Leveraging non-blocking Collective Communication in high-performance Applications

Torsten Hoefler, Peter Gottschling and Andrew Lumsdaine

Open Systems Laboratory
Indiana University
501 N. Morton Street
Bloomington, IN 47404 USA
{htor, pgottsch, lums}@cs.indiana.edu

Abstract

Although overlapping communication with computation is an important mechanism for achieving high performance in parallel programs, developing applications that actually achieve good overlap can be difficult. Existing approaches are typically based on manual or compiler-based transformations. This paper presents a pattern and library-based approach to optimizing collective communication in parallel high-performance applications, based on using non-blocking collective operations to enable overlapping of communication and computation. Common communication and computation patterns in iterative SPMD computations are used to motivate the transformations we present. Our approach provides the programmer with the capability to separately optimize communication and computation in an application, while automating the interaction between computation and communication to achieve maximum overlap. Performance results results with two model applications show more than 90% decrease in communication overhead, resulting in 16% and 21% overall performance improvements.

1 Introduction

Improving the performance of distributed-memory parallel scientific applications has long focused on optimizing two primary constituents, namely optimizing (single process or single thread) computational performance and optimizing communication performance. With more advanced communication operations (such as non-blocking operations), it has also become important to optimize the interactions between computation and communication, which introduces yet another dimension in which to optimize. Moreover, these advanced operations continue to grow even more sophisticated (with the advent, e.g., of non-blocking collective operations), presenting the programmer with yet more complexity in the optimization process. As systems continue to grow in scale, the importance of tuning along all of these dimensions becomes more important.

The most straightforward way to improve basic communication performance is to employ high-performance communication hardware and specialized middleware to lower the latency and increase the bandwidth. However, there are limits to the gains to overall application performance that can be achieved just by focusing on bandwidth and latency. Approaches that overlap computation and communication seek to overcome these limits by hiding the latency costs by performing communication and computation simultaneously.

Obtaining true overlap (and the concomitant performance benefits) requires hardware and middleware support, sometimes placing high demands on communication hardware. Some studies, e.g., [31], found that earlier communication networks did not support overlapping well. However, however, most modern

network interconnects perform communication operations with their own co-processor units and much more effective overlap is possible [26]. To take advantage of this parallelism, specific non-blocking semantics must be offered to the programmer. Many modern library interfaces (even outside of HPC) already offer such non-blocking interfaces, examples include asynchronous filesystem I/O, non-blocking sockets and MPI non-blocking point-to-point communication.

Another key component to efficient hardware utilization (and therefore to high performance) is the abstraction of collective communication. High-level group communications enable communication optimizations specific to hardware, network, and topology. A principal advantage of this approach is performance-portability and correctness [20].

Non-blocking collective operations for MPI, which combine the benefits of the two concepts discussed above, have been proposed and implemented in [28]. Several studies [24, 25] showed that the high performance of blocking collective operations can be effectively combined with overlap to achieve up to 35% application speedup. In addition, the use of non-blocking collective operations avoids data-driven pseudo-synchronization of the application [29]. Iskra et al. and Petrini et al. show in [32] and [38] that this effect can cause a dramatic performance decrease.

It is easy to understand how non-blocking collective operations mitigate the pseudo-synchronization effects and hide the latency costs. However, properly applying those techniques to real-world applications turned out to be non-trivial because code must often be significantly restructured to take full advantage of non-blocking collective operations. We learned in several application studies [24, 25] that using non-blocking collectives to achieve good computation and communication overlap is likely to be labor-intensive and error-prone–and may decrease code readability as well. The main goal of our current work is to overcome these difficulties by making non-blocking collective operations much more straightforward to use. In particular, we propose an approach for incorporating non-blocking collectives based on library-based transformations. Our current implementation uses C++ generic programming techniques, but the approach would be applicable in other languages through the use of other appropriate tools.

Overview In our paper we address in particular the following issues:

1. We show how scientific kernels can take advantage of non-blocking collective operations,
2. We show which code transformations can be applied and how they can be automated,
3. We analyze the potential benefits of such transformations, and
4. We discuss limitations and performance trade-offs of this approach.

Related Work Many scientists recognized the fact that overlapping communication and computation can yield a substantial performance benefit. Practical application performance has been shown to improve up to a factor of 1.9 [6, 35]. Dimitrov [15] explains the gains of overlapping on cluster systems. Kale et al. show the applicability of a non-blocking collective personalized communication for a set of applications in [33].

Several application studies have been conducted to analyze the possible benefits of overlapping for parallel applications. Sancho et al. [39] show a high potential of overlap for a set of scientific applications. Brightwell et al. [7] state clearly that many parallel applications could substantially benefit from non-blocking collective communication.

Possible transformations to parallel codes to enable overlapping have been proposed in many studies [1, 4, 8, 13]. However, none of them investigated transformations to non-blocking collective communication. Danalis et al. [13] even suggest replacing collective calls with non-blocking send-receive calls. This is clearly against the philosophy of MPI and destroys performance portability and many possibilities of optimization with special hardware support (cf. [30, 36]) completely. Our approach transforms codes that use

blocking collective operations and enables the use of non-blocking collective operations and thus combines the performance portability and machine-specific optimization with overlap and easy programmability.

Several languages, like Split-C [12], UPC [10], HPF [22] or Fortran-D [23], have compilers available that are able to translate high-level language constructs into message passing code.

2 Overlapping Collectives

Many parallel algorithms apply some kind of element-wise transformation or computation to large (potentially multi-dimensional) data sets. Sancho et al. show several examples for such "Concurrent Data Parallel Applications" in [39] and prognose a high overlap potential. Other examples are Parallel Sorting [11], Finite Element Method (FEM) calculations, 3D-FFT [2] and parallel data compression [40].

We chose a dynamic, data-driven application, parallel compression, as a more complex example than the simple static-size transformations that were shown in previous works. The main difference here is that the size of the output data can not be predicted in advance and strongly depends on the structure of the input data. Thus, a two-step communication scheme has to be applied to the problem.

In our example, we assume that the N blocks of data are already distributed among the P processing elements (PE) and each PE compresses its blocks by calling compress(). The data will finally be gathered to a designated rank. Gathering of the compression results must be performed in two steps where the first step collects the individual sizes of the compressed data at the master process, and determines so the parameters of the second step, the final data gathering. This naive scheme is shown in Listing 1.

```
my_size = 0;
for (i=0; i < N/P; i++) {
  my_size += compress(i, outptr);
  outptr += my_size;

}
gather(sizes, my_size);
gatherv(outbuf, sizes);
```

Listing 1: Parallel compression naive scheme

```
for (i=0; i < N/P; i++) {
  my_size = compress(i, outptr);
  gather(sizes, my_size);
  igatherv(outbuf, sizes, hndl[i]);
  if(i>0) waitall(hndl[i−1], 1);
}
waitall(hndl[N/P], 1);
```

Listing 2: Transformed compression

This kind of two-step communication process is common for dynamic data-driven computations (e.g., parallel graph algorithms [37]), making it rather challenging to be generated automatically by parallelizing compilers or other tools. None of the projects discussed in the related works section can deal with the data-dependency in this example and optimize the code for overlap. Thus, we propose a library-based approach that offers more flexibility and supports the dynamic nature of the communication.

It seems that two main heuristics are sufficient to optimize programs for overlap: First, the communication should be started as early as possible. This technique is called "early binding" [15]. Second, the communication should be finished as late as possible to give the hardware as much time as possible to perform the communication. In some communication systems, messages can overlap each other. So called communication-communication overlap [5, 35] can also be beneficial.

Manual Transformation Technique Listing 2 shows the transformed code of Listing 1. This simple scheme enables the communication of the n^{th} element to overlap with the computation of the $(n+1)^{\text{st}}$ element. The call gather(sizes, my_size) collects the local data size of all nodes into a single array sizes and igatherv(outbuf, sizes, hndl[i]) starts a non-blocking communication of the buffers, gathering the correct size from every PE. The non-blocking communication is finished with a call to waitall(hndl, num) that waits for num communications identified by handles starting at hndl.

However, our previous works involving overlap, such as optimization of a three-dimensional Poisson solver [24] or the optimization of a three-dimensional Fast Fourier Transformation [25] showed that this simple heuristic is not sufficient to achieve good overlap. The two main reasons for this have been found in a theoretical and practical analysis of non-blocking collective operations [28]. This analysis shows that the overlap of non-blocking collective operations is relatively low for small messages but grows with message-size. This is due to some constant CPU intensive overheads such as issuing messages or managing communication schedules and the faster "bulk" transfer [3] of larger messages. Furthermore it can be more beneficial to give every communication more time to complete before waiting for it. Another conflicting issue could be that some MPI implementations have problems to manage many outstanding requests [28] and this results in significantly degraded performance. We provide separate solutions to those problems in the following.

Loop Tiling The fine grained communication can be coarsened by loop tiling. This means that more computation is performed before a communication operation is started. Listing 3 shows such a transformation.

```
for (i=0; i < N/P/t; i++) {
  my_size = 0;
  for (j=i; j < i+t; j++) {
    my_size += compress(i*t+j, outptr);
    outptr += my_size;
  }
  gather(sizes, my_size);
  igatherv(outbuf, sizes, hndl[i]);
  if(i>0) waitall(hndl[i-1], 1);
}
waitall(hndl[N/P/t], 1);
```

Listing 3: Compression after loop-tiling

```
for (i=0; i < N/P; i++) {
  my_size = compress(i, outptr);
  outptr += my_size;
  if (i > w) waitall(hndl[i-w], 1);
  igather(sizes, my_size);
  igatherv(outbuf, sizes, hndl[i]);


}
waitall(hndl[N/P-w],w);
```

Listing 4: Compression with a window

Communication Window Allowing more than a single outstanding request at a time gives the opportunity to finish the communication later (possibly at the end) and allow communication/communication overlap. However, too many outstanding requests create large overhead and can slow down the application significantly. So, the number of outstanding requests must be chosen carefully. This transformation is shown in Listing 4. Both schemes can be combined efficiently to transform our example as illustrated in Listing 5.

```
for (i=0; i < N/P/t; i++) {
  my_size = 0;
  for (j=i; j < i+t; j++) {
    my_size += compress(i*t+j, outptr);
    outptr += my_size;
  }
  if (i > w) waitall(hndl[i-w], 1);
  gather(sizes, my_size);
  igather(outbuf, sizes, hndl[i]);
}
waitall(hnld[N/P/t-w], w);
```

Listing 5: Final compression transformation

Tuning the Parameters These schemes introduce a trade-off that is caused by the pipelined fashion of the transformed algorithm (cf. pipeline theory [21]). The speedup of a pipeline is usually limited by the start-up overhead needed to fill the pipe. Our two-stage pipeline has a start-up time of a single communication. In our special case, this overhead is not a start-up time at the beginning but rather a drain time at the end because the last operation can usually not be overlapped. This means that a high tiling factor can lead to higher drain times without overlap. Thus, the tiling factor has to be chosen big enough to allow for bulk transfers and efficient overlap but also no too big to cause high pipeline drain times. The window factor is important to give the single

communications more time to finish but a too high window might cause performance degradation in the request matching of the underlying communication system. Both factors have to be optimized carefully to every parallel system and application.

3 Programmer-directed Collectives Overlap

The previous section described several code transformation schemes to leverage non-blocking collective operations. However, our experiences show that applying those schemes manually is error-prone and time-consuming. Fully automatic transformation with the aid of a compiler will demand an extremely elaborated data dependency analysis to guarantee inter-loop independence and can simply not handle many cases. We propose a flexible generic approach that does not require external software but only a standard compliant C++ compiler. The basic idea is to separate the communication from the computation and rearrange them while optimizing tiling factor and window-size to get the highest possible performance. The resulting generic communication pattern represents the class of applications discussed in this paper.

The separation of communication and computation introduces two functors that have to be implemented by the programmer. The interplay between those functors is defined in the template library and can be parametrized in order to achieve the maximum performance. The two functors are called computation and communication.

computation(i) computes step i of an iterative parallel application. The input data can either be read in the object or generated on the fly. The results of the computation are written into a buffer object, which is selected by the template. A buffer object stores the computed data and acts as source and destination buffer for the communication() functor. The communication functor communicates the data in the buffer it is called with. The buffer object itself is not required to store actual data, it can contain references to some other containers to avoid copying. All functors and the buffer object are implemented by the application programmer. Our template library combines those building blocks to enable efficient pipelining. The elements are described in detail in the following.

Computation defines the parenthesis (or application) operator that is called for every input element by the template. A user-supplied buffer to store the output data is also passed to the function. Listing 6 shows the computation functor for our compression example.

```
class computation_t {
  void operator() (int i, buffer_t& buffer) {
    buffer.size += compress(i, buffer.ptr);
    buffer.ptr += buffer.size;
} } computation;
```

Listing 6: Computation functor

Communication defines the parenthesis operator and gets called with a buffer in order to communicate the buffer's contents. The computation functor for the compression is given in Listing 7.

```
class communication_t {
  void operator() (buffer_t& buffer) {
    gather(sizes, buffer.size);
    igatherv(outbuf, sizes, buffer.hndl);
} } communication;
```

Listing 7: Communication functor

Buffer is used to store computation results and to communicate them later. All communication book-keeping is attached to the buffer.

With the two functors at hand, communication overlap of real-world applications can be augmented by tiling and communication windows without modifying the user code. This is demonstrated in Listing 8.

```
std::vector<buffer> buffers[w];
pipeline_tiled_window(N/P, tile_size,
    computation, communication, buffers);
```

Listing 8: Templated function call

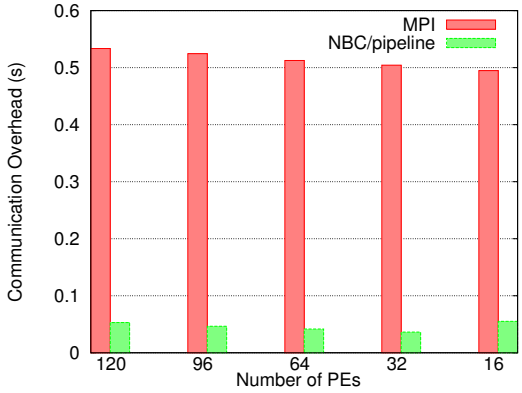

Figure 1: Communication overhead of the parallel compression using InfiniBand for different number of PEs (optimal tiling and window size).

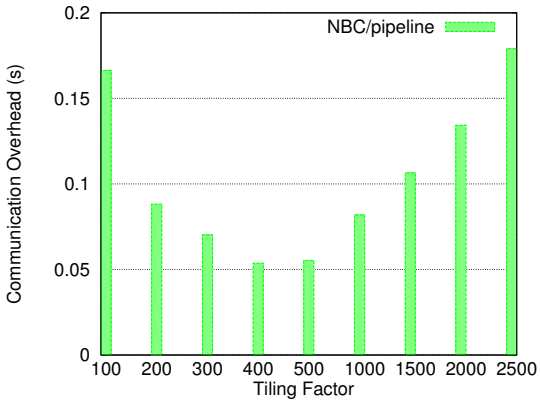

Figure 2: Communication overhead of parallel compression on 120 PEs depending on tile factor.

The function template pipeline_tiled_window is parameterized in order to tune the tile size—second function argument—and the number of windows. The latter is defined implicitly by the number of buffers (buffers.size()). In the remainder of the paper we demonstrate the performance impact of this tuning.

3.1 Performance Results

We implemented the example with our template transformation scheme to analyze the parallel performance. We used the g++ 3.4.6 compiler with Open MPI 1.2 [18] and the InfiniBand optimized version of LibNBC [27] to implement the MPI communication and libbzip2 for the data compression. All benchmarks have been executed on the "Odin" cluster at Indiana University. Odin consists of 128 nodes with dual dual core 2GHz Opteron 270 HE CPUs and 4 GB RAM per node. We used only a single processor per node to reflect our communication optimization. The nodes are interconnected with Mellanox InfiniBand^{TM} adapters and a single 288 port switch.

Figure 1 shows the communication overhead for the compression of 146 MiB random data on different number of PEs (strong scaling). The MPI graph shows the performance of the original untransformed code, the NBC/pipeline the optimized code using our template library.

Our benchmarks show that the communication overhead of the parallel compressions example can be significantly reduced with the application of our transformation templates and the subsequent use of non-blocking collective communication in a pipelined way. The compression time on 120 PEs was reduced from $2.18s$ to $1.72s$ which is an application performance gain of 21%. Figure 2 the influence of the tiling factor on 120 PEs.

4 Extended Example: Three-dimensional Fast Fourier Transform

A more complex example, the parallel implementation of three-dimensional Fast Fourier Transformations (FFTs), has been studied by many research groups [2, 9, 16, 19]. The application of non-blocking communication resulted in controversial results. The optimized collective all-to-all communication was replaced by a non-blocking point-to-point message pattern. Even if this enabled overlap, the communication was not optimized for the underlying architecture and different effects, such as network congestion, decreased the performance. Dubey et al. [16] mention the applicability of a non-blocking all-to-all, but they implemented a point-to-point based scheme and they did not see a significant performance improvement. Calvin et al. showed performance benefits using the same scheme [9]. However, the test-cases of their study used

```
transform all line in z−direction
for all z−planes {
  transform all lines in x−direction
  parallel transpose from x−distribution
    to y−distribution /∗ all−to−all ∗/
}
for all y−lines {
  transform line in y−direction
}
```

Listing 9: Pseudo-code of 3D-FFT

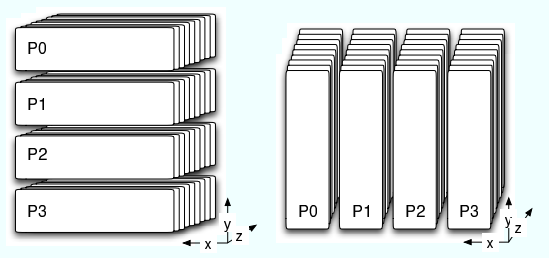

Figure 3: Block distribution in y-direction (left) and x-direction (right)

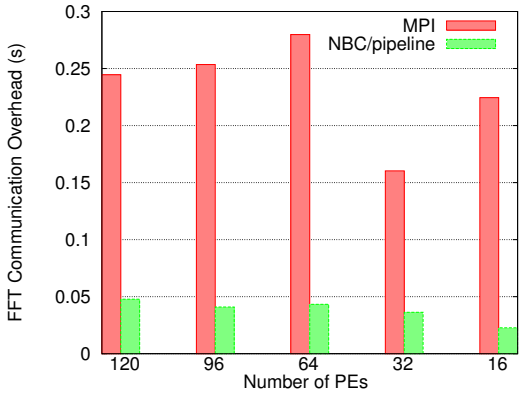

Figure 4: Communication overhead of 3D-FFT for different number of PEs (optimal tiling and window size)

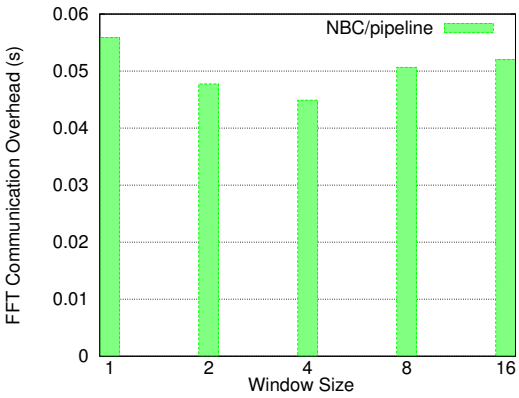

Figure 5: Impact of the window size for a (optimal) fixed tiling factor to communication overhead on 120 PEs

only four compute nodes which limited the impact of congestion dramatically. We use the optimized non-blocking collective communication operations provided by LibNBC [28] to avoid congestion and ensure proper scaling and overlap.

The three-dimensional FFT can be decomposed into three sweeps of one-dimensional FFTs, which are performed using FFTW [17] in our implementation. The data can be distributed block-wise such that the first two sweeps are computed with the same distribution. Between the second and third sweep the data must be migrated within planes of the cuboids. The computation scheme is depicted on a very high level in Listing 9 and Figure 3.

The n^3 complex values are initially distributed block-wise in y-direction (our transformation schemes can be used to redistribute the data if this is not the case). The first step consists of n^2 1D-FFTs in z-direction. The calculation in x-direction can still be performed with the same distribution. Before computing the FFT in the y-direction the data in every z-plane must be redistributed, ideally with an all-to-all communication. Performing the calculations within the z-planes in two steps establishes a loop with independent computation and communication so that our pipelining schemes can be applied.

We applied our generic scheme to the parallel 3D-FFT. The computation() functor performs a single serial one-dimensional transformation with FFTW and packs the transformed data to a buffer. The communication() uses LibNBC's non-blocking NBC_Ialltoall to initiate the communication. The "wait" function unpacks the received data from the buffer object into the FFT-buffer.

The benchmark results of a weak scaling 3D-FFT (720^3, 672^3, 640^3, 480^3 and 400^3 double complex values on 120, 96, 64, 32 and 16 PEs respectively) using InfiniBand on Odin are shown in Figure 4. The communication overhead in this more complex example can also be reduced significantly. Our optimizations were able to improve the running time of the FFT on 120 PEs by 16% from $2.5s$ to $2.1s$.

We demonstrate the same performance gain for the parallel FFT as we showed for the parallel compression. Combining window and tiling leads to the highest improvement. Figure 5 shows the communication overhead with regards to the window size (for a fixed optimal tiling).

5 Conclusions and Future Work

Although non-blocking collective communication operations offer significant potential for improving application performance, they need to be used appropriately within applications to actually provide performance improvements. In this paper, we presented different programming patterns to enable efficient overlapping of collective communication for a wide class of scientific applications. Applying non-blocking collective communication with our approach allowed applications to scale far beyond the parallelism of simple non-blocking point-to-point communication schemes.

The implementation of our transformation scheme as a generic library function template allows these techniquest to be used without requiring the programmer to explicitly encode them. Compression and FFT benchmarks using our template demonstrate a significant decrease of communication overhead for both applications. The communication overhead was reduced by more than 92% for the parallel compression and 90% for the parallel 3D-FFT which led to an application speedup of 21% and 16% respectively.

The source-code is available at the LibNBC webpage: `http://www.unixer.de/NBC`.

Future work includes the design of similar transformation schemes for loops where dependencies on remote data elements exist in a single loop. An example is the class of "pipelined algorithms" [34] or "pipelined data parallel applications" [39] where each process computes on its own grid elements and the data is pipelined through the processors. Therefore, the process must wait for the new data to arrive before it can proceed with the computation. Other application patterns exist, like LU factorization [14], where the "multiplier" must be broadcasted to all ranks before the computation can start.

Acknowledgments

The authors thank Douglas Gregor for many helpful comments. This work was supported by a grant from the Lilly Endowment, National Science Foundation grant EIA-0202048 and a gift the Silicon Valley Community Foundation on behalf of the Cisco Collaborative Research Initiative and a grant by the Saxon Ministry of Science and the Fine Arts.

References

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

[2] A. Adelmann, W. P. Petersen A. Bonelli and, and C. W. Ueberhuber. Communication efficiency of parallel 3d ffts. In *High Performance Computing for Computational Science - VECPAR 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004, Revised Selected and Invited Papers*, volume 3402 of *Lecture Notes in Computer Science*, pages 901–907. Springer, 2004.

[3] 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.

[4] 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.

[5] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, and Katherine Yelick. An Evaluation of Current High-Performance Networks. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 28.1, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (CAC 06)*, April 2006.

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

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

[9] C. Calvin and F. Desprez. Minimizing communication overhead using pipelining for multidimensional fft on distributed memory machines. In *Proceedings of the International Conference on Parallel Computing '93, Advances in Parallel Computing. North Holland, 1993.*, 1993.

[10] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. "a performance analysis of the berkeley upc compiler", 2003.

[11] Mark J. Clement and Michael J. Quinn. Overlapping computations, communications and i/o in parallel sorting. *J. Parallel Distrib. Comput.*, 28(2):162–172, 1995.

[12] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in split-c. In *Supercomputing*, pages 262–273, 1993.

[13] 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.

[14] F. Desprez, J. Dongarra, and B. Tourancheau. Performance Study of LU Factorization with Low Communication Overhead on Multiprocessors. *Parallel Processing Letters*, 5(2):157–169, 1995.

[15] 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.

[16] 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.

[17] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[18] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[19] S. Goedecker, M. Boulet, and T. Deutsch. An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes. *Computer Physics Communications*, 154:105–110, August 2003.

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

[21] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1st edition, 1990.

[22] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, High Performance Fortran Forum, Houston, Tex., 1993.

[23] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for fortran d on MIMD distributed-memory machines. In *Supercomputing*, pages 86–100, 1991.

[24] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. 6 2007. To be published in the Elsevier Journal of Parallel Computing (PARCO).

[25] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine. A Case for Standard Non-Blocking Collective Operations. 10 2007. accepted for publication at the EuroPVM/MPI 2007.

[26] T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 03 2007.

[27] T. Hoefler and A. Lumsdaine. Optimizing non-blocking Collective Operations for InfiniBand. 04 2008. Accepted for publication at the CAC 2008 in conjunction with the IPDPS 08.

[28] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. 11 2007. Accepted for publication at the Supercomputing 2007 (SC07).

[29] T. Hoefler, J. Squyres, W. Rehm, and A. Lumsdaine. A Case for Non-Blocking Collective Operations. In *Frontiers of High Performance Computing and Networking - ISPA 2006 Workshops*, volume 4331/2006, pages 155–164. Springer Berlin / Heidelberg, 12 2006.

[30] 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.

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

[32] Kamil Iskra, Pete Beckman, Kazutomo Yoshii, and Susan Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Proceedings of Cluster Computing, 2006 IEEE International Conference*, 2006.

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

[34] C. T. King, W. H. Chou, and L. M. Ni. Pipelined data parallel algorithms-i: Concept and modeling. *IEEE Trans. Parallel Distrib. Syst.*, 1(4):470–485, 1990.

[35] 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.

[36] J. Liu, A. Mamidala, and D. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. Technical report, OSU-CISRC-10/03-TR57, 2003.

[37] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[38] 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.

[39] Jose Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, and Kei Davis. Mpi tools and performance studies—quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 125, New York, NY, USA, 2006. ACM Press.

[40] M. E. Gonzalez Smith and J. A. Storer. Parallel algorithms for data compression. *J. ACM*, 32(2):344–373, 1985.