Static Nonconcurrency Analysis of OpenMP Programs

Yuan Lin
Sun Microsystems, Inc.
yuan.lin@sun.com

1 Introduction

Writing correct and efficient parallel programs is more difficult than doing so for sequential programs. One of the challenges comes from the nature of concurrent execution of a parallel program by different threads.[1] Determining exact concurrency is NP-hard[10], and is impossible for real-world programs at compile time.

OpenMP provides an easy and incremental way to write parallel programs. The well-structured OpenMP constructs and well-defined semantics of OpenMP directives make compiler analyses more effective on OpenMP programs than on more loosely structured parallel programs that are solely based on runtime libraries, such as MPI and Pthreads.

In this paper, we present a static nonconcurrency analysis technique that detects, at compile time, whether two statements in an OpenMP program will not be executed concurrently by different threads in a team. Similar to the method presented in [5], ours is a close underestimation of the real nonconcurrency in a program. When our method determines that the executions of two statements are nonconcurrent, these two statements will not be executed concurrently. When the method fails, the two statements may, but need not, execute concurrently.

Our nonconcurrency analysis models and uses the semantics of OpenMP directives. For example, in the following codes,

```
 1. !$omp parallel
 2.
 3.   a = ...
 4.
 5.   !$omp single
 6.      b = ...
 7.      c = ...
 8.   !$omp end single
 9.
10.   !$omp do
11.      do i=1, 100
12.         c(i) = ...
13.      end do
14.   !$omp end do nowait
15.
16. !$omp end parallel
```

there is one implicit barrier at line 8, which partitions the statements inside the parallel region (lines 3-14) into two phases. Phase one contains statements 3 through 8, and phase two contains statements 10 through 14. No two statements from different phases (such as statements 3 and 12) will ever be executed concurrently, while statements within the same phase (such as statements 3

[1]Concurrency is where the execution order of different threads is not enforced, and thus synchronization must be used to control shared resources. Parallelism is where different threads actually execute in parallel. Parallelism is an instance of concurrency. Parallel execution is concurrent, but concurrent execution is not necessarily parallel.

$ORC(N)$	the immediately enclosing OpenMP construct for node N.
$ORC(N).type$	the type of $ORC(N)$, i.e. **root**, **do**, **sections**, **section**, **critical**, **single**, **master**, **ordered**.
$ORC(N).crit.name$	the name for $ORC(N)$ whose $ORC(N).type$ is **critical**.
$ORC(N).ordered.bound$	the binding worksharing **do** loop for $ORC(N)$ whose $ORC(N).type$ is **ordered**
$ORC(N).parent$	the parent OpenMP construct of $ORC(N)$ in the OpenMP region tree
$ORC(N).pregion$	the parallel region that encloses $ORC(N)$

Table 1: Notations: attributes of different types OpenMP constructs

and 6, or two instances of statement 3) may. In addition, the **single** directive mandates only one thread can execute statements 6 and 7. Therefore statements 6 and 7, though in the same phase, will never execute concurrently. Our analysis is able to recognize the OpenMP directives and use them to derive the nonconcurrency information.

This paper makes the following contributions,

- It gives a graph representation (OpenMP control flow graph) to model the control flow in parallel OpenMP programs, and a tree representation (OpenMP region tree) to model the hierarchical structure of loops and OpenMP constructs. Similar to the control flow graph and loop tree representations for sequential programs, these two representations serve as the base for further compiler analysis of parallel OpenMP programs.

- It presents an efficient static nonconcurrency analysis for OpenMP programs which are more tractable than general parallel programs. The phase partitioning algorithm has a complexity that is linear to the size of the program being analyzed for most real-life applications.

- It shows the usefulness of the nonconcurrency analysis by building a compile-time data race detection technique upon it.

The rest of the paper is organized as follows. Section 2 describes the OpenMP control flow graph and OpenMP region tree. Section 3 presents the phase partition algorithm. Section 4 gives the static nonconcurrency analysis. Section 5 uses data race detection to illustrate the use of nonconcurrency analysis. Section 6 compares related work and section 7 concludes the paper.

To be concise, we use Fortran as the base language, while our technique is not specific to Fortran.

2 OpenMP Control Flow Graph and OpenMP Region Tree

2.1 Program Model

The techniques in this paper work on OpenMP standard compliant programs[1]. Nested parallelism and orphaned directives are allowed, recognized and handled accordingly. Our techniques also recognize and use the properties of all OpenMP synchronization constructs and directives (such as **barrier**, **master**, **critical** and **ordered**).

We assume 1) all parallel regions can be active and none is serialized; 2) there is an infinite number of threads available; and 3) the exact number of threads that execute any particular parallel region is unspecified. These assumptions not only simplify the problem but also make the result of our nonconcurrency analysis independent of runtime environment. We ignore calls to OpenMP runtime lock routines, and make no attempt to recognize 'roll-your-own' synchronizations, such as busy-waiting. Knowledge of this information could add to the nonconcurrency result, but could never invalidate a nonconcurrency relationship between statements that our method finds.

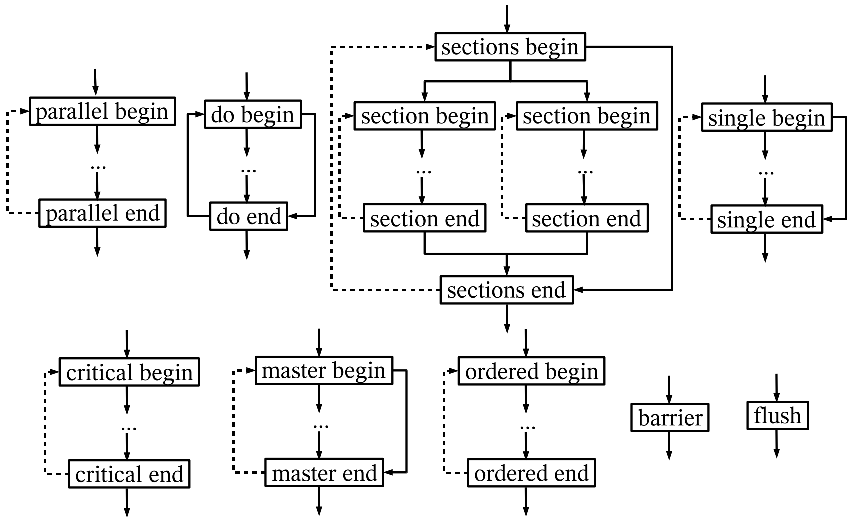

Figure 1: Directive nodes in OpenMP control flow graph (solid lines are flow edges and dotted lines are construct edges)

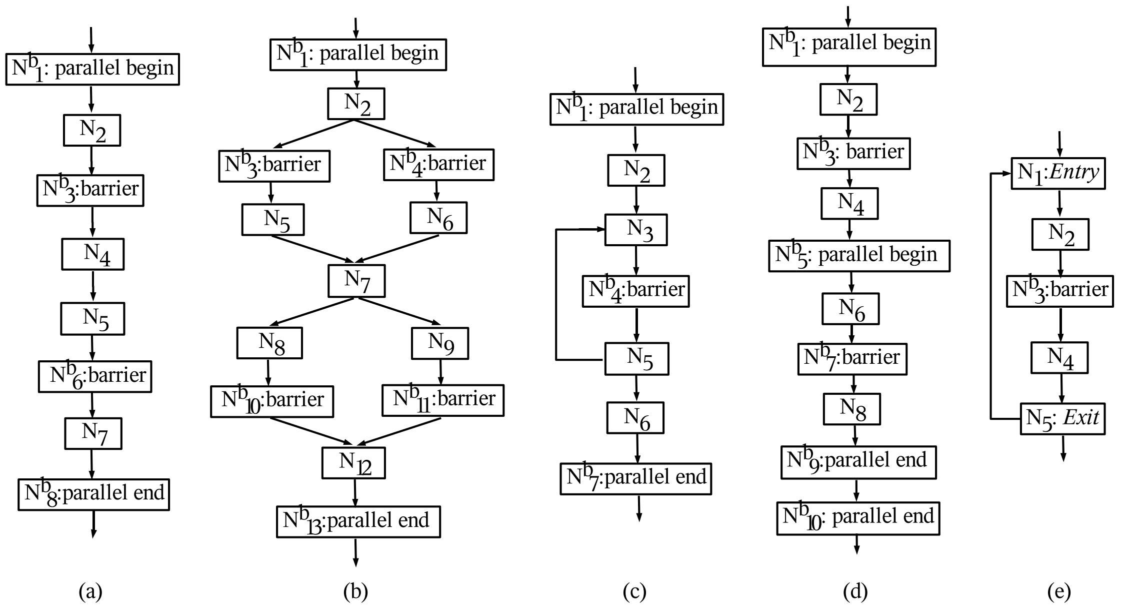

Figure 2: (a) simple phases in one parallel region. (b) phases when there is a branch (c) phases in a loop. (d) phases in nested parallel regions. (e) orphan phases

2.2 OpenMP Control Flow Graph

An OpenMP control flow graph (OMPCFG) models the transfer of control flow in a subroutine of an OpenMP program.

The statements in an OpenMP subroutine are partitioned into basic blocks and each OpenMP directive is put into an individual block. Each block becomes a node in OMPCFG. The nodes representing basic blocks are called basic nodes, and the nodes representing directive blocks are called directive nodes. A single *Entry* node and a single *Exit* node are created for an OMPCFG.

In an OMPCFG, to make compiler analysis easier, implicit barriers are made explicit[2], and each combined parallel work-sharing construct (such as **parallel do** and **parallel sections**) is separated into a **nowait** work-sharing construct nested in a parallel region. **parallel begin** directive nodes and **parallel end** directive nodes are considered as barrier nodes in the parallel region defined by the two directive nodes. Fortran specific 'WORKSHARE' construct can be converted into a set of other OpenMP constructs, therefore it is not presented directly in an OMPCFG.

An edge in OMPCFG represents a possible transfer of control flow executed by a thread. Edges between basic nodes are created in a way similar to that in sequential programs. Edges between basic nodes and directive nodes and edges between directive nodes are created according to OpenMP semantics.

Statements inside an OpenMP construct form a single-entry/single-exit region. For each OpenMP construct, an edge is created from the directive begin node to the single entry node of the region for the construct, and an edge is created from the single exit node of the region to the directive end node. Edges to and from the **barrier** and **flush** nodes are created as if they are basic nodes. An edge is created from a **sections begin** node to each binding **section begin** node. And an edge is created from each **section end** node to its binding **sections end** node. For the **do** construct, the loop control statements are not represented in the OMPCFG.

Figure 1 illustrates all the directive nodes and the corresponding edges. The construct edges (dotted lines) are explained in the next section.

2.3 OpenMP Region Tree

In OpenMP programs, we use a region tree to model both the hierarchical loop structure and the hierarchical OpenMP construct structure in a subroutine.

In OMPCFG, for each OpenMP construct (except for **do** constructs), we add an edge from the end construct directive node to the begin construct directive node. We call this edge a construct edge and represent it using a dotted line in an OMPCFG. A construct edge does not reflect any control flow. It is inserted so that an OpenMP construct forms a cycle in the OMPCFG. Therefore, the normal loop tree detection algorithm for sequential programs can be used to find both loops and OpenMP construct regions in an OMPCFG.

Because the statements in an OpenMP construct form a single-entry/single-exit region, the OpenMP constructs in a subroutine are properly nested. If we treat the whole subroutine as a **root** construct, then all the OpenMP constructs form a tree structure. The OpenMP constructs are also properly nested with loops in the subroutine. When we combine the loop tree with the OpenMP construct tree, we get the OpenMP region tree. Each node in an OpenMP region tree represents either a loop or an OpenMP construct.

For a node N in an OMPCFG, we use $ORC(N)$ to represent the immediately enclosing OpenMP construct for node N in the OpenMP region tree. Table 1 lists the notations used to represent the attributes of different types of OpenMP construct.

[2]To model the dataflow in an OpenMP program, it would be better to make all implicit flushes explicit. To be concise, we do not do so in this paper because our nonconcurrency analysis does not depend on inter-thread dataflow information.

OMPCFG	Phase	Nodes in Phase
(a)	$\langle N_1^b, N_3^b \rangle$	N_2
	$\langle N_3^b, N_6^b \rangle$	N_4, N_5
	$\langle N_6^b, N_8^b \rangle$	N_7
(b)	$\langle N_1^b, N_3^b \rangle$	N_2
	$\langle N_1^b, N_4^b \rangle$	N_2
	$\langle N_3^b, N_{10}^b \rangle$	N_5, N_7, N_8
	$\langle N_3^b, N_{11}^b \rangle$	N_5, N_7, N_9
	$\langle N_4^b, N_{10}^b \rangle$	N_6, N_7, N_8
	$\langle N_4^b, N_{11}^b \rangle$	N_6, N_7, N_9
	$\langle N_{10}^b, N_{13}^b \rangle$	N_{12}
	$\langle N_{11}^b, N_{13}^b \rangle$	N_{12}
(c)	$\langle N_1^b, N_4^b \rangle$	N_2, N_3
	$\langle N_4^b, N_4^b \rangle$	N_5, N_3
	$\langle N_4^b, N_7^b \rangle$	N_5, N_6
(d)	$\langle N_1^b, N_3^b \rangle$	N_2
	$\langle N_3^b, N_{10}^b \rangle$	N_4, N_5^b, N_6, N_7^b, N_8, N_9^b
	$\langle N_5^b, N_7^b \rangle$	N_6
	$\langle N_7^b, N_9^b \rangle$	N_8
(e)	$\langle N_3^b, N_3^b \rangle$	N_1, N_2, N_4, N_5

Table 2: Phases and nodes in each phase for each OMPCFG in Figure 2

3 Phase Partitioning

3.1 Phases in a Parallel Region

Barrier is the most frequently used synchronization method in OpenMP. Barriers can be inserted by using the BARRIER directive, and are also implied at the end of worksharing constructs or parallel constructs.

In addition, OpenMP standard requires[1]

> BARRIER directives must be encountered by all threads in a team or by none at all, and they must be encountered in the same order by all threads in a team.

The restriction the OpenMP standard imposes on the use of barriers essentially partitions the execution of a parallel region into a set of distinct, non-overlapping run-time phases. No two statement instances in two different run-time phases will ever be executed concurrently by different threads in a team. For example, the barriers in Figure 2(a) (barrier nodes are marked with a superscript `b`) put the non-barrier nodes into three phases - one phase with node N_2, another phase with node N_4 and node N_5, and yet another with node N_7. Statements in N_2 and statements in N_4 will not be executed concurrently by different threads in a team. We should also note that the restriction the OpenMP language imposes on barriers does not apply to threads in different teams.

3.2 Static Phases

In this section, we give an algorithm that computes the static phases in an OpenMP subroutine at compile-time. Our algorithm works on basic blocks instead of statements. All construct edges in OMPCFG are ignored since they do not represent any control flow. A special edge from the *Exit* node to the *Entry* node is added to help analysis of subroutines that contain orphaned OpenMP directives.

A static phase $\langle N_i^b, N_j^b \rangle$ consists of a sequence of nodes along all barrier free paths in the OMPCFG that start at one barrier node N_i^b and end at another (possibly the same) barrier node N_j^b in the same parallel region. Table 2 lists the phases for each OMPCFG in Figure 2.

Note that in Figure 2(e), node N_2 and node N_4 are in the same phase. Node N_3^b is an orphaned barrier, and there is no lexically visible parallel region in the subroutine. It is possible that the call-site of the subroutine is inside a loop, therefore there might be a barrier free path from node N_4 to node N_2 at runtime. Without interprocedural analysis, we have to assume such a loop exists. That's the reason why a special edge from the *Exit* node to the *Entry* node is inserted.

Also note that a node may belong to different static phases. For example, in Figure 2(b), node N_5 belongs to two both phase $\langle N_3^b, N_{10}^b \rangle$ and phase $\langle N_3^b, N_{11}^b \rangle$.

Each static phase has its owner parallel region, which is its immediate enclosing parallel region. A static phase is not considered as a static phase in a parallel region that is not its owner parallel region. For example, in Figure 2(d), the owner parallel region of static phase $\langle N_5^b, N_7^b \rangle$ is the inner parallel region, and it is not a static phase in the outer parallel region. For a static phase that starts and ends at orphaned barriers, its owner parallel region is **root**.

3.3 Algorithm to Compute Static Phases

The algorithm to partition an OMPCFG into phases is shown in Figure 3. In the following text, when we say 'phase', we mean 'static phase'.

We use the following notations in the algorithm:

- $phase(N_i^b, N_j^b)$

  the set of nodes that belong to phase $\langle N_i^b, N_j^b \rangle$.

- $in\_phase(N)$

  the set of phases that node N belongs to.

- $p\_start(N)$

  the set of starting barriers of phases that node N belongs to, i.e. $\{N_i^b | \langle N_i^b, N_j^b \rangle \in in\_phase(N)\}$.

- $p\_end(N)$

  the set of ending barriers of phases that node N belongs to, i.e. $\{N_j^b | \langle N_i^b, N_j^b \rangle \in in\_phase(N)\}$.

The algorithm does a forward depth-first-search, and a backward depth-first-search from each barrier node (including pseudo barrier nodes, i.e. *Entry*, *Exit*, parallel-begin, and parallel-end). During each search, if a barrier node in the same parallel region is encountered, the search does not continue with successors or predecessors of the barrier node. In a forward search from a barrier N^b, we put N^b in $p\_start(N)$ of each node N reached. In a backward search from a barrier N^b, we put N^b in $p\_end(N)$ of each node N reached. After all searches finish, for each non-barrier node N, we compute $in\_phase(N)$ as $\{\langle N_i^b, N_j^b \rangle \mid N_i^b \in p\_start(N),\ N_j^b \in p\_end(N),\ ORC(N_i^b).pregion = ORC(N_j^b).pregion\}$.

In general, the complexity of this algorithm is $O(\sum_{i=1}^{K} \sum_{j=1}^{M_i} node(i,j)\, nbar(i,j))$. Basically, if the OMPCFG for a parallel region is disconnected at each barrier node, then the OMPCFG is separated into several disconnected sub-graphs. Here, K is the number of parallel regions; M_i is the number of the disconnected sub-graphs for parallel region i; $node(i,j)$ is the number of nodes in sub-graph j of parallel region i; and $nbar(i,j)$ is the number of barrier nodes that separates sub-graph j from other sub-graphs in parallel region i. Nested-parallel regions are rarely used and each sub-graph of a parallel region contains only two barrier nodes (one starting barrier node and one ending barrier). Therefore, in most cases, the complexity of the algorithm is $O(n)$, where n is the number of nodes in OMPCFG.

4 Nonconcurrency Analysis

In this section, we describe our static nonconcurrency analysis. Given two statements in a subroutine and an OpenMP parallel region, the analysis detects at compile time whether these two statements can be executed concurrently by different threads in the team for the parallel region.

```
foreach barrier $N_i^b$ in OMPCFG
    foreach successor $N_j$ of $N_i^b$
        $forward\_mark(N_j, N_i^b)$ ;
    foreach predecessor $N_k$ of $N_i^b$
        $backward\_mark(N_k, N_i^b)$ ;

foreach non-barrier node $N$ in OMPCFG
    foreach $N_i^b$ in $p\_start(N)$
        foreach $N_j^b$ in $p\_end(N)$ that $ORC(N_i^b).pregion = ORC(N_j^b).pregion$
            $phase(N_i^b, N_j^b) := phase(N_i^b, N_j^b) \cup \{N\}$ ;
            $in\_phase(N) := in\_phase(N) \cup \{\langle N_i^b, N_j^b \rangle\}$ ;


$forward\_mark(N, N^b)$
{
    if ($N$ is a barrier node and $ORC(N).pregion = ORC(N^b).pregion$)
        return ;

    $p\_start(N) := p\_start(N) \cup \{N^b\}$ ;

    foreach successor $N_j$ of $N$
        $forward\_mark(N_j, N^b)$ ;
}


$backward\_mark(N, N^b)$
{
    if ($N$ is a barrier node and $ORC(N).pregion = ORC(N^b).pregion$)
        return ;

    $p\_end(N) := p\_end(N) \cup \{N^b\}$ ;

    foreach predecessor $N_k$ of $N$
        $backward\_mark(N_k, N^b)$ ;
}
```

Figure 3: Algorithm: phase partitioning

The algorithm in Section 3 partitions a subroutine into phases. Depending on whether the two statements belong to the same phase or not, we use two different methods to check the nonconcurrency. Because two statements are executed concurrently if and only if their basic blocks are executed concurrently, we will work on basic blocks instead of statements.

4.1 Two Nodes in Different Phases

If two nodes in a parallel region do not share any static phase, then the runtime instances of these two nodes will be in different runtime phases. Therefore these two nodes will not be executed concurrently by different threads in the team that executes the parallel region.

For example, in Figure 2(a), node N_2 and node N_4 will never be executed concurrently. However, node N_4 and node N_5 may be executed concurrently, because $N_4 \in \langle N_3^b, N_6^b \rangle$, and $N_5 \in \langle N_3^b, N_6^b \rangle$.

In Figure 2(b), node N_5 and node N_6 will never be executed concurrently. Node N_5 and node N_9 may be executed concurrently.

In Figure 2(c), node N_2 and node N_5 will never be executed concurrently. Node N_3 and node N_5 may be executed concurrently.

In Figure 2(d), node N_6 and node N_8 will never be executed concurrently by different threads in the team that executes the inner parallel region. However, these two nodes may be executed concurrently by different threads in the team that executes the outer parallel region.

In Figure 2(e), node N_2 and node N_4 may be executed concurrently.

In summary, given two nodes N_1 and N_2 whose immediate common enclosing parallel region is PR (could be **root**), if there does not exist a phase in $in\_phase(N_1) \cap in\_phase(N_2)$ whose owner parallel region is PR, then N_1 and N_2 will not be executed concurrently by different threads in the team that executes PR.

4.2 Two Nodes in the Same Phase

The semantics of OpenMP constructs also prohibits some statements within the same phase to be executed concurrently, e.g. statement 6 and statement 7 in the example at the beginning of this paper.

Given two basic blocks N_1 and N_2 (possibly the same) that $\langle N_i^b, N_j^b \rangle \in in\_phase(N_1) \cap in\_phase(N_2)$, and the owner parallel region of $\langle N_i^b, N_j^b \rangle$ is PR, the two blocks N_1 and N_2 will not be executed concurrently by different threads in a team that executes PR in the following situations.

1. **master**

   Both N_1 and N_2 are in **master** constructs that belong to PR.

   $$ORC(N_1).type = ORC(N_2).type = master, ORC(N_1).pregion = ORC(N_2).pregion = PR$$

2. **ordered**

   Both N_1 and N_2 are in **ordered** constructs in PR and are bound to the same **do** construct.

   $$ORC(N_1).type = ordered,\ ORC(N_2).type = ordered$$
   $$ORC(N_1).pregion = ORC(N_2).pregion = PR$$
   $$ORC(N_1).ordered.bound = ORC(N_2).ordered.bound$$

3. **single**

   Both N_1 and N_2 are in the same **single** construct in PR

   $$ORC(N_1) = ORC(N_2),\ ORC(N_1).type = ORC(N_2).pregion = single$$

$$ORC(N_1).pregion = ORC(N_2).pregion = PR$$

and one of the following is true.

- the **single** construct is not in any loop within the parallel region PR.
- the **single** construct is in a loop within the parallel region PR, and there is no barrier-free path from the **single** end directive node to the header of the immediately enclosing loop.
- the **single** construct is in a loop within the parallel region, and there is no barrier-free path from the header of the immediately enclosing loop to the **single begin** directive node.

OpenMP requires a **single** construct to be executed by only one thread in a team. However, it does not specify which thread. If the **single** construct is inside a loop, then two different threads may each execute one instance of the **single** construct in different iterations. If there is no barrier, then the two threads may execute the construct concurrently.

Also note that we do not check for critical sections. A critical section enforces serial execution, but does not enforce synchronization. Different instances of the statements in a critical section cannot be executed in parallel, but can be executed concurrently.

5 Application: Static Race Detection

Static nonconcurrency analysis can help many useful analyses and optimizations, such as race detection, lock/barrier removal, synchronization optimization, etc. A static nonconcurrency analysis similar to the above has been implemented in Sun Studio$^{\text{TM}}$ 9 compilers. It serves as one of the analysis engines for the OpenMP autoscoping feature, which automatically detects the data sharing attributes of variables in an OpenMP application[6]. It also serves as an engine for the static OpenMP error detection feature provided in Sun compilers. Here, we show how to build a static race detection algorithm upon the static nonconcurrency analysis.

5.1 The Method

There are two different types of races, *synchronization races* and *data races*, which are collectively called *general races* [8]. A general race happens when the order of two accesses (at least one is write) to the same memory location is not enforced by synchronizations. A data race happens when a general race happens and the access to the memory is not guarded by a critical section. A general race that is not a data race is called synchronization race. A correct OpenMP program may contain synchronization races, but is usually expected to be free of data race. For example, in a producer/consumer code, the producer and the consumer may execute asynchronously, but they should not corrupt the shared data. Many OpenMP programs are parallelized from serial codes and their behavior is usually deterministic. Such programs should be free of both synchronization races and data races.

If any two accesses to the same memory location cannot be executed concurrently, then these two accesses must be ordered and a general race is impossible. If the two accesses can be executed concurrently and the accesses are guarded by critical sections, then a synchronization race may happen while a data race is impossible. Based on the above logic and our nonconcurrency analysis, we can develop a static race detection method for OpenMP programs.

Given two statements s_1 and s_2 that access the same shared memory location (at least one of them writes to the location) and a parallel region PR, the following steps detect whether the two statements may cause a race in PR.

1. Find the basic block N_1 for s_1 and the basic block N_2 for s_2.

2. Use the method in Section 4 to check the nonconcurrency relationship between N_1 and N_2 in parallel region PR.

3. If N_1 and N_2 will not be executed concurrently, then the two statements will not cause a race in PR.

4. Otherwise, if both N_1 and N_2 are in **critical** constructs that have the same name or both are unnamed.

   $$ORC(N_1).type = critical, \; ORC(N_2).type = critical$$

   $$ORC(N_1).crit.name = ORC(N_2).crit.name$$

   then the two statement may cause a synchronization race, but will not cause a data race in PR.

5. Otherwise, the two statements may cause a data race in PR.

5.2 Example

```
      function foo (n, x, y)
      integer       n, i
      real          x(*), y(*)
      real          w, mm, m, foo

      w = 0.0

c$omp parallel private(i,mm,t), firstprivate(n),
c$omp+ shared(m,x,y), reduction(+:w)

c$omp single
      m = 0.0
c$omp end single nowait

      mm = 0.0

c$omp do
      do i = 1, n
         t = x(i)
         y(i) = t
         if (t .gt. mm) then
            w = w + t
            mm = t
         end if
      end do
c$omp end do nowait

c$omp critical
      if ( m .le. mm ) then
         m = mm
      end if
c$omp end critical

c$omp end parallel

      foo = w - m

      return
      end
```

Function foo contains a parallel region (line 8-34), whose purpose is to copy array `x()` to array `y()`, set the maximum value of all positive elements of `x()` to a scalar variable `m`, and compute the sum `w` of some elements of `x()`. Scalar `m` is a shared variable. A **single** construct (line 11-13) is used to initialize `m`. Each thread uses a private variable `mm` to store the maximum value the thread gets in the worksharing **do** loop (line 17-26). At the end of the parallel region, the shared variable `m` is updated by all threads in a critical section (line 28-32) which is used to avoid data race.

In an attempt to speed up the execution of the parallel region, two **nowait** clauses (line 13 and line 26) are inserted to remove the implicit barriers. Threads not executing the **single** can go ahead to work on the worksharing **do** without having to wait for the thread who is initializing `m`. And threads that have finished their share of the work in the **do** can continue to update the shared variable `m`, and don't have to wait for other threads.

However, the program may not deliver the expected result because of the use of these two **nowait** clauses. Our nonconcurrency analysis will find that statements 11 through 32 are all in one static phase, and statements 12 and 29, as well as statements 12 and 30 may be executed concurrently. Because statement 12 is not guarded by a critical section, either case will cause a data race and lead to a nondeterministic execution result.

Sun Studio 9 Fortran compiler will give the following warning when the above code is compiled with the parallel error checking option `-vpara`.

```
>f90 -xopenmp -vpara -xO3 -c t.f
``t.f'', line 8: Warning: inappropriate scoping
        variable 'm' may be scoped inappropriately as 'SHARED'
        . write at line 30 and write at line 12 may cause data race
```

If the **single** construct is changed to a **critical** construct, then our race detection method will find that the code may cause a synchronization race and the execution result may still be nondeterministic.

If either one of the two **nowait** clauses is not there, our nonconcurrency analysis will find that the parallel region is partitioned into two nonconcurrent phases - one contains statement 12 and the other contains statement 29 and statement 30. Therefore statement 12 will never be executed concurrently with either statement 29 or statement 30. According to the our race detection method, the code has no race conditions now.

6 Related Work

Many researchers [4][3][2][7] have proposed different methods to detect race conditions and nondeterminacy in parallel programs that use low-level event variable synchronization, such as post/wait and locks. Our technique is different from theirs because ours uses high-level semantic information exposed by OpenMP directives and constructs. Our method is simpler and more efficient for analyzing OpenMP programs. It is not clear how to represent OpenMP semantics using event variables. Nevertheless, since our method does not handle OpenMP lock API calls, their techniques can be incorporated into our method to refine analysis results.

Jeremisassen and Eggers [5] present a compile-time nonconcurrency analysis using barriers. Their method assumes the SPMD model and is similar to our method in Section 4.1 as it also divides the program into a set of phases separated by barriers. They assume a general SPMD model that is not OpenMP specific. Therefore they cannot take advantage of restrictions that OpenMP has on the use of barriers. For example, their method will say node N_5 and node N_6 in Figure 2(b) may be executed concurrently while ours does not. Their method is purely based on barriers and does not detect nonconcurrency within one phase.

In [9], Satoh et. al. describe a 'parallel flow graph' that is similar to our OMPCFG. They connect flush operations with special edges that present the ordering constraints between the flushes. They do not have construct edges as they do not build a hierarchical structure like our OpenMP region tree. The two representations are different because they serve different purposes. Theirs is more data flow oriented, while ours is more control flow oriented. It is possible to combine these two graphs together.

7 Conclusion

We have presented a method for compile-time detection of nonconcurrency information in OpenMP programs. The analysis uses the semantics of OpenMP directives and takes advantage of the fact that standard compliant OpenMP programs are well-structured. The analysis has a complexity that is linear to the size of the program in most applications, and can handle nested parallelism and orphaned OpenMP constructs. The OpenMP control flow graph and the OpenMP region tree developed in this work can be used for other compiler analyses/optimizations of OpenMP programs as well. We have also demonstrated the use of the nonconcurrency information by building a compile-time race detection algorithm upon it.

Acknowledgement The author would like to thank Nawal Copty and Eric Duncan for reviewing the paper and providing insightful comments.

References

[1] OpenMP Fortran Application Program Interface, Version 2.0. *http://www.openmp.org/specs*, November 2000.

[2] Vasanth Balasundaram and Ken Kennedy. Compile-time detection of race conditions in a parallel program. In *Conference Proceedings, 1989 International Conference on Supercomputing*, pages 175–185, Crete, Greece, June 5–9, 1989. ACM SIGARCH.

[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 and Practice of Parallel Programming*, pages 21–30, Seattle WA, March 1990.

[4] P. Emrath and D. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24(1), pages 89–99, New York, NY, January 1989. ACM Press.

[5] T. E. Jeremiassen and S. J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 171–180, August 1994.

[6] Yuan Lin, Christian Terboven, Dieter an Mey, and Nawal Copty. Automatic scoping of variables in parallel regions of an openmp program. In *Proceedings of the 2004 Workshop on OpenMP Applications and Tools*, Houston, TX, May 2004.

[7] Robert H. B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In Kang G. Shin, editor, *Proceedings of the 1992 International Conference on Parallel Processing. Volume 2: Software*, pages 242–246, Ann Arbor, MI, August 1993. CRC Press.

[8] Robert H. B. Netzer and Barton P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[9] Shigehisa Satoh, Kazuhiro Kusano, and Mitsuhisa Sato. Compiler optimization techniques for openMP programs. *Scientific Programming*, 9(2-3):131–142, 2001.

[10] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.