University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers

Yuan Zhang
Evelyn Duesterwald†
Guang R. Gao

CAPSL Technical Memo 079
July, 2007

Copyright © 2007 CAPSL at the University of Delaware

†IBM T.J.Watson Research Center, Hawthorne, NY 10532. Email: duester@us.ibm.com

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Concurrency analysis is a static analysis technique that determines whether two statements or operations in a shared memory program may be executed by different threads concurrently. Concurrency relationships can be derived from the partial ordering among statements imposed by synchronization constructs. Thus, analyzing barrier synchronization is at the core of concurrency analyses for many parallel programming models. Previous concurrency analyses for programs with barriers commonly assumed that barriers are named or textually aligned. This assumption may not hold for popular parallel programming models, such as OpenMP, where barriers are unnamed and can be placed anywhere in a parallel region, i.e., they may be textually unaligned. We present in this paper the first interprocedural concurrency analysis that can handle OpenMP, and, in general, programs with unnamed and textually unaligned barriers. We have implemented our analysis for OpenMP programs written in C and have evaluated the analysis on programs from the NPB and SpecOMP2001 benchmark suites.

Contents

List of Figures

List of Tables

1 Introduction

Concurrency analysis is a static analysis technique that determines whether two statements or operations in a shared memory program may be executed by different threads concurrently. Concurrency analysis has various important applications, such as statically detecting data races [6, 9], improving the accuracy of various data flow analysis [16], and improving program understanding. In general, precise interprocedural concurrency analysis in the presence of synchronization constraints is undecidable [15], and a precise intraprocedural concurrency analysis is NP-hard [18]. Therefore, a practical solution is to make a conservative estimate of all possible concurrency relationships, such that two statements that are not determined to be concurrent cannot execute in parallel in any execution of the program. If two statements are determined to be concurrent, they *may* execute concurrently.

In this paper we present a new interprocedural concurrency analysis that can handle parallel programming models with unnamed and textually unaligned barriers. We present our analysis in the context of the OpenMP programming model but our approach is also applicable to other SPMD (Single Program Multiple Data) parallel programming models.

OpenMP is a standardized set of language extensions (i.e., pragmas) and APIs for writing shared memory parallel applications in C/C++ and FORTRAN. Parallelism in an OpenMP program is expressed using the `parallel` construct. Program execution starts with a single thread called the *master thread*. When control reaches a `parallel` construct, a set of threads, called a *thread team*, are generated, and each thread in the team, including the master thread, executes a copy of the parallel region. At the end of the parallel region the thread team synchronizes and all threads except for the master thread terminate. The execution of the parallel region can be distributed among the thread team by work-sharing constructs (e.g., `for`, `sections` and `single`).

Synchronization is enforced mainly by global barriers and mutual exclusion (i.e., `critical` constructs and lock/unlock library calls). When a thread reaches a barrier it cannot proceed until all other threads have arrived at a barrier. In OpenMP, barriers are unnamed and they may be textually unaligned. Thus, threads may synchronize by executing a set of textually distinct barrier statements. Textually unaligned barriers make it difficult to reason about the synchronization structure in the program. Some parallel languages, therefore, require barriers to be textually aligned [19]. Textually unaligned barriers also hinder concurrency analysis because understanding which barrier statements form a common synchronization point is a prerequisite to analyzing the ordering constraints imposed by them. Our analysis is the first interprocedural concurrency analysis that can handle barriers in OpenMP and, in general, programs with unnamed and textually unaligned barriers. Figure 1 shows an OpenMP example program with a parallel region.

Barriers structure the execution of a parallel region into a series of synchronized execution phases, such that threads synchronize on barriers only at the beginning and at the end of each phase. Computing these execution phases for each parallel region provides the basic skeleton for ordering relationships among statements. Statements from different execution phases cannot

```
main()
{
     int my_ID, num, i, y, sum = 0;
     ......
     #pragma omp parallel private(my_ID, num, y)
     {
          my_ID = omp_get_thread_num();
C1:       if(my_ID > 2){
S1:            i = 0;
               #pragma omp barrier  // b1
S2:            y = i + 1;
          } else {
S3:            i = 1;
               #pragma omp barrier  // b2
S4:            y = i − 1;
          }
P1:       sum += my_ID;
C2:       if( my_ID == 0){
S5:            ......
               #pragma omp barrier  // b3
S6:            ......
          }
```

```
C3:       if( my_ID != 0){
S7:            ......
               #pragma omp barrier  // b4
S8:            ......
          }
P2:       num = omp_get_num_threads();
C4:       if(num > 2){
S9:            ......
               #pragma omp barrier  // b5
S10:           ......
          } else {
S11:           ......
               #pragma omp barrier  // b6
S12:           ......
          }
C5:       if(my_ID == 0)
               printf("i = %d\n", i);
     } // end of parallel
} // end of main
```

Figure 1: Example OpenMP program. The OpenMP library function calls *omp_get_thread_number()* and *omp_get_num_threads()* return the thread identifier of the calling thread and the total number of threads in the current team, respectively.

execute concurrently. Thus, only statements within the same phase need to be examined for computing the concurrency relation.

To illustrate the concept of execution phases consider the sample program shown in Figure 1. The first execution phase, denoted as $(begin, \{b_1, b_2\})$, starts at the beginning of the parallel region and extends up to barriers b_1 and b_2. Note that barriers b_1 and b_2 establish a common synchronization point, i.e., they *match*. The next barrier synchronization point is at barriers $\{b_3, b_4\}$. Hence, the next execution phase is $(\{b_1, b_2\}, \{b_3, b_4\})$.

It is easy to see that statements from two different execution phases are ordered by barriers and thus cannot be concurrent. On the other hand, two statements from the same execution phase may be concurrent, such as S_1 and S_3 in Figure 1. However, barriers are not the only constructs that need to be considered to determine execution phases. Additional ordering constraints may be imposed by control constructs. Consider statements S_9 and S_{11} in Figure 1, which are on different branches of the condition C_4. Since all threads agree on the value of predicate C_4 (i.e., the predicate is *single-valued*), statements S_9 and S_{11} can never be executed together in one execution, hence they cannot be concurrent. On the contrary, predicate C_1 is evaluated differently by different threads (i.e., the predicate is *multi-valued*), so that statements on the two branches may execute concurrently. Thus, another key issue in understanding the concurrency constraints is determining whether a control predicate is single- or multi-valued.

In this paper, we propose an interprocedural concurrency analysis technique that addresses the above ordering constraints imposed by synchronization and control constructs. Our analysis computes for each statement s the set of statements that may execute concurrently with s. The analysis proceeds in four major steps:

Step 1: CFG construction: The first step consists of constructing a control flow graph

(CFG) that correctly models the various OpenMP constructs.

Step 2: Barrier matching: As a prerequisite to computing execution phases we need to understand which barrier statements synchronize together, i.e, which barrier statements *match*. We solve this problem as an extension to barrier matching analysis [20]. Barrier matching verifies that barriers in an SPMD program are free of synchronization errors. For verified programs, a *barrier matching function* is computed that maps each barrier statement s to the set of barrier statement that synchronize with s in at least one execution. Barrier matching was previously described for MPI programs and we have extended it to handle OpenMP programs. The computed barrier matching function is an input to the next step.

Step 3: Phase partition and aggregation: In this step, we first partition the program into a set of static execution *phases*. A *phase* (b_i, b_j) consists of a set of basic blocks that lie on a barrier-free path between barrier b_i and b_j in the CFG. We then aggregate phases (b_p, b_q) and (b_m, b_n) if b_p matches b_m, and b_q matches b_n. A dynamic execution phase at runtime is an instance of an aggregated static execution phase.

Step 4: Concurrency relation calculation: We first conservatively assume that statements from the same execution phase may be concurrent but statements from different phases are ordered and non-concurrent. We then apply a set of ordering rules that reflect the concurrency constraints from other OpenMP synchronization and work-sharing constructs to iteratively refine the concurrency relation.

We have implemented the analysis for OpenMP programs written in C and evaluated it on programs from the NPB [5] and SpecOMP2001 [17] benchmark suites. Our evaluation shows that our concurrency analysis is sufficiently accurate with the average size of a concurrency set for a statement being less than 6% of total statements in all but one program.

The rest of the paper is organized as follows. We first present related work in Section 2. The control flow graph is presented in Section 3. In Section 4 we first review the barrier matching technique and then present extensions to handle multi-valued expressions in OpenMP and structurally incorrect programs. Phase partition and aggregation is presented in Section 5, and the concurrency relation calculation is presented in Section 6. We present experimental results in Section 7, and finally conclude in Section 8.

2 Related Work

A number of researchers have looked at concurrency analysis for programs with barriers. Lin [9] proposed a concurrency analysis technique (called non-concurrency analysis) for OpenMP programs based on phase partitioning. Lin's analysis differs from our concurrency analysis in two main aspects. First, Lin's method is intraprocedural and cannot compute non-concurrency relationship across procedure calls. Second, Lin's method cannot account for synchronization across textually unaligned barriers. The analysis does not recognize that textually unaligned barriers may in fact synchronize together, resulting in spurious non-concurrency relationships.

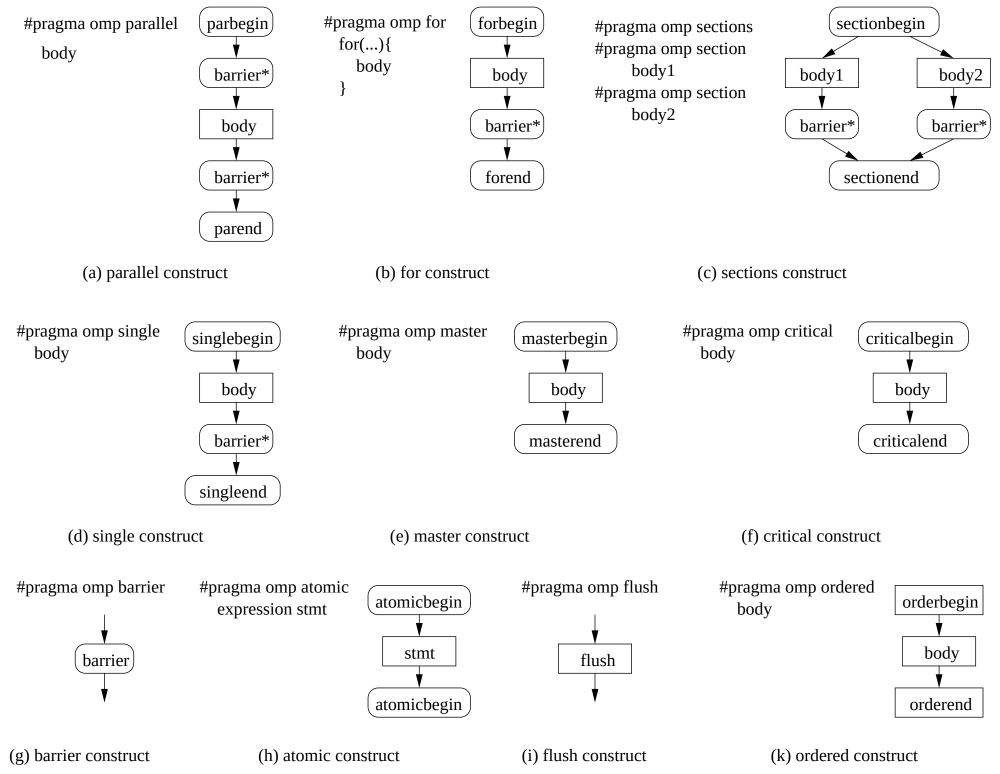


Figure 2: Control flow graph construction

For instance, Lin's technique would wrongfully conclude that S_1 and S_3 in Figure 1 are non-concurrent.

Jeremiassen and Eggers [7] present a concurrency analysis technique that, similar to our analysis, first partitions the program into phases, then aggregates some phases together. Their analysis avoids the problem of having to identify whether textually unaligned barriers synchronize together by assuming that barriers are named through barrier variables. Barriers statements that refer to the same barrier are assumed to be matched. Their techniques also does not account for concurrency constraints imposed by control constructs with single-valued predicate. For instance, in Figure 1 their analysis would conclude that S_9 and S_{11} are concurrent.

Kamil and Yelick [8] proposed a concurrency analysis method for the Titanium language [19] in which synchronization across textually unaligned barriers is not allowed.

There also has been a lot of work on concurrency analysis for other parallel programming languages, such as Ada and Java [3, 2, 6, 11, 13] in which synchronization is mainly enforced by event-driven constructs like post-wait/wait-notify. Agarwal et.al. [1] presents a may-happen-in-parallel analysis for X10 programs.

3 Step 1: Control Flow Graph Construction

The control flow graph for an OpenMP program is an extension of the control flow graph for a sequential program. Figure 2 illustrates the graph construction for each OpenMP construct. *Begin* and *end* nodes are inserted for each OpenMP directive with a construct body. To model the `sections` construct, we insert a control flow edge from the *begin* node to the first statement node of each section in the construct, and a control flow edge from the last statement node of each section to the *end* node of the `sections` construct. Constructs without a body statement (e.g., `barrier` and `flush`) are represented by a single block.

There is an implicit barrier at the end of the work-sharing constructs `for`, `sections` and `single`, unless the `nowait` clause is specified. Implicit barriers are depicted as `barrier*` in Figure 2. Similarly, there is an implicit barrier at the beginning of a parallel region, and an implicit barrier at the end of a parallel region.

4 Step 2: Barrier Matching

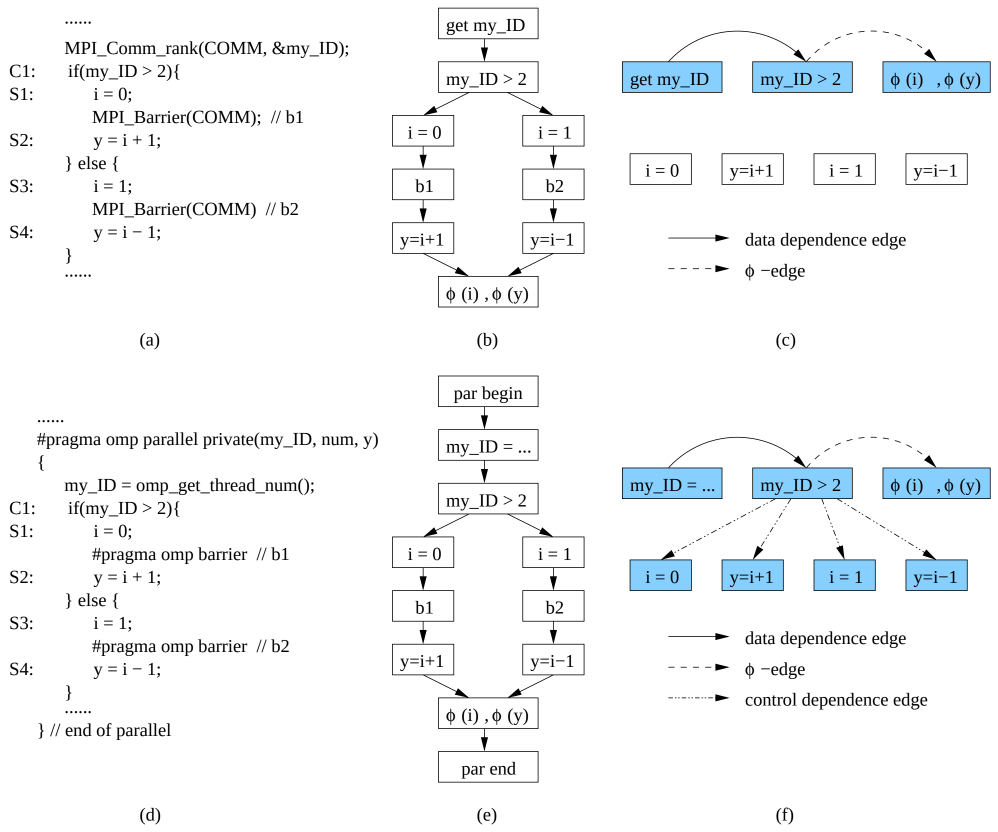
The second step in our concurrency analysis consists of identifying the matching barrier statements that synchronize together. Barrier matching analysis [20] was previously described for MPI programs. In this section we first review the MPI barrier matching analysis and then show how to extend it to handle OpenMP.

4.1 Review of Barrier Matching for MPI Programs

Barrier matching is an analysis and verification technique to detect stall conditions caused by barriers. When the program is verified, the analysis computes a barrier matching function that maps each barrier statement s to the set of barrier statements that synchronize with s in at least one execution. The MPI barrier matching analysis proceeds in three main steps:

Multi-valued Expression Analysis: In SPMD-style programs all threads execute the same program but they may take different program paths. The ability to determine which program paths may be executed concurrently requires an analysis of the *multi-valued* expressions in the program. An expression is called multi-valued if it evaluates differently in different threads. If used as a control predicate, multi-valued expressions split threads into different program paths that are executed concurrently by different threads. An example of a multi-valued expression is my_ID shown in Figure 3(a). Conversely, an expression that has the same value in all threads is called *single-valued*. SPMD programming paradigms like MPI or OpenMP usually contain multi-valued seed expressions, such as library calls that return the unique thread identifier. All other multi-valued expressions in the program are directly or indirectly dependent on these multi-valued seed expressions.

The interprocedural multi-valued analysis is solved as a forward slicing problem based on a revised program dependence graph. The revised program dependence graph contains nodes to



```
        ......
        MPI_Comm_rank(COMM, &my_ID);
C1:      if(my_ID > 2){
S1:           i = 0;
              MPI_Barrier(COMM);  // b1
S2:           y = i + 1;
        } else {
S3:           i = 1;
              MPI_Barrier(COMM)  // b2
S4:           y = i − 1;
        }
        ......
```

```
     ......
     #pragma omp parallel private(my_ID, num, y)
     {
        my_ID = omp_get_thread_num();
C1:      if(my_ID > 2){
S1:           i = 0;
              #pragma omp barrier  // b1
S2:           y = i + 1;
        } else {
S3:           i = 1;
              #pragma omp barrier  // b2
S4:           y = i − 1;
        }
        ......
     } // end of parallel
```

Figure 3: An MPI program (a), its CFG (b), and its revised program dependence graph (c). An OpenMP program (d), its CFG (e), and its revised program dependence graph (f). The multi-valued expression slices are shown as shaded nodes in (c) and (f).

represent statements that are connected through data dependence edges and so called ϕ-edges. ϕ-edges are based on the notion of ϕ-nodes in Static Single Assignment (SSA) form [4]. In SSA, a ϕ-node is inserted at a join node where multiple definitions of a variable merge. The predicate that controls the join node is called a ϕ-gate. A ϕ-edge connects a ϕ-gate with the corresponding ϕ-node. Multi-valued expressions result as those expressions that are reachable from a multi-valued seed expression along either data-dependence or ϕ-edges in the revised program dependence graph.

Figure 3(c) illustrates the revised program dependence graph and multi-valued expression analysis for the MPI program shown in Figure 3(a). It is important to note that variables i and y are single-valued for the executing threads inside the conditional statement but they become multi-valued after the conditional paths merge at the ϕ-node.

Barrier Expressions: A barrier expression at a node n in the CFG represents the sequences of barriers that may execute along any paths from the beginning of the program to node n. Barrier expressions are regular expressions with barrier statements and function labels as terminal symbols, and three operators: concatenation ($\cdot$), alternation ($|$) and quantification ($*$), which represents barriers in a sequence, in a condition, and in a loop, respectively. For

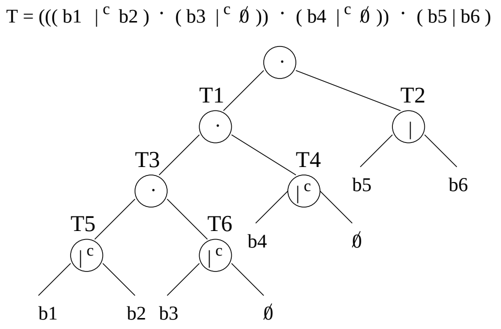

T = (((b1 |^c b2) · (b3 |^c ∅)) · (b4 |^c ∅)) · (b5 | b6)
T1
T2
T3
T4
T5
T6
b1
b2
b3
∅
b4
∅
b5
b6

Figure 4: The barrier expression tree for the program in Figure 1. The symbol $|^c$ denotes alternation with a multi-valued predicate.

example, the barrier expression for Figure 3(a) is $(b_1|\ b_2)$. A barrier expressions is usually represented by a barrier expression tree. Figure 4 shows the barrier expression tree for the program shown in Figure 1.

Barrier matching: The final step combines the results of the previous two steps to detect potential stall conditions caused by barriers. Recall that multi-valued predicates create concurrent paths. Thus, a barrier subtree whose root is an alternation with a multi-valued predicate describes two concurrent barrier sequences. Similarly, a quantification tree with a multi-valued predicate describes concurrent barrier sequences in a loop in which threads concurrently execute different numbers of iterations.

A barrier tree that does not contain either concurrent alternation or concurrent quantification describes a program in which all threads execute the same sequence of barriers (although the sequence may be different across different executions of the program). Such a tree is obviously free of barrier synchronization errors. A concurrent quantification tree signals a synchronization error because concurrent threads execute different numbers of loop iterations and hence different numbers of barriers. Therefore, the barrier verification problem comes down to checking that all concurrent alternation subtrees in the program's barrier tree are well-matched, i.e., the two alternation subtrees always produce barrier sequences of the same length. The barrier matching analysis implements this check by a counting algorithm that traverses the two subtrees of each concurrent alternation tree. Details of the counting algorithm can be found in [20].

After verifying a concurrent alternation barrier tree, the analysis computes the barrier matching function by ordering the leave nodes from each of its two subtrees in a depth-first order, and then matching barriers in the same position of the two ordered sequences.

4.2 Multi-valued Expressions Analysis for OpenMP Programs

In order to use barrier matching for our concurrency analysis, we developed an extension of the multi-valued expressions analysis for shared variables. In MPI programs all variables are local to the executing thread. In OpenMP programs, on the other hand, variables are either shared or private. Private variables are stored in thread private memory and observable only by the executing thread. Private variables in OpenMP can therefore be handled in the same way as variables in MPI programs. Shared variable are stored in global memory, and observable by all threads simultaneously. However, due to the relaxed memory consistency model in OpenMP, a thread may have its own temporary view of memory which is not required to be consistent with global memory at all times. "A value written to a (shared) variable can remain in the thread's temporary view until it is forced to memory at a later time. Likewise, a read from a variable may retrieve the value from the thread's temporary view unless it is forced to read from memory" [10].

Consider again our sample program shown in Figure 3(d) and the two concurrent assignments to the shared variable i. When a thread reads the value of i subsequent to one of the assignments, it may retrieve the value assigned to i by itself from the thread's temporary view, or the value assigned by other threads from global memory. Thus, the shared variable assignment to i makes i multi-valued.

We extend multi-valued expression analysis for shared variables by incorporating the following additional rule on how a shared variable may become multi-valued. An expression e involving a shared variable v is multi-valued, if e is control dependent on a multi-valued predicate p (i.e., e lies on a concurrent path) and there exists at least one definition of v that is control-dependent on p.

In order to model shared variable in the revised program dependence graph we insert selected control dependence edges. Specifically, let n be a ϕ-node for a shared variable v that is connected to a ϕ-gate with a predicate p. We insert control dependence edges from predicate p to all control dependent nodes that contain a reference of the shared variable v. Thus, if predicate p becomes multi-value during the slicing computation, so will any expression involving the shared variable v that is control dependent on p. Figure 3(f) illustrates this modification to the revised program dependence graph for shared variable i.

Based on this extension of the revised program dependence graph, we can apply the interprocedural forward slicing algorithm used in original MPI analysis to compute multi-valued expressions for private and shared variables in OpenMP. Figure 3(f) shows the resulting multi-valued expressions as the set of shaded nodes in the graph. The computed slice correctly indicates that shared variable i is multi-valued at statements $S2$ and $S4$.

Note that the `flush` construct in OpenMP makes the calling thread's temporary view of memory consistent with global memory. Therefore, a shared variable is always single-valued after a `flush` construct. However, a `flush` construct only takes effect on the calling thread. Correspondingly, shared variables become single-valued only on the flushing thread's program path, at the point immediately following the flush.

As in the original MPI multi-valued expression analysis, we assume OpenMP and other library calls are annotated as either single- or multi-valued. Arrays are treated as scalar variables and pointers are conservatively handled by treating every pointer dereference and every variable whose address is taken as multi-valued.

4.3 Barrier Trees and Barrier Matching for OpenMP Programs

Once the multi-valued expressions have been computed, barrier tree construction and barrier matching for OpenMP programs proceeds as described for MPI programs. Figure 4 shows the barrier expression tree for the program shown in Figure 1. Barrier matching checks the three concurrent alternation subtrees T_5, T_6 and T_4. The analysis verifies subtree T_5 as correct and reports that barriers b_1 and b_2 match. However, the two subtrees T_6 and T_4 cannot be statically verified and the analysis would report a potential error, warning that the subtrees are structurally incorrect.

4.4 Handling Structurally Incorrect Programs

Barrier matching analysis produces a barrier matching function only for verified programs. As a static analysis, barrier matching is conservative and may therefore reject a program, although the program produces no synchronization errors at runtime. Programs that will always be rejected are so called *structurally incorrect* programs. 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.). In other words, a structurally incorrect program contains a component that, if looked at in isolation, has a synchronization error, although in the context of the entire program no runtime error may result. Figure 1 is an example of a structurally incorrect program because it contains two structural components, the conditionals C_2 and C_3 that, if looked at in isolation, are incorrect. Thus, the overall program is deemed incorrect although no runtime synchronization error would result because C_3 is the logical complement of C_2. As reported in the previous section, barrier matching analysis reports a potential error for each of the two conditional components.

We discuss in this section modifications to compute partial barrier matching information for programs whose synchronization structure is dynamically correct (i.e., the program terminates) even if they cannot be statically verified. Our approach to handling structural incorrectness is to isolate the program region that cannot be statically verified, and to partition the program into structurally correct and structurally incorrect regions. Based on this partition we can apply barrier matching and, in turn, our concurrency analysis for the structurally correct components of the program. For the structurally incorrect regions we conservatively assume that all statements may execute concurrently.

When barrier matching encounters a program with a structurally incorrect component p, a synchronization error is detected when processing the root of the barrier expression subtree that represents p. We refer to such structural component as an error component. For example, the barrier tree in Figure 4, contains two error components T_4 and T_6.

Based on these error components we define two well-matched regions of a structurally incorrect program. The first well-matched region consists of any sequence of statements along an error-component-free path in the CFG that starts at the program entry and terminates at a program point immediately preceding an error component. Similarly, the second well-matched region consists of any sequence of statements along an error-component-free path in the CFG that starts at a program point immediately following an error component and terminates at the program exit. We define the "structurally incorrect region" as the remainder of the program, that is, any statement that is not included in one of the above well-matched regions. We conservatively treat all statements in the structurally incorrect region as concurrent and compute barrier matching functions for the structurally correct regions.

Consider again our example in Figure 4 and recall that barrier matching reports two error components, T_4 and T_6. The two well-matched regions of the program in Figure 1 are defined as follows. The first region starts at program entry and terminates at program point P_1 in Figure 1 which immediately precedes the error component T_6. The second region starts at program point P_2 which immediately follows the error component T_4 and extend up to program exit. All statements between P_1 and P_2 are assumed to be concurrent.

5 Step 3: Phase Partition and Aggregation

The third step of the OpenMP concurrency analysis uses the computed barrier matching function to divide the program into a set of static phases. A static phase (b_i, b_j) consists of a sequence of basic blocks along all barrier-free paths in the CFG that start at the barrier statement b_i and end at the barrier statement b_j. Note that b_i and b_j may refer to the same barrier statement.

The phase partition method proceeds as proposed by Jeremiassen and Eggers [7]. First we assume each barrier statement b_i corresponds to a new global variable V_{b_i}. We then treat each barrier statement as a use of its corresponding barrier variable, followed by definitions of all barrier variables in the program. The problem of phase partition is then reduced to computing live barrier variables in the program. Recall that a variable v is live at program point p if the value of v at p is used before being re-defined along some path in the control flow graph starting at p. Precise interprocedural live analysis has been described in [12]. Let $Live(b)$ denote the set of barrier variables live at the barrier b. The set of static phases in an OpenMP program is then summarized as:

$$(b_i, b_j) | V_{b_j} \in Live(b_i), \ \ for \ \ all \ \ i \ \ and \ \ j$$

In order to determine which phases a basic block u belongs to, we need to reverse the control flow edges in the CFG and calculate live barrier variables for each basic block again. Let $LiveR(u)$ denote the set of live barrier variables at basic block u in the reversed CFG. The phases which block u belongs to are:

$$\{(b_i, b_j) | b_i \in LiveR(u) \wedge b_j \in Live(u)\}$$

According to the barrier matching information, we then aggregate phases (b_m, b_n) and (b_p, b_q) if barriers b_m matches b_p and b_n matches b_q. A dynamic execution phase is an instance of an aggregated phase at runtime.

6 Step 4: Concurrency Relation Calculation

The final step of our concurrency analysis consists of calculating the concurrency relation among basic blocks. Since basic blocks from different aggregated phases are separated by barriers, no two blocks in different phases can be executed concurrently. We can therefore establish a first safe approximation of the concurrency relation in the program by assuming that all blocks from the same aggregated phase may be concurrent. However, this first approximation is overly conservative and does not take concurrency constraints from certain OpenMP constructs into account. We have developed the following set of concurrency rules that address these constraints to refine the initial concurrency approximation.

1. **(Concurrency Rule)** Any two (possibly identical) basic blocks from the same aggregated phase are concurrent. The set of concurrency relationships obtained from this rule is denoted as CR.

2. **(Non-concurrency Rules)**

   (a) Any two basic blocks from a `master` construct under the same parallel region are not concurrent because they are executed serially by the master thread.

   (b) Any two basic blocks from the `critical` constructs with the same name (or from within the lock regions, enclosed by the omp_set_lock() and omp_unset_lock() library calls, that are controlled by the same lock variable) are not concurrent because they are executed mutually exclusively. Note that we treat two potentially aliased lock variables as different.

   (c) Two blocks in the same `ordered` construct are not concurrent because the `ordered` construct body within a loop is executed in the order of loop iterations.

   (d) Two blocks from the same `single` construct that is not enclosed by a sequential loop are not concurrent. Note that OpenMP requires a `single` construct body to be executed by one thread in the team, but it does not specify which thread. Therefore two instances of a `single` construct inside a sequential loop might be executed by two different threads concurrently.

   The set of non-concurrency relationships obtained from the non-concurrency rules is denoted as NCR.

Finally, the concurrency relation among units results as $CR - NCR$.

Returning to our sample program in Figure 1. S_1 and S_3 are concurrent because they are in the same aggregated phase $(start, \{b_1, b_2\})$. The same holds for S_2 and S_4. However, S_9 and S_{11} are not concurrent because barrier b_5 does not match barrier b_6 (due to the single-valued predicate C_4) thus S_9 and S_{11} are in different phases.

Benchmark	FT	IS	LU	MG	SP	quake
Source	NPB2.3-C	NPB3.2	NPB2.3-C	NPB2.3-C	NPB2.3-C	SpecOMP2001
# Souce Lines	1162	629	3471	1264	2991	1591
# Blocks	682	278	2132	909	2503	1191
# Procedures	17	9	18	15	22	27
# Barriers	13	5	30	28	67	13
OpenMP constructs	`single` `master` `for` `critical`	`for`	`for` `single` `critical` `master` `flush`	`single` `for` `critical`	`for` `master`	`for`
# Aggr. phases	29	11	41	103	223	24
Max. concurrency set size	101	59	83	256	130	33
Relative max. concurrency. set size	14.8%	21.2%	3.9%	28.1%	5.2%	2.8%
Avg. concurrency set size	40	36	23	50	52	15
Relative avg. concurrency. set size	5.9%	12.9%	1.1%	5.5%	2.1%	1.3%

Table 1: Experimental results

7 Experimental Evaluations

We have implemented the concurrency analysis for OpenMP/C programs on top of the open-source CDT (C Development Tool) in Eclipse. The Eclipse CDT constructs Abstract Syntax Trees for C programs. We evaluated the effectiveness of our OpenMP concurrency analysis on a set of OpenMP programs from the NPB (Nas Parallel Benchmarks) and SpecOMP2001 benchmark suites, as shown in Table 1.

FT (3-D FFT), LU (LU solver), MG (Multigrid), and SP (Pentadiagonal solver) are derived from the serial Fortran versions of NPB2.3-serial by the Omni OpenMP compiler project [14]. IS (Integer sort) is an OpenMP C benchmark from NPB3.2. Quake from SpecOMP2001 benchmark suite simulates seismic wave propagation in large basins.

The top part of Table 1 lists several characteristics of the benchmark programs such as the number of source lines, the number of barriers, either explicit or implicit, and the various OpenMP constructs used in each benchmark.

The results of the concurrency analysis are shown in the lower part of the table. As an intermediate result, the table lists the number of aggregated phases that have been computed. To estimate the accuracy of our concurrency analysis we computed the average and maximum set size among the concurrency sets for all nodes in the CFG. Our CFG is based on the CDT and includes statement level block nodes. Set sizes would be smaller if statements would be composed into basic block nodes. The table shows the absolute set size and the relative size which is the percentage of the total number of nodes in the CFG. Recall that the concurrency set of a block b consists of a set of blocks that might execute concurrently with b in at least one execution. A concurrency set is usually a superset of the real concurrency relation. Therefore the smaller the concurrency set, the less conservative our concurrency analysis is. Table 1 indicates that our analysis is not overly conservative since the size of the average concurrency set is less than 6% of the total blocks for all benchmarks except IS, for which the average concurrency set is 12.9% of the total number of blocks in the program.

8 Conclusions

In this paper we present the first interprocedural concurrency analysis that can handle OpenMP and, in general, shared memory programs with unnamed and textually unaligned barriers. Our approach is built on the barrier matching technique that has previously been described to verify barrier synchronization in MPI. We extended barrier matching to handle shared variables and OpenMP. We have implemented our analysis for OpenMP C programs and evaluated the effectiveness of our analysis using benchmarks from the NPB and SpecOMP2001 benchmark suites. The experimental results confirm that our analysis is not overly conservative. We are currently exploring the use of our concurrency analysis in combination with a dynamic data race detection tool by limiting the instrumentation points that have to be considered during dynamic checking. Other potential uses are in combination with performance tools to point the user to areas with low levels of concurrency.

Acknowledgement

This work is, in part, based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193, New York, NY, USA, 2007. ACM Press.

[2] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *PPOPP '90: Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, 1990.

[3] David Callahan and Jaspal Sublok. Static analysis of low-level synchronization. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, 1988.

[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 Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[5] NASA Advanced Supercomputing Divsion. Nas parallel benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[6] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV4: Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 36–48, 1991.

[7] T. Jeremiassen and S. Eggers. Static analysis of barrier synchronization in explicitly parallel systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Montreal, Canada, 1994.

[8] Amir Ashraf Kamil and Katherine A. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. Technical Report UCB/EECS-2006-41, EECS Department, University of California, Berkeley, Apr 2006.

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

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

[11] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *PPOPP '93: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–138, 1993.

[12] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–230, New York, NY, USA, 1981. ACM Press.

[13] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 338–354, 1999.

[14] Omni OpenMP Compiler Project. Omni OpenMP Compiler. In http://phase.hpcc.jp/Omni/home.html.

[15] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming languages and Systems (TOPLAS)*, 22(2):416–430, 2000.

[16] Vugranam Sreedhar, Yuan Zhang, and Guang Gao. A new framework for analysis and optimization of shared memory parallel programs. Technical Report CAPSL-TM-063, University of Delaware, Newark, DE, 2005.

[17] Standard Performance Evaluation Corporation. SPEC OMP (OpenMP benchmark suite). In http://www.spec.org/omp/.

[18] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs, 1983.

[19] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

[20] Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 194–204, 2007.