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

A New Framework for Analysis and Optimization of Shared Memory Parallel Programs

Vugranam C. Sreedhar†
Yuan Zhang
Guang R. Gao

CAPSL Technical Memo 63
July 18, 2005

Copyright © 2005 CAPSL at the University of Delaware

†IBM T.J.Waston Research Center, Hawthorne, NY 10532. Email: vugranam@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

In this paper we present a new framework for analysis and optimization of shared memory parallel programs. Our framework is based on two key concepts: (1) concurrency relation and (2) isolation semantics. Two statements are said to be concurrent if there is an execution in which two threads can execute the two statements simultaneously. Isolation semantics essentially treats critical sections as being isolated or atomic. The main contributions of this paper are as follows:

- We present a simple algorithm for computing concurrency relation among two or more statements in a parallel program that includes `cobegin/coend`, `parallel for`, `barrier`, `post/wait`, and `critical` sections.
- We propose a new framework for solving data flow problems by *reifying* two key concepts: *concurrency relation* and *isolation semantics* for critical sections. As an example we solve the pointer analysis problem using our framework To the best of our knowledge ours is the first work that directly incorporates isolation semantics for solving data flow analysis.
- We formulate the lock assignment problem for assigning locks to critical sections. For a special class of non-interfering critical sections, we will show that the optimal lock assignment problem can be reduced to a graph coloring problem. In general, we conjecture that the optimal lock assignment problem is NP-hard. We present a simple algorithm that tries to assign the minimal set of locks to critical sections without reducing the fine-grained parallelism. To the best of our knowledge, ours is the first work that formulates and solves the problem of lock assignment for critical sections.
- To study the feasibility and validity of our approach we implemented lock assignment and pointer analysis using the Omni [33] OpenMP Compiler. We present some preliminary experimental results of our implementation.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contribution . . . 3

2 Programming Model **4**

3 Concurrency Analysis **5**
- 3.1 Parallel Control Flow Graph . . . 5
- 3.2 Barriers and Epochs . . . 7
- 3.3 Concurrency Relation . . . 8
- 3.4 Discussion . . . 9

4 Pointer Analysis with Isolation Semantics **9**
- 4.1 Intraprocedural Analysis . . . 10
- 4.2 Discussion . . . 12

5 Lock Assignment **13**
- 5.1 Non-interference Critical Section . . . 13
- 5.2 General Critical Sections . . . 14
- 5.3 Discussion . . . 15

6 Synchronization Anamolies **16**

7 Experimental Results **17**

8 Related Work **20**

9 Conclusions **22**

A Parallel Region Tree **27**

List of Figures

1 (a) Example program (b) PCFG (c) Concurrency graph (d) Concurrency graph with only critical sections, (e) Possible values that can be printed out . . . 2
2 PCFG Construction . . . 6
3 (a) Example programs (b) PCFG (c) Region tree (d) Concurrency graph . . . 7
4 Example of Epochs . . . 8
5 Example programs with and with out critical sections for pointer analysis . . . 10
6 An example of (a) Non-interfering concurrency graph and (b) General concurrency graph. In the figure [ ] denotes lock assignment and { } denote location set. . . . 13
7 Location constrain graph and suboptimal lock assignment . . . 14
8 Examples illustrating arrier and post/wait anomalies . . . 17
9 Omni OpenMP Compiler and Runtime System Infrastructure . . . 18
10 Lock assignment performance for *transfb_c_2* . . . 20

1 Introduction

OpenMP [28] is an open standard for writing portable and scalable shared memory parallel applications. Analysis and optimization of such shared memory parallel programs is challenging. For instance, OpenMP programs often use a single global lock for all critical sections. Although the global lock for critical sections provides a simpler programming model, it has severe performance penalty. One of the key problem that we will solve in this paper is the *lock assignment problem*: Given a set of critical sections, find the minimum number of locks that are needed to control the critical sections without reducing parallelism among critical sections. In order to solve the lock assignment problem, we require an understanding of data interferences among critical sections. Now in the presence of concurrency, computing data interferences among critical sections is difficult and requires data flow analysis. The second key problem that we will address in this paper is data flow analysis for multithreaded programs. We propose a new framework for solving data flow problems by *reifying* two key concepts: *concurrency relation* and *isolation semantics* for critical sections. We show how to extend the classical sequential "meet-over-all-paths" data flow analysis to parallel programs by reifying these two concepts. We illustrate our parallel program analysis framework for pointer analysis and use the result for solving the problem of lock assignment for critical sections.

1.1 Motivation

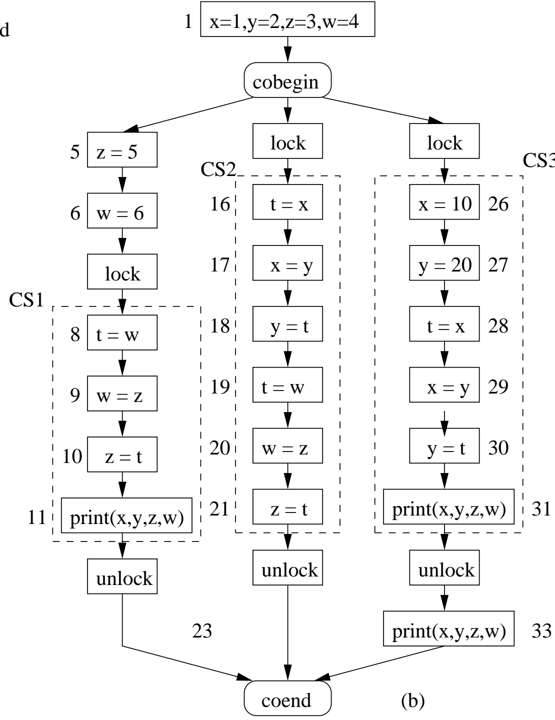
Consider the example parallel program shown in Figure 1. We use OpenMP/HPF like notation for expressing parallelism and synchronization. The program begins execution as a single (parent) thread. When a parallel region (that is, `cobegin`) is encountered, the parent (master) thread generates a *team of threads* to execute the enclosed code `sections`. We will assume that each `section` is executed by a single thread. A `critical` section in the code enforces the mutual exclusion among multiple threads. It specifies a "structured block" that can be executed exclusively by one thread at any time. When the team of threads complete, they synchronize and terminate at `coend`, leaving only the parent thread to proceed. In our framework we use two key concepts: *concurrency relation* and *isolation semantics*.

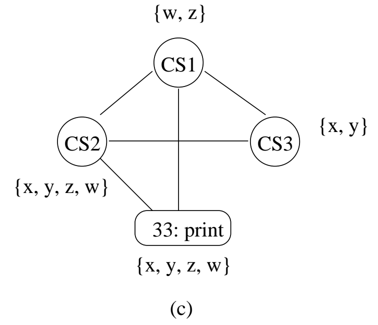
- Concurrency relation will allow us to determine when two or more statements are potentially concurrent. In our framework we do not introduce any explicit ordering among critical sections in different parallel sections. Figure 1(b) illustrates the parallel control flow graph (PCFG) for the example program shown in Figure 1(a). For the example program we can use simple graph reachability to determine concurrency relation. Consider the two statements 5 and 33, since the two statements are in two different parallel sections, and there is no path from 5 to 33 or from 33 to 5, we consider them to be concurrent.[1] Now consider the statements 8 and 18, the two statements are also concurrent, and since they are in critical sections they are also mutually exclusive. For now let us assume that all critical sections are controlled by the same "global lock". Notice that two

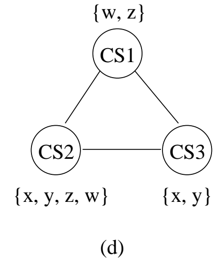
[1] Barrier and post/wait constructs can complicate concurrency relation.

```
0:  main() {
1:    int x=1, y=2, z=3, w=4; //shared
2:    cobegin
3:    section:
4:        int t;   // private
5:        z = 5;
6:        w = 6;
7:        critical {  // CS1
8:            t = w;
9:            w = z;
10:           z = t;
11:           print(x, y, z, w);
12:       }
13:   section:
14:       int t;   // private
15:       critical {  // CS2
16:           t = x;
17:           x = y;
18:           y = t;
19:           t = w;
20:           w = z;
21:           z = t;
22:       }
23:   section:
24:       int t;  // private
25:       critical { // CS3
26:           x = 10;
27:           y = 20;
28:           t = x;
29:           x = y;
30:           y = t;
31:           print(x, y, z, w);
32:       }
33:       print(x, y, z, w);
35:   coend
35: }
```

(a)







	x	y	z	w
11: print	{1, 2, 10, 20}	{1, 2, 10, 20}	{3, 4, 5, 6}	{3, 4, 5, 6}
31: print	{20}	{10}	{3, 4, 5, 6}	{3, 4, 5, 6}
33: print	{2, 10, 20}	{1, 10, 20}	{3, 4, 5, 6}	{3, 4, 5, 6}

(e)

Figure 1: (a) Example program (b) PCFG (c) Concurrency graph (d) Concurrency graph with only critical sections, (e) Possible values that can be printed out

statements are mutually exclusive only if they are also concurrent. Figure 1(c) illustrates the concurrency relation, represented as a *concurrency graph*, for the example program in which we have collapsed all statements in a critical section into one node.

- The second key concept that we will use in our framework is the *atomicity* of critical sections: *the result of an execution of a critical section is the same as if the execution happens as a single, indivisible unit with respect to all activities of other atomic operations.* Therefore critical sections have a transaction-like semantics [12] in which all other (concurrent) threads either observe all of the execution of the critical section or none of it. We will call the resulting semantics for critical sections as *isolation semantics*.

Data Flow Analysis: To illustrate how we use concurrency information and isolation semantics during data flow analysis, consider the reaching definition data flow problem (we discuss pointer analysis problem later in the paper). The definition `z = 5` at line 5 can reach the `print` statement at line 33, and that is because statement 5 and 33 are concurrent. Now consider the

statements inside the critical section `CS3`. The only definitions of `x` and `y` that can reach the `print` statement at line 31 are those that are defined inside the critical section at 26 and 27. This is because of isolation semantics of the critical section. Figure 1(e) illustrates all possible values of `x`, `y`, `z`, and `w` that can be printed at the corresponding print statements. Therefore the reaching definition problem consists of propagating definition points over PCFG and at each program point we include the reaching definition information from concurrent statements, and we handle data flow information inside a critical section in isolation (see Section 4 for details).

Lock Assignment Next we briefly show how to use concurrency information for lock assignment. Once again consider the three critical sections shown in Figure 1. If one uses a single *global lock* for all three critical sections then we unnecessarily reduce parallelism between the two critical sections $CS1$ and $CS3$. That is because there is no data that is common between these two critical sections, and hence can be executed concurrently and without mutual exclusion. Figure 1(d) shows the concurrency graph with only critical sections. Using our lock assignment algorithm we assign the locks to critical section as follows: $CS1 = \{0\}$, $CS2 = \{0, 1\}$, and $CS3 = \{1\}$. Notice that we have assigned a set of locks to $CS2$, we will discuss the semantics such lock sets later in the paper. In this paper we will formulate lock assignment problem and present an algorithm that answers the following question: what is the minimum number of locks that can be assigned to critical sections without reducing parallelism?

1.2 Contribution

To summarize the main contribution of this paper are as follows:

- We present a simple technique for computing concurrency relation among two or more statements in a parallel program that includes `cobegin/coend`, `parallel for`, `barrier`, `post/wait`, and `critical` sections. We use the result of our concurrency analysis for all further analysis and optimization, including pointer analysis and lock assignment.

- We propose a new framework for solving data flow problems by *reifying* two key concepts: *concurrency relation* and *isolation semantics* for critical sections. We show how to extend the classical sequential "meet-over-all-paths" data flow analysis to parallel programs by reifying these two concepts. As an example we solve the pointer analysis problem using our framework. To the best of our knowledge ours is the first work that directly incorporates isolation semantics for solving data flow problems.

- We formulate the lock assignment problem for assigning locks to critical sections. For a special class of non-interfering critical sections, we will show that optimal lock assignment problem can be reduced to a graph coloring problem. In general, we conjecture that optimal lock assignment problem is NP-hard. We present a simple algorithm that tries to assign the minimal set of locks to critical sections without reducing the fine-grained parallelism. To the best of our knowledge, ours is the first work that formulates and solves the problem of lock assignment for critical sections.

- To study the feasibility and validity of our approach we implemented lock assignment and pointer analysis using the Omni [33] OpenMP Compiler. We present some preliminary experimental results of our implementation.

Organization The rest of this paper is organized as follows. In Section 2 we introduce a simple language called μSMP for the purpose of illustrating our framework. In that section we also briefly discuss isolation semantics for critical section. Concurrency Analysis, pointer analysis and lock assignment are presented in Section 3, Section 4 and section 5, respectively. In Section 6 we discuss synchronization anomalies that can occur due to the incorrect placement of barriers and post/wait constructs. In Section 7 we present empirical results using OpenMP model to validate our approach. In Section 8 we discuss related work and conclude in Section 9.

2 Programming Model

In this section we introduce a simple language μSMP for the purpose of illustrating our framework. A μSMP program is a structured programs and does not contain gotos, breaks, and continue statements. The language μSMP follows a nested fork-join execution model. The program begins execution as a single thread called the *parent thread* (also called the *master thread*). When a parallel region is encountered, the parent (master) thread generates a team of threads (child threads) to execute the enclosed code sections. When they complete, they synchronize and terminate, leaving only the parent thread to proceed.

Parallelism is expressed using two parallel constructs: `parallel for` and `cobegin/coend`. When control reaches a `parallel for` construct, all iterations of the loop body are started and proceed concurrently and asynchronously, each by one thread. A `cobegin/coend` parallel region consists of a set of sections, and each `section` can itself have other `cobegin/coend` regions or `parallel for` regions. Each `section` in a cobegin/coend region is executed concurrently and asynchronously with other sections in the region. To simplify the presentation, in μSMP we restrict the class of programs that support only structured control flows for `parallel for` and `cobegin/coend` — one cannot jump in and out of these parallel constructs arbitrarily.

There are three synchronization constructs in μSMP: `barrier`, `post/wait`, and `critical` sections. The `barrier` construct enforces a coarse-grained, explicit ordering among a team of threads. When a thread encounters a `barrier`, it waits until the whole team of threads reach the same (barrier) point. After that, each thread begins executing the code (after the barrier) concurrently. The barrier semantics requires that either all threads in a team or none of them executes the barrier. A `barrier` *binds* to the closest enclosing `cobegin/coend` or `parallel for` region. One or more barriers that bind to the same parallel region are considered to be the same barrier (see Section 6). We also assume that the end of a parallel region contains an implicit barrier in which all members of the region team terminate, except for the master thread, which continues with the execution. The `post/wait` constructs can be used to implement the producer/consumer synchronization. When a consumer thread executes a `wait`, it waits until the corresponding producer thread executes a `post`.

A `critical` section enforces the mutual exclusion among multiple threads. It specifies a "structured block" that can be executed exclusively by one thread at any time. In this paper we restrict that critical sections cannot contain any parallel and synchronization constructs. We also assume that all critical sections are controlled by a global lock, and support *atomicity*: the result of an execution of a critical section is the same as if the execution happens as a single, indivisible unit with respect to all activities of other atomic operations. An operational semantics of a critical section execution can be described as follows. First, the executing thread needs to acquire the lock. The successful acquisition of a lock will begin the execution of the critical section, making a copy of the shared data it needs to access into the private space of the invoking thread (referred to as *copy-in*). Then, it starts the operations in the structured block and all memory accesses are made to the private copy of the shared data. Finally, when the execution of the code is finished, the private copy of the shared data will be copied back to the shared address space (referred to as *copy-out*). Upon completion of the write back operations, it releases the lock and the execution of the critical section terminates.

Note that a critical section has a transaction-like *isolation* semantics [12]: all other concurrent threads either observe all of the execution of the critical section or none of it. In other words, the execution of a critical section will not affect or be affected by any other concurrent accesses in any other atomic section instances. We will rely on this isolation semantics for all of our data flow analysis and optimization. In the context of this paper, we assume that memory coherence property is observed on the shared memory: i.e. reads and writes from different threads to the same memory location (including those incurred by copy-in and copy-out actions in the critical section execution) should appear in a total order. Based on memory coherence, a sequential consistent (SC) memory model [23] or its variations (SC-derived memory models: e.g. weak consistency, release consistency, etc. [15, 7, 13]) can be implemented.

3 Concurrency Analysis

Concurrency analysis is a technique for determining whether two or more statements in a multi-threaded program can potentially be executed either concurrently or mutually exclusively. Synchronization constraints and mutual exclusion constraints among statements make the problem more difficult. In general, precise interprocedural concurrency analysis in the presence of synchronization constraints is undecidable [34], and a precise intraprocedural concurrency analysis is NP-hard [41]

3.1 Parallel Control Flow Graph

The set of statements in a μSMP program and the control flow among them form a graph, called the *parallel control flow graph* (PCFG). Since a μSMP program is a structured program, the corresponding PCFG is a structured graph. Figure 2 illustrates how to construct PCFG for each program construct in μSMP language. We insert control flow edges from a *cobegin* node to the first statement node of each of the parallel sections of the corresponding parallel region,

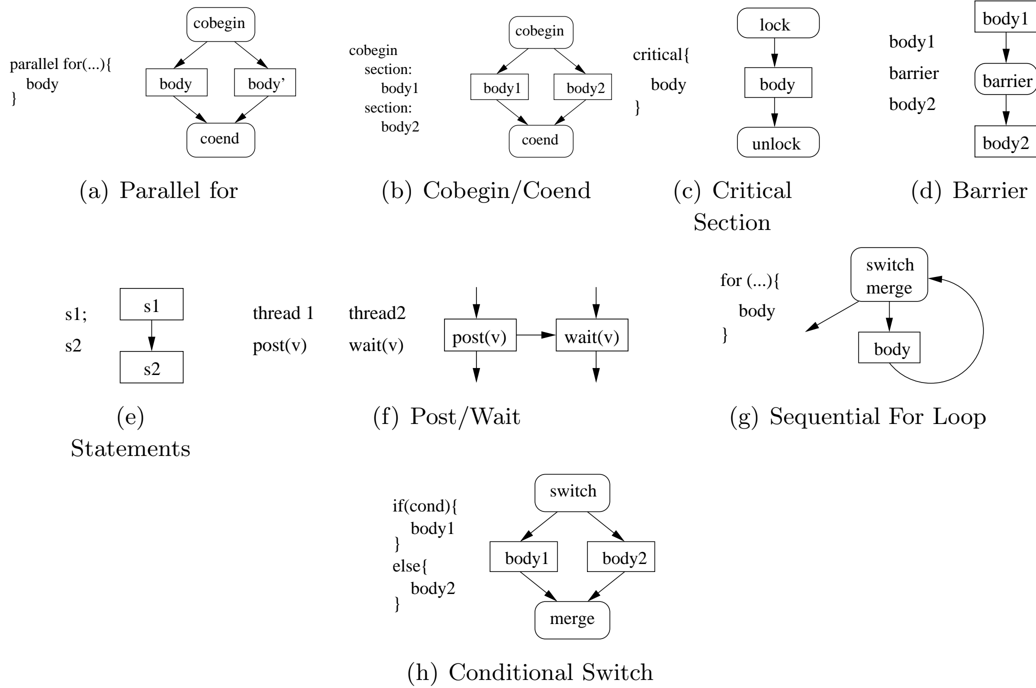


Figure 2: PCFG Construction

and we also insert control flow edges from each of the last statement node in the corresponding parallel section to the *coend* node. For `parallel for` we clone the body of the `parallel for` once and treat the body and its clone as two parallel sections of a `cobegin/coend` parallel region. Therefore we represent `parallel for` as in `cobegin/coend` parallel region. We interpret the body and its clone as being two different iterations of the parallel loop, and for our analysis purpose we do not care what those two iterations are. We insert a control flow edge from a `post` node to a `wait` node that are parameterized with the same name. We also insert a control flow edge from a statement node to another statement node if the corresponding statements follow one another. We treat barriers as any other sequential statement. We represent sequential loops, conditional, and simple statements as is traditionally done in the sequential programs. Figure 3 illustrates an example program and its PCFG.

Since μSMP contains only structured constructs, we can organize the nodes in a PCFG into a set of nested regions and represent them as a tree. In this paper we are only interested in the following three kinds of regions (1) parallel region, (2) section region, and (3) critical region. We will assume that the outer most region in a function is a `cobegin/cobegin` region with a single `section`. Appendix gives a simple algorithm to construct a region tree. Figure 3(c) gives region tree for the PCFG shown in Figure 3(a).

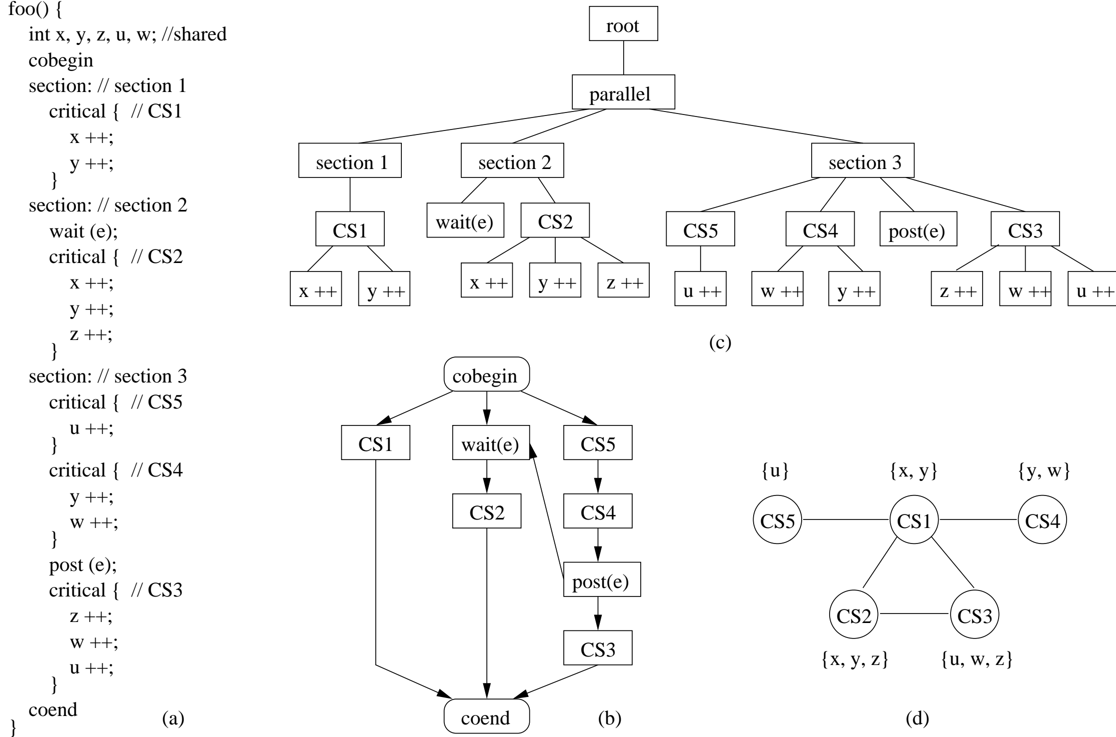


Figure 3: (a) Example programs (b) PCFG (c) Region tree (d) Concurrency graph

3.2 Barriers and Epochs

We introduce the notion of *epochs* that will help simplify concurrency analysis in the presence of barriers. The barrier semantics requires that either all threads in a team executes a barrier or none of them executes the barrier. In a μSMP program the set of barriers in a parallel region divide the parallel region into a set of "epochs" or "phases", and each epoch in the parallel region will be executed by all members of the team that belong to the same parallel region. Consider the example program and its PCFG shown in Figure 4. Recall that we clone the body of `parallel for`. The six barriers divides the parallel region in to three epochs: epoch 1 is made of `S1`, `S2`, `S1'`, and `S2'`, epoch 2 is made of `S3` and `S3'` and epoch 3 is made of statement `S4` and `S4'`. In Section 6 we give a barrier analysis technique for computing epochs. When computing the epochs for a region we consider the inner nested parallel and critical region to be like a single non-barrier statement. We also assume that there is an implicit barrier at the end of a parallel region. Also, if a parallel region is inside a sequential loop, the two iterations of the sequential loop will induce two different epoch sets of the parallel region. In μSMP one can easily construct programs in which barriers are not correctly placed. Our epoch analysis, described in Section 6, can also identify incorrectly placed barrier and post/wait synchronization. For now let us assume programs are correct with respect to barrier semantics. Notice that if two statements belong to the different epochs, they cannot be concurrent.

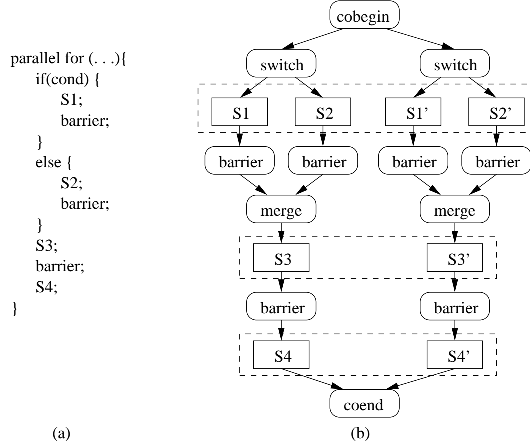


Figure 4: Example of Epochs

3.3 Concurrency Relation

Two statements s_1 and s_2 in a program P is said to be concurrent if there *exists an execution* of P such that there are two threads T_1 and T_2 than can either execute s_1 and s_2 simultaneously or mutually exclusively. Recall that in our PCFG we do not introduce explicit control flow edges between `critical` sections in different parallel regions. Novillo et al., for instance, use such explicit synchronization edges across critical section [31]. Such explicit control edges between `critical` sections can miss some concurrency relation. In our work we use an auxiliary representation, called the *concurrency relation*, to capture the concurrency information. The corresponding concurrency graph consists of a set of nodes representing statements and undirected edges among nodes representing concurrency relation among the statements. There is an edge from statement node s to another statement node t if the corresponding statements s and t can either be executed simultaneously or mutually exclusively.

Proposition 3.1 *Let s and t be any two statements in a PCFG. We say that s and t are* concurrent*, denoted as $Conc(s,t)$ if: (1) s and t belong to two different parallel sections of a parallel region, (2) s and t are in the same epoch of the parallel region, and (3) there is no path from s to t or from t to s.*

The first condition is straight forward — if the two statements s and t are in the same parallel section then they cannot be concurrent. The second condition is needed because two epochs of a parallel region cannot be executed concurrently, and last condition is needed because of `post/wait` control flow edges. It is important to remember that our concurrency relation obtained using the above proposition is conservative: $Conc(s,t)$ implies that there exists an execution of the program so that two threads can execute s and t concurrently or mutually exclusively. It is possible, due to resource constraint or scheduling constraints, that s and t can be ordered (see also next section). It is important to note that $Conc(s,t)$ is not transitive but is symmetric. Sometimes we will use the notation $Conc(s)$ to denote the set of all statements

(nodes) that are concurrent with s. Note that $r \in Conc(s)$ if and only if $Conc(r, s)$.

3.4 Discussion

So far we have discussed intraprocedural concurrency analysis. Now consider the following simple program.

```
bar() {                        foo(int *x, int *y) {
   cobegin                         S1 ;
     section:                      S2 ;
       foo(p, q) ;             }
     section:
       foo(a, a) ;
   coend
}
```

Our concurrency analysis for `foo()` based on Proposition 3.1 will conclude that `S1` and `S2` are not concurrent. The conclusion is actually not true in the context of interprocedural analysis, because `foo()` is concurrently invoked twice from the parallel region, and `S1` of first invocation is concurrent with `S2` of the second invocation. In general, computing interprocedural context sensitive concurrency information is undecidable [34]. We can handle concurrency relation in such a situation in two ways: (1) inline all functions at the expense of exponential blowup (assuming recursive functions are not allowed) and apply Proposition 3.1, or (2) conservatively assume two statements $s1$ and $s2$ in a function f are concurrent if f can be invoked either directly or indirectly from a parallel region.

4 Pointer Analysis with Isolation Semantics

In Section 1 we briefly discussed the reaching definition problem. In this section we will discuss the pointer analysis problem and show how to incorporate concurrency relation and isolation semantics into the corresponding data flow equations. Pointer analysis consists of computing points-to information at each program point. Given two program variables p and q, we say that p *points-to* q, denoted as $p \rightarrow q$, if p can contain the address of q. For heap allocated objects, we say that p can point-to an object O if p can contain the address of O. The address of O is not known at compile-time and one typically assigns compile-time address to heap objects. A points-to graph consists of a set of nodes denoting either program variables, objects, or object fields and edges denoting the points-to relations. We need pointer information to improve the precision of lock assignment.

Pointer analysis for parallel programs introduce additional complexity due to interference among multiple threads. Rugina and Rinard [36] proposed a interprocedural, flow-sensitive,

```
int *p, *q, a, b,c ;        int *p, *q, a, b, c ;
void Foo()  {               void Foo()  {
1:   cobegin                1:   cobegin
2:     section:             2:     section:
3:        p = &a ;          3:       critical{  // CS1
4:        q = &c ;          4:          p = &a ;
5:        p = q ;           5:          q = &c ;
6:     section:             6:          p = q ;
7:        p = &b ;          7:       }
8:        p = &d ;          8:     section:
9:   coend ;                9        critical {  // CS2
}                           10:        p = &b ;
                            11:      }
                            12:      p = &d ;
                            13:   coend ;
                            }
    (a)                         (b)
```

Figure 5: Example programs with and with out critical sections for pointer analysis

and context-sensitive pointer analysis algorithm for structured parallel programs, which generates a points-to graph at each program point, and takes the interference information into account when computing the effect of each statement on the points-to graph for the next program point. Rugina and Rinard, in their formalism, ignore synchronization constructs such as locks, semaphores and critical sections, and so we cannot directly apply their technique for our purpose.

4.1 Intraprocedural Analysis

In sequential programs the points-to information at node n depends only on what is *generated* and *killed* at the node and the *incoming* points-to information at node n. In the context of parallel programs, we also have to consider the points-to information generated by nodes $m \in Conc(n)$. We use the result of concurrency analysis to handle the points-to information from concurrent nodes. To simplify the presentation we only consider the following kinds of statements that affect the pointer analysis and present transfer function for each kind: (1) $p = \&q$, (2) $p = q$, (3) $p = *q$, and (4) $*p = q$. Let $PointsTo(p) = \{r | p \rightarrow r\}$, the points-to set of p at a program point. Consider the example shown in Figure 5(a). Since statement 7 is concurrent with statements 3,4, and 5, $p \rightarrow b$ cannot be killed by any of these statement. On the other hand the pointer assignment, `p = q` at statement 5 can kill $p \rightarrow a$ that is generated at statement 3. Therefore, the points-to set after statement 5 is $\{p \rightarrow c, p \rightarrow b\}$.

Now let us slightly modify the program and introduce critical sections, which shown in Figure 5(b). In this case the pointer $p \rightarrow b$ and $p \rightarrow d$ can be killed within the critical

section CS1 defined at 3, but not outside of the critical section. This is because of the *isolation semantics* of the critical section. Therefore the points-to set after statement 5 and within critical section is $\{p \rightarrow c\}$. On the other hand the points-to set after the critical section is $\{p \rightarrow c, p \rightarrow b, p \rightarrow d\}$. Similarly, the points-to after critical section CS2 is $\{p \rightarrow c, p \rightarrow b\}$. Recall that the memory model for critical sections follows isolation semantics (see Section 2). We incorporate the isolation semantics in our data flow analysis. The classical Kildall data flow analysis of sequential programs consists of two equations [20].

$$Out(s) = (In(s) - Kill(s)) \cup Gen(s) \tag{1}$$

$$In(s) = \bigcup_{r \in Pred(s)} Out(r) \tag{2}$$

where $Pred(s)$ denote the set of predecessor statements in the CFG. Let $Conc(s)$ be the set of statements that are concurrent with s. Let CS denote the set of statements in a critical section (that is, a critical region). First let us define the following set.

$$F(s) = \bigcup_{r \in Conc(s) \ and \ r \neq CS} Gen(r) \cup \bigcup_{r \in Conc(s) \ and \ r = CS} Out(r)$$

$F(s)$ is what we will add to classical data flow equation defined above. The first part of of $F(s)$ deals with concurrency relation for non critical sections and the second part deals with critical sections.

Case 1: Inside a Critical Section Let s be a statement inside a critical section.

$$Out(s) = (In(s) - Kill(s)) \cup Gen(s) \tag{3}$$

$$In(s) = \bigcup_{r \in Pred(s)} Out(r) \tag{4}$$

If s is the first statement inside the critical section, then $In(s)$ is the points-to set at the beginning of the critical section. Notice once again that we assume isolation semantics for critical section. Therefore none of the concurrent statements can affect any pointer information *within* a critical section.

Case 2: Critical Section With respect to statements outside the critical section, we treat critical section as being atomic, and again due to isolation semantics of our memory model. Let s be a critical section. The points-to set at the end of the critical section is essentially the output set of the last statement inside the critical section. The data flow equations for a critical section s are as follows:.

$$Out(s) = (In(s) - Kill(s)) \cup Gen(s) \tag{5}$$

$$In(s) = \bigcup_{r \in Pred(s)} Out(r) \cup F(s) \tag{6}$$

Case 3: Non-Critical Section The data flow equations for a statement s that is not in a critical section are as follows:

$$Out(s) = (In(s) - Kill(s)) \cup Gen(s) \cup F(s) \quad (7)$$

$$In(s) = \bigcup_{r \in Pred(s)} Out(r) \cup F(s) \quad (8)$$

Notice that in the above three cases if $F(s)$ is empty we get the classical sequential data flow equations. The above data flow equations can be applied to any monotone data flow problems. For pointer analysis we define the Gen and the $Kill$ for the following statements. Let $K(s) = PointsTo(p)$ prior to the execution (analysis) of s.

- [$\mathbf{s} : \mathbf{p} = \&\mathbf{q}$] The $Gen(s) = (p \rightarrow q)$, and $Kill(s) = K(s)$.
- [$\mathbf{s} : \mathbf{p} = \mathbf{q}$] Let $Q = PointsTo(q)$ then $Gen(s) = \{(p \rightarrow t) | t \in Q\}$ and the $Kill(s) = K(s)$.
- [$\mathbf{s} : \mathbf{p} = *\mathbf{q}$] Let $Q = PointsTo(q)$ and $T = \bigcup_{t \in Q} PointsTo(t)$. $Gen(s) = \{(p \rightarrow u) | u \in T\}$ and $Kill(s) = K(s)$.
- [$\mathbf{s} : *\mathbf{p} = \mathbf{q}$] Let $P = PointsTo(p)$, $T = \bigcup_{t \in P} PointsTo(t)$, and $Q = PointsTo(q)$. $Gen(s) = \{(t \rightarrow q) | t \in T \; and \; q \in Q\}$ and $Kill(s) = \emptyset$.

To summarize, our equations for pointer analysis rely on two important properties: (1) isolation semantics for critical sections and (2) concurrency information. Our equations are a simple extension to the classical Kildall's equations for sequential programs. The above equations are general data flow equations that can be applied to other data flow problems.

4.2 Discussion

In this paper we have made a simplified assumption that computing concurrency relation is independent of data flow in the program. This assumption is typically not valid in some parallel programming model. For instance, we use name-based mechanism to link `post(e)` and `wait(e)`. Often if the parameter to `post/wait` is an object, we may need to perform alias analysis to capture the control flow from `post(a)` to `wait(b)`. The interaction between concurrency relation and aliasing is more prevalent in object-oriented programming model, such as Java. We will not go into the details of such interaction in this paper.

We use the result of pointer analysis to identify the set of locations that can be accessed inside a critical section. We define *location set* $LS(s)$ for a statement s as a set of locations that can be either read or written by a statement. We simply follow Ghiya and Hendren approach for computing the location sets for each statement [14]. For critical section, we take the union of location set of each statement inside the critical section, and denote the location set of a critical section cs as $LS(cs)$. We will use the location set during lock assignment to determine the data interference among critical sections.

5 Lock Assignment

OpenMP requires that only one critical section is allowed to execute at one time anywhere in the program, and therefore the corresponding critical section lock is a *global lock*. A single global lock can drastically reduce the parallelism. Two concurrent critical sections that do not access the same set of locations can be assigned two different locks. In this section we present a solution to lock assignment. We show that for a class of programs in which there are no data interference among critical section, the problem of lock assignment can be reduced to graph coloring problem, and so the optimal solution for such programs is NP-hard. For general programs where there are data interference among critical section, we conjecture that the optimal solution is also NP-hard. We present a simple heuristic method for assigning locks to critical sections.

5.1 Non-interference Critical Section

Consider a class of programs P_{ni} that contain only non-interfering critical sections. Two concurrent critical sections cs_1 and cs_2 are said to be non-interfering if they do not access the same set of locations, that is, $LS(cs_1) \cap LS(cs_2) = \emptyset$. Note that if two critical sections are not concurrent in P_{ni} then they could have overlapping location sets. Ideally we should assign two different locks to any two concurrent critical sections in P_{ni}. Otherwise we are unnecessarily constraining the parallelism between those two concurrent critical sections.

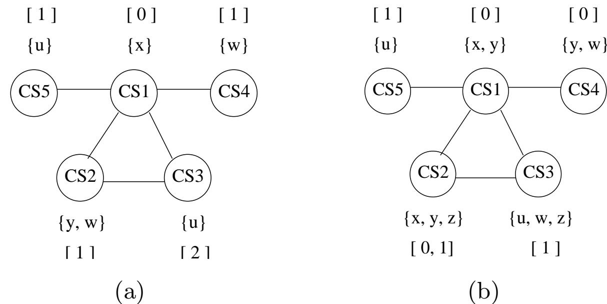


Figure 6: An example of (a) Non-interfering concurrency graph and (b) General concurrency graph. In the figure [ ] denotes lock assignment and { } denote location set.

We call the corresponding concurrency graphs of such a class of non-interfering programs as *non-interfering concurrency graph*s (NICG). Figure 6(a) illustrates an example of NICG. It is important to note that we cannot eliminate a critical section even if they do not interfere with any other critical sections, since a critical section may be concurrent to itself. This can happen if a critical section is inside a method $Foo()$ and the method $Foo()$ is invoked multiple times from different parallel sections. Now one can define a "measure of parallelism" among critical sections in a NICG as the number of edges in the graph. Given this definition we can state the optimal lock assignment problem for NICG as follows: *Find the minimum number of locks that can be assigned to each critical section in NICG such that if there is an edge between*

two critical sections cs_1 and cs_2 then they both get different locks. One can immediately see that the above problem is equivalent to the classical graph coloring problem — given a NICG what is the minimum number of colors (i.e., locks) that are needed so that if there is an edge (i.e., concurrency) between cs_1 and cs_2 the two nodes (critical sections) get different colors (i.e., locks). We can immediately see that even for this special class of programs which contain only non-interfering critical sections, the problem of optimal lock assignment is NP-hard.[2]

5.2 General Critical Sections

For concurrency graphs with interfering critical sections, the problem of optimal lock assignment can be hard. Consider the concurrency graph with interfering location sets shown in Figure 6(b). One solution to the lock assignment for the critical section is shown in Figure 6(b), and the solution is a minimal lock assignment, which uses two locks. Notice that we have assigned the same lock (lock 1) to CS_5 and to CS_2 and CS_3, although there is no common data set between CS_5 and CS_2 or between CS_5 and CS_3. This should be fine since CS_5 is *not* concurrent with either CS_2 or with CS_3. For general concurrency graph, a critical section may be assigned more than one locks, for example, CS_2 in Figure 6(b). The semantics of such *lock set* in a critical section is as follows: Before a thread can execute a critical section it has to acquire all locks in the lock set, and once finished it has to release all locks in the lock set.[3] Given the notion of lock set for critical section, we can state the optimal lock assignment problem for general concurrency graph as follows: *Find the minimum number of locks that are needed so that (1) two concurrent critical sections that have some common locks in their lock set only if the two critical sections interfere, and (2) two concurrent critical sections get different lock set only if they do not interfer.* We conjecture that the stated optimal lock assignment problem is also NP-hard.[4]

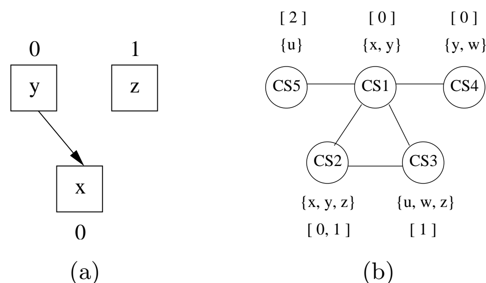


Figure 7: Location constrain graph and suboptimal lock assignment

Next we present a simple algorithm for computing lock assignment for the set of critical sections in a μSMP program. First let us introduce some notations. Let $cs_0, cs_1, \ldots cs_n$ be

[2]For certain classes of graphs, such as interval graphs, graph coloring problem can be solved in polynomial time. We conjecture that NICG do not fall in those categories of graphs.

[3]The concept of lock set has been used in the context of concurrency control in transactional processing, see `http://www.iona.com/support/docs/manuals/orbix /33/html/orbixots33_pguide/concurrent.html`.

[4]We believe the proof should be straightforward reduction of a known NP-hard problems, such as 3-SAT.

the set of critical sections. Let $LS(cs_i)$ be the set of memory locations that is accessed in cs_i. Let $CS(x) = \{cs | x \in LS(cs)\}$, essentially $CS(x)$ is the set of critical sections which accesses the location x. For instance, consider Figure 7(b), $LS(CS1) = \{x, y\}$ and $CS(y) = \{CS1, CS2, CS4\}$. Let $LS_{com} = \bigcup_{i,j} LS(cs_i) \cap LS(cs_j)$ **and** $Conc(cs_i, cs_j)$. Essentially, we add a location x to LS_{com} if x is accessed in at least two concurrent critical sections. For example in Figure 7(b), $LS_{com} = \{x, y, z\}$. Now we can make the following observation: If $CS(x) \subseteq CS(y)$ for any $x, y \in LS_{com}$ then x can be controlled by the same lock that controls y. Using this heuristics we can easily construct a simple lock assignment algorithm as follows.

1. Construct a simple location constraint graph (LCG) $G = (D, E)$, where nodes in D represents the elements of LS_{com} and there is an edge from y to x in G if $CS(x) \subseteq CS(y)$. The LCG for example in Figure 7(a) consists of three nodes $\{x, y, z\}$ and one edge $y \rightarrow x$ (see Figure 7). Notice that if $CS(x) = CS(y)$ then there is an edge from x to y and from y to x.

2. Compute the strongly connected components (SCCs) of G and collapse all non-trivial SCCs and construct the acyclic graph G_a.

3. Initialize each root node in G_a with a different lock (a node is a root node if it does not contain a predecessor node). For our example, we assign the two roots y and z $Lock(y) = 0$ and $Lock(z) = 1$.

4. Propagate the lock assignment from root nodes to all other nodes as follows: $Lock(x) = \bigcup_{y \in Pred(x)} Lock(y)$. Once again for our example, after propagation $Lock(x) = 0$.

5. Map the lock set assigned to each node in LCG back to critical section as follows: $CLock(c) = \bigcup_{x \in LS(c)} Lock(x)$. The lock assignment after mapping back to critical section is illustrated in Figure 7(b).

6. Assign new locks to all unassigned critical sections. For instance, after the previous steps, $CS5$ is not (yet) assigned a lock, and so we simply give a new lock to such "orphaned" critical sections. This simplifies out implementation rather than use graph coloring heuristics for "orphaned" critical sections. Note that we cannot simply remove the critical section $CS5$ since it can be concurrent to itself.

5.3 Discussion

Automatic lock assignment for critical sections will help alleviate some of the complexity of multithreaded programming. In OpenMP all unnamed critical sections are given the same global lock.[5] From a programmers perspective, the global lock model is straightforward: at any instance only one thread is active inside any critical section. Consider only a data race free program, global lock essentially supports the atomicity property that we assume in this paper.

[5]OpenMP has only recently introduced named critical sections, and all critical sections with the same name should be assigned the same lock.

Our lock assignment essentially preserves the semantics of global lock, even when different locks are used for critical sections.

Our lock assignment algorithm requires assigning multiple locks to the same critical section. If we constraint that each critical section must get a single lock, then the only way to do lock assignment is to reduce parallelism among critical sections or *split* critical sections. Consider, for instance, the motivating example in given in Figure 1. Using our approach we will assign locks to the three critical section as follows: $CS1 = \{0\}$, $CS2 = \{0, 1\}$ and $CS3 = \{1\}$. If we constraint that only one lock can be assigned to each critical section then it is impossible to assign locks to the critical sections with out reducing parallelism between $CS1$ and $CS3$. In other words, all three critical sections will be assigned the same lock, and we loose the parallelism between $CS1$ and $CS3$. An alternative approach is to split the critical section $CS2$ in to two parts $CS2'$ and CS'' such that $LS(CS2') = \{x, y\}$ and $LS(CS'') = \{z, w\}$. Now we can easily assume that we can use two different locks and improve on the parallelism. One can use graph partition algorithm for splitting critical sections. Splitting critical section is a complementary optimization problem that is beyond the scope of this paper.

6 Synchronization Anamolies

In this section we briefly discuss anomalies that can occur due to the incorrect placement of `barrier` and `post/wait` statements. We present techniques to detect some of the anomalies, and as part of barrier anomaly detection we also show how to construct barrier epochs. Given a μSMP program we want to ensure that one or more threads do not wait infinitely long at a barrier point. This can happen when some members of a team never reach a barrier point whereas others reach the barrier point. The threads that reach a barrier point will wait infinitely long at the barrier point. In this section we present a simple static analysis to determine when a μSMP program can give rise to infinite-wait state. In μSMP a parallel region can have more than one barrier. Incorrect usage of barriers can introduce inconsistent epochs, and hence infinite-wait error. For instance, consider piece of code show in Figure 6(a).

The two barriers at statement 4 and 8, can create inconsistent epochs — the epochs are consistent only if the `cond` at statement `7` is true. A program contains inconsistent epochs if one or more statements in a parallel region belong to more than one epoch. Barrier analysis is a static analysis for determining consistent epochs in a parallel region. It is important to note that epochs are defined with respect to a parallel region. In barrier analysis again assume that all paths are feasible ignore all back edges in the PCFG. We also ignore post/wait edges during barrier analysis.

Recall that a μSMP program is made of a set of nested parallel regions, and hence we can represent the set of of parallel regions using a rooted tree.[6] We perform barrier analysis one region at a time. When analyzing an outer parallel region we consider the inner nested parallel and critical region to be like a single non-barrier statement. With this understanding,

[6] We assume that the outermost region is also an implicit parallel region with only one section.

```
1: cobegin           1: cobegin
2:   section:        2:   section:
3:     S1 ;          3:     S1 ;
4:     barrier ;     4:     wait(e)
5:     S2 ;          5:     barrier ;
6:   section:        6:     S2 ;
7:     if(cond){     7:   section:
8:       barrier ;   8:     barrier ;
9:     }             9:     post(e)
10:     S3 ;         10:    S3 ;
11: coend            11: coend
      (a)                  (b)
```

Figure 8: Examples illustrating arrier and post/wait anomalies

our barrier analysis simply consists of propagating "barrier information" starting from the beginning of a parallel region to the end of the parallel region. We model each barrier in the PCFG as an element of a data flow set called the *barrier set*. Associated with each element is an epoch counter which indicates the current "epoch" — whenever we visit a barrier we increment the epoch counter of all barriers that reach the current barrier. At control flow merge we take the union of barrier set and also check if the epoch counter of the corresponding elements are equal. If not, we trigger an epoch inconsistency error, which in turn can give rise infinite wait at barrier points.

For post/wait analysis we want to determine which of the post/wait edges are illegal and spurious. Consider the example program shown in Figure 6(b). We can see that the post/wait edge from `post(e)` at 9 to `wait(e)` at 4 is will create deadlock. Now supposing we swap the `post(e)` and `wait(e)` statements at line number 4 and 9. In this case we introduce unnecessary control flow between them. The main reason for this is due to the presence of barriers. An important observation to make is the fact that a `post(e)` and the corresponding `wait(e)` have to be in the same (barrier) epoch. Otherwise we get unnecessary post/wait synchronization that can lead to deadlocks. Therefore when inserting post/wait control flow edges we only insert them if the corresponding post and wait are in the same epoch. It should be noted that there can be illegal post/wait synchronization even within the same epoch and which can lead deadlocks. We will not discuss analysis to detect such illegal post/wait edges in this paper.

7 Experimental Results

To study the feasibility and validity of our approach we implemented lock assignment and pointer analysis using the Omni OpenMP Compiler. OpenMP is a portable and scalable programming model and application developers can OpenMP directives to write parallel programs. In this section we briefly discuss our implementation framework and present preliminary results for lock assignment.

Omni OpenMP Compiler (OOC) [33] is a source-to-source compiler that translates OpenMP Fortran and C/C++ code with directives to C programs with Omni runtime support. Figure 9 shows our implementation part within the OOC. We implemented concurrency analysis, pointer analysis and lock assignment as part of Omni's backend. The Omni runtime systems is composed of three parts. The runtime library API provides implementations for each OpenMP constructs and directives to the general C compiler. The execution framework part executes the parallel executable generated by the compiler in a fork-join model in the target platforms, with the help of scheduling and resource management part. Lock acquisition, release and deadlock avoidance for critical sections are also implemented in the runtime system.

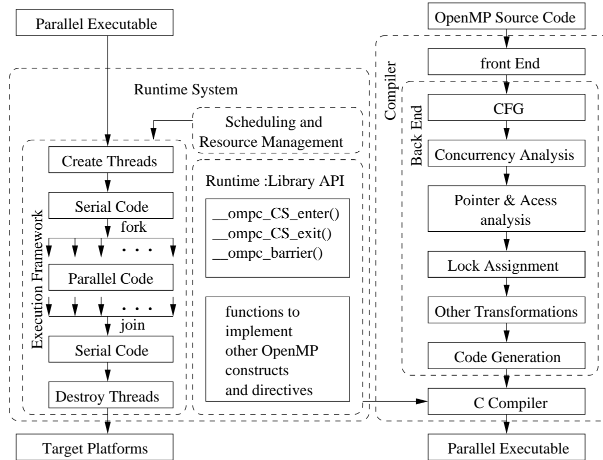


Figure 9: Omni OpenMP Compiler and Runtime System Infrastructure

We chose OpenMP implementation of NAS parallel benchmark (NPB) version 3.2 suite to evaluate our approach [2]. NPB consists of several kernels and simulated CFD (Computation Fluid Dynamics) applications derived from important classes of aero-physics applications. The main characteristics of NAS suite is given in Table 1. Some of the NAS applications use `atomic` directives and we treat them as critical sections[7].

We compiled the whole benchmark suite using Omni, calculated the number of statements and edges in PCFG, and number of concurrency pairs. The results are summarized in Table 2.

An interesting example for lock assignment problem is the program UA in NPB. There are 44 critical sections (atomic directives) in 3 different parallel regions, and 18 of them are in the parallel region in function *transfb_c_2*. Concurrency analysis shows that each pair of critical sections are concurrent, i.e., the interference graph is a clique of size 18. Three different arrays are accessed in such critical sections - *tmort*, *tx* and *mormult*. Data access analysis shows that 9 critical sections access *tmort* and *tx*, and the other 9 access *mormult* only. According to our lock assignment algorithm in section 5, we assign one lock to critical sections accessing *tmort* and *tx*, and another lock to critical sections accessing *mormult*.

Both the original code and Omni-generated code are run at Sun Sparc 400MHz 4CPU SMP

[7]The atomic directive ensures that a specific memory location is updated atomically.

Code	Main Feature	Base Language	Parallel Regions	Barriers	Critical Sections
BT	Navier-Strokes equations	Fortran	10	0	0
CG	Conjugate Gradient method to calculate matrix eigenvalue	Fortran	14	0	0
DC	Arithmetic data cube	C	2	0	2
EP	Embarrassingly parallel benchmark	Fortran	2	0	1
FT	FFT kernel	Fortran	8	0	0
IS	Integer sort kernel	C	5	1	0
LU	Navier-Strokes equations	Fortran	9	3	2
LU-HP	serial version of LU	Fortran	15	0	2
MG	MultiGrid method to solve Poisson equation	Fortran	11	0	0
SP	Navier-Strokes equations	Fortran	14	0	2
UA	Unstructured Adaptive NPB	Fortran	58	0	44

Table 1: NPB Program Characteristics

	Max	Min	Average
Statements in PCFG	485	5	63
Edges in PCFG	393	3	21
Edges in Concurrency Graph	114720	2	2440

Table 2: Concurrency Analysis Statistics for NPB3.2

machine. Table 3 summarizes the total execution time for the function *transfb_c_2* for different number of threads, and Figure 10 shows corresponding speedup. We evaluated the performance of only the function *transfb_c_2* instead of the whole program, since *transfb_c_2* takes a small portion of the whole execution time.

Number of Threads	1	2	3	4
Without Lock Assignment	0.19s	0.51s	0.62s	0.76s
With Lock Assignment	0.20s	0.35s	0.35s	0.41s
Performance Improvement	-5.3%	31.4%	43.5%	32.9%

Table 3: Performance Result (execution time) for Function *transfb_c_2*

From Table 3 we can observe a 30% to 44% improvement in performance by refining the global locks of critical sections in a multithreaded environment. One difficulty we faced during experimentation is the availability of OpemMP benchmarks that contain critical sections. We found only one benchmark that contains 44 atomic directives. We speculate that many scientific benchmarks avoid critical sections due to performance problem associated with global locking. Our lock assignment algorithm we can improve the performance of critical section, and we hope this will encourage application developers to write critical sections with global locks.

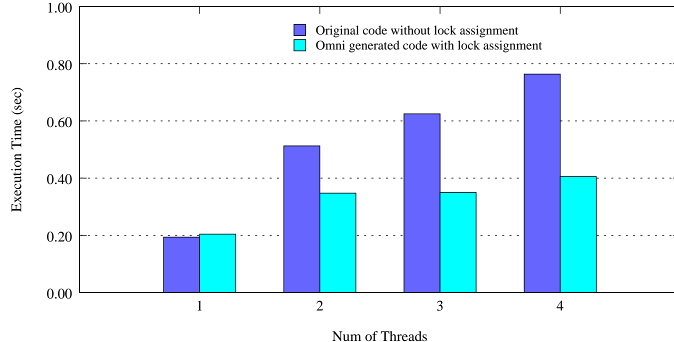


Figure 10: Lock assignment performance for *transfb_c_2*

8 Related Work

Concurrency analysis is a widely studied field, and several approaches have been presented. Callahan and Subhlok [5] proposed a data flow method to calculate a set of blocks which must be executed before a block can be executed for a parallel program. Parallelism is expressed by *parallel case*, and the synchronizations among threads are enforced by *post* and *wait* pair. This method was extended by Callahan, Kennedy and Subhlok [4] to analyze parallel loops. Duesterwald and Soffa [8] proposed a similar method in Ada rendezvous model, and uniquely, extended it to interprocedural analysis. Masticola and Ryder [29] presented an iterative non-concurrency analysis (a complement problem of concurrency analysis) framework. It first assumes a pessimistic estimation on *CHT* (*Can't Happen Together*) relation, then refines it iteratively. It still works for Ada, but includes binary semaphores as well as rendezvous. Naumovich and Avrunin [30] proposed a different way to build up the program graph, in which synchronization is expressed as a node, instead of edges. They used data flow equations to compute a *MHP* (*May Happen in Parallel*) set for each node. The method proposed by Jeremiassen and Eggers [18] was a bit different from all works listed above since it deals with course-grained, explicitly parallel program with only *barrier* synchronization. Its basic idea is to divide the program into a set of phases, and compute the control flow between them. Each phase consists of one or more sequences of statements that are delimited by barrier and can execute concurrently. Lin has essentially implemented Jeramiassen and Eggers work for OpenMP programming model [27]. Both Jeramiassen and Eggers and Lin do not focus on capturing inconsistent barrier placement. Our work goes beyond just concurrency analysis, we use the result of concurrency analysis for pointer analysis and lock assignment. Also, our programming model also support `post/wait` synchronization. Our barrier analysis is closely related to barrier inferencing proposed by Aiken and Gay [1]. Our approach uses a simple barrier flow analysis to detect inconsistent barrier placement and also to identify epochs. We use the result of barrier analysis to filter inconsistent barrier programs and also to filter inconsistent `post/wait` programs.

There also have been a lot of efforts on parallel program representation. Sarkar and Simons [39] proposed parallel program graphs (PPGs) that subsume program dependence graphs (PDGs) [11] and conventional control flow graphs. Lee, etc. [26] proposed a concurrent con-

trol flow graph (CCFG) for explicitly parallel shared memory programs with `cobegin/coend` and `parallel do` parallel constructs and `post/wait` synchronization. They also proposed a concurrent static single assignment (CSSA) form based on CCFGs. Lee [24] extended both the CCFG and CSSA to cover `barrier` and `locks` synchronization constructs. Novillo, etc. [32] proposed a parallel flow graph (PFG) which is an extension of CCFG with a different mutual exclusion synchronization representation.

There have been a number of work for data flow analysis to verify explicitly stated program properties [9, 3]. Programs are modeled as finite state or pushdown automata, the stated properties are varified using reachability properties. We take traditional data flow analysis based on Kildall's framework. Data flow analysis for explicit parallel program is another line of work that is closely related to ours [16, 21, 25, 38] Grunwald and Srinivasan present data-flow equations for computing reaching definition information for explicit parallel programs [16]. The data flow equations are specialized for different kinds parallel and synchronization constructs. This complicates the data flow analysis. Also, Grunwald and Srinivas require data independence with copyin/copyout semantics at parallel sections and assumes weak consistency memory model. Grunwald and Srinivasan does not handle critical sections or atomic sections. Knoop et al. present a bit-vector data flow analysis for explicit parallel programs [21]. They assume stronger consistency model with interleaving semantics, and do handle any synchronization constructs. Lee et al present an algorithm for constructing concurrent static single assignment (CSSA) for explicitly parallel programs and assume interleaving memory semantics [25]. They also restrict to only event-based post/wait synchronization. Sarkar presents data flow equations for reaching definitions using parallel program graph model of programs, that is an improvement over Grunwald and Srinivasan work [38]. Also, Sarkar does not deal with critical sections. There are a number of difference between our approach to the above related work. First of all, our data flow equation is independent of parallel and synchronization constructs — our data flow equation depend only on concurrency relation and isolation semantics for critical sections. We chose isolation semantics mostly because we want to treat critical sections as being atomic, and so we can naturally handle transaction memory model. Our data flow equation are a natural extension to classical Kildall equations, and if one drops $F(s)$ from the equations, we get the sequential data flow equation. Our equations can be applied to any monotone data flow problems and not just reaching definition problem or bitvector problems. We have applied our analysis to a non trivial pointer analysis data flow problem.

Shasha and Snir seminal work show how to model concurrency and synchronization relation for programs that assume sequential consistency and apply the model for analyzing and optimizing parallel programs [40]. The main focus of Shasha and Snir's work is to detect critical cycles in programs with implicit synchronization. Krishnamurthy and Yelick improve on Shasha and Snir work with knowledge of exploit synchronization to eliminate certain spurious critical cycles [22]. Navillo et al also model critical sections using explicit edges and cycels [31]. As discussed in Section 3.3 such explicit edges for modeling critical section can complicate concurrency analysis. Krishnamurthy and Yelick also present post-wait and barrier analysis and our work complements them. Our main focus in this paper is on data flow and pointer analysis

rather than synchronization analysis to ordering among statements.

Recently Rugina and Rinard present pointer analysis for explicit parallel programs. There are a number of differences between our pointer analysis and Rugina and Rinard pointer analysis for multithreaded programs. Two key concepts that we use in our pointer analysis are (1) atomicity memory model for critical sections and (2) concurrency relation. We directly incorporate these two concepts with in the classical data flow equations for pointer analysis. Unlike Rugina and Rinard's work we can naturally handle synchronization constructs, including critical sections, post/wait, and barriers.

There have been a number of recent work on pointer and escape analysis for multithreaded Java programs [6, 35, 37]. In Java threads are treated like objects and an explicit `start` instruction is used to create new threads. When a `start` instruction is executed the corresponding `run` method defined in a class that implements `Runnable` interface is executed. Interthread analysis essentially consists of mapping parent thread information to child thread information and the mapping of child thread information back to parent information. Often interthread analysis is treated like interprocedural analysis and therefore the resulting pointer information can be conservative. Salcian and Rinard introduce program interaction graph for pointer/escape analysis of Java programs. Unlike other pointer/escape analysis Salcian and Rinard analysis model Java threads more precisely. Although we have explicitly modeled Java threads in our work, we can easily extend the work to deal with Java thread model. Once concurrency information is computed for Java programs, we can reuse the data flow analysis that is described in this paper. Recently there have been some work on concurrent slicing for multithreaded programs [10, 17] Due to space constraint we will not discuss this line of research work.

Besides critical sections, atomic section is another mutual exclusion construct which follows the atomicity semantics. Atomic sections is an important synchronization mechanism in X10 [19] - a new programming language for DARPA/IBM PERCS architecture.

To be the best of our we are not aware of any work on lock assignment for improving the performance of parallel programs. We believe that with compiler taking over lock assignment (similar to register assignment) one could simplify the programming model for by assuming global locks for all critical sections (which is the case for OpenMP).

9 Conclusions

In this paper we proposed a new framework for analyzing and optimizing shared memory parallel programs. Our approach reifies concurrency relation and isolation semantics and uses them for all of our analysis and optimization. We showed how to apply our framework for solving pointer analysis problem and lock assignment problem. We believe unnamed critical sections provide a uniform model and simpler programming model for end users. We can then use lock assignment to assign locks to critical sections to improve the performance of the program. Our approach to pointer analysis is a straightforward extension to classical data flow analysis. We are currently working towards applying our approach to object-based multithreaded programs, such as Java

programs and X10 programs.

Acknowledgement

We would like to acknowledge Vivek Sarkar and Kemal Ebcioglu from IBM T.J. Watson Research Center for fruitful discussion on this topic. We also thank useful discussions from members of the CAPSL group at the University of Delaware, in particular Weirong Zhu, Hongbo Rong, Hongbo Yang, and Joseph Manzano. Finally, the last two authors wish to acknowledge the support in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

References

[1] Alexander Aiken and David Gay. Barrier inference. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–354, New York, NY, USA, 1998. ACM Press.

[2] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[3] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 62–73, New York, NY, USA, 2003. ACM Press.

[4] 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.

[5] David Callahan and Jaspal Sublok. Static analysis of low-level synchronization. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 100–111. ACM Press, 1988.

[6] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[7] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, Tokyo, Japan, June 1986.

[8] 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, New York, NY, USA, 1991. ACM Press.

[9] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 62–75, 1994.

[10] Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[11] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[12] Guang R. Gao and Vivek Sarkar. Analyzable atomic sections: Integrating fine-grained synchronization and weak consistency models for scalable parallelism. Technical report, Dept. of Electrical and Computer Engineering, University of Delaware, 2004. CAPSL TM 52.

[13] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[14] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 121–133, New York, NY, USA, 1998. ACM Press.

[15] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Department of Computer Science, University of Wisconsin, Madison, February 1991.

[16] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 159–168, New York, NY, USA, 1993. ACM Press.

[17] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis Symposium*, pages 1–18, 1999.

[18] 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.

[19] Vivek Sarkar Kemal Ebcioglu and Vijay Saraswat. X10: Programming for hierarchical parallelism and nonuniform data access. *LaR'04, Language Runtimes*, 2004.

[20] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973.

[21] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[22] Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38(2):130–144, 1996.

[23] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[24] Jaejin Lee. *Compilation techniques for explicitly parallel programs.* PhD thesis, 1999. Adviser-David A. Padua.

[25] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In *Languages and compilers for parallel computing. Proceedings of the 10th international workshop. Held Aug., 1997 in Minneapolis, MN.*, Lecture Notes in Computer Science. Springer-Verlag, New York, 1998. (to appear).

[26] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 1999. ACM Press.

[27] Yuan Lin. Static nonconcurrency analysis of openmp programs. In *FIRST INTERNATIONAL WORKSHOP on OpenMP*, 2005.

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

[29] 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.

[30] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 24–34, New York, NY, USA, 1998. ACM Press.

[31] Diego Novillo, Ron Unrau, and Jonathan Schaeffer. Concurrent SSA form in the presence of mutual exclusion. In *Proceedings of the 1998 International Conference on Parallel Processing (ICPP'98)*, pages 356–364, 1998.

[32] Diego Novillo, Ronald C. Unrau, and Jonathan Schaeffer. Concurrent ssa form in the presence of mutual exclusion. In *ICPP '98: Proceedings of the 1998 International Conference on Parallel Processing*, page 356, Washington, DC, USA, 1998. IEEE Computer Society.

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

[34] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[35] Erik Ruf. Effective synchronization removal for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 208–218, New York, NY, USA, 2000. ACM Press.

[36] Radu Rugina and Martin C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25(1):70–116, 2003.

[37] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001.

[38] Vivek Sarkar. A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs (Extended Abstract). *Springer-Verlag Lecture Notes in Computer Science*, 757:16–30, 1992. Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, Yale University, August 1992.

[39] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 633–655, Portland, Oregon, August 1993. Springer-Verlag. Published in 1994.

[40] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. Technical report, IBM, 1987. Report RC12936.

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

A Parallel Region Tree

The following algorithm creates a nested region tree, where $Child(s)$ denotes the children of s in the region tree, and $Succ(m)$ denotes the set of successor nodes of m in the PCFG.

```
RegionTree() {
1:     create a parallel region r and a section region s
2:     insert s in Child(r)
3:     VisitRegion(s, Succ(root))
}
VisitRegion(Region s, NodeSet ns) {
4:     foreach n ∈ ns and n is not yet visited {
5:       Switch(Type(n)) {
6:         case cobegin:
7:           Create a parallel region node p
8:           Insert p in Child(s)
9:           VisitRegion(p, Succ(n))
10:          break ;
11:        case section:
12:          Create a section region p
13:          Insert p in Child(s)
14:          VisitRegion(p, Succ(n))
15:          break ;
16:        case critical:
17:          Create a critical region p
18:          Insert p in Child(s)
19:          VisitRegion(p, Succ(n))
20:          break ;
21:        case coend: Skip
22:          break ;
23:        case others:
24:          Create a simple node p
25:          Insert p in Child(s)
26:          VisitRegion(s, Succ(n))
27:          break ;
28:      }
29:    }
  }
```