Barrier Matching for Programs with Textually Unaligned Barriers

Yuan Zhang
University of Delaware
zhangy@capsl.udel.edu

Evelyn Duesterwald
IBM T.J. Watson Research Center
duester@us.ibm.com

Abstract

Barriers, a common synchronization primitive in SPMD-style programs, are used to partition a program into a sequence of parallel phases. Popular parallel programming models, such as MPI and OpenMP, allow barriers to be textually unaligned. Textually unaligned barriers make it difficult for the programmer to understand the synchronization phases in the program, and they can easily lead to synchronization errors. In this paper, we present an interprocedural analysis for matching barriers in a program in order to detect synchronization errors, or, if no such errors exist, to determine the synchronization phases of the program. Our analysis uses a combination of path expressions and interprocedural program slicing to match synchronizing barrier statements. If the barrier matching succeeds, the analysis determines the sets of barrier statements that synchronize together. A matching failure indicates the presence of a synchronization error and the analysis constructs a counter example to illustrate the error. We have implemented the analysis in an MPI checker tool for programs written in C and successfully analyzed the synchronization structure of several MPI benchmarks.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Verification

Keywords Barrier synchronization, static analysis, program slicing, path expressions, MPI

1. Introduction

SPMD (Single Program Multiple Data) is a popular parallel programming paradigm. Typically, SPMD-style programs have a barrier synchronization primitive that can be used to partition the program into a sequence of parallel phases. When a thread reaches a barrier statement it cannot proceed until all other threads have arrived at the barrier statement. Barriers are textually aligned if all threads must reach the same textual barrier statement before they can proceed. A barrier synchronization error occurs if a thread bypasses a barrier, leaving the remaining threads stalled.

MPI [22] and OpenMP[14], two widely used parallel programming models, place few or, in the case of MPI, no constraints on the placement of barrier statements in the program. Barrier statements may be textually unaligned making it more difficult for programmers to understand the synchronization structure of the program and, thus, easier to write programs with synchronization errors. Textually unaligned barriers also hinder concurrency analysis [6, 15, 10, 12, 13] because understanding which barrier statements form a common synchronization point is a prerequisite to analyzing the ordering constraints imposed by the program. Some concurrency analyses therefore require barriers to be named or textually aligned [12, 10, 13].

In this paper we present an interprocedural barrier matching technique for SPMD-style programs with textually unaligned barriers. Barriers matching detects synchronization errors by matching synchronizing barrier statements. If the matching succeeds, the program is free of barrier synchronization errors and a barrier matching function is computed that maps each barrier statement s to the set of barrier statements that synchronize with s for at least one instance of s. A matching failure indicates a synchronization error for which our analysis provides a counter example that illustrates the error.

Barrier matching functions provide more information than a verification of the program's synchronization structure alone because they expose barriers that are textually unaligned. Programmers can use this information to improve readability of their code by eliminating textually unaligned barriers. The information can also be used to validate the programmer's mental view of the synchronization structure of the program. Used in this way, barrier matching information can aid in the detection of more subtle algorithmic problems with the use of synchronization.

Figure 1 shows a sample SPMD-style program fragment with two barriers. The function *get_rank()* returns the unique thread identifier of the calling thread. *Get_rank()* is similar to the "MPI_Comm_rank()" library function in MPI and corresponds to the "omp_get_thread_num()" library function in OpenMP. The barrier construct shown in the figure is known as "#pragma omp barrier" in OpenMP and as the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'07 March 14-17, 2007, San Jose, California.
Copyright © 2007 ACM 978-1-59593-602-8/07/0003...$5.00.

```
main(){                          S2: if(x > 0){
    int x=0;                             ...
    int rank = get_rank();           } else {
    ...                                  ...
S1: if(rank == 0){                       barrier; //b2
        ...                          }
    } else {                     }
        ...
        barrier; //b1
    }
```

Figure 1. Example program

"MPI_Barrier(<communicator>)" function call in MPI. A synchronization error occurs in the conditional S1 because all threads except that with rank 0 reach the barrier b1 and get stalled. However, the conditional S2 is free of errors because all threads agree on the value of x. Our barrier matching analysis will report a matching error at barrier b1 and calculate the matching function for barrier b2 as $\mathcal{M}$(b2)={b2}.

To detect barrier synchronization errors and to compute the barrier matching function, our analysis uses a combination of path expressions [24] and interprocedural slicing [9]. The analysis proceeds in three steps:

Step 1: Multi-Valued Expressions: In SPMD-style programs all threads execute the same program but they may execute different program paths. The ability to determine which program paths may be executed concurrently is a key component in our analysis approach. Our analysis determines concurrent paths by computing the *multi-valued* expressions in the program. An expression is multi-valued if it evaluates differently in different threads. If used as control predicates, multi-valued expressions split threads into different concurrent program paths. In the example shown in Figure 1, $rank$ is a multi-valued variable while x is not. We present a new interprocedural solution to the multi-valued expression problem that is based on interprocedural program slicing [9].

Step 2: Barrier Expressions: The next step consists of constructing a barrier expression at each program point. A barrier expression is a special form of a path expression [24] that, for a given program point, describes the sequences of barriers that may execute until a thread reaches that point. Barrier expressions provide a compact representation of the synchronization structure of the program.

Step 3: Barrier Matching: The final step uses the results of the previous steps to match barrier expressions against each other. We show that, for the program to be correct, barrier expressions have to match at points where concurrent threads meet. We describe an efficient barrier matching algorithm that, in case of a successful match, provides the corresponding barrier matching function. A matching failure indicates a synchronization error for which the analysis provides a counter example by constructing two program paths that illustrate the error.

The verification of textually unaligned barriers and the related problem of determining multi-valued expressions were first addressed by Aiken and Gay [1]. They developed a set of inference rules implemented for Split-C [11]. Their rule system cannot automatically handle procedures and assumes user annotations to describe the effect of procedures. Our barrier matching analysis detects the same class of synchronization errors. However, we present an interprocedural solution. Furthermore, in addition to verifying correct synchronization, our analysis establishes the barrier matching functions to expose the synchronization phases of the program and the presence of textually unaligned barriers.

After detecting one synchronization error our analysis continues to analyze the portions of the program that are unaffected by the error. Thus, the analysis may report multiple synchronization error warnings for some areas of the program and a matching function for others. The results of our barrier matching analysis can be used to detect synchronization errors, to expose textually unaligned barrier, and to provide the input to a concurrency analysis.

We have implemented barrier matching as an MPI checking tool as part of the Eclipse Parallel Tools Platform (PTP) project (www.eclipse.org/ptp). We applied our checker to MPI/C programs and have successfully analyzed the synchronization structure of several MPI benchmarks.

In summary, the contributions of this paper are as follows:

- We present an interprocedural solution to the multi-valued expression problem to determine the concurrent program paths in an SPMD-style program.
- We introduce barrier matching as an interprocedural barrier verification analysis.
- Our analysis goes beyond verification by computing a matching function exposing textually unaligned barriers in addition to the verification result.
- We have implemented an MPI barrier checking tool for C and present an evaluation of the tool on a set of MPI benchmarks.

The rest of the paper is organized as follows. Section 2 provides an overview of the barrier matching problem. The interprocedural multi-valued expression analysis is described in Section 3. Sections 4 and 5 describe the barrier expression construction and the barrier matching algorithm, respectively. The experimental evaluation is presented in Section 6. We present related work in Section 7 and conclude in Section 8.

2. Barrier Matching Overview

Our analysis is applicable to SPMD-style programs with barriers that may be unnamed and textually unaligned. The goal of our analysis is to determine whether the barriers in the program are *well-matched*, and if they are, to compute the barrier matching function. The barriers in a program are well-matched if all concurrent threads execute the same number of barriers.

As commonly done for the analysis of sequential programs, we would like to formulate this problem as a path flow problem over a graphical representation of the program, such as the program's control flow graph (CFG).

The CFG of a program is a directed graph $G = (N, E)$ with a set of nodes N that represent the program's basic blocks and a set of control flow edges E connecting the

nodes. A program path is a connected sequence of nodes in G. Figure 2(a) shows a program fragment in C and its control flow graph is shown in Figure 2(b).

Sequential data flow problems are usually solved by computing a solution over all paths in the CFG. There is an important distinction between an all-paths solution in a sequential program and the notion of all-paths in an SPMD-style program. In SPMD-style programs, some of the program paths may be concurrent while others may not be. Program properties may hold for all concurrent paths, but not for all program paths.

The CFG in Figure 2(b) illustrates this point. There are three paths from node A to node G: P_1: A B G, P_2: A C D F G, and P_3: A C E F G. The number of barriers executed along these paths differs: two barriers for P_1 and one barrier for P_2 and P_3. However, not all three paths are concurrent. The predicate at node C is multi-valued creating two concurrent paths P_2 and P_3. The program is well-matched because along the two concurrent paths the number of barriers is the same.

To formulate a path problem in the presence of concurrent paths we need additional mechanisms.

Definition 2.1. (Execution Trajectory) *Given an SPMD-style program, an execution trajectory $\mathcal{T}$ with respect to an execution E of the program is the set of program paths from start to exit that are executed by each thread in E.*

Definition 2.2. (Concurrent Paths) *Two program paths p_1 and p_2 with $p_1 \neq p_2$ are called concurrent if there exists an execution trajectory that contains both paths.*

We can now define the barrier matching problem as a path problem as follows:

Definition 2.3. (Barrier Matching Problem) *Barriers in an SPMD-style program are well-matched if for every set of concurrent paths from program start to program exit the number of barriers is same along each path.*

Thus, a prerequisite to our analysis is the ability to determine the concurrent paths in the program. In many SPMD programming styles, concurrent paths are not explicit in the program text. They can be derived from the program points at which concurrent threads split and the points at which they meet again. To determine these concurrent split and meet points we compute the *multi-valued* expressions in the program as described in detail in Section 3.

2.1 Analysis Scope

Our analysis can handle programs with arbitrary control flow. However, for structured programs (i.e., programs without goto statements) we can often use simpler algorithms. We can transform every program into a structured program using goto elimination [7]. Whenever simpler algorithms for structured programs are available we will describe both the general algorithm for arbitrary control flow and the simpler algorithm for structured programs.

Similar to others [1, 5] we observe that programmers typically use synchronization in a highly structured way. We follow the assumption, introduced by Aiken and Gay [1], that correct parallel programs are also *structurally correct*. Informally, structural correctness means that a program property holds for a program if it holds for every structural component of the program, (i.e., every statement, expression, compound statement, etc.). The assumption of structural correctness simplifies the analysis in that we can break down the verification problem for the whole program into a series of smaller problems, one for each structural component.

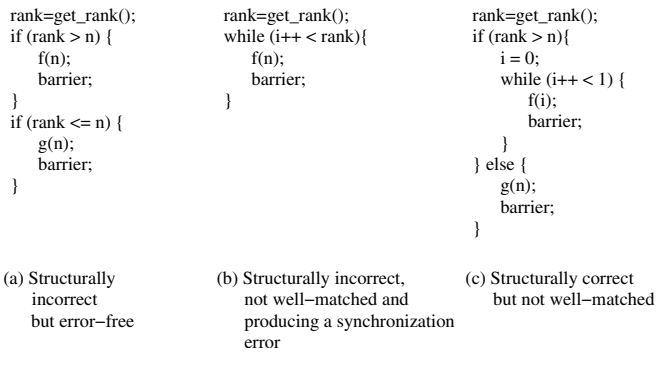


Figure 3. Synchronization examples that produce synchronization error warnings

More formally, we define structural correctness with respect to a property $\mathcal{P}$ based on the abstract syntax tree (AST) of a structured program as follows:

Definition 2.4. *(Structural Correctness) Let T be the AST of a structured program P, P is structurally correct with respect to property $\mathcal{P}$ if each subtree of T is structurally correct with respect to $\mathcal{P}$.*

The structural correctness assumption enables us to show that the barriers in a program are well-matched by inductively showing that the barriers in all subtrees of the program's AST are well-matched.

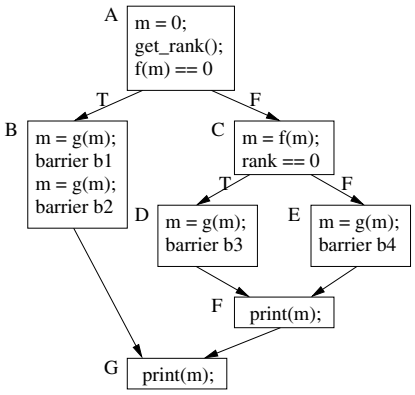
Structural correctness is sufficient but not necessary for a program to be correct. However, we have yet to find a realistic program that is correct but not structurally correct.

Our example from Figure 2 is structurally correct. Figure 3 (a) shows a code fragment that is structurally incorrect because each conditional, if viewed in isolation, is incorrect. However, the fragment is free of synchronization errors. Figure 3(b) shows an example of a structurally incorrect program that also contains a synchronization error.

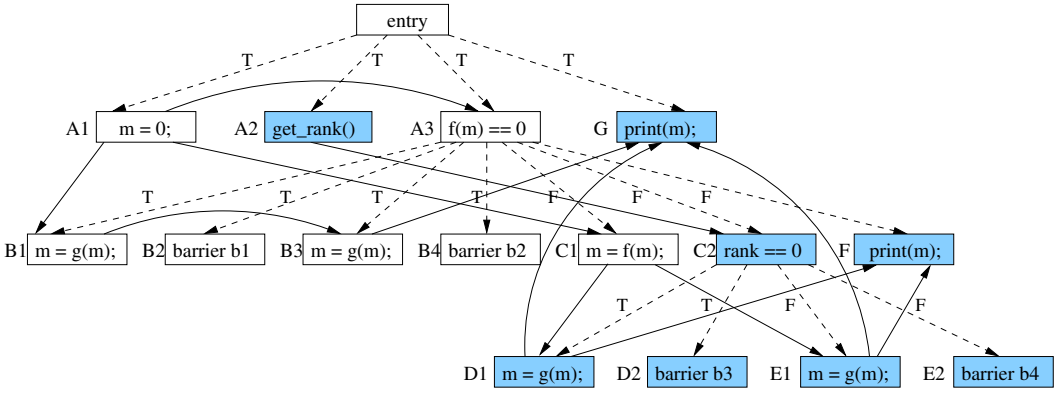
Our barrier matching analysis verifies that an SPMD-style program is structurally correct and free of barrier synchronization errors. However, our analysis may report spurious synchronization error warnings for structurally incorrect programs, such as the example in Figure 3(a). Furthermore, due to the conservative nature of static analysis, our analysis may report spurious warnings in the presence of infeasible paths. For example, the code segment in Figure 3(c) is structurally correct and free of synchronization errors. However, our analysis assumes that any number of iterations through the loop is feasible and would issue a warning. Spurious warnings may also result from an overestimate of the multi-valued expressions in the program. Such an overesti-

```
m = 0;
rank=get_rank();
if( f(m) == 0 ){
    m = g(m);
    barrier b1
    m = g(m);
    barrier b2
} else {
    m = f(m);
    if( rank == 0){
        m = g(m);
        barrier b3
    } else {
        m = g(m);
        barrier b4
    }
    print(m);
}
print(m);
```

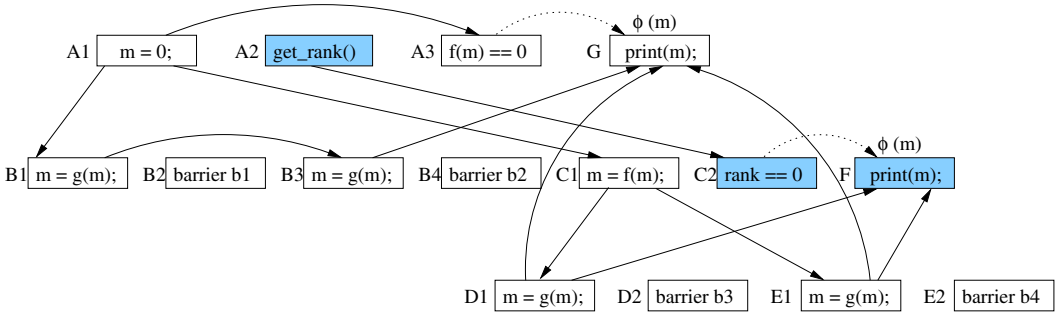
(a) Example program



(b) Control flow graph



(c) Program Dependence Graph



(d) Modified dependence graph with ϕ −nodes and ϕ −dependence edges

Figure 2. Barrier program example. For illustration purposes we assume that function f and g are library functions free of side-effects. (a) Example program (b) Control flow graph (c) Program dependence graph. Control dependence edges and data dependence edges are represented by dashed lines and solid lines, respectively. (d) Modified dependence graph with ϕ nodes and ϕ-dependence edges. Data dependence edges and ϕ dependence edges are represented by solid and dotted lines, respectively. Slicing based on variable "rank" at node A2 is shown through shaded boxes.

mate may be caused by an overly conservative handling of pointers and array variables. We will discuss the impact of pointer analysis further in Section 3.

Importantly, although possible due to the static nature of the analysis, our evaluation in Section 6 shows that spurious warnings are very rare and not a problem in practice.

We describe our analysis in the context of MPI programs. Applying the analysis to OpenMP requires some adjustments, not described here, to represent OpenMP's concurrency constructs in the control flow graph and to account for shared variables in the multi-valued expression analysis.

3. Multi-Valued Expressions

Multi-valued expressions are computed to determine the concurrent paths in the program, or more specifically, to determine the thread split and meet points. In addition to being used as input to barrier matching, information about concurrent paths is useful by itself and can provide an important tool for program understanding.

An expression is multi-valued if it evaluates differently in different threads. Conversely, an expression that has the same value in all threads is called single-valued.

Parallel programs typically contain multi-valued seed expressions, such as library calls that return a thread identifier (e.g., "MPI_Comm_rank" in MPI, and "omp_get_thread_num" in OpenMP). All multi-valued expressions are directly or indirectly derived from these initial multi-valued seeds. These dependence relationships suggest that we may be able to solve the multi-valued expression problem as a program slicing problem.

Program slicing was first introduced by Weiser [29]. Venkatesh [28] later defined a *forward slice* as follows: given a program point p and a variable v, the forward slice is the set of statements that are affected by the value of variable v at point p. Ottenstein and Ottenstein [17] showed that we can recast the slicing problem as a graph reachability problem using the program dependence graph. The program dependence graph contains nodes for all statements in the program and two types of edges: data dependence edges and control dependence edges. The forward slice for a node n that defines a variable v is the set of nodes reachable from n in the program dependence graph. Horwitz et al. [9] extended the work by Ottenstein and Ottenstein on program dependence graphs by developing an interprocedural solution to program slicing. Their interprocedural extension of the program dependence graph is called a system dependence graph.

Figure 2(c) shows the program dependence graph for our example. The slice for variable *rank* at node A2, shown as the set of shaded nodes, contains all nodes reachable along control or data dependence edges.

Unfortunately, forward slicing overestimates the multi-valued expressions. There is a subtle difference between the dependence information used in traditional program slicing and the dependence information needed for computing multi-valued expressions. Figure 2 (c) illustrates this difference. Variable m is single-valued at node C and first becomes multi-valued at node F when the threads that split at node C meet again. Variable m is single-valued at nodes D and E because it has the same value for all threads that execute these nodes. However, the forward slice on $rank$ includes nodes D and E because they are control dependent on C2.

Procedure *MultiValuedSlicing*
1. Build a CFG of the program
2. Insert ϕ nodes and ϕ gates
3. Build the system dependence graph as described in [9] using ϕ edges in place of control dependence edges
4. Mark every multi-valued seed expression in the graph
5. Compute the interprocedural slice as the set of nodes reachable from seed expressions using the algorithm in [9]

Figure 4. Algorithm *MultiValuedSlicing*

Thus, computing multi-valued expressions requires some adaptations of existing slicing algorithms. Instead of control dependence edges we need edges from multi-valued predicates to the points where the values of variables that are control-dependent on the predicates merge. In the example in Figure 2(c), we need an edge from C2 to F.

This notion of value merge dependence can be found in Static Single Assignment Form (SSA) [4]. SSA Form uses ϕ-nodes to represent the new value of a variable at join points where multiple definitions of the variable are merged. In Figure 2(d) ϕ-nodes would be placed at nodes F and G. Gated SSA form is a refinement of SSA in which each ϕ-node is connected to the controlling predicate [27]. We refer to such a predicate as the ϕ-gate and we call the edges connecting a ϕ-gate with the corresponding ϕ-nodes "ϕ-edges". ϕ-edges are exactly the dependence edges we need for our multi-valued slicing problem. Figure 2(d) shows a modification of the program dependence graph from (c) with ϕ-nodes and ϕ-edges.

For structured programs we can determine ϕ-gates and ϕ-nodes directly from the nesting structure. In the general case, the algorithm described in [27] can be used.

Based on the notions of ϕ-nodes and ϕ-gates we can inductively define multi-valued expressions as follows:

Definition 3.1. (Multi-Valued Expressions) *An expression e is multi-valued if one of the following holds:*
(i) e is a multi-valued seed (e.g., a thread library call that returns a different value in each thread),
(ii) e is data-dependent on a multi-valued expression,
(iii) e is a ϕ-node with a multi-valued ϕ-gate.

We can now define multi-valued expression analysis as a simple adaptation of program slicing. After replacing control dependence edges with ϕ-edges we can use the existing algorithm for interprocedural slicing developed by Horwitz et al. [9] to compute the multi-valued expressions in the program. Figure 4 shows an overview of this approach.

After computing the multi-valued expression slice we determine the concurrent paths in the program by marking the concurrent split and meet points in the graph. The concurrent split points are determined as the ϕ-gates that are contained in the slice. Similarly, the concurrent meet points result as the ϕ-nodes contained in the slice.

3.1 Library Calls

Thread library calls may produce both multi-valued expressions (such as MPI_Comm_rank in MPI), or single-valued expressions (such as a broadcast). We assume that thread library interfaces are annotated as either single- or multi-valued. If no such annotations are available, we can conservatively treat all thread library calls as producing multi-valued seed expressions. Other library function calls return single-valued expressions.

3.2 Handling Pointers and Arrays

Pointers and arrays impact the multi-valued expression computation by complicating the determination of accurate data dependences. The program dependence graph has to safely represent all possible data dependences in order for the multi-valued expression slicing to be a safe approximation. A simple conservative handling of arrays models each array as a single object. Pointers can be modeled safely by treating every dereference of a pointer and every variable whose address is taken as a multi-valued seed. A conservative slicing computation overestimates the multi-valued expressions in the program, which, in turn, can lead to spurious synchronization error warnings. The accuracy of the slice can be improved by applying a pointer analysis [2] prior to the construction of the program dependence graph.

As our evaluation in Section 6 shows, for scientific SPMD applications, simple pointer and array approximations often suffice to avoid spurious error warnings.

4. Barrier Expressions

The next step of our analysis consists of constructing barrier expressions. Barrier expressions are a special form of Tarjan's path expressions [24]. A path expression at a node n in a CFG represents all paths from the beginning of the program to node n. Path expressions are regular expressions built using node labels as terminal symbols and the operators: $\cdot$ (concatenation), $|$ (alternation), and $\star$ (quantification). For example, in Figure 2 the path expression at node G is A $\cdot$ (B $|$ (C $\cdot$ (D $|$ E) $\cdot$ F) $\cdot$ G.

Path expressions have been used to build data flow analyzers by interpreting node labels as data flow functions, path concatenation as flow function composition, alternation as taking the meet of functions, and quantification as finding the fixed point of a function [25].

A barrier expression is a path expression that uses barrier statement labels instead of node labels as terminal symbols. A barrier expression at a program point n represents the sequences of barriers that may execute along any path from the beginning of the program to node n. If a barrier expression B represents a barrier sequence $b_1, b_2, ..., b_n$ we say that B "derives" n barriers.

We can obtain a barrier expression from a path expression by replacing the node labels in the path expression with barrier labels as follows. If a node n contains a sequence of barriers $b_1, ..., b_n$ we replace n with the concatenation $b_1 \cdot ... \cdot b_n$. If n contains no barriers we replace n with the empty symbol $\emptyset$. We assume that redundant empty symbols are eliminated from the expression whenever possible to improve readability of the expression. If we remember the sequence of $\emptyset$ eliminations, we can always translate a barrier expression back to its original path expression, if needed.

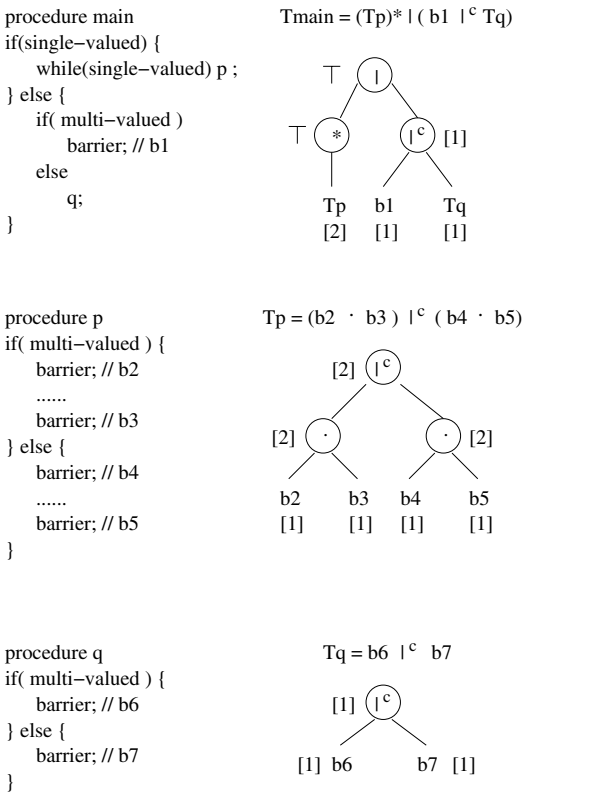


Figure 5. Barrier trees. The fixed length $cnt(t)$ for each subtree t is shown within brackets.

Using Tarjan's fast path algorithm [23] building barrier expressions takes $\mathcal{O}(E \log N)$ time. For structured programs, constructing barrier expressions can be done in linear time over the program's AST by following the nesting structure of the program.

To simplify the discussion we assume barrier expressions are represented by their expression trees as shown in Figure 5. We refer to these tree representations as *barrier trees*.

To compute barrier trees for the whole program we compute a separate barrier tree T_p for each procedure p. The tree T_p represents the barrier expression determined for the exit node of procedure p. A call inside p to another procedure q is represented by the label T_q for the barrier tree for procedure q. The whole program is thus represented by a set of barrier trees and the tree set contains $(B + C)$ leaf nodes, where B is the number of barrier statements and C the number of procedure call sites in the program. It follows that the size of all barrier trees for a programs is $O(B + C)$.

Figure 5 shows a program example consisting of three procedures along with their barrier trees.

We complete the barrier tree construction by incorporating the information about concurrent meets from the multi-valued expression slice. The alternation and quantification symbols in a barrier tree corresponds to the meet nodes for

conditionals and loops in the CFG of the program. We annotate barrier trees by marking each alternation and quantification symbol as concurrent if the corresponding meet node in the CFG is marked as concurrent. In Figure 5, a marked concurrent alternation is shown as $|^c$. The resulting barrier trees contain regular as well as concurrent alternation and quantification symbols.

We can now restate the barrier matching problem from Definition 2.3 in terms of barrier trees as follows: A barrier tree t is well-matched if all concurrent barrier sequences that can be derived from t have the same number of barriers.

5. Barrier Matching

We introduce the following terminology for barrier trees:

Definition 5.1. (Fixed-length Barrier Tree): *A barrier tree t is called a fixed-length tree if all barrier sequences derivable from t have the same number of barriers.*

A tree that is not fixed-length is called variable-length. We can easily show by induction on the size of t that if t is fixed-length then all subtrees of t are fixed-length.

Fixed-length provides a sufficient but not necessary condition for a tree to be well-matched. In a fixed-length tree *all* derivable sequences have the same number of barriers, in a well-matched tree only the concurrent sequences are of the same length. The relationship between fixed-length and well-matched trees is captured in the following claim:

Claim 5.1. (Matching Conditions) *A barrier tree t is well-matched if and only if the following two conditions are satisfied:*
(1) t contains no concurrent quantification subtrees
(2) all concurrent alternations subtrees are fixed-length

Proof. Clearly, if any of the conditions (1) or (2) are violated, t cannot be well-matched. Conversely, assume t is not well-matched, that is, t derives at least two concurrent barrier sequences of different lengths. It follows that t must contain concurrent subtrees that are not fixed-length. In other words, there exists a variable-length subtree that is either a concurrent quantification subtree violating condition (1), or a concurrent alternation tree violating condition (2). □

We next describe an algorithm that determines whether a tree is fixed-length using a linear-time bottom-up traversal of the barrier trees. The traversal computes a barrier count $cnt(t)$ for each subtree t. If t is fixed-length $cnt(t)$ is the length of barrier sequences derivable from t. We use the symbol $\top$ to denote a variable number of barriers. For all integer numbers n: $\top + n = \top$ and $\top + \top = \top$.

For each procedure p with a barrier tree T_p, $cnt(T_p)$ is initialized as follows:

$$cnt(T_p) = \begin{cases} 0 & \text{if } T_p \text{ is empty} \\ \top & \text{otherwise} \end{cases}$$

Note that T_p is empty only if procedure p contains no barrier statements and no procedure calls.

The calculation of $cnt(t)$ proceeds by applying the rules shown in Figure 6 during a bottom-up traversal of t. It is easy to show by induction on the size of t that the calculation rules

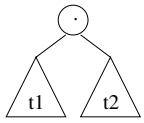
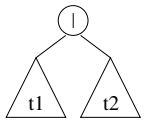
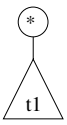
Barrier Tree t	Fixed−length Calculation Rule
b	cnt (t) = 1
∅	cnt (t) = 0
Tp	cnt (t) = cnt (Tp)
	cnt (t) = cnt (t1) + cnt (t2)
	cnt (t) = n if cnt (t1) = cnt (t2) = n ⊤ otherwise. If t is a concurrent alternation tree, report warning
	cnt (t) = cnt (t1) + ⊤ If t is a concurrent tree, report warning

Figure 6. Fixed-length calculation rules

are correct, that is, $cnt(t) = n$ with $n \neq \top$ if and only if t is fixed-length and derives n barriers.

To analyze the barrier expressions interprocedurally we traverse the call graph bottom-up and calculate $cnt(T_p)$ for each procedure p. Recursion is handled safely through the initialization of $cnt(T_p)$. For a non-empty tree T_p, the initial value $\top$ propagates throughout any recursive cycle, correctly indicating that the number of barriers that result from recursion is variable.

We use the fixed-length information to verify the matching conditions from Claim 5.1 during a single traversal of the barrier trees. At each visited concurrent alternation or quantification subtree the two matching conditions are verified by inspecting the computed cnt value. A synchronization error warning is issued when one of the conditions is violated. If no warnings were produced we have verified that the program is free of barrier synchronization errors.

Figure 5 shows the computed cnt values in brackets next to each barrier tree node. The computed counts show that the trees T_P and T_Q are fixed-length and T_{main} is variable-length. The example is well-matched because all concurrent alternation subtrees are fixed-length and there are no concurrent quantification subtrees.

The complexity of applying the fixed-length calculation rules is linear, i.e., $O(B + C)$.

5.1 Barrier Matching Functions

If no synchronization warnings are reported, we compute the matching function $\mathcal{M}$ that maps each barrier statement b to the set of barrier statements that synchronize with b, for some instance of b. If $b_1 \in \mathcal{M}(b_2)$ we call the pair (b_1, b_2) a matching pair.

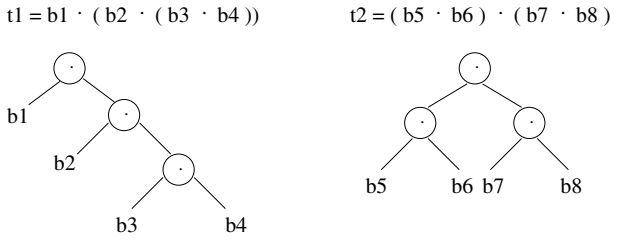


Figure 7. Two concatenation trees

Each verified concurrent alternation tree $t = t_1 \mid^c t_2$ contributes to the matching function by providing new matching pairs that match barriers in t_1 with the concurrently executing barriers in t_2.

First, consider the case where t_1 and t_2 are simple concatenation trees, that is, t_1 and t_2 only contain concatenation operations and all leaf nodes are barriers, as shown in the example in Figure 7. Thus, t_1 and t_2 each derive a single unique barrier sequence and, because t was verified, the two unique sequences have the same length. Computing the matching sets for the two trees is easy; we can match barriers from t_1 with barriers from t_2 left-to-right, as they are encountered during a depth-first traversal of the trees.

In more general terms, let *Match*(t_1, t_2) denote the query to compute all matching pairs that result from matching barriers in t_1 with barriers in t_2. We compute *Match*(t_1, t_2) by simultaneously traversing the two subtrees in depth-first order through a finite number of applications of a set of matching rules shown in Figure 8. To model a depth-first traversal each rule is associated with a state that denotes the current direction of the traversal (upwards $\uparrow$ or downwards $\downarrow$). Each matching rule transforms a directed query *Match*(t_1, t_2) into a new query with a new direction. Note that rules 1 and 4 are the only rules that change direction.

No.	Direc–tion	Match(t1, t2)	Actions	Direc–tion
1	↓	t1: b1 t2: b2	Add matching set { (b1, b2) }	↑
2	↓	t1 t2 (t3)	Match(t1, t3)	↓
3	↑	t1 t3 (t2)	Match(t1, t3)	↑
4	↑	(t1 t3) (t2 t4)	Match(t3, t4)	↓

Figure 8. Barrier matching rules for concatenation trees

To proof that the matching rules in Figure 8 correctly compute the matching sets between t_1 and t_2 we make the following claims.

Claim 5.2. *Applying the matching rules from Figure 8 to two equal-length concatenation trees t_1 and t_2 leads to a complete depth-first traversal of t_1 and t_2.*

Proof. It can easily be seen that rules 1 through 4 only produce moves in depth-first order. Furthermore, it can easily be shown that for any combination of subtrees from t_1 and t_2 there is a rule that applies. Thus, all nodes in t_1 and t_2 will eventually be visited, completing the depth-first traversal. □

Claim 5.3. *Applying the matching rules from Figure 8 to two equal-length concatenation trees t_1 and t_2 matches each barrier leaf node in t_1 with exactly one leaf node in t_2, and vice versa.*

Proof. Claim 5.2 implies that each leaf node will be matched at least once. Let $\{b_1, b_2)\}$ be one of those matches for leaf node b_1 in tree t_1. Thus, rule 1 has just been applied and the direction is upwards. For b_1 to be involved in a subsequent match, direction would have to be changed again in order to descend to another leaf node in t_2. Hence, rule 4 would have to be applied. However, rule 4 can only be applied to both subtrees simultaneously. Hence, b_1 cannot be matched with another leaf in t_2. The analogous argument shows that b_2 cannot be matched against additional leaf nodes in t_1. □

No.	Direc–tion	Match(t1, t2)	Actions	Direc–tion
5	↓	t1 t2 (t3 t4)	Match(t1, t3); Match(t1, t4)	↓
6	↑	t1 t3 (t2)	Match(t1, t3)	↑
7	↓	t1: Tp t2	Match (Tp, t2); Push t1 onto LeftStack	↓
8	↑	t1: Tp t2	t3 = pop(LeftStack); Match(t3, t2)	↑

Figure 9. Barrier matching rules for alternation and procedure calls

Claims 5.2 and 5.3 imply that for two concatenation trees t_1 and t_2 the matching rules match each barrier in t_1 correctly with a barrier in t_2.

In general, the subtrees t_1 and t_2 of a verified concurrent alternation tree do not only contain concatenations but also alternations and leaf nodes that denote procedure calls. Figure 9 shows the extensions of the matching rules that handle these cases. For each matching rule in Figure 9 there exists a symmetric counterpart, not shown here, where the roles of t_1 and t_2 are interchanged.

Alternation (rule 6) is handled by proceeding with two traversal sequences, one along each subtree of the alternation. The queries along the left subtree are resolved first before proceeding to the right subtree.

Procedure call leaf nodes (rule 7) are handled by continuing the traversal at the appropriate callee. To remember the correct call node when returning in the upwards direction, a stack is maintained for each traversal (*LeftStack* and *RightStack*).

To compute the complete matching function $\mathcal{M}$ for the entire program, the matching rules in Figures 8 and 9 are applied for each verified concurrent alternation subtree.

Consider now the complexity of applying the matching rules. There are $O(B + C)$ nodes in the barrier trees for the entire program so that the number of different queries that can be generated is $O((B + C)^2)$. Repeated processing of the same query in the same direction is redundant and would only reproduce the same matches. We can avoid redundant re-traversals by maintaining a visited flag for each pair of trees. Using visited flags we can ensure that the rules are applied to each pair of trees at most twice (once in each direction) so that the overall complexity of applying the matching rules is $O((B + C)^2)$.

Applying the matching rules to our example from Figure 5 produces *Match* queries for the three fixed-length concurrent alternation subtrees in Figure 5. The matching function results as follows: $\mathcal{M}(b_1) = \{b_6, b_7\}$, $\mathcal{M}(b_2) = \{b_4\}$, $\mathcal{M}(b_3) = \{b_5\}$, $\mathcal{M}(b_4) = \{b_2\}$, $\mathcal{M}(b_5) = \{b_5\}$, $\mathcal{M}(b_6) = \{b_1, b_7\}$, and $\mathcal{M}(b_7) = \{b_1, b_6\}$. Matching sets containing more than one element indicate the presence of textually unaligned barriers.

5.2 Counter Example

If the fixed-length calculation rules from Figure 6 reveal a synchronization error we construct a counter example to illustrates the error. Assume t is an error tree, that is, t is either a concurrent quantification tree or a variable-length concurrent alternation tree. The counter example for t consists of two concurrent program paths that include different numbers of barriers.

We construct the counter example by extracting appropriate barrier sequences from the error tree t and then expanding these sequences into program paths. If the error tree t contains alternations it represent multiple barrier sequences and extracting a single sequences requires making a selection at each alternation point. The kind of selection depends on the characteristics of the error tree.

Consider an error tree t that is a concurrent quantification tree, that is, $t = (t_1)^\star$ with $cnt(t) = \top$. Any sequence selected from t_1 exemplifies the error because the sequence is cyclic. Thus, we extract a sample sequence from t_1 by arbitrarily selecting one of the alternatives at each alternation operation.

Now consider an error tree t that is a concurrent alternation tree, that is, $t = t_1 \mid^c t_2$. We select four sequences from t, the shortest and the longest barrier sequence from each subtree t_1 and t_2. Among the four choices we obtain at least two sequences that have a different number of barriers because t_1 and t_2 are not equal-length. We construct the shortest (longest) barrier sequence from a subtree during a bottom-up traversal. The traversal recomputes the count values cnt using the calculation rules from Figure 6. However, when encountering an alternation subtree t we select the alternative with the lower (higher) barrier count and recompute the barrier count for t by copying the barrier count from the selected alternative.

It remains to expand the selected barrier sequences into program paths. This expansion is done by reversing the transformations that were performed when first constructing the barrier expression from the corresponding path expression.

6. Experimental Evaluations

We have implemented multi-valued expression slicing and barrier matching for MPI/C programs as part of the Eclipse Parallel Tools Platform (PTP) project (www.eclipse.org/ptp). The analysis was built on top of the open-source CDT (C Development Tool) in Eclipse that constructs ASTs for C programs.

MPI offers the concept of communicators to limit the scope of a barrier to a specific subset of the executing threads. The default communicator is MPI_COMM_WORLD which includes all MPI threads. To handle MPI-style communicators our tool analyzes the program separately for each communicator.

Our current implementation treats pointers conservatively – every dereference of a pointer and every variable whose address is taken, except function parameters, is considered to be multi-valued. Function pointers are currently not handled. Aliases of communicators are handled by conservatively assuming separate communicators. In most cases a simple Anderson's style pointer analysis [2] would have been sufficient to accurately determine the aliases of communicators. However, our experimentation shows that even a conservative handling of pointers does not produce spurious warnings for our benchmark set.

We evaluated our MPI barrier checker on a set of MPI benchmarks, listed in Table 1. None of these benchmarks have any known synchronization errors so the main purpose of our experimentation is to determine whether our checker tool generates spurious synchronization error warnings when applied to realistic applications.

Armci [1] is an AggregateRemote Memory Copy Interface. FFTW [2] is a C subroutine library for computing the Discrete Fourier Transform (DFT). MPB (MIT Photonic

[1] http://www.emsl.pnl.gov:2080/docs/parsoft/armci

[2] http://www.fftw.org

Benchmarks	Armci	FFTW	MPB	ParMetis	SBLAS	Skampi	Tcgmsg
# Source lines	19413	67901	8027	579167	4090	15430	3885
# Procedures	445	558	148	338	71	402	84
# Barrier statements	44	6	3	67	24	3	6
# Communicators	1	1	2	67	1	3	2
# Nodes in barrier trees	2936	579	450	1014133	3297	628	1290
Size of the largest matching set	1	1	2	1	1	1	1
Fraction of multi-valued predicates in barrier trees	0	0	14.3%	0	0	0	0

Table 1. MPI benchmark programs and their barrier checking results

Bands) [3] computes the band structures (dispersion relations) and electromagnetic modes of periodic dielectric structures. ParMetis [4] is a parallel library that implements a variety of algorithms for partitioning unstructured graphs and meshes, and for computing fill-reducing orderings of sparse matrices. SBLAS is a set of single precision Basic Linear Algebra (BLAS) subroutines. Skampi [5] is a suite of tests designed to measure the performance of MPI. Tcgmsg is an MPI version toolkit for writing portable parallel programs using MPI. Table 1 lists the sizes of these benchmarks and the number of barrier statements in each benchmark.

Armci, FFTW, MPB and sBLAS use the global communicator MPI_COMM_WORLD, which includes all processes. Tcgmsg uses two distinct communicators MPI_COMM_WORLD and TCGMSG_Comm. Although TCGMSG_Comm is an alias of the default communicator MPI_COMM_WORLD, we conservatively treat it as a separate communicator. ParMetis refers to communicators through pointer dereferences, and Skampi refers to communicators through function calls. We conservatively treat each textual communicator in ParMetis and Skampi as a separate communicator.

Our prototype was capable of verifying the synchronization structure in all benchmarks without producing spurious error warning. The size of the largest matching set provides information about textually unaligned barriers. The size of the largest matching set is one for all benchmarks except MPB. Thus,the barriers in all benchmarks except MPB are textually aligned.

7. Related Work

The most relevant previous work on verifying barrier synchronization is the work by Aiken and Gay on a barrier inference rule system [1]. Their analysis detects the same class of synchronization errors as ours, however, they require user annotations to handle procedures and their analysis does not explicitly compute the matching function among barriers. There have been other approaches to verifying synchronization in parallel programs using model checking [21, 20]. The techniques based on model checking do not share the assumption of structural correctness but they are more expensive resulting in scalability problems. There have also been some efforts on static checking of shared memory programs. One such example is Calvin [8], which is based on automatic theorem proving.

Other related work includes barrier optimization approaches that optimize the usage of barriers [5, 26, 16] by eliminating unnecessary barriers or optimizing the placement of barriers. Some research work identifies communication patterns, such as send/receive pairs, for MPI programs [19].

The multi-valued expression problem has first been addressed by the inference rule system by Aiken and Gay [1]. Aiken and Gay suggest to introduce a *single* qualifier as was done in the Titanium language [18] to explicitly describe expressions that are single-valued.

There has been a large of body of work on concurrency analysis of parallel programs, including SPMD programs [3, 6, 15]. Concurrency analysis uses the synchronization constructs in the program to determine which portions of the program may execute in parallel. Some concurrency analyses focus on analyzing the barriers in the program to establish concurrency information [10, 12, 13]. However, these approaches do not verify the correctness of barrier synchronization.

8. Conclusions

We present in this paper a new approach to verifying barrier synchronization that uses a combination of program slicing and path expressions. Our analysis computes barrier expressions as a compact representation of the synchronization structure of the program and as a foundation to barrier matching. The presented barrier matching analysis computes the synchronizing barrier statements in addition to the verification result. Information about synchronizing barriers can be used to identify and eliminate textually unaligned barriers. Our analysis is practical and scales well to large applications. We implemented the analysis in a MPI barrier checking tool and successfully analyzed a number of realistic MPI benchmarks.

We are currently working on completing out MPI checker tool with a more comprehensive alias analysis and a graphical user interface to visualize the analysis results. So far we have used our path expression based approach to model barrier synchronization. In the future we plan to extend this work to analysis point-to-point message passing communication.

[3] http://ab-initio.mit.edu/wiki/index.php/MPI_Photonic_Bands

[4] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

[5] http://liinwww.ira.uka.de/ skampi/

Acknowledgments

We would like to thank Beth Tibbitts from the IBM T.J. Watson Research Center for providing Eclipse and PTP expertise to support the tool implementation. This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004 as part of the IBM PERCS project.

References

[1] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–354, 1998.

[2] L.O. Andersen. *Program Analysis and Specialization For the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[3] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 21–30, Seattle, Washington, March 1990.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] Alain Darte and Robert Schreiber. A linear-time algorithm for optimal barrier placement. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 26–35, 2005.

[6] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 36–48, 1991.

[7] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, Toulouse, France, May 1994.

[8] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.

[9] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[10] Tor E. Jeremiassen and Susan J. Eggers. Static analysis of barrier synchronization in explicitly parallel systems. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 171–180, Montréal, Québec, August 1994. North-Holland Publishing Company.

[11] A Krishnamoorthy, D Culler, A Dusseau, S Goldstein, S Lumetta, T von Eicken, and K Yelick. Parallel Programming in Split-C. In *Supercomputing '93 Proceedings*, pages 262–273, November 1993.

[12] Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space programs. *J. Parallel Distrib. Comput.*, 38(2):130–144, 1996.

[13] Yuan Lin. Static nonconcurrency analysis of openmp programs. In *Fist International Workshop on OpenMP*, 2005.

[14] OpenMP C/C++ Manual. http://www.openmp.org/specs/.

[15] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 129–138, San Diego, California, May 1993.

[16] Michael O'Boyle and Elena Stohr. Compile time barrier synchronization minimization. *IEEE Trans. Parallel Distrib. Syst.*, 13(6):529–543, 2002.

[17] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *Software Engineering Notes*, 9(3), 1984.

[18] P.Hulfinger, D.Bonachea, K.Datta, D.Gay, S.Graham, B.Liblit, G.Pike, J.Su, and K.Yelick. Titanium language reference manual. Technical Report UCB/EECS-2005-15, U.C.Berkeley, 2005.

[19] Shuyi Shao, Alex K. Jones, and Rami Melhem. A compiler-based communication analysis approach for multiprocessor systems, 2006.

[20] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free mpi programs for verification. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 95–106, 2005.

[21] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 157–168, 2006.

[22] The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[23] Robert E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[24] Robert E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[25] Steven W. K. Tjiang and John L. Hennessy. Sharlit—a tool for building optimizers. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82–93, San Francisco, California, June 1992.

[26] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 144–155, Santa Barbara, California, July 1995.

[27] Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Conference Proceedings, 1995 International Conference on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.

[28] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, Toronto, Ontario, June 1991.

[29] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.