Implications of Application Usage Characteristics for Collective Communication Offload

Ron Brightwell, Sue Goudy, Arun Rodrigues, and Keith D. Underwood
Sandia National Laboratories
P.O. Box 5800
MS-1110
Albuquerque, NM 87185-1110
Email: {rbbrigh, spgoudy, afrodri, kdunder}@sandia.gov

***Abstract*— The performance of collective communication operations is known to have a significant impact on the scalability of some applications. Indeed, the global, synchronous nature of some collective operations directly implies that they will become the bottleneck when scaling to hundreds of thousands of nodes. This fact has led many researchers to try to improve the efficiency of collective operations. One popular approach improves the implementation of MPI collective operations by using intelligent or programmable network interfaces to offload the burden of communication activities from the host processor(s). Such implementations have shown significant improvement for micro-benchmarks that isolate collective communication performance, but these results have not been shown to translate to significant increases in performance for real applications. In order for collective offload implementations to benefit real applications, a greater understanding of application behavior is needed. In this paper, we describe several characteristics of applications and application benchmarks that impact collective communication performance. We analyze network resource usage data in order to guide the design of collective offload engines and their associated programming interfaces. In particular, we provide an analysis of the potential benefit of non-blocking collective communication operations for MPI.**

***Index Terms*— MPI, non-blocking, collective, resource usage, resource management, network interface**

I. Introduction

One of the most common models for parallel scientific codes within the Department of Energy (DOE) complex is to compute for one time step, perform global reductions to compute global information (e.g. the length of the next time step), and compute the next time step. This process repeats until the desired simulation time has completed. As the number of nodes used for the computation grows, the best case scenario is a $log_k(N)$ growth in collective time (where k is the radix of the collective) and a constant time (per node) to compute the time step. The worst case is a superlinear growth (or, the measured linear growth[1]) in collective time due to such things as low-level system software effects [2] and a decrease in the time (per node) to compute the time step (because of a shrinking problem size per node). In either scenario, the scalability of the application is clearly limited by the effectively serialized global operation.

The only solution to achieve scalability is to minimize this serial fraction of the code. This quest has spurred numerous research efforts into collective offload designs[3], [4], [5], [6], [7], [8]. Recently, much research has focused on leveraging the programmable processor that has become common on high-performance network interfaces to offload collective operations. While these studies have considered many of the important issues facing the use of collectives in applications, they lack an analysis of the applications that will use the collective offload engine. Questions that must be addressed include: Which collectives are typically used? Do they require floating-point? What are the resource usage characteristics? Without answers to these types of questions, it is difficult to design an effective collective offload engine. This research attempts to answer many of these questions in the context of large scale scientific applications used at Sandia.

Unfortunately, applications that run at the largest scale are frequently unavailable to the broader research community because of licensing or export control restrictions. Instead, supercomputer users try to make "representative" benchmarks available to supercomputer researchers. The most prominent example of this is the NAS Parallel Benchmark (NPB) suite[9], [10]. Regrettably, benchmark suites must frequently sacrifice fidelity for simplicity and portability. Because of this, it is often difficult to gauge whether they are representative of real applications in any given aspect. In addition to analyzing the way large-scale scientific applications use collectives, this paper compares that usage to the properties of the NAS Parallel Benchmark suite.

Finally, simply offloading collective operations is a short-term fix. The effectively serial portion of the code that is spent in many collectives will grow with the number of nodes. With processor counts projected to reach past 100,000 and into the millions, this will prove to be a fundamental limitation. Thus, the true limitation is the blocking nature of many collective operations (e.g. MPI_Allreduce) that prevent the application from exploiting offload and overlap capabilities provided by the hardware. The alternative is to use "non-blocking" or "split-phase" collective operations. For this to be practical, however, there must be instructions that can execute between the time that the collective could be started and the time that

A. Rodrigues is also with the University of Notre Dame Computer Science and Engineering Department, 384 Fitzpatrick Hall, Notre Dame, IN 46556.

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 result of the collective is needed. This paper assesses the feasibility of non-blocking collectives in the context of current applications. This is a difficult property to study since applications are not designed and written to use non-blocking collectives; however, by using instruction tracing, we are able to estimate the number of instructions in current codes that could be overlapped with collective operations. The results indicate that non-blocking collectives are indeed feasible and that a significant amount of overlap can be achieved.

In the next section, previous work on collectives is discussed. Section III covers the approach used to collect this data. Section IV discusses data from the NAS parallel benchmarks and Section V presents data from scientific applications at Sandia National Labs. Data supporting non-blocking collectives is presented in Section VI, followed by conclusions in Section VII.

II. Background

Collective operations are well-known parallel programming primitives that have been, and continue to be, widely studied. A vast amount of work has been devoted to optimizing collective algorithms (particularly synchronization) on shared memory systems. Examples include [11], [12], [13], [14], [15]. Other works have focused on message passing architectures [16], [17], [18], [3], [19], [4], [5], [6].

It has been demonstrated that collectives are important to application performance [1], [19], but it has also been demonstrated that the performance of collectives is typically obscured by application performance [19]. Indeed, even a thirty-fold increase in collective performance may yield only a three-fold increase in application performance [19]. Thus, for the sake of clarity, it has become commonplace to investigate collective performance outside the context of application performance.

Recent research on collective operations has continued to follow this trend to the extent that all application context has been lost. A tightly synchronized loop that calls a single collective operation thousands of times is not representative of how applications use collectives. These research efforts have included studies of how to leverage the remote DMA (RDMA) capabilities of modern networks [20], [21] as well as how to offload collectives onto the network interface [7].

More realistic work has studied issues with performing reduction operations on the NIC. These include the issues such as slow floating-point implementations[22] and even how to deal with unexpected messages[8] and application skew[23], [24]. However, they lack the application data to support selecting an optimal mechanism to deal with the issues. This paper provides some of that essential data.

This paper provides insight into the way that applications actually use collectives to provide appropriate data as a baseline for future research on dedicated hardware and programming interfaces for collectives. Furthermore, the wealth of work on collective offload designs provides an opportunity to consider applying "application bypass" [25], [24], [23] principles to collective operations. Leveraging application bypass would require a non-blocking collective interface. This paper establishes that such an interface could be beneficial.

III. Approach

Two types of data were collected. First, applications were analyzed on a parallel platform to measure timing and scale-sensitive characteristics of applications. Second, applications were traced on a desktop platform to measure timing and scale-insensitive parameters.

A. MPI Profiling

In previous work, we collected and analyzed data related to the MPI posted receive and unexpected message queues, and measured expected versus unexpected messages [26]. We used these measurements to characterize the amount of processing and memory resources that are needed for offloading MPI semantics. Our approach to measuring data for collective operations is similar. While the previous analysis included MPI point-to-point and collective operations together, we have separated the two for this analysis.

In order to separate the resource usage information of collective and point-to-point operations, we modified the MPICH/GM implementation to have a separate posted receive and unexpected queues for collective messages. Since collective operations are layered on the MPI point-to-point operations in the MPICH/GM implementation, the individual messages are tagged as being either point-to-point or collective. We use the MPI profiling interface to turn on a flag that indicates whether the code is in a collective operation. This flag is recognized by code at the MPI device level and is used to distinguish between point-to-point and collective messages.

In addition to the network-level instrumentation needed to collect MPI queue and unexpected message data, we also used the MPI profiling interface to collect higher-level information about collective communication characteristics. We used the profiling interface to track the frequency of each collective communication call and gathered various information about the individual collective operations, such as whether reduction operations were integer or floating point.

B. Tracing

Applications were traced using the `amber` instruction trace filter for the PowerPC[27] processor. Analysis of the traced values for the program counter allowed MPI calls to be identified. Function call arguments were extracted from the traced register values for each collective call. These arguments allowed message buffers to be recognized. Any future loads from an address in a receive buffer were recorded so that the consumption of this buffer could be tracked. To analyze outgoing buffers, a record was kept of when each memory address was last written. Whenever an outgoing buffer was recognized, this record was consulted to determine when the buffer was constructed. The evaluation metric used was the number of elapsed instructions that could have been overlapped with a non-blocking collective. In modern systems, this approximately correlates to a metric of time, though that metric will vary based on the system. Instructions executed within the MPI library or system calls were not counted.

Each application or benchmark was traced for 10 billion instructions, or until the program completed. Ideally, the

complete application would be traced in all cases; however, realistically, time constraints were a restriction — tracing yields an execution rate of only 0.2 MIPS. Ten billion instructions is of a length greater than that typically used for processor research and is considered to be representative of application behavior. Benchmarks from the NPB used the 'S' (sample) data sets. Most applications were run with 2 nodes and process 0 was traced. BT and SP from the NAS Suite were run on 4 nodes. CG was omitted, as it only uses point to point and `MPI_Reduce()`.

C. Platforms

Two different platforms were used for these experiments. The first platform provided parallel execution for the applications at larger scales while the second platform was used to collect instruction tracing information. The parallel execution platform was the Vplant machine at Sandia National Laboratories. Vplant is a large Linux cluster with approximately 320 compute nodes composed of Intel Pentium-3 and Pentium-4 processors. These experiments were run on a 126 node partition of the machine containing dual-processor Pentium-4 Xeon nodes running at 2.0 GHz. Each node has 1 GB of main memory and a Myrinet-2000 [28] network interface. The nodes are connected in a Clos topology. Vplant was running a Linux 2.4.20 kernel, GM version 2.0.11, and MPICH/GM version 1.2.5..11. The crossover point between short messages and long messages for MPICH/GM is 16 KB. All of our runs used only one process per node.

Instruction tracing used two node run and was performed on Apple Macintosh G4 and G5 systems running MacOS-X version 10.3.3. Applications were compiled with the Absoft ProFortran compiler for MacOS (version 8.0) or gcc (version 3.3). All of the binaries were 32 bit PowerPC executables. The LAM implementation of MPI[29] was used.

IV. Benchmark Analysis

One of the most commonly studied set of benchmarks is the NAS Parallel Benchmarks (NPB) suite version 2.4 [10]. These benchmarks are a collection of MPI applications distilled from real computational fluid dynamics applications. We omitted the EP benchmark from our study, since it does virtually no message passing. We chose to present the class B problems since they run more quickly and most of the salient features (collective types, data types, and queue usage behavior) do not change with problem size. Class C benchmarks were also run and deviations from the class B data presented are noted. These benchmarks have been well-studied, and are provided for comparison with the real application data.

A. Unexpected Messages

The first point for analysis was the behavior of the unexpected message queue. Previous work [26], [30] has studied the queue behavior of various applications and benchmarks and found that unexpected message queues could grow quite long. Figure 1 breaks out the component of unexpected message queue behavior that is specific to collective operations.

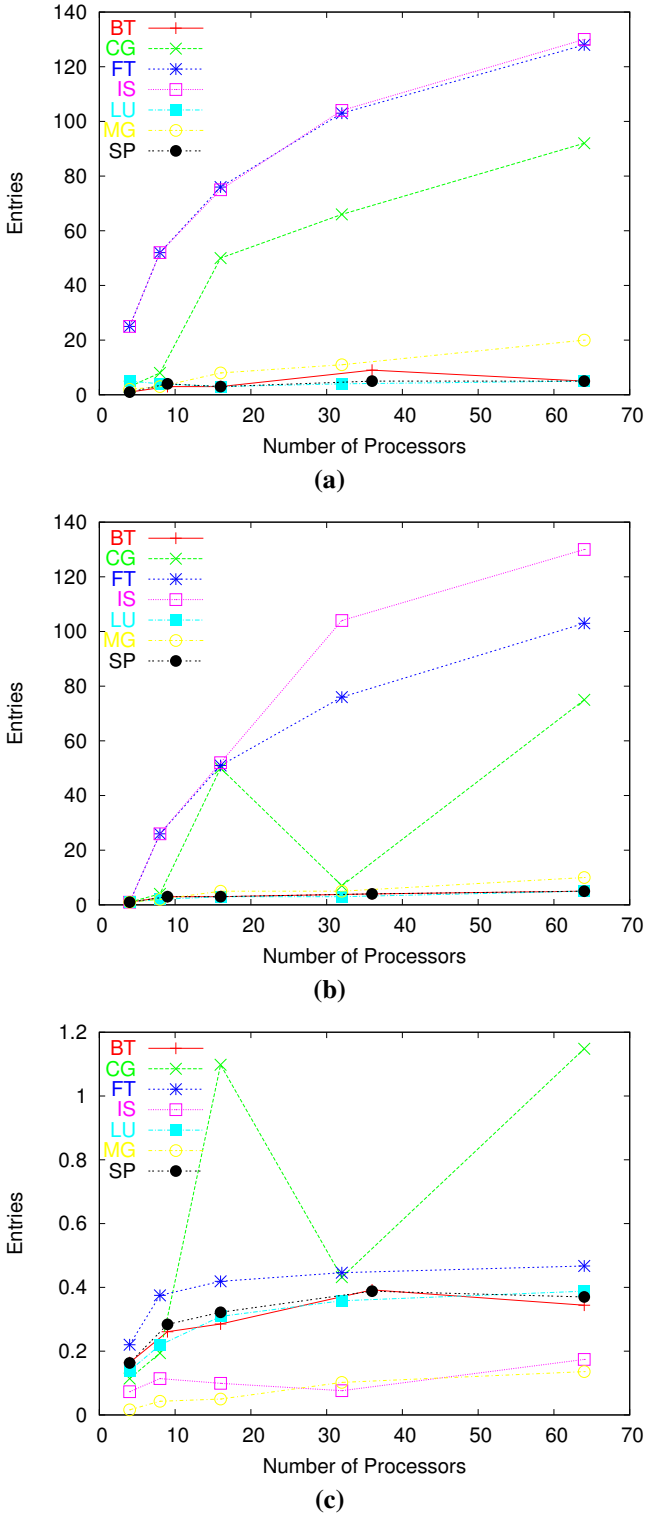


Fig. 1. NPB unexpected queue statistics: **(a)** max length, **(b)** max search depth, **(c)** average search depth

Figure 1(a) shows the maximum length of the unexpected message queue. It illustrates that the unexpected message queue associated with collective operations can grow nearly linearly with the number of processors for a number of the NAS parallel benchmarks. More disturbingly, Figure 1(b), which shows the maximum search depth of the queue, indicates that it is possible to have to search this entire queue; however, Figure 1(c) (the average search depth of the queue) indicates that this is a relatively uncommon event.

The data from Figure 1 stands in contrast to what would be expected in the standard micro-benchmark that evaluates collective optimizations. At least for the NPB suite, applications can and do enter collective operations at disparate times. A NIC implementation of collective operations must be aware of this fact as it may require the allocation of additional resources and may affect the way the algorithms should be optimized.

Offload implementations must also be careful to maintain message ordering semantics. Even though MPI mandates that collective functions must be called in the same sequence across all ranks in the same communicator, tree-based implementations of certain collective routines could cause messages to arrive out of order. For example, two successive `MPI_Bcast` calls with different root values could cause messages to arrive out of order at a node that is a leaf node in the first broadcast, but is near the root in the second broadcast.

Since unexpected messages appear in the collectives for these benchmarks, Figure 2 separates the messages in the collectives into the percentage of each of four types: long expected, long unexpected, short expected, and short unexpected. IS has (predominantly) expected messages, though the length of the unexpected queue can be long. As the number of nodes increases, the messages shift from long to short (due to the small, fixed problem size). In contrast, FT is split almost evenly between expected long messages and unexpected long messages. Collective protocols will clearly need to deal with long unexpected messages on a regular basis (not a common occurrence in standard collective benchmarks). Numerous codes (BT, CG, LU, and SP) have predominantly short collectives with some significant fractions of the short messages being unexpected. Again, unexpected messages must be handled by an offloading engine. Remarkably, despite the relatively large number of unexpected messages, the average search depth of the unexpected queue remains low.

B. Posted Receive Queues

Figure 3 shows the statistics of the posted receive queue usage for the NPB suite. Unlike the unexpected message queue, the maximum posted receive queue length is generally restrained for all but the IS benchmark. For the IS benchmark, the max queue length, the max search depth, and the average search depth all grow linearly with the number of processors. This clearly indicates a limitation in the implementation of the MPI_Alltoallv call used by the IS benchmark.

The comparison of Figure 3 and Figure 2 is striking. Collectives being used by other applications (not IS) in the benchmark suite clearly do not pre-post a large number of receives, but they do receive a large number of unexpected messages.

C. Collective Counts, Sizes, and Types

Table I indicates that the majority of the NAS parallel benchmarks make very limited use of collective operations. Only FT, IS, and MG make significant use of collectives. FT and IS both perform all of their communication through the use of `MPI_Alltoall` and `MPI_Alltoallv`. They also make use of `MPI_Reduce` and `MPI_Allreduce`, respectively. FT uses complex, double precision floating-point operations and performs 20 complex, double precision floating-point operations per node (40 total floating-point operations), per run. MG uses a significant number of `MPI_Allreduce` calls. Approximately half of them are integer operations and the other half are floating-point. A wide variety of run times can be seen in Table I. Time is shown in seconds with the range indicating the shortest and longest run-times over all numbers of nodes. The most interesting property of the collective operations for the NPB suite is that the number of collectives does not vary based on the size of the problem (class B or class C) or the number of nodes.

The maximum sizes of various collective operations (in bytes) are shown in Table II. The maximum is shown because it has the largest implications for requirements that are placed on an offloading network interface. Calls which have no data associated with them are not listed. In virtually all cases, the collective operations are small with their size being unrelated to the size of the problem or the number of nodes. For IS and FT, the total size of the collectives varies with the problem size and the size per node varies with the number of nodes.

V. Application Analysis

In addition to the benchmarks studied, three scientific applications in use at Sandia National Labs were studied. These applications were:

- *LAMMPS* — a classical molecular dynamics (MD) code designed to simulate systems at the atomic or molecular level[31], [32], [33]. A *Bead-Spring Polymer Chains* input deck and a *Lennard-Jones System* input deck were used for simulation.
- *CTH* — a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories. The *2 Gas* problem was used as one input[34] and two other examples from production runs were used as well.
- *Integrated TIGER Series (ITS)* — a suite of codes to perform Monte Carlo solutions of linear time-independent coupled electron/photon radiation transport problems. The ITS data is from an input deck used in a production run.

This set of applications and inputs is representative of three distinct types of codes in use at Sandia. Further details of these applications and input sets can be found in [30].

A. Unexpected Messages

Figure 4 indicates that the collectives in Sandia's applications behave very differently from those in the NPB suite (Figure 1). The growth of the maximum length of the queues is

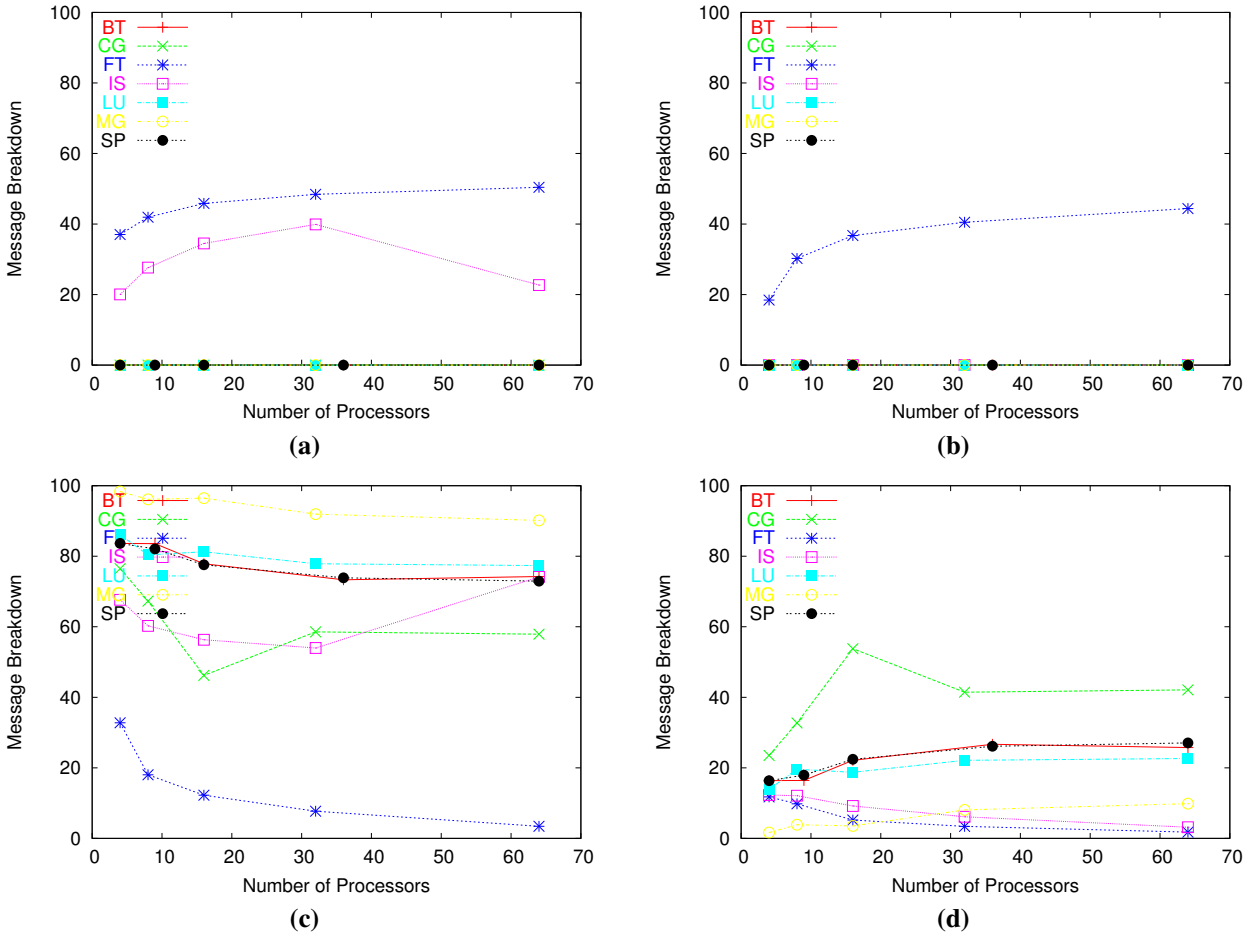


Fig. 2. NPB unexpected queue statistics breakdown, percentage of: **(a)** long expected messages, **(b)** long unexpected messages, **(c)** short expected messages, and **(d)** short unexpected messages

TABLE I

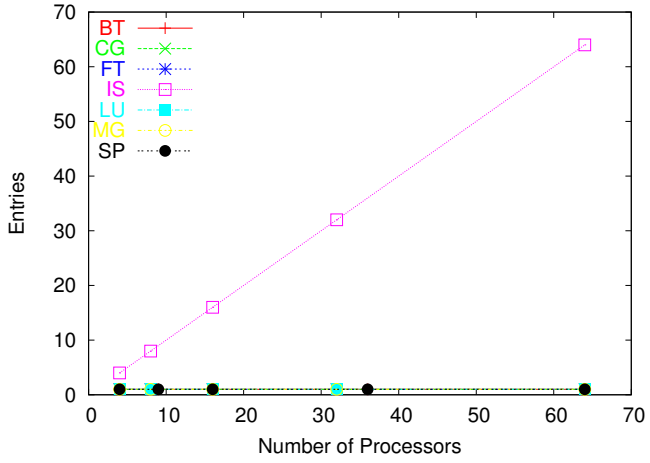
COLLECTIVE COUNTS FOR NAS PARALLEL BENCHMARKS

Benchmark	Barrier	Bcast	Reduce	Allreduce	Alltoall	Alltoallv	FP Ops	Int Ops	Time	Growth
BT	2	4	1	2	0	0	3	0	37-522	none
CG	1	0	1	0	0	0	1	0	13-202	none
FT	1	2	20	0	22	0	40	0	8-111	none
IS	0	0	2	11	11	11	1	192	1-6	none
LU	1	9	0	8	0	0	8	0	19-307	none
MG	6	6	1	88	0	0	49	40	1-14	none
SP	2	3	1	2	0	0	3	0	32-444	none

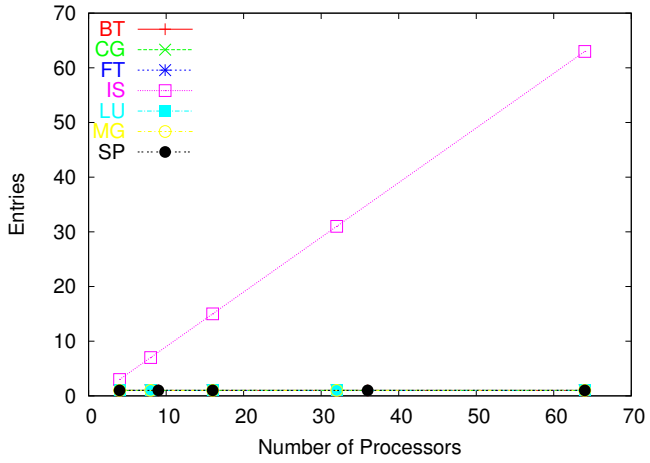
TABLE II

COLLECTIVE SIZES FOR NAS PARALLEL BENCHMARKS

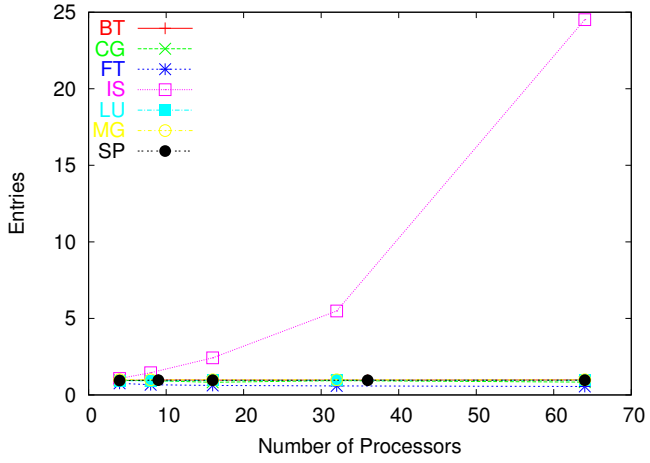
Benchmark	Bcast	Reduce	Allreduce	Alltoall	Alltoallv	Growth
BT	12	8	40	0	0	none
CG	0	8	0	0	0	none
FT	12	16	0	2097152	0	size
IS	0	8	4116	4	545900	size
LU	40	0	40	0	0	none
MG	32	8	16	0	0	none
SP	12	8	40	0	0	none



(a)

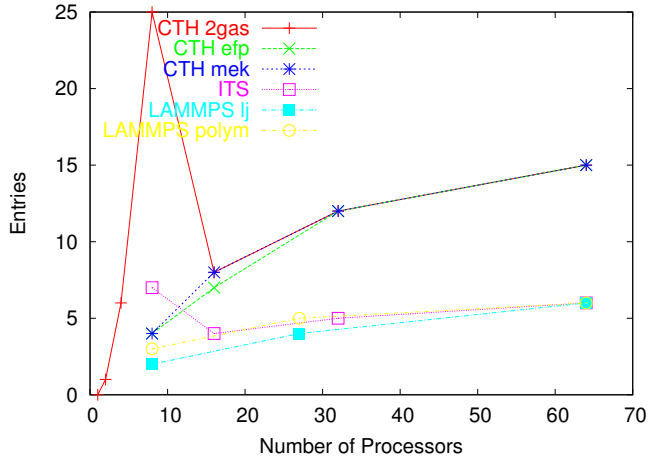


(b)

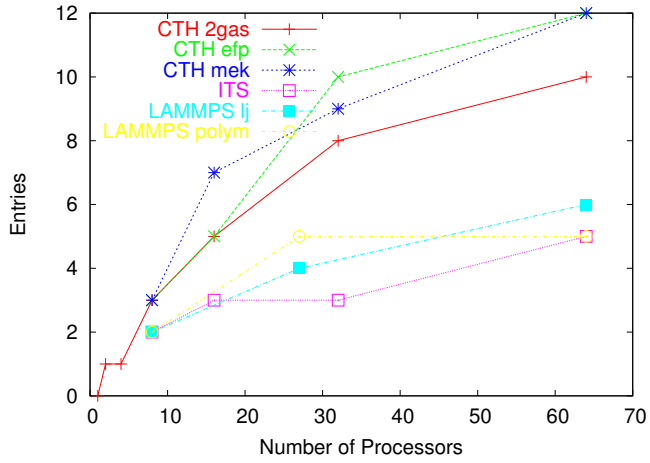


(c)

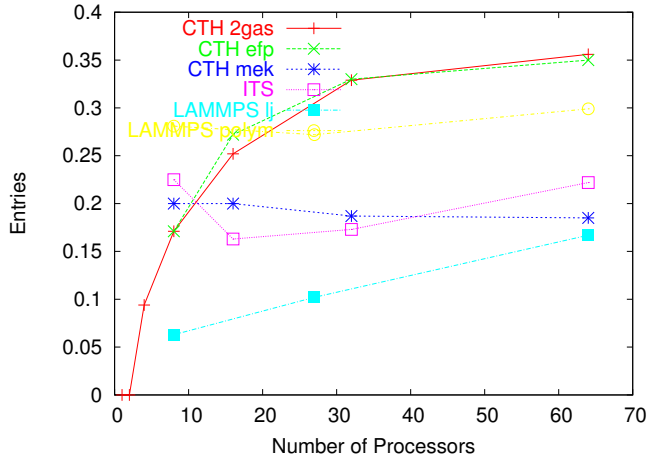
Fig. 3. NPB posted receive queue statistics: **(a)** max length, **(b)** max search depth, **(c)** average search depth



(a)



(b)



(c)

Fig. 4. Application unexpected queue statistics: **(a)** max length, **(b)** max search depth, **(c)** average search depth

closer to log_N, rather than linear. The maximum search depth grows in a similar pattern, but remains approximately a factor of two below the maximum length. The average search depth is trivially short. The primary difference from the NPB suite is that *all* of the applications have queues that grow with the number of nodes because all of the applications use collectives extensively. This reinforces the expectation that collective offload engines must deal with unexpected messages — likely because of load imbalance in the application causing different nodes to enter collectives at different times. Unlike point-to-point operations, there is no explicit way for an application to avoid unexpected messages from collective operations. That is, without a non-blocking collective API, an application has no way to pre-post receive buffers for messages associated with a collective operation.

The difference between collective behavior in Sandia's applications and the NAS parallel benchmarks is further highlighted in Figure 5. It shows that collective messaging is dominated by short, expected messages with no applications seeing the high number of long or unexpected messages seen in the NPB suite (Figure 2).

B. Posted Receive Queues

Since Sandia's applications do not use all-to-all style calls (see Table III), Figure 6 indicates that the posted receive queue is always short. Again, this is in contrast to the NAS parallel benchmarks, where some benchmarks grow the posted receive queue significantly. Combining this data with the data from Figure 4 indicates that there are some cases where a long posted receive queue could be created, but that this never happens. Instead, those messages arrive unexpectedly.

C. Collective Counts, Sizes, and Types

Table III highlights how dramatically different Sandia's applications are from the typical benchmarks shown in Table I. Applications tend to make much heavier use of `MPI_Allreduce`. They also make extensive use of both integer and double precision floating-point operations as part of these collectives. The common mode of operation for many scientific applications is to simulate a time step, perform some number of `MPI_Allreduce` calls, and perform the next time step. Thus, the number of collectives that are performed is dependent on the number of time steps that are simulated. Furthermore, Sandia's applications are designed for a variety of uses; thus, the number of collectives and the types of computations performed as part of the collective varies based on the particular problem being solved. As an additional note, unlike the NAS parallel benchmarks, Sandia's applications make no use of `MPI_Alltoall` or `MPI_Alltoallv`. Discussions with developers indicate that codes requiring an all-to-all communication write their own routine over point-to-point operations. Presumably, this is to avoid the poor performance of many all-to-all implementations, and/or possibly to take advantage of overlap. Numbers presented here were taken from 8 node runs but were consistent across run sizes.

The variance in the size of the collectives is also different between Sandia's applications and the NPB suite. For CTH, Table IV shows that the maximum size of the collectives often depends on the problem, but in other cases (LAMMPS, ITS) only the size of the (relatively infrequent) broadcast operations depends on the size of the problem. Reduction operations are of comparable size between Sandia's applications and the NPB suite. The broadcast operations, however, tend to be much larger for Sandia's applications.

VI. Non-blocking Collectives

Collective offload engines provide a unique capability: the ability to overlap collectives with computation. Traditional MPI implementations lack this capability because the application must remain in the MPI library to perform the messaging (and often computation). With the possible exception of handler routines such as those found in active messages[35], a collective offload engine is necessary to offer this capability. Unfortunately, the MPI interface to collectives is a blocking call. Calls such as `MPI_Allreduce`, `MPI_Alltoall`, `MPI_Alltoallv`, and `MPI_Barrier` require that all of the nodes enter the call and none can leave until the operation is (mostly) complete. Even `MPI_Reduce` and `MPI_Bcast` can require this on systems without a collective offload engine[1]. In explanation, broadcast and reduction tend to involve tree based communication patterns where non-leaf nodes must perform communication, and possibly computation, as the operation progresses through the tree. Even with an offload engine, the root of the reduction must wait for all nodes to complete.

An alternative that would enable applications to leverage the ability to overlap communication with computation is the design of a non-blocking collective API. A non-blocking (or split-phase, or two-phase) collective operation would have a call to initiate the collective (and contribute any data the local node might have) and a second call to block on the data. In between, the application could execute instructions that are not dependent on the result of the collective.

A split-phase variant of the MPI collective operations were proposed and debated for inclusion in the MPI-2 specification, but did not become part of the Standard. The final version of the API that was voted down is included in Chapter 7 of the MPI-2 Journal of Development [36]. IBM has implemented a non-blocking version of the MPI collective operations for their SP line of systems [37].

The question, however, is how well applications could use such non-blocking collectives. Is there a need for it? This section presents data which suggests that, indeed, it is possible for applications to leverage non-blocking collectives.

A. Execution Window

The data in Figure 7 indicates that in Sandia's applications, as they are currently written, a significant number of instructions could be overlapped by using a non-blocking collective operation. There are two parts to the figure. On the left, the distribution of the number of instructions that could be overlapped with a non-blocking collective is shown as a

[1]With an offload engine, leaf nodes in a reduction and the root node in a broadcast could contribute a result and return.

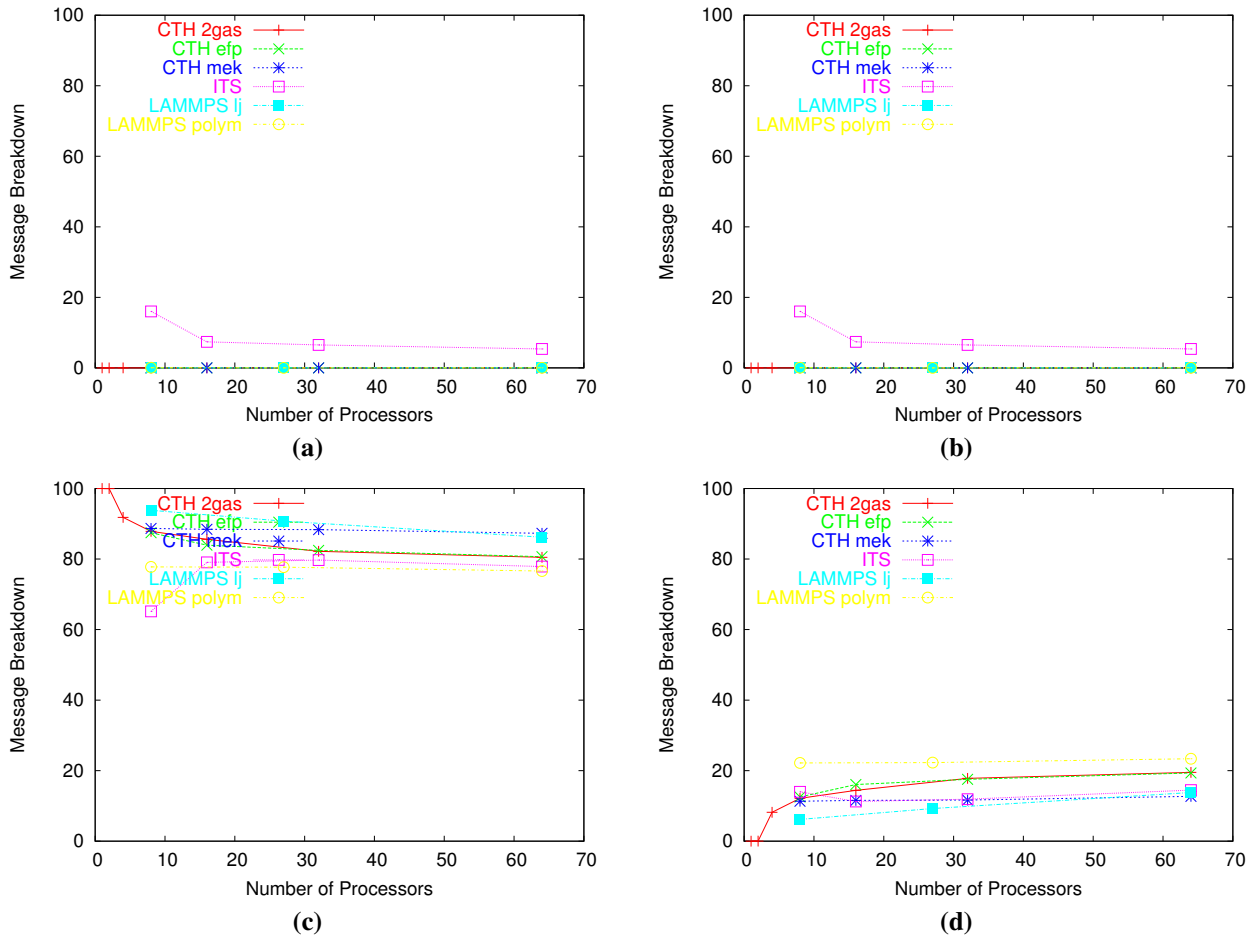


Fig. 5. Application unexpected queue statistics breakdown, percentage of: **(a)** long expected messages, **(b)** long unexpected messages, **(c)** short expected messages, and **(d)** short unexpected messages

TABLE III
COLLECTIVE COUNTS FOR APPLICATIONS

Benchmark	Barrier	Bcast	Reduce	Allreduce	FP Ops	Int Ops	Time	Growth
CTH 2Gas	1473	266	3	2325	540	1788	189-387	problem
CTH EFP	76221	2574	3	100134	21708	78429	1358-3779	problem
CTH MEK	58024	4270	3	134455	56418	78040	339-1627	problem
ITS SAT	2	24	0	0	0	0	733-1283	problem
LAMMPS LJ	6	53	0	29583	20086	9497	12-18	problem
LAMMPS Bead	10	332	0	60170	99	60071	677-1317	problem

TABLE IV
COLLECTIVE SIZES FOR APPLICATIONS

Benchmark	Bcast	Reduce	Allreduce	Growth
CTH 2Gas	256	4	8	problem
CTH EFP	256	4	112	problem
CTH MEK	256	4	304	problem
ITS SAT	13730320	0	0	none
LAMMPS LJ	56000	0	40	none
LAMMPS Bead	72208	0	40	none

minimum, mean, maximum, and median for each application. These are created by counting the number of instructions between the last touch of the data buffer used in the collective and the first read of that buffer after the collective. Instructions in the MPI library are not counted. The bars are broken into instructions before and instructions after the collective[2]. Most of the window of opportunity occurs before the collective. This is because applications tend not to call the blocking collective until the data is needed (rather than calling it when its input has been computed). For each of these measures, only collectives containing data were counted (i.e. barrier was excluded since it is unclear when the application is ready to call a barrier and when it needs the barrier to complete).

The right side of Figure 7 shows the distribution in another fashion — the percentage of collectives that have windows over a threshold size. The minimum threshold listed (1000 instructions) is a conservative estimate of the overhead that

[2]The minimum, maximum, and median windows were found and then those windows were decomposed.

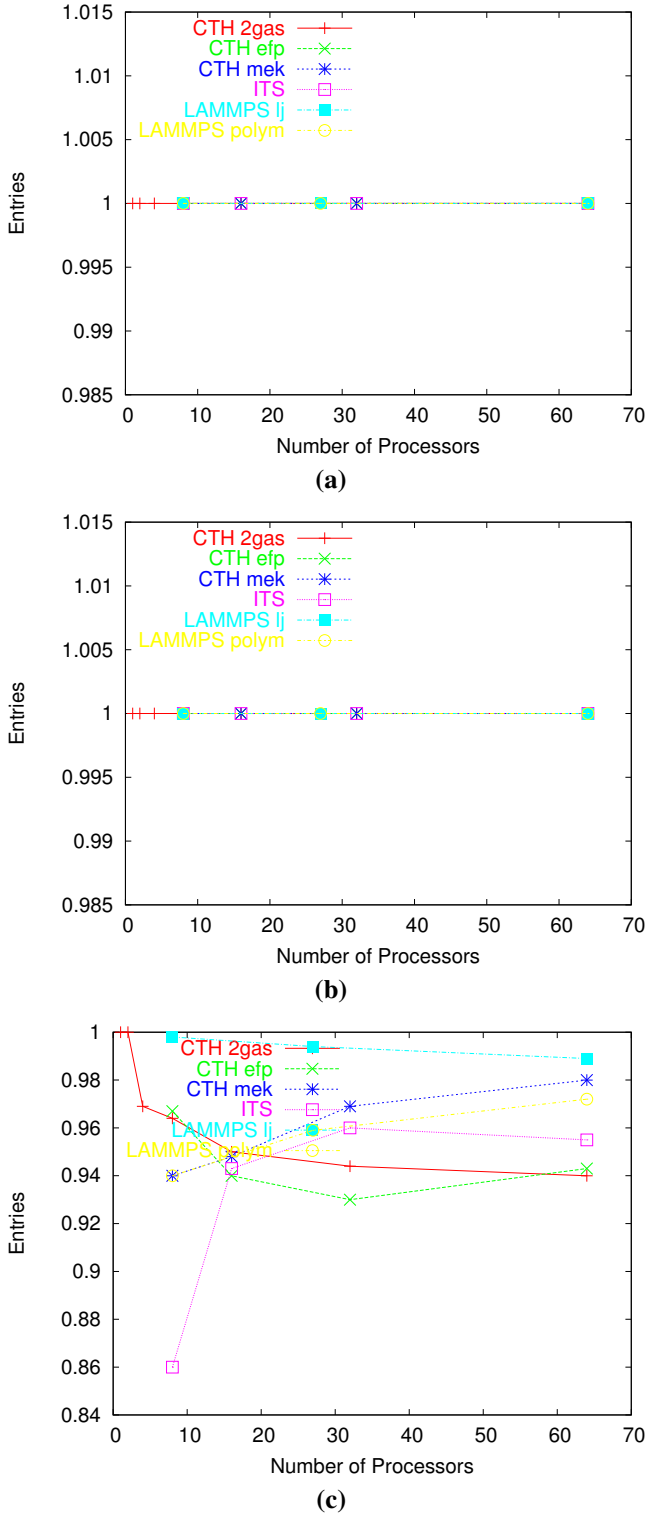


Fig. 6. Application posted receive queue statistics: **(a)** max length, **(b)** max search depth, **(c)** average search depth

might be introduced by an extra function call to create a non-blocking collective. CTH has particularly interesting characteristics with a relatively small percentage of collectives that can leverage a non-blocking interface, but massive instruction windows for those collectives. The window size for LAMMPS is much more dependent on the nature of the problem being executed, but at least one type of problem can leverage non-blocking collectives extensively.

Looking at the source code for CTH and LAMMPS reveals an interesting limitation of the methodology. All of the data presented in Figure 7 is extremely conservative. Two types of coding practices are present that thwart the measurement technique used. First, CTH abstracts the collective into a communication library. Some of the calls into the communication library immediately copy the collective result into a temporary return variable (thus, immediately touching the result buffer), but do not immediately use the return variable. In LAMMPS, blocking collectives lead to code structures that call the collective long before it is needed and use the result unnecessarily early. Further, temporary variables are used that again obscure the true data dependencies. Finally, LAMMPS has segments of code with long strings of independent collectives (all sharing a temporary variable) which could be overlapped. The current methodology does not count any of the MPI instructions, which causes it to miss the overlap potential.

When comparing Sandia's applications to the NAS parallel benchmark suite, the NPB suite appears significantly more amenable to this type of optimization. Unfortunately, the NPB suite makes very little use of collectives (Figure I). This is another instance where the NPB suite is shown to be less than ideal for future studies of collective behavior.

B. Data Consumption Rate

A second metric to consider for non-blocking collectives is the rate at which data is produced and consumed. A collective buffer which is filled (or drained) over a large number of instructions could be broken into several smaller collectives if a non-blocking interface to a collective offload engine was provided. Figure 8 examines the average production and consumption rate for buffers over 8 bytes long that are used in collectives. "Produce window" is the number of instructions required to fill the buffer while "produce total" is the time between first touching the buffer and calling the collective. "Consume window" is the number of instructions required to drain the buffer while "consume total" is the time between exiting the collective and touching the buffer for the last time.

This data indicates that splitting collectives into multiple smaller collectives could increase the size of the non-blocking opportunity window significantly. Further analysis will be needed to determine if breaking a single collective into multiple collectives will be beneficial since it may involve increasing the number of accesses to collective offload hardware.

VII. Conclusions

This paper addresses a key gap in the knowledge about how applications use collectives. Previous work [23], [24], [8], [22] expressed concerns over how collective operations might

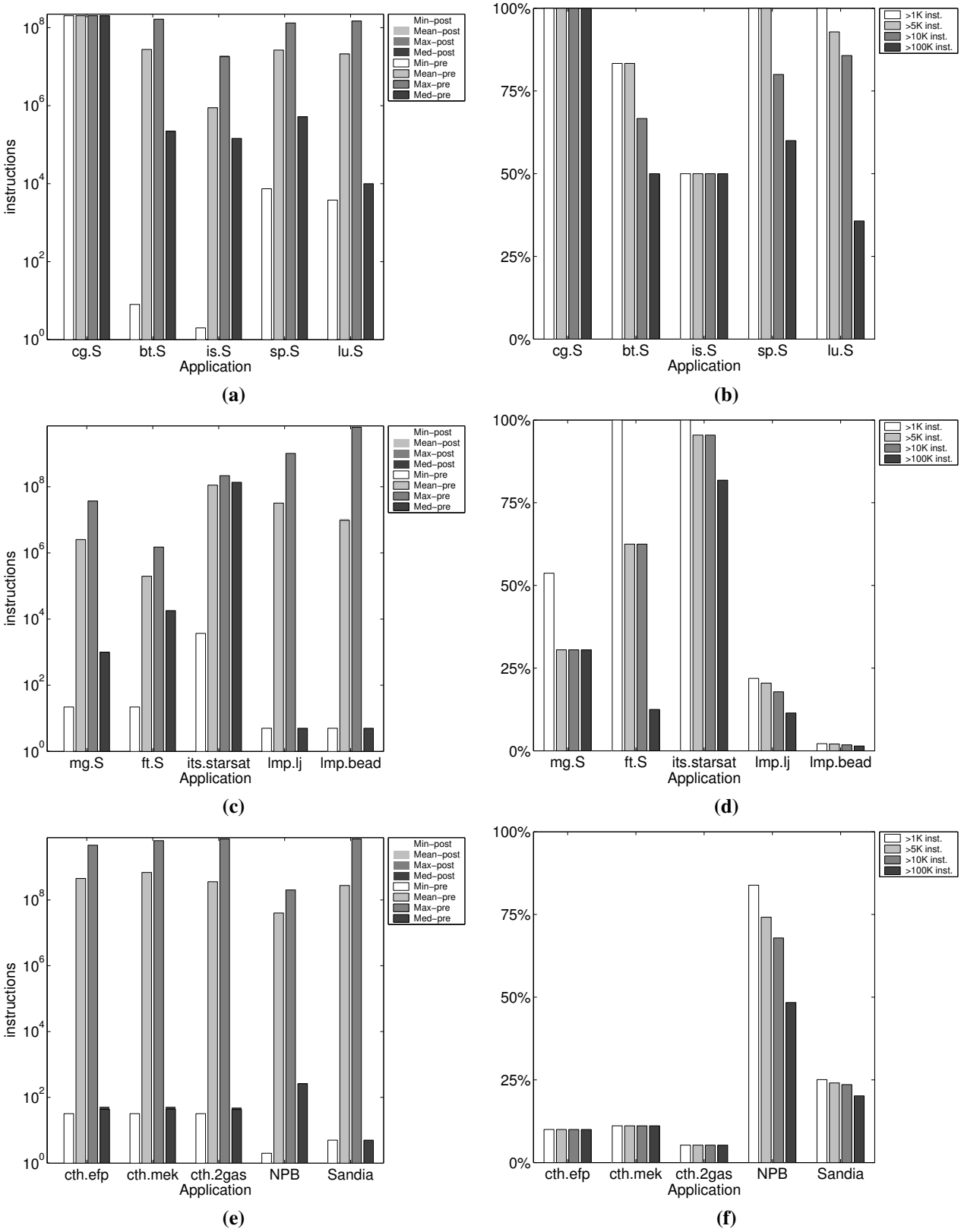


Fig. 7. **(a)**, **(c)**, **(e)**: Instructions available for overlap; **(b)**, **(d)**, **(f)**: percent of calls with sufficient threshold for overlap

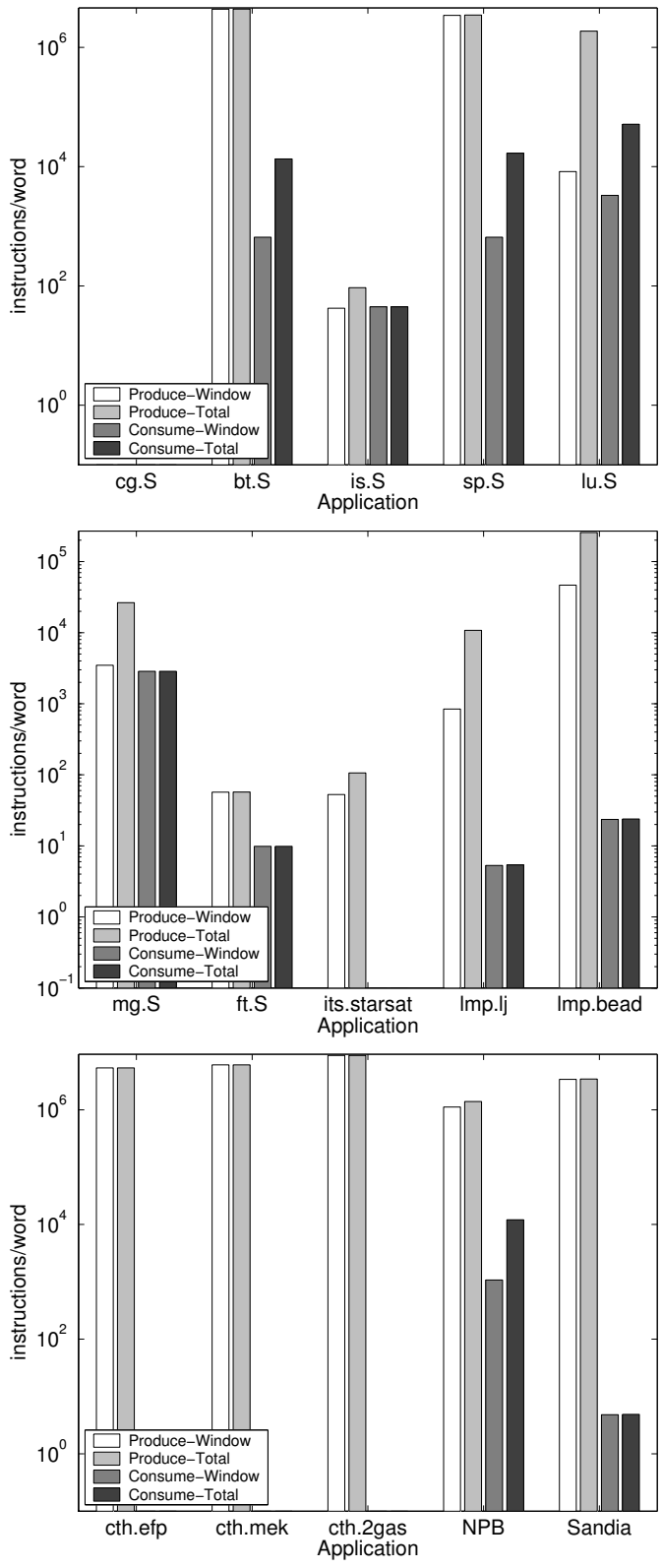


Fig. 8. Data consumption rate

be used. Those questions, such as the number of floating-point operations and how often are unexpected messages received, are answered here. Clearly, unexpected messages are common (Figure 4), but not overwhelming (Figure 5). Applications make extensive use of `MPI_Allreduce` and often `MPI_Barrier` and split their operations between floating-point and integer (Table III). Collectives can be large, but typically at the scale of a few hundred bytes (Table IV).

A second contribution of this paper is to compare the collective use of the NAS parallel benchmark suite to applications that Sandia is currently running. In general, the NPB suite proves to be a completely unsuitable environment for developing offload engines and analyzing collectives. The number and types of calls are completely different as are their relative importance. Even the queue behavior for the collectives that are called differs significantly.

Finally, this paper provides concrete data on the value of non-blocking collective operations. It can be seen (Figure 7) that there are a significant number of instructions that can be overlapped with collective operations. This number would be expected to grow if application developers had the option of designing and coding for non-blocking collectives. Even so, current code shows the opportunity to overlap from thousands of instructions (several microseconds) to hundreds of thousands of instructions (approaching a millisecond) with many of the collective calls. As the scale of machines continues to grow, this will become critically important to minimize the effectively serial portion of the code found in current blocking collective operations.

REFERENCES

[1] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.

[2] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Identifying and eliminating the performance variability on the ASCI Q machine," in *Proceedings of the 2003 Conference on High Performance Networking and Computing*, November 2003.

[3] R. R. Hoare and H. G. Dietz, "A case for aggregate networks."

[4] R. Hoare, H. Dietz, T. Mattox, and S. Kim, "Bitwise aggregate networks," in *Eigth IEEE Symposium on Parallel and Distributed Processing*, 1996, pp. 306–313.

[5] H. G. Dietz, T. M. Chung, and T. I. Mattox, "A parallel processing support library based on synchronized aggregate communication," in *1995 Workshop on Languages and Compilers for Parallel Computing*, Ohio State University, Ohio, August 1995.

[6] H. G. Dietz, R. Hoare, and T. I. Mattox, "A fine-grain parallel architecture based on barrier synchronization," in *International Conference on Parallel Processing*, August 1996.

[7] D. Buntinas, D. K. Panda, and P. Sadayappan, "Fast NIC-based barrier over Myrinet/GM," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

[8] D. Buntinas and D. K. Panda, "Nic-based reduction in myrinet clusters: Is it beneficial?" in *Proceedings of the SAN-02 Workshop (in conjunction with HPCA)*, February 2002.

[9] D. H. Bailey *et al.*, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[10] R. F. V. der Wijngaart, "NAS Parallel Benchmarks Version 2.4," Tech. Rep., October 2002.

[11] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott, "High performance synchronization algorithms for multiprogrammed multiprocessors," in *Proceedings of the fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, vol. 30, August 1995.

[12] J. B. Carter, C.-C. Kuo, and R. Kuramkote, "A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors," University of Utah, Salt Lake City, Utah 84112, Tech. Rep. UUCS-96-011, September 1996.
[13] E. D. Brooks III, "The butterfly barrier," *International Journal of Parallel Programming*, vol. 15, no. 4, pp. 295–307, 1986.
[14] U. Legedza and W. E. Weihl, "Reducing synchronization overhead in parallel simulation," in *Proceedings of Parallel and Distributed Simulation (PADS 96)*, 1996, pp. 160–169.
[15] A. Krishnamurthy and K. Yelick, "Optimizing parallel programs with explicit synchronization," in *Proceedings of the conference on Programming Language Design and Implementation (SIGPLAN 95)*, vol. 30, June 1995.
[16] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby, "Efficient algorithms for all-to-all communications in multi-port message-passing systems," in *Proceedings of the 6th annual ACM Symposium on Parallel Algorithms and Architectures*, Cape Cod, New Jersey, 1994.
[17] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, and R. van de Geijn, "Building a high-performance collective communication library," in *Proceedings of Supercomputing '94*, 1994, pp. 107–116.
[18] A. D. Knies, F. R. Barriuso, W. J. Harrod, and G. B. Adams III, "SLICC: A low latency interface for collective communications," in *Proceedings of Supercomputing '94*, 1994, pp. 89–96.
[19] R. S. Hyder and D. A. Wood, "Synchronization hardware for networks of workstations: Performance vs. Cost," in *Proceedings of the 1996 International Conference on Supercomputing*, Philadelphia, Pennsylvania, 1996.
[20] R. Gupta, P. Balaji, D. K. Panda, and J. Nieplocha, "Efficient collective operations using remote memory operations on VIA-based clusters," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.
[21] V. Tipparaju, J. Nieplocha, and D. K. Panda, "Fast collective operations using shared and remote memory access protocols on clusters," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.
[22] A. Moody, J. Fernandez, F. Petrini, and D. K. Panda, "Scalable NIC-based reduction on large-scale clusters," in *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.
[23] A. Wagner, D. Buntinas, R. Brightwell, and D. K. Panda, "Application-bypass reduction for large-scale clusters," in *Proceedings 2003 IEEE Conference on Cluster Computing*, December 2003.
[24] D. Buntinas, D. K. Panda, and R. Brightwell, "Application-bypass broadcast in MPICH over GM," in *Proceedings of The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, May 2003.
[25] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen, "Portals 3.0: Protocol building blocks for low overhead communication," in *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.
[26] R. Brightwell and K. D. Underwood, "An analysis of NIC resource usage for offloading MPI," in *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, Santa Fe, NM, April 2004.
[27] Apple Architecture Performance Groups, *Computer Hardware Understanding Development Tools 2.0 Referenc e Guide for MacOS X*, Apple Computer Inc, July 2002.
[28] N. J. Boden, D. Cohen, R. E. F. A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb. 1995. [Online]. Available: m10029.pdf
[29] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," in *Proceedings of Supercomputing Symposium*, 1994, pp. 379–386. [Online]. Available: `http://www.lam-mpi.org/download/files/lam-papers.tar.gz`
[30] R. Brightwell, S. Goudy, and K. D. Underwood, "On characterizing the network resource usage of mpi applications," *submitted*, May 2004.
[31] S. J. Plimpton, "Lammps web page," July 2003, http://www.cs.sandia.gov/ sjplimp/lammps.html.
[32] ——, "Fast parallel algorithms for short-range molecular dynamics," *Journal Computation Physics*, vol. 117, pp. 1–19, 1995.
[33] S. J. Plimpton, R. Pollock, and M. Stevens, "Particle-mesh ewald and rRESPA for parallel molecular dynamics," in *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, Mar. 1997.
[34] R. Brightwell, H. E. Fang, and L. Ward, "Scalability and performance of CTH on the Computational Plant," in *Proceedings of the Second International Workshop on Cluster-Based Computing*, May 2000.
[35] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proceedings of the 19th annual International Symposium on Computer Architecture*, May 1992, pp. 256–266.
[36] MPI Forum, "MPI-2 journal of development," http://www.mpi-forum.org/docs/mpi-20-jod.ps, July 1997.
[37] IBM, *Parallel Environment for AIX 5L V4.1 - MPI Programming Guide*, SA22-7945-00.

Ron Brightwell received the BS degree in mathematics in 1991 and the MS degree in computer science in 1994 from Mississippi State University. He is currently a principal member of technical staff at Sandia National Laboratories. 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 a member of the IEEE Computer Society and the ACM.

Sue P. Goudy is employed at Sandia National Laboratories, where she is an applications generalist for Scalable Systems Test and Integration Department. Ms. Goudy's current research interest is analytic model methodology for performance of large-scale parallel applications.

Arun Rodrigues received the Master's Degree from the University of Notre Dame in 2003 and the BS degree from the University of Notre Dame in 2001. He is currently a PhD candidate at the University of Notre Dame in Computer Engineering. His research concentrates on programming models for novel architectures, with a focus on large scale processing-in-memory systems for scientific computing. Aside from this, he works on low power microarchitectures, non-silicon architectures, and efficient architectural simulation.

Keith D. Underwood received the BS and PhD degrees in computer engineering from Clemson University in 1995 and 2002, respectively. He is currently a senior member of the technical staff at Sandia National Laboratories. His research interests include cluster computing, programmable network interfaces, and the role of reconfigurable computing in high performance computing systems. He is a member of the IEEE, IEEE Computer Society, and the ACM.