Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters*

Edmond Chow† and David Hysom‡

Abstract

Computational experiences with hybrid message passing and multithreading techniques on SMP clusters generally show poorer performance than pure message passing approaches. This paper attempts to understand the performance of hybrid MPI and OpenMP programs by decomposing and describing performance using four parameters: multithreading efficiency, relative cache efficiency, network interface efficiency, and message passing scaled efficiency. These parameters are used to assess a sparse matrix-vector product kernel, which is typical of many parallel scientific computations, running on an IBM SP computer. Tests with various problem sizes using up to 216 nodes (864 processors) reveal, for example, the benefit of using a hybrid implementation compared to an MPI implementation when the computation uses small messages and is not network bandwidth limited. Otherwise, the MPI implementation generally shows superior performance.

1 Introduction

Large parallel computers are increasingly built by connecting commodity symmetric multiprocessor (SMP) nodes via a relatively inexpensive interconnect. Each node consists of a number of processors and a large pool of shared memory. Examples of these machines include recent IBM SP parallel computers, Compaq AlphaServer clusters, and advanced Beowulf systems. These machines may be programmed by using message passing between all processors on all nodes involved in a computation, but there is the possibility of achieving better performance by using a hierarchical programming style that matches the hierarchical shared and distributed memory architecture. We refer to a hybrid programming style as one that uses multithreading to use shared memory within a node and message passing to use memory distributed across nodes.

On the surface, hybrid programs should give better performance than pure message passing approaches for three reasons: 1) message passing within a node is replaced by fast shared memory accesses, 2) there is smaller communication volume on the interconnect since intra-node messages are not necessary (this might not affect performance unless the interconnect is

*Tech. Report UCRL-JC-143957, Lawrence Livermore National Laboratory, Livermore, CA, 2001. This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, L-560, Box 808, Livermore, CA 94551 (echow@llnl.gov).

‡Computer Science Department, Old Dominion University, 228 Education Building, Norfolk, VA 23529 (hysom@cs.odu.edu).

congested), and 3) fewer processes are involved in communication, which should lead to better scalability, particularly for global communications. Despite these advantages, however, most hybrid codes do not even achieve the performance of an equivalent message passing code (or a shared memory code), and in many cases, the performance is much worse [8, 9, 21, 12, 16, 19, 15]. Performance may only be comparable, for example, in applications that are very parallel [5].

Depending on the application program or computing environment, various factors have been used to explain poor hybrid program performance, including 1) critical sections that cannot be multithreaded, 2) thread sychronization, loop scheduling, and other overheads, and 3) data partitionings based on SMP nodes rather than threads, leading to poor cache utilization compared to pure message passing programs. The goal of this paper is to help quantify and better understand these and other factors and the performance that can be achieved in hybrid programs.

Some of the above difficulties can be overcome by writing hybrid codes carefully, for example, by using coarse-grained thread parallelism and partitionings based on threads. These hybrid programs, however, can involve significantly more effort to code than message passing programs, e.g., see the overviews in [4, 17, 20, 7]. In reality, many hybrid programs are simply adapted from message passing programs.

At Lawrence Livermore National Laboratory, our early IBM SP system software only allowed slower IP mode communications when it was desired to use more than one MPI task per node, making it imperative to use some form of hybrid programming for better performance. Users combined MPI with either Pthreads, OpenMP, or threaded libraries [17]. The IP mode limitation has now been removed, and the faster US mode may be used. However, US mode limits the maximum number of MPI tasks and this limit is smaller than the number of processors on our largest IBM SP systems. Thus hybrid programming is still imperative in order to solve very large problems using all processors of these large machines.

There are also algorithmic reasons why hybrid programs are important to investigate: 1) computation within a node can be automatically load balanced using multithreading, e.g., [14, 12], and 2) the use of larger subproblems (one per node rather than one per processor) gives many numerical simulation algorithms better mathematical properties [13, 11]. The hybrid and message passing programs are not mathematically equivalent in the latter case.

A number of strategies are available for combining message passing with multithreading. This paper studies perhaps the simplest strategy, that of combining MPI and OpenMP. In this model, typically one process is used per node, and only one thread in the process performs MPI communications at any one time, i.e., in a critical section. Loop iterations that can be parallelized are multithreaded with OpenMP directives.

Besides explicitly combining message passing and multithreading in a program, hybrid programming styles include using versions of message passing libraries that use shared memory transport when communicating within an SMP node [18]. Other alternatives include programming methodologies and libraries [1, 2] and various shared virtual memory environments. These solutions generally trade ease of programming for some loss of performance.

This paper proposes four measurable parameters that help explain the performance of hybrid programs on a given parallel computer. This is done in section 2 after introducing the relevant issues. Section 3 illustrates the use of these parameters for a sparse matrix-vector product kernel (SPMV) running on an IBM SP. This kernel models the behavior of many scientific computations. Some conclusions about the overall performance of SPMV in hybrid mode are drawn in section 4.

2 Hybrid program performance

2.1 Factors affecting hybrid program performance

In this section, we discuss the factors that improve or degrade the performance of hybrid programs compared to equivalent message passing programs. Many of these factors have been suggested in the literature, and others are derived from our own experience. Most of these factors are pointed out in the excellent paper by Cappello and Etiemble [8]. The factors affecting performance in a hybrid code are:

1. critical sections that cannot be multithreaded,
2. thread synchronization, loop scheduling, and other overheads,
3. data partitionings based on SMP nodes rather than threads, leading to poor cache utilization compared to pure message passing programs,
4. memory bandwidth on a node limiting multithreaded performance,
5. differences in communication performance when fewer message passing processes share a network interface,
6. differences in scalability since fewer processes are involved in message passing communication.

The first four factors are related to computation rather than communication, and often degrade hybrid program performance. The last two factors are related to communication and may either improve or degrade performance.

The actual improvement or reduction in performance depends on both application program characteristics and characteristics of the computing environment. Factors 1 to 3 are generally program-specific and can occasionally be avoided or reduced in a carefully written hybrid program. Factors 4 to 6 are less in the programmer's control, but better hybrid programs can be written if these factors are understood.

Two of the factors require more explanation. When fewer processes share a network interface (factor 5), communication latency decreases and per-process bandwidth increases. This may improve hybrid program performance. However, experiments suggest that a single message passing process cannot transfer data fast enough to the network interface to achieve the maximum bandwidth of the network. Thus the *aggregate* bandwidth is lower when fewer message passing processes are used. Thus programs that are bandwidth limited may suffer under a hybrid model.

A potential advantage of hybrid programs over message passing programs is that for the same number of processors in a computation, only a fraction of those processors need to be involved in message passing (factor 6). Particularly when global communications are involved, hybrid programs may be more scalable. Unfortunately, if global communication costs are logarithmic in the number of processors, the advantage of hybrid programs diminishes for larger numbers of processors.

When there is no global communication, programs still lose scalability due to load imbalances or imperfect sychronization between processes when they need to communicate. Again,

these losses may depend on the number of processes participating in communication and may be different for message passing and hybrid programs. Imperfect synchronization is not well-understood. Our results in section 3.5 show a large variation in communication timings, and the variation depends on the number of processes involved in the computation as well as the programming model used.

2.2 Parameters for assessing hybrid program performance

To understand the performance of hybrid programs compared to message passing programs, we propose decomposing and describing performance using four parameters. The first parameter, *multithreading efficiency*, measures the loss in computation rate due to multithreading, namely, the loss due to factors 1, 2, and 4. The second parameter, *relative cache efficiency* identifies cache utilization differences between hybrid and message passing programs, i.e., factor 3. The third and fourth parameters, *network interface efficiency* and *message passing scaled efficiency*, measure the gain or loss in communication performance due to factors 5 and 6, respectively. All the parameters except the fourth parameter are with respect to a small number of nodes, to try to isolate these parameters from scaling effects. These parameters and the ratio of communication-to-computation help determine the relative influence of the various factors on overall hybrid program performance.

In the following, we denote communication time and computation time as t^{comm} and t^{comp}. Subscripts indicate the programming model used: t_{mpi} and t_{hyb} indicate message passing and hybrid models, respectively, and t_{ser} indicates the hybrid model using a single thread per node (or the message passing model using one processor per node). We also denote the number of threads per node as n_t.

2.3 Multithreading efficiency

Many programs do not have perfect speedup when they are multithreaded. This may be due to critical sections in the code, synchronization and other overheads, and memory bandwidth that is not sufficient for the multithreaded computation. *Multithreading efficiency* (e_{mt}) quantifies these factors by comparing the execution time of a threaded program with its execution time if it had perfect speedup. More precisely, for a given program running on a given computer, multithreading efficiency is defined as

$$e_{mt} = \frac{t_{ser}^{comp}}{n_t t_{hyb}^{comp}}.$$

Multithreading efficiency measures the efficiency of the computational part of the program and does not involve communication. Therefore multithreading efficiency can be measured by timing a multithreaded program and its nonthreaded counterpart, each running on a single node.

The multithreading efficiency parameter is affected by cache utilization differences between the multithreaded and nonthreaded versions of the program. Usually, the multithreaded version will have better cache utilization and efficiencies greater than unity are possible. Thus multithreading efficiency must be interpreted with this effect in mind.

Multithreading efficiency may change when the problem size or size of the computation is changed. This may help pinpoint whether or not the inefficiencies are due to fixed-cost overheads.

2.4 Relative cache efficiency

Hybrid and message passing programs may have different cache behavior. Most often, data for parallel programs is partitioned for each process rather than for each thread, and thus message passing programs may have better cache utilization than hybrid programs. If each thread in the hybrid program and each process in the message passing program perform the same work (i.e., the losses due to critical sections and overheads are negligible) then comparing the timings of these two programs' computation phases gives a rough indication of the differences in cache utilization. We define *relative cache efficiency* (e_{cache}) as

$$e_{cache} = \frac{t_{mpi}^{comp}}{t_{hyb}^{comp}}.$$

Here, we assume that the number threads per node in the hybrid program is equal to the number of processors on the node.

2.5 Network interface efficiency

Network interface efficiency (e_{ni}), or NI efficiency, compares the communication performance when different numbers of processes must share the same network interface. In hybrid programs, fewer processes share a network interface than in message passing programs. Both latency and bandwidth (performance for both short and long messages) are affected, and thus NI efficiency depends on message length. We define NI efficiency as

$$e_{ni}(m_2) = \frac{t_{mpi}^{comm}(m_1)}{t_{hyb}^{comm}(m_2)}$$

where m_1 and m_2 are the typical message lengths in the message passing and hybrid programs. The message length in hybrid programs is typically larger than the message length in equivalent message passing programs. To make sure that NI efficiency is not affected by factors related to the number of processors or nodes, it should be based on the communication time of programs using a small number of processors or nodes.

2.6 Message passing scaled efficiency

Message passing scaled efficiency (e_{mp}), or MP efficiency, is the parameter that considers message passing communication performance as a function of the number of nodes or processors used by a program. A pure message passing program using p processors may be less scalable than a hybrid program with p/n_t processes participating in message passing communication. The communication time of the hybrid program cannot be easily predicted from the communication time of the message passing program using p/n_t processors, since the message passing program uses more processors per network interface. Thus NI efficiency is embedded in MP efficiency, and MP efficiency also depends on message length.

Another consideration when many processes are used is the time that processes may need to wait for each other in order to communicate. When there are more processes, it is more likely that communication is not perfectly synchronized, and "imperfect synchronization" is a function

Threads per node	Subdomain size
2	$2n \times n \times n$
4	$2n \times 2n \times n$
8	$2n \times 2n \times 2n$

Table 1: Subdomain sizes depending on the number of threads (processors) per node.

of the number of processes and the programming model. This is a form of load imbalance and causes imperfect scalability even when no global communication is used.

We define MP efficiency as the ratio of the communication time of a hybrid program to that of an equivalent message passing program,

$$e_{mp}(p/n_t) = \frac{t_{hyb}^{comm}(p/n_t)}{t_{mpi}^{comm}(p)}$$

where p is the number of processors used in the computation. The scaled efficiency is measured using the communication timings for a computation with size proportional to the number of processors.

3 Assessing hybrid performance for a model program

The parameters proposed in the previous section help reveal how a program's characteristics and characteristics of the hardware and system software affect that program's performance in hybrid mode. This section describes a model program that attempts to simulate the communication patterns and the communication-to-computation ratio of many scientific computing codes. The above parameters are then applied to help explain the performance of the model program.

3.1 Model program

3.1.1 Data partitioning

Parallel programs often partition a computational space into subdomains, with each subdomain handled by a processor. A communication phase allows the processors to exchange subdomain boundary data, which is then followed by a computation phase. These phases are repeated several times.

The computation that the model program performs is a sparse matrix-vector product (SPMV). This kernel is typical of many scientific computing applications. A simple 3-D computational space is used and partitioned regularly into equal-sized subdomains to reduce effects of load imbalance.

In the hybrid case, Table 1 shows the subdomain sizes that are used, depending on the number of threads (processors) that are used per node. These subdomains are *not* further partitioned for each thread, which is consistent with common practice [17]. In the table, n is used to parameterize the subdomain size (also called the problem size in the tables), and varies from 5 to 40. The subdomains are arranged in a $p \times p \times p$ topology when p^3 nodes are used.

In the pure message passing case, the subdomains (of the hybrid case) are further partitioned into n_t blocks, where n_t is the number of processors used per node. The resulting subdomains have dimensions $n \times n \times n$. These dimensions make the message passing computation comparable to the hybrid computation.

The sparse matrix used for SPMV is from a 27-point discretization of a 3-D partial differential equation. The matrix has at most 27 nonzeros per row, and the arrangement of the nonzeros and the partitioning implies that each processor or node will communicate with at most 26 others.

3.1.2 Hybrid implementation

The structure of SPMV is as follows:

1. Fill buffers for outgoing data
2. Send outgoing data (nonblocking)
3. Receive incoming data (nonblocking)
4. Wait for sends and receives to complete
5. Perform local part of sparse matrix-vector multiply

In the hybrid model, steps 1 and 5 (computation) are loops and are threaded using OpenMP directives, while steps 2 to 4 (communication) comprise a serialized section. There are no other serial sections.

Part of the communication phase may be overlapped with the computation phase. Also, in many scientific applications, the communication phase includes global communication operations. For simplicity, however, these two effects will not be considered in the model program.

3.1.3 Timings collected

We collected timing data for the communication and computation phases of SPMV running in three modes:

SPMV-MPI Traditional message passing model using MPI.

SPMV-Hybrid Hybrid model using MPI with OpenMP.

SPMV-Serial The hybrid model using one thread per node; alternatively, this is the message passing model using one processor per node. This model performs exactly the same communication as SPMV-Hybrid, but the computational work is not multithreaded.

These programs were executed using various numbers of subdomains and subdomain sizes. We varied the subdomain sizes to alter the communication-to-computation ratio and the message sizes in a realistic fashion. Varying the number of subdomains revealed effects due to scalability factors.

Individual communication and computation timings were measured. Communication timings varied significantly from run to run, however, especially in the message passing and hybrid cases (but not in the serial case). These variations are discussed briefly in section 3.5. To assess average performance, we measured the total time for a set of 100 calls to SPMV.

Tests were performed on two IBM SP machines at Lawrence Livermore National Laboratory, one with 244 4-way nodes and another with 16 8-way nodes.

3.2 Multithreading efficiency

Table 2 compares SPMV-Hybrid and SPMV-Serial, showing timings and multithreading efficiencies on one 4-way IBM SP node using various problem sizes. The table shows that SPMV-Hybrid does not have perfect multithreading efficiency. This is the case although the model program avoids several causes of inefficiency: 1) there are no critical sections when SPMV-Hybrid runs on a single node, 2) the multithreading overheads in SPMV-Hybrid are very low relative to the granularity of the thread parallel work (see for example [10, 3, 6] for estimates of OpenMP overheads), and 3) SPMV-Hybrid should have better, rather than worse cache utilization with regularly structured and partitioned problems.

Except for small problem sizes, the multithreading efficiency is approximately constant, showing that the loss in efficiency is proportional to the amount of work done, and is not due to fixed overhead losses (which would cause the efficiency to increase with problem size). The best explanation for the loss in efficiency is that memory bandwidth limits the rate of computation. In support of this, the SPMV-MPI computation timings are very similar to the SPMV-Hybrid timings on a single node (see Table 5 for these timings). Also, as shown in Table 3, the multithreading efficiency is higher when fewer threads per node are used in SPMV-Hybrid.

The tables also check whether or not the execution time is proportional to the problem size by computing the number of rows processed per second by the three programs. The results show that for small problem sizes, the computation is much faster, which may be explained because the matrix fits into cache for these sizes. For moderate sized problems, multithreading efficiency can exceed 1, which may be explained if SPMV-Hybrid is operating in cache and SPMV-Serial is not.

Multithreading efficiencies on an 8-way SMP node with larger cache and higher memory bandwidth are shown in Table 4. The results are similar, showing that multithreading efficiency improves when fewer threads per node are used.

3.3 Relative cache efficiency

Since SPMV-MPI and SPMV-Hybrid perform essentially the same work in their computational phases, we can compare timings of these phases to check differences in cache utilization. Table 5 shows these timings and the relative cache efficiencies for various problem sizes. The results show that except for very small problem sizes, the cache utilization is similar.

3.4 Network interface efficiency

3.4.1 Latency and bandwidth test

Tables 6 and 7 show measured latency and bandwidth for 4-way and 8-way SMP nodes, respectively. Our benchmark program for measuring these parameters is slightly different from standard benchmarks, but more closely matches our application code. Instead of measuring half the roundtrip times for short and long messages (to compute latency and bandwidth, respectively), we measure the time for all processors to send outgoing data (nonblocking) and then receive incoming data (blocking). (The bandwidth we report is based on the number of bytes *sent* per processor and is based on 1000 iterations.) In the test, each processor communicates with one other processor, either off-node, or on-node. The tables show results when different

Prob. Size	Time (s)		Rows per second		e_{mt}
	Hybrid	Serial	Hybrid	Serial	
500	0.015	0.035	33333	14286	0.58
4000	0.216	0.779	18519	5135	0.90
13500	0.825	2.754	16364	4902	0.83
32000	1.997	6.764	16024	4731	0.85
62500	3.877	13.250	16121	4717	0.85
108000	6.888	23.500	15679	4596	0.85
171500	10.795	37.017	15887	4633	0.86
256000	16.228	55.988	15775	4572	0.86

Table 2: Computation timings, rates, and multithreading efficiencies for SPMV on a single 4-way SMP node. Timings are the sums from 100 runs.

Prob. Size	Time (s)		Rows per second		e_{mt}
	Hybrid	Serial	Hybrid	Serial	
250	0.010	0.016	25000	15625	0.80
2000	0.179	0.371	11173	5390	1.04
6750	0.700	1.329	9642	5079	0.95
16000	1.749	3.330	9148	4804	0.95
31250	3.434	6.574	9100	4753	0.96
54000	5.992	11.492	9012	4698	0.96
85750	9.580	18.408	8950	4658	0.96
128000	14.628	28.103	8750	4554	0.96

Table 3: Computation timings, rates, and multithreading efficiencies for SPMV on a single 4-way SMP node, *using 2 threads per node*. Timings are the sums from 100 runs.

Prob. Size	Multithreading efficiency		
	8 thr/node	4 thr/node	2 thr/node
1000	0.39	0.48	0.55
8000	0.92	0.87	0.87
27000	1.18	1.07	1.04
64000	1.07	1.08	1.06
125000	0.96	1.00	1.05
216000	0.89	0.95	1.01
343000	0.86	0.94	0.98
512000	0.85	0.92	0.97

Table 4: Multithreading efficiencies for SPMV on a single 8-way SMP node with respect to the number of threads per node.

Prob. Size	Time (s)		e_{cache}
	MPI	Hybrid	
500	0.008	0.015	0.53
4000	0.200	0.216	0.93
13500	0.781	0.825	0.95
32000	1.889	1.997	0.95
62500	3.781	3.877	0.98
108000	6.632	6.888	0.96
171500	10.627	10.795	0.98
256000	16.079	16.228	0.99

Table 5: Computation timings and relative cache efficiency for SPMV, running on a single 4-way SMP node. Timings are the sums from 100 runs.

	Number of pairs	Overhead (μs)	Bandwidth (Mb/s)	
			Max per proc	Aggregate
off-node	1	43.5	42.9	42.9
	2	71.5	38.2	76.4
	3	98.1	27.6	82.8
	4	125.2	20.8	83.2
on-node	1	76.1	36.8	73.6
	2	136.0	20.7	82.8
on-node (sh.mem)	1	28.2	54.8	109.6
	2	30.3	59.8	239.2

Table 6: Communication parameters for 4-way SMP nodes.

numbers of pairs of processors are used. A single pair in the off-node case is analogous to a hybrid program when a single MPI process is used per node. As anticipated, increasing the number of pairs increases the latency, decreases the bandwidth per processor, and increases the aggregate bandwidth up to the limit supported by the network.

For interest, we also show the measured latency and bandwidth when the communication is purely on-node, with and without using shared memory in MPI. When shared memory is not used, the parameters closely match the parameters in the off-node case (given the same number of processors per node being used). With shared memory, the latency and bandwidth parameters are improved, and seem mostly independent of the number of pairs involved in communication.

Since bandwidth is actually a function of message size, we plot bandwidth for the off-node and on-node cases in Figure 1. Timings were measured for message sizes from 1 to 4194304 bytes at intervals of powers of 2. Figure 1(a) plots the bandwidth per processor when one to four processors on a node are communicating with the same number of processors on another node. The bandwidth per processor is lower when more processors are performing communication in a node. The kinks in the plots are due to a change in communication protocol (a rendezvous communication protocol is used for message sizes above the eager limit) and may also be due to better cache utilization when message sizes are small.

Figure 1(b) plots the bandwidth when communication is within an SMP node. When two or

	Number of pairs	Overhead (μs)	Bandwidth (Mb/s)	
			Max per proc	Aggregate
off-node	1	52.8	130.5	130.5
	2	51.6	123.8	247.6
	4	46.5	84.4	337.6
	8	63.0	43.1	344.8
on-node	1	46.2	128.9	257.8
	2	44.3	88.3	353.2
	4	61.3	43.8	350.4
on-node	1	27.5	175.4	350.8
(sh.mem)	2	28.7	174.3	697.2
	4	30.3	168.8	1350.4

Table 7: Communication parameters for 8-way SMP nodes.

four processors on a node are communicating, the plots are very similar to the case where two or four processors are communicating off-node. Figure 1(b) also plots the bandwidth when shared memory MPI is used. In this case, the bandwidth is much higher. The maximum bandwidth is achieved with a moderate message size rather than the largest message size. This is due to cache effects in the implementation of shared memory MPI [18]. This curve is similar when four processors on the node are communicating via shared memory MPI.

To determine network interface efficiency, we compare the communication timings for various message lengths when 1 MPI process is used per node and when 4 processes are used per node. When 1 MPI process is used, messages must be four times longer in order to compare to the case when 4 processes are used. Figure 2(a) plots these communication timings. The use of a single MPI process is faster for small message sizes, but for message sizes larger than about 5000 bytes, using 4 MPI processes is faster.

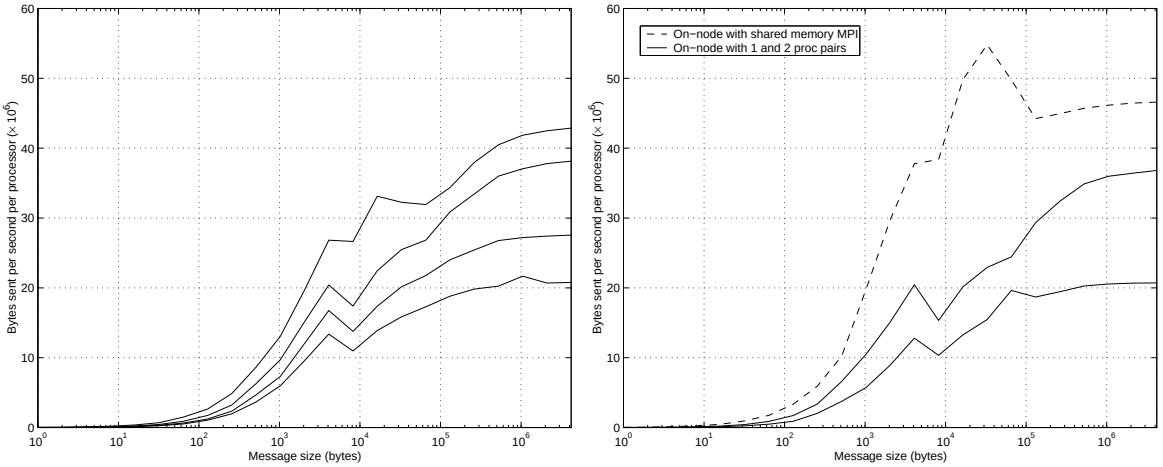
Figure 2(b) plots the network interface efficiency relative to 4 and to 2 MPI tasks per node. (The first curve is the ratio of the two curves in 2(a).) It is also clear from Figure 2(b) that it is not advantageous to use 2 MPI tasks per node to try to balance between latency and bandwidth parameters. Although performance is improved for large message sizes, it is only improved marginally and still does not match the pure MPI program's communication performance.

The results are similar for the 8-way SMP nodes, except that the network interface efficiency is lower. This machine also has a single network interface per node. Table 7 shows that the latency does not deteriorate as severely when additional processors are used per node, however, the single processor bandwidth is a smaller fraction of the maximum aggregate bandwidth.

We also tested the case where multiple threads within the same MPI task make calls to MPI. IBM's thread-safe MPI library was used. However, the latency and bandwidth test showed that per-thread bandwidth was approximately halved when 2 or more threads share the same MPI task this way. We conclude that this is not currently an effective strategy for hybrid codes.

3.4.2 Model program test

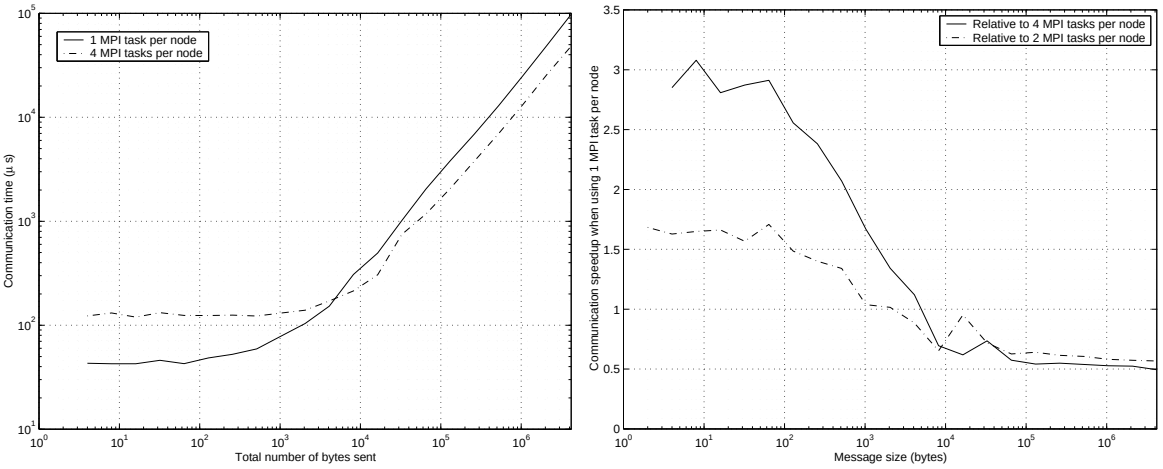
Table 8 shows communication timings for SPMV running on two nodes (8 processors) in various modes. The results show the benefit of hybrid mode in this case. Note that although SPMV-



(a) Off-node communication using 1 (top curve) to 4 (bottom curve) processors per node

(b) On-node communication with 1 and 2 processors per node; the dashed curve shows the shared memory MPI case between 1 pair of processors

Figure 1: Megabytes sent per second per processor, off-node and on-node cases.



(a) Communication time with 1 processor per node (solid curve) and 4 processors per node (dashed curve)

(b) Network interface efficiency: ratio of message passing to hybrid communication timings

Figure 2: Communication time and network interface efficiency.

Prob. Size	Communication time (s)				e_{ni}
	MPI	MPI-sh.mem	Hybrid	Serial	
500	0.041	0.033	0.010	0.020	0.410
4000	0.075	0.057	0.038	0.029	0.197
13500	0.081	0.068	0.067	0.072	0.121
32000	0.140	0.115	0.086	0.094	0.163
62500	0.202	0.126	0.125	0.102	0.162
108000	0.210	0.172	0.150	0.131	0.140
171500	0.315	0.253	0.195	0.163	0.162
256000	0.406	0.326	0.213	0.189	0.191

Table 8: Communication timings for SPMV on two nodes (8 processors) in various modes, and the network interface efficiency. Timings are the sums from 100 runs.

Hybrid and SPMV-Serial perform the same communication, the communication timings are different. This may be due to cache and/or memory bandwidth considerations affecting the communication performance.

The hierarchical partitioning of the computational domain for SPMV-MPI allowed us to test the efficacy of shared memory MPI, i.e., shared memory transport is used for on-node communication. The results, shown in Table 8 as MPI-sh.mem, are not significantly better than the SPMV-MPI timings. This was also verified in tests up to 216 nodes. This result suggests that the off-node communication masked the gains of the on-node communication.

3.5 Message passing scaled efficiency

Figure 3 plots 100 communication timings for SPMV in ascending order for three examples with different numbers of nodes and different problem sizes. The graphs show high variability, e.g., from about 5 ms to almost 300 ms in Figure 3(a) for the SPMV-MPI and SPMV-Hybrid cases. It is believed that these variations are caused by system daemons briefly interrupting the work done on one or more processors.

Given that this variation is not nearly as strong in SPMV-Serial, which places lower demand on the memory system, this variation may be related to memory bandwidth limits. Also, the communication timing variation is larger in the 125 node case than in the 27 node case in the sense that there are more cases when the communication time is extraordinarily large. This suggests that the variation is stronger when there are more nodes.

Despite these variations, a few observations can be made. Communication time for problem size 500 is about 5 times lower than for problem size 256000. In addition, for the smaller problem size, SPMV-Hybrid (and SPMV-Serial) communication timings are better than SPMV-MPI communication timings. This is in agreement with the observation in the previous subsection that not sharing network interfaces is preferred when small messages are used.

In practice, an average or aggregate communication timing must be used to determine the performance of a program that executes SPMV several times. This is used in Table 9, which shows timings for all three implementations for various local problem sizes and numbers of nodes. Since SPMV-Hybrid and SPMV-MPI occasionally have very large communication timings, their average communication time is larger than the average communication time for SPMV-Serial.

Graphs of MP efficiency are plotted in Figure 4 for three problem sizes. For the smallest problem size, 500, the SPMV-Hybrid communication time is always less than the MPI communication time (the curve remains below 1). For the larger problem sizes, 32000 and 256000, hybrid communication is faster for small numbers of nodes and slower for large numbers of nodes.

Finally, we note that programs that perform global communication may give very different scalability results.

4 Conclusions

In this paper, the performance of hybrid programs was decomposed into various factors. For programs where computation dominates communication, the performance of hybrid programs is mostly determined by the size of overheads and critical sections and the memory bandwidth. These factors are captured by the multithreading efficiency parameter. The relative cache efficiency parameter can be used in some cases to check the difference in cache utilization between hybrid and message passing codes. For SPMV computation time, these parameters show that limited memory bandwidth causes a loss of about 15 percent. This loss is the same for both hybrid and message passing codes.

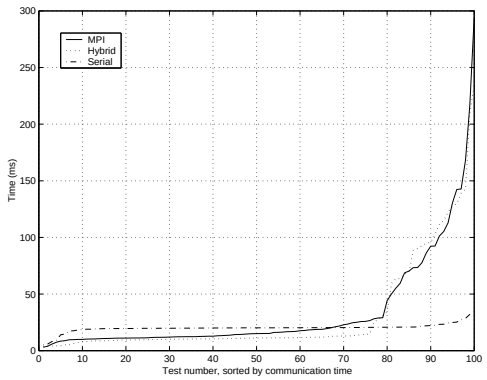
The results in Table 9 show that hybrid programs can perform better than message passing programs when small problem sizes are used. This effect is related to lower latency when network interfaces are not shared, and is quantified by the network interface efficiency parameter.

It is somewhat surprising that SPMV-Hybrid was not better than SPMV-MPI for large numbers of nodes. The MP efficiency parameter shows that SPMV-MPI communication is more scalable than SPMV-Hybrid communication for moderate to large problem sizes. From timings shown in Figure 3, we infer that this is due to occasional extraordinarily large communication timings in SPMV-Hybrid.

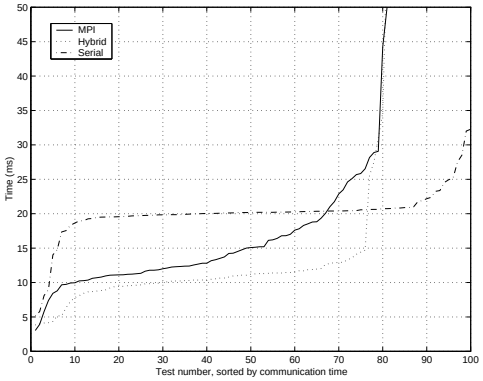
Although hybrid programs are not currently advantageous for computations like SPMV on IBM SP machines, as SMP nodes are built with even more processors and system software is improved, this situation may change. The parameters proposed in this paper can help track progress in this area. Further, the proposed parameters abstract the characteristics of hybrid programs and their computing environments and may lead to models that can predict hybrid program performance.

Acknowledgments

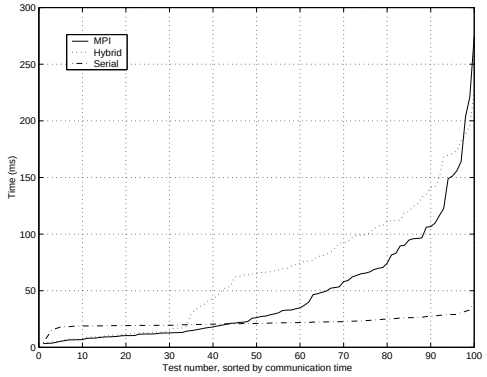
The authors wish to thank Bronis de Supinski, Michael A. Heroux, Leonid Oliker and Jeffrey S. Vetter for helpful discussions during the preparation of this paper.



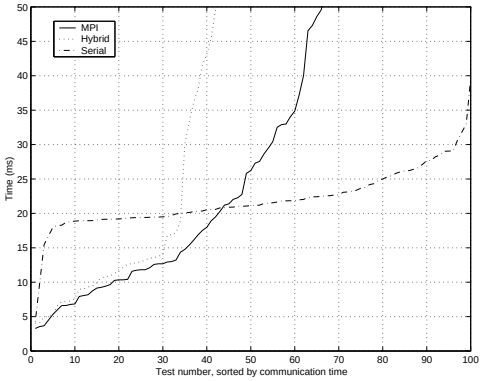
(a) 27 nodes, Prob. size = 256000



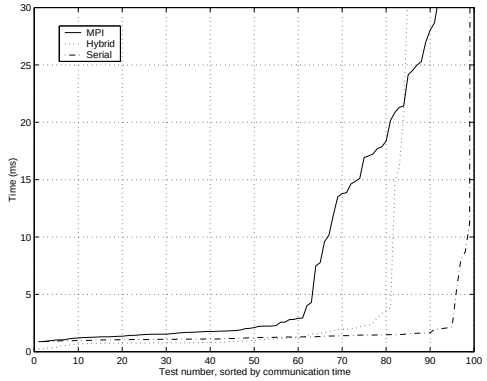
(b) Close-up of (a)



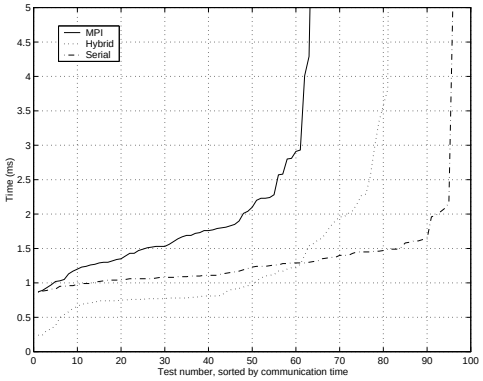
(c) 125 nodes, Prob. size = 256000



(d) Close-up of (c)



(e) 125 nodes, Prob. size = 500



(f) Close-up of (e)

Figure 3: Communication timings in 100 tests, in ascending order.

Prob. Size	Number of Nodes	Total time (s)			Communication time (s)		
		MPI	Hybrid	Serial	MPI	Hybrid	Serial
500	1	0.029	0.015	0.035	0.021	0.000	0.000
	8	0.071	0.042	0.090	0.063	0.027	0.055
	27	0.119	0.101	0.157	0.111	0.086	0.122
	64	0.141	0.126	0.173	0.133	0.111	0.138
	125	0.297	0.148	0.182	0.289	0.133	0.147
	216	0.416	0.182	0.182	0.408	0.167	0.147
4000	1	0.232	0.216	0.779	0.032	0.000	0.000
	8	0.336	0.327	0.892	0.136	0.111	0.113
	27	0.434	0.425	1.021	0.234	0.209	0.242
	64	0.474	0.454	1.031	0.274	0.238	0.252
	125	0.833	0.491	1.035	0.633	0.275	0.256
	216	1.436	0.790	1.044	1.236	0.574	0.265
13500	1	0.838	0.825	2.754	0.057	0.000	0.000
	8	0.988	1.002	3.001	0.207	0.177	0.247
	27	1.216	1.139	3.266	0.435	0.314	0.512
	64	1.338	1.554	3.275	0.557	0.729	0.521
	125	1.349	1.592	3.305	0.568	0.767	0.551
	216	2.769	2.488	3.339	1.988	1.663	0.585
32000	1	1.985	1.997	6.764	0.096	0.000	0.000
	8	2.217	2.305	7.197	0.328	0.308	0.433
	27	2.375	2.737	7.546	0.486	0.740	0.782
	64	2.708	3.299	7.604	0.819	1.302	0.840
	125	3.374	3.686	7.643	1.485	1.689	0.879
	216	4.253	5.286	7.680	2.364	3.289	0.916
62500	1	3.938	3.877	13.250	0.157	0.000	0.000
	8	4.216	4.478	14.011	0.435	0.601	0.761
	27	4.697	5.207	14.511	0.916	1.330	1.261
	64	4.910	5.712	14.652	1.129	1.835	1.402
	125	6.228	7.338	14.646	2.447	3.461	1.396
	216	6.922	8.204	14.732	3.141	4.327	1.482
108000	1	6.818	6.888	23.500	0.186	0.000	0.000
	8	7.351	7.369	24.312	0.719	0.481	0.812
	27	7.896	8.753	25.232	1.264	1.865	1.732
	64	9.183	10.213	25.335	2.551	3.325	1.835
	125	9.885	11.103	25.351	3.253	4.215	1.851
	216	10.004	12.399	25.577	3.372	5.511	2.077
171500	1	10.909	10.795	37.017	0.282	0.000	0.000
	8	11.594	11.748	38.509	0.967	0.953	1.492
	27	12.518	13.451	39.704	1.891	2.656	2.687
	64	13.478	15.344	39.657	2.851	4.549	2.640
	125	14.358	16.293	39.945	3.731	5.498	2.928
	216	15.278	16.614	39.859	4.651	5.819	2.842
256000	1	16.359	16.228	55.988	0.280	0.000	0.000
	8	17.614	17.399	57.850	1.535	1.171	1.862
	27	19.103	19.912	59.499	3.024	3.684	3.511
	64	19.472	21.960	59.773	3.393	5.732	3.785
	125	20.331	22.859	59.837	4.252	6.631	3.849
	216	21.178	24.057	59.982	5.099	7.829	3.994

Table 9: Timings for SPMV. Timings are the sums from 100 runs. For perfect scalability, the timings should be the same for a given problem size.

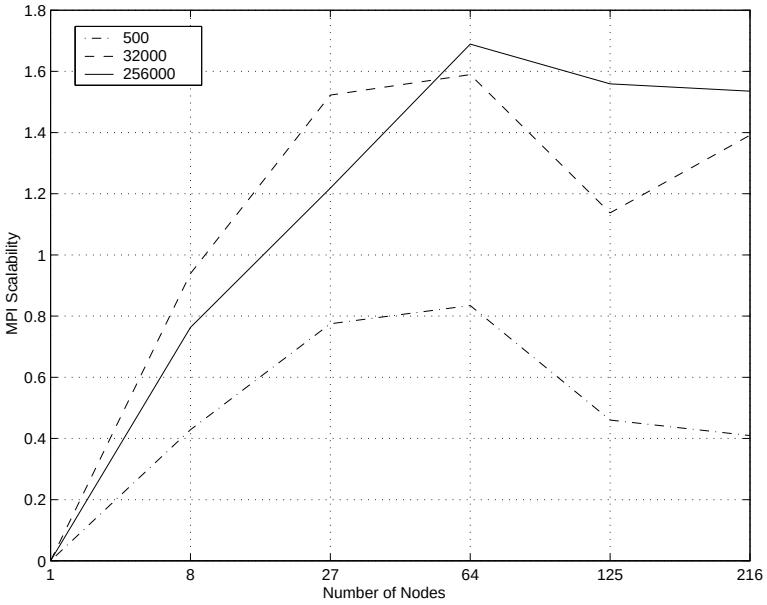


Figure 4: Message passing scaled efficiency: ratio of hybrid to message passing communication timings as a function of the number of nodes. The curves are shown for problem sizes 500, 32000, and 256000.

References

[1] S. B. Baden and S. J. Fink. A programming methodology for dual-tier multicomputers. *IEEE Trans. Softw. Eng.*, to appear.

[2] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). Technical Report UMIACS-TR-97-48, University of Maryland, College Park, 1997.

[3] R. Berrendorf and G. Nieken. Performance characteristics for OpenMP constructs on different parallel computer architectures. In *First European Workshop on OpenMP*, Lund, Sweden, 1999.

[4] S. Bova, C. Breshears, R. Eigenmann, H. Gabb, G. Gaertner, B. Kuhn, B. Magro, S. Salvini, and V. Vatsa. Combining message-passing and directives in parallel applications. *SIAM News*, 32, 1999.

[5] S. W. Bova, C. P. Breshears, C. E. Cuicchi, Z. Demirbilek, and H. A. Gabb. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. *Intl. J. High Perf. Comput. Appl.*, 14:49–64, 2000.

[6] J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *First European Workshop on OpenMP*, Lund, Sweden, 1999.

[7] I. J. Bush, C. J. Noble, and R. J. Allan. Mixed OpenMP and MPI for parallel Fortran applications. In *European Workshop on OpenMP 2000*, Edinburgh, UK, 2000.

[8] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Supercomputing 2000*, 2000.

[9] F. Cappello, O. Richard, and D. Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *IEEE HPCA6*, 2000.

[10] B. de Supinski and J. May. Benchmarking Pthreads performance. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 1999.

[11] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. High-performance parallel CFD. *Parallel Computing*, 27:337–362, 2001.

[12] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing 2000*, 2000.

[13] M. A. Heroux. Exploiting distributed shared memory architectures in sparse matrix computations. In *Solving Irregularly Structured Problems in Parallel, 5th International Symposium*, Berkeley, CA, 1998.

[14] W. Huang and D. Tafti. A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications. In *Proceedings of Parallel Computational Fluid Dynamics*, Williamsburg, VA, 1999.

[15] P. Lanucara and S. Rovida. Conjugate-gradients algorithms: An MPI-OpenMP implementation on distributed shared memory systems. In *First European Workshop on OpenMP*, pages 76–78, Lund, Sweden, 1999.

[16] D. J. Mavriplis. Parallel performance investigations of an unstructured mesh Navier-Stokes solver. Technical Report 2000-13, ICASE, Hampton, VA, 2000.

[17] J. May and B. de Supinski. Experience with mixed MPI/threaded programming models. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 1999.

[18] Boris V. Protopopov and Anthony Skjellum. Shared-memory communication approaches for an MPI message-passing library. *Concurrency: Practice and Experience*, 12:799–820, 2000.

[19] H. Shan, J. P. Singh, L. Oliker, and R. Biswas. Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing*, submitted.

[20] L. Smith. Mixed mode MPI/OpenMP programming. Technical Report Technology Watch 1, UK High-End Computing, EPCC, Edinburgh, UK, 2000.

[21] L. Smith and P. Kent. Development and performance of a mixed OpenMP/MPI quantum Monte Carlo code. *Concurrency: Practice and Experience*, 12:1121–1129, 2000.