Symbolic Expression Analysis for Compiled Communication*

Shuyi Shao[1], Yu Zhang[2], Alex K. Jones[3], Rami Melhem[4]

[1,4]University of Pittsburgh
Department of CS
Pittsburgh, PA 15260 USA
{[1]syshao, [4]melhem}@cs.pitt.edu

[2,3]University of Pittsburgh
Department of ECE
Pittsburgh, PA 15260 USA
[2]yuz28@pitt.edu,[3]akjones@ece.pitt.edu

Abstract

Enabling circuit switching in multiprocessor systems has the potential to achieve more efficient communication with lower cost compared to packet/wormhole switching. However, in order to accomplish this efficiently, assistance from the compiler is required to reveal the communication pattern in the parallel application. In this paper we present symbolic expression analysis techniques in a MPI parallel compiler. Symbolic expression analysis allows the identification and representation the communication pattern and also assists in the determination of communication phases in MPI parallel applications at compile-time. We demonstrate that using the compiler analysis based on symbolic expression analysis to determine the communication pattern and phases provides an average of 2.6 times improvement in message delay over a threshold-based runtime system for our benchmarks with a maximum improvement of 9.7 times.

1. Introduction

As the size of multiprocessor systems continues to scale higher, the cost of the interconnection networks potentially dominates the cost of the entire system. Therefore, low cost alternatives to current popular interconnects that provide comparable performance become attractive. Circuit switching networks have the potential of achieving higher efficiency than packet and wormhole networks with relatively lower cost. As such, enabling optical circuit switching in the high performance computing domain has drawn the interest of several researchers [1, 9, 11].

However, the overhead of circuit establishment can be relatively large for optical circuit switching networks. Thus, the benefits of circuit switching can only outweigh its drawbacks when communication exhibits locality and when this locality is appropriately exploited. Thus, to effectively leverage optical circuit switching, a method to determine the pattern of communication in the system is necessary. The two main approaches are the use of a compile-time analysis and the use of run-time analysis.

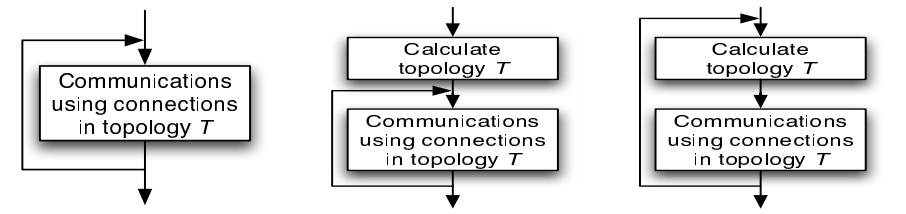
In this paper we explore using compiler techniques to explore the communication patterns. This has been shown to be a promising approach for achieving efficient communications in the high performance computing domain [10,11]. One of the key issues of this approach is the extraction of the communication pattern in a parallel application from its source code.

Unlike traditional compiler passes which do things like manipulate constants or reorder code for faster execution, our goal is to analyze the communication operations within a parallel application. In many cases the expression for a communication destination or volume will contain unresolved variables. Thus, our approach manipulates symbolic expressions within the compiler to determine how these values are calculated. Then the knowledge of the communication pattern is used to pre-configure the interconnection network in the circuit switching enabled high performance computing system.

In this paper, we present the techniques of symbolic expression analysis within the compiler. We demonstrate how symbolic expression analysis can be exploited in a compiled communication framework for parallel applications. We demonstrate how symbolic expressions can be created starting with simple blocks containing no control flow, and how this can be expanded to blocks containing conditionals and loops.

Our compiled communication approach targets MPI [7] parallel applications. MPI programs are written in Single-Program-Multiple-Data (SPMD) style. Each MPI process independently executes the same program on its private data. Nevertheless, often different MPI processes take different execution paths, which when analyzed using execu-

*: This work is partially supported by NSF award number 0702452



(a) Static: T is resolvable at compile-time.

(b) Persistent: T is calculated before a phase.

(c) Dynamic: T changes inside a phase.

Figure 1. Static, persistent and dynamic communications when a phase is a loop.

tion or trace-based methodologies can create different results. In contrast our technique extracts the communication pattern represented through symbolic expressions from an MPI parallel application source code.

1.1 Communication Locality

Applications tend to exhibit locality of memory access, which is exploited by caches. This locality is due to loop structures in the code and leads to the subdivision of the application into computational phases typically based on loop structures [3]. Much like the computational phases within the application, for parallel applications, there exists a similar locality of communication. This locality leads to communication phases within the application again fundamentally built from loop structures in the code [11].

Thus, we classify communications during a phase of an application's execution into three categories: *static*, *persistent* and *dynamic*. In this context, the topology of communication is the specification of the source and destination of the messages exchanged. Figure 1 describes each type of communication in the context of a single loop. Each type of communication is described as follows:

Static – Communication is static if it can be completely determined through compile-time analysis. That is the compiler can identify both the temporal locality and the exact topology of the communication. This is shown in Figure 1(a).

Persistent – Communication is persistent if, though the compiler cannot determine the exact topology of communication, it can determine that the topology does not change during the phase. That is, its temporal locality can be identified by the compiler, but its spatial properties remains unknown until run-time. This is shown in Figure 1(b).

Dynamic – Communication is dynamic if it is neither static nor persistent. In the example from Figure 1(c), this is illustrated by a communication call whose source and destination change between loop iterations.

In typical parallel applications the communication operations are either static or persistent [11]. A compiler can take advantage of these characteristics to identify and expose the communication pattern in the parallel application. This information can be exploited by a circuit switching interconnect to improve the efficiency of the communication and as such improve the application performance. Through existing compiler techniques, such as inter-procedural constant propagation and folding, staticly known communications can be resolved. However, to leverage persistent communication, the compiler requires symbolic expression analysis. Symbolic expression analysis creates expressions for unresolved variables used to represent the communication topology.

The remainder of this paper is organized as follows: Related work is described in Section 2. In Section 3 we briefly introduce the compilation paradigm of the compiled communication framework. Following that we provide the details of our symbolic expression analysis techniques in Section 4. We demonstrate how the communication patterns extracted from the applications in part due to the symbolic expression analysis can exploit a circuit switching interconnect more efficiently than a runtime system. Finally, we relate some conclusions in Section 6.

2 Related Work

Symbolic analysis was a hot topic in parallel compiler research in the middle of 1990s. Several researchers studied symbolic analysis techniques in the context of parallel compilation for High Performance Fortran (HPF)-like programs [5, 13]. Most of the work emphasizes the parallelization of the program with the assistance of symbolic analysis with a focus on loops and arrays. Additionally, Yuan *et al.* explored using compiled communication for HPF-like parallel applications as an alternative to dynamic network control [16].

To the best of our knowledge, as message passing pro-

grams written in Single-Program-Multiple-Data (SPMD) style has become one of the dominant parallel programming models, there is no significant research of applying symbolic analysis to this class of applications. Shires *et al.* presented an algorithm for building a program flow graph representation of message passing applications [12]. This work provides an interesting basis for important program analysis useful in software testing, debugging and code optimization of an MPI program.

There has been additional related work in the area of compiled communication. Cappello and Germain proposed an approach to associate compiled communications and a circuit switched interconnection network [2]. Our own previous work describes the details of an MPI based compilation approach [10, 11] and how it can be applied to a time-division multiplexing (TDM) switching architecture [4].

3. Compilation Paradigm

The symbolic expression analysis concept studied in this paper is part of a larger compilation framework that includes standard compiler techniques such as the creation of control and data flow graphs (CDFGs), constant propagation and folding, inter-procedural analysis, array analysis, etc [8]. However, the compiler incorporates some analysis specific to the construction of parallel applications required compose the communication pattern. We presented the design of the parallel compiler in [10, 11] and in this section we provide a brief overview of the compilation paradigm. The symbolic expression analysis described in detail in this paper is not part of our previous work. However, our previous work does mention how symbolic expression analysis could be used in our compiler.

Our experimental compilation flow is shown in Figure 2. The compiler is built within the SUIF research compiler infrastructure from Stanford University [14]. First the compiler identifies the MPI functions in the code and uses various analyses to construct a communication pattern for the application. Using in part symbolic expression analysis, the compiler infers and composes the communication pattern and according to particular interconnection network specification, the communication pattern identified can be leveraged through network configuration instructions inserted into the application.

Using the details of the communication analysis and the details of the target interconnect, the application is broken up into communication phases. This process uses loop structures as the basis for constructing phases and combines nearby structures into larger phases when their communication patterns and the network capacity allow it. A detailed algorithm for constructing phases is beyond the scope of this paper and is described in [10].

For static communication patterns, the compiler generates a network configuration file that can be used by the loader and runtime system to program the network. It also inserts network configuration setup instructions into the program prior to the change in communication phase. For persistent communication patterns, the compiler inserts symbolic network configuration instructions resolved at runtime prior to the communication phase. Thus, the runtime system can pre-establish the needed connections prior to the actual communication operations. The technique for creating these symbolic expressions is described in the following section.

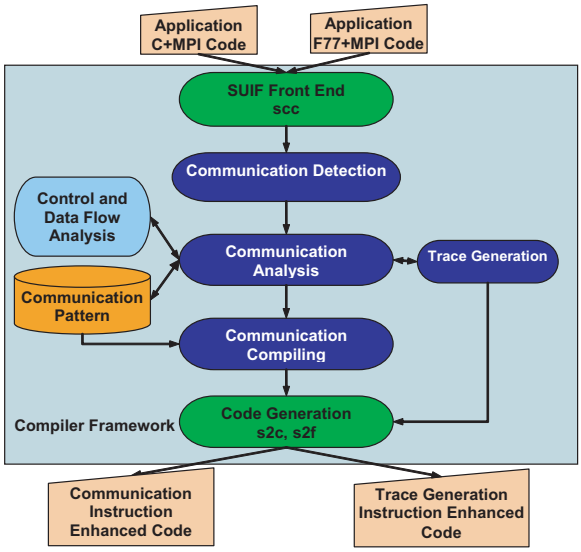


Figure 2. Experimental compilation flow.

4. Symbolic Expression Analysis

To complete symbolic expression analysis we use a control and data flow graph representation of the program execution [8]. The construction of the CDFG is described in Section 4.1.

4.1 CDFG

CDFGs are essential for many types of compiler analysis. A control flow graph (CFG) is a directed graph representation of all possible traversal paths within a program during its execution. In the graph each node represents a basic block. Each directed edge represents a possible control path (*e.g.* a branch or jump instruction). There are two special "pseudo-nodes" in a

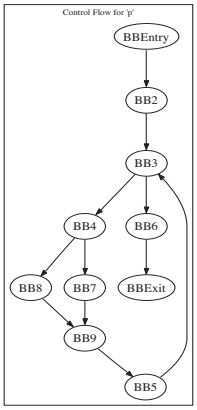


Figure 3. CFG.

CFG–the entry block, which is the unique entry point to the entire flow graph, and the exit block, which is the unique exit point leaving the flow graph. An example CFG is shown in Figure 3.

Within each basic block is a data flow graph (DFG). A DFG, which is a directed graph, represents the data dependencies within the code between control points. In a DFG, each node represents an operator (*e.g.* addition or logical shift) or an operand (*e.g.* a constant, a variable, or an array element). Each directed edge represents a data dependency that denotes the transfer of a value. An example DFG is shown in Figure 4.

4.2 Conditional Control Flow

As MPI parallel programs are written in SPMD style each processor independently executes the same program on its private data. Nevertheless, often different processors take different execution paths determined based on the wrapping branch structures. Branch structures may constrain a MPI communication operation such that only a subset of the processes perform the communication. Thus, each connection in the communication pattern is represented in the format: $source, destination, volume$. The $source$ represents the source processes that participate in the pattern; the $destination$ is the destination expression; and the $volume$ is the expression for the size of the message. If a communication operation is not constrained by a condition related to the process id or $rank$, the $source$ is set to $null$. Multiple conditions can be represented as $c_1 \wedge c_2 \wedge \ldots \wedge c_n$.

Consider the example code from Figure 5. If the branch condition is ignored (or if the condition is not related to the processor rank) the symbolic representation of the communication matrix, *A*, is:

$$\begin{pmatrix} \{null, (myrank+k)\%nprocs, 4000\}, \\ \{null, (myrank+2*k)\%nprocs, 4000\}, \\ \{null, (myrank-k)\%nprocs, 4000\}, \\ \{null, (myrank-2*k)\%nprocs, 4000\} \end{pmatrix}$$

When taking the branch condition into consideration, the actual symbolic communication matrix, *B*, is:

$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+k)\%nprocs, 4000\}, \\ \{myrank < nprocs/2, (myrank+2*k)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-k)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-2*k)\%nprocs, 4000\} \end{pmatrix}$$

At run-time the values of the variables in the symbolic expressions are resolved. For the values `nprocs=8` and `k=1`, the corresponding communication matrices for symbolic matrices A and B are shown in Figure 6(a) and Figure 6(b), respectively. Note that because the $source$ field in symbolic matrix *A* is $null$ it is necessary to calculate the destinations for each process. However for symbolic matrix *B*, we perform the same operation limited to those processes that satisfy the $source$ condition.

```
void p(int k) {
int i;
reconfigNetwork(config1);
for(i=0; i<1000; i++) {
  if (myrank < nprocs/2) {
    MPI_Send(buf,1, MPI_INT_TYPE,
      (myrank+k)%nprocs,1000,COMM);
    MPI_Send(buf2, 1, MPI_INT_TYPE,
      (myrank+2*k)%nprocs,1000,COMM);
  } else {
    MPI_Send(buf,1,MPI_INT_TYPE,
      (myrank-k)%nprocs,1000,COMM);
    MPI_Send(buf2, 1, MPI_INT_TYPE,
      (myrank-2*k)%nprocs,1000,COMM);
} } }
```

Figure 5. Example code.

$$\begin{pmatrix} & 1 & 1 & & & & 1 & 1 \\ 1 & & 1 & 1 & & & & 1 \\ 1 & 1 & & 1 & 1 & & & \\ & 1 & 1 & & 1 & 1 & & \\ & & 1 & 1 & & 1 & 1 & \\ & & & 1 & 1 & & 1 & 1 \\ 1 & & & & 1 & 1 & & 1 \\ 1 & 1 & & & & 1 & 1 & \end{pmatrix} \qquad \begin{pmatrix} & 1 & 1 & & & & & \\ & & 1 & 1 & & & & \\ & & & 1 & 1 & & & \\ & & & & 1 & 1 & & \\ & & 1 & 1 & & & & \\ & & & 1 & 1 & & & \\ & & & & 1 & 1 & & \\ & & & & & 1 & 1 & \end{pmatrix}$$

(a) Resolved matrix A. (b) Resolved matrix B.

Figure 6. Example of a communication matrix.

4.3 Traversal Algorithm for Symbolic Expression Analysis

Our method for determining the symbolic expressions for each communication operation is to locate the communication operation within the CDFG. The variables representing the destination and volume are determined. Then we traverse the CDFG from this point backward to the top of the graph building the symbolic expression as we go. Source conditions are appended during the traversal as they are discovered in the graph. The traversal completes when all of the leaf nodes have been specified and any paths that have diverged have re-converged.

We assume that the MPI program is well structured and there are no `goto` instructions. Even with this restriction, loop structures still introduce cycles in the CFG, which create problems for traversing the CDFG. To determine the symbolic expressions it is necessary to remove cycles. To solve this problem, we take advantage of the information from the abstract syntax tree and treat the all the CFG nodes (basic blocks) in a loop as an individual super-node. The processing of loop structures will be detailed in Section 4.4. Thus, our CDFG becomes a directed acyclic graph.

As previously mentioned, each basic block in the CFG represents a DFG. Hence, we complete local symbolic

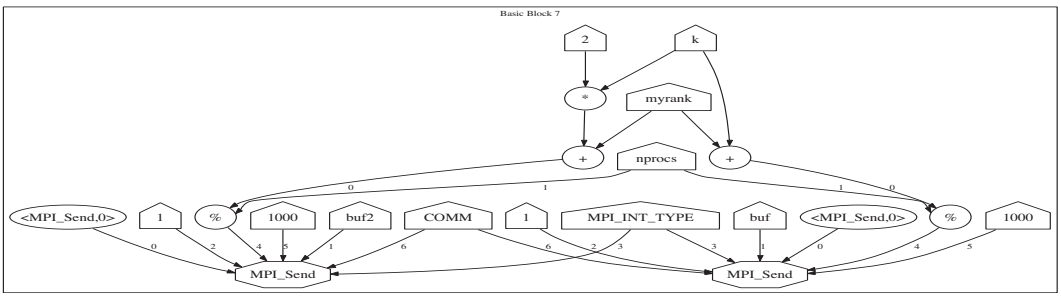


Figure 4. DFG.

expression analysis for each DFG prior to traversing the CDFG. Each DFG is by definition a directed acyclic graph. The values from input variables flow down through the graph through a number of operator nodes and finally converge at the output nodes.

The within-DFG symbolic expression analysis is completed through backward traversal along the DFG edges starting from each of the output nodes until it reaches the input nodes. The detailed algorithm is presented in Figure 7.

For example, consider the example DFG from Figure 4. The destination operation for the left `MPI_send` node is the 4th parameter, which comes from the `%` node. To build the symbolic expression we continue to traverse upward to start building a expression from the `+` node `% nprocs`. Traversing upward from the `+` node we get an expression from the `*` node `+ myrank) % nprocs`. After traversing up from the `*` the final expression becomes `(2 * k + myrank) % nprocs`.

We can describe our general symbolic expression analysis traversal algorithm in terms of the within-DFG traversal algorithm. It traverses the CFG bottom-up from program exit basic block using breadth-first search. If the application exit block is B_{exit}, Q is a queue of CFG nodes, each of which is either a basic block or a super-node representing a loop structure, and C is a list of references of symbolic connections. The traversal algorithm is detailed in Figure 8.

Consider the example from Figure 3. We begin at the BBExit node. As this is a pseudo-node, BB6 becomes our new node as it is the only predecessor. BB3-6,7-9 contains a loop, and as such is combined into a super block called BB3'. Thus symbolic expressions in BB6 are defined in terms of input variables into BB6, these variables are output variables from BB3' with symbolic expressions associated with them. As we traverse upwards, the symbolic expressions from BB3' and BB6 are combined. BB4,7-9 denote an *if-then-else* structure. During the creation of the expressions for BB3', these expressions are resolved as described in Figure 8. The expressions from BB9 are in terms of variables defined in either BB8 or BB9 or both. Thus, the ex-

```
1   do_DFG(CFG_Node B)
2    mark all DFG nodes in B as un-visited
3    foreach DFG node n in the output list do
4     generate_symbolic_expr(n)
5
6   symbolic-expression generate_symbolic_expr(n)
7    if n is visited then
8     return the symbolic expression of n
9    if n represents a call instruction then
10    foreach predecessor p that is a parameter of n do
11     generate_symbolic_expr(p)
12    switch type of n:
13    case unary-operator:
14     p=the predecessor of n
15     s1=generate_symbolic_expr(p)
16     return operator.s1
17    case binary-operator:
18     p1=the 1st predecessor of n
19     p2=the 2nd predecessor of n
20     s1=generate_symbolic_expr(p1)
21     s2=generate_symbolic_expr(p2)
22     return s1.operator.s2
23    case variable or constant:
24     return variable name or value
25    mark n as visited
```

Figure 7. Pseudocode for within-DFG symbolic analysis.

```
1  mark all CFG nodes as un-visited
2  add all predecessors of B_exit to Q
3  mark B_exit as visited
4  while (Q is not empty) do
5   remove a node n from Q
6   if (n is a basic block) then
7    global_do_DFG(n)
8   else
9    process loop /*detailed in Section 4.4*/
10  foreach predecessor p of n do
11   if (all successors of p except n are visited) then
12    add p to Q
13    propagate partial symbolic expressions from n to p
14  mark n as visited
15
16 global_do_DFG(n)
17  add all items in list c to C
18  foreach symbolic expression e in n do
19   substitute unresolved variables in e with symbolic expressions for
     the outputs of n
20  if (n is branch test) then
21   augment true condition to the source field of connections from
     then part
22   augment false condition to the source field of connections from
     else part
```

Figure 8. Pseudocode for CDFG traversal algorithm.

pressions are replicated as described in line 20 based on the condition. The traversal continues until BB2 and BBEntry are reached.

We have not discussed loops in this section except to say that they are combined into a super block to remove the cycle to allow the BFS to terminate when the entry node is reached. In the next section, we describe how the symbolic expression of an iterative structure is determined in detail.

4.4 Generating Symbolic Expressions for Loops

A large portion of MPI communication operations are embedded in loop structures. We see this characteristic in both benchmark applications such as the NAS parallel benchmarks, COMOPS, etc. as well as full blown parallel applications. This is not surprising as the loop structures in an application can often be classified as a phase of application in which communication configuration is persistent. This justifies that the communication information from loop analysis is a good candidate to configure the network. Additionally, for our symbolic expression analysis, the destination variables of MPI communication operations are usually defined and calculated in loop structures.

The goal of our communication destination analysis is to acquire a list of symbolic expressions for destination variables in MPI calls for each loop iteration or/and a final symbolic expression for destination variable after the whole loop structure has finished executing. We have two kinds of loops in our CDFG—FOR loops, whose iteration number is deterministic statically, and DO loops, which terminate based on the condition following the loop body. WHILE loops are represented as a DO loops embedded within an IF structure. Our target language constructs are formulated loops, e.g. FOR loops in C and DO loops in Fortran. The reason why we only focus on formulated loops is that we can determine their symbolic expressions through static loop analysis.

4.4.1 Loop Analysis

Note that loop structures form cycles in the CDFG. Normal graph traversal algorithms, i.e. DFS or BFS, can fall into infinite loops when cycles exist. Hence, we use an analytical approach during symbolic expression analysis when processing loops in the CDFG.

We observe that the number of symbolic analysis target variables, such as destination and volume variables, is typically small in each benchmark. We also find that these variables are initialized once in the program and rarely changed latter on. Thus, it makes sense to analyze each destination variable individually.

As we have stated, a loop structure in the CDFG is treated as an individual super-node. Our loop analysis tar-

gets each such communciation-related super-node which contains information about that loop including the upper bound, lower bound, loop index, step, and loop condition.

For each loop structure, we scan the loop body to see if it contains any definitions of communication destination variables. These variables fall into two classes: those that are dependent on loop index and those that are not. We handle each of these situations differently. If the variables are independent of loop index, our goal is to get a final symbolic expression after all loop iterations. In such case, we can leverage static symbolic loop solving functionality in Mathematica. Mathematica is a fully integrated environment for technical and scientific computing, especially strong in symbolic expression manipulation and calculation.

Mathematica provides an application programming interface (API) called Mathlink which allows users to communicate with the Mathematica kernel in other programming languages.

Our compiler feeds Mathematica via MathLink all the loop information and the non-iterative symbolic expression generated for destination variables. Mathematica's do_loop command will automatically sort the loop information, iterate over the loop iterations and return a symbolic expression for input variables after loop calculation. For example, consider the following code segment:

$$for(\text{int } i = 1; i <= 3; i++)$$

$$y = y^2 + i$$

Our compiler feeds the following expression to Mathematica to represent the structure of the loop:

$$y = y0; Do[y = y^2 + i]\{i, 1, 3, 1\}; y$$

where $y0$ represents the expression from the block preceding the loop. The result returned from Mathematica is:

$$3 + (2 + (1 + y0)^2)^2)^2$$

If the destination variables depend on the loop index, e.g. the destination is defined as an array $dest[i]$, each element of the array may correspond to a different communication destination In this case, it is necessary to generate a symbolic expression for each loop iteration with the help of Mathematica. First, we generate symbolic expressions of the destination variables in the CDFG with the loop index unchanged. The initial value of the loop index and its symbolic expression as it increments between iterations is easy to obtain since the loop information is already recorded. We feed these expressions to Mathematica through MathLink for exach itereation in loop order (e.g. N times, where $N = \frac{loop\_upbound - loop\_lowbound}{loop\_step}$). As a result, Mathematica will return a list of symbolic expressions for the destination variables for each loop iteration.

5. Results

The effectiveness of the symbolic expression technique can be demonstrated through performance analysis. We use a simulation-based approach to study the impact on performance. We have developed a multiprocessor system simulator [10]. In a simulated system, each processor has a circuit switching network interface (NIC) connected to configurable number of circuit switching networks, and a packet switching NIC connected to a single slower packet switching network.

We ran our parallel compiler to apply symbolic expression analysis on MPI parallel applications and for the identification of communication patterns and the generation of network configurations. The compiler also instruments the MPI applications with instructions to pre-establish circuits in a particular network configuration. We demonstrate the benefits of using symbolic expression analysis by comparing the message delay using the compiler directed scheduling to the message delay using a runtime threshold based scheduling. In the following discussion, we report the *average message delay* as the performance metric.

We modeled the circuit switches as optical micro-electro-mechanical-system (MEMS)-based switches whose connection setup overhead is 3ms [1, 15] and the bandwidth of each circuit is set to 4G bps. Note that we include one slow packet switching network with link bandwidth of 400M bps in the simulated multiprocessor system in addition to the configurable number of circuit switching networks. The packet switch helps to reduce the impact of the large circuit establishment delay. It can effectively avoid the circuit establishment penalty that would occur for small and/or dynamic messages if there was no packet-switch.

This network was selected to maintain a high performance with a reduced cost for parallel machines with optical links traversing several racks [1] and we compare the performance of this network to comparable fully-buffered crossbar switches in [10, 11]. We selected message delay rather than total execution time as our presentation metric, as the benchmarks far underload the networks, even with the network capacities described here, which trail state of the art networks, which can achieve 10-20G bps. We have measured the application runtimes and the runtimes are reduced by the savings in communication time.

We present the results of three MPI programs. First, we show a synthetic program, SYN, which has 3 phases, each phase performs 2-D mesh communications along 2 dimensions on a 3-D mesh and has a maximal communication degree 4 while the communication degree of the entire program is 6. The second application is the ASCI COMOPS benchmark [6], which consists of three phases: ping-pong, 2-D ghost cell update, and 3-D ghost cell update. Here the topology of the 2-D communication is not embedded in the

topology of the 3-D communication. The third application is the multigrid solver application (MG) from the NAS parallel benchmark suite. The communication degree for the problem size we study is 13.

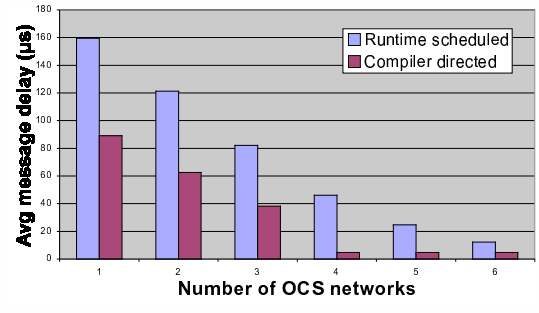


Figure 9. Average message delay of SYN.

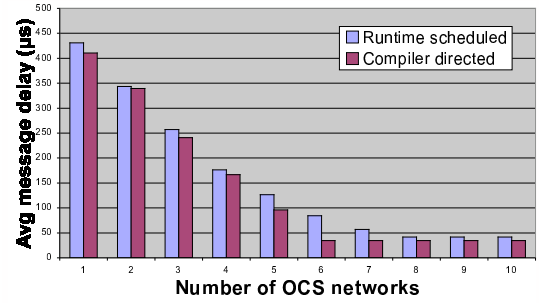


Figure 10. Average message delay of COMOPS.

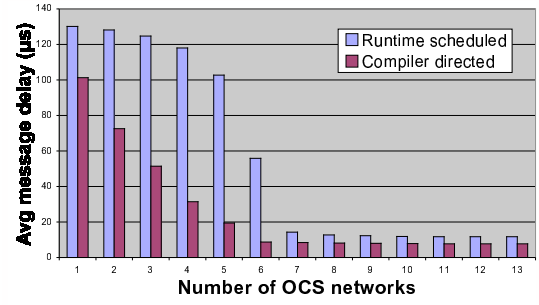


Figure 11. Average message delay of MG.

The results, shown in Figure 9, Figure 10, and Figure 11 repectively, demonstrate that with the assistance from symbolic expression analysis (labeled as *Compiler directed* in the charts), we can significantly reduce the message delay in circuit switching enabled multiprocessor systems compared to a run-time scheduling approach (labeled *Runtime scheduled* in the charts). Actually, the average speedup of the compiler directed approach for SYN, COMOPS, and MG are 3.9, 1.3, and 2.5 respectively. The maximal speedup observed appears at SYN with 4 circuit switching switches, whose value reaches 9.7.

6. Conclusion

Circuit switching is a promising alternative to packet/wormhole switching in multiprocessor systems because it can effectively reduce the cost of interconnection networks. However, the effectiveness of using circuit switching heavily relies on the the communication in a parallel application exhibiting good locality and that the locality can be exploited. The existence of locality has been well recognized. Hence we use symbolic expression analysis in a parallel compiler to identify and compose communication patterns for message passing style parallel applications.

In this paper, we present our symbolic expression analysis techniques in detail. Our approach relies on certain traditional compiler techniques, (*e.g.* control and data flow analysis, inter-procedural analysis, etc.. The handling of parallelism makes our symbolic expression analysis more sophisticated than traditional techniques. We take advantage of Mathematica as a expression manipulation back-end for the processing of expressions within loop structures.

The effectiveness of symbolic expression analysis is demonstrated by simulation based performance analysis. Up to 9.7 speedup of communication efficiency is observed.

References

[1] 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 SC*, 2005.

[2] F. Cappello and C. Germain. Toward high communication performance through compiled communications on a circuit switched interconnection network. In *Proc. of the Int. Symp. on High Performance Computer Architecture (HPCA)*, pages 44–53, 1995.

[3] V. Delaluz, M. Kandemir, N. Vijakrishnan, A. Sivasubramaniam, and M. J. Irwin. Dram energy management using software and hardware directed power mode control. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 159–169, 2001.

[4] 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 the Int. Parallel and Distributed Procssing Symp. (IPDPS)*, 2005.

[5] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on*

Programming Languages and Systems, 18(4):477–518, July 1996.

[6] LLNL. The asci comops benchmark code. `http://www.llnl.gov/asci_benchmarks/asci/limited/comops/asci_comops.html`.

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

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

[9] J. Shalf, S. Kamil, L. Oliker, and D. Skinner. Analyzing ultra-scale application communication requirements for a reconfigurable hybird interconnect. In *Proc. of SC*, 2005.

[10] S. Shao, A. K. Jones, and R. Melhem. Compiler techniques for efficient communication in multiprocessor system. Submitted for journal review since October 2007.

[11] S. Shao, A. K. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *Proc. of the Int. Parallel and Distributed Procssing Symp. (IPDPS)*, 2006.

[12] D. Shires, L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of mpi programs. In *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications(PDPTA)*, June 1999.

[13] P. Tu and D. Padua. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In *Proc. of SC*, pages 414–423, 1995.

[14] 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.

[15] T. Yamamoto, J. Yamaguch, R. Sawada, and Y. Uenishi. Development of a large-scale 3d mems optical switch module. *NTT Technical Review*, 1(7):37–42, October 2003.

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