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

Concurrency Analysis and Its Applications

Yuan Zhang† Guang R. Gao

CAPSL Technical Memo 59
May 28th, 2005

†Dept. of Electrical and Computer Engineering
University of Delaware
zhangy,weirong,fchen,hu,ggao@capsl.udel.edu

Abstract

Concurrency analysis is a static analysis technique to determine whether two statements in a concurrent program can be executed in parallel. In this paper we propose a concurrency analysis method for programs with **parallel for** and **parallel sections** constructs, and **barrier**, **locks** and **post/wait** synchronization mechanisms. Our method contains two parts. One is divide the programs into *concurrent blocks*, and build up a *concurrent block flow graph* (CBFG). The other is calculate the concurrency relations among blocks using data flow equations. We also briefly introduce two application of concurrency analysis - pointer analysis for parallel programs, and static data race detection.

Contents

List of Figures

List of Tables

1 Introduction

Concurrency analysis is a static analysis technique which determines whether two statements in a parallel program can be executed in parallel. Information about this behavior has a wide range of applications in debugging, optimization, data flow analysis, and synchronization anomaly detection, etc.

The problem of precisely determining whether two statements in a parallel program is concurrent is known as NP-complete [1]. A practical solution is to compute a conservative estimation, so that if there exists an execution in which statement S_1 and statement S_2 in a parallel program are executed in parallel, then the concurrency relation (S_1, S_2) is in the conservative estimation. However, some additional pairs may also be included in conservative estimation.

Concurrency analysis is closely coupled with some specific programming model and synchronization mechanisms. It is a widely studied field, and several approaches have been presented. Callahan and Subhlok [2] 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 [3] to analyze parallel loops. Duesterwald and Soffa [4] proposed a similar method in Ada rendezvous model, and uniquely, extended it to interprocedural analysis. Masticola and Ryder [5] 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 [6] proposed a different way to build up the program graph, in which synchronization is expressed as a node, instead of edges in previous work listed above. Based on that, they used data flow equations to compute a *MHP* (*May Happen in Parallel*) set for each node in polynomial time. The method proposed by Jeremiassen and Eggers [7] 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.

Almost all work listed above are targeted to Ada (or Ada-like) programs with event based synchronization. However, their effectiveness is limited when applied to modern shared memory programming languages, such as OpenMP [8] and Pthread [9], in which synchronization are often enforced by high level mutex constructs such as locks and critical sections. In such programs, it is difficult to define the ordering among threads during the static analysis stage. Thus one concern of our work is to propose a more effective way to analyze mutex structures. Moreoever, these programming languages usually contain a wide range of parallel and synchronization constructs. How to build up a uniform infrastructure for them is our another concern. The language constructs covered in our work include:

1. Parallelism constructs: **parallel do** and **parallel sections**;
2. Synchronization constructs: **post/wait**, **barrier** and **locks**;

Last but not the least, we try to refine the analysis on **barrier** and obtain a less conservative solution than previous work.

The remainder of the paper is organized as follows. In section 2 we briefly introduce the programming model, and the semantics of each language construct. In section 3 we discuss our concurrency analysis method in detail. In section 4 we present several applications of concurrency analysis, such as pointer analysis and automatic lock assignment. Finally we conclude and summarize in section 5.

2 Programming Model

This section describes a language $\mathcal{L}$ with asynchronous parallel constructs and synchronization constructs. We present them as a subset of OpenMP [8] extended with event based synchronization **post/wait**, although these constructs are essentially independent with any of the language.

2.1 Parallel Constructs

The language $\mathcal{L}$ follows a nested fork-join execution model. The program begins execution as a single thread called *master thread*. When parallel constructs are encountered, the master thread generates a team of threads to execute the enclosed code. When they complete, they synchronize and terminate, leaving only the master thread to proceed.

Parallelism is explicitly expressed by two language constructs: **parallel for** and **parallel sections**. **Parallel for** is same as **parallel DO** in Callahan, Kennedy and Subhlok [3]'s work. When control reaches the **parallel for** construct, all iterations of the loop body are started and proceed concurrently and asynchronously. The **parallel sections** has the same semantics with **cobegin/coend**. It specifies a fixed set of tasks to be executed concurrently in such a way that the code segment S_i is executed by the ith thread in the team. Figure 1 illustrates examples for both constructs. Note that the parallel region is a structured block, i.e., branches are not allowed from within the parallel region to outside, or vice versa.

2.2 Synchronization Constructs

There are three synchronization mechanisms included in $\mathcal{L}$: event variables, locks, and barriers. An event variable is always in one of two states: *clear* or *posted*. The initial value of an event variable is *clear*. It can be set of *posted* with the construct **post** as in Figure 2(a). A **wait** construct suspends the executing thread until the specified event variable's value is set to *posted* by another thread.

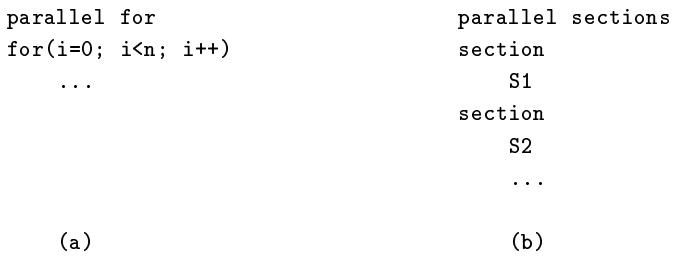

Figure 1: Asynchronous Parallel Constructs

The **locks** constructs, as shown in Figure 2(b), enforces the mutual exclusion among threads. It defines a region of code, usually called the *critical section*, that can be executed exclusively by one thread at any time. The lock variable has two states: *locked* and *unlocked*. The initial value is *unlocked*. Before entering the critical section, threads compete for the exclusive permission. One thread wins, and it sets the local variable state to *locked*. This action is always referred to as *lock acquisition*. After it finishes the critical section, it *releases* the lock by setting its state to *unlocked*, and other threads can compete to enter. In this work we don't consider the nested **lock** constructs. Although it is easy to extend the semantics to cover this case, we have not observed any application of it.

The **barrier** construct enforces a coarse-grained, explicit ordering among threads. When a thread encounters a **barrier**, it must be suspended until the whole team of threads reach the same point. After that, each thread begins executing the code after the barrier concurrently. An example of **barrier** is shown in Figure 2(c).

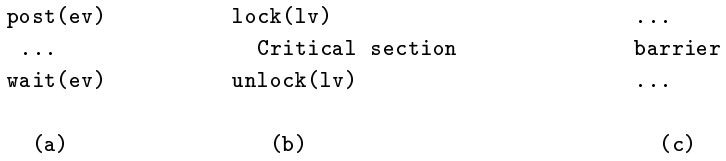

Figure 2: Synchronization Constructs

3 Concurrency Analysis

The core of the concurrency analysis technique is to compute a partial order of statements based on some data flow framework. It consists of two key problems, one is how to represent, usually graphically, the dependences in the parallel program, the other is how to calculate the concurrency relationship based on that graph. The basic idea is to figure out dependences among threads (usually specified by synchronization, thread local control flow, and memory model, etc.) that must be preserved for program correctness, and two *independent* statements from two threads can be concurrent. We discuss each of them in detail in this section.

3.1 Parallel Program Representation

As mentioned in section 1, some previous work have proposed methods to build up graphs for parallel programs with **parallel for** and **parallel sections** constructs, and **post/wait** synchronizations. But they didn't have solutions for representation of **locks** and **barriers**.

Besides, there have been some other work done on parallel program representations, although their original objects are not on concurrency analysis. Sarkar and Simons [10] proposed parallel program graphs (PPGs) that subsume program dependence graphs (PDGs) [11] and conventional control flow graphs. Lee, Midkiff and Padua [12] proposed a concurrent control 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 [13] extended both the CCFG and CSSA to cover **barrier** and **locks** synchronization constructs. Novillo, Unrau and Schaeffer [14] proposed a parallel flow graph (PFG) which is an extension of CCFG with a different mutual exclusion synchronization representation.

We found that PFGs presented by Novillo etc. [14] would miss some concurrent pairs due to its lock representation. For instance, consider the code segment shown in Figure 3. In its corresponding PFG in Figure **??**, solid lines represents the control edges, and dotted lines represents the synchronization. The dependences shown in the PFG requires the statement "$x++$" in thread 1 to be finished before "$y--$" in thread 2, or the statement "$y++$" in thread 2 to be finished before "$x--$" in thread 1. However, **locks** guarantees mutual exclusion, but not enforce any order between threads. Hence "$x++$" might be concurrent with "$y--$", and "$y++$" might be concurrent with "$x--$".

```
Thread 1          Thread 2

x ++              y ++
lock(lv)          lock(lv)
  CS1               CS2
unlock(lv)        unlock(lv)
x --              y --
```

Figure 3: Example of the problem on **locks** representation

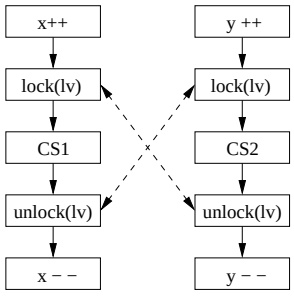

Figure 4: The PFG for Figure 3

In this work we propose a concurrent block flow graph (CBFG) which takes the advantages of the synchronization control flow graph [2] and CCFG [13], and represents locks in a different way. This CBFG is especially suitable for applications such as pointer analysis and automatic lock assignment to be mentioned in section 4.

The CBFG of a parallel program on $\mathcal{L}$ is a collection of *concurrent blocks* connected according to their control flow orders and synchronization orders. A *concurrent block* is a maximum set of statements that is uninterrupted by thread interactions. It is defined as follows.

Definition: Concurrent Block
A concurrent block has the following properties:

1. A sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end.
2. Only the first statement can be a **wait**.
3. Only the last statement can be a **post**.
4. If a concurrent block contains a **barrier**, **parallel begin/parallel end**, or **section begin/section end** statement, then that statement is the only one in the concurrent block.
5. The entire *critical section* and the enclosing **lock** and **unlock** statement are reduced to one concurrent block.

The concurrent block flow graph $G = (N, E, Ntype, Etype, entry, exit)$ is defined as follows.

Definition: Concurrent Block Flow Graph
A *concurrent block flow graph (CBFG)* is a directed graph $G = (N, E, Ntype, Etype, entry, exit)$, such that:

1. N is the set of nodes in G. Each node is a concurrent block.
2. $E \subseteq N \times N \times Etype$ is a set of edges in G. $E = E_c \cup E_s$, where
   - $E_c = \{(m, n, Etype(m, n)) | m, n \in N \wedge Etype(m, n) \in \{T, F, U\}\}$ is a set of control flow edges.
   - $E_s = \{(m, n, Etype(m, n)) | m, n \in N \wedge Etype(m, n) \in \{SE, SB\}\}$ is a set of synchronization flow edges.
3. $Ntype$ is a node type mapping. Its domain and range are as follows: $Ntype : N \rightarrow T_N, T_N = \{Sbegin, Send, Fbegin, Fend, Bar, Evt, Lock, Comp\}$.
4. $Etype$ is an edge type mapping. Its domain and range are as follows: $Etype : E \rightarrow T_E, T_E = \{T, F, U, SE, SB\}$.

The *Sbegin* and *Send* nodes correpsond to the begin and the end of the **parallel sections** statement. One outgoing edge corresponds to one section. Threads are created at the *Sbegin* node, and completed and joint at the *Send* node.

The *Fbegin* and *Fend* nodes correspond to the begin and the end of the **parallel for** statement. A team of threads are created at *Fbegin*, and joins at *Fend*. For the purpose of the static concurrency analysis, we assume there are two virtual threads generated, and we

duplicate the CBFG of the associated loop body for each of them. The back edge of the loop is ignored.

The Bar node corresponds to the **barrier** statement. The same Bar nodes in different virtual threads are connected by a bidirected edge with type SB.

A node with the type Evt corresponds the concurrent block containing the **post** statement or the **wait** statement. There is a synchronization edge with type SE from **post(ev)** to **wait(ev)**, where ev is an event variable.

The $Lock$ node corresponds to the entire critical section with the enclosing **lock** and **unlock** statements. All other nodes have the type $Comp$, connected by control flow edges with unlabeled type U, true type T, or false type F.

As an example, Figure 6 shows the CBFG of the sample code in Figure 5. Note that all unlabeled edges have type of U.

```
x = 0                        parallel for
y = 0                        for(i=0; i<N; i++){
z = 0                           if(a > 0 && b > 0){
a = ReadFile()                     x = Random()
b = ReadFile()                     lock(lv)
parallel sections                     z = z + x
section                            unlock(lv)
{                                  barrier
   a = 1                           lock(lv)
   post(ev)                           z = z - y
}                                  unlock(lv)
section                         }
{                               else
   wait(ev)                        print(a, b)
   b = a + 1                 }
}
```

Figure 5: A sample code

3.2 Concurrency Analysis Algorithm

As mentioned before, the core of the concurrency analysis technique is to compute a partial execution order among statements. Statement ordering in a concurrent program is enforced by various mechanisms. One is control flow within a thread, the other is inter-thread interactions enforced by synchronization.

In our analysis framework, we have split the concurrent program into concurrent blocks, which contains a maximum set of statements executed by a thread *at one time* without any inter-thread interaction (but threads can still communicate with each other *implicitly* through reading or writing shared data). This concurrent block has a desirable property that if two blocks B_1 and B_2 are concurrent, then $\forall S_1 \in B_1$ are concurrent with $\forall S_2 \in B_2$, where S_1 and

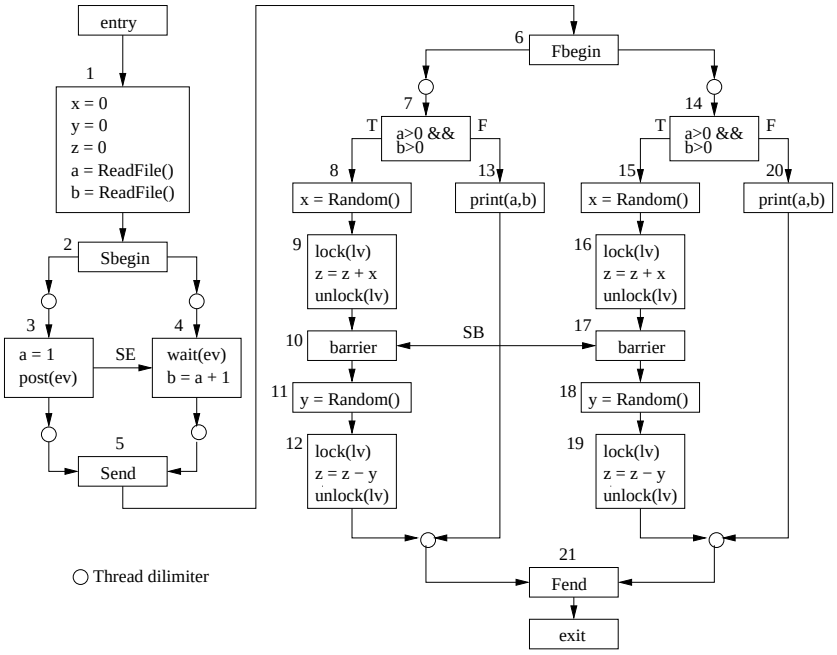

Figure 6: The concurrent block flow graph of the code in Figure 5

S_2 are statements in B_1 and B_2, respectively. Thus the concurrency analysis among statements can be reduced to the discussion of concurrency among blocks.

Given a concurrent block flow graph $G = (N, E, Ntype, Etype, entry, exit)$, we say concurrent block B_1 *must happen before* (MHB) concurrent block B_2 if and only if B_1 must be finished before B_2 can be executed in any execution in which both blocks are executed. We define the predicate $MHB(u, v)$ as:

$$MHB(m,n) = \begin{cases} true & : \textit{if u must happen before v} \\ false & : \textit{otherwise} \end{cases}$$

Define a path from block u to block v in $G = (N, E, Ntype, Etype, entry, exit)$ as a finite sequence of vertices $n_0, n_1, \cdots, n_k \in N$, such that $(n_{i-1}, n_i) \in E, 1 \leq i \leq k$, and $n_0 = u \wedge n_k = v$. MHP(u, v) can be inferred from the following theorem.

Theorem I: $MHB(u, v)$ if there is a path in G from u to v.

Proof: Given a path $n_0, n_1, \cdots, n_k \in N$, where $(n_{i-1}, n_i) \in E, 1 \leq i \leq k$, and $n_0 = u \wedge n_k = v$, in order to prove $MHB(u, v)$, we must prove that $MHB(n_{i-1}, n_i)$, $1 \leq i \leq k$. We have three cases:

1. $Etype(n_{i-1}, n_i) \in \{T, F, U\}$, (n_{i-1}, n_i) is a thread local control flow edge, $MHB(n_{i-1}, n_i)$ holds.

2. $Etype(n_{i-1}, n_i) = SE$, then from CBFG's definition we know n_{i-1} is **post**, and n_i is **wait**. n_i's executing thread has to be suspended until the event variable is posted by n_{i-1}. Therefore $MHB(n_{i-1}, n_i)$.

3. $Etype(n_{i-1}, n_i) = SB$, then $Ntype(n_{i-1}) = Ntype(n - i) = Bar$. This is a special case. According to **barrier**'s semantics in section 2, only when both threads have reached the same barrier, can they proceed together. So n_{i-1} and n_i can be considered as concurrent, or executed one by one. $MHB(n_{i-1}, n_i)$ holds.

Therefore $MHB(u, v)$.

End of proof.

The importance of this **Theorem I** is that it transforms the ordering problem into the reachability calculation: block u must happen before v if u can reach v.

For each concurrent block b, define the *happen before set* $HB(b)$ as:

$$HB(b) = \{n | MHP(n, b)\}$$

$HB(b)$ can be calculated by the following data flow equations:

$$HB(i) = \begin{cases} \phi & \text{if } i = entry \\ \bigcup_{(j,i)\in E} (HB(j) \cup \{j\}) - \{i\} & otherwise \end{cases}$$

As an example, Table 1 lists the *happen before set* of concurrent blocks in Figure 6.

Block i	$HB(i)$	Block i	$HB(i)$
1	ϕ	12	{1-11, 14-17}
2	{1}	13	{1-7}
3	{1-2}	14	{1-6}
4	{1-3}	15	{1-6, 14}
5	{1-4}	16	{1-6, 14-15}
6	{1-5}	17	{1-10, 14-16}
7	{1-6}	18	{1-10, 14-17}
8	{1-7}	19	{1-10, 14-18}
9	{1-8}	20	{1-6, 14}
10	{1-9, 14-17}	21	{1-20}
11	{1-10, 14-17}		

Table 1: Happen before sets for concurrent blocks in Figure 6

In Duesterwald and Soffa's work [4], two units u and v are defined as concurrent if u cannot happen before v, and u cannot happen after v, and vice versa, i.e. they are not connected. Denote the set of concurrent blocks with block b as $Cur(b)$, the above condition can be summarized into the following equation:

$$Cur(u) = \{v | \neg(MHB(u, v) \vee MHB(v, u))\} \quad (1)$$

Applied to Figure 6, we obtain the concurrency results shown in Table 2.

Block u	$Cur(u)$	Block u	$Cur(u)$
1	ϕ	12	{13, 18-20}
2	ϕ	13	{8-12, 14-20}
3	ϕ	14	{7-9, 13}
4	ϕ	15	{7-9, 13, 20}
5	ϕ	16	{7-9, 13, 20}
6	ϕ	17	{13, 20}
7	{14-16, 20}	18	{11-13, 20}
8	{13-16, 20}	19	{11-13, 20}
9	{13-16, 20}	20	{7-13, 15-19}
10	{13, 20}	21	ϕ
11	{13, 18-20}		

Table 2: Concurrency Analysis Result

However, we should pay attention to two special cases. One is ordering among critical sections. The other is ordering with respect to barriers. As mentioned in section 2, **locks** enforces mutually exclusive accesses to a critical sections. This mutex is guaranteed by setting and unsetting the lock variable when the executing thread acquires and releases it, respectively. Different critical section instances also have to be executed in a sequential order if they are guarded by the same lock variable. In Figure 6, block 9 and block 16 are two instances of the same critical section, they are executed in a sequential order even though two threads might reach the same lock acquisition site concurrently. Similarly, block 9 and block 19 (or block 12 and block 16) are not concurrent since they are guarded by the same lock variable lv, although in this specific example they have been explicitly ordered by the barrier. Summarize this condition into Equation 1, we have

$$Cur'(u) = \begin{array}{l} \{v|(\neg(MHB(u,v) \vee MHB(v,u))) \\ \wedge((Ntype(u) = lock \wedge Ntype(v) = lock) \rightarrow LV(u) \neq LV(v))\} \end{array} \quad (2)$$

where $LV(b)$ returns block b's lock variable if b is a lock block.

The other elaboration arises from the program correctness consideration at the presence of barriers. Consider block 8 and block 20 in Figure 6. Equation 1 determines that they are concurrent since they are not connected. However, due to the global barrier in block 10 and 17, both threads must either take the true path together, or take the false path together. Otherwise, one of them will wait at the barrier for ever, and the program will never halt. Hence in any "correct" program, blocks in one path are not concurrent with those in the other path.

Let's consider a **parallel for** construct [1]. Since the subgraphs for both virtual threads in CBFG are identical, we only care about one copy of the loop body. Figure 7 shows such an example. Due to the barrier, B_4 and B_5, B_4 and B_6 are not concurrent pairs. Similarly, B_2 and

[1]We don't consider the **parallel sections** case since **barrier** constructs cannot be used in **parallel sections** for the same reason.

B_5, B_2 and B_6 are not concurrent. But B_2 and B_4 are concurrent, because even if two threads take different paths, the one passing B_3 still can choose to take the path of B_4 from B_3.

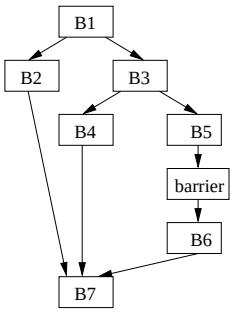

Figure 7:

Denote the subgraph of a **parallel for** construct loop body as SG, and ignore any synchronization edge from other threads, then SG contains only thread local control flow edges. Block b's *dominator* $Dom(b)$, *post-dominator* $PDom(b)$, *immediate dominator* $IDom(b)$, and *immediate post-dominator* $IPDom(b)$ are defined on SG in the same way as in [15]. Define the *common immediate dominator* of block u and v as follows:

Definition: In SG, the *common immediate dominator* $CIDom(u, v)$ *of block* u *and* v *satisfies:*

1. $CIDom(u, v) \in Dom(u) \wedge CIDom(u, v) \in Dom(v)$

2. *There does not exist* w, $w \in Dom(u) \wedge w \in Dom(v)$, *and* w *dominates* $CIDom(u, v)$.

Similarly, we can define the common immediate post-dominator $CIPDom(u, v)$ of block u and v. The following theorem determines whether two blocks u and v, satisfying $u \in Cur'(v)$ and $v \in Cur'(u)$ by equation 2, are concurrent due to the program correctness property at the presence of barriers.

Theorem II: Two blocks u and v are concurrent if:

1. $u \in Cur'(v) \wedge v \in Cur'(u)$, and

2. There exists a barrier-free path from $CIDom(u, v)$ to $CIPDom(u, v)$ which passes u, and there exists a barrier-free path from $CIDom(u, v)$ to $CIPDom(u, v)$ which passes v.

The proof of this theorem is left to future work.

The concurrent set of block b specified by **Theorem II** is denoted as $Cur''(b)$. Table 3 shows the final concurrency analysis results for Figure 6.

Block i	$Cur''(i)$	Block i	$Cur''(i)$
1	ϕ	12	{18}
2	ϕ	13	{14, 20}
3	ϕ	14	{7-9, 13}
4	ϕ	15	{7-9}
5	ϕ	16	{7-8}
6	ϕ	17	ϕ
7	{14-16}	18	{11-12}
8	{14-16}	19	{11}
9	{14-15}	20	{7, 13}
10	ϕ	21	ϕ
11	{18-19}		

Table 3: Final concurrency analysis result for Figure 6

4 Applications

4.1 Pointer Analysis for Parallel Programs

Pointer analysis is one of the most important program analysis techniques, which tracks information about the memory locations to which pointers may point. There have been a lot of mutual pointer analysis techniques for sequential programs. However, it is difficult to apply them to parallel programs, due to the potential interference among parallel threads. Rugina and Rinard [16] proposed a interprocedural, flow-sensitive, 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. Although effective for structured parallel constructs (including fork-join constructs, parallel loops, and conditional spawned threads), it ignores synchronization constructs such as locks, semaphores and critical sections, hence is conservative for programs using these constructs.

We claim that concurrency analysis results can help to deal with synchronization constructs, or other cases that cannot be solved in Rugina and Rinard's algorithm. The intuition is simple: interference can only occur among concurrent statements. In concurrent block flow graph (CBFG), the interference edges from other threads which influence the points-to calculations for statements in block b is the union of points-to edges created by blocks concurrent with b. To keep the integrity of this paper, we repeat the definition of *multithreaded points-to information* in [16] as follows:

"**Definition:** Let L be the set of location sets in the program and $P = 2^{L \times L}$ the set of all points-to graphs. The *multithreaded points-to information MTI(p)* at a program point p of the parallel program is a triple $< C, I, E > \in P^3$ consisting of:

- the *current points-to graph* C,

- the set I of *interference edges* created by all the other concurrent threads (for the current statement),

- the set E of *edges created by the current thread*."

The basic data flow equations for statement st is:

$$
\begin{array}{l}
[st] < C, I, E >=< C', I', E' >, \text{ where} \\
C' = \begin{cases} (C - kill) \cup gen \cup I & \textit{if strong} \\ C \cup gen \cup I & \textit{if not strong} \end{cases} \\
I' = I \\
E' = E \cup gen
\end{array}
$$

And the data flow equations for concurrent block b is:

$$
\begin{array}{l}
[b] < I, E >=< I', E' >, \text{ where} \\
I' = \bigcup_{i \in Cur''(b)} I(b) \\
E' = \bigcup_i E_i \text{ for } st_i \in b \\
[st_i] < C_i, I, E_i >=< C_i', I, E_i' > \text{ for } st_i \in b
\end{array}
$$

4.2 Automatic Lock Assignment

4.3 Data Race Detection

Data races occur in a parallel program when two threads access a shared data concurrently without any ordering constraints, and at least one of them is a write. Data races are usually bugs in parallel programs, but hard to detect and debug. The reason is that they may exhibit different behaviors when executed on the same inputs. The data race detection techniques can be classified into two categories, one is static data race detection, which detects all possible race conditions at the compilation time; the other is dynamic data race detection, which detects races at the execution time.

Once the concurrency relationship among blocks are available, we can statically detect the data races by analyzing the definitions and uses of shared data among concurrent block.

Claim: Let $Def(b)$ be the set of shared data defined in concurrent block b, and $Use(b)$ be the set of shared data used in b. A data race occurs between concurrent blocks u and v if and only if:

1. u and v are concurrent, and

2. $Def(u) \cap Def(v) \neq \phi$, or $Def(u) \cap Use(v) \neq \phi$, or $Use(u) \cap Def(v) \neq \phi$.

For example, in the program shown in Figure 6, block 8 and block 16 are concurrent, $Def(8) = \{x\}$, and $Use(16) = \{x, z\}$, $Def(8) \cap Def(16) = \{x\} \neq \phi$, thus a potential data race. Similarly, block 11 and block 18, which are duplicates of the same set of statements in the parallel program, are concurrent, and $Def(11) = Def(18) = \{y\}$, hence they may generate another data race.

5 Conclusions

In this paper we present a concurrency analysis method for parallel programs with **parallel for** and **parallel sections** constructs, and **barrier**, **locks** and **post/wait** synchronization mechanisms. Our method contains two parts. One is divide the program into a set of *concurrent blocks* and build up a *concurrent block flow graph* (CBFG). The *concurrent block* has a good property that the executing thread cannot be interrupted by synchronization. The CBFG catches dependences in the program which must be preserved for correctness. The other part is to calculate the concurrency relations among blocks using data flow equations. Compared with previous work, our method has several advantages:

1. It analyzes the mutex structures (specified by **locks**) more accurately.
2. It gives less conservative solution to analysis related with the **barrier** construct.
3. It provides a uniform infrastructure for a wide range of parallel language constructs.

We also illustrate the application of this technique in parallel program pointer analysis and static data race detection.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] *Threads Extension for Portable Operating Systems*, 1994.

[10] 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.

[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] 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.

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

[14] 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.

[15] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

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