Automatic MPI application transformation with ASPhALT

Anthony Danalis[1], Lori Pollock[1], and Martin Swany[1]

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

Abstract

This paper describes a source to source compilation tool for optimizing MPI-based parallel applications. This tool is able to automatically apply a "prepushing" transformation that causes MPI programs to aggressively send data as soon as it is available, thus improving communication-computation overlap and improving application performance.

In this paper we present asphalt_transformer*; the Open64-based component of our framework,* ASPhALT*, responsible for automatically performing the prepushing transformation. We also present an extensive study of the performance gains witnessed from automatically transformed codes. In particular, we demonstrate how different levels of aggregation affect the performance of parallel programs executing various computation kernels on different clusters. Furthermore, we discuss the differences in performance improvement between the hand-optimized and automatically optimized codes, as well as the effect of automation on time-to-solution.*

1 Introduction

Cluster computing is a common way to achieve high processing power. Clusters are used by groups with budgets and requirements as large as those in national laboratories and supercomputing centers, or as small as university research teams with a few members. Nevertheless, there are some negative aspects associated with cluster computing. Namely, unlike shared memory machines, cluster nodes communicating over a network introduce delays

to the executing parallel applications due to the interconnect's latency. Furthermore, the applications executing on cluster environments need to incorporate explicit message passing to achieve communication across nodes, which increases the complexity of the applications and prolongs the time to solution. We assert that given this complexity, it is highly desirable to have an *automatic system for transforming parallel programs* in order to improve the performance of those programs and make reasonable application performance available to a wider range of engineers and scientists.

To minimize the communication latency, specialized interconnects are used in clusters. Several vendors of such technologies [5, 21], have been providing solutions with extremely low latency and high bandwidth. Nevertheless, at the time this paper is being written the fastest networks have a best case latency in the order of hundreds of nanoseconds, while the fastest CPUs are clocked in the order of 5GHz. This results in message delivery times in the order of thousands of CPU cycles. In addition, contrary to simple benchmarks, complex applications using abstract communication libraries such as MPI, may not always be able to fully utilize the underlying hardware, resulting in even higher communication overhead. Furthermore, the very design of an application can have an effect on performance. In particular, if computation and communication are separated, in order to achieve a modular design with improved maintainability, the communication-computation overlapping features of the underlying hardware are effectively disabled. Even if the hardware supports advanced features such as Remote Direct Memory Access (RDMA), if the communication starts after the completion of the computation, the transfer cannot be overlapped and will lead to thousands of idle CPU cycles.

In an effort to hide from the programmer the complexity of explicit message passing, several global-address space (GAS) languages [12, 17, 18, 19, 25] have been proposed, along with techniques to automatically translate programs

written in a shared memory model such as OpenMP to MPI [3], or even enable programs written in Matlab to execute in parallel [23, 8]. Nevertheless, explicit message passing through MPI remains the dominant parallel programming paradigm in the high performance computing community.

The work presented in this paper is part of a bigger project, *ASPhALT*, that aims to address the performance issues associated with MPI communication, while hiding from the programmer the complexity of highly efficient message passing codes. Namely, we present *asphalt_transformer*, a tool that can automatically apply a prepushing transformation [10] on MPI codes, without requiring any knowledge of the esoterics of our approach by the parallel application developer. We demonstrate that our tool considerably reduces the communication latency of the transformed applications, by enabling communication-computation overlapping. Our tool can provide a developer with sufficiently efficient code, even if the initial code has the simple form of Figure 1. As such simple code is easy to write and understand, we argue that our tool can reduce the complexity of developing efficient message passing parallel applications and improve time to solution.

This paper proceeds with a discussion of the prepushing compiler transformation process as well as experimental results that provide an evaluation of the compiler-transformed codes in terms of performance. We have conducted experiments on more than one cluster environment and we present a detailed study that shows the performance improvement of the transformed code as compared to the original version, as well as the effect of aggregation on the performance of the transformed code.

2 Prepushing Transformation

Figure 1, depicts the abstract form of the input code to the transformer and the output after applying the prepushing transformation. As shown in the figure, the class of applications that can benefit from this transformation are those that consist of two parts.

First, the application executes a computation loop, with no (or limited) dependencies across iterations. This loop defines (i.e., alters, or generates) the application data. Then, a communication call (such as an MPI collective operation) , following the loop, exchanges the data that was defined by the computation. In contrast, the resulting code has a less modular design, with the computation organized in tiles, and the communication of each tile overlapped with the computation of future tiles through the use of asynchronous data exchange mechanisms (such as `MPI_Isend()` and `MPI_Irecv()`). In a sense, prepushing can be viewed as reversed (or producer initiated) prefetching as it is pushing the data to the consumer as soon as it becomes available,

even if it is not needed yet by the consumer. The goal of this transformation is to hide, or at least reduce, the communication latency by overlapping the data transfer with useful computation.

As can be seen in Figure 1, the transformation has the following effect on the code:

- restructures the computation into tiles

- replaces synchronous communication by asynchronous communication

- issues the asynchronous calls in correspondence with each computation tile, such that the communication of a tile is overlapped with the execution of future tiles, enabling the tiles to proceed in a pipelined fashion
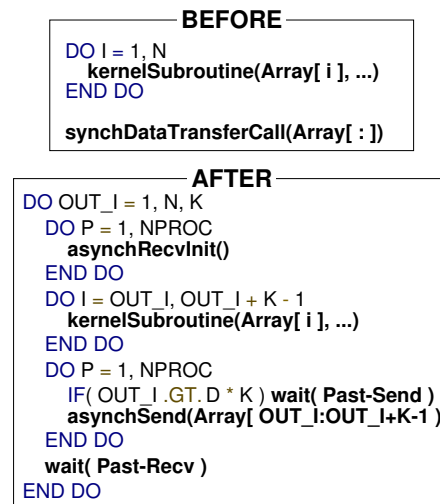
```
┌─────────── BEFORE ───────────┐
│ DO I = 1, N                   │
│    kernelSubroutine(Array[ i ], ...) │
│ END DO                        │
│                               │
│ synchDataTransferCall(Array[ : ]) │
└───────────────────────────────┘
┌─────────── AFTER ────────────┐
│ DO OUT_I = 1, N, K            │
│    DO P = 1, NPROC            │
│       asynchRecvInit()        │
│    END DO                     │
│    DO I = OUT_I, OUT_I + K - 1 │
│       kernelSubroutine(Array[ i ], ...) │
│    END DO                     │
│    DO P = 1, NPROC            │
│       IF( OUT_I .GT. D * K ) wait( Past-Send ) │
│       asynchSend(Array[ OUT_I:OUT_I+K-1 ) │
│    END DO                     │
│    wait( Past-Recv )          │
│ END DO                        │
└───────────────────────────────┘
```

Figure 1. Code Transformation Overview

Sorting [9], LU Factorization [11], Finite differences, and multidimensional FFT, are examples of algorithms that fit this abstract canonical form, and can be optimized with the use of prepushing.

3 Transformer Design and Implementation

Our tool, which we refer to as *asphalt_transformer*, is implemented as a compilation phase of Open64 [2] as depicted in Figure 2. Open64 is able to parse C, C++ and Fortran 95 code into an intermediate representation called WHIRL and output a program's Abstract Syntax Tree (AST) represented in WHIRL, along with the symbol tables, to a file. After this point, *asphalt_transformer*, using Open64 libraries, can perform the actual analysis and manipulation of the AST and the symbol tables. At the end of
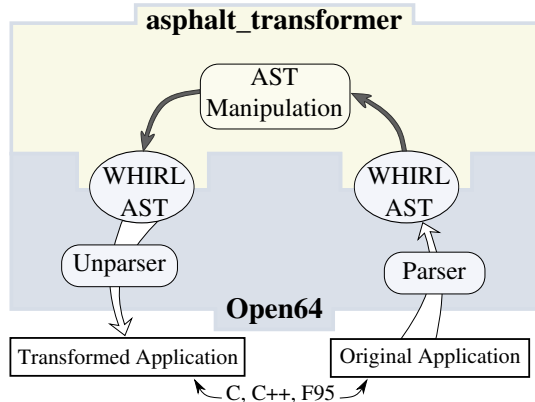
Figure 2. asphalt_transformer

the transformation, with the use of Open64 provided "unparsers", the resulting AST can be translated back to the source language.

We presented an earlier preliminary version of an automatic transformer in [15]. Illuminating as it was, regarding the process of designing and implementing such as system, it did not produce the results one would expect. This happened because the earlier version was considerably different than the current one in several ways. First of all, it was based on Nestor [26], rather than Open64. More importantly, in an effort to achieve generality, we designed our first transformer to handle general case codes, without making assumptions that one could expect to hold for scientific codes. Last, but not least, our preliminary version was built as a stand-alone program, in contrast with the current effort to build a fully automated optimization framework (*AS-PhALT*), in which the transformer is the central component, but not the only one.

3.1 Transformation Process

At the beginning of the transformation process, *asphalt_transformer* traverses the Abstract Syntax Tree, WHIRL AST, representing the code, examining all the function calls. When a function call is recognized as one of the MPI communication calls that the system knows how to optimize, details about that call are extracted. The current version of our tool focuses on MPI_ALLTOALL(), but adding support for more communication calls requires only the handling of the particulars of each call. After a known data transfer call is found, information from the function's prototype is used to identify the array that is being transmitted (A_{snd}) by the call.

Consecutively, the AST is traversed again, for the computation kernel that includes the definition of A_{snd} to be found. In other words we try to identify the code segment where data is stored in A_{snd}. This can be achieved through

the use of *Ud-chain*, or *Reaching Definitions* data flow information. In the current implementation we assume that there are no if-then-else branches in the code that stores data in the message buffers, nor any other incoming control edges between the definition and the use (such as goto labels). Due to this assumption, there is only one reaching definition for A_{snd} and it is a *dominator* of the use. The discovering of the dominating reaching definition is implemented as a simple backward traversal of the AST.

Since our transformation aims to improve performance by increasing the communication-computation overlapping, we focus on array assignments that occur within loops, so that there will be some substantial computation to overlap with the transfers. The inner most loop, that encloses the definition is considered the computation kernel loop, L_{krn}.

When a known data transfer call and the corresponding computation loop have been identified, *asphalt_transformer* tries to identify if part of the computation consists of an unnecessary array to array copy. We do that because some collective calls, such as MPI_ALLTOALL, exchange data in a particular pattern and so programmers commonly use a special "data packaging" loop before them, so that the data is placed in the send array (A_{snd}) the way that the MPI function "expects" to find them. If *asphalt_transformer* can identify such a copying, and can conservatively analyze the array access patterns, it attempts to remove it. Note that for the definition of A_{snd} to be conservatively removed, there needs to be only one use of this definition, and it needs to be the function call we are optimizing. In other words, removing the original communication call should render the definition of the source array as dead code, and therefore conservatively removable.

If such a "data packaging" array copy is found, the analysis phase searches further up the AST to find the definition of the array that is copied into A_{snd}. This "source-array", A_{src}, can then be sent directly to the receiver, using a data exchange pattern equivalent to what would have taken place by the exchange via the collective call. If the definition of the A_{src} cannot be found , or if no such data packaging copy took place, A_{snd} is transmitted instead.

After the array to be sent (either A_{src}, or A_{snd}) has been identified, the innermost loop that encloses its definition, L_{krn}, is marked as the target for tiling. By tiling the loop, blocks of computation are generated, followed by the corresponding aggregated communication of the data defined (generated, or altered) within the block. In this new form of the code, the communication of each block can be overlapped with the computation of future blocks. In addition, by controlling the size of the blocks, we control the aggregation of the data transfers which has a direct effect on the performance of the optimized code as we have shown in [10] and we also discuss in Section 5. Clearly, a very small tile size would lead to a large number of small message trans-

fers, while a very large tile size would lead to a very large final tile which would inhibit performance as the final tile's communication cannot be overlapped.

Tiling turns L_{krn} into the inner loop of a double loop nest, with the outer loop, L_{out}, being a newly introduced one. In addition, the boundaries of L_{krn} are transformed, so that it executes one tile, of size $\mathcal{K}$ kernels, and L_{out} is used to iterate over tiles. The asynchronous communication loops are inserted inside L_{out} as well. In addition, blocking operations need to be inserted in L_{out}, to guarantee correctness while keeping the duplicate buffers at a manageable level. One could avoid using `wait` operations inside the loop, and wait for the arrival of the data at the end of the whole process but such an approach would require a very large amount of resources. This is true, because an asynchronous `send` operation returns before the data has actually finished being transferred. Therefore, if a subsequent computation reused the same send buffer, it could not be guaranteed that the previous data was not overwritten before being submitted. Consequently, if blocking is not used, the application needs a different send buffer for every `send` operation. Similarly, asynchronous `recv` operations need matching `wait` operations before the received data can be safely saved into a permanent array, or the program needs to use a different temporary receive buffer for every `recv` operation. By using blocking `wait` operations and maintaining limited duplicate buffers, the asynchronous communication can have a managed unacknowledged window, which can be thought of as a form of tile pipelining.

Finally, some implementations of MPI, notably MVAPICH[1], need an additional call to a probing function such as `MPI_Test` in order to proceed with large asynchronous transfers. This is true, due to the nature of the rendezvous protocol used for large messages. Namely, when `MPI_Isend` is called, with a large message size, the sender initiates a handshake by transmitting a small message with meta-data about the real message, in order to prepare the receiver to accept the message. Only when the receiver responds with a message acknowledging the availability of a receive buffer can the actual message transfer be initiated. The problem that can occur with computationally intensive applications, such as those we are considering, is that the acknowledgment message may not arrive until the sender is executing the computation kernel. In such a case, the actual transfer will not be initiated, let alone completed, until the application makes the next call into the MPI library, after the kernel has completed. Should that occur, the opportunity for communication-computation overlapping will have been lost, since the computation will have been completed. To prevent this from happening, we insert an MPI call with no side effects, namely `MPI_Test` inside the computation loop, L_{krn}. This way, even if the handshake response arrives while the application is executing the computation loop, the MPI library will be informed, and the asynchronous message transfer will be overlapped with the computation. A more detailed discussion on the rendezvous protocol in MVAPICH can be found in [27].

The analysis and transformation performed by our system is implemented at the WHIRL, or intermediate implementation level of Open64. Therefore it is in principle independent of the source language. In reality though, different features of the different potential source languages could complicate phases of the analysis. For example, C/C++ pointers and aliasing could make reaching definitions analysis impossible or intractable. Nevertheless, if the analysis described earlier in this section can be conservatively performed, then the transformation can be applied without major differences between source languages. In the experiments performed so far, we have focused on Fortran, since we consider it to be the language of choice among domain scientists, as well as more tractable to analyze.

4   Experimental Study

The main question targeted by this experimental study is whether an automatically transformed code can experience performance improvements comparable to what we witnessed in our proof of concept study [10], in which the applications were transformed manually. To answer this question, we used our tool, *asphalt_transformer*, to automatically transform an MPI code, and we compared the performance of the code before and after the transformation. Since the performance of the transformed, tiled code depends heavily on the tile size, we profiled a large part of the parameter space. In other words, we ran the transformed code multiple times, using several different values of tile size. In our graphs, we show the results for all the different values we used.

```
                        ┌─── F2 ───┐
STRT = 1
DO I=1,NX
   STRT = (NX+STRT)/2
   DO CNT=1,4
     DO J=STRT,NX
        K = STRT+MOD(J+1,NX-STRT+1)
        L = STRT+MOD(J+NX/2,NX-STRT+1)
        TMP1 = COS( ( AUX(K)+AUX(STRT) )/2 )
        TMP2 = COS( ( AUX(L)+AUX(STRT)-AUX(1) )/3 )
        TMP3 = COS( (TMP1 + TMP2) * 4*ATAN(1.0) )
        AUX(J) = ( AUX(I)+AUX(J)+ (TMP1-TMP2)/(TMP3+2.0) )/2
     ENDDO
   ENDDO
ENDDO
```

Figure 3. Custom Computation Kernel

To demonstrate the generality of our transformation, as far as the type of computation performed by the applica-

tion is concerned, we created a synthetic application where the computation kernel consists of a loop nest that performs multiple arbitrary, irregular accesses to the input array, taking time $O(n \cdot log n)$, shown in Figure 3. Since we have full control over this synthetic computation kernel, we can alter its behavior to control the computation/communication ratio, emulating the load of different types of computation kernels (`FFT`, finite differences, etc).

4.1 Experimental Methodology

For the purpose of this study we used two clusters. The first cluster is located in the Chemical Engineering department at the University of Delaware and has 41 nodes with 2.2GHz AMD Opteron 248 CPUs. The network interconnect uses the Ammasso 1100 Gigabit Ethernet Server Adapter, which supports RDMA. Each node runs Linux 2.6.9 and has 2GBytes of main memory.

The second cluster is located at the University of Tennessee and consists of 55 nodes with dual 1.4GHz AMD Opteron 240 CPUs. The network interconnect is Myrinet with NIC information: 333.2 MHz LANai, 132.9 MHz PCI bus, 2 MB SRAM and MX version 1.0.3. Each node runs Linux 2.6.13 and has 2GBytes of main memory.

Regarding the MPI library, we used the MPI versions provided by the corresponding network vendors.
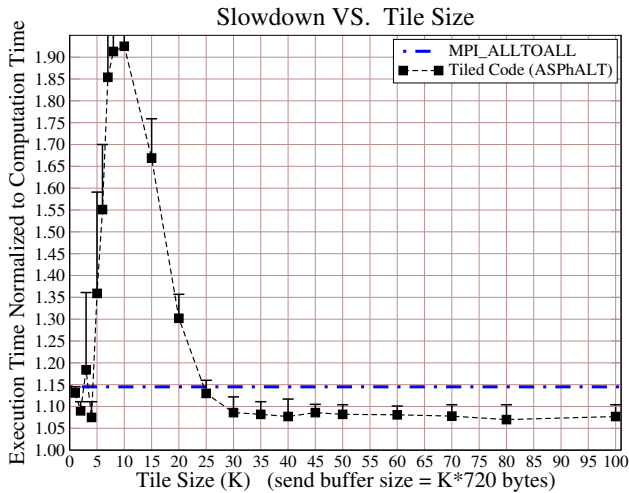
5 Results and Analysis

Figure 4. NP=16, Ammasso, 1440x1440x48

The results of our experiments are demonstrated in Figures 4, 5, 6, 7. The `Y` axis in these graphs depicts the `slowdown` due to communication. This value is obtained by normalizing the total execution time witnessed by the
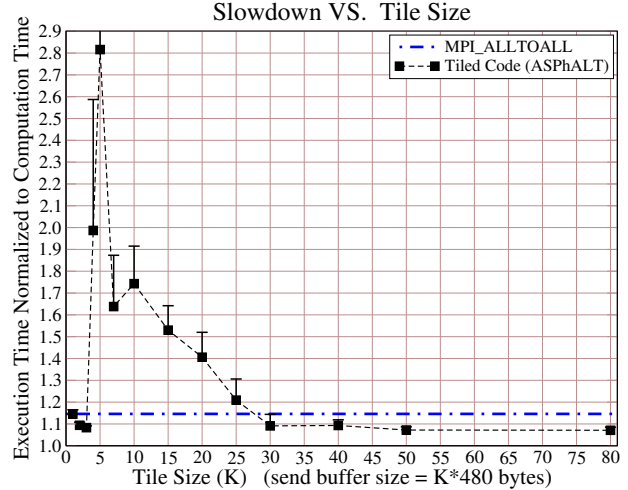

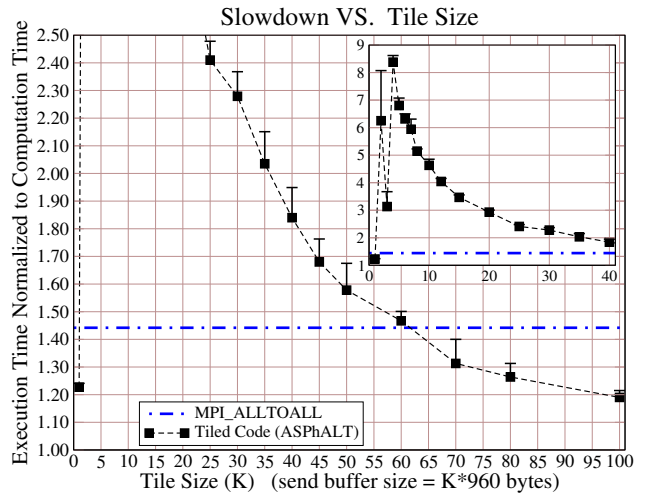**Figure 5.** NP=24, Ammasso, 1440x1440x48

Figure 6. NP=24, Ammasso, 2880x1440x96

application to the computation time of the application. In other words, the `Y` axis shows the value $1 + O_{comm}$ where O_{comm} is the communication overhead. We chose to normalize the execution time in order to emphasize the communication overhead, since our work focuses on overlapping the communication with the computation in order to reduce this particular type of overhead.

The `X` axis in the graphs depicts the size of the tile ($\mathcal{K}$), or the number of computation kernels that are executed before a data transfer is initiated. The `X` axis also depicts the size of the send buffer, since the transferred data size is linearly proportional to the tile size.

The (blue) dashed-dotted line in each graph represents the slowdown witnessed by the original code (`MPI_ALLTOALL`). Note that the original code is not tiled
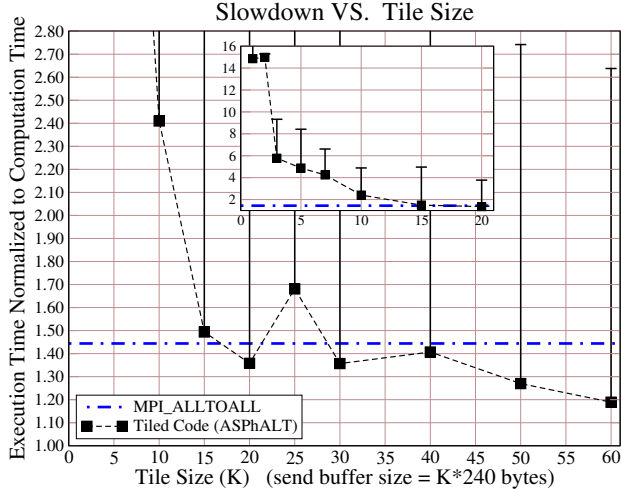
Figure 7. NP=48, Myrinet-MX, 1440x1440x48

and therefore the slowdown is the same for all values of the X axis.

All measurements were repeated in the order of 10 times, in order to reduce the noise of our experiments. In graphs 4, 5, 6, 7, each black box represents the minimum execution time observed by the transformed code for a given tile size. The error bars indicate the distance to the median. Therefore, the range depicted in the graphs represents 50% of our executions with the least noise.

5.1 Performance and Tile Size

Unmistakably, for every cluster and for all different numbers of processors, there are more than one tile sizes (values on the X axis) that make the optimization successful (black box lower than dashed line). Therefore, our experiments indicate that the prepushing transformation, can reduce the communication overhead if a good tile size is used in the transformed application.

However, it is interesting to observe that there is no consistent "safe" value for the tile size. That is, in some cases we start witnessing good performance at relatively smaller tile sizes, while in others higher aggregation is needed for good performance. In two of the cases (6, 7) the performance for small tile sizes was so poor that presenting them in the same graph with the rest of the values, would render the rest of the values unreadable. For this reason, we include inset graphs in order to show the behavior of the code for small sizes.

These observations highlight the fact that transformed codes tuned for a given cluster environment might not perform as well if they are transferred to a different cluster, or the environment changes in any way. For this reason, or current and future work is focused on automatically tuning the

transformed codes, or at least guiding the tuning process.

5.2 Automatic vs. Manual Transformation

In our previous work [10], we performed the transformations by hand in order to study the potential gains of prepushing. In particular, for every benchmark application we created two transformed codes; one using MPI's asynchronous I/O, and another using one-sided I/O through calls to a thin library we developed to make the low level API of the hardware vendor accessible from Fortran code.

asphalt_transformer, although still under development, is already able to transform the MPI_ALLTOALL code the same way as we would transform it by hand. **In other words, by letting the compiler perform the transformation automatically, we do not sacrifice performance and are able to automatically utilize more complicated features, such as MPI asynchronous I/O.**

As far as the transformation that utilizes one-sided I/O is concerned, it is beyond the current capabilities of *asphalt_transformer*, but we are planning to include it in future versions of our system. This scheme is more difficult to implement, because it bypasses the abstraction of MPI, which creates the need to explicitly handle more details. Nevertheless, the very reason that creates these difficulties, (i.e., the bypassing of MPI's abstractions) yields significant additional performance improvements as we showed in our previous study, and as Bell et al. show in [4].

6 Time to Solution

A system such as *ASPhALT* provides an obvious benefit to parallel application developers. It can make their applications run faster, or scale to more CPUs. Although execution speed and scalability are two metrics that are intuitive to understand and easy to measure, the high performance computing community is becoming increasingly interested in a different metric of productivity; namely, time to solution [7, 14]. This metric is important to productivity because it is in a sense the real total time that a developer spent from the conception of the idea to the final results of the execution. We argue that *ASPhALT* can reduce time to solution much further than it can reduce the pure execution time, and we believe this for two reasons.

First, contrary to solutions that add new layers of complexity or instrumentation, or low level APIs in order to achieve better performance, our work allows developers to write their code in the simplest possible form (i.e., a computation loop followed by a collective communication call). In this way, domain scientists, who are the most common users of HPC systems, do not have to spend their time learning,

developing and debugging low level, asynchronous, or one-sided I/O codes. They can rather focus on what they do best, their science, and leave the efficient code generation to an automated system such as *ASPhALT*.

Furthermore, as one can see in Figures 4- 7, just writing code that initiates asynchronous transfer calls ahead of time does not guarantee a performance improvement. If parameters of the transformed code, such as the level of aggregation, are not selected properly, the "optimized" code could run orders of magnitude slower than the original code! It is clear that a system is needed that can transform the application codes, as well as automatically (or semi-automatically), select values for the parameters that control the performance of the transformed code.

7 Related Work

Significant amount of research has been performed in the field of Global Address Space (GAS) languages [12, 17, 18, 19, 25]. These projects are similar to our work in that they try to hide from the programmer the complexity of highly efficient parallel programming, while achieving efficient execution. GAS languages deploy compiler analysis and optimization along with low level, high performance APIs [6, 24] in order to achieve high performance while exposing a high level language to the programmer. The main difference with our project is that we aim to optimize MPI-based codes instead of introducing a new language or set of interfaces. This way, our optimizations can be applied to new or legacy applications written by developers oblivious to our work.

Two additional studies that considered a transformation similar to ours are presented in [22] and [20]. These papers, suggest *peeling*, or *strip mining* of the computation loops in order to achieve prefetching of the data. In our study, we consider applications that generate data using local arrays, and we try to prepush them to the destination before they are needed. This difference can have important implications as true `get` operations are not common[1], while one-sided `put` operations exist in all RDMA-capable networks. Furthermore, we have gone beyond identifying the transformation by developing an actual compilation phase (*asphalt_transformer*) capable of performing the transformation on existing scientific applications.

Bell et al. [4] presented very similar work to our own. They too transform parallel codes in order to prepush the generated data and achieve better communication-computation overlapping. Although our results are similar to theirs, the two projects are different in several ways. They focus on one-sided communication and they utilize a communication API (GASnet) lower than MPI, while we

show that significant benefits can be gained by the use of regular MPI asynchronous I/O (although we acknowledge the low-level, one-sided I/O can provide even further performance gains). In addition, we perform an extensive investigation of the effects of different levels of aggregation (i.e., different values of tile size) to performance, while they use two fixed message sizes for their study. Furthermore, their work is performed in the context of UPC, while our transformer (*asphalt_transformer*), is developed as part of *ASPhALT*, a project that aims to optimize MPI applications. Finally, the most important difference of the two projects is that we present a tool that can **automatically** transform a scientific application, where they showed benefits by transforming the applications manually.

Transforming MPI collective operations into point-to-point operations for performance is also considered by Faraj and Yaun [13]. Their work focuses on optimizing based on the topology of the network and takes no account of data dependence.

8 Conclusions and Future Work

The overarching hypothesis of which this work is a part, is that it is possible to create an integrated system where not only the program transformation can be performed automatically, but empirical active probing (similar to that performed by systems such as ATLAS [28] or FFTW [16]) can automatically determine the best parameter values needed in a given cluster. In a non-automatic environment, where the programmer has optimized the code by hand, every time the characteristics of the cluster change, profiling, or other time consuming techniques are necessary to adapt the application to the changed cluster. *ASPhALT* is being developed to be an automated and fully integrated system able to assist developers of parallel applications in generating highly efficient code without requiring the developers to have knowledge of advanced APIs or programming techniques, or the esoteric details of our system.

In this paper we presented *asphalt_transformer*, a compilation phase developed within Open64, which is able to automatically perform the prepushing transformation on MPI-based codes. We showed that the automatic transformation can reduce the communication overhead by a significant amount, regardless of the details of the computation kernel that the application executes, or the type of the cluster used.

In addition to developing the parameter selection and tuning part of *ASPhALT*, we are currently working on expanding the applicability of *asphalt_transformer* by enabling it to optimize more communication functions. In addition, we are planning to enable *asphalt_transformer* to generate code that can bypass MPI and perform low level one-sided I/O.

---

[1]Some APIs provide `get` operations that are implemented as a multi-step negotiation between the producer and the consumer.

Acknowledgments

We would like to thank the University of Tennessee and professor Dionisios G. Vlachos at the University of Delaware for providing us with access to their clusters.

References

[1] Network-Based Computing Laboratory. MPI over Infini-Band Project. http://nowlab.cse.ohio-state.edu/projects/mpi-iba.

[2] Open64. http://open64.sourceforge.net.

[3] A. Basumallik and R. Eigenmann. Towards automatic translation of openmp to mpi. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 189–198, 2005.

[4] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *20th International Parallel & Distributed Processing Symposium (IPDPS)*, 2006.

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

[6] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[7] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In *IPDPS 2004 PMEO workshop*.

[8] R. Choy. Parallel matlab survey. http://theory.lcs.mit.edu/~cly/survey.html.

[9] M. J. Clement and M. J. Quinn. Overlapping Computations, Communications and I/O in parallel Sorting. *Journal of Parallel and Distributed Computing*, 28(2):162–172, 1995.

[10] A. Danalis, K. Kim, L. Pollock, and M. Swany. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[11] F. Desprez, J. Dongarra, and B. Tourancheau. Performance study of LU factorization with low communication overhead on multiprocessors. *Parallel Processing Letters*, 5:157–169, 1995.

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

[13] A. Faraj and X. Yuan. Automatic generation and tuning of mpi collective communication routines. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402, New York, NY, USA, 2005. ACM Press.

[14] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, and L. Votta. Measuring hpc productivity. *International Journal of High Performance Computing Applications*, 18(4):459–473, 2004.

[15] L. Fishgold, A. Danalis, L. Pollock, and M. Swany. An Automated Approach to Improving Communication-Computation Overlap in Clusters. In *Parallel Computing 2005*, 2005.

[16] M. Frigo. A fast fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 169–180, 1999.

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

[18] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. tech report ucb/csd-01-1163, u.c. berkeley, november 2001.

[19] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for fortran d on MIMD distributed-memory machines. In *Supercomputing*, pages 86–100, 1991.

[20] C. Iancu, P. Husbands, and W. Chen. Message Strip Mining Heuristics for High Speed Networks. In *VECPAR*, 2004.

[21] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

[22] G. Liu and T. Abdelrahman. Computation-Communication Overlap on Network-of-Workstation Multiprocessors. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[23] F. M, M. C, K. K, and J. G. Strategy for Compiling Parallel Matlab for General Distributions Rice University, (TR06-877), Houston, TX, 2006.

[24] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *RTSPP IPPS/SDP'99*, 1999.

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

[26] G.-A. Silber and A. 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, Apr. 1999.

[27] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 32–39, 2006.

[28] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.