Where's the Overlap?

An Analysis of Popular MPI Implementations

J.B. White III and S.W. Bova

Abstract— **The MPI 1.1 definition includes routines for nonblocking point-to-point communication that are intended to support the overlap of communication with computation. We describe two experiments that test the ability of MPI implementations to actually perform this overlap. One experiment tests synchronization overlap, and the other tests data-transfer overlap. We give results for vendor-supplied MPI implementations on the CRAY T3E, IBM SP, and SGI Origin2000 at the CEWES MSRC, along with results for MPICH on the T3E. All the implementations show full support for synchronization overlap. Conversely, none of them support data-transfer overlap at the level needed for significant performance improvement in our experiment. We suggest that programming for overlap may not be worthwhile for a broad class of parallel applications using many current MPI implementations.**

Keywords— **Message passing, MPI, nonblocking communication, overlap, parallel performance, CRAY T3E, IBM SP, SGI Origin, MPICH.**

I. Introduction

OVERLAPPING interprocessor communication with useful computation is a well-known strategy for improving the performance of parallel applications, and the Message-Passing Interface (MPI) is a popular programming environment for portable parallel applications [1]. The MPI 1.1 definition includes routines for nonblocking point-to-point communication that are intended to support the overlap of communication with computation [2]. Separate routines for starting communication and completing communication allow the passing of messages to proceed concurrently with computation.

Though such overlap of communication and computation motivated the design of nonblocking communication in MPI, the MPI 1.1 standard does not require that an MPI implementation actually perform communication and computation concurrently [2]. We describe experiments that illustrate the degree to which MPI implementations support overlap and the degree to which programming for overlap can improve actual performance. We present results for popular MPI implementations on parallel systems available at the CEWES MSRC[1]: the Cray Message Passing Toolkit (MPT) and MPICH on the CRAY T3E, the IBM Parallel Environment (PE) on the IBM SP, and MPT for IRIX on the SGI Origin2000.

The two experiments we describe are intended to represent two broad categories of parallel applications: asynchronous and synchronous. *Asynchronous* applications are those where different processes perform communication at significantly different times or rates. Examples of asynchronous applications include boss-worker, client-server, metacomputing, and other MPMD[2] applications. Conversely, *synchronous* applications are those where all the processes perform communication at approximately the same time and rate. SPMD[3] applications often qualify as synchronous.

Just as parallel applications can fall into two categories, MPI implementations can support two different levels of overlap: synchronization overlap and data-transfer overlap. Point-to-point message passing is explicitly two sided; one side sends a message, and the other side receives it. If an MPI implementation can send a message without requiring the two sides to synchronize, it supports *overlap of synchronization with computation*. This level of overlap does not necessarily imply that the implementation performs useful computation while data are actually in transit between processes. If an implementation performs this physical transfer of data concurrently with computation, it supports *overlap of data transfer with computation*.

For each of these two levels of overlap, we describe an experiment used to evaluate the MPI implementations at the CEWES MSRC. We analyze the results for each experiment, describe each system and MPI implementation, and discuss the implications for asynchronous and synchronous applications. We conclude with a summary of the results and comments on the effectiveness of programming explicitly for overlap.

II. Overlap of Computation with Synchronization

MPI implementations that do not support the overlap of computation with synchronization must require the two processes involved in a message to synchronize. Figure 1 presents a schematic of an experiment to test support for synchronization overlap in MPI implementations. The experiment measures the completion time for a single message.

Immediately after a mutual `MPI_Barrier`, the sending process issues an "immediate" send, `MPI_Isend`. As the name implies, the immediate send returns immediately, allowing execution to continue. This process "computes" for 4 seconds before issuing an `MPI_Wait`, which blocks execution until the message is actually sent. The receiving

Trey White is with Oak Ridge National Laboratory, Oak Ridge, Tennessee. E-mail: trey@ccs.ornl.gov .

Steve Bova is with Mississippi State University, Mississippi State, Mississippi. E-mail: swb@erc.msstate.edu .

[1]Department of Defense (DoD) High Performance Computing Modernization Program (HPCMP) Corps of Engineers Waterways Experiment Station (CEWES) Major Shared Resource Center (MSRC), in Vicksburg, Mississippi.

[2]Multiple Program, Multiple Data.

[3]Single Program, Single Data.

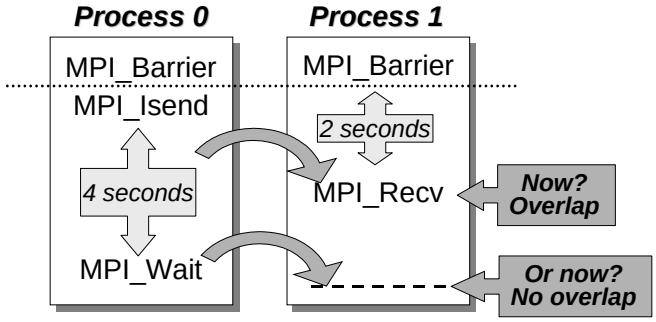

Fig. 1. An experiment to test support for synchronization overlap.

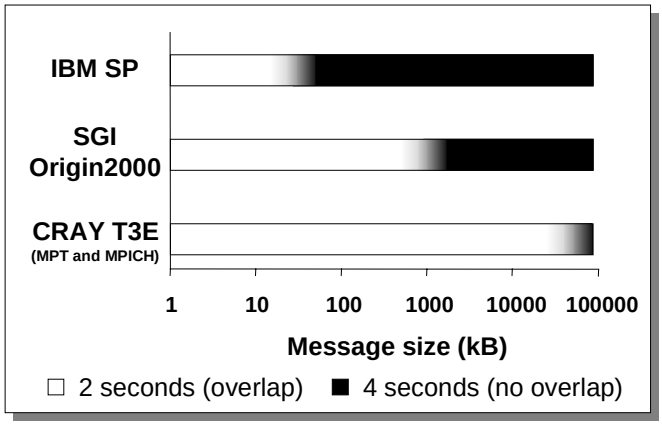

Fig. 2. Results of the synchronization-overlap experiment in the default execution environment of each system.

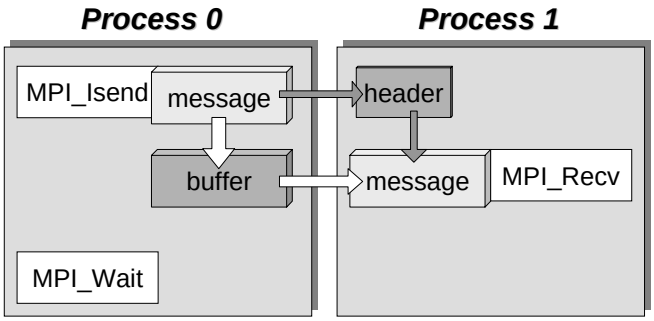

Fig. 3. Synchronization overlap on the CRAY T3E for messages smaller than available memory.

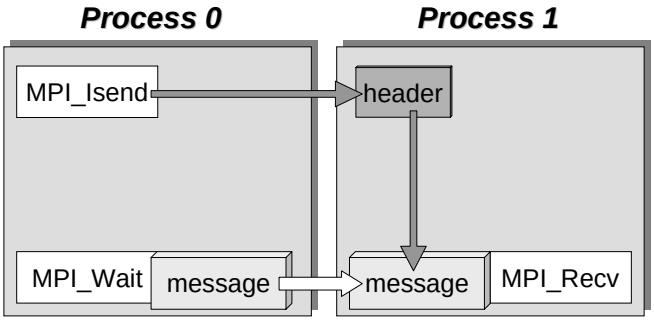

Fig. 4. No overlap on the CRAY T3E for messages larger than available memory.

process waits for 2 seconds after the `MPI_Barrier` before issuing an `MPI_Recv`, which blocks execution until the message arrives.

The "result" of the experiment is the completion time of the `MPI_Recv` after the `MPI_Barrier`. For MPI implementations that support synchronization overlap, the `MPI_Recv` completes almost immediately, yielding a time of about 2 seconds. For MPI implementations that do not support this overlap, the two processes must synchronize. The `MPI_Recv` does not complete until the `MPI_Wait` executes, yielding a time of about 4 seconds.

Figure 2 displays the results of the experiment for a range of message sizes. The results represent the default environment on each machine; no environment variables have been modified. For all message sizes in the figure, the time for data transfer is at least an order of magnitude smaller than the multiple-second delay in the experiment. Therefore, any time for data transfer does not contribute significantly to the `MPI_Recv` time. The change from a 2-second to a 4-second delay indicates a change from synchronization overlap to no overlap.

Each system shows a distinctive range of message sizes where overlap is supported by default. The differences in the ranges reflect differences in the MPI implementations. In addition, each system has a unique method for increasing the range of overlap well beyond the default range shown in figure 2.

A. *Cray MPT and MPICH on the CRAY T3E*

The CRAY T3E is a multiprocessor with a single system image and distributed, globally addressable memory [3]. Both the vendor-supplied MPI and MPICH support synchronization overlap up to very large message sizes. Figure 3 models how the vendor-supplied MPI implements synchronization overlap. The CRAY T3E supports one-sided communication; each processing element can access the memory of a remote processing element without the involvement of the remote CPU [3]. Using this one-sided communication, the `MPI_Isend` writes a message header in memory local to the receiving process [4]. It also make a copy of the message in a local buffer. The `MPI_Recv` then uses the header information and one-sided communication to read the contents of the remote buffer.

For this procedure to work, the sending process must have enough free memory to allocate the buffer. Figure 4 models how the vendor-supplied MPI implements message passing when the message is too large to copy into a buffer. As before, the `MPI_Isend` writes a header to the receiving process. The `MPI_Recv` blocks until the sender calls `MPI_Wait`, however, eliminating the opportunity for overlap.

The range of message sizes allowing overlap is limited only by available buffer space, and thus only by available memory. Increasing available memory increases the range

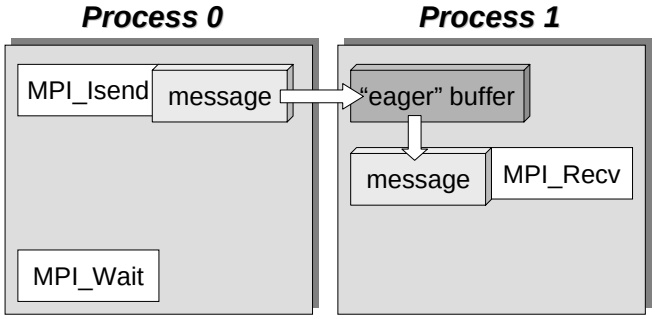

Fig. 5. Synchronization overlap on the IBM SP using eager communication.

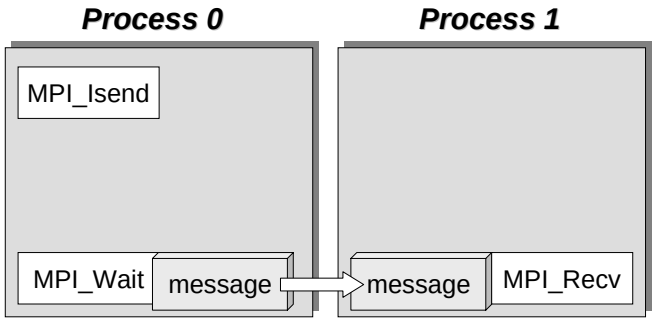

Fig. 6. No overlap on the IBM SP for messages larger than the eager limit.

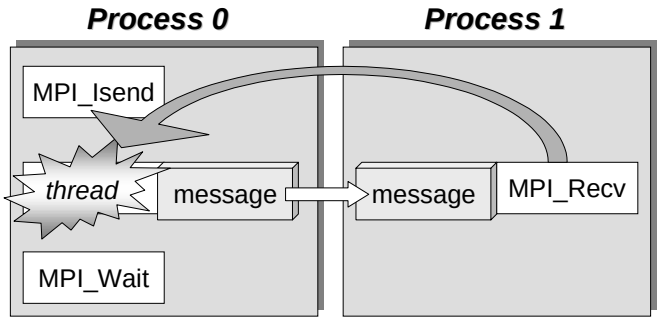

Fig. 7. Synchronization overlap on the IBM SP with `MP_CSS_INTERRUPT` set to `yes`.

accordingly.

It is interesting to ask why the vendor-supplied MPI bothers to buffer the outgoing message. A cynical—though possibly accurate—answer is that the buffering makes the MPI implementation "streams safe". Early models of the T3E have a limitation on the use of the interprocessor communication hardware with the unique stream-buffer hardware used to improve memory bandwidth [5]. These early T3Es can crash if both these hardware subsystems attempt to access nearby memory locations at the same time. The vendor-supplied MPI either buffers each message or blocks execution, avoiding all dangerous memory accesses. More modern T3Es, such as the one at the CEWES MSRC, do not have this hardware limitation and do not require message buffering for safety. Communication-bandwidth experiments described in [6] imply that the vendor-supplied MPI has not been modified for the improved hardware.

B. IBM PE on the IBM SP

The IBM SP is a multicomputer; each node has its own system image and local memory [7]. By default, the vendor-supplied MPI implementation on the SP supports synchronization overlap only for messages up to tens of kilobytes. The support for overlap ends at the "eager limit," which is defined by the environment [8]. Figure 5 models how the vendor-supplied MPI implements synchronization overlap through "eager" communication. Each process allocates a buffer in local memory for messages from each of the other processes. The `MPI_Isend` writes the message to the buffer space assigned to the sending process in the local memory of the receiving process. The `MPI_Recv` then copies the message from this buffer.

Figure 6 models how the vendor-supplied MPI implements message passing for messages larger than the eager limit, in the default environment. The `MPI_Isend` contributes little to actual data transfer, and the `MPI_Recv` simply blocks until the `MPI_Wait` executes. In other words, the processes must synchronize.

One method of increasing the range of overlap is simply increasing the eager limit using the `MP_EAGER_LIMIT` environment variable [8]. Increasing the eager limit is not a scalable solution, however. To receive eager sends, each process must have separate buffer space for every other process. Therefore, the total buffer space grows as the square of the number of processes. Even if more than enough buffer space is available, the IBM Parallel Environment imposes a hard limit of 64 kB on the eager limit [8].

A more scalable method of increasing the range of overlap is to allow the SP's communication subsystem to interrupt execution [9]. If the environment variable `MP_CSS_INTERRUPT` is set to `yes`, the communication subsystem will interrupt computation to send or receive messages. The default setting is `no`.

Figure 7 models how the vendor-supplied MPI implements synchronization overlap when interruption is allowed. The `MPI_Recv` causes the communication subsystem to interrupt the sending process, and a communication thread becomes active and completes the send. Computation resumes once the message is sent.

C. MPT for IRIX on the SGI Origin2000

The SGI Origin2000 is a multiprocessor with one system image and physically distributed, logically shared memory [10]. By default, the vendor-supplied MPI implementation on the CEWES MSRC Origin supports synchronization overlap for messages up to about a megabyte. Other Origins may have different defaults.

Within a single Origin system, all MPI messages move through shared buffers [4]. Figure 8 models how the vendor-supplied MPI implements synchronization overlap when the message fits within the shared buffer. The

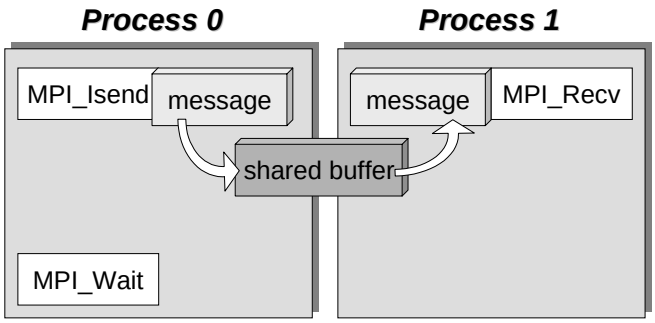

Fig. 8. Synchronization overlap on the SGI Origin2000 for messages fitting within the shared buffer.

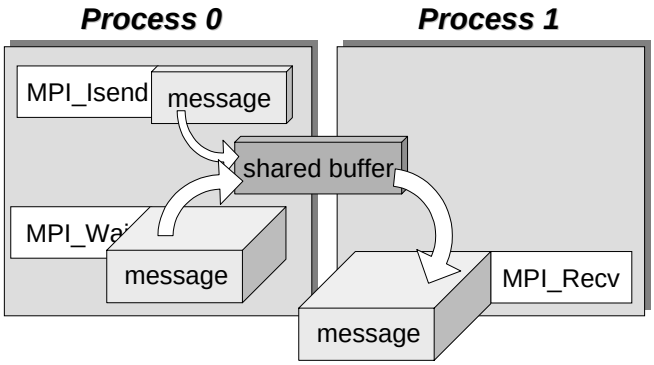

Fig. 9. Incomplete synchronization overlap on the SGI Origin2000 for messages larger than the shared buffer.

`MPI_Isend` copies the message to the shared buffer, and the `MPI_Recv` copies it back out.

Figure 8 models how the vendor-supplied MPI implements message passing when the message does not fit within the shared buffer. The `MPI_Recv` does not complete until the `MPI_Wait` transfers the remaining blocks of the message through the shared buffer. As a side effect, the two processes must synchronize.

A straightforward method for increasing the range of overlap is simply increasing the size of the shared buffers using the `MPI_BUFS_PER_PROC` environment variable, where the value of the variable indicates the number of memory pages [4]. Unlike eager-buffer space on the IBM SP, the total shared-buffer space on the Origin varies with the number of processes, not with the square of that number. Therefore, this method of increasing the range of overlap does not suffer from such poor scalability.

D. Summary

Each of the tested implementations support the overlap of computation and synchronization over a wide range of message sizes, either by default or with minor changes to the environment. Therefore, all the implementations should support the efficient execution of asynchronous applications. The experiment itself is asynchronous to an extreme. The benefits, if any, of synchronization overlap for synchronous applications are not so obvious.

III. Overlap of Computation with Data Transfer

MPI implementations that support synchronization overlap may or may not support data-transfer overlap. Support for synchronization overlap can fall into three categories according to the type of data transfer: two sided, one sided, and third party.

An MPI implementation may support synchronization overlap without any support for data-transfer overlap. This corresponds to *two-sided* data transfer, where one process must interrupt the execution of another process for transfer to occur.

This interruption can be avoided in implementations that support *one-sided* data transfer. In this case, only one of the two processes sharing a message must be involved in data transfer at a time. The other process is free to continue computing, resulting in a limited form of data-transfer overlap.

In contrast, *third-party* data transfer allows full data-transfer overlap. The computing processes operate concurrently with a third party that handles the communication.

The experiment described in the previous section does not differentiate between these three possible implementations of synchronization overlap or the corresponding levels of data-transfer overlap. The experiment does not directly measure the time for data transfer. Even if it did, this time would often be orders of magnitude smaller than the computation time.

To differentiate between implementations of synchronization overlap and to test support for data-transfer overlap, we employ a larger experiment based on a semiconductor device simulation. This application is described in detail in [11]. It uses an unstructured two-dimensional grid that is partitioned among parallel processes, and it relies on a Krylov subspace iterative solver. The runtime is dominated by the many sparse matrix-vector multiplications executed by the solver. Each parallel multiplication requires the communication of values at the interfaces of the distributed grid partitions. With carefully chosen partitions, the computation and communication are both load balanced, resulting in a synchronous application.

The experiment uses two versions of this application that differ only in the implementation of the parallel matrix-vector multiplication. One version has separate phases of communication and computation, and the other tries to overlap communication with computation.

Figure 10 presents a schematic of the version with separate phases of communication and computation, the nonoverlap version. Each process first computes its share of the matrix-vector multiplication. It then issues `MPI_Irecv`s and `MPI_Isend`s to share partition-interface values, followed immediately by an `MPI_Waitall` that blocks until communication completes.

Figure 11 presents a schematic of the version of parallel multiplication designed to overlap communication with computation. The computation is split into two pieces. Each processor first calculates the multiplication results

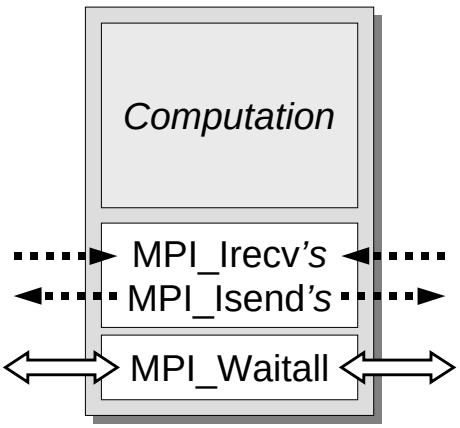

Fig. 10. Parallel matrix-vector multiplication with separate phases of communication and computation.

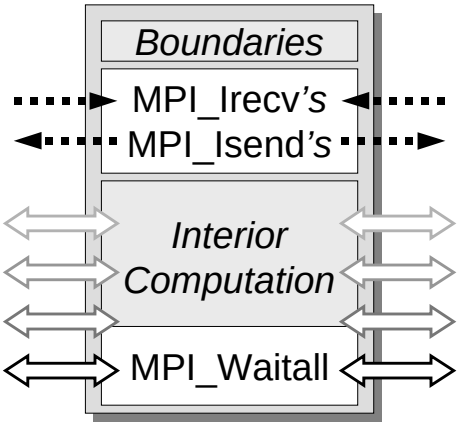

Fig. 11. Parallel matrix-vector multiplication programmed explicitly to overlap communication with computation.

for the points at the interfaces of its partition of the grid. The bulk of the computation, the interior of the partition, occurs after the `MPI_Irecv`s and `MPI_Isend`s but before the `MPI_Waitall`. The intent of this design is to allow communication to occur concurrently with the interior computation.

A. Expected results

Common wisdom suggests that the overlap version of parallel sparse matrix-vector multiplication should provide better performance [2]. The caveat in [2] of "suitable hardware" should not be underestimated, however. The degree to which any performance improvement is realized depends on the level of overlap supported by the MPI implementation and the underlying system. All the implementations tested here support synchronization overlap, but they differ in their specific implementation of this overlap.

Consider what might happen with the parallel multiplication experiment for synchronization overlap that uses two-sided communication. This case corresponds to the MPI implementation on the IBM SP for messages larger than the eager limit when `MP_CSS_INTERRUPT` is set to `yes`.

In both versions of the application, no data-transfer overlap can occur for any one message. The runtime is at least the sum of computation time and maximum per-process data-transfer time. Any advantage of the overlap version is a higher-order effect related to inherent asynchrony of the application; load imbalance in the communication pattern may cause synchronization overhead that the overlap version can avoid. For a primarily synchronous application such as this, the performance improvement should be minimal.

A similar result is likely for synchronization overlap that uses one-sided communication. Though one-sided communication can provide partial data-transfer overlap, the synchronous nature of the application avoids overlapping data transfer with actual computation. In both versions of the application, all the processes communicate at roughly the same time. Therefore, each one-sided data transfer overlaps with other one-sided data transfers, not with computation. Again, any advantage of the overlap version is a higher-order effect, and performance improvement should be minimal. All the tested MPI implementations correspond to this one-sided case for some range of message sizes: under the eager limit on the SP, smaller than the shared buffers on the Origin, and smaller than available memory on the T3E.

For no message sizes, however, do any of the tested implementations correspond to the remaining category of synchronization overlap, where data transfer is handled by a third party. Yet it is just this category that offers the greatest promise for the overlap version of the application. In the nonoverlap version, the processes block at the `MPI_Waitall` while the third party completes the communication. Conversely, the overlap version performs useful computation while the third party transfers data. If the computation and communication take comparable time, the overlap version can take as little as half the time of the nonoverlap version.

B. Actual results

Table I presents the results for the performance improvement of the overlap version of the test application over the nonoverlap version. The results are for a specific data set of about 10,000 grid points run on 16 processors. Other problem and system sizes found in [11] show similar results. The performance improvement is calculated as follows, where T_{overlap} and $T_{\text{nonoverlap}}$ are the execution times for the overlap version and nonoverlap version, respectively:

$$\text{Improvement} = \frac{\mathrm{T}_{\text{overlap}} - \mathrm{T}_{\text{nonoverlap}}}{\mathrm{T}_{\text{nonoverlap}}} \times 100\%.$$

As predicted, the overlap version shows little improvement over the nonoverlap version on the SP and the Origin. On the T3E, however, the overlap version runs dramatically faster, contradicting the prediction. More surprisingly, the improvement of almost 80% is significantly higher than the theoretical peak improvement of 50% given by perfect overlap!

TABLE I

PERFORMANCE IMPROVEMENT OF PROGRAMMING FOR OVERLAP

MPI implementation	*Improvement*
IBM PE on IBM SP	0.4%
MPT for IRIX on SGI Origin2000	5.3%
Cray MPT 1.2.0.1 on CRAY T3E	78.6%

TABLE II

COMPARISON OF MPI IMPLEMENTATIONS ON THE CRAY T3E

Implementation	$T_{\text{nonoverlap}}$	T_{overlap}	*Improvement*
MPT 1.2.1.1	55.61	14.85	73.3%
MPT 1.2.1.2	15.32	14.59	4.1%
MPICH	12.48	12.54	−0.4%

Times are in seconds.

Table II presents newer results for the T3E that include different implementations of MPI: MPT 1.2.1.1, MPT 1.2.1.2, and the MPICH implementation available from Mississippi State University [12]. Unlike MPT 1.2.1.1, MPT 1.2.1.2 validates the prediction by almost eliminating the advantage of the overlap version. MPICH goes a step further by completely eliminating this advantage, in addition to providing better absolute performance. The dramatic "improvement" shown by MPT 1.2.0.1 and MPT 1.2.1.1 appears to be a result of a bug that has been removed from MPT 1.2.1.2.

IV. CONCLUSIONS

MPI implementations provide different levels of support for the overlap of computation with communication. The implementations tested here all support the overlap of computation with synchronization, a capability particularly useful for asynchronous parallel applications. In contrast, these MPI implementations provide only limited support for the overlap of computation with data transfer.

This limitation has important implications for applications programmed explicitly for overlap. For synchronous applications, "programming for overlap" actually implies a specific level of overlap: full data-transfer overlap. This level is only provided by implementations using third-party communication, not two-sided or even one-sided communication.

The parallel systems and MPI implementations tested here are common targets for large-scale parallel computations. None of them provide third-party communication, and none show significant performance improvement for the overlap experiment modeled in figures 10 and 11. This negative result shows that programming explicitly for overlap is clearly not a portable performance enhancement for the synchronous application used in the experiment.

According to the same arguments used to predict this result, programming for overlap may be of little benefit for synchronous applications in general. Many current MPI implementations—all those tested here—do not support the required level of overlap. The additional development effort and code complexity required to program for overlap seem unjustified.

ACKNOWLEDGMENTS

The authors are grateful to Purushotham Bangalore and Shane Hebert of Mississippi State University for providing insight into the mechanics of MPI implementations. This research was sponsored in part by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through Programming Environment and Training. Views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of Defense Position, policy, or decision unless so designated by other official documentation.

REFERENCES

[1] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, 1994.

[2] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, University of Tennessee, Knoxville, 1995.

[3] *Performance of the CRAY T3E Multiprocessor*, Silicon Graphics, 1999.

[4] *Message Passing Toolkit: MPI Programmer's Manual*, Cray Research, 1998.

[5] *CRAY T3E Programming with Coherent Memory Streams*, Version 1.2, Cray Research, 1996.

[6] J. White III, *Using the CRAY T3E at OSC*, Ohio Supercomputer Center, 1998.

[7] H. Kitzhöfer, A. Dunshea, F. Mogus, *RS/6000 SP Performance Tuning*, IBM, 1998.

[8] *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, Version 2, Release 3, IBM, 1997.

[9] *IBM Parallel Environment for AIX: Operation and Use, Volume 1: Using the Parallel Operating Environment*, IBM, 1997.

[10] D. Cortesi, *Origin2000 and Onyx2 Performance Tuning and Optimization Guide*, Silicon Graphics, 1998.

[11] S. Bova and G. Carey, "A Distributed Memory Parallel Element-by-Element Scheme for Semiconductor Device Simulation," *Computer Methods in Applied Mechanics and Engineering*, in press.

[12] L. Hebert, W. Seefeld, and A. Skjellum, *MPICH on the CRAY T3E*, PET Technical Report 98-31, CEWES MSRC, 1998.