ANALYZING THE IMPACT OF OVERLAP, OFFLOAD, AND INDEPENDENT PROGRESS FOR MESSAGE PASSING INTERFACE APPLICATIONS

Ron Brightwell
Rolf Riesen
Keith D. Underwood

Abstract

The overlap of computation and communication has long been considered to be a significant performance benefit for applications. Similarly, the ability of the Message Passing Interface (MPI) to make independent progress (that is, to make progress on outstanding communication operations while not in the MPI library) is also believed to yield performance benefits. Using an intelligent network interface to offload the work required to support overlap and independent progress is thought to be an ideal solution, but the benefits of this approach have not been studied in depth at the application level. This lack of analysis is complicated by the fact that most MPI implementations do not sufficiently support overlap or independent progress. Recent work has demonstrated a quantifiable advantage for an MPI implementation that uses offload to provide overlap and independent progress. The study is conducted on two different platforms with each having two MPI implementations (one with and one without independent progress). Thus, identical network hardware and virtually identical software stacks are used. Furthermore, one platform, ASCI Red, allows further separation of features such as overlap and offload. Thus, this paper extends previous work by further qualifying the source of the performance advantage: offload, overlap, or independent progress.

Key words: Message passing interface, overlap, offload, progress

The International Journal of High Performance Computing Applications,
Volume 19, No. 2, Summer 2005, pp. 103–117
DOI: 10.1177/1094342005054257

1 Introduction

The ability to efficiently overlap communication and computation can be a critical performance requirement for many applications. The emergence of operating system (OS) bypass communication technologies, such as Myrinet (Boden et al. 1995), Quadrics (Petrini et al. 2002), InfiniBand (http://www.infinibandta.org), and Portals (Brightwell et al. 1999, 2002), specifically address the desire to reduce the impact of message passing on the host processor by allowing the network interface adapter to perform communication functions asynchronously, thus achieving low host processor overhead as well as low latency and high bandwidth. However, the network programming interface can limit the ability to leverage overlap in two critical ways.

The first area where many OS-bypass technologies fall short is in the way in which progress is made on outstanding non-blocking Message Passing Interface (MPI) communication requests. The MPI Standard mandates a Progress Rule for completion of asynchronous peer communication operations, but there is disagreement (even among the members of the MPI Forum) as to the exact semantics that this rule mandates. This disagreement has led to implementations that support making progress on outstanding non-blocking calls in different ways. Some implementations adhere to a strict interpretation, where outstanding operations make progress independent of subsequent calls to the MPI library. Others depend on the application to periodically make calls into the library so that outstanding requests can be progressed. This difference in how outstanding communication operations make progress can also have a significant affect on performance.

A second key to fully leverage overlap is the ability to offload MPI matching and protocol processing to an intelligent or programmable network interface. While all OS-bypass technologies support delivering data directly from the network into an application's address space, most networks require the host processor to perform MPI matching and queue traversal functions. Others, such as Quadrics and the co-processor implementation of Portals on ASCI Red, do not. Support for offloading MPI matching and protocol processing on the network interface rather than on the host can significantly impact processor overhead and thus impact performance.

Implementations of MPI typically strive to utilize the capabilities of the underlying network to the fullest and exploit as many features as possible. This research takes the opposite approach. The capabilities of an MPI implementation are purposefully limited to try to better understand how those specific capabilities impact application performance. Rather than simply using identical compute hardware, we use identical networking hardware and a significant portion of the same network software stack. This approach allows us to compare two different MPI

implementations that have nearly identical microbenchmark performance only on the basis of the features (independent progress, overlap, and offload) that differ. This overcomes the primary limitation of numerous previous studies that compare different network hardware, and thus different software stacks, on identical compute hardware.

The rest of this paper is organized as follows. The following section provides background on the features that we have isolated in this experiment. In Section 3 we provide an overview of research that motivated this effort, the platforms studied, and the software approaches used. In Section 4 we then discuss how this work complements other previous and ongoing projects in this area. In Section 5 we present an analysis of the data followed by the conclusions presented in Section 6.

2 Background

In this paper we seek to quantify the impact of independent progress, overlap, and offload from an application perspective. Rather than focusing simply on whether or how a network and its associated protocol stack can provide these features, we wish to quantify their benefit. In the following we discuss the importance of these features in detail.

2.1 PROGRESS

The MPI Standard (MPI Forum 1994) defines a Progress Rule for asynchronous communication operations. Unfortunately, the wording in the Standard is imprecise, which has led to differing views of the how non-blocking communication operations are completed. This ambiguity was addressed by the MPI-2 Forum, but consensus on a clarification could not be reached. The following appears in the MPI-2 Standard:

> [The] issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that [a] send–receive pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes. The MPI forum decided not to decide which interpretation of the standard is the correct one, since the issue is very contentious, and a decision would have much impact on implementors but less impact on users (MPI Forum 1997).

The strict interpretation is that once a non-blocking communication operation has been posted, a matching operation will allow the operation to make progress regardless of whether the application makes further library calls. For example, if rank 0 posts a non-blocking receive and performs an infinite loop (or a significantly long computation) and rank 1 performs a matching blocking send operation, this operation will complete successfully on rank 1 regardless of whether or not rank 0 makes another MPI call. In short, this rule mandates non-local progress semantics for all non-blocking communication operations once they have been enabled.

The weak interpretation allows a compliant implementation to require the application to make library calls in order to make progress on outstanding communication operations. Independent of compliance, it can be easily observed that an implementation that adheres to the strict interpretation offers an opportunity for a performance advantage over one that supports the weak interpretation. This is the quantitative advantage to which the above passage refers. One of the goals of this work is to better understand the impact of independent progress, where an intelligent and/or programmable network interface is responsible for making progress on outstanding communications independent of making MPI library calls.

Independent progress is particularly important for large messages, since they offer the greatest opportunity for overlap of computation and communication. Most implementations of MPI use a rendezvous protocol for exchanging a large message. This protocol involves the sender sending an initial message that describes the message to be sent. For networks that support remote DMA (RDMA) read operations, the receiver can receive this request, use an RDMA read operation to pull the data directly from the sender's buffer, and then send a message to the sender to indicate that the transfer has been completed. For networks that do not support RDMA read operations, the receiver must send back information to the sender describing where the data are to be placed.

Suppose the receiving process posts a non-blocking receive before the send request arrives. When the send request eventually does arrive, all of the information needed to complete the request is available. For some network interfaces, such as Quadrics, the RDMA read request (and subsequent completion message) can be issued without any further involvement from the receiv-

ing process or the host CPU. For implementations that depend on MPI calls to be made, the RDMA read operation cannot be issued until the application calls an MPI library routine. In this case, performance is dependent on how often MPI library calls are made when there are outstanding requests. This interval is independent of the network and is dependent on the structure of the application and, to some extent, on the operating system.

2.2 OVERLAP

The benefit of overlapping communication with computation is well understood in parallel computing. Indeed, message passing interfaces such as MPI have used the concept of asynchronous communication operations for the performance opportunity that they offer. (They are also needed to support semantic correctness required for buffering.) The benefit of overlap is that the host processor need not directly be involved in the transfer of data to its final destination, allowing the CPU to be dedicated to computation.

Measuring the overlap potential of a network is straightforward, but measuring the impact of overlap on an application is not. Most microbenchmarks that attempt to quantify overlap are written to measure the degree to which a network can satisfy the potential to completely overlap computation and communication. Few studies have been performed that actually quantify how much overlap potential exists within an application. For applications that provide little opportunity to overlap, providing support for overlap may actually decrease performance.

It is possible to support overlap without supporting independent MPI progress. Networks capable of performing RDMA read and write operations can fully overlap communication with computation. However, the target address of these operations must be known. If the transfer of the target address depends on the user making an MPI library call (after the initial operation has begun), then progress is not independent. If the transfer of the target address is handled directly by the network interface, or by a user-level thread, then independent progress can be made. Conversely, it is possible to have independent progress without overlap. An example of this is the implementation of MPI for ASCI Red (Brightwell and Shuler 1996), where the interrupt-driven nature of the network interface ensures that progress is made, but the host processor is dedicated to moving data to the network (at least, in the default mode of operation where the communication co-processor is not used).

2.3 OFFLOAD

The third network feature of interest is offload. Offload involves moving functions of an MPI implementation onto the network interface. Most commonly, these include the send semantics (to enable progress on a nonblocking send without host processor involvement) and receive-side matching semantics, so that a message can be delivered to the correct user buffer without host processor involvement. An implementation that offloads enables both overlap and independent progress to occur with minimal (or no) host processor overhead. As previous studies have indicated that overhead is a key performance limiting factor (Martin et al. 1997), this can be a key network feature.

3 Motivation and Approach

Previous work (Brightwell and Underwood 2004a) has indicated that a number of the NAS parallel benchmarks require excessively long posted receive and unexpected message queues in the MPI library. We have hypothesized that this could degrade performance on platforms that offload the traversal of these queues onto much slower network interface hardware. The Quadrics environment provides the ideal opportunity to evaluate this hypothesis as it offers a native MPI implementation, using a programming interface called Tports, which offloads the traversal of these queues onto a 100-MHz processor (in the case of the ELAN-3 hardware). Quadrics also offers a Cray SHMEM (Cray Research 1994) compatibility interface, which can be used as a lightweight layer for building an MPI library without offloading any of the traditional MPI queue management onto the NIC (Brightwell 2004). Microbenchmarks (Underwood and Brightwell 2004) indicate that, indeed, message latency is dramatically smaller on an MPI built over SHMEM when queue lengths grow. This work seeks to determine if application performance correlates to that finding.

A second motivation for comparing MPI libraries built on SHMEM and Tports is that it allows the comparison of a library that provides capabilities for computation and communication overlap as well as independent progress in MPI to a library that does not. The MPI implementation using Quadrics Tports provides independent progress and overlap, since a thread running on the NIC is able to respond to incoming MPI requests without host processor intervention. In contrast, the SHMEM interface still allows messages to be deposited directly into user memory without intervention by the host, but a host CPU must still be used to handle MPI queue management and protocol messages. Using the host processor removes the opportunity for significant overlap and does not allow for independent progress of outstanding communication requests. Thus, a single platform (compute and network hardware) with a similar network software stack can be used to evaluate both approaches.

Overlap and independent progress are believed to be important performance enhancements, but it is seldom possible to evaluate the relative merits of each on identical hardware. By reducing the differences in the system to the MPI implementation, this work is able to evaluate these more independently. This has important implications for newer parallel computer networks, such as InfiniBand (http://www.infinibandta.org), which use RDMA for MPI implementations. Most implementations of MPI for InfiniBand (Liu et al. 2003a, 2004; Rehm et al. 2004) do not efficiently support overlap or independent progress (Liu et al. 2003b).

Finding that the detrimental impact of long posted receive queues are significantly outweighed by the differences in network features (Brightwell and Underwood 2004b), this work extends to a second platform (ASCI Red) where the impact of individual network features can be further studied. Furthermore, two applications have been added to the study to investigate whether the trends observed in the NAS parallel benchmarks also occur in real applications.

3.1 PLATFORMS

Results were gathered on two platforms: a Linux cluster with a commodity high performance network and a large-scale massively parallel processing system with a proprietary interconnect. The following describes the hardware and software components of these systems.

The Linux cluster used for our experiments is a 32-node cluster at Los Alamos National Laboratory. Each node in the cluster contains two 1-GHz Intel Itanium-2 processors, 2 GB of main memory, and two Quadrics QsNet (ELAN-3) network interface cards. The nodes were running a patched version of the Linux 2.4.21 kernel. All applications were compiled using Version 7.1 Build 20031106 of the Intel compiler suite.

Compared to a traditional Linux cluster, ASCI Red (Wheat et al. 1996) is a relatively unique system. It is a large-scale supercomputer comprised of more than 4500 dual-processor nodes connected by a high performance custom network fabric. Each compute node has two 333-MHz Pentium II Xeon processors. Each compute node has a network interface, called a CNIC, that resides on the memory bus and allows for low-latency access to all of physical memory on a node. This interface is effectively a DMA engine that can be instructed to transfer the contents of messages into memory in up to 4 KB chunks. The CNIC interface connects each node to a three-dimensional mesh network that provides a 400 MB/s unidirectional wormhole-routed connection between the nodes. The CNIC interface is capable of sustaining the 400MB/s node-to-node transfer rate to and from main memory across the entire machine.

The software environment on ASCI Red is also significantly different from the standard commodity model. The compute nodes run Cougar, a variant of the Puma lightweight kernel that was designed and developed by Sandia and the University of New Mexico for maximizing both message passing throughput and application resource availability (Shuler et al. 1995).

Cougar uses a simple network protocol built around the Portals message passing interface (Shuler et al. 1995). Portals are data structures in an application's address space that determine how the kernel should respond to message passing events. Portals allow the kernel to deliver messages directly from the network to the application's memory.

Cougar is not a traditional symmetric multiprocessing operating system. Instead, it supports four different modes that allow different distributions of application processes on the processors. The following provides an overview of two of these processor modes that are relevant to this paper. More details can be found in Maccabe et al. (1996).

The simplest processor usage mode is to run both the kernel and application process on the system processor. This mode (P0 mode) is commonly referred to as the "heater mode" since the second processor is not used and only generates heat. In this mode, the kernel runs only when responding to network events or in response to a system call from the application process.

In the second mode, message co-processor mode (or P1 mode), the kernel runs on the system processor and the application process runs on the user processor. When the processors are configured in this mode, the kernel runs continuously waiting to process events from external devices or service system call requests from the application process. Because the time to transition from user mode to supervisor mode and back can be significant, this mode offers the advantage of reduced network latency and faster system call response time.

3.2 MPI LIBRARIES

Table 1 summarizes the characteristics of the MPI implementations used in our analysis. On the Quadrics cluster, two versions of MPI software were used: the default MPICH variant from Quadrics using the Tports API (version 1.24-27) and a variant of MPICH 1.2.5 built at Sandia (Brightwell 2004) using the Cray SHMEM API (Cray Research 1994; using version 1.4.12-1 of the QsNet libraries that contained the Cray SHMEM compatibility library). The MPI version built using the SHMEM API performs quite well as seen in Figures 2 and 3; thus, the only appreciable difference in the two libraries is their capabilities. Tports provides MPI with the capability to overlap computation and communication, the ability to

Table 1
MPI implementation characteristics

	ASCI Red				Quadrics	
	Eager		Rendezvous		Tports	SHMEM
	P0	P1	P0	P1		
Progress	✓	✓			✓	
Overlap		✓		✓	✓	✓
Offload		✓		✓	✓	

achieve independent progress, and the ability to offload matching semantics to the network interface. In previous work, it was found that the combination of these factors had a significant impact on the performance of applications (Brightwell and Underwood 2004b).

Unfortunately, using only these two implementations on the Quadrics network makes it quite difficult to distinguish the actual source of this performance improvement. ASCI Red offers a platform where finer distinctions can be made. On ASCI Red, the default MPI (based on MPICH 1.0.12) uses interrupt-driven Portals processing in the Cougar kernel to provide independent progress. MPI only needs to set up the appropriate Portal for a receive operation and a matching put will be placed directly into user memory without application intervention. This is only possible because the sender uses an eager protocol for all messages. Long messages that arrive without a matching posted receive place MPI envelope information into a buffer and discard the data (to be retrieved later when the matching receive is posted). As part of the interrupt-driven processing, the processor is kept busy (in the kernel) for the entire message arrival time because the kernel must continually service the network interface until the entire message has been received.

Using various processor modes, the default MPI library can provide a variety of capabilities to the application. In P0 mode, message reception occurs on the application processor. Thus, the MPI library can provide independent progress, but not overlap or offload. In P1 mode, message reception happens on the message coprocessor, so the MPI library can provide overlap and offload as well as independent progress.

The primary additional requirement for ASCI Red is a baseline for comparison. An implementation of MPI using a rendezvous protocol for long messages was developed as part of previous work (Brightwell and Underwood 2003) for this purpose. When using a rendezvous protocol in P0 mode, MPI is unable to provide overlap, independent progress, or offload. This library can also be used in P1 mode to test a design point without independent progress, but with overlap and offload capabilities.

4 Related Work

The work in this paper spans the areas of performance analysis, network-based protocol offload, and characterizing the behavior of parallel applications. As such, this study is similar to previous work that characterizes the message passing behavior of applications and application benchmarks in an attempt to understand or predict performance. Examples of this type of analysis can be found in Vetter and Mueller (2002) and Wong et al. (1999). Rather than just reporting on observed behavior, our work in this paper pinpoints specific characteristics and their impact on performance. However, we know of no work that characterizes the benefits and drawbacks of offload using identical hardware and nearly identical software stacks.

There is an abundant amount of work that compares different network technology using identical compute hardware in order to evaluate networks as a whole. The most recent and comprehensive study for MPI and commodity high performance networks is Liu et al. (2003b). However, we are interested in characterizing an application's ability to leverage network performance rather than simply evaluating and comparing network performance. Thus, the work we present here attempts to isolate specific properties of a network—host processing versus network interface offload—in order to evaluate the impacts of the different strategies.

It is typical for papers that describe MPI implementations to explore different strategies within the implementation. An example of this is Dimitrov and Skjellum (2000), where polling versus blocking message completion notification is explored. This kind of MPI implementation strategy work explores different methods of using the capabilities offered by a network, but limiting network capabilities in order to characterize specific network functionality has not been explored.

A description of the MPI implementation for Cray SHMEM as well as communication microbenchmark performance can be found in Brightwell (2004).

5 Results

The first step in determining the impact of overlap, offload, and independent progress is to ensure that the MPI implementations being compared are sufficiently similar in performance. In the next section we evaluate MPI implementations that provide various combinations of overlap, offload, and independent progress in terms of microbenchmark characteristics including latency, bandwidth, and collective performance. Following that, the widely studied NAS parallel benchmarks are analyzed on each of the MPI implementations. Finally, two applications are evaluated on a subset of the MPI implementations to examine the impact of these characteristics.

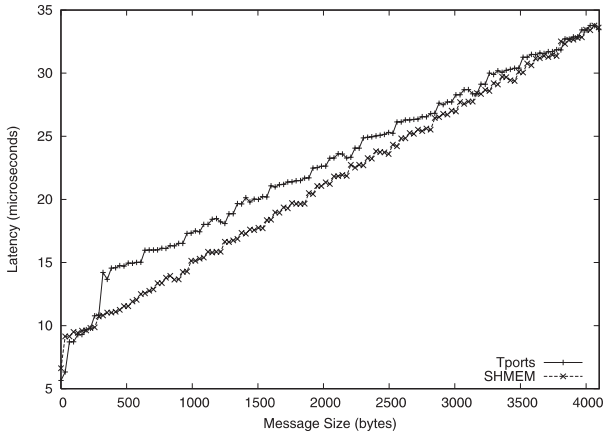

Fig. 1 Message latency for SHMEM and Tports based MPI implementations.

5.1 MICROBENCHMARKS

The first step in comparing MPI implementations built over SHMEM and Tports was to compare their characteristics using microbenchmarks. Figure 1 compares the standard latency metrics for SHMEM MPI and Tports MPI. For small messages (one Elan3 packet), the Tports implementation is slightly faster. As the message size crosses one packet, the Tports implementation becomes slower because it uses a rendezvous protocol on the NIC. The SHMEM implementation uses a larger small message size and does not pay this penalty yet. When message size increases further, the latencies incurred by the two implementations converge as the overheads are amortized over larger data transfers. Thus, in terms of traditional ping-pong latency, the SHMEM MPI and the Tports MPI are roughly equivalent.

For larger message sizes, bandwidth, not latency, becomes the dominant factor. Thus, Figure 2 compares the bandwidth achieved by the two MPI implementations. Figure 2(a) specifically targets medium sized messages and indicates that the two bandwidth curves are virtually identical. Figure 2(b) portrays a similar picture for long message bandwidth. Between Figures 1 and 2, microbenchmarks indicate that the two implementations perform almost identically.

Figure 3 shows the relative performance of the various MPI implementations for ASCI Red on standard microbenchmarks. In almost every case, the implementation that does not provide overlap or independent progress appears to be of approximately equal performance to the one that does. The notable exception is the latency of the rendezvous implementation immediately after it crosses over the short message threshold. The erratic behavior of P1 mode on ASCI Red is still being debugged.

For the FT benchmark, all of the communication time is concentrated in the `MPI_Alltoall` collective. The performance of this collective is shown on the left-hand side of Figures 4(a), (c), and (e) for each of the MPI implementations evaluated. The performance of the collective is very close for the various sizes of the collective for both the Tports and SHMEM implementations of MPI on Elan3. On ASCI Red, P0 and P1 modes perform very similarly, but the rendezvous implementation (with no independent progress) has a significant advantage

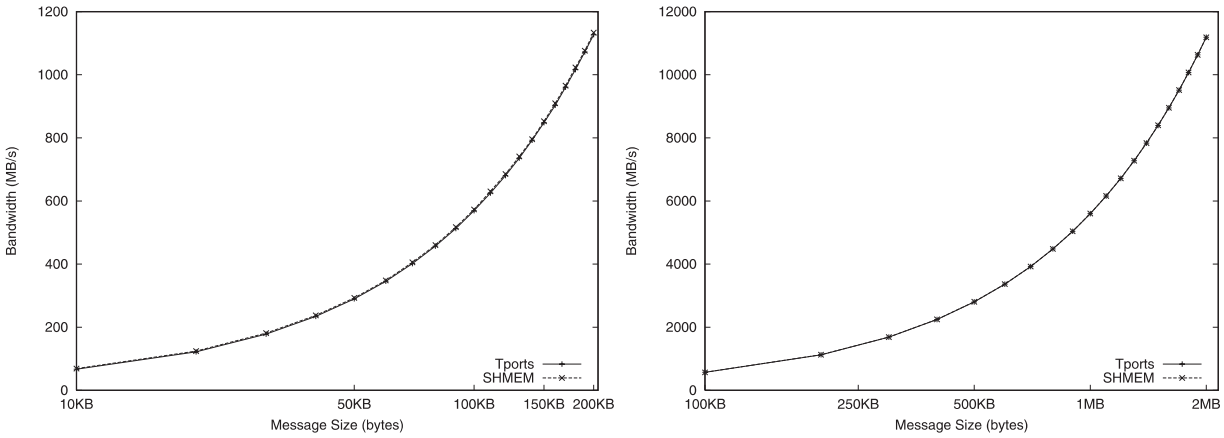

Fig. 2 A comparison of (a) medium message bandwidth, and (b) long message bandwidth for MPI implementations on Elan3.

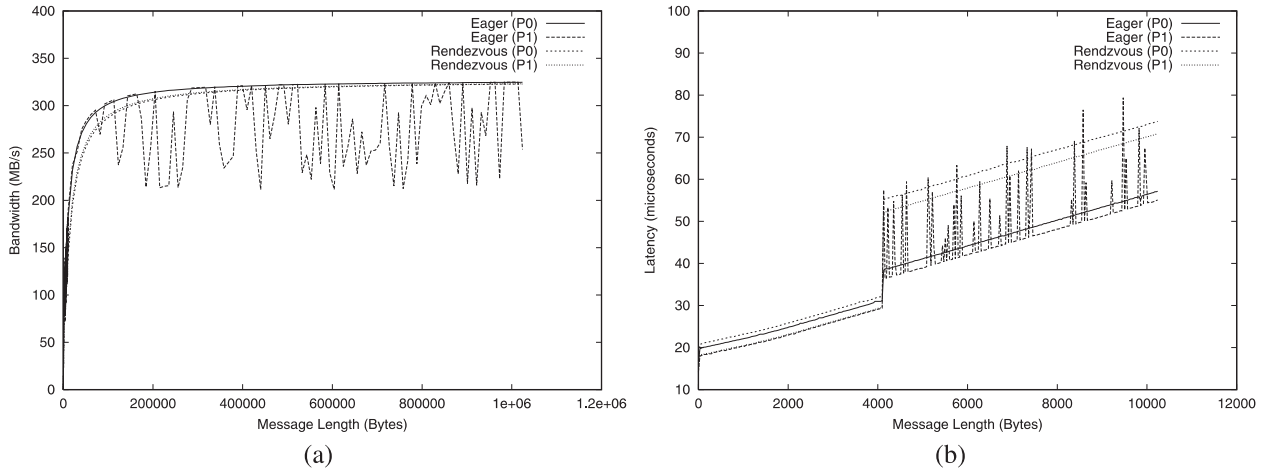

Fig. 3 A comparison of (a) bandwidth and (b) latency for the two implementations on ASCI Red.

over the eager implementation (with independent progress). This is a logical result from previous work that indicated that the naive MPICH implementation of this collective caused a large number of unexpected messages (Brightwell and Underwood 2004a) and that the eager MPI implementation that enables independent progress on ASCI Red causes significant performance degradation in the presence of a large number of unexpected messages (Brightwell and Underwood 2003). Furthermore, when benchmarking collectives, there is no computation to overlap with the communication.

The `MPI_Alltoallv` collective is the only significant contributor to communication time for the IS benchmark. As such, the time to perform `MPI_Alltoallv` is shown in the right-hand side of Figures 4(b), (d), and (f) for each of the MPI implementations evaluated. On ASCI Red, there are effectively two classes of performance on this collective: slower performance (longer time) for the default MPI implementation and faster performance (shorter time) for the MPI implementations that use a rendezvous protocol. The same properties and analysis from `MPI_Alltoll` apply here as well.

What is more surprising is the data from the Quadrics network. At small message sizes (under 12 KB and not visible on the graph), the SHMEM implementation of `MPI_Alltoallv` holds a slight performance advantage. This is an impact of the slower processor on the Quadrics network interface that must handle the matching semantics (Underwood and Brightwell 2004). By 12 KB, this advantage has vanished as the overhead of match list traversal is amortized over larger messages. When the message size reaches 1 MB, the Tports implementation is over 30% faster.

5.2 NAS PARALLEL BENCHMARKS

The next phase of the evaluation compared the performance of the class B NAS parallel benchmarks on each of the MPI implementations. EP (the embarrassingly parallel benchmark) was excluded since it does not do any significant communication. Each benchmark was run four times for each number of processors per job. The average of the four runs is reported in Figures 5, 6, and 7. The measurements from ASCI Red were extremely stable (varying by less than 0.5%); thus, even small differences can be directly attributed to a cause. These measurements offer an opportunity to attempt to separate the impacts of overlap, independent progress, and MPI offload.

Figures 5 and 6 show data for the BT, IS, MP, and SP class B benchmarks taken from a Quadrics cluster and the ASCI Red platform. These are the only benchmarks that show measurable advantage from the introduction of overlap, offload, or independent progress on Elan3.

The data for the MG benchmark are shown in Figures 5(a) and (b). Independent progress provides a slight but consistent advantage. The Tports implementation outperforms the SHMEM implementation by a slight, but consistent, margin. For ASCI Red, the eager implementation on P0 mode provides independent progress and outperforms the rendezvous implementation on P0 mode.

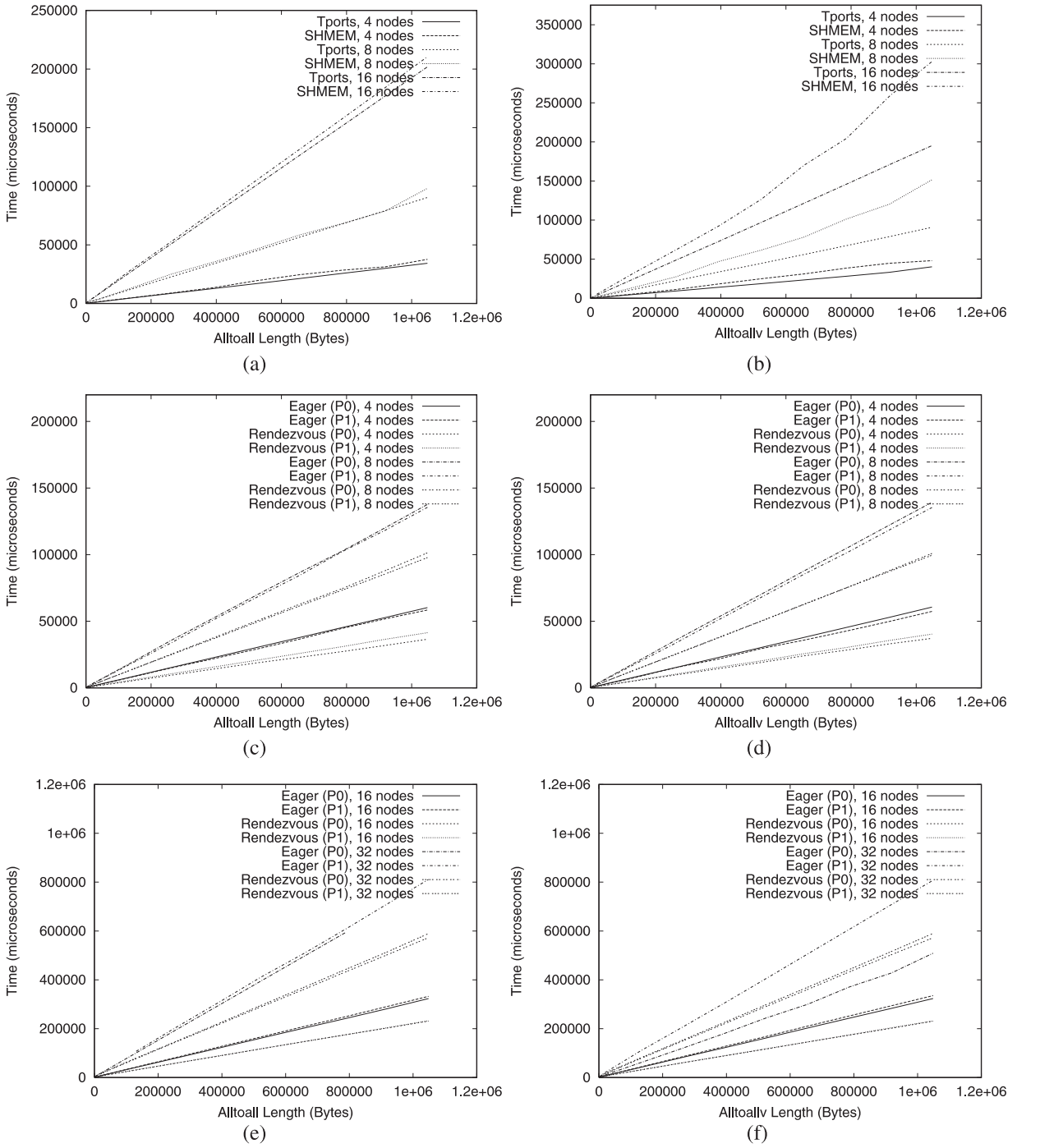

Fig. 4 A comparison of (a) Alltoall and (b) Alltoallv performance for two implementations on Elan 3 and (c), (e) Alltoall and (d), (f) Alltoallv performance for two implementations on ASCI Red.

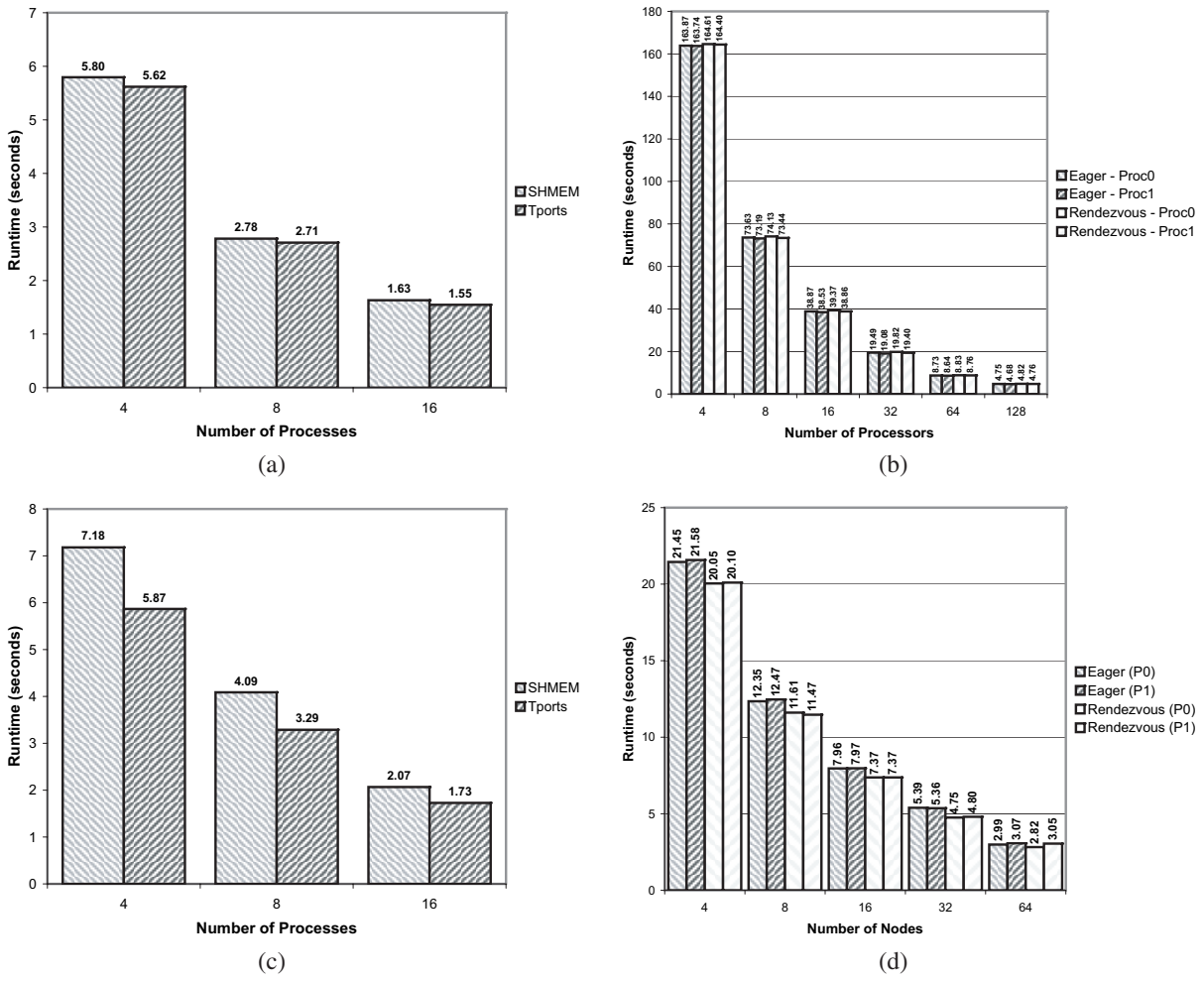

Fig. 5 Comparison for (a) MG on Elan3, (b) MG on ASCI Red, (c) IS on Elan3, and (d) IS on ASCI Red.

Moving from P0 mode to P1 mode gives an advantage to both the eager and the rendezvous implementations. This demonstrates that the additions of overlap and offload are also an advantage to the MG benchmark, but independent progress still contributes to the performance advantage as the P1 mode runs with the eager MPI implementation outperform the proc1 mode runs with the rendezvous MPI implementation.

Figure 5(c) indicates that Tports outperforms SHMEM by a significant margin on the IS benchmark. This measurement was taken without using the enhanced collectives provided by the default Quadrics MPI (for fairness), but using the enhanced collectives provided virtually no benefit for IS. This is somewhat (but not completely) explained by the significant difference in the `MPI_Alltoallv` performance differences seen in Figure 4. Using Figure 5(d), this difference can be explained by independent progress. Doing so requires some insight. First, the advantages of offload and overlap can be excluded by noticing that P0 performance and P1 performance are virtually identical. Secondly, contrary its appearance, Figure 5(d) does not contradict the conclusion that the performance advantage Tports over SHMEM is independent progress. This is because the independent progress mechanism on ASCI Red is an eager protocol which experiences a significant reduction in bandwidth in the presence

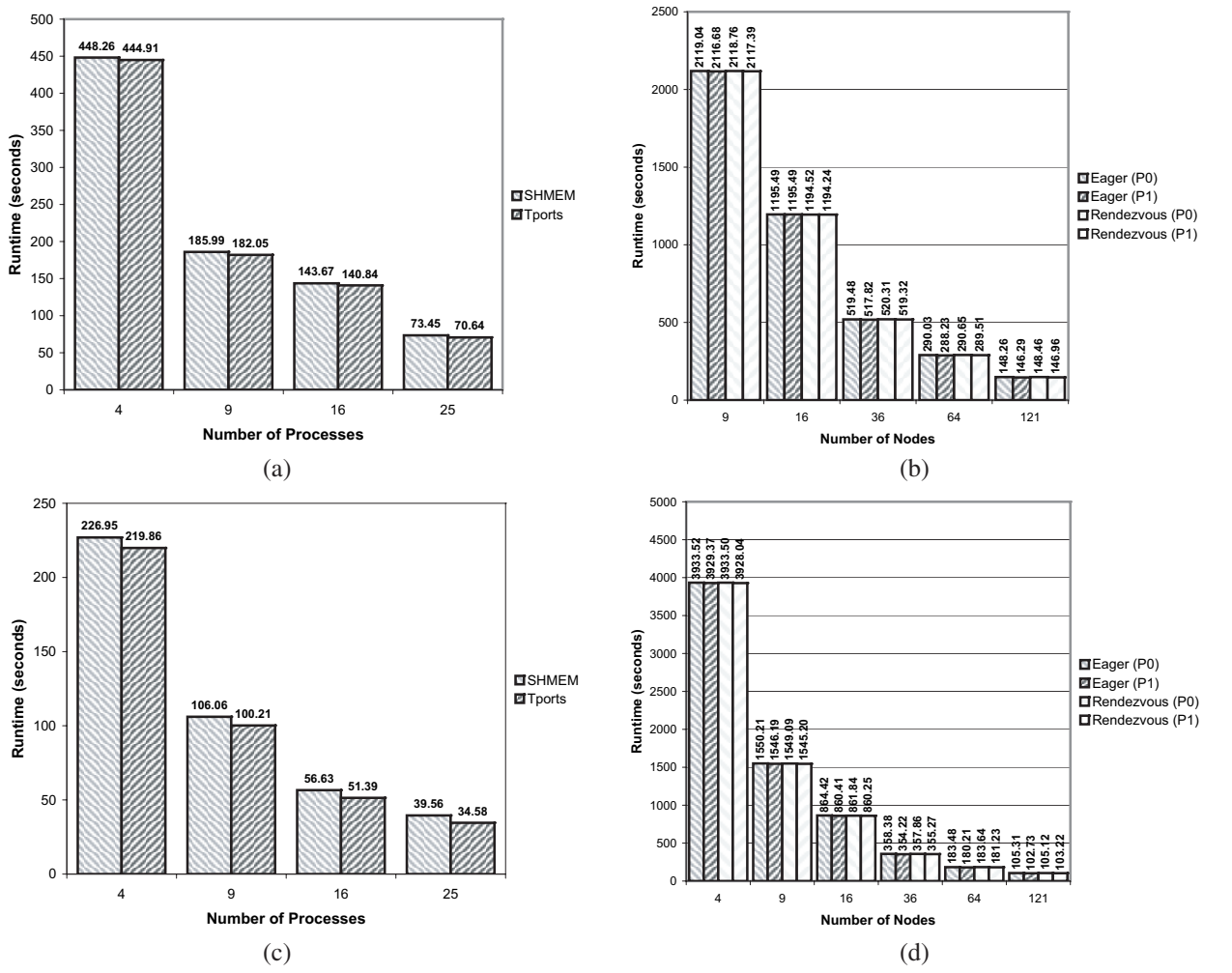

Fig. 6 Comparison for (a) BT on Elan3, (b) BT on ASCI Red, (c) SP on Elan3, and (d) SP on ASCI Red.

of unexpected messages (Brightwell and Underwood 2003) (as seen in the IS benchmark; Brightwell and Underwood 2004a). This is further reinforced by the data from the rendezvous library run in P1 mode. The addition of overlap and offload to the rendezvous library (without independent progress) adds no performance improvement.

The benefits of independent progress for IS are counterintuitive since the primary communications routine in the IS benchmark is a collective operation; however, the Rogue OS effect (Petrini et al. 2003) is known to cause interference with collective operations. Specifically, in this case, it is key that the independent progress is independent of the application process altogether. Although this is achieved with offload for the Quadrics Tports interface, it can also be achieved with interrupt-driven progress, as it is on ASCI Red; thus, it is independent progress, not offload, that affects the performance of the IS benchmark.

The SP and BT benchmarks are more clear cut, especially as the applications scale in the number of nodes. The SP and BT benchmarks using MPI over Tports consistently outperform MPI over SHMEM on the Quadrics network. The advantage is 3–6% in overall execution time for SP and 2–4% in overall execution time for BT.

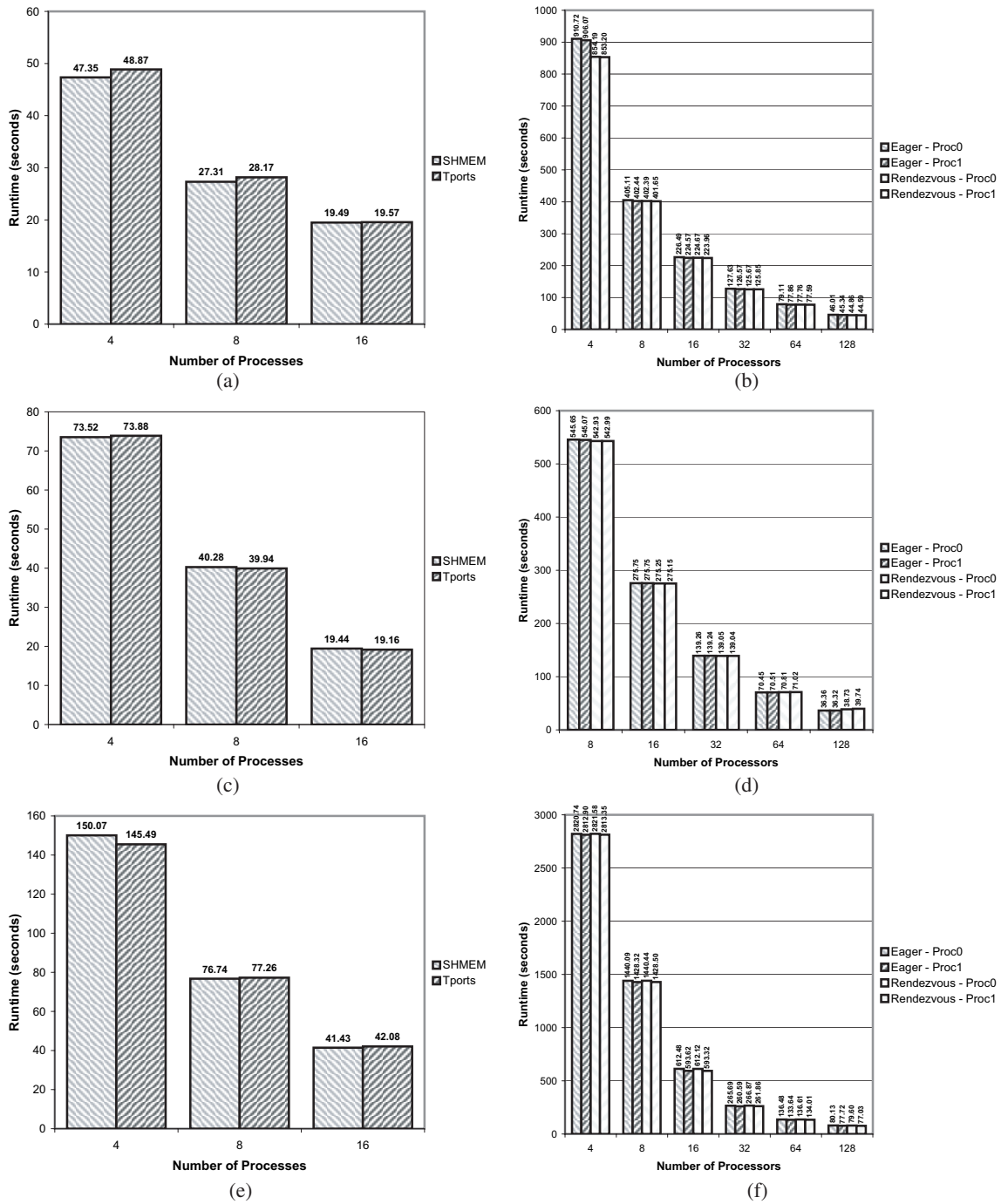

Fig. 7 Comparison for (a) CG on Elan3, (b) CG on ASCI Red, (c) FT on Elan3, (d) FT on ASCI Red, (e) LU on Elan3, and (f) LU on ASCI Red.

Using the 64 node runs1 from ASCI Red, SP sees a fractional gain from adding independent progress (moving from rendezvous to an eager implementation). It sees another 1.5% gain when moving from P0 to P1 mode. This is the combined advantage of adding protocol offload and adding overlap. Notably, when the rendezvous implementation is run in P1 mode, overlap and offload offer less than 1% gain. This is a clear indication that independent progress needs overlap and offload to achieve maximal performance, and vice versa. Similarly, BT sees a fractional gain from independent progress but an additional 2.5% gain from the combination of offload and overlap. Again, overlap and offload alone offer only a portion of the advantage.

Moving to 121 nodes changes the picture slightly. At 121 nodes, most of the messages use the short message protocol. Thus, the rendezvous protocol reaps many of the advantages of independent progress without truly providing it. Indeed, when adding only independent progress, the BT benchmark actually slows down; however, the combination of independent progress, overlap, and offload still offers a 0.5% advantage over overlap and offload alone.

The most important result here is that overlap must be combined with independent progress to achieve the best results. Independent progress alone suffers from issues such as cache pollution and context switch overheads. Offload and overlap alone suffer because they must wait for the application to return to the MPI library to progress communications. The combination of the three enables a measurable impact on application performance.

Figure 7(a) shows the performance of CG on Elan3. CG is one of the few benchmarks where SHMEM MPI appears to have advantages. For small numbers of nodes, SHMEM MPI has up to a 3% performance advantage over Tports MPI with measurements that vary by less than 0.5%. This performance advantage is surprising since since CG was found to be a well-behaved application in terms of MPI queue usage (Brightwell and Underwood 2004a). The cause is likely to be the use of messages that are predominantly smaller than 2 KB as discussed in Liu et al. (2003b). However, this advantage evaporates at 16 nodes.

Considering the same analysis from ASCI Red that is shown in Figure 7(b) illustrates that independent progress still offers no advantage. Since CG has a significant number of unexpected messages (Brightwell and Underwood 2004a), and unexpected messages significantly decrease bandwidth with the eager protocol (Brightwell and Underwood 2003), this is not surprising. It would also tend to indicate that CG is not able to overlap significant computation with communication. Looking at the minor improvements achieved by moving from P0 mode to P1 mode would indicate that CG is able to derive some benefit from offloading the network processing.

The FT benchmark would not be expected to see a particular advantage from independent progress because all of the communications occur in the `MPI_Alltoall` collective. For smaller configurations, we find that FT runs at the same speed (or slower) when using the MPI implementation that delivers independent progress rather than the MPI implementation that does not. This follows from the data seen in Figure 4 for the collective benchmarks. What is interesting, however, is that FT receives an 6% advantage at 128 nodes from independent progress. Timings for these runs varied by less than 0.2%, so this was not experimental error. Furthermore, since the code was run on a 128-node version of ASCI Red, only one allocation was possible. All of the advantage appears to arise from independent progress, since P0 and P1 mode runs performed the same.

Tports showed no advantage over the SHMEM implementation for the LU implementation; however, on ASCI Red, the results were somewhat different. P1 mode shows a distinct advantage over P0 mode for both the eager and rendezvous MPI implementations. The variability in the runs was only 0.07% and the difference in the run time between the different implementations was 2–3%. Since P1 mode showed advantage over P0 mode in both cases, we can isolate the advantage to either overlap or offload. Since SHMEM and Tports have the same performance, we can determine that overlap is the source of the performance advantage.

An interesting side note is that the FT and IS benchmarks were expected to pose a significant performance issue for an MPI implementation that offloads queue handling to a slower processor on the NIC (e.g. MPI over Tports). Both of these benchmarks have long unexpected queues, long posted receive queues, and long average traversals of each (Brightwell and Underwood 2004a). Traversing these queues on an embedded processor should have proved costly; however, Figures 5(c) and 7(c) clearly indicate otherwise. Performance for FT is effectively equivalent between the two implementations and Tports MPI has a significant advantage (20%) for IS. Similarly, despite the high number of long unexpected messages for FT, it breaks even on ASCI Red and begins to win at larger numbers of nodes.

5.3 APPLICATIONS

The final portion of the analysis places a focus on real applications. Two different applications were evaluated with two input sets being evaluated on one of the implementations. Further data about the applications and input sets can be found in previous work (Rodrigues et al. 2004). The results for each application from ASCI Red are shown in Figure 8. The two input sets for LAMMPS were required to be run on a cubed number of processors.

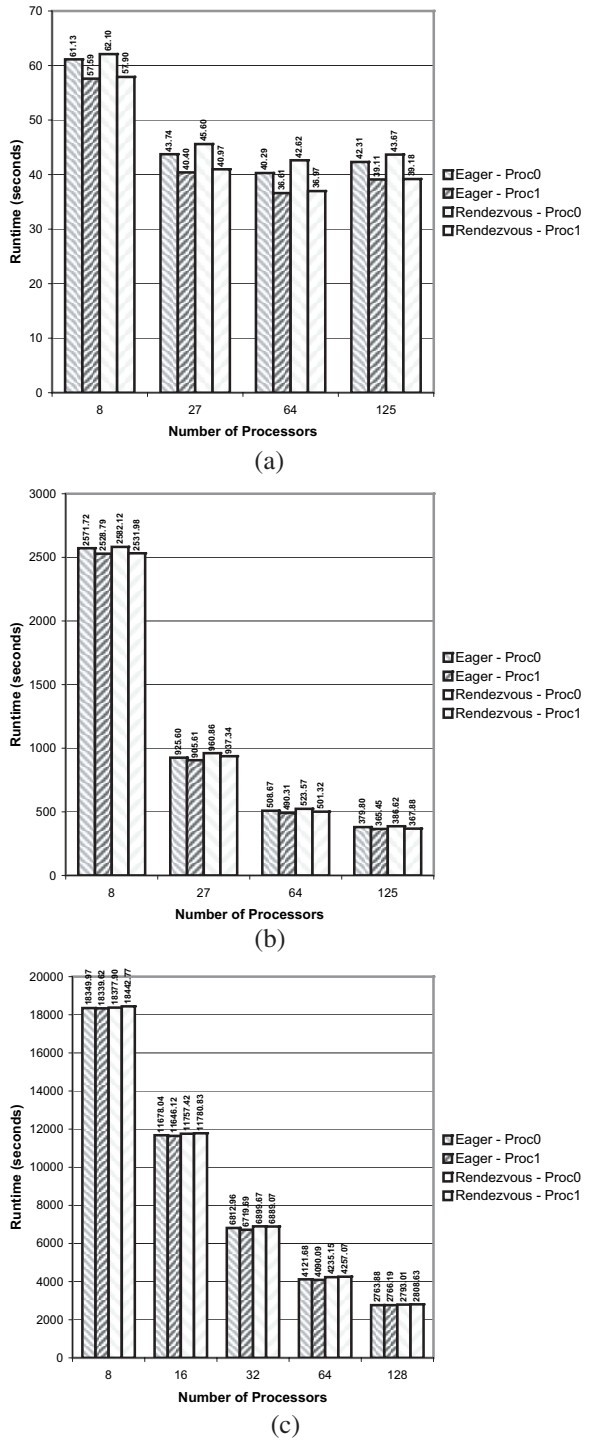

Fig. 8 Comparison for (a) LAMMPS with the Lennard–Jones system input set, (b) LAMMPS with the bead spring polymer chain input set, and (c) CTH with EFP3D.

The Lennard–Jones system input to the LAMMPS application provides the most dramatic and consistent of the three results. Independent progress provides a 2% benefit when overlap and offload are not available. The combination of overlap and offload provide a 10% improvement in performance, and in the presence of overlap and offload, independent progress provides no advantage for this application.

The bead spring polymer chain input to LAMMPS is much more difficult to interpret. In all cases, the addition of overlap and offload by shifting from P0 mode to P1 mode is significant. The advantage ranges from as little as 2% when independent progress is already available and the compute time is long to as much as 8% when independent progress is not available and the compute time is shorter. Independent progress alone has an advantage of 2–3% when overlap and offload are not available. What is harder to interpret is the advantages of independent progress when overlap and offload are both present. In 8 and 128 node runs, the added advantage of independent progress in the presence of overlap and offload is minimal. In 27- and 64-node runs, it is 2–3%. We believe that the discrepancy arises from the ratio of the computation to communication. In the eight-node runs, computation time swamps communication time and hides many of the differences in the MPI implementations. In the 128-node runs, this fixed-size problem has begun to experience decreased scaling efficiency. This occurs when communication time begins to outweigh computation time; thus, the application is spending more time in MPI. Applications that spend much of their time in MPI tend to see little benefit from independent progress.

The EFP3D input to the CTH application show relatively consistent behavior across the various sizes of runs. Independent progress consistently improves performance by 1%. Overlap and offload alone (rendezvous P0 to rendezvous P1) consistently shows a slight decrease in performance. Adding overlap and offload to the independent progress enabled MPI implementation shows up to a 1% gain, but this is a less consistent advantage.

6 Conclusions

In this paper we have quantitatively explored the advantages of overlap, independent progress, and offload using the NAS parallel benchmarks and two applications. Combined, they offer a 2–20% advantage in overall execution time depending on the benchmark. This is particularly significant in the context of $100 million acquisitions such as those found in the ASC program. In such cases, a 2% improvement in overall execution time justifies the expenditure of almost $2 million.

In this study we set out to evaluate the impacts of an MPI implementation that provides independent progress and overlap capabilities by using network interface- or processor-

based MPI offload. The expectation was that both advantages and disadvantages would be uncovered: advantages for "well-behaved" applications that had short MPI queues, and disadvantages for "poorly-behaved" applications that used long MPI queues. In the process, we sought to answer three questions. Does offload onto a slower network interface processor negatively affect applications with long MPI queues? Does independent progress yield an advantage to applications? Do overlap and offload impact application performance? The answers were somewhat surprising.

The Tports MPI (the offloading MPI) wins almost uniformly (and sometimes by a significant margin) despite the fact that the SHMEM MPI has almost identical performance. In the rare cases (three of 23 data points) where SHMEM MPI is faster, it is by a very small margin. In contrast, Tports MPI has a surprisingly large margin of victory over SHMEM MPI (20%) on the "poorly behaved" IS benchmark, a 5–10% advantage for SP, and a 2–4% advantage for BT. Also surprising was the fact that SHMEM MPI had a slight edge in performance for CG (a "well-behaved" application). In summary, the benefits of independent progress, overlap, and offload were not outweighed by the slow network interface processor that offloads MPI queue handling for "poorly-behaved" applications; however, improving network interface performance may benefit these applications further.

Taken by itself, independent progress is a critical contributor to application performance improvements. For BT, SP and all of the real applications, independent progress improves performance as seen on ASCI Red and the Quadrics network. Similarly, the combination of overlap and offload in the absence of independent progress accelerates a number of the benchmarks (particularly LU, BT, SP, and all of the applications). The striking result is that the combined performance improvement from independent progress, overlap, and offload is often more than simply the composition of the parts. Another important observation comes from the IS benchmark. The results for IS are a dramatic improvement of 20% on the Quadrics network, but an actual loss of performance on the ASCI Red system. This highlights an important aspect for implementers of independent progress for MPI: the implementation must be careful not to sacrifice too much performance for non-optimal applications, such as those like IS that have a significant number of unexpected messages.

ACKNOWLEDGMENTS

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000

The authors would like to gratefully acknowledge the CCS-3 group at Los Alamos National Laboratory, especially Fabrizio Petrini, for providing access to the Elan3 cluster used for our experiments.

AUTHOR BIOGRAPHIES

Ron Brightwell received his B.S. in mathematics in 1991 and his M.S. in computer science in 1994 from Mississippi State University. He joined Sandia National Laboratories in 1995 after serving as a research assistant in the system software group at the MSU/NSF Engineering Research Center for Computational Field Simulation. While at Sandia, he has designed and developed high performance implementations of the MPI Standard on several large-scale, massively parallel computing platforms, including the Cray T3D and T3E, the Intel Paragon and TeraFLOPS (ASCI/Red), and Sandia's Computational Plant clusters. His research interests include high performance, scalable communication interfaces and protocols for system area networks, operating systems for massively parallel processing machines, and parallel program performance analysis libraries and tools. He is also currently pursuing a Ph.D. in the Department of Computer Science at the University of New Mexico.

Rolf Riesen is a principal member of technical staff at Sandia National Laboratories in Albuquerque, New Mexico. He holds a Ph.D. degree in computer science from the University of New Mexico. His research interests include message passing systems, operating systems, and runtime software for massively parallel computers. Over the last 10 years in this field he has primarily concentrated on topics related to efficient, scalable message passing and interactions at the software/hardware boundary. Dr. Riesen has been a key member of the design team for SUNMOS for the nCUBE-2 and the Intel Paragon, as well as Puma/Cougar, the second generation lightweight kernel for the Intel ASCI Red machine. He is also a key designer of the Portals message passing mechanism. He has been a principal designer of Cplant, the largest commodity cluster for scientific applications in the world. He was involved in the overall design and the low-level message passing layers, including the scalable wire protocol used by Cplant.

Keith Underwood received B.S. and Ph.D. degrees in computer engineering from Clemson University in 1995 and 2002, respectively. His dissertation research focused on incorporating FPGA devices in the network interface to create an intelligent NIC that was capable of processing the entire network data stream. He joined Sandia National Labs as a senior member of the technical staff in 2002. He has recently been exploring the ability of FPGAs to perform IEEE compliant floating-point arithmetic to support scientific computation. He has also researched hardware mechanisms to accelerate MPI processing. His research

interests include high performance networking for next generation supercomputers, programmable network interfaces, and the role of reconfigurable computing in high performance computing systems.

NOTES

1 These runs have more comparable execution times to those on the Quadrics cluster and therefore more comparable balances of communication and computation.

References

Boden, N. J., Cohen, D., Kulawik, R. E. F. A. E., Seitz, C. L., Seizovic, J. N., and Su, W-K. 1995. Myrinet: a gigabit-per-second local area network. *IEEE Micro* 15(1):29–36

Brightwell, R. 2004. A new MPI implementation for Cray SHMEM. *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 18–21.

Brightwell, R. and Shuler, P. L. 1996. Design and implementation of MPI on Puma portals. *Proceedings of the 2nd MPI Developer's Conference*, July, pp. 18–25.

Brightwell, R. and Underwood, K. 2003. Evaluation of an eager protocol optimization for MPI. *Proceedings of the 10th European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, Venice, Italy, September.

Brightwell, R. and Underwood, K. D. 2004a. An analysis of NIC resource usage for offloading MPI. *Proceedings of the 2004 Workshop on Communication Architecture for Clusters*, Santa Fe, NM, April.

Brightwell, R. and Underwood, K. D. 2004b. An initial analysis of the impact of overlap and independent progress for MPI. *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September.

Brightwell, R., Hudson, T. B., Maccabe, A. B., and Riesen, R. E. 1999. The Portals 3.0 message passing interface. Sandia National Laboratories, Technical Report SAND99-2959, December.

Brightwell, R., Lawry, W., Maccabe, A. B., and Riesen, R. 2002. Portals 3.0: protocol building blocks for low overhead communication. *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April.

Cray Research, Inc. 1994. *SHMEM Technical Note for C, SG-2516 2.3*, October.

Dimitrov, R. and Skjellum, A. 2000. Impact of latency on applications' performance. *Fourth MPI Developers' and Users' Conference*, March.

Liu, J., Wu, J., Kini, S. P., Wyckoff, P., and Panda, D. K. 2003a. High performance RDMA-based MPI implementation over InfiniBand. *Proceedings of the 2003 International Conference on Supercomputing (ICS-03)*, June 23–26, ACM Press, New York, pp. 295–304.

Liu, J., Chandrasekaran, B., Wu, J., Jiang, W., Kini, S., Yu, W., Buntinas, D., Wyckoff, P., and Panda, D. K. 2003b. Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. *Proceedings of the International Conference for High Performance Computing and Communications (SC2003)*, November.

Liu, J., Chandrasekaran, B., Yu, W., Wu, J., Buntinas, D., Kini, S. P., Wyckoff, P., and Panda, D. K. 2004. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro* 24(1):42–51.

Maccabe, A. B., Riesen, R., and van Dresser, D. W. 1996. Dynamic processor modes in Puma. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)* 8(2):4–12.

Martin, R. P., Vahdat, A. M., Culler, D. E., and Anderson, T. E. 1997. Effects of communication latency, overhead, and bandwidth in a cluster architecture. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June.

MPI Forum. 1994. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing* 8(3–4):159–416.

MPI Forum. 1997. MPI-2: Extensions to the Message-Passing Interface, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

Petrini, F., Chun Feng, W., Hoisie, A., Coll, S., and Frachtenberg, E. 2002. The Quadrics network: high-performance clustering technology. *IEEE Micro* 22(1):46–57.

Petrini, F., Kerbyson, D. J., and Pakin, S. 2003. The case of the missing supercomputer performance: identifying and eliminating the performance variability on the ASCI Q machine. *Proceedings of the 2003 Conference on High Performance Networking and Computing*, Phoenix, AZ, November.

Rehm, W., Grabner, R., Mietke, F., Mehlan, T., and Siebert, C. 2004. An MPICH2 channel device implementation over VAPI on InfiniBand. *Proceedings of the 2004 Workshop on Communication Architecture for Clusters*, April.

Rodrigues, A., Murphy, R., Kogge, P., and Underwood, K. 2004. Characterizing a new class of threads in scientific applications for high end supercomputers. *Proceedings of the 2004 International Conference on Supercomputing (ICS2004)*, St. Malo, France, June.

Shuler, L., Jong, C., Riesen, R., van Dresser, D., Maccabe, A. B., Fisk, L. A., and Stallcup, T. M. 1995. The Puma operating system for massively parallel computers. *Proceedings of the 1995 Intel Supercomputer User's Group Conference*.

Underwood, K. D. and Brightwell, R. 2004. The impact of MPI queue usage on message latency. *Proceedings of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August.

Vetter, J. S. and Mueller, F. 2002. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, April, pp. 27–29.

Wheat, S., Mattson, T. G., and Scott, D. 1996. A TeraFLOPS Supercomputer in 1996: the ASCI TFLOP System. *Proceedings of the 1996 International Parallel Processing Symposium.*

Wong, F., Martin, R., Arpaci-Dusseau, R., and Culler, D. E. 1999. Architectural requirements and scalability of the NAS parallel benchmarks. *Proceedings of the SC99 Conference on High Performance Networking and Computing*, November.