages [15]. As our understanding of adaptive applications have grown since then, DMCS has undergone significant design and implementations changes.

The rest of this paper is organized as follows: Section 2 gives a brief introduction to an application targeted by DMCS. Section 3 discusses the two parallel execution models presented by DMCS. Section 4 briefly introduces the API of DMCS in order to discuss the implementation of the system in greater detail in Section 5. Section 6 illustrates the performance results found by running various tests on the system. Section 7 discusses two of the applications currently built using DMCS and the results from some experimentation. Finally, Sections 8 and 9 give concluding remarks and briefly discuss future plans for DMCS.

2 Application Description

The development of DMCS can be best understood by examining some of the application contexts in which it is used. Mesh generation is a basic building block for the discretization of continuous partial differential equations (PDEs) and the generation of discrete linear systems of algebraic equations. Delaunay triangulation algorithms have been used very successfully to develop guaranteed-quality adaptive unstructured mesh generation applications on scalar machines. Delaunay-based algorithms generate unstructured meshes by adding new points on demand and modifying the existing triangulation by means of purely local operations. The basic kernel for Delaunay-based algorithms is a four step procedure: The first step, *point creation*, creates a new point 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 [16]. Finally the fourth step, *element creation* builds new triangles by connecting the new point with old points such that the resulting triangulation satisfies certain geometric properties. This kernel often is called the Bowyer-Watson (BW) kernel [3, 19].

The parallel implementation of the BW algorithm, for 3D domains, begins 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, and each processor is responsible for managing multiple regions. The third step in the BW kernel is the source of unpredictable computation and communication, since 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. Binary communica-

tion protocols like MPI-1 increase code complexity in order to handle the uncertainty in placing *receive* calls in the code and frequency of receiving messages. The *receive* calls are required to get messages that may or may not have been sent (using a corresponding *send* primitive) from a set of processors whose members and cardinality varies continuously.

On the other hand, 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. For example, one of the most frequent operations is: *given a point p and an element e , perform a breadth-first search among all adjacent elements to e and identify those that violate the Delaunay property*. The breadth-first search, in average 20% to 30% of the times expands on 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 side, 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.

Other than the third step in the BW kernel, 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. Remeshing is required to handle changes in the topology and geometry for applications like crack propagation [4]. 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. One problem is setbacks in the progress of the algorithm. 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 are: one-sided, non-blocking, and asynchronous data movement primitives like *get/put* and *get_op/put_op* and *remote procedure calls*. The latency of small size (half kilobyte) data movement primitives is very critical for the performance of adaptive applications. Figure 1(left) 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(right) indicates that the the total time spend in message passing is about 15% of the total execution time. Other computation activities are depicted in Figure 1(right) for comparison, for more detailed data see Section 7.

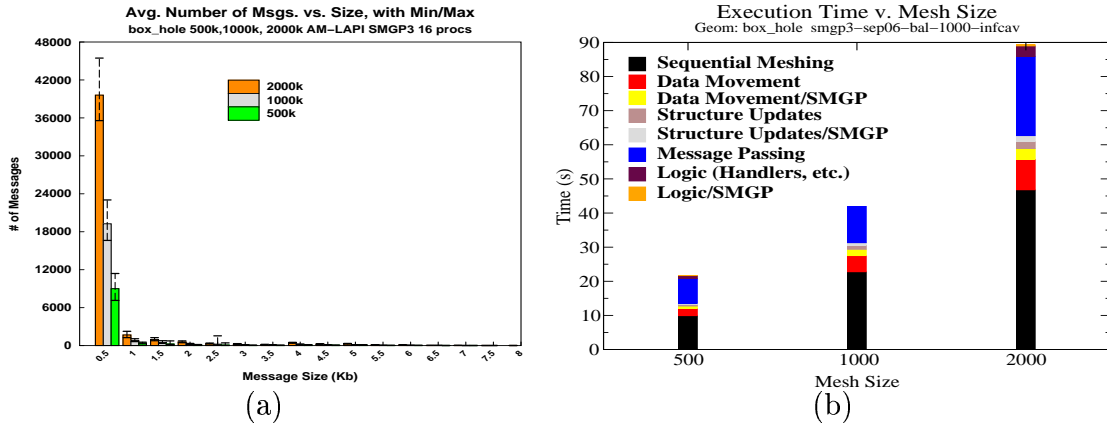


Figure 1: (a) Distribution of messages with respect to message size. (b) Breakdown of the total execution time of an adaptive application, parallel guarantee quality Delaunay triangulation.

3 Parallel Execution Model

Single Threaded: The DMCS system provides two separate execution models pertaining to developers of different types of applications. First, we provide a single-threaded execution model² that may be very familiar to MPI programmers. This offers the absolute thinnest possible layer on top of MPI by disregarding the need for mutexes around critical sections. In this mode, every user-level remote call, whether it be an RSR or some type of remote memory manipulation, is received and executed within a user-level poll routine on the receiving computational node. This polling routine, however, must be called from the only existing communication thread (main thread) in order to adhere to the single-threaded model. This model allows the user to determine when it is best to poll the network for incoming messages without knowing exactly what type of message is coming. Figure 2 (a) depicts the single-threaded execution model of the runtime system. Process 0 sends a message, in this case an RSR, to process 1. Process 1 receives the message during a user-level poll operation and executes the user defined handler.

Multi-Threaded: The second communication paradigm provides a thread-safe environment that allows the user to execute communication from any of the computational threads. The user must be forewarned, however, that this adds overhead to the system in order to ensure the integrity of critical data structures. In the single-threaded model, communications are received via a user-level poll operation, whereas in the thread-safe environment, this poll routine may be called from any running thread on the processor. Again, this is the only time that a user-level remote method is invoked and remote memory is manipulated. This model allows the user to create a “polling thread” whose job is to intermittently check the network for incoming messages. The pros and cons of this approach have been studied in [13, 8]. Using this polling thread, messages are received in a simulated interrupt mode and executed almost immediately upon arrival, as depicted

²This implies a single communication thread and a user defined number of computation threads.

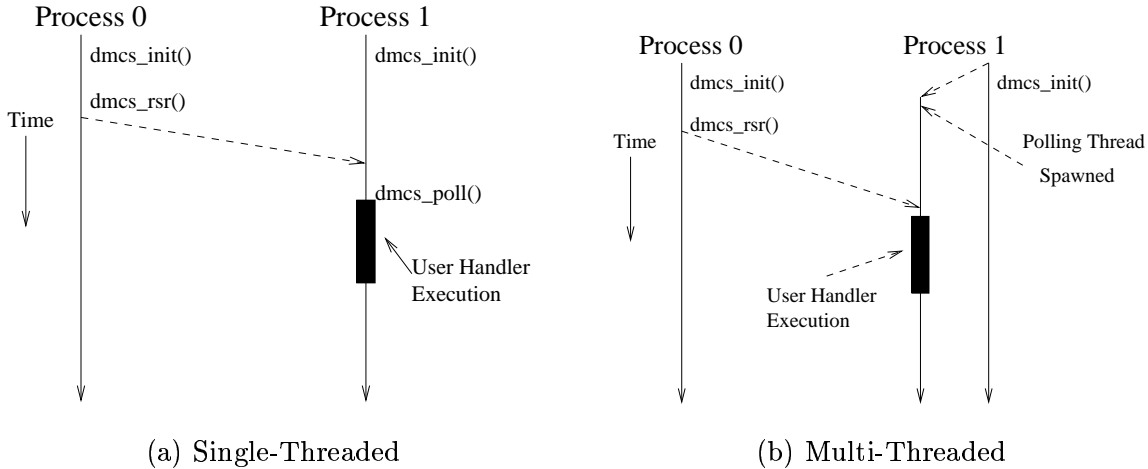


Figure 2: The parallel execution model of the one-sided system uses polls to receive incoming messages

in Figure 2 (b). A thread on process 0 sends a remote service request to process 1. The message is received in the polling thread on process 1, and the user defined handler is executed within that thread. Note that the polling thread is spawned after DMCS has been initialized. The initialization and shutdown calls still must be executed from a single-thread on each processor. In the future, DMCS will be used with intensive graphics code. Mult-threading is extremely important in such code because of the amount of I/O time spent in the application.

4 One-Sided API

As previously stated, there are three major groups of API calls in the DMCS system. The first group contains the *environmental* calls which contains those calls familiar to all parallel programmers. The `dmcs_init()` and `dmcs_shutdown()` are used to initialize and properly cleanup system level data structures and information. Common calls such as `dmcs_my_proc()` and `dmcs_num_procs()` reveal the current processor ID and the total number of currently running processors to the application. Also, several `dmcs_register()` functions are used to register the various user-level functions that will be used in remote service requests.

The second group of DMCS calls contains functions to manipulate remote memory. The two most basic calls are `dmcs_malloc()` and `dmcs_free()`. These two methods are used to allocate and free memory on remote processors. This is especially useful when using the following two calls. For remote reading and writing of data, DMCS provides `dmcs_get()` and `dmcs_put()` in both the synchronous and asynchronous varieties. The user must provide the remote pointers to which data is to be written or those from which data is to be read. The combination of `dmcs_malloc()` and `dmcs_put()` is very

common.

The final group of API functions are *Remote Service Requests* (RSRs). These are more commonly known as Remote Method Invocations (RMIs) or Remote Procedure Calls (RPCs). DMCS provides six different RSR calls, available in synchronous and asynchronous form, that vary in the number of arguments the user level handler takes. For methods that receive zero to four arguments, there is `dmcs_rsr0()` to `dmcs_rsr4()`. For methods needing more than four arguments, DMCS provides `dmcs_put_op()`, which has a slightly different format than the other five RSRs. The user is responsible for packing all the data needed for the execution of the handler into contiguous memory. The entire data buffer is then sent, and the user level handler is responsible for unpacking that data.³ The first five calls are used simply as an optimization. The arguments are packaged in a preallocated structure (described in Subsection 5.1) and sent to the remote node as one message. The `put-ops`, however, require two messages to be sent: one containing system level information about the message, and the second containing the user-level data buffer that is need for the handler execution. Although it is slower to send two messages in terms of network latency, it is faster from the user's perspective than allocating memory and using memory copies to create one larger send buffer. Recall that this system is created with asynchronous applications in mind. The user has no knowledge of when sent messages will be received, and therefore it is more beneficial to have as little overhead on the sending side as possible than minimize network latency. On the receive side, it is also beneficial to receive two separate messages. The first message is received into a preallocated system buffer (discussed further in Section 5.1), and the second is received directly into the user-level address space. This eliminates the need for system level memory allocation and deallocation as well as a memory copy from the system level to the user level buffer. For more information on the specifics of API calls, please refer to the man pages provided with the DMCS distribution.

5 Implementation Details

The implementation of DMCS on top of MPI has to consider four main goals: *performance*, *portability*, *maintainability*, and *correctness*. In order to pass on the greatest possible performance to the user, the overall latency of DMCS has to be as close as possible to that of MPI itself. This is done by avoiding memory allocation, deallocation, and copying where ever possible. Achieving portability in this case is fairly easy. By adhering to the MPI standard, and implementing the entire system in ANSI C, the system is as portable as possible. Maintainability implies that the code is straight forward to use and to implement. The system itself must be understandable so that additional features may be added to the system in the future. It must also be usable in order to make user-level code maintainable. Finally, the system must provide correct semantics. Message

³Although this may seem to burden the user, it is really quite simple. For example, if the handler only needs a vector of integers, no packing or unpacking is necessary, the user only needs typecast that buffer in the handler.

```

void dml_async_rsr0( int tgt, dml_handler_t handler, dml_arg_t nArg1) {

    dml_message_t *buffer = dml_outgoing_table_remove( );

    assert( (tgt < tot_procs) && (tgt >= 0) );

    buffer->type      = RSR1_TYPE;
    buffer->rem_rsr   = dml_lookup_handler( handler );
    buffer->sync_flag = DML_ASYNC;
    buffer->tag       = RSR_TAG;

    dml_send_message( tgt, buffer, sizeof( dml_message_t ), DML_ASYNC );
}

```

Figure 3: A code segment from an asynchronous remote service request taking zero arguments

ordering must be guaranteed to the user, as well as the execution of handlers on remote nodes. Also, deadlock must be avoided in order to guarantee the proper termination of applications. Each of these concerns must be carefully considered, and are therefore expanded on in the following sections.

5.1 Performance Issues

The DMCS system must provide the user with almost negligible overhead in comparison to MPI in order to prove itself useful. As was briefly discussed in Section 4, most messages sent are of a fixed size, and consist of a common data structure (`message_t`). For messages of varied size (put, get, etc.) two messages are sent. The first is a `message_t` and the second is the user level buffer. By always sending messages of the same size, it is possible to use preallocated message pools to minimize both send and receive overhead. These message pools are in fact no more than arrays of the small `message_t` structures that are used when ever a message is sent or received. When a message is sent, a `message_t` structure is removed from the outgoing pool and the appropriate data is entered into the fields of the `message_t`. This message is then sent using an MPI send call. On the receiving side, a `message_t` structure is removed from the incoming pool. The incoming message is then received into that structure. Using these structures eliminates the need to allocate and free memory for each incoming and outgoing message. Figure 3 shows the actual code that performs an RSR with 0 arguments. The first line of the function is the removal of the `message_t` from the outgoing pool. Certain structure attributes are then completed, and the message is sent. Notice the minimal number of lines of code present in such a function. The DMCS system propagates the minimal possible overhead to the user.

5.2 Portability Issues

As previously stated, by implementing the code using the MPI-1 standard and ANSI C, we can be confident that it will port between different architectural systems and different implementations of MPI. Although most of the DMCS implementation was done using x86 Linux boxes with LAM MPI [17], the system has been tested on Solaris and AIX using LAM MPI, MPICH [18], and IBM's MPI for the RS6000. On each of these configurations, DMCS ran wonderfully. We have also tested DMCS on Windows NT using MPIPro [11]. On this configuration, simple DMCS tests ran as intended, though tests performing heavy asynchronous communication caused some difficulty. MPIPro uses a message buffering technique unlike any of the other implementations which causes problems when many asynchronous messages are left pending. This illustrates that even though an MPI implementation may adhere to the MPI standard, the implementation details may vary. We are currently working on a method to circumvent this problem, and buffer the user from any difference in MPI implementations.

5.3 Maintainability

If possible, we would like to take advantage of high performance communication substrates supplied by vendors of specific parallel machines. Throughout this process, we must keep a constant API. Also, we would like to be able to use the threaded abilities of DMCS on different platforms knowing that thread packages differ between platforms. For this reason, we have created DMCS through the multi-layer approach. Currently, there are in fact three layers to DMCS. The first is the Data Movement Layer (DML). This layer provides the interface to MPI. This is the only layer that would need to change in order to support a high performance vendor specific runtime system.⁴ It is also the bulk of the code. The second layer is the Thread Safety Layer (TSL). As its names states, it provides thread safety to the entire DMCS package. Currently there is only a pthreads implementation of this module. The final layer is the DMCS API layer. This is the layer that the application developer actually uses. It is often a one-to-one mapping with the DML, but certain DMCS functionality (remote malloc and free) can be implemented using just the DML. By using the *separation of concerns* approach to DMCS, we are able to plug and play with these modules in order to configure the runtime system for the current parallel machine being used.

5.4 Correctness

Correctness is obviously a very important aspect of code development. We have 3 major concerns when dealing with correctness. The first is message ordering. We need to guarantee that a series of messages sent from one machine to another are received in the order they were sent. Fortunately, MPI provides the means of accomplishing this by simply

⁴In fact a LAPI version of the DML and an AM version of DMCS has been implemented in the past.

adhering to certain semantics. Another concern is remote handler execution. Although many systems use uniform address mapping across processors, this cannot be guaranteed. For the sake of portability, we use the common method of handler registration. The user is required to register all functions that are going to be used for RSRs prior to their use. During this time, DMCS maps each function pointer to a small integer. When an RSR is called, the function pointer is found in a table containing the corresponding small integer. That integer is then sent to the receiving processor. The integer is used as an index into the function pointer table on the remote node, and the user handler is executed.

Another major concern in regards to the correctness of the system is deadlock avoidance. When two processors send synchronous messages to each other there is the possibility of deadlock. In order to combat this, we use another common trick. Instead of using an MPI blocking synchronous call, we use an MPI *non-blocking* synchronous call. We can then use the `MPI_Test` function to determine when that operation completes. If a predetermined amount of time passes and the operation has not completed, DMCS will poll the network to see if there are any pending messages that may be causing deadlock. These messages are received and executed in order to relieve the system of any deadlock possibilities. In order to adhere to the execution model, the message handlers are placed in a queue to be executed the next time the user polls.

6 Performance Analysis

The performance figures depicted in this section were collected using two systems. The Linux numbers were collected from a network running 1GHz Pentium III machines each with 128 megabytes and connected by 100 Mbit fast-ethernet. The Solaris data were collected on a network of Sun Ultra 5 machines with 333 MHz processors each with 256 megabytes of memory connected by a 100 Mbit fast-ethernet. It is apparent from the tables 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 graph shown in Figure 4. What is even more interesting is that the DMCS overhead time is completely independent of message size. The DMCS overhead from Table 2 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.

	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 1: 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.

	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 2: 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 3: 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 4: 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.

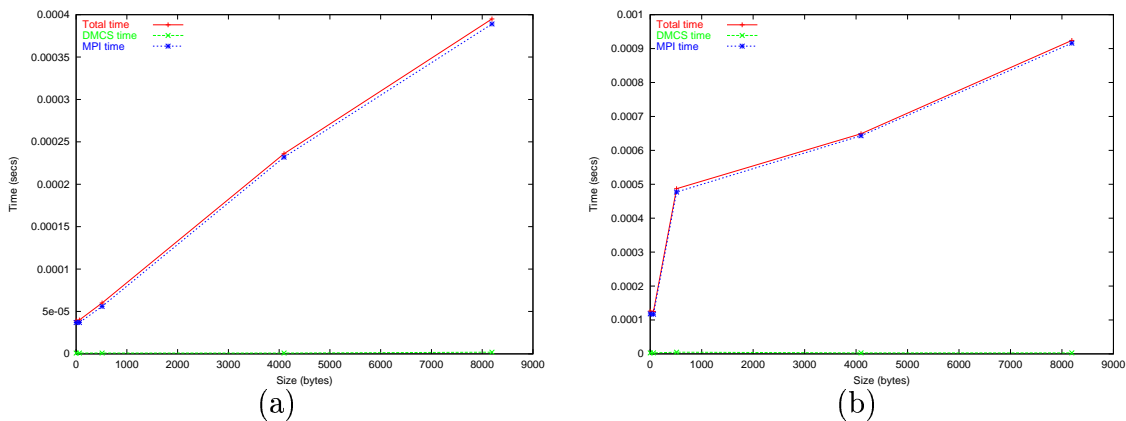


Figure 4: Plots of Put-Op times. Each graph contains DMCS overhead, MPI overhead, and total execution time. The tests in graph (a) were run on Intel PIII 1GHz machines running Linux connected by 100 Mb Fast Ethernet. The tests in graph (b) were run Sun Ultra 5 333Mhz machines running Solaris connected by 100 Mb Fast Ethernet.

7 Applications

In this section we evaluate the DMCS overhead in the context of a complete applications and a netsort kernel. The application is a 3-dimensional Parallel Guaranteed Quality Delaunay Triangulation [7]. The netsort kernel consists of two parallel network sorting routines each using different scenarios: *static netsort* and *mobile netsort*. In the former, data is not moved across the network. However, in the second scheme data is moved around randomly to minimize load imbalance across processors. The following paragraphs describe these applications in some detail.

Guaranteed Quality Delaunay Triangulation (GQDT): The performance of the 3-dimensional Parallel GQDT depends heavily upon an 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, send many messages to other processors containing subdomains which own tetrahedra violated by the newly inserted point. The communication required for computing these distributed cavities is variable and unpredictable so the standard message optimization techniques do not apply.

Table 5 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. This table shows that an inefficient message passing system adds significant overhead even if each cavity only sends a few messages to its neighboring nodes. On average, each distributed cavity sends fourteen

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

P	# Tets	Time	<i>DMCS Overhead</i>			<i>MPI Overhead</i>		
			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 6: 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.

messages.

Table 6 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 [7]. 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 in average varies from 0.5% (one million elements generated on two nodes) to 2.5% (one million elements generated on 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 [6].

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 that push 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

Processors	<i>Linux Cluster</i>			<i>Solaris Cluster</i>		
	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 7: 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	<i>Linux Cluster</i>			<i>Solaris Cluster</i>		
	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

Table 8: 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.

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 integer’s 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 7 and 8 depict the MPI and DMCS overheads which varies from 0.04% to 0.9%. The DMCS and MPI overheads are higher on the Solaris cluster because the nodes are much slower.

8 Conclusion

As parallel programming codes become more complex, the need for a one-sided communication substrate becomes more pronounced. The introduction of the MPI-2 implementations have offered users the ability to use some one-sided functionality. However, portability becomes an issue when dealing with MPI-2 because different distributions have implemented the MPI-2 standard to various degrees. Also, remote method invocations are noticeably absent from this standard. Throughout this paper we have shown how to create a one-sided communication substrate on top of MPI-1 that supports *remote service requests* and adheres to four particular goals. First, *portability* is achieved by using MPI-1 and ANSI C. Second, *performance* is enhanced by using a preallocated messaging

system, and offering the user only the necessary functionality. Also, by making thread-safety a compile time option we eliminate the need for critical section locking if it will not be used. Third, *maintainability* is achieved by using the multi-layer approach to software design. We separate all platform specific code and thread specific code from the API. This allows us to plug in different code modules that work on different platforms with different threading systems. Finally, *correctness* is guaranteed through such common methods as handler registration and deadlock avoidance. We have also shown why such a system is important to asynchronous applications such as Guaranteed Quality Mesh Generation (Section 7). Such applications require data movement at unknown intervals, and therefore rely heavily on an asynchronous one-sided communication paradigm. Our performance section has shown that the implementation of such a system creates less than 2% overhead, while providing greater flexibility to the application developer. The code is currently being used by the Cornell Theory Center and Mississippi State's Engineering Research Center. The code along with documentation will soon be available at <http://www.cs.wm.edu/~jdobbel/dmcs>

9 Future Work

There are many new items currently in development that will be available in the next version of DMCS. One of these includes adding more threading features to the current implementation. The addition of threaded handlers could prove useful to certain application developers. In this case, when a message is received, it will immediately begin executing in its own thread. For optimization purposes, the runtime system will maintain an inactive thread pool from which threads can be removed to execute these threaded handlers. We also intend to implement C++ bindings for the object oriented software designer. This will provide a more comfortable environment for the developer who mainly deals with object oriented code. As stated in Section 5 there is currently difficulty when using DMCS with Windows NT and MPIPro. We plan on devising a scheme to alleviate these difficulties and provide a system that will port between UNIX-like environments and Windows environments. Also we will explore the possibilities of dynamic resource allocation with DMCS⁵ and delve into fault tolerance options.

10 Acknowledgments

Kevin Barker for his insight into the system and his LAPI implementation of DMCS. Demian Nave for the data he provided using the system with his parallel mesher. 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 '90s on the PORTS API and implementation issues.

⁵Of course MPI-1 does not support such functionality, so we will have to explore elsewhere.

References

- [1] PORTS: POrtable RuntTime System Consortium.
<http://www.cs.uoregon.edu/research/paracomp/ports>.
- [2] Pete Beckman and Dennis Gannon. Tulip: Parallel Runtime Support System for pC++. <http://www.extreme.indiana.edu>.
- [3] A. Bowyer. Computing Dirichlet Tessellations. 1981.
- [4] B. Carter, C. S. Chen, L. P. Chew, Nikos Chrisochoides, G. R. Gao, G. Heber, A. R. Ingraffea, R. Krause, C. Myers, D. Nave, K. Pingali, P. Stodghill, S. Vavasis, and P. A. Wawrzynek. Parallel fem simulation of crack propagation – challenges, status, and perspectives. 2000.
- [5] N. Chrisochoides, I. Kodukula, and K. Pingali. Data movement and control substrate for parallel scientific computing, 1997.
- [6] N. Chrisochoides, N. Mansour, and G. Fox. A Comparison of Optimization Heuristics for the Data Mapping Problem. Technical report, Center For Theory and Simulation in Science and Engineering, Cornell University, 1995.
- [7] Nikos Chrisochoides and Démian Nave. Parallel Guaranteed Quality h-refinement and Mesh Generation. *International Journal for Numerical Methods in Engineering*, To be submitted spring 2001.
- [8] Nexus/Globus Community. Nexus Functional Reference.
- [9] Ian Foster, Carl Kesselman, and Steve Tuecke. The NEXUS Approach to Integrating Multithreading and Communication.
- [10] IBM RS6000 Group. IBM Parallel Environment for AIX - MPI Library.
<http://www.qpsf.edu.au/software/ppe.html>.
- [11] MPIPro Group. <http://www.mpi-softtech.com>.
- [12] Sun MPI Group. Sun HPC clustertools 3.1.
<http://www.sun.com/software/hpc/overview.html>.
- [13] Matthew Haines, David Cronk, and Piyush Mehrotra. On the Design of Chant: A Talking Threads Package. Technical report, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1994.
- [14] Ewing Lusk and et. al. *MPI-2: Extensions to the Message-Passing Interface*.
- [15] Alan Mainwaring and David Culler. Active Messages API and Communication Subsystem Organization. Technical report, University of California at Berkeley, 1999.

- [16] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, New York, 1985.
- [17] LAM Team. LAM/MPI Parallel Computing. <http://www.mpi.nd.edu/lam/>.
- [18] MPICH Team. MPICH - A Portable Implementation of MPI. <http://www-unix.mcs.anl.gov/mpi/mpich/>.
- [19] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to voronoi polytopes. *Computer J.*, 24(2):167–172, 1981.