

ORNL/TM-13682

ornl

**OAK RIDGE
NATIONAL
LABORATORY**

LOCKHEED MARTIN 

**Impact of Communication
Protocol on Performance**

P.H. Worley

MANAGED AND OPERATED BY
LOCKHEED MARTIN ENERGY RESEARCH CORPORATION
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

IMPACT OF COMMUNICATION PROTOCOL ON PERFORMANCE

P. H. Worley
Computer Science and Mathematics Division

Date Published: February 1999

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6285
managed by
LOCKHEED MARTIN ENERGY RESEARCH CORP.
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-96OR22464

CONTENTS

	Page
LIST OF FIGURES	v
LIST OF TABLES	vii
ABSTRACT	ix
1. Introduction	1
2. PSTSWM	3
3. Communication Protocols	4
4. Platforms	7
5. Serial Performance	10
6. Point-to-Point Communication Performance	13
7. Parallel Algorithm Sensitivities	18
8. Parallel Algorithm Comparisons	22
9. Full Simulation Performance	27
10. Conclusions	32
11. Acknowledgements	33
REFERENCES	35

LIST OF FIGURES

Figure		Page
1.	Serial MFlop/second rates.	12
2.	Peak observed bandwidth (megabytes/second).	16
3.	2MB SWAP experiments.	17
4.	Transpose FFT algorithm comparisons: relative performance degradation as compared to best algorithm.	24
5.	Distributed LT algorithm comparisons: relative performance degradation as compared to best algorithm.	25
6.	DTH comparisons.	29
7.	DR comparisons.	30

LIST OF TABLES

Table		Page
1.	MPI SWAP protocols (simplified)	5
2.	Unordered SHMEM SWAP protocols (simplified)	7
3.	Parallel platforms	9
4.	Serial MFlop/second rates with (m) and without math libraries	11
5.	Peak observed bandwidth (megabytes/second) and latency (microseconds) for optimal protocol	15
6.	Problem sizes used for parallel algorithm studies	20
7.	Effect of SHMEM protocol on performance of parallel algorithms	21
8.	Effect of MPI protocol on performance of parallel algorithms	22

IMPACT OF COMMUNICATION PROTOCOL ON PERFORMANCE

Patrick H. Worley

We use the PSTSWM compact application benchmark code to characterize the performance behavior of interprocessor communication on the SGI/Cray Research Origin 2000 and T3E-900. We measure

1. single processor performance,
2. point-to-point communication performance,
3. performance variation as a function of communication protocols and transport layer for collective communication routines, and
4. performance sensitivity of full application code to choice of parallel implementation.

We also compare and contrast these results with similar results for the previous generation of parallel platforms, evaluating how the relative importance of communication performance has changed.

1. Introduction

Communication costs often represent a significant fraction of the run time of parallel application codes, and the choice of communication protocol is an important step in porting and tuning codes. Here we refer to *communication protocol* as any aspect of the interprocessor communication logic that does not change the basic functionality of the parallel algorithms. Low-level examples include the transport layer (e.g., MPI or PVM) and the message-passing commands (MPI_BSEND, MPI_SEND, MPI_ISEND, etc.). Higher-level examples include the number, order, and size of messages sent in some collective or extended operation, as long as the final locations of the results are fixed, or code restructuring to overlap communication with computation. Differences in parallel algorithms that affect problem decomposition, computational complexity, or load balance are not communication protocol issues.

Although general techniques exist for optimizing interprocessor communication, the importance of optimization and efficacy of the different techniques are often platform specific. For example, in previous work we examined the effect of different communication protocols on performance for such platforms as the Intel Paragon, IBM SP2, and Cray Research T3D [4],[14]. Although these machines are all “classic” distributed-memory MIMD parallel systems, their performance characteristics differ significantly, as do their optimal tuning techniques and parameters. In this paper we examine communication performance sensitivities of the SGI/Cray Research Origin 2000 and T3E-900 systems. We also compare and contrast these results with similar results for the older platforms, evaluating how the performance characteristics of interprocessor communication have changed.

A typical approach to evaluating interprocessor communication and communication libraries is to measure the performance of individual commands in isolation or in small kernels representing common communication functions [2], [7]. For example, this approach has been used in [9] and [8] to evaluate communication performance on the Origin 2000 and the T3E. Although these types of experiments are an important step in an evaluation, the communication protocols and the controlled measurement environment used in the experiments may not be typical of how the commands are used in practice, making it difficult for an application developer to interpret the results. The “low-level” measurements are also not sufficient for evaluating many of the optimization

techniques, for example, latency hiding or overlapping communication and computation. The performance of full application codes can also be used to report performance, but the protocols used in these are typically fixed, and the sensitivities are not easily identified. To deal with these issues, benchmark suites that include low-level measurement codes, kernel codes, and compact application codes can be used [1],[6],[7],[12]. But the linkages between the different levels of measurement are often difficult to establish, primarily because the measurements specified in the benchmark suite are not focused on any particular performance question. Instead, large data sets are generated, often requiring significant computational resources, that hopefully are sufficient to address the desired performance questions. How to use the data to address such questions is left to the user.

To help assess the performance impact of tuning interprocessor communication protocols, we use an integrated suite of tests that are derived from or motivated by the Parallel Spectral Transform Shallow Water Model (PSTSWM) parallel application code [15], [16]. PSTSWM was developed to evaluate strategies for parallelizing spectral global atmospheric circulation models [4], [5], and has imbedded a large number of parallel algorithm options. Among these options are numerous choices for the communication protocols used to implement the different parallel algorithms and numerous choices of message-passing layer. We use PSTSWM to examine

1. single processor performance,
2. peak achievable point-to-point communication performance,
3. performance variation as a function of communication protocols and transport layer for parallel fast Fourier transforms (FFT), transpose, and global vector sum algorithms,
4. performance of vendor-supplied collective communication routines, and
5. performance sensitivity of full application code to choice of parallel implementation (including both choice of parallel algorithm and choice of communication protocol)

using both the MPI [10] and SHMEM libraries to implement interprocessor communication. The performance of PSTSWM is sensitive to communication performance, both point-to-point and collective, and both local and distant. Thus, while the results

of this study necessarily reflect the peculiarities of the PSTSWM application code, the overall conclusions as to communication performance sensitivities should be more generally applicable.

2. PSTSWM

The PSTSWM is a message-passing parallel benchmark code and parallel algorithm testbed that solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method. PSTSWM was developed by the author and by I. T. Foster at Argonne National Laboratory from the serial code STSWM, written by J. J. Hack and R. Jakob of the National Center for Atmospheric Research (NCAR). PSTSWM was used to evaluate parallel algorithms for the spectral transform method as it is used in global atmospheric circulation models. It is also a “compact application” in the Parallel Kernels and Benchmarks Suite (ParkBench) [7].

PSTSWM is a spectral timestepping code. During each timestep of the model simulation, the state variables of the problem are transformed between the physical domain, where most of the physical forces are calculated, and the spectral domain, where the terms of the differential equations are evaluated. The physical domain is a tensor product longitude-latitude-vertical grid, and transforming from physical coordinates to spectral coordinates involves performing a real FFT for each line of constant latitude, followed by integration over latitude using Gaussian quadrature, approximating the Legendre transform (LT). The inverse transform involves evaluating sums of spectral coefficients (“inverse LT”) and inverse real FFTs.

The parallel algorithms in PSTSWM are based on decompositions of the physical and spectral computational domains over a logical two-dimensional processor mesh of size $PX \times PY$. For the FFT and LT, there are two general families of parallel algorithms: distributed algorithms, using a fixed data decomposition and computing results where they are assigned, and transpose algorithms, remapping the domains to allow the transforms to be calculated serially.

The supported domain decompositions all have the property that FFTs in different processor rows are independent. Each row of PX processors collaborates in computing a “block” of FFTs, and all interprocessor communication for a given FFT is restricted to a given processor row. Similarly, the LTs in different processor columns are inde-

pendent. Each column of PY processors collaborates in computing a block of LTs, and all interprocessor communication for a given LT is restricted to a given processor column. It is important to keep this in mind in the later discussion of the communication patterns for the different parallel algorithms.

Parallel performance of PSTSWM is determined by

- communication costs in the parallel FFT and LT algorithms,
- copy costs in the parallel FFT and LT algorithms,
- computation rate, and
- load balance.

As is described later, our choices of parallel algorithms and domain decompositions used in the experiments minimize load imbalance, and the performance variation between the different parallel algorithms is primarily attributable to differences in communication costs and related issues (like copy costs).

We have found PSTSWM to have many characteristics that make it useful for performance studies. First of all, it is easy to use. It was designed for these types of studies, and we have previously developed numerous scripts and other tools for running and analyzing experiments. PSTSWM also makes interesting and varied demands on the communication subsystem, both in terms of communication protocol and communication patterns. Finally, PSTSWM is still relevant to an important application area. It is an excellent predictor of performance of parallel spectral atmospheric models, and optimized algorithms developed in PSTSWM can be ported easily to the NCAR spectral atmospheric models. See <http://www.epm.ornl.gov/champp/pstswm/index.html> for a partial bibliography of other performance studies using PSTSWM.

3. Communication Protocols

Performance-critical interprocessor communication in PSTSWM is implemented using two basic types of commands: SWAP and SENDRECV. The message-passing transport layer used to implement these commands is specified at compile time, while the rest of the communication protocols are specified at run time. For the Origin 2000 and the T3E-900, we use either MPI or SHMEM.

The options in PSTSWM for implementing SWAP using MPI are listed in Table 1. (Analogous options exist for SENDRECV.) Two general classes of communication protocols are available: *unordered* (ping-ping) and *ordered* (ping-pong), where the unordered protocols attempt to exploit bidirectional bandwidth and the ordered explicitly avoid it. Note that the examples have been simplified (to save space) and do not accurately represent the MPI implementations. For example, handshaking messages required for correct use of the *ready send* command have been omitted. These protocols are described in more detail in [16].

Table 1. MPI SWAP protocols (simplified)

Unordered	Ordered
(0,0): <u>simple</u> Processors 1 and 2 MPI.BSEND MPI.RECV	(1,0): <u>simple</u> Processor 1 Processor 2 MPI.SEND MPI.RECV MPI.RECV MPI.SEND
(0,1): <u>nonblocking send</u> Processors 1 and 2 MPI.ISEND MPI.RECV	(1,1): <u>nonblocking send</u> Processor 1 Processor 2 MPI.ISEND MPI.RECV MPI.RECV MPI.SEND
(0,2): <u>nonblocking receive</u> Processors 1 and 2 MPI.IRECV MPI.SEND	(1,2): <u>nonblocking receive</u> Processor 1 Processor 2 MPI.IRECV MPI.RECV MPI.SEND MPI.SEND
(0,3): <u>nonblocking send & receive</u> Processors 1 and 2 MPI.IRECV MPI.ISEND	(1,3): <u>nonblocking send & receive</u> Processor 1 Processor 2 MPI.IRECV MPI.RECV MPI.ISEND MPI.SEND
(0,4): <u>ready send</u> Processors 1 and 2 MPI.IRECV MPI.IRSEND	(1,4): <u>ready send</u> Processor 1 Processor 2 MPI.IRECV MPI.RECV MPI.IRSEND MPI.IRSEND
(0,5): <u>nonblocking ready send</u> Processors 1 and 2 MPI.IRECV MPI.IRSEND	(1,5): <u>nonblocking ready send</u> Processor 1 Processor 2 MPI.IRECV MPI.RECV MPI.IRSEND MPI.IRSEND
(0,6): <u>native sendrecv</u> Processors 1 and 2 MPI.SENDRECV	(1,4): <u>synchronous</u> Processor 1 Processor 2 — MPI.RECV MPI.SEND — — MPI.SEND MPI.RECV —

For a single SWAP request, we would expect MPLSENDRECV to be the most efficient implementation. We examine the other options for three reasons.

1. In some previous studies on other platforms, MPLSENDRECV was not the optimal implementation for SWAP or SENDRECV.
2. Many of the other message-passing systems do not have an equivalent to MPLSENDRECV, although they do have equivalents to the other protocol options. Including these options provides for a consistent testing methodology.
3. Techniques for overlapping communication with computation or for hiding latency require that we “expand” the SWAP command, posting some requests early (send or receive) and delaying other requests as long as possible. We can also combine these components for multiple SWAPs, allowing us to increase the granularity of the communication. The non-MPLSENDRECV protocols are used as the basis for these expansions and reorganizations.

However, we refer to the (0,6) (MPLSENDRECV) protocol as the generic or default MPI communication protocol, as it is what we would expect to be optimal knowing nothing else about the platform.

There are many other MPI communication commands that can be used to implement SWAP and SENDRECV. Our choices are primarily historical, reflecting the capabilities of the NX [11] communication library more than MPI. However, little is actually being ignored. The communication patterns, message sizes, and buffer addresses vary throughout the code, and the MPI persistent communication requests are not appropriate for this code. The MPI synchronous commands are also unlikely to be performance enhancers. Generally, relaxing the order requirements specified by the communication routines is the most effective way of improving communication performance in situations when MPLSENDRECV is not the optimal protocol.

The SHMEM protocols are not as straightforward. SHMEM provides the ability to write directly into or read directly from another processor’s address space, using the commands `put` and `get`, respectively. For PSTSWM, we implement SWAP using the protocols described in Table 2.

There are also analogous ordered protocols: (1,1), (1,2), and (1,6). Although the SHMEM `put` and `get` commands are themselves “blocking,” we refer to these SWAP

Table 2. Unordered SHMEM SWAP protocols (simplified)

- (0,1): nonblocking send
put address of outgoing message on the other processor
wait for address of incoming message
get incoming message from the other processor
- (0,2): nonblocking receive
put address where to put incoming message on the other processor
wait for address of where to put outgoing message
put outgoing message on the other processor

implementations as nonblocking. Once processor 1 has **put** the address of the buffer on processor 2, processor 1 does not need to wait for processor 2 to complete the SWAP before doing something else. Processor 1 must simply check that the SWAP has completed before (re)using the message buffers, as is characteristic of nonblocking communication. As with the descriptions of the MPI options, the descriptions of the SHMEM protocols have been simplified.

Unlike the MPI implementations, there is no obvious default SHMEM protocol. On the T3D, **put** is twice as fast as **get**. On the T3E, **get** is somewhat faster than **put**. For the purposes of this study, we (arbitrarily) choose (0,1) to be the default SHMEM protocol.

Until MPI-2, and its one-sided communication primitives, become more commonplace, SHMEM functionality will not be generally available on non-SGI/Cray platforms. For this reason we will use the performance of the MPI (0,6) protocol as the baseline upon which to compare the performance improvements possible from optimizing the communication protocols. We will also compare performance when using the optimal MPI protocol to that when using the optimal SHMEM protocol as an indicator of what is being lost by using MPI instead of a lower-level transport layer.

4. Platforms

Although we focus on the Origin 2000 and T3E-900 in these studies, we also include some measurements from all of the platforms listed in Table 3. The platforms are listed by the approximate date of introduction of the architecture.

The SGI Origin 2000, henceforth referred to simply as the Origin, is a distributed

shared memory (DSM) parallel system made up of “nodes” consisting of two processors sharing a common memory. Nodes are interconnected via a high-performance, highly connected, but still nonuniform access, network. Thus, although all memory is globally accessible, access time varies with the network distance between the memory and accessing processor. The Origin supports traditional shared memory programming models, but current experience indicates that the “programming discipline” natural to message-passing is important for performance, and message-passing is a reasonable approach to using the machine. For these experiments we used version 6.5 of the IRIX operating system, MPI and SHMEM from version 1.2.0.1 of the SGI Message Passing Toolkit, and the MipsPro 7.20 Fortran compiler with compiler options -O3 -64.

The T3E-900, henceforth referred to as the T3E, is the second-generation distributed memory parallel system designed by Cray Research. Each node consists of a single processor/memory pair interconnected via a high-performance, three-dimensional bidirectional torus network. Hardware support exists for accessing remote memory directly, but experience has shown that message-passing is still the best programming paradigm to use if high performance is required. For these experiments we used version 2.0.2.28 of the UNICOS/mk operating system, MPI and SHMEM from version 1.2.0.2 of the Message Passing Toolkit, and the Cray CF90 version 3.0.2.1 Fortran compiler with compiler options -dp -Oscalar3. Other details about the Origin and the T3E used in these experiments are described in Table 3.

Performance metrics are rarely static over time, so please note the date that data were collected and the system specifics before extrapolating performance to current machines. In particular, the Origin 2000 system used at Los Alamos National Laboratory is part of a research effort to build a large parallel system from individual commercial-scale component systems, and performance may not reflect that at a more production-oriented site. The variability in performance is still interesting and important data.

As is shown in the section describing serial performance, an important performance enhancer is optimized math libraries. For PSTSWM, the most important routines are real and complex FFTs for multiple vectors (block transforms). The vectors being transformed are relatively short, but many independent transforms are needed. Thus, nonblock FFT routines are not useful, often being slower than simple Fortran rou-

Table 3. Parallel platforms

- Paragon: Intel Paragon XP/S 150 MP at Oak Ridge National Laboratory.** This machine has 1024 MP nodes (3 50-MHz iPSC/860 processors per node) in a 16×64 grid interconnect. Only one processor per node was used for computation. The OSF operating system, NX and MPI message-passing libraries, and Kuck and Associates (KAI) math routines were used. Some measurements were also taken using the SUNMOS operating system. OSF measurements were taken in January 1998. SUNMOS measurements were taken earlier.
- T3D: CRI T3D at Cray Research in Eagen, Minn..** This machine had 128 150-MHz DEC Alpha EV4 processors. SHMEM and CRI/EPCC MPI message-passing libraries were used. No math libraries were available. Measurements were taken in August 1996.
- SP2/66: IBM SP2 at NASA Ames Research Center.** This machine had 160 RS6000/590 nodes ("wide," 66.7 MHz POWER2). MPL and MPI message-passing libraries and ESSL math routines were used. Measurements were taken in August 1996.
- SPP-1200: Convex SPP-1200 at the National Center for Supercomputer Applications.** This machine has 64 120-MHz HP PA-RISC 7200 processors (8 Hypernodes). The MPI message-passing library was used. No math libraries were available. Measurements were taken in September 1996.
- T3E-900: SGI/CR T3E-900 at the National Energy Research Scientific Computing Center.** This machine has 532 450-MHz DEC Alpha EV5 RISC processors. SHMEM and CRI/EPCC MPI message-passing libraries and LIBSCI math routines were used. Measurements were taken in May 1998.
- SPP-2000: HP/CONVEX SPP-2000 at the National Center for Supercomputer Applications.** This machine has 64 180-MHz HP PA-RISC 8000 processors (4 Hypernodes). The MPI message-passing library and VECLIB math routines were used. Measurements were taken in April 1998.
- PII/266: Intel PII-266 cluster at Oak Ridge National Laboratory.** This machine has 10 266-MHz dual Pentium II nodes. Portland Group f77 compilers were used. Only serial measurements were taken for this report. Measurements were taken in February 1998.
- Origin/195: SGI/CR Origin 2000 at Los Alamos National Laboratory.** This machine has 128 195-MHz MIPS R10000 processors. SHMEM and MPI message-passing libraries and SCSL math routines were used. Measurements were taken in May 1998.

tines that do block the transforms. The description for each system indicates whether optimized math libraries with the needed capabilities were available at the time the performance data were collected.

5. Serial Performance

To determine parallel scalability (and the effect of communication costs on performance), it is important to establish a serial baseline. The goal of the study described in this section was to determine the “peak achievable” serial performance that would be attained in long (production) simulations.

PSTSWM computes the solution by timestepping, advancing the approximation to a new time level (in simulation time) by using the approximations at the two previous time levels. In the following, we will refer to the process of advancing the approximation to a new time level as a step.

The computational complexity and code executed for a step in PSTSWM are identical for all steps, and all steps “should” have the same execution time. The code was run multiple times for a given problem size, and the fastest time was used. We also measured the minimum and maximum execution times for individual steps for a given run, and this information was used to determine whether extraneous effects (rare system interrupts or other users) contaminated the timing unacceptably.

To better approximate the performance achieved in long simulations when timing only a relatively short run, we calculated one step, then reinitialized, before beginning timing. This guaranteed that the code and data memory were all “touched” before timing began, eliminating some transient caching effects. It also eliminated the time for loading and initializing the program.

The results (in Fig. 1 and Table 4) are presented in terms of MFlop/second rates for one processor runs for a number of different problem sizes. The MFlop rate was approximated using the floating point operation count returned by the hardware performance monitor for a single processor run on a Cray C90. Multiple runs were performed with differing compiler optimization options, and we used the minimum number of floating point operations measured. The number of steps computed were also varied in the measurements, so that operations corresponding to initialization and other startup overhead could be removed, and the operation counts used correspond to the timings.

This MFlop rate metric is not the actual MFlop rate on any of the given platforms. But the ratings are consistent between problem sizes and across platforms and are easier to use for comparisons than the raw timings.

We used the standard benchmark problem for the shallow water equations, global steady-state nonlinear zonal geostrophic flow [13], and three problem size classes: T42, T85, and T170, characterized by the following computational grids and complexity.

	physical grid	Fourier grid	spectral coefficients	flops per timestep
T42	64×128	64×64	946	4129859
T85	128×256	128×128	3741	24235477
T170	256×512	256×256	14706	153014243

The problem size also has a vertical component. For example, T42L16 is a T42 horizontal grid with 16 vertical levels. The complexity of solving the problem is linear in the number of vertical levels.

In all experiments, a 64-bit precision floating point computation was used. Timings were taken for 241 and 481 steps for the T42 problem sizes and for 49 and 97 steps for the T85 and T170 problem sizes.

The plots in Fig. 1 describe the best results for each platform, using math library routines when available. Note that the x-axis does not use a uniform coordinate system. Table 4 compares performance with and without math routines for those platforms for which this was an option.

Table 4. Serial MFlop/second rates with (m) and without math libraries

	T42L1	T42L16	T85L1	T85L16	T170L1
Paragon	8.9	8.9	9.1		
Paragon (m)	13.9	13.9	13.1		
T3E-900	95.2	78.6	100.2	81.2	107.6
T3E-900 (m)	112.3	89.8	113.8	90.9	118.0
SPP-2000	120.0	82.0	103.3	82.7	126.6
SPP-2000 (m)	148.8	89.7	123.8	95.1	134.5
Origin/195	127.6	79.4	115.5	88.4	109.0
Origin/195 (m)	153.1	92.4	134.1	104.5	120.5

These performance results are typical of what we have observed for spectral atmospheric models, especially those with a “vector” heritage. The C90 hardware perfor-

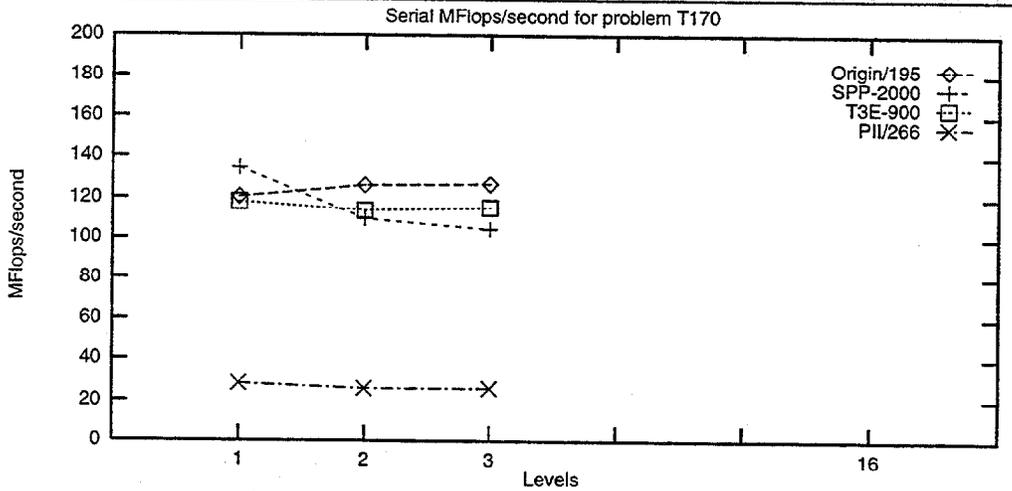
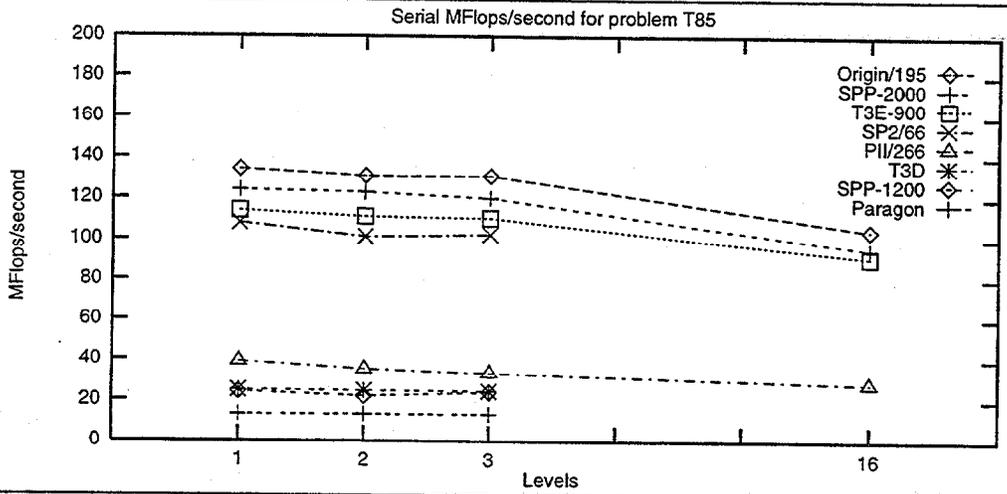
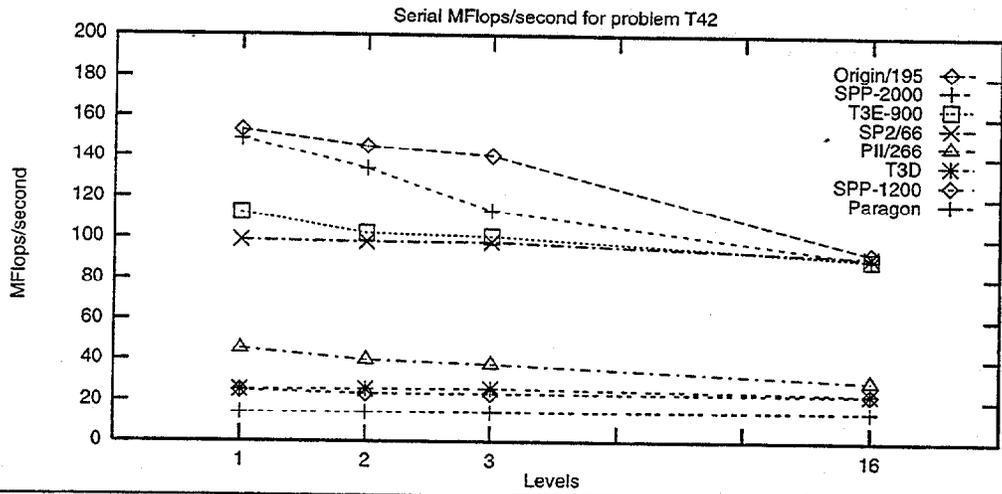


Figure 1. Serial MFlop/second rates.

mance monitor indicated that the ratio of floating point operations to floating point loads is only 1.3, so there is relatively little reuse of operands in the code.

From these data it is clear that the (serial) performance of the processors used in massively parallel processor systems (MPP) has generally improved over the past few years and that optimized math libraries are important performance enhancers. Also note that some effects of the memory hierarchy on performance can be observed from the variation in the MFlop/second rate as the problem size varies. This is important in understanding the performance of the parallel runs. For example, for the transpose-based parallel algorithms, increasing the number of processors typically involves a further decomposition of the vertical dimension, resulting in fewer vertical levels assigned to each processor. In this situation, we would expect the computational rate to increase with increasing numbers of processors.

6. Point-to-Point Communication Performance

Communication overhead is best measured in the context of the full code, but it is useful to establish a performance baseline by determining the peak achievable point-to-point interprocessor communication performance, analogous to the serial computation baseline described previously. To characterize the basic communication capabilities in terms relevant to PSTSWM, we used the PSTSWM SWAP command. Using the SWAP commands adds one or two extra subroutine calls to the overhead of calling the underlying transport layer, and multiple "native" commands may be required to implement the SWAP semantics. Thus the measurements will not necessarily agree with the measurements reported by other researchers, but they should be comparable. More importantly, our measurements correspond exactly to the basic interprocessor communication primitives in PSTSWM and should be consistent and fair across the different platforms.

PSTSWM performance is more sensitive to bandwidth than to latency, and the primary focus of these experiments was on determining the achievable bandwidth when exchanging moderate- to large-size messages. To achieve this, we measured the time required to exchange 262,144 64-bit floating point numbers between two neighboring processors. The experiments varied the packet size/number of messages used to exchange the information and the protocol used for the exchange (using the SWAP

protocols described in Sect. 3). The smallest message sent was 2 KB (256 REAL*8 values), and the largest was 2 MB. In the results that follow, we refer to these as the *2MB* experiments. For completeness sake, we also measured the time to swap 1024 and 16,384 64-bit values, using message sizes ranging from 8 B to 8 KBs and from 128 B to 128 KB, respectively. We refer to these as the *8KB* and *128KB* experiments.

By varying the number of messages and message size, but leaving the total volume fixed, we can easily observe both the constant overhead in sending and receiving messages (looking at differences between measurements) and the achieved bandwidth. If communication performance satisfied a (latency, bandwidth) linear model [3], these parameters would be determined exactly, but the effects of changes in protocol, contention for shared resources, and the memory hierarchy make the observed parameters functions of the message packet size (and the state of the system). Spot estimates of the parameters are still of interest, and the minimum timing can be used to calculate the maximum achieved bandwidth for a given experiment.

These experiments also have the practical advantage of not requiring the measurement of either very small or very large execution times. For the larger message counts, the experiments are similar to the typical measurements of communication cost using repeated sends and receives. Note however that they move linearly through memory, with each send and receive accessing unique memory locations. For the smaller message counts, the larger overhead of the first call (instruction cache miss) becomes more noticeable, but the smaller message counts involve the largest messages, so the data movement itself should dominate the cost. Some intrinsic interest also exists in determining whether it is better to explicitly send a large message in packets, or whether the message-passing transport layer takes care of such optimization issues automatically.

Table 5 contains the maximum observed bandwidth and typical SWAP overhead (“latency”) for the corresponding communication protocol. (Note that this protocol does not necessarily have the smallest latency.) Figure 2 is a graphical representation of the bandwidth data. Figure 3 shows details for the *2MB* experiments for the optimal and default protocols for the Origin and T3E platforms.

The following general observations on communication performance on the T3E and the Origin can be drawn from the summary data:

- The T3E and the Origin demonstrate significant performance improvement over

Table 5. Peak observed bandwidth (megabytes/second) and latency (microseconds) for optimal protocol

	2MB			Unordered 128KB			8KB		
	BW	lat.	prot.	BW	lat.	prot.	BW	lat.	prot.
Paragon									
: MPI	73	82	(0,6)	70	136	(0,3)	48	139	(0,3)
: NX	76	32	(0,3)	71	83	(0,3)	57	74	(0,3)
: SUNMOS	293	63	(0,3)	—	—	—	—	—	—
T3D									
: SHMEM	183	19	(0,2)	—	—	—	—	—	—
SP2									
: MPI	96	136	(0,4)	—	—	—	—	—	—
SPP-1200									
: MPI	45	104	(0,4)	—	—	—	—	—	—
T3E-900									
: MPI	286	30	(0,2)	245	24	(0,0)	66	25	(0,2)
: SHMEM	543	7	(0,1)	494	7	(0,1)	258	7	(0,1)
SPP-2000									
: MPI	654	39	(0,6)	629	15	(0,6)	145	23	(0,2)
Origin/195									
: MPI	142	39	(0,1)	128	29	(0,6)	57	30	(0,1)
: SHMEM	287	15	(0,1)	222	14	(0,1)	114	12	(0,2)
	2MB			Ordered 128KB			8KB		
	BW	lat.	prot.	BW	lat.	prot.	BW	lat.	prot.
Paragon									
: MPI	118	75	(1,0)	107	105	(1,3)	52	82	(1,0)
: NX	118	50	(1,0)	114	62	(1,0)	75	52	(1,0)
: SUNMOS	154	35	(1,0)	—	—	—	—	—	—
T3D									
: SHMEM	126	12	(1,2)	—	—	—	—	—	—
SP2									
: MPI	71	74	(1,1)	—	—	—	—	—	—
SPP-1200									
: MPI	29	27	(1,3)	—	—	—	—	—	—
T3E-900									
: MPI	163	29	(1,6)	134	20	(1,0)	47	21	(1,0)
: SHMEM	336	9	(1,2)	340	5	(1,2)	210	5	(1,2)
SPP-2000									
: MPI	541	20	(1,0)	547	9	(1,0)	158	5	(1,0)
Origin/195									
: MPI	126	33	(1,5)	98	17	(1,3)	40	17	(1,0)
: SHMEM	166	8	(1,1)	140	8	(1,1)	71	10	(1,2)

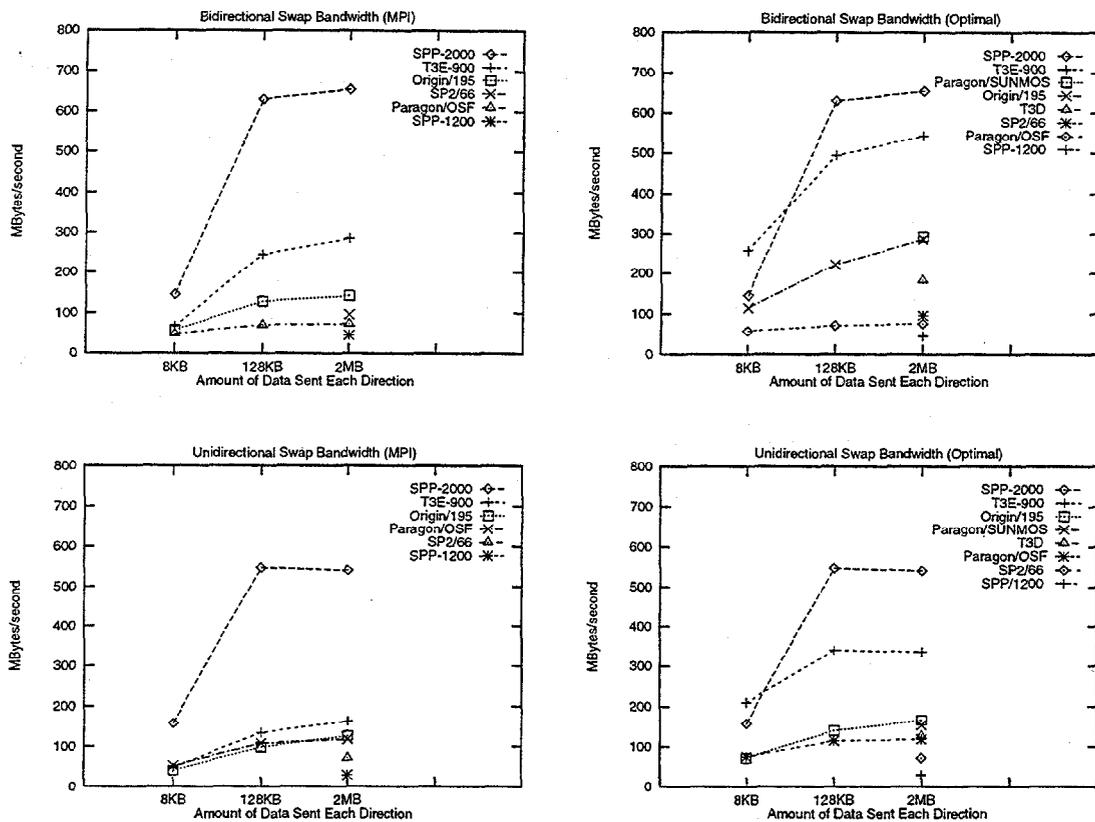


Figure 2. Peak observed bandwidth (megabytes/second).

previous generation MPPs of like architecture. (Note that the SPP-2000 performance is better than both, for these particular tests.)

- SHMEM achieves considerably higher bandwidth and lower latency than MPI, but MPI performance is still an improvement over what was achievable on earlier systems.
- Significant bidirectional bandwidth is possible on the T3E using either MPI or SHMEM. On the Origin, MPI does not support bidirectional communication, although SHMEM does.
- Looking just at the ratio of serial computation and communication rates described here and in the previous section, the Origin is nearly as sensitive to communication costs as the most sensitive of the previous generation machines, the SP2/66.

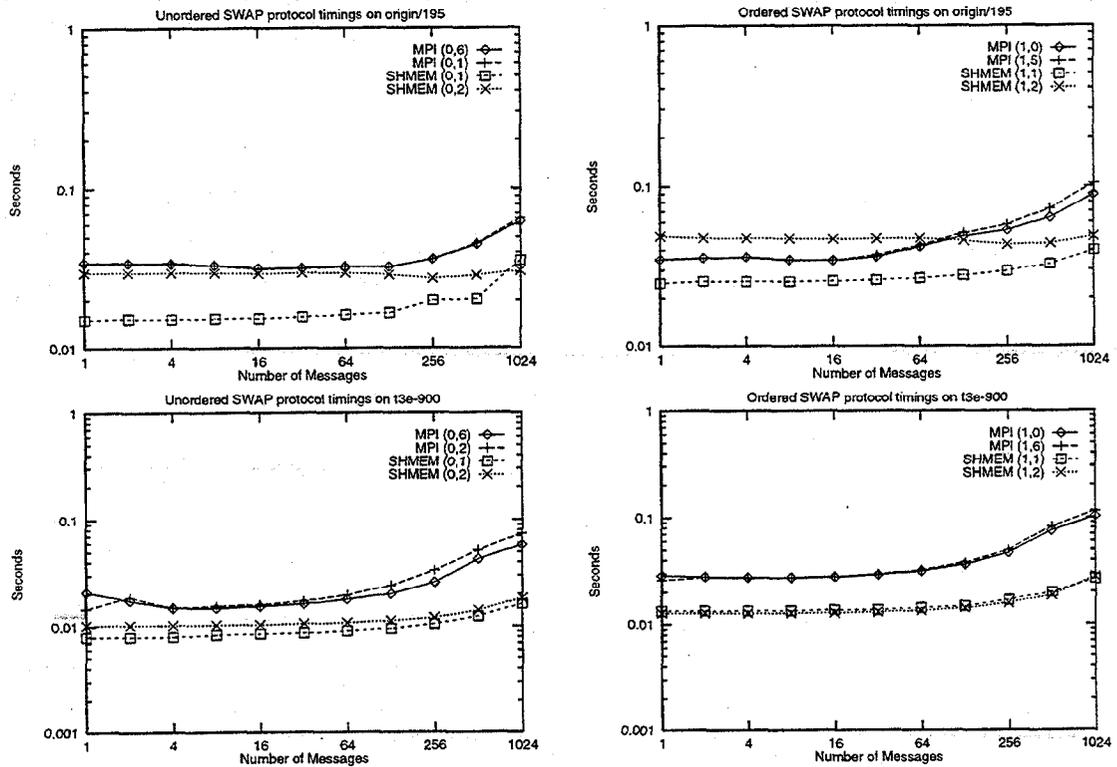


Figure 3. 2MB SWAP experiments.

By the same metric, the T3E is more sensitive to communication cost than its predecessor, the T3D, but not by a large margin.

Space considerations prevent us from presenting the detailed timing data for all of the different communication protocols, but from these data we conclude the following.

- Latency hiding techniques are not effective on the T3E.
- Latency hiding techniques are effective on the Origin when using SHMEM but not when using MPI.
- The best performance using MPLSENDRECV is near the MPI optimal for both the Origin and T3E. On the T3E this is somewhat misleading as the optimal MPLSENDRECV performance requires that the message be manually broken into packets.

On the T3E, the achievable bandwidth shows little sensitivity to the communication protocol when using MPI, and generally the simple protocols are slightly better. On the Origin, MPI performance is somewhat more sensitive to the communication protocol but the communication protocol is still not too important. This is a significant difference from earlier results on the Paragon and the SP2, but is similar to the T3D results, and appears to reflect the SGI/CR implementation of MPI. When using SHMEM, the variability is higher (for both systems), but the `get`-based protocols are optimal or near optimal.

7. Parallel Algorithm Sensitivities

Some indication of the impact of communication protocol on performance can be seen from the point-to-point communication tests, but it is difficult to use these results to predict the effect on application code performance. Here we examine this issue in more detail, looking at the effect on the performance of specific parallel algorithm options in PSTSWM.

As described previously, PSTSWM parallel performance is primarily a function of the performance of the parallel FFT and of the parallel LT, and these have both distributed and transpose-based implementations. The transpose-based algorithms use communication within either rows (for the FFT) or columns (for the LT) of the processor grid to redo the domain decomposition and allow the use of serial transforms.

Three communication algorithms for the transpose were examined, each of which is functionally equivalent to `MPI_ALLTOALLV`:

- `srtrans`— sends $P-1$ messages using `SENDRECV` to transpose across P processors;
- `swtrans`— sends $P-1$ messages using `SWAP` to transpose across P processors;
and
- `logtrans`— sends $\Theta(\log P)$ messages using `SWAP` to transpose across P processors.

The algorithms `srtrans`, `swtrans`, and `logtrans` each use different orderings of interprocessor communication between processors, and `logtrans` sends more data than

the other two. For this paper, we restrict our discussion of the communication algorithms to their use in transpose-based parallel FFTs. However, the performance of the algorithms is not dependent on whether they are used for the FFT or LT except in the amount of data being moved, and results for the parallel LT were similar. One distributed FFT was also examined:

- **dfft**— sends $\Theta(\log P)$ messages using SWAP to calculate Fourier transform distributed across P processors.

The distributed LT algorithms in PSTSWM are based on the evaluation of distributed vector sums (within processor columns). Four distributed vector sum algorithms were examined:

- **exchsum**— an exchange-based algorithm implemented using SWAP;
- **halfsum**— a recursive halving-based algorithm implemented using SWAP;
- **ringsum**— a ring-based algorithm implemented using SENDRECV; and
- **ringpipe**— a pipeline-based algorithm implemented using SENDRECV.

The first three algorithms are functionally equivalent to `MPLALLREDUCE`. The algorithm **ringpipe** interleaves communication with the computation of the LT. **ringpipe** also employs a different decomposition of the spectral domain than the other distributed LT algorithms, resulting in a slightly smaller parallel complexity.

Each of these eight algorithms can be implemented using the protocols described in Table 1 in two different ways. The first uses the basic SWAP and SENDRECV commands to exchange the data. The second reorders the elements of the SWAP or SENDRECV protocol in an attempt to overlap communication with computation and to hide communication latency. These algorithms and protocols are described in more detail in [16]. The overlap algorithms using unordered and ordered communication protocols will be designated by $(2, x)$ and $(3, x)$, respectively, where $x \in \{0, 1, 2, 3, 4, 5, 6\}$.

To compare the performance of the different implementation options, we ran the following experiments. We used one-dimensional decompositions of the form 8×1 or 1×8 and 32×1 or 1×32 , where the first decomposition in each pair was for examining parallel Fourier transform algorithms and the second was for examining parallel LT

algorithms. Thus the “other” transform was computed serially, and all interprocessor communication was restricted to the parallel algorithm under examination. The problem sizes were based on T42L16 and T85L32 as they would appear on a two-dimensional processor grid of size 8×8 , 16×32 , or 32×16 . This was accomplished by modifying the problem size to achieve the desired granularity (problem size per processor) as described in Table 6 and allowed us to examine the performance for problem granularities that are typical of what would be seen in practice.

Table 6. Problem sizes used for parallel algorithm studies

Distributed LT Algorithms						
Problem Identifier		Simulated Problem		Actual Problem		
Problem	P	Problem	P	Problem	physical grid	P
T42	8	T42L16	8×8	T42L2	$64 \times 128 \times 2$	8×1
T42	32	T42L16	32×16	T42L1	$64 \times 128 \times 1$	32×1
T85	8	T85L32	8×8	T85L4	$64 \times 128 \times 4$	8×1
T85	32	T85L32	32×16	T85L2	$64 \times 128 \times 2$	32×1

Transpose FFT Algorithms						
Problem Identifier		Simulated Problem		Actual Problem		
Problem	P	Problem	P	Problem	physical grid	P
T42	8	T42L16	8×8	T21L8	$32 \times 64 \times 8$	8×1
T42	32	T42L16	32×16	T10L16	$16 \times 32 \times 16$	32×1
T85	8	T85L32	8×8	T42L16	$64 \times 128 \times 16$	8×1
T85	32	T85L32	32×16	T21L32	$32 \times 64 \times 32$	32×1

Results for the individual parallel algorithms are presented in Tables 7 and 8. In Table 7, the first column shows the overall best SHMEM protocol for each parallel algorithm. Multiple protocols are given when no single protocol is good for all problem sizes and numbers of processors. The other columns indicate how much performance is lost by using the (0,1) protocol instead of the optimal SHMEM protocol. In Table 8, the first column is the overall best MPI protocols for each parallel algorithm. The other columns indicate how much performance is lost by using the (0,6) protocol instead of the optimal MPI protocol and by using the optimal MPI protocol instead of the optimal SHMEM protocol. Note that the run times used in determining performance are for the full code, not just the individual algorithms, and that the indicated performance

loss is a function of both the size of the messages and the communication/computation ratio for a given experiment.

Table 7. Effect of SHMEM protocol on performance of parallel algorithms

T3E-900					
	good SHMEM protocols	$\frac{t_{(0,1),shmem} - t_{opt,shmem}}{t_{opt,shmem}}$			
		$P = 8$		$P = 32$	
		T42	T85	T42	T85
dfft	(0,1),(3,1)	0%	3%	0%	1%
exchsum	(0,1),(2,2)	2%	0%	4%	0%
halfsum	(0,1)	0%	0%	0%	0%
logtrans	(0,1)	0%	0%	0%	1%
ringpipe	(2,2)	8%	10%	2%	17%
ringsum	(0,1)	0%	0%	1%	0%
srtrans	(2,2)	1%	0%	10%	2%
swtrans	(2,2)	0%	1%	1%	1%

Origin/195					
	good SHMEM protocols	$\frac{t_{(0,1),shmem} - t_{opt,shmem}}{t_{opt,shmem}}$			
		$P = 8$		$P = 32$	
		T42	T85	T42	T85
dfft	(0,2),(3,1)	0%	1%	10%	7%
exchsum	(0,2)	5%	2%	0%	2%
halfsum	(0,1)	0%	0%	0%	0%
logtrans	(0,1)	0%	0%	1%	0%
ringpipe	(2,2)	5%	3%	1%	2%
ringsum	(2,1)	1%	0%	5%	2%
srtrans	(0,1),(2,1)	3%	0%	3%	0%
swtrans	(0,1),(2,1)	2%	0%	3%	4%

On the T3E, (0,6) is a good MPI communication protocol for every algorithm except **ringpipe** and **logtrans**, where overlap protocols are sometimes significantly better. It is surprising, however, to see the good performance of the (0,0) protocol, involving as it does an explicit local buffer copy via the `MPLBSEND` command. This may indicate something about how the other variants of `SEND` are implemented on the T3E. The default SHMEM protocol (0,1) is also a good SHMEM protocol for all but a few algorithms, in which case overlap protocols are superior. SHMEM performance is significantly better than that of MPI for all algorithms and problem granularities.

On the Origin, the conclusions are less clear. Although (0,6) is generally a good

Table 8. Effect of MPI protocol on performance of parallel algorithms

T3E-900									
	good MPI protocols	$\frac{t_{(0,6),mpi} - t_{opt,mpi}}{t_{opt,mpi}}$				$\frac{t_{opt,mpi} - t_{opt,shmem}}{t_{opt,shmem}}$			
		P = 8		P = 32		P = 8		P = 32	
		T42	T85	T42	T85	T42	T85	T42	T85
dfft	(0,6)	1%	1%	0%	1%	14%	9%	40%	14%
exchsum	(0,0),(2,2)	1%	2%	1%	0%	9%	6%	32%	15%
halfsum	(0,0),(2,2)	0%	2%	1%	1%	9%	4%	44%	12%
logtrans	(0,0),(2,2)	0%	1%	8%	2%	16%	7%	83%	22%
ringpipe	(2,0)	7%	9%	2%	1%	19%	5%	109%	49%
ringsum	(0,6)	0%	0%	0%	0%	11%	4%	71%	28%
srtrans	(0,6)	2%	0%	0%	0%	18%	5%	148%	31%
swtrans	(0,6)	1%	1%	0%	0%	16%	5%	129%	29%

Origin/195									
	good MPI protocols	$\frac{t_{(0,6),mpi} - t_{opt,mpi}}{t_{opt,mpi}}$				$\frac{t_{opt,mpi} - t_{opt,shmem}}{t_{opt,shmem}}$			
		P = 8		P = 32		P = 8		P = 32	
		T42	T85	T42	T85	T42	T85	T42	T85
dfft	(0,6),(2,3)	1%	22%	0%	9%	8%	11%	3%	-8%
exchsum	(0,1),(0,4)	0%	21%	15%	72%	-1%	2%	-17%	23%
halfsum	(0,1),(0,4)	1%	0%	2%	18%	2%	5%	11%	5%
logtrans	(0,6)	0%	3%	0%	0%	3%	12%	57%	25%
ringpipe	(2,1)	6%	3%	2%	3%	2%	-3%	66%	0%
ringsum	(0,1)	1%	0%	1%	6%	2%	-8%	79%	-9%
srtrans	(0,1)	1%	0%	1%	1%	1%	0%	112%	34%
swtrans	(0,1)	0%	0%	0%	1%	1%	-1%	115%	34%

MPI protocol, when it is bad, it is very bad. There is some indication that the poor performance occurs when large messages are being exchanged. The performance of the default SHMEM protocol is more consistent, with only one case where it should not be used. Unlike on the T3E, MPI is competitive with (or better than) SHMEM in many cases. Although MPI performs much worse than SHMEM for most of the smaller granularity cases, where latency is more important than bandwidth, these experiments are not relevant. The Origin being tested has only 128 processors, so the simulated 512 processor runs do not reflect the granularities of interest. Concentrating solely on the 8-processor experiments, SHMEM still appears to be the better choice, but the bandwidth advantage of SHMEM over MPI that is evident in the SWAP tests in Sect. 6 does not have a noticeable performance impact in these experiments.

8. Parallel Algorithm Comparisons

The three transpose algorithms `logtrans`, `srtrans`, and `swtrans` are functionally equivalent to each other and to `MPI_ALLTOALLV`. Similarly, the three distributed LT algorithms are functionally equivalent to `MPI_ALLREDUCE`. Changing from one algorithm to another within one of these subsets changes only the communication cost, not the computational complexity or the load balance. Thus the performance differences between the algorithms in each subset are also issues of the communication protocol, where the protocol now includes the number, size, and order of messages.

In this section, we use the data collected in the experiments described in Sect. 7 to compare performance within the sets of equivalent algorithms. For each algorithm, we use the optimal communication protocol for a given problem size and number of processors. We include in the comparison the performance of a “generic” algorithm, using a robust protocol that represents what we would choose having no other information about the performance characteristics of the platform. For the transpose FFT, the generic algorithm is `srtrans`, using the (0,6) protocol for MPI and the (0,1) protocol for SHMEM. For the distributed LT, the generic algorithm is `ringsum`, using (0,6) for MPI and (0,1) for SHMEM.

We also include in the comparison the performance of `MPI_ALLTOALLV` and `MPI_ALLREDUCE`. We would hope that the collective commands provided with the communication library would perform better than our hand-coded Fortran implementations, but that was not true in our previous studies [14].

The results are given in Figs. 4 and 5. In each figure, the left column of graphs is for the Origin and the right column is for the T3E. The first row compares the MPI algorithms, including `MPI_ALLTOALLV` or `MPI_ALLREDUCE`. The second row of graphs compares the SHMEM algorithms. The third row of graphs compares the best MPI algorithm, the best SHMEM algorithm, and the implementation using the MPI collective communication routine. The comparisons are in terms of the relative performance degradation from not using the best algorithm in the relevant set: $(t - t_{\text{opt}})/t_{\text{opt}}$. As before, these are timings of the entire code, not just the collective communication routines, using the one-dimensional parallel decompositions and modified problem sizes described in Sect. 7.

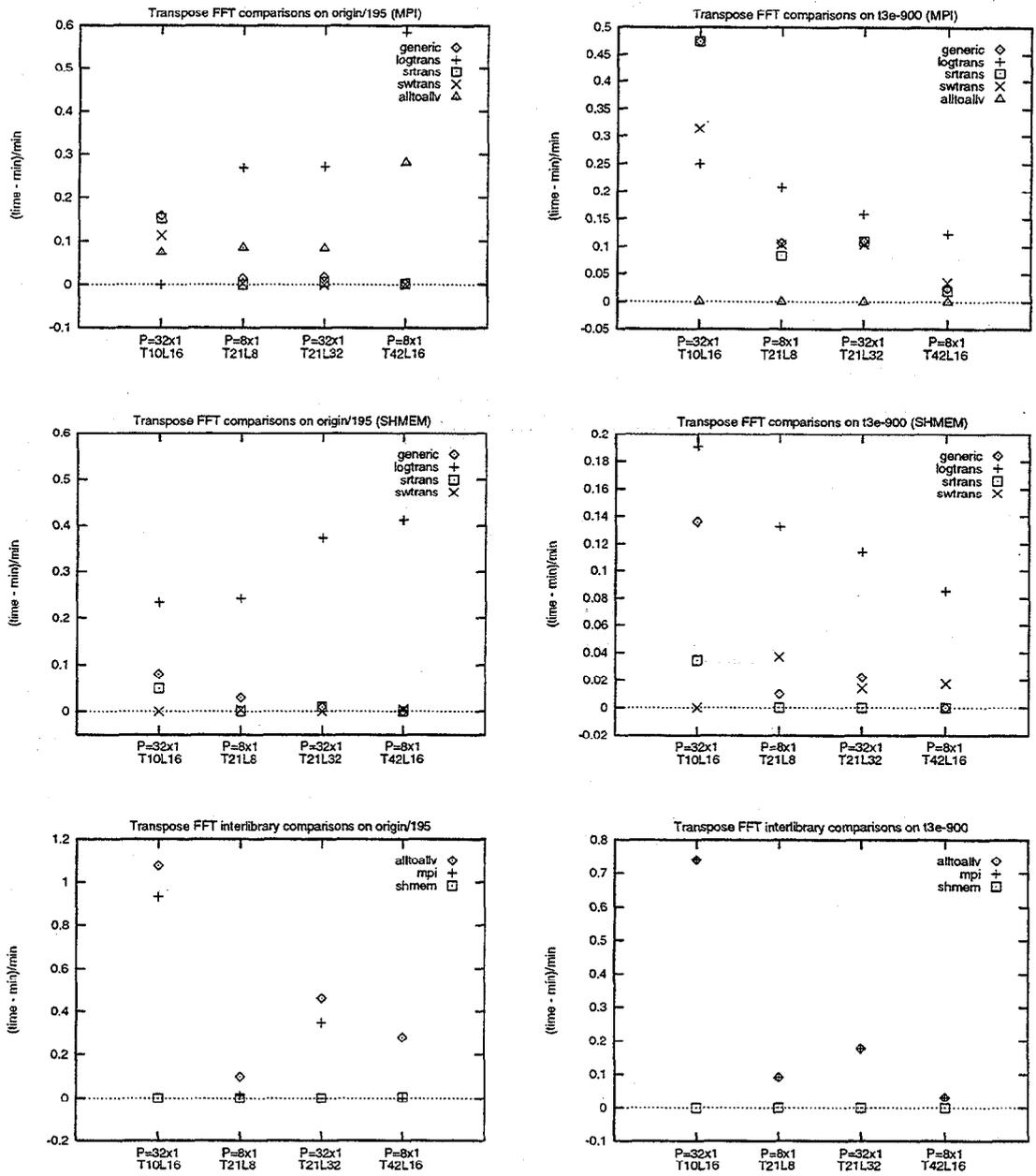


Figure 4. Transpose FFT algorithm comparisons: relative performance degradation as compared to best algorithm.

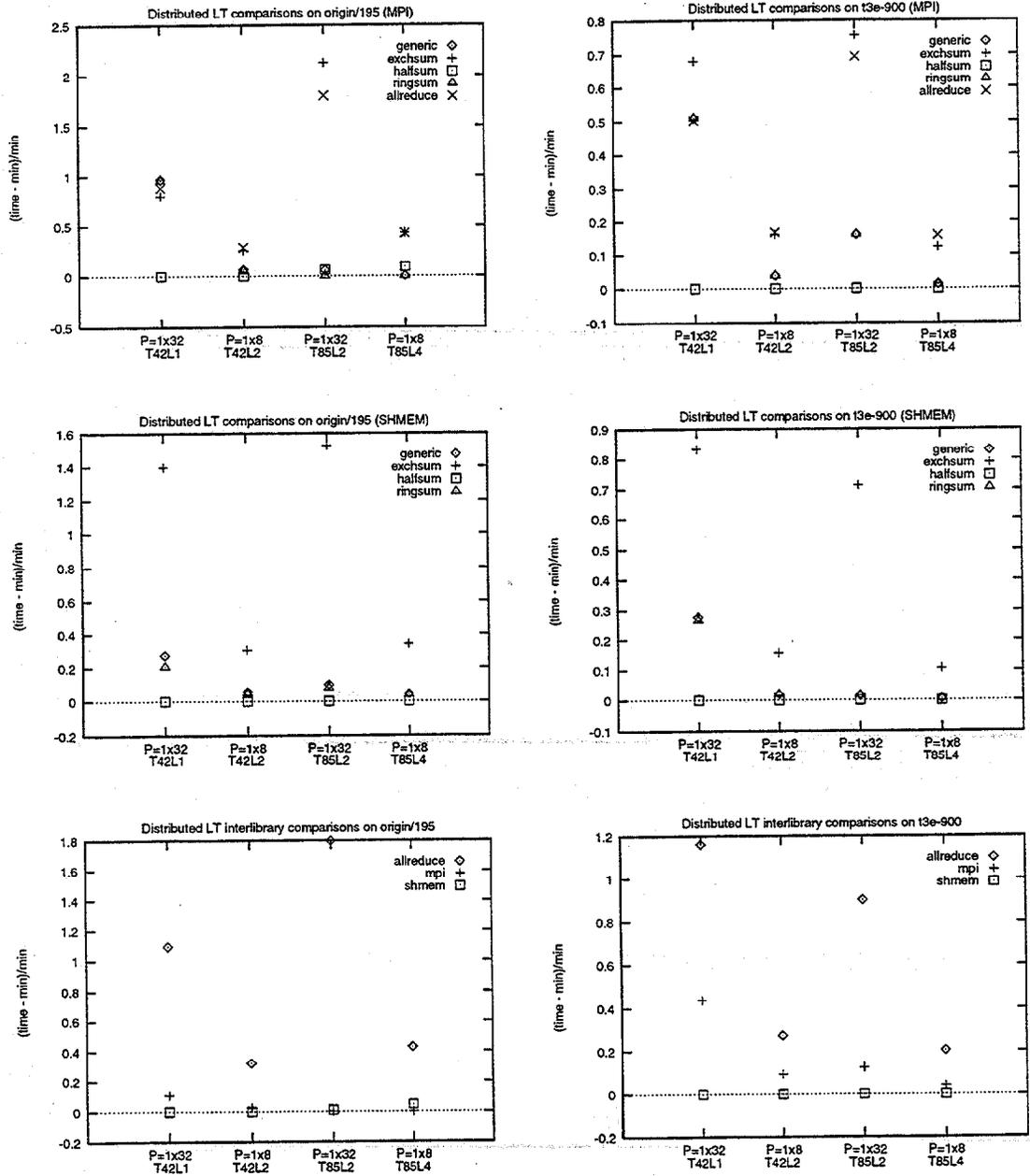


Figure 5. Distributed LT algorithm comparisons: relative performance degradation as compared to best algorithm.

Tranpose FFT Analysis.

- Excluding `MPI_ALLTOALLV`, `srtrans` or `swtrans` is optimal for the large granularity cases, and `logtrans` is optimal for the smallest granularity case in the MPI comparisons on the Origin and on the T3E.
- The `srtrans` or `swtrans` algorithms are optimal for all cases in the SHMEM comparisons on the Origin and on the T3E.
- `MPI_ALLTOALLV` is the optimal MPI implementation for the T3E but it is not competitive on the Origin.
- The optimal SHMEM algorithms are better than the optimal MPI algorithms on the T3E, especially for large numbers of processors.
- The optimal SHMEM algorithms are near optimal in all cases on the Origin, but the best MPI algorithms are competitive in the large granularity cases.
- When compared with the best SHMEM algorithms, `MPI_ALLTOTALLV` performance is never competitive.

In summary, the T3E and the Origin performance comparisons have many similarities. The major differences are in the comparisons between MPI and SHMEM and in the relative performance of `MPI_ALLTOALLV`.

Distributed LT Analysis.

- The `halfsum` algorithm is optimal or near optimal in both the MPI and SHMEM comparisons on the Origin and on the T3E.
- `MPI_ALLREDUCE` performance is very poor compared with other MPI algorithms on the Origin and on the T3E.
- The optimal SHMEM algorithms are consistently better than the optimal MPI algorithms on the T3E.
- The optimal SHMEM algorithms are near optimal in all cases on the Origin, but the best MPI algorithms are competitive in all but the smallest granularity case.

- `MPI_ALLREDUCE` performance is never competitive.

In summary, the Origin and the T3E distributed LT performance results are very similar. The only significant differences are in the comparisons between MPI and SHMEM, and even here the differences are more quantitative than qualitative.

When compared with results on older platforms, the areas of commonality between the Origin and the T3E are also areas of more general agreement. Either `swtrans` or `srtrans` is best for most problem granularities, and both are competitive. The best `MPI_ALLREDUCE`-equivalent distributed LT algorithm is `halfsum`. The two MPI collective commands generally perform poorly. The optimality of `MPI_ALLTOALLV` in the T3E MPI comparisons stands out as the most significant difference.

9. Full Simulation Performance

Efficient parallelizations of PSTSWM exploit two-dimensional decompositions of the domain, parallelizing both the FFTs and LTs [4]. The studies described in Sect. 8 used one-dimensional decompositions that do not capture the performance issues of process placement and network contention that affect two-dimensional decompositions. Although some of these deficiencies can be addressed by modifying the one-dimensional experiments, it is just as convenient to run full two-dimensional simulations at this stage. We made the assumption that only the most competitive of the many protocol options examined earlier needed to be reexamined in this context. (However, we were conservative in this reexamination, retaining 3 to 6 of the most promising protocols for each algorithm.) Note that some elimination was required. The studies described in Sect. 7 and Sect. 8 involved approximately 3500 experiments for each platform and communication library. Retaining all of these options was not feasible for the experiments described subsequently.

We considered two classes of parallel algorithms.

- **DTH**: double transpose for the FFT and `halfsum` for the LT. The double transpose algorithm uses a transpose to serialize the FFTs, then another transpose to return to a domain decomposition analogous to the original. This approach has the best load balance among the parallel algorithm options. The best `MPI_ALLREDUCE`-equivalent algorithm is clearly `halfsum`, being optimal

on both the T3E and the Origin and for both MPI and SHMEM. We do not reexamine this evaluation here.

- **DR: dfft/ringpipe.** This parallel algorithm combination has good load balance and has the maximum potential for communication/computation overlap.

DTH and **DR** stress the underlying transport mechanisms in significantly different ways and represent different tests of the communication protocol sensitivity. Because of their good load balances, the performance differences between them reflect primarily the differences in communication costs.

For each platform we measured the run times when simulating 5 days of the standard benchmark problem for problem sizes T42L16 and T85L16 using

- *SHMEM*— the best transpose algorithms (for **DTH**) and the best communication protocols for each parallel algorithm for SHMEM implementations, determined empirically;
- *MPI*— the best transpose algorithms (for **DTH**) and the best communication protocols for each parallel algorithm for MPI implementations, determined empirically;
- *GEN*— *srtrans* (for **DTH**) and (0,6)-based MPI parallel implementations; and
- *COL*— MPI collective communication routines `MPI_ALLTOALLV` and `MPI_ALLREDUCE` (for **DTH**),

for logical processor meshes of sizes: 4×4 , 4×8 , 8×8 , 8×16 , 16×16 , and 16×32 . Algorithms *GEN* and *COL* represent the typical algorithm choices if nothing is known about the communication protocol sensitivities. Measurements were also taken using 8×14 for **DR** and 14×8 for **DTH** since the 128-processor experiments may not run efficiently on a 128 processor Origin (because of competition with system processes). Note, however, that the ring algorithm used in **DR** is sensitive to load imbalances, and 14 does not divide the chosen problem sizes as well as the powers of two. This is especially clear in the plots of the T3E results.

Results are presented in Figs. 6 and 7. In each figure, the left column of graphs is for the Origin and the right is for the T3E. The first row describes performance for the

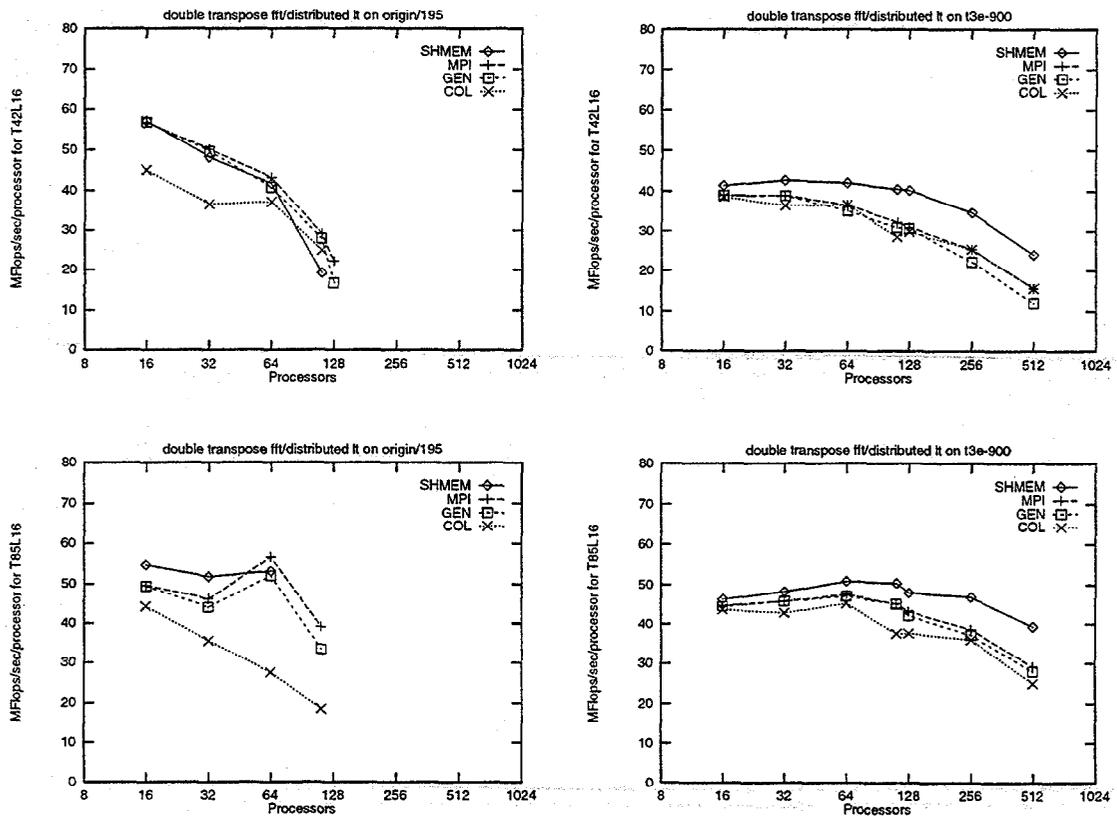


Figure 6. DTH comparisons.

T42L16 problem size and the second row describes performance for the T85L16 problem size. The average Mflop rate per processor is graphed as a function of the number of processors for each problem class and problem size. This allows scaling behavior to be observed easily. Remember that load balance is very good for these parallel algorithms and that the computational rate tends to increase with the number of processors, so performance degradation is primarily caused by increased communication costs.

All of the graphs begin with per-processor rates between 40 and 70 Mflops, indicating significant degradation from the peak serial rates described in Sect. 5. Note, however, that the total volume of data moved in the parallel algorithms is only a weak function of the number of processors and that significant data movement is required even when using small numbers of processors. This is a distinct difference between spectral models like PSTSWM and models having primarily local dependencies (like finite difference models) that have a “surface-to-volume” communication-cost scaling

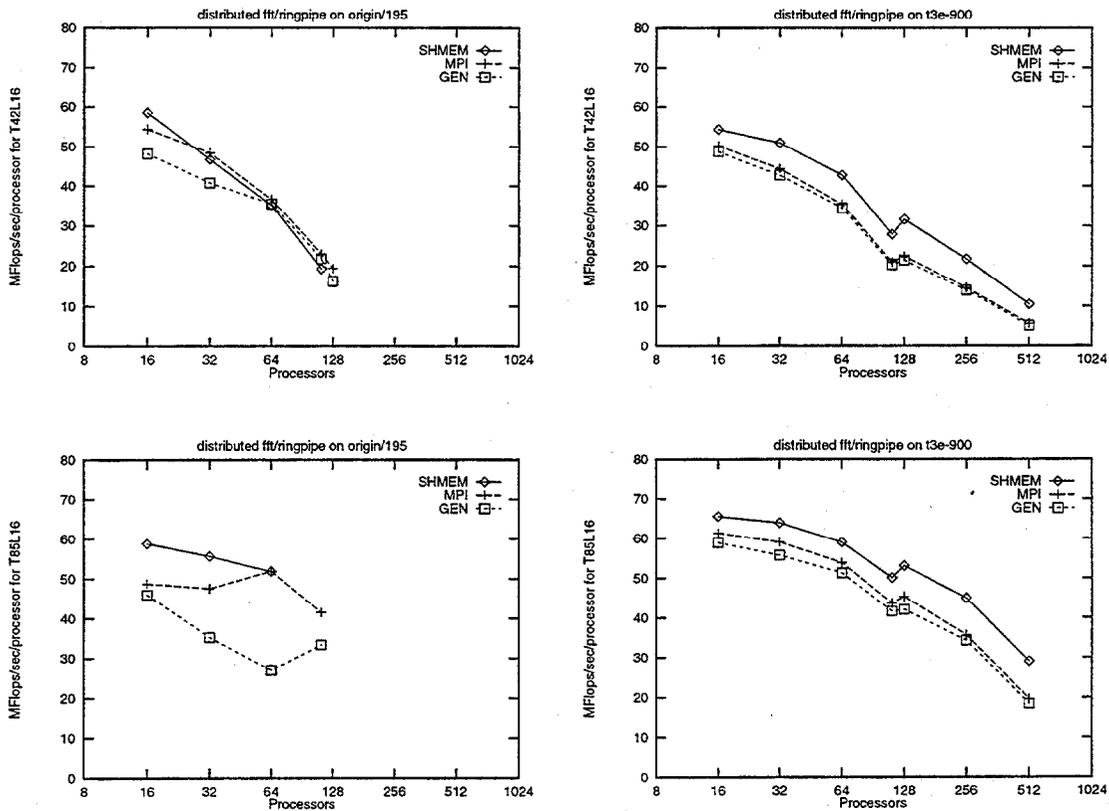


Figure 7. DR comparisons.

behavior.

DTH Analysis.

- SHMEM implementations were significantly better than the MPI-based approaches on the T3E. On the Origin, SHMEM had some advantage over MPI for T85L16 when using small numbers of processors, but the advantage disappeared for smaller problem granularities. SHMEM was also less robust on the Origin and was unable to complete the T85L16 problem when using more than 64 processors.
- On the Origin, the MPI collective communication implementation *COL* performed poorly, especially for T85L16. On the T3E, *COL* performed worse than the other MPI-based implementations except for the very smallest granularity cases and was never competitive with the SHMEM implementations.

- On both the Origin and the T3E, *GEN* performance was reasonably close to that of the optimal MPI implementation.
- Performance was better on the Origin than on the T3E when using no more than 64 processors. However, T3E performance scaled much better than Origin performance for larger numbers of processors.

These results agree well with those of the earlier sections, especially the relative insensitivity of MPI implementations to the choice of protocol and the poor performance of the MPI collective communication routines. One surprise was the strong showing of SHMEM on the Origin for the large granularity cases, indicating better bandwidth, but not elsewhere, demonstrating no practical impact of the lower latency. This analysis is the opposite of that in Sect. 7, and is caused at least partially by a deficiency in the testing methodology. The smallest simulated processor array used in the earlier section was 8×8 , although the bandwidth effect shows up for even smaller numbers of processors. Similarly, the practical effect of lower latency using SHMEM was apparent in the simulated 512-processor runs, which is a much smaller granularity than used in these experiments.

The difference in scaling between the Origin and the T3E is not unexpected. Part of the difficulty for the Origin is that the operating system was still in development for the 128 processor machine at the time of these experiments. However, the T3E was built to allow scalable performance, but Origin's shared memory architecture is more susceptible to contention for bandwidth.

DR Analysis.

- The SHMEM performance behavior was identical to that observed for the **DTH** experiments. Performance was significantly better than the MPI-based approaches on the T3E. On the Origin, the SHMEM advantage was limited to the large granularity cases and it suffered from stability problems.
- On the T3E, *GEN* performance was reasonably close to that of the optimal MPI implementation. On the Origin, performance was somewhat worse for T42L16 and significantly worse for T85L16. Apparently, the message patterns used by

DR are subject to performance problems on the Origin that can be avoided if the communication protocol is chosen appropriately.

- Performance on the T3E was comparable to or better than that on the Origin for small numbers of processors and demonstrated better scaling for all numbers of processors.

One of the major differences between **DR** and **DTH** is the potential for communication/computation overlap in **DR**. This may be one reason that the generic protocol does not perform as well as the MPI optimal on the Origin. As indicated earlier, overlap does not seem to be supported in MPI on the T3E, so *GEN* is essentially equivalent to the optimal MPI implementation there.

An additional difference between the Origin and the T3E is the relative performance of **DR** and **DTH**. Qualitatively, the two algorithm classes perform very similarly on the Origin. However, on the T3D, **DR** performance begins much better but scales much worse and is worse for more than 128 processors.

10. Conclusions

Both the T3E and the Origin 2000 results indicate the importance of considering the interprocessor communication protocols when tuning performance, but the similarity in the results ends there. On the T3E, performance is optimized primarily by using the SHMEM communication library. However, the choice of SHMEM protocol also makes a difference, and overlap techniques can be very effective. Because SHMEM communication is blocking, this simply indicates that the relaxed scheduling constraints of the overlapping logic leads to better performance.

Even the choice of the parallel algorithm can have a significant impact on the T3E, as indicated by the different scaling behaviors of the **DR** and **DTH** parallel algorithms. If an MPI implementation is required, either for portability or for specific MPI functionality, the simple (0,6) or (0,0) protocols perform well. Some tuning may still be required to determine whether overlap is worth exploiting. Note that `MPI_ALLTOALLV` is worth using in an MPI implementation, but that `MPI_ALLREDUCE` should be avoided.

On the Origin, indications are that SHMEM can improve performance for either

very large granularity (improved bandwidth) or very small granularity (lower latency), but neither of these were relevant for the number of processors or problem sizes examined here. Moreover, the SHMEM implementations were not as stable as the MPI implementations. The (0,6) MPI protocol performed well in most cases, but other choices performed better overall and were more robust. In particular, certain conditions saw the performance of the simple protocols degrade seriously, and care must be taken to examine protocol sensitivity using the full codes with the number of processors and problem sizes to be used in production. The two MPI collective commands examined here performed very poorly.

The methodology described here for examining communication protocol sensitivity proved very useful. Although some of the results were initially misleading, because of inappropriate problem granularities, the complete set, from “peak achievable” to full code measurements, allowed us to identify and understand the important issues. We were able to investigate multiple aspects of communication protocol sensitivity without consuming an inordinate amount of resources and have some confidence that we understand many of the reasons behind the observed performance. We also have some understanding as to how performance will change if problem size or numbers of processors are scaled further.

This study concentrated on determining how sensitive performance is to the choice of communication protocol. The results for the Origin and the T3E show that this continues to be an issue but that the particulars are platform specific. Note, however, that this sensitivity is a feature, not a bug, and simply reflects a continued high computation rate/communication rate ratio. The sensitivity would diminish if the vendors used slower processors, which is not an acceptable solution. Given that the sensitivity is a feature, it is a feature that the user needs to be aware of to write codes that perform well.

11. Acknowledgements

This research was supported by the U.S. Department of Energy under Contract DE-AC05-96OR22464 with Lockheed Martin Energy Research Corp. We thank NASA-Ames for access to their SP2 system and Cray Research for access to a T3D system. We thank the Advanced Computing Laboratory at Los Alamos National Laboratory for

access to the SGI/Cray Research Origin 2000. The Intel XP/S 150 MP Paragon operated by the Center for Computational Science at ORNL is funded by the Department of Energy's Mathematical, Information and Computational Sciences Division of the Office of Computational and Technology Research. Access to the CONVEX Exemplar SPP-1200 and the HP/CONVEX Exemplar SPP-2000 was supported by the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign under grant ASC960028N. Access to the SGI/Cray Research T3E-900 at the National Energy Research Scientific Computing Center was supported by the Environmental Sciences Division, U.S. Department of Energy.

REFERENCES

- [1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishan, and S. K. Weeratunga, *The NAS Parallel Benchmarks*, Internat. J. Supercomputer Applications, 5 (1991), pp. 63–73.
- [2] J. D. Dongarra and T. H. Dunigan, *Message-passing performance of various computers*, Concurrency: Practice and Experience, 9 (1997), pp. 915–926.
- [3] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
- [4] I. T. Foster, B. Toonen, and P. H. Worley, *Performance of parallel computers for spectral atmospheric models*, J. Atm. Oceanic Tech, 13 (1996), pp. 1031–1045.
- [5] I. T. Foster and P. H. Worley, *Parallel algorithms for the spectral transform method*, SIAM J. Sci. Comput., 18 (1997), pp. 806–837.
- [6] A. J. G. Hey, *The Genesis distributed-memory benchmarks*, Parallel Computing, 17 (1991), pp. 1275–1283.
- [7] R. Hockney and M. B. (Eds.), *Public international benchmarks for parallel computers, parkbench committee report-1*, Scientific Programming, 3 (1994), pp. 101–146.
- [8] G. R. Luecke and J. J. Coyle, *Comparing the performance of MPI on the Cray T3E-900, the Cray Origin 2000 and the IBM P2SC*, Performance Evaluation and Modelling of Computer Systems, (1998). <http://hpc-journals.ecs.soton.ac.uk/PEMCS/>.
- [9] G. R. Luecke, J. J. Coyle, and W. ul Haque, *Comparing communication performance of MPI on the Cray Research T3E-600 and IBM SP-2*, Performance Evaluation and Modelling of Computer Systems, (1997). <http://hpc-journals.ecs.soton.ac.uk/PEMCS/>.
- [10] MPI Committee, *MPI: a message-passing interface standard*, Internat. J. Supercomputer Applications, 8 (1994), pp. 165–416.

- [11] P. Pierce, *The NX message-passing interface*, *Parallel Computing*, 20 (1994), pp. 463–480.
- [12] A. J. van der Steen, *EuroBen experiences with the SGI Origin 2000 and the Cray T3E*, *Performance Evaluation and Modelling of Computer Systems*, (1998). <http://hpc-journals.ecs.soton.ac.uk/PEMCS/>.
- [13] D. L. Williamson, J. B. Drake, J. J. Hack, R. Jakob, and P. N. Swarztrauber, *A standard test set for numerical approximations to the shallow water equations on the sphere*, Tech. Report ORNL/TM-11773, Oak Ridge National Laboratory, Oak Ridge, TN, 1991.
- [14] P. H. Worley, *MPI performance evaluation and characterization using a compact application benchmark code*, in *Proceedings of the Second MPI Developers Conference and Users' Meeting*, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 170–177.
- [15] P. H. Worley and I. T. Foster, *Parallel Spectral Transform Shallow Water Model: a run time-tunable parallel benchmark code*, in *Proc. Scalable High Performance Computing Conf.*, J. J. Dongarra and D. W. Walker, eds., IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 207–214.
- [16] P. H. Worley and B. Toonen, *A users' guide to PSTSWM*, Tech. Report ORNL/TM-12779, Oak Ridge National Laboratory, Oak Ridge, TN, July 1995.

INTERNAL DISTRIBUTION

- | | |
|------------------|--------------------------------|
| 1. A. S. Bland | 6. C. H. Romine |
| 2. J. B. Drake | 7-11. P. H. Worley |
| 3. T. H. Dunigan | 12. T. Zacharia |
| 4. M. R. Leuze | 13. Central Research Library |
| 5. C. E. Oliver | 14. Laboratory Records-RC |
| | 15-16. Laboratory Records-OSTI |

EXTERNAL DISTRIBUTION

17. David C. Bader, Environmental Sciences Division, SC-74, Department of Energy, 19901 Germantown, Rd., Germantown, MD 20874-1290
18. Patrick A. Crowley, Environmental Sciences Division, SC-74, Department of Energy, 19901 Germantown, Rd., Germantown, MD 20874-1290
19. Jerry W. Elwood, Environmental Sciences Division, SC-74, Department of Energy, 19901 Germantown, Rd., Germantown, MD 20874-1290
20. Dan Hitchcock, Acting Division Director, Division of Mathematical, Information, and Computational Sciences. U. S. Department of Energy, ER-31, 19901 Germantown Road, Germantown, MD 20874-1290
21. Fred Howes, Division of Mathematical, Information, and Computational Sciences. U. S. Department of Energy, ER-31, 19901 Germantown Road, Germantown, MD 20874-1290
22. Tom Kitchens, Division of Mathematical, Information, and Computational Sciences. U. S. Department of Energy, ER-31, 19901 Germantown Road, Germantown, MD 20874-1290
23. Robert Malone, Los Alamos National Laboratory, C-3, Mail Stop B265, Los Alamos, NM 87545
24. Andrew B. White, Los Alamos National Laboratory, P. O. Box 1663, MS-265, Los Alamos, NM 87545

