

An Integrated Shared-Memory / Message Passing API for Cluster-Based Multicomputing[†]

Evan Speight, Hazim Abdel-Shafi, and John K. Bennett
Department of Electrical and Computer Engineering
Rice University
6100 Main Street
Houston, TX 77005
{espeight,shafi,jkb}@rice.edu
Fax: (713) 524-5237

Abstract

This paper is concerned with the choice of the appropriate programming model for clusters of symmetric multiprocessor (SMP) machines. Software vendors who wish to take advantage of cluster-based multiprocessing are currently faced with three equally unpleasant choices. Whether they currently support shared memory or message passing, they can either choose to ignore the memory model that they do not currently support, they can support both memory models, or they can rewrite for shared memory on top of a DSM runtime system. Software vendors entering the marketplace are similarly faced with the difficult choice between a well-supported existing message passing standard (MPI) and an emerging shared memory standard (OpenMP) that offers convenience and the promise of performance. We contend that these distinctions are unnecessary. Instead of forcing a choice between shared and distributed memory models, we propose a single unified programming model. This model supports access to memory anywhere in the cluster using either shared memory or message passing interaction. Further, shared memory and message passing accesses can be interleaved at fine granularity, while preserving the communication and synchronization semantics of each model. This paper describes modifications to the Brazos DSM system that support this unified model. To demonstrate the feasibility of our approach, we have implemented significant subsets of MPI and OpenMP within the Brazos Common API. To demonstrate the potential performance benefit of the Brazos Common API, we modified two well-known shared memory applications to incorporate the proposed integrated programming style. For the applications studied, we observed performance improvements of 138 and 36 percent over the DSM-only version. We also report several qualitative advantages of the Brazos Common API.

Keywords: distributed shared memory, parallel programming, cluster-based computing

1 Introduction

Distributed memory machines use message passing to communicate data between processors at programmer-determined points in the program's execution. Message passing offers the potential of high performance, but the need to explicitly send data when needed requires considerable care by the programmer. However, it is quite easy and inexpensive to scale the number of processor in a distributed memory system by simply adding another workstation to the network.

Shared memory multiprocessors present a common memory interface to all processors. As a result, a shared-memory parallel program is typically easier to write than a message-passing program that performs the same task. In the absence of heavy contention, shared memory machines also offer the potential for high performance. However, contention for shared resources typically limits the performance and scalability of these machines. Hardware solutions to this contention problem also increase the cost of shared memory machines significantly, particularly for machines with more than a few processors.

By providing an abstraction of globally shared memory on top of the physically distributed memories present on networked workstations, it is possible to combine the programming advantages of shared memory and the cost advantages of distributed memory. Distributed shared memory (DSM) runtime systems transparently intercept user accesses to remote memory and translate these accesses into messages appropriate to the underlying communication media. The programmer is thus given the illusion of a large global address space encompassing all available memory, eliminating the task of explicitly moving data between processes located on separate machines. Both hardware DSM systems (e.g., Alewife [13], FLASH [14]) and software DSM systems (e.g., Munin [4], TreadMarks [11], Brazos [18]) have been implemented.

The ability of current microprocessors to directly support symmetric multiprocessing for two or four processors has created a "sweet spot" of low-priced SMP workstations that are rapidly becoming the commodity PC's of the

[†]This research was supported in part by grants from Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Tandem Corporation, Packet Engines, Inc., and by the Texas Advanced Technology Program under Grant No. 003604-022.

enterprise environment. The low cost of these machines has led to their use in clustered compute farms. These clusters further blur the distinction between shared and distributed memory. A cluster of SMP machines presents the programmer with two distinct memory interfaces: shared memory between the processors local to each machine, and distributed memory between processors that are not co-located. The programmer is thus faced with a dilemma. There are clear performance advantages to using the native SMP load/store interface for communication between co-located processors, but accesses to remote memory requires an entirely different communication model. Utilizing two different communication models in the same program is problematic, particularly in the case where the number of threads and processors per machine is not known at compile time. Distributed shared memory (DSM) systems attempt to alleviate this situation by providing a single shared view of all memory in the cluster, regardless of where it is located. For example, programs written using the native Brazos [18] API allow the same executable to be run without recompilation on a uniprocessor, an SMP multiprocessor, or a cluster of such multiprocessors. The Brazos runtime system uses the appropriate communication mechanism transparently to the programmer for each memory access, depending upon the location of the memory being accessed. There are two reasons, however, why DSM alone is an insufficient solution to the problems posed by clustered SMPs:

- The data access behavior of an application may overwhelm the DSM system with coherence and/or synchronization related message traffic. Programs that exhibit frequent changes in sharing behavior, large amounts of data movement, or very fine synchronization will typically exhibit poor performance on DSM platforms.
- The absence until recently of programming standards in the shared memory community, and the presence in the message passing community of such standards (most notably MPI [7]), has led to the existence of a significant body of “legacy” message passing code. Many enterprises have a significant investment in highly optimized message passing programs, particularly for numeric applications. It is unreasonable to expect that these organizations will be willing (or even able) to retool their applications for a shared memory environment in the near term.

The emergence of OpenMP [1] as a standard for programming shared memory machines therefore confronts current software vendors who wish to take advantage of cluster-based multiprocessing with three choices. They can ignore the memory model that they do not currently support, they can support two standards, or they can rewrite to the new shared memory standard on top of a DSM runtime system. Software vendors entering the marketplace are similarly faced with the difficult choice between a well-supported existing message passing standard and an emerging shared memory standard that offers convenience and the promise of performance.

We contend that these distinctions are unnecessary. Instead of forcing a choice between shared and distributed memory models, we propose a single integrated programming model within the Brazos Common API. This model supports access to memory anywhere in the cluster using either shared memory or message passing interaction. Further, shared memory and message passing accesses can be interleaved at fine granularity, while preserving the communication and synchronization semantics of each model.

This paper describes modifications to the Brazos DSM system that support this unified model. To demonstrate the feasibility of the Brazos Common API, we have implemented significant subsets of MPI and OpenMP within Brazos. To demonstrate the potential performance benefit of the Brazos Common API, we modified two well-known shared memory applications to incorporate the proposed mixed programming style. For the applications studied, we observed performance improvements of 138 and 36 percent over the DSM-only version. The Brazos Common API also offers the following qualitative advantages:

1. Existing MPI and OpenMP programs run without modification by the programmer (during compilation minor transformations to the source code are made by the Brazos preprocessor).
2. Using a single API, programmers are able to use the most appropriate communication model in each instance within the program, without concern for whether or not coherence and synchronization constraints are being correctly maintained when switching from one model to the other. The Brazos runtime system transparently chooses the best mechanism based upon thread location.
3. Multithreading issues and optimizations are handled transparently by the underlying DSM substrate.
4. Programs are easier to develop. Global shared memory can be assumed except in those instances where performance constraints require the use of message passing.

5. Finally, existing shared memory programs that can benefit from bulk data transfer will exhibit improved performance.

2 Design and Implementation of the Brazos Common API

One of our goals in the design of the Brazos Common API was to ensure that it was possible to finely interleave shared memory and message passing programming models. It would have been much simpler to have required programs to be divided into phases during which either shared memory or message passing was used (but not both), and then to insert global synchronization (in the form of a barrier) between each phase change. In addition to the synchronization performance penalty, we believe that this coarse-grained interleaving is unnecessarily constraining. There are many instances when it is useful to send bulk data while maintaining full coherence. This section discusses the issues involved in the design of the Brazos Common API.

2.1 An Overview of Brazos

Brazos [17] uses page-based memory protection and low-level message passing primitives to implement the abstraction of shared memory on a network of clustered SMP computers under Windows NT. Brazos currently runs on x86 multiprocessors connected by FastEthernet, Gigabit Ethernet, or ServerNet [10], and supports either a Winsock or VIA [2] messaging layer. Users write Brazos programs using familiar shared-memory programming semantics. Brazos provides thread, synchronization, and data sharing facilities like those found in other shared memory parallel programming systems. Any shared data in the system can be transparently accessed by any thread without regard to where in the system the most current value for that data resides. The Brazos runtime system is responsible for intercepting accesses to stale data and bringing shared pages up-to-date before program execution is allowed to continue.

Brazos consists of three parts: a user-level library of parallel programming primitives, a service that allows the remote execution of DSM processes, and a Windows-based graphical user interface. The Brazos user-level library provides the interface between user code and DSM code. The most important part of the Brazos library is the interface for capturing accesses to invalid data and initiating messages with other processes in the system. The Brazos API also includes synchronization primitives in the form of locks and barriers, which can be used to provide synchronization both between threads in different processes as well as between threads in the same process. Routines for error reporting, statistics gathering, and data output are also provided in the Brazos API.

2.2 OpenMP on Brazos

There are three components required to support the OpenMP API in Brazos: (1) an implementation of the OpenMP library functions using the Brazos API, (2) a preprocessor that accepts Fortran programs containing OpenMP directives and library calls and transforms them into Fortran programs that will compile and run on the Brazos DSM system, and (3) Fortran interface declarations of Brazos API functions (since Brazos is written in C). Brazos currently supports OpenMP using Digital Visual Fortran 90.

The code fragment below shows a simple example of the transformations performed by the Brazos preprocessor. This code transformation tool scans the program file and performs a few important changes to the code to comply with the Brazos Fortran support. First, the tool scans the source-code to identify all shared data structures. Every shared data structure has to be allocated using a Fortran-specific memory allocation function in the Brazos API. Second, the main program body is transformed to only include a call to the Brazos initialization routine. Third, a subroutine called `PARALLEL_BEGIN` is inserted as the starting point of all user threads at the beginning of the parallel section. `PARALLEL_BEGIN` is organized as described below.

Shared data declarations are copied from the `PROGRAM` section of the original code, and altered to use Brazos memory allocation routines. Next, additional variables needed for parallel execution (e.g., thread id and the total number of threads) are declared. The first code fragment that appears in `PARALLEL_BEGIN` assigns the appropriate values to the thread id and the total number of threads using Brazos API or OpenMP library calls. Then, the initialization code from the original program and all memory allocation calls are enclosed in a serial section executed by Thread 0. A barrier follows this initialization section where all other threads wait for the initialization to complete. The parallel section starts immediately after the first barrier and the work in loops is partitioned according to the current value of the `OMP_SCHEDULE` environment variable.

<pre> PROGRAM OMP_EXAMPLE INTEGER I1, I2, I3 REAL RARRAY1(DIM1,DIM2), RARRAY2(DIM1,DIM2) COMMON /MAIN/ I1, I2, I3 COMMON /SHARED/ RARRAY1, RARRAY2 -----Sequential initialization code----- !\$OMP PARALLEL SHARED(RARRAY1,RARRAY2) DEFAULT(PRIVATE) -----Parallel code block----- !\$OMP END PARALLEL -----Serial section----- END PROGRAM OMP_EXAMPLE </pre>	<pre> PROGRAM BRAZOS_OMP_EXAMPLE C include the Brazos interface declarations INCLUDE 'brazos_fortran.f' CALL BRAZOS_INIT() END PROGRAM OMP_EXAMPLE C parallel_begin() is called by brazos_init SUBROUTINE PARALLEL_BEGIN() INCLUDE 'brazos_fortran.f' INTEGER I1, I2, I3 INTEGER, AUTOMATIC :: MYID, NPROCS C make loop bound variables thread-local INTEGER, AUTOMATIC :: DOBEGIN1, DOEND1, DOBEGIN2, DOEND2, ... REAL RARRAY1(DIM1,DIM2), RARRAY2(DIM1,DIM2) POINTER (RARRAY1PTR, RARRAY1) POINTER (RARRAY2PTR, RARRAY2) COMMON /MAIN/ I1, I2, I3 COMMON /SHARED/ RARRAY1, RARRAY2 MYID = OMP_GET_THREAD_NUM() NPROCS = OMP_GET_NUM_THREADS() C allocate globally shared memory IF (MYID .EQ. 0) THEN FORTRAN_G_MALLOC(RARRAY1PTR, ...) FORTRAN_G_MALLOC(RARRAY2PTR, ...) -----Sequential initialization code----- ENDIF CALL DSM_BARRIER() -----Parallel code block----- CALL DSM_BARRIER() IF (MYID .EQ. 0) THEN -----Serial section----- ENDIF CALL DSM_BARRIER() END SUBROUTINE </pre>
---	--

Figure 1: OpenMP Transformations Performed by the Brazos Preprocessor Tool

The default OpenMP scheduling policy is `STATIC`, which is the only policy currently supported in our implementation. Additional loop control variables and loop-bound calculations are inserted as necessary to create the appropriate schedule. Barriers are inserted according to OpenMP semantics or as they exist in the original program source-code. Since creating threads can be expensive in a software DSM system, the Brazos implementation of OpenMP does not terminate user threads at the end of OpenMP parallel sections. Instead, a single thread (usually Thread 0) executes serial sections between parallel sections. Finally, an additional barrier is placed at the end of `PARALLEL_BEGIN` that is required so that all nodes participating in the program can continue to service DSM-related messages. The main transformation performed on other functions and subroutines is to declare all non-common data structures as `AUTOMATIC`; this is needed to support multithreading.

2.3 MPI on Brazos

The Brazos Common API implements a large subset of the Message Passing Interface (MPI) API described in [7]. Few changes are required to existing MPI-compliant code in order to execute on the Brazos system. Here we

mention a few of the implementation issues associated with supporting MPI on top of a software distributed shared memory system such as Brazos.

- **MPI_Send** Brazos transparently translates an **MPI_Send** that specifies a receiving thread that resides on the same machine into a call to **memcopy**, taking advantage of the shared memory hardware available to co-located threads. Thus programmers may use **MPI_Send** and **MPI_Recv** between any threads in the system without regard to thread. A blocking remote **MPI_Send** returns when the corresponding network send has been acknowledged from the remote machine specified in **MPI_Send**. The remote process will copy the send data directly into the buffer provided by the **MPI_Recv** in the remote process if the buffer has been posted prior to the send. If there has not yet been an **MPI_Recv** in the destination process, the data is copied into a thread-specific temporary buffer until the corresponding **MPI_Recv** has occurred. This allows the acknowledgement of the **MPI_Send** to be returned as soon as this copy completes, reducing the stall time of the process initiating the **MPI_Send**.
- **MPI Group Communication** Because Brazos already utilizes multicast communication when available, implementing the corresponding MPI functions (**MPI_BCAST**, **MPI_ALLTOALL**) is trivial. The underlying support, flow control, and reliability hooks already present in Brazos are used to implement the required MPI group communication calls.
- **MPI_Recv** In Brazos, application-level threads do not directly call network receive primitives. Rather, a dedicated DSM system thread receives all incoming messages and routes them to the corresponding application thread. Thus, a thread that calls **MPI_Recv** simply registers the destination buffer with the system thread responsible for receiving incoming messages and waits on a Windows NT kernel event. When the appropriate message is received, the system thread will perform the copy into the buffer supplied during the **MPI_Recv** call and wake up the application thread. If the **MPI_Recv** is posted after the corresponding **MPI_Send** has completed, the application thread simply copies the received data out of the system temporary buffer into the final destination.

2.4 Integrating Message Passing and Shared Memory in the Brazos Common API

Integrating message passing and shared memory at a fine granularity on a cluster of multiprocessors poses several challenges. The first issue that must be addressed is support for multithreading. MPI assumes that the unit of computation is a process, and that there is an implicit assumption of a one process – one processor model. In Windows NT, threads are the unit of computation. For this reason, the Brazos implementation of MPI supports messages between threads. In addition, fine grain interaction of message passing and shared memory in Brazos leads to the following considerations:

Buffer Management:

1. All message buffers that reside in shared memory space must be allocated using Brazos primitives.
2. Sends to local threads do not cause data movement, but do have synchronization implications.
3. The address of the send and receive buffer can be the same, in which case DSM update may be used to communicate data.

Synchronization:

1. MPI barriers provide global synchronization, but do not guarantee coherence.
2. **MPI_Send**, in addition to transferring data, denotes the release of an implied lock between the message sender and the message receiver. **MPI_Recv** denotes an acquire of the same lock. The Brazos Common API makes these locks explicit, although this fact is transparent to the programmer.

We now briefly describe the implementation of the coherent message passing supported by the Brazos Common API. Consider the situation depicted in Figure 2, in which a thread on SMP 1 sends a large (several pages in length) message to a thread on SMP 2. For this example, assume that most of the pages being sent are valid on SMP 1, but that SMP 3 has the most up-to-date copy of some portions of the pages contained in the message. The following actions will occur:

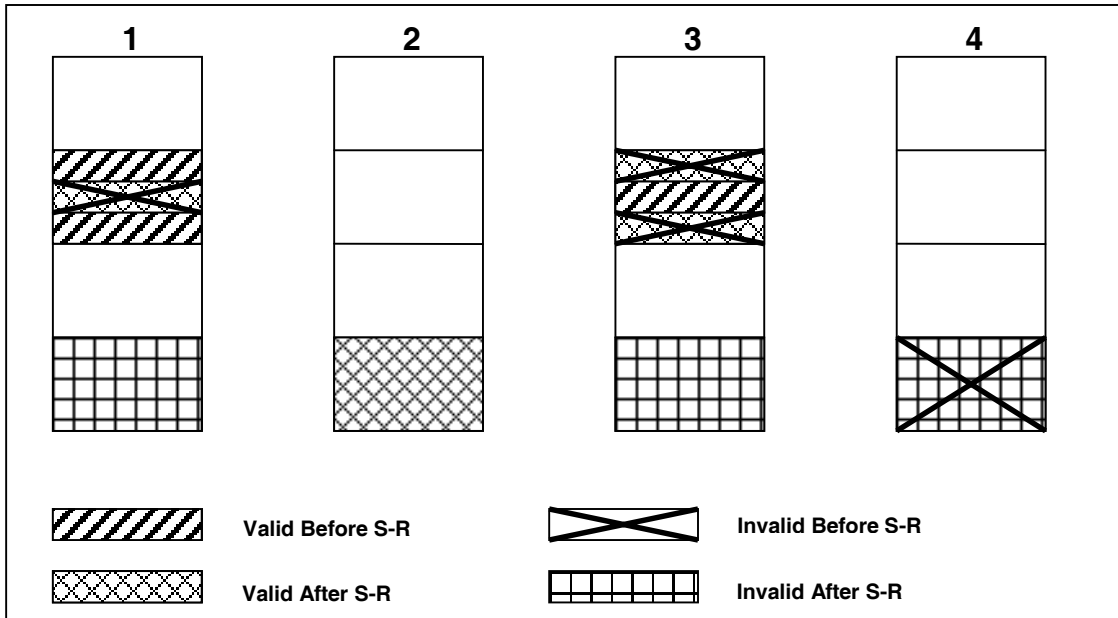


Figure 2: Coherent Message Passing Example

1. When SMP 2 performs the receive, it will multicast an *Acquire-Read-Invalidate* to the sender (SMP 1) and the machines that may have portions of the required data (in this case, SMP 3). All prior receives must have been performed before this receive performs.
2. SMP 3 provides either *diffs* [4] or whole pages, as most appropriate. Where the data is coalesced depends upon whether or not the send occurs before the receive. If the send occurs first, the data is coalesced in a temporary buffer created on SMP 2 when the send occurs. If the receive occurs first, the data will be coalesced in the user space of SMP 1 prior to the actual send.
3. SMP 1 provides either *diffs* or pages when the send is performed.
4. The destination buffer will be invalid on all machines except the receiver (SMP 2 in this case) when the receive completes.

3 Example Programs

To demonstrate the feasibility of our approach, we report the results of a simple experiment using two well-known numeric applications. The data presented was obtained on a cluster of 4 four-processor Compaq 8000 workstations running Windows NT 4.0 and connected by Gigabit EtherNet. The first program is a locally written implementation of Jacobi, a nearest neighbor algorithm used to solve systems of partial differential equations. The second is an implementation of the Fast Fourier Transform taken from the SPLASH-2 [19] benchmark suite. Three implementations of each application were examined: (1) a **DSM_ONLY** version in which all updates used the underlying DSM-provided shared memory, (2) an **IMPLICIT_DSM_MPI** version in which Brazos transparently optimized local sends and receives into shared memory accesses, and (3) an **EXPLICIT_DSM_MPI** version in which the program explicitly checks the location of the destination. This allows unnecessary buffer copies to be avoided, as well as message combining from multiple threads on the same machine.

```

...
<Threads performs Jacobi calculations on matrix using distributed shared memory>
BARRIER
if (this is a display iteration) {
#ifdef DSM_ONLY // Display thread (Global Id #0) uses DSM for all data accesses.
    if (MyGlobalThreadId == 0)
        < Read and Display All Array Elements>
    BARRIER
#else if defined IMPLICIT_DSM_MPI // All threads send their data to display thread using MPI.
    if (MyGlobalThreadId != 0)
        MPI_Send(MyArrayPortion to GLOBAL thread 0)
    else {
        for(j = 1; j < GlobalNumThreads; j++)
            MPI_Recv(PortionOfArray from thread j)
        <Display All Array Elements>
    }
    MPI_Barrier()
#else //EXPLICIT_DSM_MPI// Local thread #0's marshall local data and forward to display thread.
    if ((MyLocalThreadId == 0) && (MyGlobalThreadId != 0)) {
        MPI_Send(All local threads' portions to GLOBAL thread 0)
    }
    else if (MyGlobalThreadId == 0) {
        for(j = 1; j < NUM_PROCESSES; j++) {
            FirstThread = GetFirstThreadInProcess(j)
            MPI_Recv(All threads' portions from FirstThread (LOCAL # 0's))
        }
        <Display All Array Elements>
    }
    MPI_Barrier()
#endif
}

```

Global shared memory used

Local shared memory implicitly used by Brazos MPI implementation

Local shared memory explicitly used by programmer

Figure 3: Jacobi DSM_ONLY, IMPLICIT_DSM_MPI, and EXPLICIT_DSM_MPI Code

To further motivate this example, each thread of the application is required to periodically update Thread 0 with the current contents of the thread's portion of shared memory. This behavior models the situation in which a graphics device might be used to display the results of the computation while it was in progress. In each iteration, threads perform nearest-neighbor averages on all elements assigned to them in the global array using shared memory semantics. Threads are statically assigned a group of contiguous rows at the beginning of the program. At regular intervals, Thread 0 displays the entire array. Figure 3 shows how this is accomplished in each of the three versions of Jacobi. In the **DSM_ONLY** case, Thread 0 simply reads the entire array, with the underlying DSM system providing the data motion necessary to ensure that the values read by Thread 0 are consistent with other threads in the system. This method is very easy to implement from a programming perspective, and yielded a total execution time of 271 seconds for this example. Because the underlying DSM communication mechanisms work on a page-based granularity, this method results in many short messages consisting of a single page of data. By utilizing the bulk-transfer mechanisms provided by the Brazos Common API, we can significantly reduce the number of messages and the overall execution time.

For the **IMPLICIT_DSM_MPI** case, all threads other than Thread 0 explicitly send their portion of the array to Thread 0. This method is slightly more complicated programmatically, but yields better performance because data is transferred in much larger portions, resulting in fewer high-overhead messages, and better network utilization. Additional performance benefit is obtained by Brazos because the MPI library is actually running on a DSM system that supports SMP machines directly. When a thread posts an **MPI_Send** message to another thread, the receiving thread's location is first checked by the Brazos runtime system. If the receiving thread is local, the data is not sent on the network, but is simply copied into a temporary buffer from which the receiving thread obtains the data when the **MPI_Recv** is posted. If the receive is posted before the send, the data may be directly copied into the receiving buffer with no intermediate temporary copy. This provides very fast "pseudo-messaging" between threads on the

same machine, without requiring the programmer to distinguish between local and remote threads. The execution time for this version is 192 seconds, just 71% that of the **DSM_ONLY** case.

Finally, the **EXPLICIT_DSM_MPI** version requires that the programmer be aware of where Thread 0 is in relation to all other computation threads, and explicitly choose the correct mechanism to communicate. In particular, threads in the same process (e.g., on the same machine) as Thread 0 do nothing, since Thread 0 can simply read their values directly from the shared matrix. This avoids the copy between local threads required for the **IMPLICIT_DSM_MPI** case. Threads in different processes (e.g., on a machine different from Thread 0) must send their portions of the array to Thread 0. A further optimization implemented in this version takes advantage of the fact that only one message need be sent to Thread 0 from each other *process*, since all threads in the remote process can share their portions of the array through local, hardware shared memory. Thus, a single message can be used to send portions for all threads in each remote process to Thread 0. This method requires the most work on the part of the programmer, as well as a “thread aware” style of programming. However, the execution time using this method is 114 seconds, 42% of the **DSM_ONLY** case and 59% of the **IMPLICIT_DSM_MPI** case. These results are shown in Figure 4.

Three versions of FFT were created in a similar fashion. The original shared memory version of FFT contains a single section of code where all-to-all communication is required. This occurs after threads have computed a one-dimensional FFT on their portion of the shared real-complex array and must transpose their data elements with other threads. In the **DSM_ONLY** version, this simply corresponds to each thread reading the data written by the thread with whom the data must be transposed. As in the case of Jacobi, this communication of large sections of data using DSM coherence mechanisms results in a large number of small messages being sent between threads. The running time for the **DSM_ONLY** version was 23 seconds.

The **IMPLICIT_DSM_MPI** version of FFT implements the transpose phase as a series of **MPI_Send** and **MPI_Recv** calls. Because the transpose operation represents all of the sharing in the program (except for the final correctness check and gathering operation), the use of the bulk-transfer mechanisms reduces the application execution time from 23 seconds to 17 seconds. The **EXPLICIT_DSM_MPI** version of FFT replaces all **MPI_Send** and **MPI_Recv** operations between threads in the same process with shared memory reads and writes.

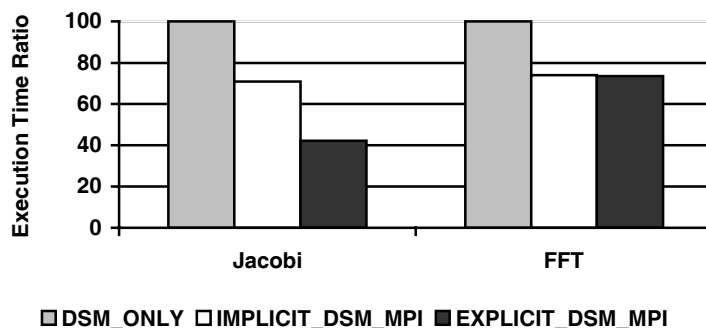


Figure 4: Performance of Jacobi and FFT

Because the Brazos MPI implementation already provides fast sends and receives between co-located threads, the **EXPLICIT_DSM_MPI** version of FFT does not achieve appreciably better performance than the **IMPLICIT_DSM_MPI** version of the program.

4 Related Work

Several systems have integrated shared memory and message passing. The System Link and Interrupt Controller on the Sequent Balance [3] supported a message passing bus separate from the shared memory bus. The BBN Butterfly [15] provided direct hardware support for block transfer. The Platinum system [6] implemented DSM using this mechanism. The Cray T3D also supports block transfer, but does not maintain cache coherence for data sent in this manner. Frank and Vernon [8] proposed a set of message passing primitives for DSM systems that allowed a “possibly stale” page to be sent between processes using their message passing primitives. Klaiber and Levy [12]

examined the inherent costs associated with shared memory and message passing and argued that future high-level languages will mask the programming style differences between these two models.

In contrast to the software-only implementation of the Brazos Common API, the Flash [14] and Alewife [13] systems both integrate message passing and cache coherent shared memory using hardware support. In Flash, a programmable processor on the network interface performs the necessary integration. In Alewife, a shared memory hardware layer is located between processors and the message passing network. LimitLESS directories [5], Alewife's shared memory coherence protocol, are implemented using the message layer to send and receive coherence packets. In addition to message passing, Flash supports cache-coherent block transfer [9]. In an evaluation of the performance of block transfer in Flash, Woo et al. reported limited benefit for the applications studied [20]. The Wisconsin Typhoon [16] also integrates shared memory and message passing by exposing both models in the Tempest interface. Like Flash, Typhoon employs a processor located at the network interface to support the message interface.

5 Conclusions

This paper has described the Brazos Common API, a programming system that supports the fine-grained integration of shared memory and message passing on top of an efficient DSM substrate. We have argued that this integrated programming style is the appropriate programming model for clusters of symmetric multiprocessor (SMP) machines. We have chosen to base the Brazos Common API on the integration of two standards: MPI and OpenMP. We have described the modifications to the Brazos DSM system that support this unified model. To demonstrate the benefits of the Brazos Common API, we have implemented large subsets of MPI and OpenMP within Brazos. We have demonstrated the significant potential performance benefit of our approach for two scientific applications. We intend to continue our work of fine-grain integration of MPI and OpenMP. Our goal is to be fully compliant with both standards, thus giving the programmer complete freedom in the choice of programming style.

Brazos is available free of charge for non-commercial use from our web site at <http://www.brazos.rice.edu/>

References

1. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, 1997.
2. *Virtual Interface Architecture Specification 1.0*, 1997.
3. B. Beck, B. Kasten and S. Thakkar. VLSI Assist for a Multiprocessor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*. p. 10-20, 1987.
4. J.K. Bennett, J.B. Carter and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type - Specific Memory Coherence. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*. p. 168-176, 1990.
5. D. Chaiken, J. Kubiatowicz and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. p. 224-234, 1991.
6. A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on System a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*. p. 32-44, 1989.
7. M.P.I. Forum, *MPI: A Message-Passing Interface Standard, Version 1.0*, 1994.
8. M.I. Frank and M.K. Vernon. A Hybrid Shared Memory/Message Passing Parallel Machine. In *Proceedings of the International Conference on Parallel Processing*. p. 232-237, 1993.
9. J. Heinlein, K. Gharachorloo, R.P. Bosch, M. Rosenblum and A. Gupta. Coherent Block Data Transfer in the FLASH Multiprocessor. In *Proceedings of the International Parallel Processing Symposium*. 1997.
10. R.W. Horst, *TNet: A Reliable System Area Network*, 1995, Tandem Computers.
11. P. Keleher, *Lazy Release Consistency for Distributed Shared Memory*. Ph.D. Thesis, Rice University, 1995.
12. A.C. Klaiber and H.M. Levy. A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Languages. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. p. 94-106, 1994.
13. D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz and B.H. Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Symposium on the Principles and Practice of Parallel Programming*. p. 54-63, 1993.
14. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. p. 302-313, 1994.
15. BBN Laboratories, *Butterfly Parallel Processor Overview*, 1986, BBN Laboratories, Cambridge, Mass.
16. S.K. Reinhardt, J.R. Larus and D.A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. p. 325-337, 1994.
17. E. Speight, *Efficient Runtime Support for Cluster-Based Distributed Shared Memory Multiprocessors*. Ph.D. Thesis, Rice University, 1997.
18. E. Speight and J.K. Bennett. Brazos: A Third Generation DSM System. In *The USENIX Windows NT Workshop Proceedings*. p. 95-106, 1997.
19. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. p. 24-36, 1995.
20. S.C. Woo, J.P. Singh, J.L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. p. 219-229, 1994.