Linking Compilation and Visualization for Massively Parallel Programs*

Alex K. Jones[1], Raymond R. Hoare[2], Joseph St. Onge[2], Joshua Lucas[3], Shuyi Shao[1], and Rami Melhem[1]
[1]University of Pittsburgh [2]Concurrent EDA, LLC [3]Lockheed Martin

Abstract

This paper presents a technique to visualize the communication pattern of a parallel application at different points during its execution. Unlike many existing tools that show the communication pattern for the entire application, our tool breaks this communication pattern down into components to allow the more detailed study of application execution. These patterns are not merely snapshots or windows of the execution but rather are tied to specific code structures comprised of loops in the application. Our technique leverages our compiler, which adds instructions into the code to record where communications and code artifacts occur during execution. This information is stored into a trace format, which is read by our visualization tool. The visualization tool can graphically represent the communication pattern and message volume to allow a user to analyze and optimize the execution. As an example, we show how this information can be used to optimize the execution time and reduce the message delay of applications executed on a system enhanced with optical circuit switch interconnections.

1 Introduction

Multiprocessor interconnection networks can benefit from both temporal and spatial communication locality just as memory systems exploit locality of references through caches [8]. Temporal locality represents the effect of temporal aggregation of the inter-processor communications [16]. High temporal locality suggests that during any given time period inter-processor communication occurs across a certain number of connections, which can be called a communication *working set*. This provides the opportunity to reduce communication latency by dynamically grouping and scheduling messages and pre-establishing statically known connections. Spatial locality is determined by the distribution of the connections in the application and determines the size of the working set, also called *communication degree*. It has been shown that each node tends to have a small number of favored destinations for the messages it sends [2, 13]. For example, the NAS parallel benchmark suite exhibits very high spatial locality and, therefore, contains small working sets [2]. A working set refers to the set of communication pairs in use during a small window of execution.

It has also been shown that the communication patterns for many types of parallel applications are regular, i.e. they contain small working sets that are either unchanged or change infrequently [4]. However, the source of locality of reference is generally considered to be a product of artifacts of the basic code structures such as conditionals and loops. As a result, the impact of loop structures in code has been extensively studied and is the impetus for many related compiler optimizations such as loop fusion and loop tiling [15].

This paper presents an analysis of several parallel applications with the particular emphasis on correlating different working sets to code structures, loops in particular, within the applications. The compilation flow presented takes as input a parallel application written in C or Fortran 77+MPI and generates a C program that has been annotated to generate a trace of the application as it executes. The annotated program includes information about the code structure for segments of the application that contain communication.

The analysis is completed through the use of a communication visualization tool presented here. The analysis tool reads the resulting trace from the application run and generates visual communication matrices of different loops within the application. For nested loops, the communication matrix can be the aggregate of one or more nested loops, or each subloop can be analyzed individually. The communication volume is represented by a color range in the matrix plots. The analysis is used to extract the working sets for two applications, CG and MG, and efficiently map the communication onto an optical circuit switch for improved performance and message delay.

The remainder of this document is organized as follows: Relevant previous work is discussed in Section 2. Section 3 presents the compilation flow for generation of traces containing code source information. The visualization tool is described in some detail in Section 4. Section 5 provides a case study of using the visualization tool with the MG application. Section 6 presents the results from analysis of parallel applications and benchmarks. Conclusions and future research directions are presented in Section 7.

*This work is in the context of the PERCS project at IBM, which is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH3039004.

2 Related Work

There have been several attempts [3, 10, 18, 19] to understand the communication characteristics of parallel applications. Shires, et. al. presented an algorithm for building a program flow graph representation of an MPI program [18]. In [19], Vetter and Mueller examined the explicit communication characteristics of several sophisticated scientific applications, while focusing on the Message Passing Interface (MPI) [14]. Faraj and Yuan [10] investigated the communication characteristics of MPI implementations of the NAS parallel benchmarks [3].

Many research projects require information about communication patterns. For example, Cappello and Germain proposed an approach to associate compiled communications and a circuit switched interconnection network [5]. Yuan, et. al. explored using compiled communication as an alternative to dynamic network control [22]. Dietz and Mattox studied the Flat Neighborhood Network (FNN) which uses the communication patterns to determine the design of the network [7]. As earlier described, our previous work introduces a switch design which can use our compilation technique to pre-program a time-division multiplexing (TDM) network switch [8]. All of the above efforts need the precise knowledge of communication patterns to reduce overhead in the network.

3 Trace Annotation with Code Artifacts

Much of the analysis of communication requirements and patterns is currently completed by the analysis of the statistics from a parallel application's execution. These application traces are generated through a variety of techniques. One popular example is to create an annotated set of communication libraries that produce trace outputs during execution. Generally, these annotated libraries consist of a set of wrappers that contain the trace file output followed by forwarding the actual communication to the underlying communication libraries.

This sort of technique has been used by many researchers to study different parallel applications of interest on a variety of parallel platforms. Unfortunately, these techniques often do not reveal the underlying regularity of the application and make the application appear to have either a large communication degree or a fairly random communication pattern. For example, CTH developed at Sandia National Laboratories (SNL) is an application that models the properties of materials under strong shocking collisions using a multidimensional representation of the items involved [6].

CTH can operate in a static partitioning mode or in a dynamic repartitioning mode. For the dynamic partitioning, CTH fluctuates between a communication degree of approximately 8 and 110 nodes with a typical range in the 20s for a 128 run [4]. Figure 1 shows a graphical plot of the communication matrix from a run of reduced functionality CTH. Each point in the matrix represents a connection between row and column processor ids. The brighter a point the higher the volume of traffic that occurred between that pair of nodes during the run. In spite of the seemly chaotic communication from Figure 1, the communication is actually predictable to a high degree of accuracy [12]. This predictability is due in large part to temporal locality in the code.

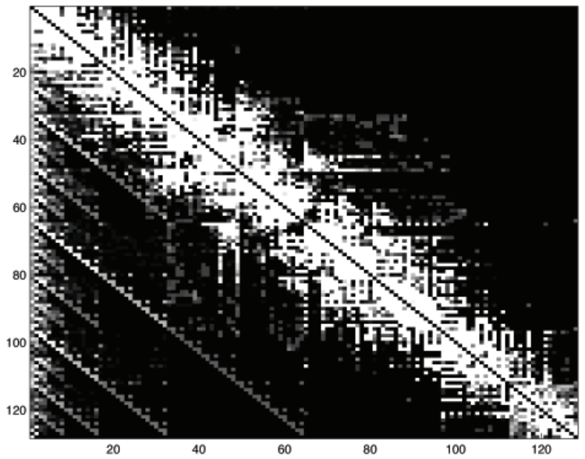


Figure 1. CTH communication matrix for 128 processor run with dynamic partitioning. Heavy traffic is in white and black represents no traffic.

The temporal and spatial locality in serial applications is widely studied, principally to improve application performance from techniques such as caching data to improve performance in processors. The locality generally comes from loop structures in the code which repeat code segments over and over and change data access patterns in a regular way so as to be able to use optimizations such as data prefetching and caching to reduce memory access bottlenecks. Several loop transformations such as interchange, fusion, and tiling have been proposed to improve locality in different situations [15].

The locality found in CTH implies that this technique can be leveraged for even largely seemingly chaotic communication as discovered by a trace analysis. To discover the inherent locality and predictability, it is necessary to retain information about the structure of the code during the trace analysis.

3.1 Compiler-assisted Trace Generation

To study the communication patterns of parallel programs a compiler was constructed for two main purposes: (1) discovery of communication patterns at compile time and (2) analysis of parallel applications with annotated traces. A compiler prototype has been developed based on the SUIF compiler [21]. The SUIF compiler is an open source, source to source compiler for both the C and Fortran 77 languages.

Figure 2 shows the paradigm of our compilation framework. The front end of SUIF compiles parallel applications into the SUIF intermediate format. We leverage many of the built in compiler passes provided with SUIF to perform ba-

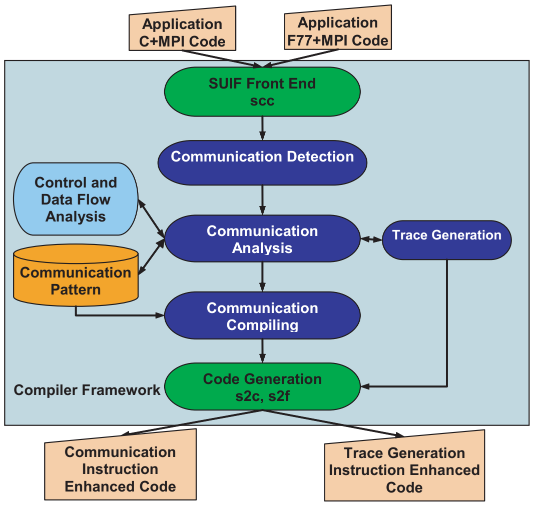


Figure 2. The compilation framework.

sic transformations, such as copy propagation, constant propagation, etc., to support the communication analysis. Much of the infrastructure shown in Figure 2 supports the detection of communication patterns in the application at compile time. For example, once a program is compiled into the SUIF intermediate representation (IR), the MPI function calls are discovered in the Communication Detection phase. The Communication Analysis phase breaks the program up into communication phases for pattern detection. Finally, the Communication Compiling phase discovers the communication pattern of each phase and generates new instructions to help configure the network to improve performance. This is the Communication Instruction Enhanced Code output that is shown in Figure 2. A detailed study of this technique is described in [17].

There are several challenges with this approach, aside from the difficulties of implementing some rather sophisticated compiler analyses such as inter-procedural analysis. Determination of an appropriate communication phase requires a way to determine temporal locality of communication. This locality may not always be obvious at compile time, particularly if many of the communication operations are dynamic, i.e. depend on input data into the application to be determined.

The Trace Generation component shown in Figure 2 provides a path to study what makes a proper phase by correlating the code structure with the communication behavior of the application during a run. Once MPI calls are detected in the compiler, the trace generation phase inserts printing instructions for both the MPI calls and the entrance and exit from important code structures such as loops and conditional statements. Thus, the Trace Generation Instruction Enhanced Code shown on the right output path provides an annotated program, which when run on a parallel machine will produce a trace of the communication and code profile of the execution. Different runs with different datasets can produce different trace results without recompiling the program.

An example of the trace output of a segment of the MG

```
subroutine comm3(u,n1,n2,n3,kk)
           :
  do 151 axis = 1, 3
     if( nprocs .ne.  1) then
           :
        call give3( axis, +1, u, n1, n2, n3, kk )
        call give3( axis, -1, u, n1, n2, n3, kk )
           :
     endif
151  continue
           :
           :
subroutine give3( axis, dir, u, n1, n2, n3, k )
           :
  if( axis .eq.  1 )then
     if( dir .eq.  -1 )then
        do 66 i3=2,n3-1
           do 65 i2=2,n2-1
              buff_len = buff_len + 1
              buff(buff_len,buff_id ) = u( 2,  i2,i3)
65         continue
66      continue
     call mpi_send(
     > buff(1, buff_id ), buff_len,dp_type,
     > nbr( axis, dir, k ), msg_type(axis,dir),
     > mpi_comm_world, ierr)
  else if( dir .eq.  +1 ) then
     do 68 i3=2,n3-1
        do 67 i2=2,n2-1
           buff_len = buff_len + 1
           buff(buff_len, buff_id ) = u( n1-1, i2,i3)
67      continue
68   continue
     call mpi_send(
     > buff(1, buff_id ), buff_len,dp_type,
     > nbr( axis, dir, k ), msg_type(axis,dir),
     > mpi_comm_world, ierr)
     endif
  endif
           :
```

Figure 3. Code segment from MG.

program shown in Figure 3 is shown in Figure 4. The first line of this trace excerpt shows the entrance to loop 151 in the code. Because this is Fortran 77 code, each loop has a label, which is retained for the trace file. In C applications, the label is replaced by an identifier generated by the compiler, such as the line number. The remainder of the line shows that this line is from the comm3 function of mg.f at line 1070. Similarly, an IF statement is entered before we see our first MPI call on line 3. This call is an mpi_irecv, and the parameters sent to the function are shown in parenthesis. Again the function name, filename, and line number are displayed in brackets (this is wrapped onto line 4). The next entry indicates timestamp information. The first entry is the ending time of the last MPI call, the second is the starting time of the current MPI call.

The [D] preceding each of these calls indicates that the compiler has determined that these MPI calls are dynamic (i.e. cannot be determined at compile time). This makes sense, as the calls are nested in conditionals, which may depend on input data to determine the remote node that is involved in the

```
--- LOOP_151 starts --- [#1070@comm3_@"mg.f"]
--- IF_152 starts --- [#1072@comm3_@"mg.f"]
[D]mpi_irecv__(count=8712 datatype=27 src=-2
tag=1100 Communicator=91 request=133)
[#1167:ready_:"mg.f"] [ 1793.078235000
1793.078257000 ] { typesize = 8 }
[D]mpi_irecv__(count=8712 datatype=27 src=-2
tag=1300 Communicator=91 request=134)
[#1167:ready_:"mg.f"] [ 1793.245772000
1793.245788000 ] { typesize = 8 }
--- IF_163 starts --- [#1196@give3_@"mg.f"]
--- IF_164 starts --- [#1198@give3_@"mg.f"]
--- IF_167 starts --- [#1212@give3_@"mg.f"]
[D]mpi_send__(count=2048 datatype=27 dst=42
tag=1300 Communicator=91) [#1224:give3_:"mg.f"]
[ 1793.245956000 1794.798499000 ] { typesize = 8 }
+++ IF_167 ends +++ [#1224@give3_@"mg.f"]
+++ IF_164 ends +++ [#1224@give3_@"mg.f"]
+++ IF_163 ends +++ [#1224@give3_@"mg.f"]
:
:
+++ LOOP_151 ends +++ [#1083@comm3_@"mg.f"]
```

Figure 4. Example trace output from MG.

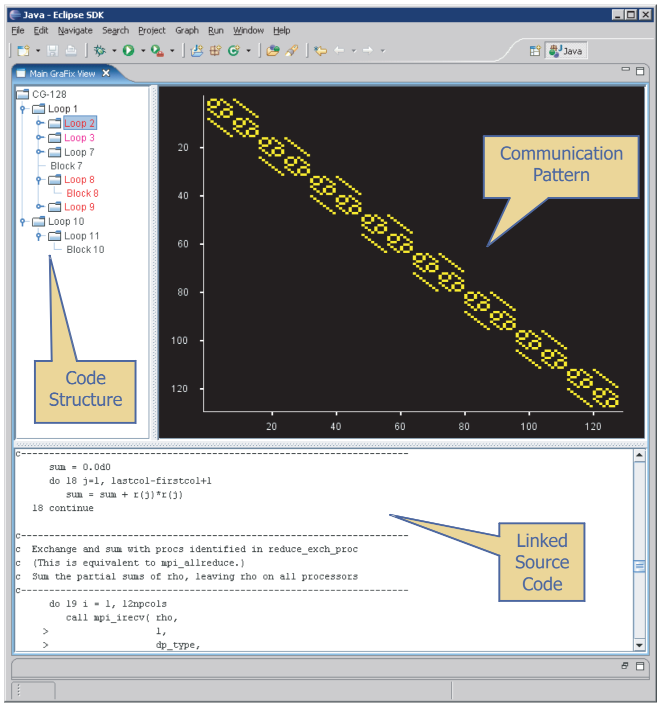
communication. However, while the nodes may appear dynamic to the compiler, they may in fact end up being regular during execution. For the trace information to be useful, it is necessary to create a technique to visualize the trace information, including relationships to the code, so that the information can be analyzed.

4 Visualization Tool Flow

In order to help graphically display the communication patterns that occur during the application execution, a visualization tool was created and is shown in Figure 5. The tool is written in Java and two versions exist: (1) a Java standalone version and (2) an Eclipse based version designed to integrate into the IBM Parallel Tools Platform [11, 20]. The tool takes both the trace and the original program source code as input. It produces three major views, in the top left segment labeled *Code Structure*, the structure of the application is shown. This view shows the different loops that comprise the program and their nesting order and code segments that reside between loops. Loops can be expanded or contracted to show nested structures. By selecting one or more loop or block structures, a graphical representation of their communication pattern is shown in the upper right view labeled *Communication Pattern*. Additionally, the corresponding source code to the first loop/block selected is shown in the bottom view labeled *Linked Source Code*.

The communication pattern representation utilizes a color scheme to represent the volume of traffic. This scale is shown in Figure 6 modeled after the colors given off by increasingly hot items starting with black moving into reds, yellows, and finally reaching white.

The visualization tool flow shown in Figure 7 contains a summary of the four major steps to go from a parallel program



```
c-----------------------------------------------------------------
      sum = 0.0d0
      do 18 j=1, lastcol-firstcol+1
         sum = sum + r(j)*r(j)
   18 continue

c-----------------------------------------------------------------
c  Exchange and sum with procs identified in reduce_exch_proc
c  (This is equivalent to mpi_allreduce.)
c  Sum the partial sums of rho, leaving rho on all processors
c-----------------------------------------------------------------
      do 19 i = 1, l2npcols
         call mpi_irecv( rho,
     >                   1,
     >                   dp_type,
```

Figure 5. Visualization tool overview.

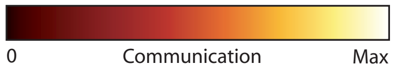
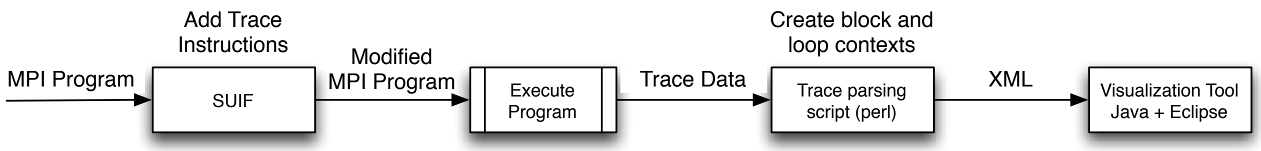


Figure 6. Traffic scale based on the *hotness* of the color.

into the final visual representation. The first component uses a SUIF pass to add trace generation instructions into the program as described in Section 3.1. Once the trace is generated it is parsed and processed with a script to extract information about the communication patterns, code structures, and application contexts for visualization. This information is stored in an XML file and fed into the visualization tool.

During trace processing, two major code structures of interest are currently extracted from the traces: *loops* and *blocks*. Loops describe segments of code from a `for`, `while`, or `do-while` loop in C and `do` loops in Fortran. Blocks are defined as segments of code between or within loops that contain MPI communication function calls.

After all trace files have been processed, matrices to represent the communication pattern are generated. These matrices are $n * n$, where n is the number of processing nodes in the system. Each element of the matrix stores the communication volume of the point-to-point communications between each pair of nodes in that specific contextual block throughout the execution of the application. Collective communications can be decomposed into point-to-point communications and added to the communication matrices if desired.



MPI Program
Add Trace Instructions
SUIF
Modified MPI Program
Execute Program
Trace Data
Create block and loop contexts
Trace parsing script (perl)
XML
Visualization Tool Java + Eclipse

Figure 7. Visualization flow.

5 Case Study of the MG Application

The multigrid algorithm (MG) is one of the applications included in the NAS Benchmarks. Multigrid methods are fast linear iterative solves typically used for generating numerical solutions to elliptic partial differential equations in two or more dimensions [1, 9]. In this section we will use the visualization tool to study the behavior of the MG application and show how this information can be used to optimize the application for execution on an example system that uses an optical circuit switched interconnect.

5.1 Performance with Optical Circuit Switching

Optical circuit switching (OCS) has recently been proposed as a high-performance and low cost alternative to direct networks, fat-tree networks, and packet switched networks currently popular for high-performance computing systems [4]. Optical switches typically use mirrors controlled with Micro-Electro-Mechanical Systems (MEMS) to configure light paths that allow the establishment of the optical circuits within the system. Optical switches provide very high bandwidth and low-latency at a much lower cost than their electronic counterparts. The main drawback of optical switches is their switching latency, which can be on the order of milliseconds. Thus, for relatively long lived connections, optical switching is very efficient. However, for shorter or less predictable connections, such as synchronization and some types of collective communication, optical switching does not work well.

The OCS system leverages the benefits and cost advantages of optical switches and combines it with a relatively low-speed (and thus low cost) electronic network to handle the less predictable and low-volume communications. The goal is to provide the performance of a high-performance packet switch network without the cost of building a fully-buffered electronic solution.

An overview of the OCS network is shown in Figure 8. Processing nodes are directly connected to a number of high bandwidth optical switches or *switch planes*, represented by the thick lines. These switches can be upwards of 10's of gigabits/s while remaining cost-effective. Each node is also connected to a relatively low-speed electronic packet switch represented by the thin lines. This network would be simpler to save cost and might be on the order of Gb/s.

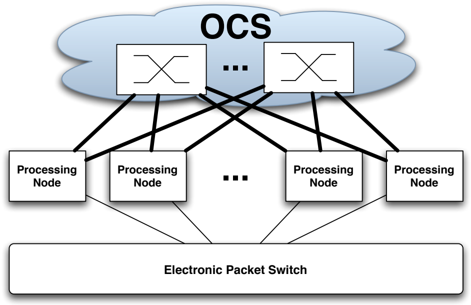


Figure 8. Combining optical circuit switching with electronic packet switching.

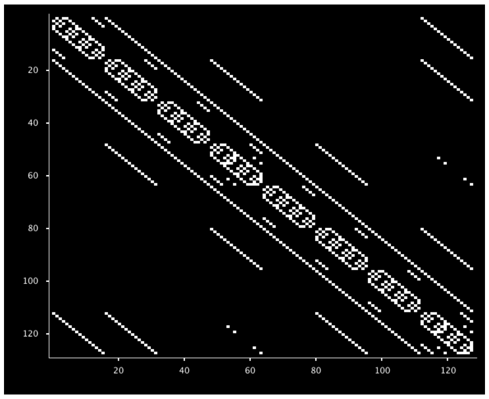


Figure 9. Communication matrix for MG benchmark.

The communication matrix for the entire execution of MG is shown in Figure 9. Based on the study of this application, the communication degree reaches 11 different destinations. It might be possible to conclude that to most efficiently implement this algorithm requires 11 independent switch planes. However, by studying the algorithm more closely, it is possible to implement the algorithm with six planes without loss of efficiency. By adding just one more switch plane to 12, it may also be possible to get twice the performance from a naive 11 plane implementation.

Figure 10 shows the communication pattern of the do loop labeled 36 by the Fortran application MG. In many ways this loop is representative of loops found in many of the NAS

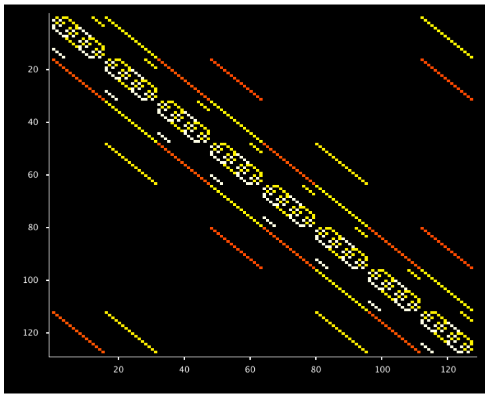


Figure 10. Loop 36 of MG.

benchmarks insomuch as its communication pattern is very regular, as reflected by its symmetry and patterns. However, this loop provides some interesting insights into the MG application. First, this loop has a variety of volumes associated with messages as shown by the traffic scale shown in Figure 6. For example, messages range from a medium volume orange, to a fairly high volume yellow, up to the maximum traffic white. Second, the communication degree per node is not fixed. There are clusters of 16 nodes that have a communication degree of six and others with degree of eight. However, by analyzing the communication in this manner, it might be possible to get away with a high bandwidth circuit switch network of six planes because the other two communication directions have lower traffic requirements. These lower bandwidth communications could be satisfied with a lower bandwidth packet switch network.

In this case, the top level loop, loop 36, may not be the best choice for creating a communication phase. The loop is somewhat poorly behaved, requiring different numbers of communication partners on a node by node basis, and requiring non-uniform traffic volume. Figure 11 shows nested subloops for loop 36. By investigating these subloops and blocks individually, it is possible to realize these communication requirements more efficiently. While implementing all of loop 36 from Figure 10 requires a minimum of 8 switch planes, each of the subloops and blocks require a maximum of 6 switch planes. Loop 151 and block 9 require all six connections for each node. Loop 159b can be implemented with only 2 planes, and the other four planes can be used to triple the bandwidth between these nodes. Loop 159a requires six planes even though the connectivity varies between zero and six depending on the node.

6 Results

The OCS network is a natural target for applications profiled with the visualization tool flow, as it allows the identification of long lived connections within loops in the application. Through analysis of the communication pattern of the application assisted by the visualization flow, important communication links were identified. These links were provided to our OCS network simulator to pre-establish the optical circuits and this was compared to a runtime strategy for establishing optical links.

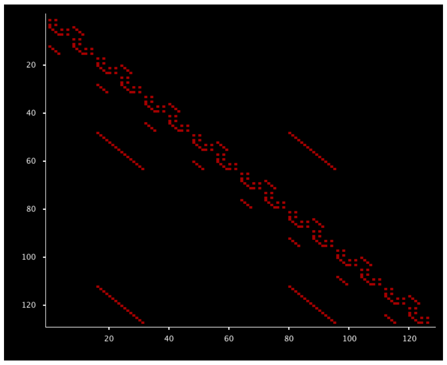
The runtime strategy for circuit establishment required messages to be larger than a particular threshold to be considered for optical switching. For the runtime strategy, least frequently used circuits were selected for replacement by new circuits that exceeded the threshold. While more complicated replacement policies were possible, policies that considered connection lifetimes, rolling threshold windows, etc. were not found to provide a significant benefit over the simpler scheme shown here.

Figure 6 shows the execution time (Figure 12(a)) and average message delay (Figure 12(b)) for executing the MG application on 128 processors using the OCS interconnect. The simulations were run varying the number of available optical planes from 1 to 11 planes. 11 planes was selected as the maximum because this allows all possible point-to-point communications to be allocated its own optical circuit.

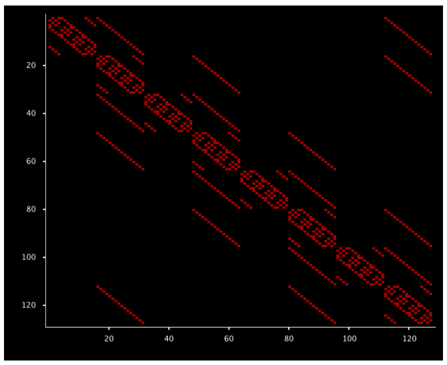
The required point-to-point connections were ranked into three orders using information from the visualization tool, the best ranking uses a ranking by message volume, the worst ranking uses a ranking by the reversed message volume, and the indexed order ranking uses a ranking in order of destination index, which represents an arbitrary ordering between the other two. In the simulator, these point to point connections are applied to optical switch planes in these orders. When a connection causes a conflict, the next connection in the list is established. If all optical switch planes are consumed, the communication from this connection is satisfied using the slower electronic network.

In all cases, as optical switch planes were added to the OCS interconnect, the runtime and average message delay decreased. However, the order of priority for adding the connections into the OCS was particularly important toward the effectiveness of adding more switch planes. Both the best configuration and indexed configuration showed significant improvement over the runtime configuration. The worst case, as expected, does not perform well in this case, particularly compared with the other preloaded configurations. This technique does not make sense for actual implementation but does provide a lower bound for comparison purposes.

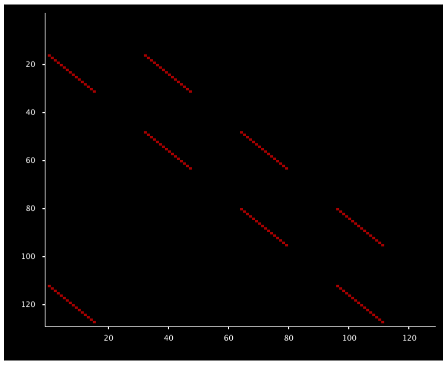
Figure 6 shows the results of running the CG application using the OCS network. In this case, the communication is more balanced between the highest and lowest volume point-to-point communication pairs. It also turns out that the indexed ordering is the best ordering. Thus, the difference between best and worst is far less pronounced than for MG and the best and indexed trends are identical. In fact, for a single switch plane the worst configuration is actually the best performance overall. In all cases the preloaded configurations beat the runtime configuration (excepting for five switch planes where the worst configuration has the poorest perfor-



(a) MG loop 159a.



(b) MG loop 151 and block 9.



(c) MG loop 159b.

Figure 11. Loop 36 decomposed.

mance and delay). For all cases of two switch planes or more, the indexed/best configuration either performs comparably to or better than the worst case or the runtime scheduler.

7 Conclusions

This paper presented a visualization tool designed to correlate the communication pattern at different segments during the execution with the code that causes its execution. Rather than just looking at the communication during a time window, loops that contribute to communication are identified using a compiler. The program is then annotated with trace generation instructions. During each execution, the program creates traces which can be read by the visualization tool flow and present a graphical depiction of the communication related back to the code in the application.

Through analysis of several applications, we showed how looking at the entire application communication pattern can lead to inappropriate analysis of interconnection needs. By appropriate analysis using the visualization flow, we showed how the program execution can be improved and message delay reduced, particularly compared to runtime scheduling techniques.

Some planned future directions are to examine the impact of collective communications in the communication patterns. Additionally, through the use of techniques in [17], we plan to automatically generate the communication patterns statically in the compiler. The pattern can be viewed in the visualization tool without needing to actually run the application and generate a trace. Additionally, since the visualization tool is integrated within the Eclipse framework, it may be possible to also integrate the compiler within the flow and generate the patterns when possible at compile time and allow the trace based flow to fill in the missing patterns when they become available.

References

[1] A. M. Abdalass, J. F. Maitre, and F. Musy. A multigrid solver for a stabilized finite element discretization of the Stokes problem. In W. Hackbusch and U. Trottenberg, editors, *Multigrid Methods II*, pages 1–6, Berlin, 1986. Springer–Verlag.

[2] A. Afsahi and N. J. Dimopoulos. Efficient communication using message prediction for clusters of multiprocessors. *Concurrency and Computation: Practice and Experience*, 12(1):41–50, 2002.

[3] D. Bailey, T. Harris, W. Sahpir, and R. van der Wijingaart. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, December 1995.

[4] K. J. Barker, A. Benner, R. Hoare, A. Hoisie, A. K. Jones, D. J. Kerbyson, D. Li, R. Melhem, R. Rajamony, E. Schenfeld, S. Shao, C. Stunkel, and P. A. Walker. On the feasibility of optical circuit switching for high performance computing systems. In *Proc. of SuperComputing (SC)*, 2005.

[5] F. Cappello and C. Germain. Toward high communication performance through compiled communications on a circuit switched interconnection network. In *Proc. of HPCA*, pages 44–53, 1995.

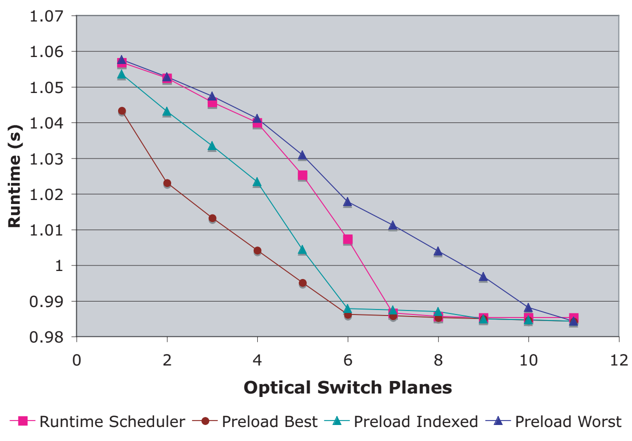
[6] D. A. Crawford, P. A. Taylor, and E. Hertel. Adaptive mesh refinement in the cth shock physics hydrocode. In *Proc. of New Models and Hydrocodes for Shock Wave Processes in Condensed Matter*.

[7] H. G. Dietz and T. Mattox. Compiler techniques for flat neighborhood networks. In *Proc. of 13th Int. Workshop on Languages and Compilers for Parallel Computing*, 2000.

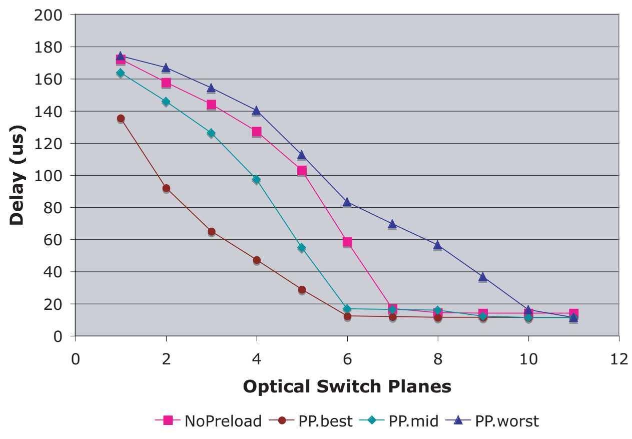
[8] Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem. Switch design to enable predictive multiplexed switching in multiprocessor networks. In *Proc. of IPDPS*, 2005.

[9] C. C. Douglas and M. B. Douglas. MGNet Bibliography. Department of Computer Science and the Center for Computational Sciences, University of Kentucky, Lexington, KY, USA and Department of Computer Science, Yale University, New Haven, CT, USA, 1991–2002 (last modified on September 28, 2002); see http://www.mgnet.org/mgnet-bib.html.

[10] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In *Proc. of PDCS*, 2002.

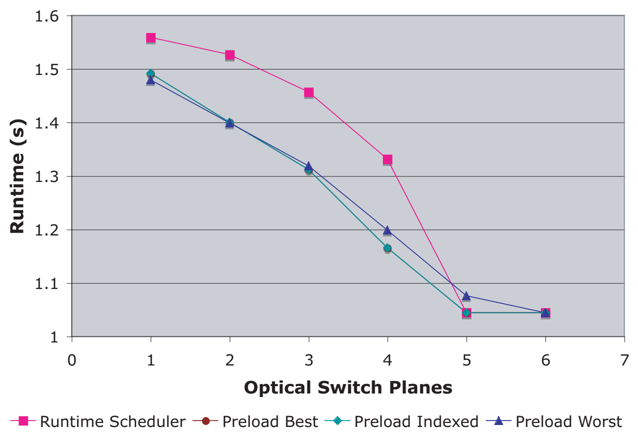


(a) MG completion time.

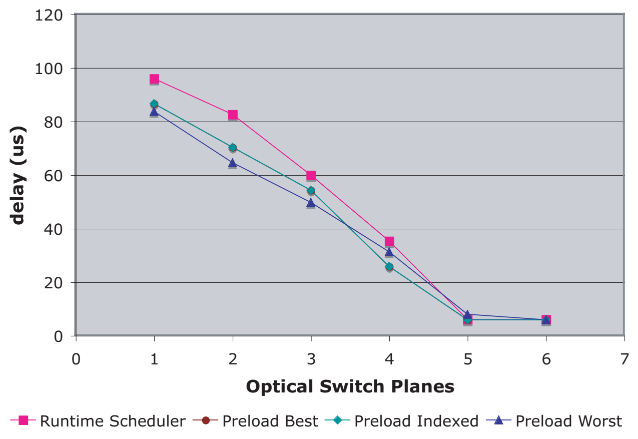


(b) MG delay.

Figure 12. MG with high-speed optical switch planes and a slower electronic network.



(a) CG completion time.



(b) CG delay.

Figure 13. CG with high-speed optical switch planes and a slower electronic network.

[11] Eclipse platform technical overview. Technical report, IBM Corporation and The Eclipse Foundation, December 2005. www.eclipse.org.

[12] A. K. Jones, J. Zhang, and A. Amer. Entropy based evaluation of communication predictability in parallel applications. *Trans. on Information and Systems*, E89-D(2), Feb. 2006.

[13] J. Kim and D. J. Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In G. Goos, J. Hartmanis, and J. Leeuwen, editors, *Proc. of the Second International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, pages 202–216, 1998.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[15] S. Muchnick. *Advanced Compiler Design and Implemenatation*. Morgan Kaufmann, 1997.

[16] C. Salisbury and R. Melhem. A high speed scheduler/controller for unbuffered banyan networks. *Computer Communications Journal*, 24(9):1158–1169, 2001.

[17] S. Shao, A. K. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *Proc. IPDPS*, 2006.

[18] D. Shires, L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of mpi programs. In *Proc. of PDPTA*, June 1999.

[19] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *JPDC*, 63(9):853–865, September 2003.

[20] G. R. Watson and C. E. Rasmussen. A strategy for addressing the needs of advanced scientific computing using eclipse as a parallel tools platform. Technical Report LA-UR-05-9114, Los Alamos National Laboratory, P.O. Box 1663, MS B287, Los Alamos, NM 87545, December 2005.

[21] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. s. Lam, and J. L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. In *SIGPLAN Notices*, 1994.

[22] X. Yuan, R. Melhem, and R. Gupta. Compiled communication for all-optical TDM networks. In *Proc. of SC*, 1996.