

Program Flow Graph Construction for Static Analysis of MPI Programs

Dale Shires (presenting)
HPC Division
U.S. Army Research Lab
APG, MD 21005
dshires@arl.mil
voice: (410) 278-5006
fax: (410) 278-2694

Lori Pollock
CIS Department
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu
(302) 831-1953
(302) 831-8458

Sara Sprenkle
Computer Science
Gettysburg College
Gettysburg, PA 17325
s318286@cs.gettysburg.edu
(717) 337-6000

Abstract *The Message Passing Interface (MPI) has been widely used to develop efficient and portable parallel programs for distributed memory multiprocessors and workstation/PC clusters. In this paper, we present an algorithm for building a program flow graph representation of an MPI program. As an extension of the control flow graph representation of sequential codes, this representation provides a basis for important program analyses useful in software testing, debugging and code optimization.*

Keywords: MPI parallel programs, flow graph, optimization, program analysis, software tools

1 Introduction

A significant number of large applications have been written in MPI[1], and it has been demonstrated that message passing programs written in MPI can be both efficient and portable to various parallel environments. However, it is often described as being analogous to assembly language programming due to the low level details required of the programmer. The step from sequential code to a correct, and not necessarily yet efficient, message passing parallel program is a challenging and time-consuming one. Few sophisticated program analysis, testing or debugging tools exist to aid the programmer in this daunting task as the design of these tools is complicated by the presence of concurrency, nondeterministic execution, data distribution, and communication.

Optimization of message passing programs has focused on aggregating communication, moving communication statements to hide communication

latency by overlapping communication and computation, and reducing communication latency and unnecessary synchronization. However, techniques such as data flow analysis, classic optimization, data flow testing, and program slicing have not been addressed in the context of MPI programs.

Data flow analysis techniques for shared memory programs[2], and data flow and dependence analysis for concurrent Ada programs[3] and distributed (non-SPMD) applications[4, 5] have been developed. Dynamic slicing methods have been developed for distributed programs with Ada-type rendezvous communication[6], synchronous message passing distributed programs[7], and shared memory parallel programs[8]. Static slicing methods for concurrent programs[9, 4, 10] have focused on shared memory parallel programs with parallel sections and object-oriented features.

MPI programs present a different model of concurrent programming than these studied models. MPI programs are written in the SPMD style, in which each process executes the same program with unique data. Special conditional statements based upon the unique process identifiers allow for selective processor execution of various code segments. Although it is now possible in MPI-2 to create an MIMD application by the addition of a dynamic task creation feature, it is preferable to create a static SPMD MPI program, primarily for performance reasons. All processes are started as the program begins execution. Each process has its own local memory address space; there are no shared global variables among processes. All communication is performed through library calls to MPI message passing routines.

This paper reports on our efforts to develop a program representation for MPI programs that will

enable static program analysis for software testing, debugging, and compiler optimization. Developing this representation for MPI programs introduces several challenges. First, the SPMD nature of the codes implies that the program representation for each process is not necessarily distinct but rather processes execute the same program with segments to be executed by a subset of the processes designated by conditional statements. All processes execute the code that resides outside of these special conditionals. This behavior needs to be modeled correctly in the program representation and taken into account during static program analysis.

Second, programmers often exploit the rich set of collective communication routines in the MPI library in addition to point-to-point communication. One way of handling programs with collective communication is to translate them into a sequence of point-to-point communication calls for program analysis. However, one intended use of the representation is to display information about the program flow to the programmer through a graphical user interface, and thus the program representation should be presented to the programmer in terms of the original MPI program. Additionally, collective communication such as `scatter` and `gather` operations involve different sections of an array being partitioned or gathered to the various processes, respectively. It would be most useful to have the data flow information reflect this sectioning of the array.

In the remainder of this paper, we present the results of a characterization study of a set of MPI programs which has guided the design of our techniques and a description and algorithm for construction of an MPI program flow graph.

2 Characterization Study

While our goal is to build a suite of “real-world” codes, finding stable MPI production codes is a common problem being addressed by consortiums and vendors. For this paper, we statically analyzed the usage of MPI in the Numerical Aerospace Simulation (NAS) Parallel Benchmark suite and two other major codes listed in table 1. The NAS codes include various kernel and application MPI benchmarks. Znsflow from the Army Research Laboratory is a computational fluid dynamics code. It solves the unsteady Reynolds averaged Navier-Stokes equations and can be targeted to various projects of interest. The NASA OVERFLOW code comes from the NASA Ames Research Center. It computes numerical solutions of the compressible

Navier-Stokes equations using finite differences in space and implicit time-stepping.

We made several concluding observations from a cursory examination of the programs listed in table 1. In all the codes, the lines of code related to MPI communications was less than, and in most cases far less than, 1.5% of the total number of lines of code. Special conditionals such as `if (myrank == 0)`, indicative of manager-worker parallelism, were present but not common. Indeed, many of these programs relied upon initialization routines to compute arrays or scalars, such as `north`, `south`, etc. to hold information about neighboring processes and domains. Some codes used nested conditionals to further refine execution paths. Source and destination fields in the communication calls contained expressions using precomputed arrays, scalars, constants and in some cases the process identifier. MPI wildcards, such as `MPI_ANY_TAG` or `MPI_ANY_SOURCE`, were not common. While these programming styles make MPI-CFG classification more difficult, it does not necessarily preclude it in most cases. Many of the scalars and arrays are defined with some reference to a unique process identifier. A static backward slice within the local CFG can be used to reformulate these expressions in terms of the process identifier.

3 MPI Program Flow Graph

Since each process in an MPI program has local space allocated for each of the declared variables in the program, and communication occurs only through matching MPI communication calls, data flow local to a given process between communication points in that process is not affected (as a side effect) by the data flow within other processes. The data flow within a given process is only affected by other processes at communication points. In point-to-point communication, a message sent by a particular send operation will be received by another process only through a receive operation executed by the other process. Specifically, a message can be received by a particular receive operation only if it is addressed to the receiving process by the sender, the send and receive have matching communicator fields, the sender field of the receive is either `MPI_ANY_SOURCE` or matches the sender’s process id, and the tag fields of the send and receive match or the tag field of the receive is `MPI_ANY_TAG`. In collective communication, all processes in the designated communicator are involved in the communication. Although multiple messages sent from one process

Code	Source Lines	MPI Sends	MPI Recvs	MPI Collective	% MPI Calls in Special Branches
NPB Block Tridiagonal	5432	12	12	7	0
NPB Multigrid	2438	12	1	13	46
NPB Scalar Pentadiagonal	4706	12	12	6	40
NPB 3-D FFT	1946	0	0	6	0
NPB LU Decomposition	5182	12	12	15	62
ARL Znsflow	16744	69	69	40	75
NASA OVERFLOW	22017	189	185	40	85

Table 1: Characteristics of MPI Usage in Parallel Programs.

to another process are guaranteed to arrive in the order they were sent, there are no assumptions made on the order of arrival of messages from two different sources to the same destination. When a message receive specifies `MPI_ANY_SOURCE` as the expected sender, the originator of the message will be indeterminate at static analysis time; otherwise, the expected sender is specified, and communication is deterministic. We conservatively represent this indeterminacy in our program representation.

A Control Flow Graph (CFG) representation for a sequential program P is a directed graph $G = (N, E, S, e)$ where each node $n \in N$ represents a basic block of instructions, each edge $n \rightarrow m \in E$ represents a potential flow of control from node n to node m , and there is a unique start node s and a unique exit node e . A path in G is a sequence of nodes (n_1, n_2, \dots, n_k) where $n_i \rightarrow n_{i+1}$ for all $1 \leq i \leq k$. We assume that every path in the CFG is a viable execution order of the statements in the program P .

An *MPI Control Flow Graph (MPI-CFG)* extends the CFG with *communication edges*, and isolates each communication statement into its own separate basic block, represented by a single node in the graph. We call these nodes *communication nodes*.

While point-to-point communication can be easily represented by a single communication edge, collective communications have distinct semantics that result in different data flow across processes. For example, a broadcast will result in every process receiving the same value and storing it into the same local variable, whereas a scatter will result in each process receiving a subset of a set of values sent from the root process, in order to distribute or partition the data stored in a single array among the processes. The representations of these communication statements were developed with the goal that each communication statement should have a unique representation that reflects its semantics.

Lastly, the control flow edges of the MPI-CFG are annotated with a value that reflects static information about the number, and possibly the process ids (if available) of the processes that could execute along that edge. The value will be either: $\langle c \rangle$ indicating the known process id c of the only process that will execute that edge,

$\langle single \rangle$ indicating that statically we can prove that only a single process will execute this process, but we cannot determine the process id,

$\langle unknown \rangle$ indicating that it could be one or more processes executing this edge, or

$\langle multiple \rangle$ indicating that definitely more than one process will execute this code if there are more than 1 executing processes. We also keep a predicate here if this is available and possible to identify. This information allows the communication edge addition step and other static program analysis to utilize the information about process ids.

Due to space limitations and the size of most interesting MPI programs, we define a *condensed* MPI-CFG as an MPI-CFG in which the nodes representing computation (non-communication) blocks between two communication nodes are collapsed into a single representative computation node. This structure is meant for presentation purposes only. Program analysis is to be performed over the MPI-CFG, not the condensed MPI-CFG. Figure 1 illustrates the condensed MPI-CFG for an SPMD-style MPI program called `summax`. We indicate control flow edges by solid lines, while communication edges are shown as dotted lines. Communication edges are labeled with the variables that are involved in the interprocess communication. The program `summax` computes the sum of an array of integers and finds the maximum value in the integer array. The conditional $\langle myrank == 0 \rangle$ is an example of a special conditional statement indicating that the left branch is to be executed only by process 0, while the rest of the processes should execute the right branch.

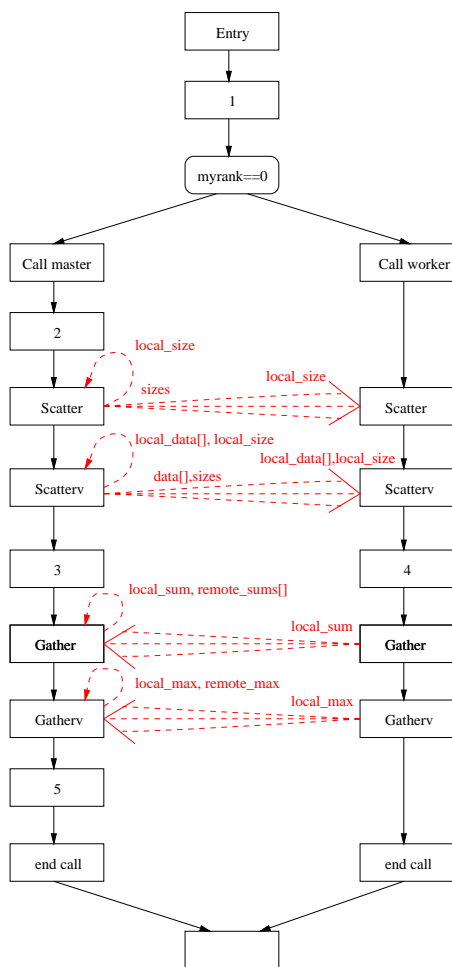


Figure 1: Condensed MPI-CFG of `summax`

4 Construction Algorithm

The MPI-CFG construction algorithm is summarized in figure 2. The first step is to create the underlying CFG by using a slight modification to the usual algorithm for CFG construction which isolates communication statements as separate nodes. Each process’s CFG is represented by some subgraph of this graph, where different processes typically have subgraphs that overlap one another. An initial pass of edge annotation based on the relational operator of the special conditionals will indicate program segments which are executed by one process versus possibly multiple processes. Many parallel programmers program in the manager-worker style of programming, where the special conditionals `if (myrank == 0)` will often be an equality test against a constant. This information is used in the constant propagation phase. Traditional con-

stant propagation can be applied to CFG representation of an SPMD program; however, it will be overly conservative in the handling of constants at join points from branches taken by different processes. More sophisticated constant propagation that recognizes constants with respect to particular processes would result in more precise information per process. Propagating constants helps to eliminate symbolic information in the parameters of communication statements as well as the information known about the expressions in special conditionals.

Algorithm: MPI-CFG Construction.

Input: MPI program `P`.

Output: MPI-CFG representation of `P`.

```

begin
Treating MPI calls as regular function calls,
Construct the CFG representation P-CFG of P;
Using the parameter of MPI_Comm_rank,
Identify special conditionals that
indicate separate process control flow;
Perform initial annotation of edges based on the
expression operator in special conditionals;
Using annotations, perform modified constant
propagation over P-CFG;
Perform final annotation of edges using new
information at special conditionals;
At each MPI communication statement,
Use constants, CFG slices and MPI matching rules
to identify potential matching communication;
Conservatively add communication edges to P-CFG;
end.

```

Figure 2: MPI-CFG Construction Algorithm

The last step is to conservatively add communication edges. Because the same code segment may represent multiple processes, it is possible for a communication that occurs at run-time to have no associated communication edges, only a communication node. Communication edges are added according to the kind of communication, variables in particular fields of the communication call, any statically determined information about constants and the annotations on control flow edges, and the matching rules for communication statements. Sometimes, the communication is ambiguous, due to unknown values for variables or wildcards in the source or tag fields. In these situations, we add an edge for any potential matching communication. In the MPI programs that we examined, there are very few communications that would cause additional edges to be added due to lack of information at analysis time.

The most challenging aspect of finding the po-

tentially matching communication statements is identifying the source and destination processes. The source and destination fields of communication statements can be categorized as being (1) a constant, (2) an expression involving the process identifier, or (3) an expression not containing the process identifier. We first perform traditional backward CFG slicing (without communication edges) to reformulate expressions that are derived from the process identifier, but do not explicitly contain the process identifier. Then, in cases (1) and (3), we use the annotations on MPI-CFG edges to refine the set of potentially matching communications. In case (2), we use variable substitution in the expression functions of these fields to determine whether the source and destination expressions of the receive and send operations, respectively, can be equal.

5 Current Directions

We are currently implementing our program flow graph construction within the SUIF compiler infrastructure[11]. We are also investigating more precise constant propagation analysis and extension of the program dependence graph representation for SPMD programs.

References

- [1] Message Passing Interface Forum. Mpi: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3-4), 1994.
- [2] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168, California, USA, 1993.
- [3] Douglas Long and Lori A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 21–35, October 1991.
- [4] Jingde Cheng. Dependence analysis of parallel and distributed programs and its applications. In *IEEE-CS International Conference on Advances in Parallel and Distributed Computing*, 1997.
- [5] Shing Chi Cheung and Jeff Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, August 1994.
- [6] B. Korel and R. Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science*, 2, 1992.
- [7] E. Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Distributed slicing and partial re-execution for distributed programs. *Languages and Compilers for Parallel Computing*, pages 497–511, 1992.
- [8] J-D Choi, B. Miller, and R. Netzer. Technique for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.
- [9] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Static slicing of concurrent object-oriented programs. In *Proceedings on IEEE-CS Twentieth Annual International Computer Software and Applications Conference*, pages 312–320, 1996.
- [10] J. Krinke. Static slicing of threaded programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, Montreal, Canada, 1998.
- [11] Stanford SUIF Compiler Group. *The SUIF Parallelizing Compiler Guide*. Stanford University, 1994.