A Case for Non-Blocking Collective Operations

Torsten Hoefler[1,3], Jeff Squyres[2], Wolfgang Rehm[3], and Andrew Lumsdaine[1]

[1]Indiana University, Open Systems Lab, Bloomington, IN 47404 USA
{htor,lums}@cs.indiana.edu
[2]Cisco Systems, San Jose, CA 95134 USA
jsquyres@cisco.com
[3]Technical University of Chemnitz, Dept. of Computer Science, 09107 Chemnitz, Germany
{htor,rehm}@cs.tu-chemnitz.de

15th June 2006

DRAFT 15/6/2006

Abstract

Non-blocking collective operations for MPI have been in discussion for a long time. We want to contribute to this discussion and to give a rationale for the usage these operations and assess their possible benefits. A LogGP model for the CPU overhead of collective algorithms and a benchmark to measures it are provided and show a large potential to overlap communication and computation. We show that non-blocking collective operations can provide at least the same benefits as non-blocking point to point operations already do. Our claim is that actual CPU overhead for non-blocking collective operations depends on the message size and the communicator size and benefits especially highly scalable applications with huge communicators. We prove that the share of the overhead of the overall communication time of current blocking collective operations gets smaller with bigger communicators and larger messages. We show that the user level CPU overhead is less than 10% for MPICH2 and LAM/MPI using TCP/IP communication, which leads us to the conclusion that, by using non-blocking collective communication, ideally 90% idle CPU time can be freed for the application.

Keywords: Collective communication, Overlap, Non-blocking communication, Message passing (MPI)

1 Introduction

Non-blocking collective operations and their possible benefits have already been discussed at meetings of the MPI standardization committee. The final decision to not include them into the MPI-2 standard fell at March 6, 1997[1]. However, the fact that the decision was extremely marginal (11

[1]see: http://www.mpi-forum.org/archives/votes/mpi2.html

yes / 12 no / 2 abstain) suggests that the role of non-blocking collective operations is still debatable. Our contention is that non-blocking collective operations are a natural extension to the MPI-2 standard. We show that non-blocking collective operations can be beneficial for a class of applications to utilize the available CPU time more efficiently and decrease the time to solution of these applications significantly. Further, we discuss two main problems of blocking collective communication which limit the scalability of applications.

First, blocking collective operations have a more or less synchronizing effect on applications which leads to unnecessary wait time. Even thought the MPI standard does not define other blocking collective operations than MPI_BARRIER to be strictly synchronizing, most used algorithms force many processes to wait for other processes due to data dependencies. In this way, synchronization with a single process is enforced for some operations (e.g., a MPI_BCAST can not be finished until the root process called it) and the synchronizing behavior of other operations highly depends on the implemented collective algorithm. In either case, pseudo-synchronizing behavior often leads to many lost CPU cycles, a high sensitivity to process skew (e.g., due to daemon processes which "steal" the CPU occasionally and introduce a pseudo-random skew between processes [1, 2]), and a high sensitivity to imbalanced programming (e.g., some processes do slightly more computation than others each round).

Second, most blocking collective operations can not take much advantage of modern interconnects which enable communication offload to support efficient overlapping of communication and computation. Abstractly seen, each supercomputer or cluster consists of two entities, the CPU which processes data streams and the network which transports data streams. In many networks, both entities can act mostly independently of each other, but the programmer has no chance to use this parallelism efficiently if blocking communication (point-to-point or collective) is used.

Another rationale to offer non-blocking semantics for collective communication is an analogy between many modern operating systems and the MPI standard. Most modern operating systems offer possibilities to overlap computation on the host CPU with actions of other entities (for example harddisks or the network). Asynchronous I/O and non-blocking TCP/IP sockets are today's standard features to communicate. The MPI standard offers non-blocking point-to-point communication which can be used to overlap communication and computation. It would be a natural extension to offer also a non-blocking interface to the collective operations.

The next section describes related work in the field of overlap of computation and communication and the avoidance of synchronization. Section 2 gives some information about possible benefits of non-blocking collective communication. Section 3 presents benchmark results for a selected set of operations followed by an example of the application of non-blocking collective operations in Section 4. The last Section concludes this work and points out directions of further research.

1.1 Related Work

The obvious benefits of overlapping communication with computation and leveraging the hardware parallelism efficiently with the usage of non-blocking communication is well documented. Analyses [3, 4, 5] try to give an assessment of the capabilities of MPI implementations to perform overlapping for point-to-point communications. Many groups analyze the possible performance benefits for real applications. Liu et al. [6] showed possible speedups up to 1.9 for several parallel programs.

DRAFT 15/6/2006

Brightwell et al. [7] classifies the source of performance advantage for overlap and Dimitrov [8] uses overlapping as fundamental approach to optimize parallel applications for cluster systems. Other studies, as [9, 10, 11, 12] apply several transformations to parallel codes to enable overlapping. However, little research has been done in the field of non-blocking collectives. Studies like [13, 14] mention that non-blocking collective operations would be beneficial but do not provide a measure for it. Kale et al. [15] analyzed the applicability of a non-blocking personalized exchange to a small set of applications in practice. However, many studies show that non-blocking communication and non-blocking collectives *may* be beneficial. Our work contributes to the field because we actually assess the potential performance benefits of a non-blocking collective implementation.

2 Possible Performance Benefits

The most obvious benefits of non-blocking collective operations are the avoidance of explicit pseudo synchronization and the ability to leverage the hardware parallelism stated in Section 1. The pseudo-synchronizing behavior of most algorithms cannot be avoided, but non-blocking collective operations process the operation in the background, which enables the user to ignore most synchronization effects. Common sources for de-synchronization, process skew and load imbalance are not easily measurable. However, results can increase the application running time dramatically, as shown in [16]. Theoretical [17] and practical analyses [18, 16] show that operating system noise and resulting process skew is definitively influencing the performance of parallel applications using blocking collective operations. Non-blocking collective operations avoid explicit synchronization unless it is necessary (if the programmer wants to wait for the operation to finish). This enables the programmer to develop applications which are more tolerant of process skew and load imbalance.

Another benefit is to use the parallelism of the network and computation layers. Non-blocking communication (point-to-point and collective) allows the user to issue a communication request to the hardware, perform some useful computation, and ask later if it has been completed. Modern interconnect networks can perform the message transfer mostly independently of the user process. The resulting effect is that, for several algorithms/applications, the user can overlap the communication latency with useful computation and ignore the communication latency up to a certain extent (or totally). This has been well analyzed for point-to-point communication (see Section 1.1). Non-blocking collective operations allow the programmer to combine the benefits of collective communication [19] with all benefits of non-blocking communication. The following subsections analyze the communication behavior of current blocking collective algorithms and implementations and show that only a fraction of the CPU time is involved into communication related computation. In relation to previous studies we show, theoretically and practically, that a similar percentage, in many cases even more, idle CPU time as with non-blocking point-to-point communication can be gained. We assume that the biggest share of the remaining (idle) CPU time can be leveraged by the user if overlap of communication and computation together with non-blocking collective communication can be applied.

DRAFT 15/6/2006

2.1 Modelling CPU and Network Activity

This subsection gives an estimation of the theoretical CPU idle time during a collective operation. The CPU idle time during the communication will be modelled and benchmarked. Precise models for collective operations are presented in [20] and for barrier synchronization in [21]. Both studies show that the LogP [22] or LogGP [23] model is able to predict the communication time sufficiently accurately if the processes enter the collective operation simultaneously.

We analyze the three collective operations MPI_BARRIER, MPI_ALLREDUCE, and MPI_BCAST without loss of generality, in detail. As shown in [24, 25, 26], these three operations are frequently used in real applications. However, the results can also be applied to all other collective operations.

We assume the usual LogP/LogGP communication parameters and γ to assess computation:

L Network latency

o CPU overhead on sender and receiver side

g Network gap (the time to wait between two consecutive message injections)

G Gap per byte (bandwidth for bulk message transfers)

m Message size in bytes

P Number of involved processors

γ Time to compute 1 byte (fetch, compute, store)

We derive simplified LogGP models for networks adhering the properties defined in Section 2.2 in [21] (full bisectional bandwidth; full duplex; unlimited forwarding rate; L, o are constant; $o > L > g$). We model point-to-point message based implementations with logarithmic running time ($O(log_2P)$) of all three operations. We assume the dissemination principle to perform MPI_BARRIER (1), analyzed in [21]. Our model for MPI_ALLREDUCE (2) assumes a simple binomial tree reduce implementation followed by MPI_BCAST and our MPI_BCAST (3) model assumes a binomial tree implementation (compare proposed models in [20]).

$$t_{barr} = (2o + L) \cdot \lceil log_2P \rceil \quad (1)$$

$$t_{allred} = 2 \cdot (2o + L + m \cdot G) \cdot \lceil log_2P \rceil + m \cdot \gamma \cdot \lceil log_2P \rceil \quad (2)$$

$$t_{bcast} = (2o + L + m \cdot G) \cdot \lceil log_2P \rceil \quad (3)$$

If we come back to the two entities, which are the network and the processor, mentioned in Section 1, we realize that each parameter is "accounted" at a specific entity. The processor is only used by o and γ while the network is used to perform the message transmission (L,g,G). Using this information, we can divide the equations presented above up into processing and network parts:

$$t_{barr}^{CPU} = 2o \cdot \lceil log_2P \rceil \qquad t_{barr}^{NET} = L \cdot \lceil log_2P \rceil \quad (4)$$

$$t_{allred}^{CPU} = (4o + m \cdot \gamma) \cdot \lceil log_2P \rceil \qquad t_{allred}^{NET} = 2 \cdot (L + m \cdot G) \cdot \lceil log_2P \rceil \quad (5)$$

$$t_{bcast}^{CPU} = 2o \cdot \lceil log_2P \rceil \qquad t_{bcast}^{NET} = (L + m \cdot G) \cdot \lceil log_2P \rceil \quad (6)$$

DRAFT 15/6/2006

We see that both, t^{CPU} and t^{NET} scale logarithmically with P. However, on modern interconnects the parameters can differ significantly. The following section provides an analysis of these parameters for modern interconect networks.

2.2 Fitting the Model to Modern Architectures

Modern interconnect architectures, like InfiniBand™, Quadrics™, or Myrinet™, which are used for HPC systems, try to offload a huge share of the communication into the network interface card. Traditional networks, like Ethernet (without offloading), still use the CPU extensively to process network protocols like TCP/IP. However, also Ethernet has been optimized for lower host overhead with simplified protocols [27] as well as direct user level access and protocol offloading [28]. All these new networks and approaches aim to reduce the overhead of the main CPU involved in communication (o parameter). The L parameter is usually greater than the o in modern networks, and the gap between t^{CPU} and t^{NET} grows with the message size as $G \cdot m$ is added. This enables efficient overlapping of computation and communication for point-to-point communication which has been described in the related work section. However, this idea can also be applied to collective communication. As one can see in equations (4),(5),(6), the gap between Network and CPU occupancy also grows with the number of involved processors P. This leads us to the prediction that especially blocking collectives which communicate large data chunks with many processors should be mostly utilizing the network (with an idle CPU). The only exception could be reduction operations, like MPI_ALLREDUCE, because they include processing (reduction) of values on the host CPU. However, in most cases, the bandwidth of the CPU should be much higher than the network bandwidth. In the following section, we evaluate these theoretical expectations with a custom benchmark set which measures the CPU usage during blocking collective operations.

3 Benchmark Results

We implemented a benchmark which measures the CPU utilization for different MPI collective operations. The benchmark uses the standard `gettimeofday()` and `getrusage()` functionality of modern operating systems to measure the idle time. It issues collective calls with different message sizes and communicator sizes. The benchmark methodology is described as pseudocode in Listing 1. The `getrusage()` call returns system time and user time used by the running process separately. We chose a high number of iterations (10000) in the inner loop (`max_iter`, Line 6) to be able to neglect the overhead and relative impreciseness of the system functions. We conducted the benchmark for different MPI implementations shown in Table 1.

Many MPI libraries are implemented in a non-blocking manner which means that the CPU overhead is, due to polling, 100% regardless of other factors. Only LAM/MPI with TCP/IP and MPICH2 with TCP/IP used blocking communication to perform the collective operations. However, it is totally correct to use polling to perform blocking MPI collective operations because, at least for single threaded MPI applications, the CPU is unusable anyways and polling has usually slightly lower overhead than interrupt based (blocking) methods.

The difference for MPI_ALLREDUCE between MPICH2 and LAM/MPI is shown in Figure 1.

DRAFT 15/6/2006

```
for( proc=1; proc<nproc; proc=proc*2) {
 create_communicator(nproc, comm);
 for( size=1; size<maxsize; proc=proc*2) {
  gettimeofday(t1);
  getrusage(r1);
  for( i=0; i<max_iters; i++)
    MPI_Coll(comm, size, MPI_BYTE, ...)
  getrusage(r2);
  gettimeofday(t2);
 }
}
```

Listing 1: Benchmark Methodology (pseudocode)

Implementation	**Networks**
LAM/MPI 7.1.2	InfiniBand, TCP/IP
MPICH2 1.0.3	TCP/IP
Open MPI 1.1a3	InfiniBand and TCP/IP
OSU MVAPICH 0.9.4.	InfiniBand

Table 1: Tested MPI Implementations

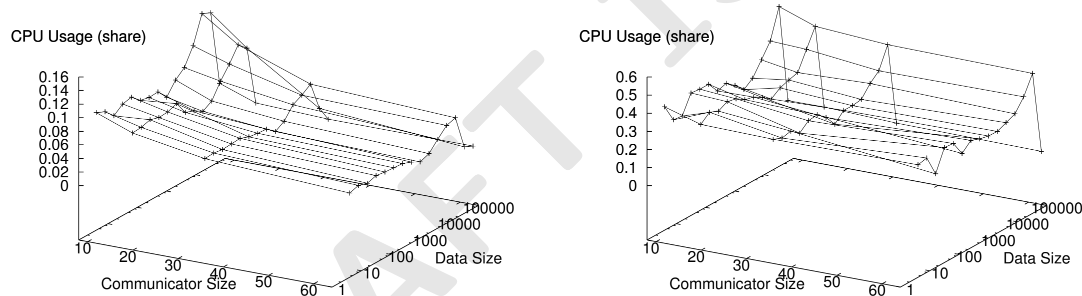

Figure 1: MPI_ALLREDUCE (user time + system time) overheads for LAM/MPI (left) and MPICH2 (right).

DRAFT 15/6/2006

Both exhibit a similar behavior and use only a fraction of the available CPU power for communicators with more than 8 nodes. MPICH2 causes in the average of all measurement points less than 30% CPU load while LAM/MPI consumes less then 10%. We see also that the data size plays an important role because there may be switching points in the collective implementation where the collective algorithms are changed (e.g., 128kb for MPICH2). However, this overhead includes the TCP/IP packet processing time spent in the kernel to transmit the messages which is measured with the `getrusage()` function as system time. User level, kernel-bypass, and offloading

communication hardware like InfiniBand, Quadrics or Myrinet does not use the host CPU to process packets and does not enter the kernel during message transmission. Figure 2 shows the user level CPU usage (without TCP/IP processing) for both examples from above. It shows that the

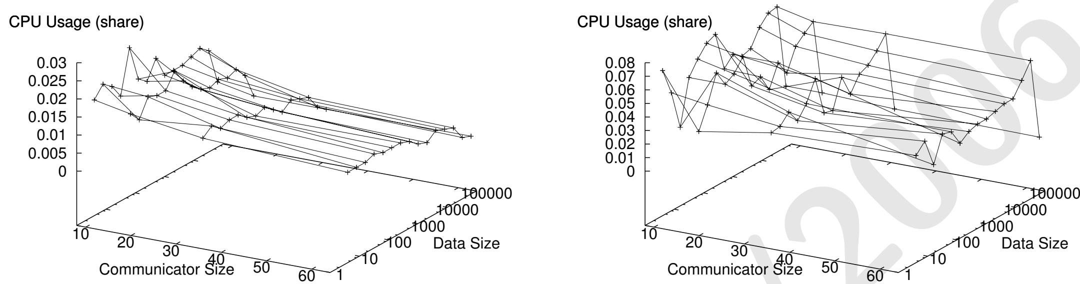

Figure 2: MPI_ALLREDUCE (user time) overheads for LAM/MPI (left) and MPICH2 (right).

CPU overhead for MPI_ALLREDUCE, which implies a user level reduction operation in our case, is below 10% in the average for MPICH2 and below 3% for LAM/MPI. These figures show also that the share of CPU idle time grows with communicator and data size. Other collective operations which are not shown here due to space restrictions exhibit a similar behavior.

However, generally speaking, the time to perform a collective operation grows also with communicator and data size. This means that the overall (multiplicative) CPU waste is even higher. Figure 3 shows the absolute CPU idle time of both implementations, several collective operations, and a fixed communicated data size with varying communicator sizes. The effect of growing CPU waste during blocking collectives is clearly visible. Especially the MPI_ALLTOALL operation, which usually scales worst, shows high CPU idle times with a growing number of participating processes.

Figure 4 shows that absolute CPU idle time of both implementations, for a fixed communicator size, and varying data sizes. The CPU waste is even higher and scales worse than for the varying communicator size, nearly linearly with the data size (the figures are plotted with a logarithmic scale).

4 An easy Example

We chose the solution of a system of n linear equations with Gaussian elimination (row partial pivoting) and backwards substitution as an easy example to demonstrate the applicability of non-blocking collective operations. We explain the application of non-blocking collective operations with an easy row-based data distribution scheme without loss of generality (other parallel distributions can use the same principle, but the resulting code is more complex). A pseudocode is shown in Listing 2.

DRAFT 15/6/2006

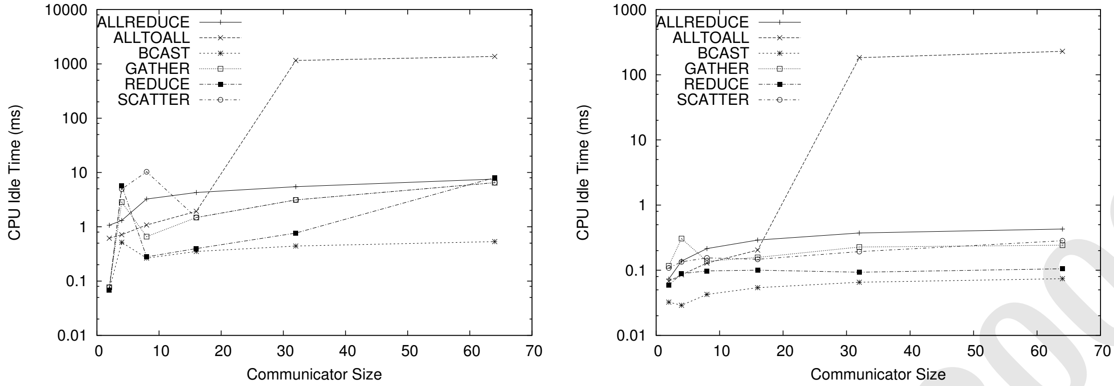

Figure 3: CPU idle time for some collective functions with varying communicator sizes for a constant data size of 1kB (left: LAM/MPI, right: MPICH2).

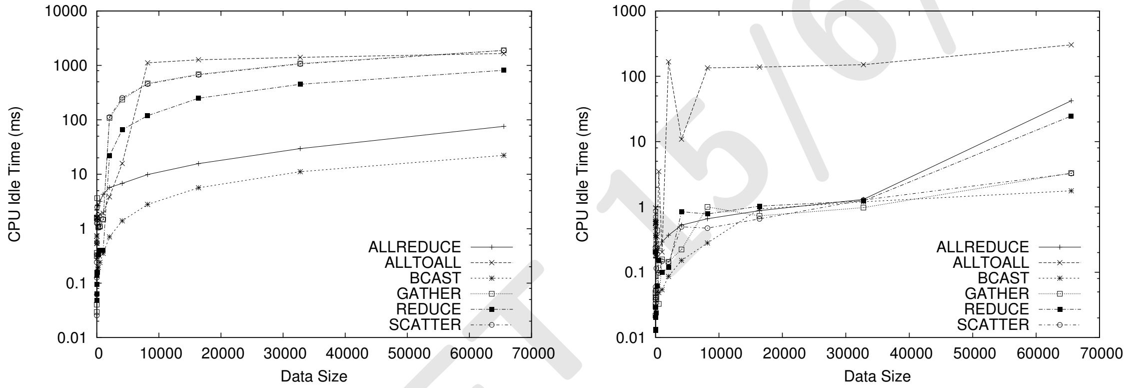

Figure 4: CPU idle time for some collective functions with varying data size for a constant communicator size of 16 processes (left: LAM/MPI, right: MPICH2).

DRAFT 15/6/2006

The simplest parallelization of this code needs two global communication operations. Both operations can be performed using MPI_BCAST. The node which owns the row that is currently processed in the outer loop has to find the pivot row (`piv`) and to broadcast it to all other processes. The second broadcast is necessary to distribute the multiplier (`mul`) of the innermost DAXPY ($a \times x + y$, Line 8) operation to all processes. The first MPI_BCAST (Line 3) is performed $n-1$ times and the second MPI_BCAST (Line 7) is called $(n-1)^2(n+1)/2$ times. This shows that the second MPI_BCAST dominates the communication time of the algorithm for huge systems.

This innermost collective communication can be efficiently overlapped with non-blocking collective operations using a well known double buffering method. Double buffering doubles the number of communication buffers so that one buffer can be involved in communication while the other buffer is used for computation. The two buffers are swapped (may be a single pointer exchange) at the end of each communication round. This enables communication to progress independently

```
for(k = 0; k < n - 1; k++) {
  if(IHaveColumn(k)) { piv = MaxCol(k); }
  MPI_Bcast(piv, 1, MPI_INT, rank_has_k, MPI_COMM_WORLD);
  exchange_row(piv, k);
  for(i = k + 1; i < n; i++) {
    if(IHaveColumn(k)) { mul = A[k][i] / A[k][k]; }
    MPI_Bcast(mul, 1, MPI_DOUBLE, rank_has_k, MPI_COMM_WORLD);
    daxpy(n, A[i], mul, 1, A[k], 1);
    b[i] -= mul * b[k];
  }
}
```

Listing 2: Solution of a system of linear equations (traditional)

of computation and vice versa. Listing 3 shows an easy way to leverage this parallelism for the innermost MPI_BCAST operation. This double buffering approach is usable for many parallel applications.

We use a 1-round look-ahead scheme where the first multiplier has to be computed and sent in a blocking manner in advance that the computation can start in the first round. The remaining multipliers are communicated one round in advance and the communication buffer is copied to the computation buffer at the end of each round. The non-blocking NBC_IBCAST[2] issues an communication request to the communication subsystem and returns directly to the caller. The communication subsystem may progress the operation independently and ideally, the operation is already finished when the application reaches the NBC_WAIT. However, the communication progress may depend on many factors (e.g., threaded asynchronous progress of MPI, hardware support, ...) which are not discussed here.

5 Conclusions

We show that the addition of non-blocking collective operations to the MPI-2 standard would be a natural extension to the existing interface. We model the potential performance benefit of overlapping communication with computation during collective operations. The model is proven and quantified with an extensive analysis of the CPU overhead for TCP/IP based networks. The results show clearly that, using TCP/IP, more than 70% of the CPU time is wasted in average during blocking collective operations. We assume that the gap is more than 90% for offloading based networks such as InfiniBand, Quadrics or Myrinet which do not process messages on the host CPU. Absolute measurements show the wasted time per collective which can easily be converted into wasted CPU cycles. These considerations lead to possible optimizations using non-blocking collective operations.

We propose a simple double buffering scheme to enable the use of non-blocking collective communication for existing parallel applications or algorithms.

[2]This is **not** MPI standardized, the prefix NBC stands for Non-Blocking Collectives

DRAFT 15/6/2006

```
for(k = 0; k < n - 1; k++) {
  if(IHaveColumn(k)) { piv = MaxCol(k); }
  MPI_Bcast(piv, 1, MPI_INT, rank_has_k, MPI_COMM_WORLD);
  exchange_row(piv, k);

  /* first element */
  if(IHaveColumn(k)) { mul = A[k][k+1] / A[k][k+1]; }
  MPI_Bcast(mul, 1, MPI_DOUBLE, rank_has_k, MPI_COMM_WORLD, handle);

  for(i = k + 1; i < n; i++) {
    if(IHaveColumn(k)) { nextmul = A[k][i+1] / A[k][k]; }
    NBC_Ibcast(mul, 1, MPI_DOUBLE, rank_has_k, MPI_COMM_WORLD, handle);
    daxpy(n, A[i], mul, 1, A[k], 1);
    b[i] -= mul * b[k];
    NBC_Wait(handle);
    mul = nextmul;
  }
}
```

Listing 3: Solution of a system of linear equations (non-blocking collective implementation)

We are going to implement a portable library (NBC) supporting non-blocking collective operations on top of MPI-1 and port scientific applications to use the new semantics. However, implementing collective semantics on top of MPI-1 cannot easily take advantage of special hardware features to support collective communication (e.g., a hardware barrier [29]). We are planning to move the non-blocking collective implementation into the extensible Open MPI collective framework [30] to enable hardware optimized non-blocking collectives.

A prototype of the NBC library is available at: `http://www.unixer.de/NBC/`.

References

[1] Adam Wagner, Darius Buntinas, Dhabaleswar K. Panda, and Ron Brightwell. Application-bypass reduction for large-scale clusters. In *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003)*, pages 404–411. IEEE Computer Society, December 2003.

[2] Paul Terry, Amar Shan, and Pentti Huttunen. Improving application performance on hpc systems with process synchronization. *Linux J.*, 2004(127):3, 2004.

[3] Costin Iancu, Parry Husbands, and Paul Hargrove. Hunting the overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society.

[4] J.B. White III and S.W. Bova. Where's the Overlap? - An Analysis of Popular MPI Implementations, 1999.

DRAFT 15/6/2006

[5] William Lawry, Christopher Wilson, Arthur B. Maccabe, and Ron Brightwell. Comb: A portable benchmark suite for assessing mpi overlap. In *CLUSTER*, pages 472–475. IEEE Computer Society, 2002.

[6] G. Liu and T.S. Abdelrahman. Computation-communication overlap on network-of-workstation multiprocessors. In *Proc. of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1635–1642, July 1998.

[7] Ron Brightwell and Keith D. Underwood. An analysis of the impact of mpi overlap and independent progress. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 298–305, New York, NY, USA, 2004. ACM Press.

[8] R. Dimitrov. *Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations.* PhD thesis, Mississippi State University, 2001.

[9] Pierre-Yves Calland, Jack Dongarra, and Yves Robert. Tiling on systems with communication/computation overlap. *Concurrency - Practice and Experience*, 11(3):139–153, 1999.

[10] Francoise Baude, Denis Caromel, Nathalie Furmento, and David Sagnol. Optimizing metacomputing with communication-computation overlap. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 190–204, London, UK, 2001. Springer-Verlag.

[11] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 58, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Tarek S. Abdelrahman and Gary Liu. Overlap of computation and communication on shared-memory networks-of-workstations. pages 35–45, 2001.

[13] Anshu Dubey and Daniele Tessera. Redistribution strategies for portable parallel fft: a case study. *Concurrency and Computation: Practice and Experience*, 13(3):209–220, 2001.

[14] Ron Brightwell, Rolf Riesen, and Keith D. Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *Int. J. High Perform. Comput. Appl.*, 19(2):103–117, 2005.

[15] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[16] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8, 192 processors of asci q. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, 15-21 November 2003, Phoenix, AZ, USA, CD-Rom*, page 55. ACM, 2003.

[17] Saurabh Agarwal, Rahul Garg, and Nisheeth Vishnoi. The impact of noise on the scaling of collectives: A theoretical approach. In *12th Annual IEEE International Conference on High Performance Computing*, Goa, India, December 2005.

[18] Terry Jones, Shawn Dawson, Rob Neely, William G. Tuel Jr., Larry Brenner, Jeffrey Fier, Robert Blackmore, Patrick Caffrey, Brian Maskell, Paul Tomlinson, and Mark Roberts. Im-

DRAFT 15/6/2006

proving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing*, page 10, November 2003.

[19] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[20] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance Analysis of MPI Collective Operations. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium, 4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS 05)*, Denver, CO, April 2005.

[21] Torsten Hoefler, Lavinio Cerquetti, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A practical Approach to the Rating of Barrier Algorithms using the LogP Model and Open MPI. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPP'05)*, pages 562–569, June 2005.

[22] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[23] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1995.

[24] Jeffrey S. Vetter and Frank Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 96, Washington, DC, USA, 2002. IEEE Computer Society.

[25] Ron Brightwell, Sue Goudy, Arun Rodrigues, and Keith Underwood. Implications of application usage characteristics for collective communication offload. *Internation Journal of High-Performance Computing and Networking*, 4(2), 2006.

[26] Rolf Rabenseifner. Automatic mpi counter profiling. In *42nd CUG Conference, CUG Summit 2000*, 2000.

[27] Torsten Hoefler, Mirko Reinhardt, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. Low overhead ethernet communication for open mpi on linux clusters. In *Submitted to EuroPVM'06*, 2006.

[28] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. Emp: zero-copy os-bypass nic-driven gigabit ethernet message passing. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 57–57, New York, NY, USA, 2001. ACM Press.

[29] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. Adding Low-Cost Hardware Barrier Support to Small Commodity Clusters. In *19th International Conference on Architecture and Computing Systems - ARCS'06*, pages 343–350, March 2006.

DRAFT 15/6/2006

[30] Jeffrey M. Squyres and Andrew Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, St. Malo, France, July 2004.

DRAFT 15/6/2006