John von Neumann Institute for Computing

An Automated Approach to Improve Communication-Computation Overlap in Clusters

L. Fishgold, A. Danalis, L. Pollock, M. Swany

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 481-488, 2006.

An Automated Approach to Improve Communication-Computation Overlap in Clusters

Lewis Fishgold[a], Anthony Danalis[a], Lori Pollock[a], Martin Swany[a]

[a]Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716
{fishgold,danalis,swany,pollock}@cis.udel.edu

Applications that execute on parallel clusters face scalability concerns due to the high communication overhead that is usually associated with such environments. Modern network technologies that support Remote Direct Memory Access (RDMA) can offer true zero copy communication and reduce communication overhead by overlapping it with computation. For this approach to be effective though, the parallel application using the cluster must be structured in a way that enables communication computation overlapping. Unfortunately, the trade-off between maintainability and performance often leads to a structure that prevents exploiting the potential for communication computation overlapping. This paper describes a source-to-source optimizing transformation that can be performed by an automatic (or semi-automatic) system in order to restructure MPI codes towards maximizing communication-computation overlapping.

1. Introduction

Clusters of workstations are in common use among engineers and domain scientists due to their high processing power to cost ratio. The major drawback of cluster-based parallel computing as compared to shared memory multiprocessors is the network delay induced by the node interconnecting technology of clusters. Several interconnection technologies such as Myrinet, Quadrics and Infiniband can improve cluster message-passing performance by providing specialized low latency, high bandwidth networks for clusters. Such technology can theoretically reduce communication latency by overlapping communication with computation through handling network traffic solely on a network co-processor, freeing the CPU to perform useful computations.

Unfortunately, many existing scientific applications follow a modular structure where the computation is separated from the communication. Although such an approach makes the code easier to maintain and alter, it prevents communication-improving network technology from being fully utilized.

To overcome the restrictions imposed by such overlap-naïve code, a program can be transformed so as to aggressively send data as soon as it is generated. In particular, the computationally expensive part of many scientific applications consists of a loop (commonly with multiple levels of nesting) that executes some basic computation kernel. The suggested transformation aims to achieve “pre-pushing” by performing the communication within the computation loop using non-blocking, asynchronous I/O operations to transfer data elements among the parallel tasks as soon as it is safe. To evaluate the potential of this transformation, Danalis et al. [3] transformed potentially benefiting applications manually and experimented with the resulting variations to study the performance gains. Their results show that near maximum communication-computation overlap can be achieved, resulting in reduction of the communication overhead and significant performance improvement in comparison to the original code, as shown in Figure 1.

Although the suggested pre-push transformation can be performed by an experienced programmer, there are several reasons to build an automated system.

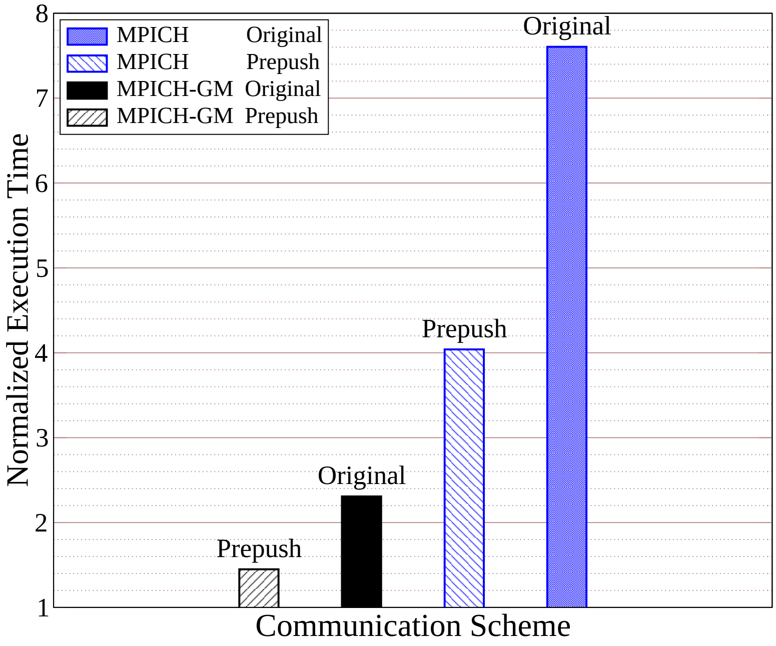

Figure 1. Performance improvement achieved by "pre-pushing"

- Asynchronous communication can be error prone and difficult to program, particularly when many processors and corresponding outstanding messages are involved creating a need for explicit synchronization.

- The performance of the transformed code depends on several cluster and application related parameters. These parameters have to be recomputed (or rediscovered through extensive profiling) every time the code changes, or the cluster CPUs, memory, or network changes.

- The suggested transformations have a negative impact on the maintainability of the code and in the case where low level communication primitives (eg., Myrinet's GM) are used, portability is also affected.

- Having an automated system perform the transformation opens the optimization to a wider audience of applications such as legacy codes, and those whose programmers are unaware of the details of the optimization.

Significant research has focused on optimizing communication latency in cluster environments but none can handle explicitly parallel codes written using MPI. Many compiler or language-based techniques translate higher-level parallel constructs into message passing primitives as appropriate. Examples of this include UPC [4], Co-Array Fortran [11], HPF [6], and Fortran-D [7]. While these approaches allow programmers to write their code in SPMD style, they focus on parallel optimization in the large, rather than focusing on optimization of messaging on a single host and do not deliver the performance that can be achieved by carefully tuned, manually parallelized applications. Systems such as Polaris [2] and PARAMAT [9] perform source-to-source transformations to achieve parallelization of serial programs that are written in *Fortran 77* or *C* without any special annotation. Nevertheless, these systems do not accept input code already parallelized with the use of MPI, but rather expect code written as a serial program. Projects such as CC-MPI [8] attempt to extend the standard MPI in order to provide support for the *compiled communication* model [16]. In this way, communications that lend themselves to static analysis can be separated from those which do not, and optimizations can be performed as appropriate. The main difference between our project and the alternative solutions is that we aim to offer a complete system able to automatically (or semi-automatically) restructure parallel applications, explicitly parallelized with the use of MPI, in order to minimize their communication overhead by performing communication-computation overlapping.

2. Communication-Computation Overlapping Transformation

The modular structure of many scientific codes, in which some data is computed, stored in an array, and then sent over the network, leaves no opportunity for communication-computation overlap. Often, as soon as the data is ready to be sent, it needs to be used (by the receivers). We propose a transformation for such codes so that the data is pre-pushed, or sent as it is generated, before it is needed.

To achieve such an early transfer model, the computation loop is restructured into blocks, or tiles, in which each tile executes only part of the iteration space and therefore performs only part of the original computation. Consequently, each tile generates only a subregion of the original array and depending on the data dependencies of the loop, it could be the case that at the end of the tile execution, the generated array subregion is not altered by future iterations (i.e., consecutive tiles). In addition, asynchronous send and receive operations are inserted at the end of each tile so that the transfer of the array subregion generated by the corresponding tile is initiated. This transfer is completed by the network co-processor, while the CPU continues computing the next tile of the array. In order for such a transformation to preserve the correctness of the original code, the subject application needs to first be analyzed. In general, to restructure code to pre-push the results of its computation, we must first determine the following information:

- the communication operations in the original code and the corresponding computation loop(s) that write(s) to the array being sent
- the pairs of matching send and receive operation(s), since both the send and the receive must be transformed in concert
- the earliest execution point where it is safe to receive pre-pushed data (This is important in cases where the receiver uses the receive array prior to the part of the code we are trying to transform. In such a case, it is only safe to transfer the data after the latest point of such use.)

The last two are difficult to determine and in some cases they can be statically undecidable. Therefore, our transformation effort focuses on cases that reveal more information about the communication and do not exhibit such undecidability. Such information is statically known in the case of collective communication operations such as `MPI_ALLTOALL`. In such operations, both sending and receiving is implemented internally and the function call appears as an atomic, or indivisible, operation at the level of the application. In addition, the semantics of `MPI_ALLTOALL` require that all participating nodes have to call it. Therefore, we do not need to match statically the senders to the receivers; we know that all nodes exchange data, and they do so in a predetermined pattern. Regarding the computation, our current analysis focuses on computation loops where every node executes the same code. In other words, there can be no branches (i.e., `if` statements) in the code that stores data into the array that is being exchanged. Many scientific codes contain frequently executed sections consisting of a multiply-nested loop in which the inner loops execute some computation kernel

```
integer A_s(1:NX)
integer A_r(1:NX)
...
do iy=1, NX  !outer loop
   do ix=1, NX  !inner computation loop
      ...
      A_s(ix) = ... !RHS is not array ref.
   enddo
   !sends A_s and receives into A_r
   call collective-comm(A_s,A_r)
enddo
```

(a) Before

```
integer A_s(1:NX)
integer A_r(1:NX)
...
do iy=1, NX  !outer loop
   do ix=1, NX  !inner computation loop nest
      A_s(ix) = ...
      !wait for comm of prev.  tile to complete
      if(ix mod K == 0) then
         to=...
         size=K
         call async-send(A_s(...),size,to,...)
         call async-recv(A_r(...),size,from,...)
      endif
   enddo
enddo
```

(b) After

Figure 2. Abstract target code segment before and after transformation

and store the results in an array which is then exchanged using MPI_ALLTOALL at the end of each iteration of the outer loop (see Figure 2(a)). This communication-computation pattern is the domain on which our current transformation is focused. Sorting, LU Factorization, Finite differences, and multi-dimensional FFT constitute examples of algorithms that could fit this abstract form, and can be transformed to exploit pre-pushing.

To demonstrate the result of the transformation, Figure 2 shows an abstract target code before and after being transformed. The tiling of the computation loop nest is controlled by the parameter K which sets the number of iterations of the tile loop per tile. Determining the optimal tile size is not a trivial task, and is best performed by an automated system, since the value may change as applications migrate across platforms. However, finding the optimal value for K is beyond the scope of this paper. A discussion about the issues related to the performance critical parameters can be found in [3].

3. Automated Transformation Technique

3.1. Opportunities for Transformation

The first step toward modifying the code is identifying opportunities for transformation. To do so, the following information needs to be collected:

- $\mathcal{C}$, a call to MPI_ALLTOALL.
- $\mathcal{A}_s$, the array sent by $\mathcal{C}$, which is the first argument to $\mathcal{C}$.
- $\mathcal{A}_r$, the array received by $\mathcal{C}$, which is the fourth argument to $\mathcal{C}$.
- ℓ, the loop nest executed by all nodes, which finalizes all elements in $\mathcal{A}_s$ before $\mathcal{C}$ is called. ℓ is the last loop nest not in a conditional statement, lexically preceding $\mathcal{C}$, that mutates $\mathcal{A}_s$. $\mathcal{A}_s$ can be mutated directly by assignment, or indirectly by passing $\mathcal{A}_s$ by reference to a called procedure. In the former case, if the source code for the procedure is unavailable, it cannot be guaranteed that $\mathcal{A}_s$ is written. To resolve this uncertainty, the user must be queried (making the system semi-automatic), but if ℓ is the only loop preceding $\mathcal{C}$, then it is a conservative assumption to consider ℓ to be a mutator.

3.2. Compute-Copy Pattern

For each transformation opportunity, we determine the pattern by which values are computed and copied into $\mathcal{A}_s$. We currently consider two cases:

direct $\mathcal{A}_s$ is the LHS of an assignment statement where the RHS is not an array reference, as seen in Figure 2. Section 3.3 describes how to analyze codes fitting this pattern.

indirect In this case, the contents of $\mathcal{A}_s$ are computed indirectly in the sense that they are first computed in a procedure, $\mathcal{P}$, which stores them in a temporary array, and are then copied to $\mathcal{A}_s$ afterwards. As in Figure 3(a), $\mathcal{A}_s$ appears on the LHS of an assignment where the RHS contains a reference to a different array, $\mathcal{A}_t$. Each call computes a portion of the final results and writes them to $\mathcal{A}_t$ which is passed by reference. After the call to $\mathcal{P}$, the contents of $\mathcal{A}_t$ are copied to $\mathcal{A}_s$ in a copy loop, ℓ_{cp}. The purpose of this pattern is to aggregate the partial results computed by each call to $\mathcal{P}$ so that they can be sent together at the end of ℓ. The goal of Section 3.4 is to remove ℓ_{cp}, and directly send the contents of $\mathcal{A}_t$, as this avoids the copy and is thus more efficient.

3.3. Handling the Direct Pattern

If $\mathcal{A}_s$ is written directly, we first determine which parts of the send array, $\mathcal{A}_s$, can be safely sent at a given point in the iteration space of ℓ. If an array reference, $\mathcal{A}_2$, overwrites elements previously written by another array reference, $\mathcal{A}_1$, then those elements are unsafe to send between $\mathcal{A}_1$ and $\mathcal{A}_2$. Using array dependence analysis to find output dependences [15], we determine whether the element referenced by a given array reference is safe to send after that reference is reached during execution. A safe array reference, denoted $\mathcal{A}_s^f$, is one with no output dependences on it and represents the latest element to be finalized.

We could transform the program so that elements referenced by $\mathcal{A}_s^f$ are sent one at a time, as each is computed. Although correct, it is desirable for reasons of efficiency to aggregate these single element sends into fewer, larger send operations. *Array access analysis* [12] can enable this aggregation, by determining the region of $\mathcal{A}_s$ written during a single tile. To simplify our prototype implementation, we use the simplest, most course-grained access representation, known as a partial triplet, which contains the symbolic upper and lower bound of an index expression, i_k, denoted as $u(i_k)$ and $l(i_k)$ respectively. This analysis determines the size, denoted $size$, of the blocks of contiguously accessed array elements, or *blocks*, written during the runtime of K iterations of ℓ, and the offsets of the blocks, denoted $offsets$.

Using the results from this analysis, a communication loop nest can be generated to iterate through all $o \in offsets$ in order to initiate the appropriate asynchronous communication calls to transmit the generated *blocks*. Note that if the array access pattern is regular, all the data might be in just one continuous block. This is the optimal case, as the transfer of the data can be performed with a single transfer, achieving minimal overhead and high bandwidth.

3.4. Handling the Indirect Pattern

In the case that $\mathcal{A}_s$ is written indirectly, as in Figure 3, a copy loop, ℓ_{cp}, aggregates temporary results into $\mathcal{A}_s$, which will be sent once all computation has concluded. Since we want to send results as they are generated in order to overlap communication with computation, this aggregation is unnecessary. Therefore, we can directly send the contents of $\mathcal{A}_t$, which can reduce runtime by eliminating the time taken to copy $\mathcal{A}_t$ to $\mathcal{A}_s$. The flow of data from $\mathcal{A}_s$ to $\mathcal{A}_r$ can be represented as $\mathcal{A}_t \xrightarrow{copy} \mathcal{A}_s \xrightarrow{send} \mathcal{A}_r$. By transitivity, we can eliminate the copy and still complete the same operation by the equivalent $\mathcal{A}_t \xrightarrow{send} \mathcal{A}_r$.

```
integer $\mathcal{A}_s$(1:10,1:10,1:10)
integer $\mathcal{A}_t$(1:100)
do iy = 1, 10  !loop nest $\ell$
   call $\mathcal{P}$(..., $\mathcal{A}_t$)
   do ix = 1, 100  !$\ell_{cp}$
      tx = ix % 10
      ty = ix/10
      $\mathcal{A}_s$(tx,ty,iy) = $\mathcal{A}_t$(ix)
   enddo
enddo
```

(a) Before

```
integer $\mathcal{A}_s$(1:10,1:10,1:10)
integer $\mathcal{A}_t$(1:100)
do iy = 1, 10  !loop nest $\ell$
   call $\mathcal{P}$(..., $\mathcal{A}_t$)
   call async-send($\mathcal{A}_t(\ldots)$,...)
   call async-recv($\mathcal{A}_r(\ldots)$,...)
   enddo
enddo
```

(b) After

Figure 3. Abstract indirect pattern code segment before and after removing the redundant copy

To determine the region of $\mathcal{A}_t$ that has been finalized during a tile, we cannot directly analyze the loop that wrote to $\mathcal{A}_t$ since it is inside a procedure with source code that is unavailable. Therefore, we have to infer the access pattern of $\mathcal{A}_t$ indirectly by analyzing ℓ_{cp}. It is reasonable to assume that the region of $\mathcal{A}_t$ accessed when being copied to $\mathcal{A}_s$ is the one that is finalized by one call to the procedure. In addition, if ℓ_{cp} is executed more than once per tile, which is usually the case, we need to aggregate the temporary results, but not to the degree that they were originally aggregated. To achieve this, we expand the capacity of $\mathcal{A}_t$ by adding an extra dimension, and modify the reference to $\mathcal{A}_t$ that is passed to the procedure accordingly. Finally, the resulting communication code must preserve the original mapping from $\mathcal{A}_t$ to $\mathcal{A}_s$ that was induced by ℓ_{cp}. In other words, the blocks of $\mathcal{A}_t$ must be sent to $\mathcal{A}_r$, in the same order that blocks of $\mathcal{A}_t$ were copied to blocks of $\mathcal{A}_s$. Further details for removing the redundant copy are beyond the scope of this paper, but can be found in [5].

3.5. Communication

In this paper, we focus on transforming communication using `MPI_ALLTOALL` [10], which divides arrays into NP partitions along the last dimension, each corresponding to a different node. To preserve the semantics and efficiency of `MPI_ALLTOALL`, we must ensure that data is written for each of the nodes to receive during every tile. We can guarantee that the entirety of the last dimension is traversed if the loop inducing the traversal of the last dimension, the *node loop*, is not the outer loop, in which iterations are being split into tiles. If the node loop is the outer loop, we could use loop interchange [1] to exchange the outermost loop with one of the inner loops. If data dependences do not allow us to perform the interchange, the semantics of `MPI_ALLTOALL` can still be preserved by having all the nodes send to a subset of the nodes during each tile, but this is not as efficient as network congestion may ensue if all of the nodes are competing to communicate with one or a few nodes. The method for generating communication code in this case is given in [5]. In Figure 4, we show the replacement communication code that preserves the semantics and efficiency of `MPI_ALLTOALL` when the node loop is outermost.

3.6. Transforming the Program

After the previous stages of analysis are performed, we transform the program according to the following steps:

1. Insert the communication code shown in Figure 4 at the end of the body of ℓ.

```
integer $\mathcal{A}_s$(...,SZ)
...
do j = 1,NP-1
   to = mod(mynum+j,NP)
   call mpi_isend($\mathcal{A}_s$(...,(to-1)*(NP/SZ)),...)
   from = mod(NP+mynum-j,NP)
   call mpi_irecv($\mathcal{A}_r$(...,(from-1)*(NP/SZ)),...)
enddo
```

Figure 4. Communication Code

2. Insert a blocking call to wait for all outstanding receives from the previous tile to complete, before the code inserted in step 1.

3. Insert code after ℓ, to exchange any leftover elements not sent by the last tile, which result from K unevenly dividing the number of iterations of ℓ (i.e., $\ell \bmod K$).

4. Insert code, after ℓ and before $\mathcal{C}$ to wait for the arrival of the last blocks of data after the end of ℓ.

5. Remove $\mathcal{C}$, the original communication.

4. Implementation and Evaluation

The automated approach presented in the last section was implemented as a Fortran 90 source-to-source code transformer, called the Compuniformer, using the Nestor program transformation framework [14]. Using a source-to-source transformer, we decouple our transformation from the specifics of any particular compiler designed for a particular architecture, allowing our optimization to be complemented with traditional compiler optimizations. Nestor is a lightweight framework for implementing transformations to Fortran 90 code, providing a parser, a transformable IR, and unparser. Nestor also includes a data dependence analysis tool which uses Petit and the Omega Test [13]. At this time, some portions of the implementation are semi-automatic, in that they require some user input, due to limitations in the built-in analysis tools; future work will develop these capabilities more fully.

We have performed a preliminary evaluation of our prototype aimed at testing the correctness and performance of the transformation. The evaluation allows us to not only verify the correctness of the implementation, but also the techniques that underly it. We wrote a test program which is simple, yet tests many of the features of the transformation process. The test code exhibits the indirect computation pattern, which complicates the transformation since we remove the redundant copy loop. The test code as transformed by our system compiles and executes, producing output identical to that of the original, suggesting the correctness of our technique and implementation.

5. Conclusions and Future Work

In this paper, we presented novel techniques to automate the transformation of explicitly parallel codes to maximize communication-computation overlap. The broader impact of this work is the performance improvement of parallel MPI codes on networked clusters, enabling more scalable application of the parallel codes to larger numbers of processors, benefiting the large community of domain scientists using such technologies. Future work includes creating heuristics to deal with some common idiosyncrasies of real-world codes, strengthening the implementation by incorporating more sophisticated program analysis, targeting other types of collective communication, and evaluating the system's performance on a variety of real-world codes, which should inform future work on extending the system's generality.

References

[1] Randy Allen and Ken Kennedy. Automatic loop interchange. *SIGPLAN Not.*, 39(4):75–90, 2004.

[2] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In *Seventh Workshop on Languages and Compilers for Parallel Computing*, 1994.

[3] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to Parallel Codes for Communication-Computation Overlap. *Supercomputing*, 2005.

[4] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC Specification v. 1.1. `http://upc.gwu.edu/documentation`, 2003.

[5] Lewis Fishgold. An Automated Approach to Improve Communication-Computation Overlap in Clusters. Senior Thesis. University of Delaware, 2005.

[6] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. CRPC-TR92225, Rice University, Houston, TX, 1993.

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

[8] Amit Karwande, Xin Yuan, and David K. Lowenthal. CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[9] C. Kessler and W. Paul. Automatic parallelization by pattern matching. In *Proceeding of Second Int. Conference of the Austrian Center for Parallel Computation*, pages 166–181, 1993.

[10] MPI Forum. MPI: A message-passing interface standard, v1.1. Technical report, University of Tennessee, Knoxville, June 12, 1995.

[11] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. ACM Fortran Forum 17, 2, 1-31, 1998.

[12] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.

[13] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE Conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[14] Georges-André Silber and Alain Darte. The Nestor library: A tool for implementing Fortran source to source transformations. In *High Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 653–662. Springer Verlag, April 1999.

[15] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] X. Yuan, R. Melhem, and R. Gupta. Algorithms for Supporting Compiled Communication. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):107–118, 2003.