Parallel Nondeterminator

He Yuxiong and Wang Junqing

Singapore-MIT Alliance

A report submitted for the course

MIT 6.895/SMA 5504 Theory of Parallel Systems

Abstract

Detecting data race is very important for debugging shared-memory parallel programs, because data races result in unintended nondeterministic execution of the program. We propose a dynamic on-the-fly race detection mechanism called Parallel Nondeterminator to check for determinacy races during the parallel execution of a program with recursive spawn-sync parallelism. A modified version of Nested Region Labeling scheme is developed for the concurrency relationship test in the spawn-sync parallel structure. Through the identification of Least Common Ancestor in the spawn tree, the Parallel Nondeterminator only needs to keep two read access records and one write access record for each shared location. The work and critical path in the instrumented codes are analyzed as well as time complexity and space requirements. Let N denote the maximum depth of the recursion in the parallel program. The worst case time increased for each spawn operation is $O(N)$ and for each sync operation is $O(1)$. The time required to monitor any shared memory location is $O(lgN)$. Moreover, the Parallel Nondeterminator shows good performance in the simulation. In summary, the Parallel Non-determinator represents a provably efficient strategy for detecting data races for shared-memory parallel programs.

Contents

List of Figures

Chapter 1

INTRODUCTION

Shared-memory parallel programs are often designed to be deterministic, both in their final results and intermediate states. But debugging such programs requires a mechanism for locating race conditions. Data race occurs when two concurrent execution threads access the same memory location and at least one access is a write to the location.

Three principal strategies have been proposed in previous research for detecting data races: static analysis, post-mortem analysis, and on-the-fly analysis [5-8]. The static technique intends to report all potential access anomalies during the parallel execution. Its drawback is that too many false positive errors are reported. Post-mortem records the log for reading and writing occurred in the program's execution and tries to find the access anomaly from them. For a large parallel program, the size of the log is likely to be very large. Thus, we propose a dynamic on-the-fly race detection mechanism called Parallel Nondeterminator to check for determinacy races during the parallel execution of a program with recursive spawn-sync parallelism.

The proposed Parallel Nondeterminator checks for determinacy races during the parallel execution of a program with recursive spawn-sync parallelism. It guarantees the race is detected in location L if and only if a determinacy race exists in location L. The efficiency of the race detection algorithm mainly depends on two aspects - how quickly the test of concurrency can be made among the threads and how many entries for each shared variable are required in the access history. In our Parallel Nondeterminator, we present a modified Nested Region (MNR) Labeling algorithm to test the concurrency relationship among the threads in the Cilk program execution. Through combining the features of MNR labeling and the concept of Least Common Ancestor in spawn-sync tree, we prove that keeping only two read and one write access records for each shared variable is sufficient for correct race detection. In the time and space complexity analysis of the algorithm, we show that it gives provably good performance for the race detection of parallel programs written in Cilk.

The report has been organized into the following sections. Section 1 to 4 was prepared by He Yuxiong and section 5 to 6 was prepared by Wang Junqing.

- Chapter 2: MNR labeling algorithm for concurrency test
- Chapter 3: Algorithm for access monitoring and race detection
- Chapter 4: Performance analysis
- Chapter 5: Simulation and Performance Evaluation
- Chapter 6: Conclusion

Chapter 2

MNR LABELING ALGORITHM

This section introduces MNR labeling algorithm and prove its correctness on the determination of the currency relationship among threads in the Cilk programs. The basic idea of MNR labeling is to assign each thread with an unique label and decide the concurrency relationship between any two threads by comparing their corresponding labels. The main challenge is to find out a correct and efficient approach to maintain and compare the labels.

MNR labeling algorithm adopts the idea of nested region in NR labeling scheme, which was developed by Y.K. Jun and K. Koh and used for Fork-Join structure in Parallel FORTRAN [1]. Our MNR labeling algorithm has several significant modifications and improvements compared with the NR labeling scheme:

1) NR labeling scheme can't be applied to Cilk structure directly. MNR was developed to have a good match with Cilk program structure.
2) The key theorem in NR labeling is incorrect, its proof is incorrect either. Here we give a systematic proof on the correctness of MNR labeling algorithm.
3) MNR labeling has improved performance compared with NR labeling scheme.

In this section, we will first describe the definition and properties of Cilk POEG which is the high-level program run-time structure MNR labeling is applied to. Then we will introduce the MNR labeling algorithm and show one example on the labeling scheme.

2.1 Cilk Program and its POEG

MNR labeling is applied in Partial Order Execution Graph (POEG) of the parallel program. POEG is a graph to represent the execution relationship among the threads. Each vertex of POEG stands for a task operation - spawn or sync, and each edge stands for a thread starting from the corresponding task operation.

Before formally defining Cilk POEG, we define some useful notations. In a POEG $G = (V, E)$, two threads x and y ($x \in E$, $y \in E$ and $x \neq y$) have relation $x \rightarrow y$ iff there is a directed path from x to y along edges in E. Similarly $x||y \equiv \neg(x \rightarrow y) \wedge \neg(y \rightarrow x)$ is true iff there is no directed path from x to y or from y to x along edges in E.

Definition 1 defines the Cilk POEG unit, while definition 2 gives a constructive definition on the Cilk POEG in terms of the series and recursive composition of Cilk POEG units.

Definition 1 (Cilk POEG unit) *A Cilk POEG unit is the basic construct of Cilk POEG $G = (V, E)$, it has*

- *One source vertex*
- *One sink vertex*
- *Zero or more spawn vertices*
- *One sync vertex after all the spawn vertices if the number of spawn vertex is not zero*
- *Merge sync vertex and sink vertex if sync vertex exists*
- *Edges corresponding to the threads in Cilk program*

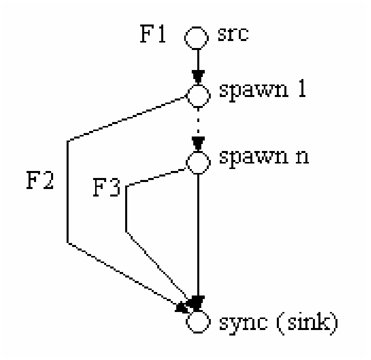


Figure 2.1: The Graph of Cilk POEG Unit

The Cilk POEG unit corresponds to a Cilk program

- It has only one sync operation in each function and the sync operation happens after all the spawn operations in the function.
- Its maximum recursive depth is equal to 1.

Inside a Cilk POEG unit, the thread between the spawn 1 operation and sync operation represents a basic function. A basic function in POEG is a function represented by single thread. For example, F_2 and F_3 are basic functions, while F_1 is not.

Definition 2 (Cilk POEG: Constructive definition of Cilk POEG) *Cilk POEG is the recursive composition of the series of Cilk POEG units.*

Series of Cilk POEG units is the series composition of Cilk POEG units with the extended sink nodes. Let's say $G_1, G_2...G_k$ are Cilk POEG units, the series of Cilk POEG units is a new Cilk POEG $G(V, E)$. It is constructed by linking G_1, G_2 to G_k in series through merging vertices V_{snk_i} with $V_{src_{i+1}}$ $(i = 1, 2..k-1)$ and adding V_{snk} with a new edge (V_{snk_k}, V_{snk}).

Recursive composition is to replace the basic function F with the series of Cilk POEG units G_s by merging the spawn operation of the function with the source of G_s and merging the sync operation of the function with the sink of G_s. The action of replacement is called **expanding the basic function** F. We denote the expanded function as $G(F)$.

Terms extended from the basic definition of Cilk POEG and the context of Cilk language:

- Function F_p is said to be the **parent** of function F_c if F_c is spawned directly from F_p . It is saying that Fc is a **child** of F_p . Similarly the **ancestor** and **descendant** of a function are also based on the spawn relation.

- **Initial thread of a function** is the thread connecting to the spawn operation of the parent of the function. The initial thread of the main function is the edge connecting to V_{src} of the POEG. **Return thread of a function** is the thread edge connecting to the sync operation of its parent. Return thread of main function is the edge connecting to V_{snk} of the POEG. Basic function has only one thread which is both the initial and return thread of a function. **Sync thread of a function** is the thread edge connecting to sync operation V_{sync}.

- **Sub-graph** $G(F)$ of a function F is the union of the edges (threads) and the vertices (operations) in the function F and all of its descendant functions. Initial and return thread of G(F) are equivalent to the initial and return thread of F.
- $F(T)$ represents the function which has the thread T.

2.2 Properties of Cilk POEG

This section shows the basic properties of Cilk POEG. These properties can be derived through simple observations of Cilk POEG. They are helpful in the correctness proof of MNR labeling algorithm

Property 1 (Property of Cilk POEG) *After we expand on a basic function F, $G(F)$ denotes the sub-graph of the expanded function F, two threads T_i and T_j , $T_i \notin G(F)$ and $T_j \notin G(F)$, T_i and T_j has the concurrency relationship*

$$\begin{cases} NR(T_i) \rightarrow NR(T_x) & \textit{if } T_i \rightarrow T_j \textit{ before expanding} \\ NR(T_i) || NR(T_x) & \textit{if } T_i || T_j \textit{ before expanding} \end{cases}$$

Property 2 (Property of Sub-graph Internal Path) *Given T_i and T_j are two threads, $G(F)$ is the sub-graph of F, $T_i \in G(F) \wedge T_j \in G(F)$, $T_i \rightarrow T_j$, the directed path from T_i to T_j does not pass through any thread outside the $G(F)$.*

Property 3 (Property of Sub-graph External Path) *Given $G(F)$ is the sub-graph of the function F, suppose $T_i \notin G(F) \wedge T_j \in G(F)$. If $T_i \rightarrow T_j$, the directed path from T_i to T_j go through the initial thread of $G(F)$. If $T_j \rightarrow T_i$, the directed path from T_j to T_i go through the return thread of $G(F)$.*

Property 4 (Property of Sub-graph initial and return thread) *Given thread T_i in the sub-graph $G(F)$ of a function F, T_i is neither $initial(F)$ nor $return(F)$, then*
$(initial(F) \rightarrow T_i) \wedge (T_i \rightarrow return(F))$

Property 5 (Property of Functional Relationship) *Given two different functions F_i and F_j , there are only three kinds of relationships between them in Cilk POEG structure:*

1) F_i is serial to F_j iff the return thread of F_i happens before the initial thread of F_j

2) F_i is parallel with F_j

Given any two threads T_i and T_j , $T_i \in F_i \wedge T_j \in F_j$

$F_i || F_j \Longrightarrow T_i || T_j$

3) F_i is an ancestor or descendant of F_j

2.3 MNR Labeling Algorithm

MNR labeling algorithm is illustrated in this section. Before the algorithm, some basic notations used in the algorithm are explained as below.

2.3.1 Notations in Algorithm

They are three types of notations: label notations, relationship notations and other notations used in the MNR labeling algorithm

Label notations are the symbols used in the label of the thread.

Label Notations:

- T is a thread
- $NR(T)$ is the basic NR label of the thread T, which is represented as a pair $\langle \alpha, \beta \rangle$ and $\alpha \leq \beta$. It is also called by the nested range of the thread.
- λ is the **sync counter** which represent the number of the sync threads of T in the critical path from initial thread to T. If T is a sync thread, the number includes T.
- $NR_{\lambda}(T)$ is a pair of λ and $NR(T)$, which is represented as $[\lambda, \langle \alpha, \beta \rangle]$
- r is the recursive depth of thread plus one
- $NR^{+}_{\lambda}(T)$ is to denote $[r, NR_{\lambda}(T)]$

Take an example in figure 2.2 in section 2.3.4:

The label of T_1 is $NR^{+}_{\lambda}(T_1) = [1, 1, \langle 1, 50 \rangle]$. The parameters involved are α_1, β_1, λ_1 and r_1.

$\alpha_1 = 1,\ \beta_1 = 50,\ \lambda_1 = 1, r_1 = 1$

The basic label of T_1 is $NR(T_1) = \langle 1, 50 \rangle$

Relationship notations represent the relationship of two labels.

Relationship Notations

- $NR(T_i) \odot NR(T_j) \equiv (\alpha_i \leq \alpha_j \wedge \beta_j \leq \beta_i) \vee (\alpha_j \leq \alpha_i \wedge \beta_i \leq \beta_j)$ **Include Relation**

- $NR(T_i) \triangle NR(T_j) \equiv (\alpha_i \leq \beta_j \wedge \alpha_j \leq \beta_i)$ **Overlap Relation**
- $NR(T_i) \diamond NR(T_j) \equiv (\alpha_i > \beta_j \vee \alpha_j > \beta_i)$ **Disjoint Relation**

Other Notations

- $Pn(T)$ is the scale of basic label calculated by $\beta - \alpha + 1$
- $RD(T)$ returns the recursive depth of the thread T plus one, which is equal to r
- $RD(F)$ returns the recursive depth of the function F plus one
- $NR(F)$ denotes the **nested range** of a function F. $NR(F) = NR(T)$ where T is the initial thread of the function F.
- I_G^X represents the access interleaving for a shared variable X by threads whose run-time concurrency relationship is modeled by a Cilk POEG $G = (V, E)$. I_G^X consists of a total-ordered sequence of accesses on X.

Definition 3 (One-way root history) *The one-way root history, denoted by $OH(T_i)$, is an ordered list of $NR_\lambda(T_x)$ in an ascending order of λ_x , where T_x is a one-way root of T_i . A one-way root of T_i is the most recent sync thread happening before T_i in each level of its ancestor functions or $F(T_i)$. $OH(T_i)$ stores one-way roots with total number less than or equal to $RD(T_i)$. If T is a sync thread, its one-way roots include itself as the last entry.*

In figure 2.2, NR label of T_7 is given by $[2, 2, \langle 1, 25\rangle] \mid [2, \langle 1, 50\rangle]$. The one-way root history of T_7 is given by $OH(T_7) = [2, \langle 1, 50\rangle]$. The thread labeled with $[2, \langle 1, 50\rangle]$ is corresponding to T_6 in the POEG. So T_6 is one of the one-way roots of T_7 . It is also the only one-way root of T_7 in this case. T_6 is the nearest sync thread in the parent function of the thread T_7 .

Definition 4 (Nearest one way root Ψ_j^i) *Given two threads T_i and T_j , $\lambda_i < \lambda_j$, the nearest one-way root of T_j to T_i , denoted by Ψ_j^i , is the thread whose sync counter is the smallest one in $OH(T_j)$ such that it is greater than λ_i .*

Take an example in figure 2.2, $\lambda_2 = 1,\ \lambda_7 = 2,\ \lambda_2 < \lambda_7$. T_7 has only one one-way root $[2, \langle 1, 50\rangle]$ whose sync counter is greater than the sync counter of T_2 . So $\Psi_7^2 = [2, \langle 1, 50\rangle]$.

2.3.2 Algorithm

The complete NR label of the task consists of $NR^{+}_{\lambda}(T_i)$ and $OH(T_i)$, which is denoted by $NR^{+}_{\lambda}(T_i) \mid OH(T_i))$. The algorithm of MNR labeling is shown below

Initialize Cilk Main:

$$\alpha_0 = 1 \tag{2.1}$$

$$\beta_0 = \ max\ integer \tag{2.2}$$

$$\alpha = 1 \tag{2.3}$$

$$\beta = \ max\ integer \tag{2.4}$$

$$\lambda = 1 \tag{2.5}$$

$$r = 1 \tag{2.6}$$

$$OH(T) = [1, \langle \alpha, \beta \rangle] \tag{2.7}$$

Cilk Spawn:
For the initial thread in the child function

$$r_c = r_p + 1 \tag{2.8}$$

$$\alpha_c = \alpha_p,\ \alpha_{c0} = \alpha_c \tag{2.9}$$

$$\beta_c = \alpha_p + \lfloor \frac{\beta_p - \alpha_p}{2} \rfloor,\ \beta_{c0} = \beta_c \tag{2.10}$$

$$\lambda_c = \lambda_p \tag{2.11}$$

$$OH(T_c) = OH(T_p) \tag{2.12}$$

For the preceding thread in the parent function

$$\alpha_p = \alpha_p + \lfloor \frac{\beta_p - \alpha_p}{2} \rfloor + 1 \tag{2.13}$$

$$Keep\ other\ parameters\ unchanged \tag{2.14}$$

Cilk Sync:

$$\lambda_p = \lambda_p + 1 \tag{2.15}$$

$$\alpha_p = \alpha_{p0},\ \beta_p = \beta_{p0} \tag{2.16}$$

$$For\ each\ child\ returned$$

$$\lambda_p = max(\lambda_p, \lambda_c + 1)$$

$$End\ For \tag{2.17}$$

$$After\ all\ children\ return,\ check\ last\ item\ in\ OH(T_p)$$

$$If\ Pn(OH_{last}(T_p)) = Pn(T_p), overwrite\ the\ last\ item\ by\ NR_{\lambda}(T_p)$$

$$OH_{last}(T_p) = NR_{\lambda}(T_p) \tag{2.18}$$

$$Else\ Add\ NR_{\lambda}(T_p)\ after\ the\ last\ item$$

$$OH_{last+1}(T_p) = NR_{\lambda}(T_p) \tag{2.19}$$

$$Keep\ other\ history\ items\ unchanged$$

The algorithm can be explained as below:

1) At the start of the program, assign the maximum integer range to nested range of the initial thread. Treat the initial thread as a sync thread so the history entry of initial thread is itself. $\langle \alpha_0, \beta_0 \rangle$ denotes the **nested range of function**. The range value is never changed since the first thread of the function is initialized.
2) Whenever there is a spawn operation, separate the nested range of the incoming thread into two halves. The first half is assigned to the thread in the child function. The second half is assigned to the preceding thread in the parent. The recursive depth of the thread in the child function is the depth of that in parent function plus one. Furthermore, the history entries in parent are copied into the thread in the child function.
3) Whenever there is a sync operation, the outgoing thread T_i of the sync operation will restore its nested range by the initial value $\langle \alpha_0, \beta_0 \rangle$ of the function $F(T_i)$. Its sync counter will be the maximum sync counter value of the incoming threads plus one. Furthermore, if its last history entry not is the sync thread in $F(T_i)$, it will add its own label at the end of the history entries. Otherwise, it will overwrite the last entry by its own label.

Formula for Concurrency Test

Given two access A_i and A_j on a shared variable X by threads T_i and T_j respectively, A_i precedes A_j in I_G^X , T_x is Ψ_j^i if $\lambda_i < \lambda_j$, $T_i || T_j$ is equivalent to

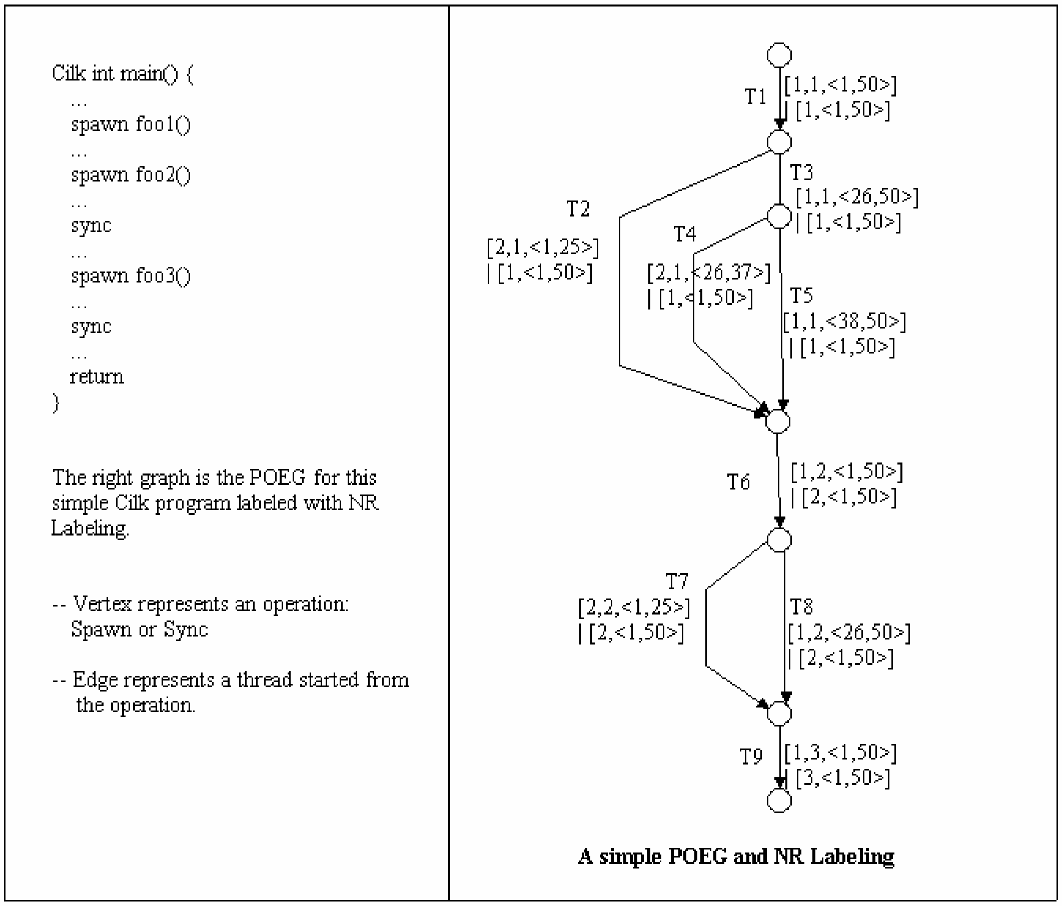
$$\begin{cases} NR(T_i) \diamond NR(T_x) & \text{if } \lambda_i < \lambda_x \le \lambda_j \\ NR(T_i) \diamond NR(T_j) & \text{if } \lambda_i = \lambda_j \\ true & otherwise \end{cases}$$

2.3.3 An example of MNR labeling

We present a simple example to illustrate the MNR Labeling in Cilk program as Figure 2.2.

From the observation, we know $T_2 \mid\mid T_4$. Since $\lambda_2 = \lambda_4 = 1$, $NR(T_2)$ should be disjoint with NR(T_4). Check the figure, $NR(T_2) = \langle 1, 25\rangle$ and $NR(T_4) = \langle 26, 37\rangle$. Their labels are disjoint.

Let's take another example, T_2 and T_8 . To decide their relationship, check their labels. Since $\lambda_2 = 1$ and $\lambda_8 = 2$, hence $\lambda_2 = \lambda_4$. $T_x = \Psi_8^2 = [2, \langle 1, 50\rangle]$. Since $\neg((T_2)\diamond NR(\Psi_8^2))$, T_2 is not parallel with T_8.



```
Cilk int main() {
   ...
   spawn foo1()
   ...
   spawn foo2()
   ...
   sync
   ...
   spawn foo3()
   ...
   sync
   ...
   return
}
```

The right graph is the POEG for this simple Cilk program labeled with NR Labeling.

-- Vertex represents an operation: Spawn or Sync

-- Edge represents a thread started from the operation.

Figure 2.2: Example of the Cilk POEG and MNR Labeling Algorithm

2.3.4 An Issue on Label Extension

MNR labeling algorithm tests the relationship of the threads by comparing the relationship of their nested regions. The initial thread is assigned the region $\langle 1, max\ integer\rangle]$. The spawn operation divides the nested region of the incoming thread into two halves and assigns them to two outgoing threads. But if the region of the incoming thread T_i is not dividable any more such as $\alpha_i = \beta_i$, what does the algorithm do? At this situation, the MNR labeling algorithm will extend the region with one more integer value. For example, $T_i = \langle 46, 46\rangle$ in function F_i and the maximum integer is 1024. After the spawn operation, the two outgoing threads will have nested region value $\langle 46 - 1, 46 - 512\rangle$ and $\langle 46 - 513, 46 - 1024\rangle$. The extension will still guarantee the correct comparison of the labels. Once the function F_i does sync operation again, the proceeding thread will restore the initial value of the function and the extension will be removed automatically.

In the situation that the function has a large number of spawn operations before the next sync operation, the label extension may be required. Fortunately, the extension is only applied to the labels whose nested region is necessarily to be extended. It doesn't require the extension of all the labels. And in general, the region initial value $\langle 1, max\ integer\rangle$ is a pretty large range and the label extension is not so common. So in the latter sections of the report, we discuss the MNR labeling algorithm without concerning the issues of label extension.

2.4 Correctness Proof of MNR Labeling Algorithm

This section proves that the MNR labeling algorithm can correctly decide the concurrency relationship among threads given their labels. We show some basic properties of the algorithm before the correctness proof.

2.4.1 Basic Properties of MNR Labeling Algorithm

This section presents some basic properties, which can be derived directly from the algorithm. These properties are very helpful in the correctness proof.

Property 6 (Property of Ψ_j^i Existence) *Given two threads T_i and T_j with $\lambda_i < \lambda_j$,Ψ_j^i always exists.*

The last item T_x in $OH(T_j)$ satisfies $\lambda_x = \lambda_j$ based on 2.17 and 2.18 in the MNR labeling algorithm. So $\lambda_x > \lambda_i$. It indicates there is at least one thread in $OH(T_j)$ whose sync counter is greater than λ_i . So Ψ_j^i always exists when $\lambda_i < \lambda_j$.

Property 7 (Property 1 of MNR Labeling) *After we expand on any basic function F, given its sub-graph G(F) and any thread T_i such that $T_i \notin G(F)$,*
$NR(T_i) = \langle \alpha, \beta \rangle$ *before the expansion* $\iff$ $NR(T_i) = \langle \alpha, \beta \rangle$ *after the expansion.*

From property of Cilk POEG, we know expanding one function does not change the concurrency relationship of the threads outside the sub-graph of the function. From property 1 of the MNR labeling, we know that the basic labels of threads do not change either. These two properties are applied widely in our proof.

Property 8 (Property 2 of MNR labeling) *Given function F and F', and F is an ancestor of F', T_i is a thread in F' and T_s is a sync thread in F, $NR(T_s) \supseteq NR(T_i)$*

Property 2 of MNR labeling indicates that the nested range of the sync thread in the ancestor function is including the nested range of any thread in the descendent function.

Property 9 (Property 3 of MNR labeling) *Given two threads $T_i \to T_j$, $\lambda_i \leq \lambda_j$*

Property 3 of MNR Labeling indicates that λ value is monotonically increasing along the directed path in Cilk POEG.

Lemma 1 $NR(T_i) \odot NR(T_j) \Longrightarrow NR(T_i) \triangle NR(T_j)$

Proof Given two threads T_i and T_j , $NR(T_i) \odot NR(T_j) \equiv (NR(T_i) \subseteq NR(T_j)) \vee (NR(T_i) \supseteq NR(T_j))$ based on the definition of include operation.

$$\begin{aligned}
&NR(T_i) \subseteq NR(T_j) \\
&\Longrightarrow (\alpha_i \leq \beta_i) \wedge (\alpha_j \leq \beta_j) \wedge (\alpha_j \leq \alpha_i) \wedge (\beta_i \leq \beta_j) \\
&\Longrightarrow (\alpha_i \leq \beta_j) \wedge (\alpha_j \leq \beta_i) \\
&\equiv NR(T_i) \triangle NR(T_j)
\end{aligned}$$

Similarly, we can get $NR(T_i) \supseteq NR(T_j) \Longrightarrow NR(T_i)\triangle NR(T_j)$

So we have the implication $NR(T_i)\odot NR(T_j) \Longrightarrow NR(T_i)\triangle NR(T_j)$ □

2.4.2 Correctness Proof of MNR labeling Algorithm

Given the labels of any two threads and the formula for the currency test, we say MNR labeling algorithm is correct if the result of the concurrency test of the two threads is the same as the real concurrency relationship between them.

The main theorem we are going to prove is the concurrency test formula in MNR labeling algorithm. Before that, we will prove lemma 2 to lemma 5. Lemma 2 shows the labels of two parallel threads are disjoint. Lemma 3 to 5 shows the relationship between their labels when the two threads are serial.

Lemma 2 *Given two threads T_i and T_j , $T_i||T_j \Longrightarrow NR(T_i)\diamond NR(T_j)$*

Proof The proof follows the construction of Cilk POEG and has three steps.

1. Prove $T_i \mid\mid T_j \Longrightarrow NR(T_i)\diamond NR(T_j)$ when T_i and T_j are in the Cilk POEG Unit
   When there is no spawn operation in G_0 , there does not exist any parallel threads. When there is only one spawn operation in G_1 , let's denote the two spawn threads by T_i and T_j . We know $T_i \mid\mid T_j$ and T_i and T_j are the only parallel pairs in G. From 2.9, 2.10 and 2.13 in the MNR labeling algorithm, we know that when the parent does spawn operation, the generated two parallel threads have disjoint labels. So $T_i \mid\mid T_j \Longrightarrow NR(T_i)\diamond NR(T_j)$

   Suppose $T_i \mid\mid T_j \Longrightarrow NR(T_i)\diamond NR(T_j)$ is true when there are n spawn operations. We will show that the implication is still true when there are $n+1$ spawn operations.

   From the following graph, we can see that thread T_x is replaced by the Cilk POEG Unit G_1 with one spawn operation when G_n is converted to G_{n+1} . Based on the position of T_i and T_j , we do analysis it in three cases:

   (a). T_i and T_j are in G_1

   We have shown that $T_i \mid\mid T_j$ is true in G_1

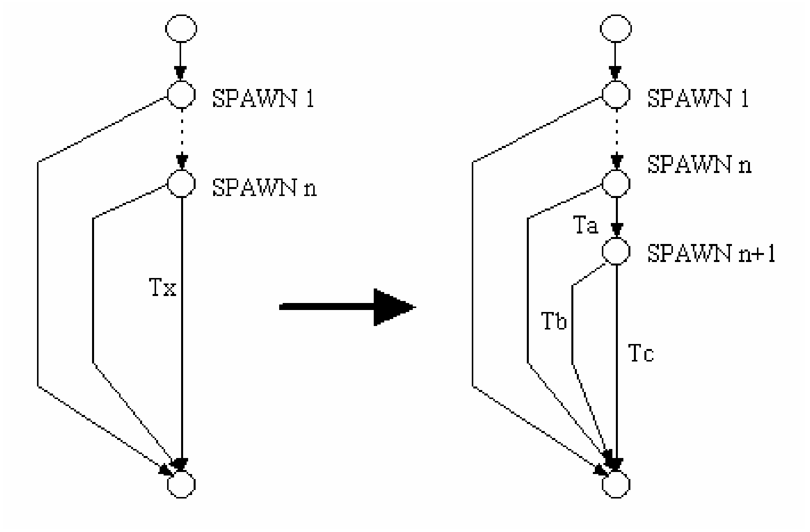


Figure 2.3: The Transform of Cilk POEG Unit from N-Spawn G_n to (N+1)-Spawn G_{n+1}

(b). T_i and T_j are outside G_1

The concurrency relationship between T_i and T_j doesn't change. $NR(T_i)$ and $NR(T_j)$ does not change. $T_i \mid\mid T_j \Longrightarrow NR(T_i) \diamond NR(T_j)$ is true in G_n , so $T_i \mid\mid T_j \Longrightarrow NR(T_i) \diamond NR(T_j)$ is true in G_{n+1} .

(c). Only one of T_i and T_j is in G_1 .

Without loss generality, let's assume T_j is in G_1 . T_i is parallel with T_j in G_{n+1} iff T_i is parallel with T_j in G_n .

$$T_i || T_j \ in \ G_{n+1} \Longrightarrow T_i || T_x \ in \ G_n \Longrightarrow NR(T_i) \diamond NR(T_j) \ in \ G_n$$
$$NR(T_x) \supseteq NR(T_j) \Longrightarrow NR(T_i) \diamond NR(T_j)$$

So when T_i and T_j are in the Cilk POEG Unit,$T_i \mid\mid T_j \Longrightarrow NR(T_i) \diamond NR(T_j)$.

2. Prove $T_i \mid\mid T_j$ when T_i and T_j are in the series of Cilk POEG Unit

   For the series of POEG unit, it is the series composition of Cilk POEG units. If T_i and T_j are in different Cilk POEG unit, they are always in serial. The parallel thread pairs only exist in every Cilk POEG unit. From step (1), we know the implication is correct in Cilk POEG unit. So the implication is also true in the series of Cilk POEG Unit.

3. Prove $T_i \mid\mid T_j$ in general Cilk POEG

   Cilk POEG is the recursive composition of the series of Cilk POEG units. In base case, Cilk POEG is just a series of Cilk POEG units. From the second step, we know that the implication is true.

Suppose we have Cilk POEG G in which $T_i \mid\mid T_j \Longrightarrow NR(T_i) \diamond NR(T_j)$. Let's prove when we do recursive composition of any basic function F in G, $T_i \mid\mid T_j \Longrightarrow NR(T_i) \diamond NR(T_j)$ still holds for the new POEG G'. Based on the position of T_i and T_j , we have three cases

(a). T_i and T_j are in $G(F)$ which is the sub-graph of the function F
$G(F)$ is a series of Cilk POEG units, the implication is true.

(b). T_i and T_j are outside $G(F)$
The concurrency relationship of T_i and T_j is not changed. $NR(T_i)$ and $NR(T_j)$ are not changed either. The implication is correct before the function expanding, so it is also correct after that.

(c). Only one of T_i and T_j is in G(F)
Without loss generality, let's assume T_j is in $G(F)$. T_f is the thread in basic function F before expanding.

$$T_i || T_j \textit{ in } G' \Longrightarrow T_i || T_f \textit{ in } G \Longrightarrow NR(T_i) \diamond NR(T_f) \textit{ in } G$$
$$(NR(T_f) \supseteq NR(T_j)) \wedge (NR(T_i) \diamond NR(T_f)) \Longrightarrow NR(T_i) \diamond NR(T_j)$$

So in any Cilk POEG, $T_i || T_j \Longrightarrow NR(T_i) \diamond NR(T_j)$. □

Lemma 3 *Given three threads T_x , T_i and T_j , $\lambda_i < \lambda_j$, T_x is Ψ_j^i*
$T_i {\rightarrow} T_j \Longrightarrow NR(T_i) \odot NR(T_x)$

Proof Given $\lambda_i < \lambda_j$, we know T_x always exists based on the property of Ψ_j^i existence. The proof follows two cases: 1) T_i and T_j are in the same functions, 2) T_i and T_j are in the different functions.

1. T_i and T_j are the threads in the same function F
Since $\lambda_i < \lambda_j$ and $T_i \rightarrow T_j$, there must be at least one sync threads between T_i and T_j in function F. The nearest one to T_j denoted by T_y is the root of T_j in level $RD(T_j)$. Since T_i and T_j are in the same function F, T_i can't reach any other roots of T_j in the directed path from T_i to T_j based on the serial path property given T_i and T_j are in $G(T_j)$. So the only

root of T_j , which is in the directed path of T_i and T_j , is T_y . So $T_x = T_y$. T_y is a sync thread in function F and T_i F, we know

$$NR(T_x) \supseteq NR(T_i) \Longrightarrow NR(T_x) \odot NR(T_i)$$

2. T_i and T_j are the threads in different functions F_i and F_j respectively
If T_i and T_j are in function F_i and F_j and $T_i \rightarrow T_j$, there is a directed path from T_i to T_j . Information can flow from F_i to F_j . Since Cilk program is the recursive composition of the series of Cilk POEG unit, the relationship of F_i and F_j is either Ancestor-Descendant or they have common ancestor function.

(a). F_i is descendant of F_j
There is at least one sync thread in F_j along the directed path between T_i and T_j , otherwise the control flow cannot return to F_j from its descendent F_i . The nearest one to T_j among those sync threads (denoted by T_y) is the root of T_j in the level $r = RD(T_j)$. The path between T_i and T_j passes through T_y . The directed path from T_i to T_j does not pass through any other roots of F_j with level $r > RD(T_j)$, based on the serial path property given that T_i and T_j are in the G(F_j). Since T_y is the only root in the path between T_i and T_j , $T_x = T_y$. Since T_x is the sync thread in F_j , which is the ancestor of F_i , $NR(T_x) \supseteq NR(T_i) \Longrightarrow NR(T_x) \odot NR(T_i)$ based on the property 2 of the NR labeling algorithm.

(b). F_i is ancestor of F_j
There are two sub cases:
When there is at least one sync thread in F_i along the path between F_i and F_j , among them the nearest sync thread T_y to T_j is the root of T_j in the level $r = RD(F_i)$. T_y is also the first root of T_j in the path between T_i and T_j , so $T_x = T_y$. Since T_y is the sync thread in F_i and $T_x = T_y$, $NR(T_x) \supseteq NR(T_i) \Longrightarrow NR(T_x) \odot NR(T_i)$.

When there is no sync thread in F_i along the path between F_i and F_j , there is at least one spawn operation between T_i and T_x . After spawn, $NR(T_i)$ is divided. $NR(T_x)$ is in the sub-branch of $NR(T_i)$ before next sync in F_i . $NR(T_i) \supseteq NR(T_x) \Longrightarrow NR(T_x) \odot NR(T_i)$

(c). F_i and F_j have common ancestor function
Denote F_x as the least common ancestor function of F_i and F_j . Since F_i and F_j are not ancestor-descendant relationship and $T_i \rightarrow T_j$, the path from T_i to T_j must pass through at least one sync thread in F_x . It implies that T_j has root T_y at level $r = RD(F_x)$. Since F_x is the least common ancestor function of F_i and F_j , T_y is the nearest root of

T_j in the path from T_i to T_j . So $T_x = T_y$. Since $T_x = T_y$ is the sync thread in the least common ancestor of F_i and F_j ,$NR(T_x) \supseteq NR(T_i) \Longrightarrow NR(T_x) \odot NR(T_i)$

From the above three cases, we know $NR(T_x) \supseteq NR(T_i) \Longrightarrow NR(T_x) \odot NR(T_i)$ when T_i and T_j are in the different functions.

No matter T_i and T_j are in the same function or in the different functions, given three threads T_x , T_i and T_j , and T_x is Ψ^i_j , $\lambda_i < \lambda_j \wedge T_i \rightarrow T_j \Longrightarrow NR(T_i) \odot NR(T_x)$.

□

Lemma 4 *Given three threads T_x , T_i and T_j , and T_x is Ψ^i_j if $\lambda_i < \lambda_j$, $T_i \rightarrow T_j$ implies*

$$\begin{cases} NR(T_i) \odot NR(T_x) & \textit{if } \lambda_i < \lambda_x \le \lambda_j \\ NR(T_i) \odot NR(T_j) & \textit{if } \lambda_i = \lambda_j \\ false & \textit{otherwise} \end{cases}$$

Proof $T_i \rightarrow T_j$ implies two cases: 1) T_i happens before T_j without sync operations, 2) T_i happens before T_j with some sync operations.

Case 1:
Without sync operations between T_i and T_j , we know $\lambda_i = \lambda_j$ and there are only spawn operations along the directed path from T_i to T_j . With spawn operation, the label of the incoming thread will be divided into two disjoint sub-ranges which are assigned to the labels of the two outgoing threads respectively. With several spawn operations, NR(T_i) will be divided into several sub-ranges and NR(T_j) is in one of them. So we have $NR(T_i) \supseteq NR(T_j) \Longrightarrow NR(T_i) \odot NR(T_j)$

Case 2:
Since T_i happens before T_j with some sync operations, $\lambda_i < \lambda_j$ based on the algorithm. Given $T_i \rightarrow T_j$ and $\lambda_i < \lambda_j$, we have $NR(T_i) \odot NR(T_x)$ based on Lemma 3.

The two cases cover all the possible results given $T_i \rightarrow T_j$. When $\lambda_i < \lambda_j$, $T_i \rightarrow T_j$ never holds. □

Lemma 5 *Given three threads T_x , T_i and T_j , and T_x is Ψ^i_j if $\lambda_i < \lambda_j$, $T_i \rightarrow T_j$ implies*

$$\begin{cases} NR(T_i) \triangle NR(T_x) & \textit{if } \lambda_i < \lambda_x \le \lambda_j \\ NR(T_i) \triangle NR(T_j) & \textit{if } \lambda_i = \lambda_j \\ false & \textit{otherwise} \end{cases}$$

Proof Lemma 5 can be derived directly from lemma 4 and lemma 1. □

Theorem 1 is the main theorem in MNR labeling algorithm. It determines the relationship between two threads based on their labels. If sync counter of two threads are equal, their labels are disjoint is equivalent to they are parallel. If sync counter of two threads are not equal, we need to introduce the nearest one way root of the larger sync counter thread from the smaller sync counter thread.

Theorem 1 *Given four threads T_x , T_y , T_i and T_j , T_x is Ψ^i_j if $\lambda_i < \lambda_j$, T_y is Ψ^j_i if $\lambda_j < \lambda_i$, T_i $||$ T_j is equivalent to*

$$\begin{cases} NR(T_i)\diamond NR(T_x) & \textit{if } \lambda_i < \lambda_x \leq \lambda_j \\ NR(T_i)\diamond NR(T_j) & \textit{if } \lambda_i = \lambda_j \\ NR(T_j)\diamond NR(T_y) & \textit{if } \lambda_j < \lambda_y \leq \lambda_i \end{cases}$$

Proof The proof follows the two directions

1. ($\Longrightarrow$)

   There are three cases:

   (a). $\lambda_i < \lambda_x \leq \lambda_j$

   If T_x $||$ T_i , we can derive $NR(T_x) \diamond NR(T_i)$. Let's prove T_x $||$ T_i by contradiction. If T_x is not parallel with T_i , $T_x \rightarrow T_i$ or $T_i \rightarrow T_x$. If $T_x \rightarrow T_i$,$\lambda_x \leq \lambda_i$. It is contradicted with $\lambda_i < \lambda_x$ in the condition. Since T_x is the one-way root of T_j , $T_x \rightarrow T_j$. If $T_i \rightarrow T_x$, $T_i \rightarrow T_j$. It is contradicted with the condition T_i $||$ T_j . So we have T_x $||$ T_i . From Lemma 2, we have T_i $||$ $T_x \Longrightarrow NR(T_i)\diamond NR(T_x)$.

   (b). $\lambda_i = \lambda_j$

   From Lemma 2, $T_i||T_j \Longrightarrow NR(T_i)\diamond NR(T_j)$

   (c). $\lambda_j < \lambda_y \leq \lambda_i$

   It can be proved in similar way as that in case (a).

2. ($\Longleftarrow$)

   There are three cases

   (a). $\lambda_i < \lambda_x \leq \lambda_j$

   From the lemma 5, we know $T_i \rightarrow T_j \Longrightarrow NR(T_i)\triangle NR(T_j)$. The contrapositive of the implication is

$$\neg(NR(T_i)\triangle NR(T_j)) \Longrightarrow \neg(T_i{\rightarrow}T_j) \Longrightarrow (T_j{\rightarrow}T_i) \vee (T_i||T_j)$$

If $T_j \rightarrow T_i$, $\lambda_i \leq \lambda_j$. It is contradicted with the condition $\lambda_i < \lambda_x \leq \lambda_j$. So we have $T_i \mid\mid T_j$ in this case. Since $NR(T_i) \diamond NR(T_x) \equiv \neg(NR(T_i) \triangle NR(T_x))$. We get $NR(T_i) \diamond NR(T_x) \Longrightarrow T_i || T_j$

(b). $\lambda_i = \lambda_j$ and (c) $\lambda_j < \lambda_y \leq \lambda_i$

Case (b) and (c) can be proved in similar way as case (a) through the contrapositive of lemma 5.

□

We'd like to prove a corollary from theorem 1. The corollary can be directly applied in the MNR labeling algorithm for thread concurrency test.

Corollary 1 *Given two access A_i and A_j on a shared variable X by threads T_i and T_j respectively, A_i precedes A_j in I_G^X , T_x is Ψ_j^i if $\lambda_i < \lambda_j$, $T_i || T_j$ is equivalent to*

$$\begin{cases} NR(T_i) \diamond NR(T_x) & \text{if } \lambda_i < \lambda_x \leq \lambda_j \\ NR(T_i) \diamond NR(T_j) & \text{if } \lambda_i = \lambda_j \\ true & \text{if } \lambda_i > \lambda_j \end{cases}$$

Proof The proof has three cases to consider based on the relationship between λ_i and λ_j

(a) $\lambda_i < \lambda_x \leq \lambda_j$ and (b) $\lambda_i = \lambda_j$

The result here can be directly derived from theorem 1 in both directions

(c) $\lambda_i > \lambda_j$ We need to prove T_i is always parallel with T_j when $\lambda_i > \lambda_j$ given the conditions in the corollary. Let's prove by contradiction. Suppose T_i is not parallel with T_j , $T_i \rightarrow T_j \vee T_j \rightarrow T_i$. If $T_i \rightarrow T_j$, $\lambda_i \leq \lambda_j$. It is contradicted with $\lambda_i > \lambda_j$ in the condition. If $T_j \rightarrow T_i$, A_j must precedes A_i in I_G^X . It is contradicted with the condition that A_i precedes A_j in I_G^X. All the consequences from the supposition are not true. The supposition is not true. So $T_i \mid\mid T_j$. It means that $T_i \mid\mid T_j$ is always true when $\lambda_i > \lambda_j$. □

Until this point, we have completed the correctness proof of the MNR labeling algorithm. We showed that the two threads are parallel if and only if they are reported to be parallel through the comparison of their labels.

2.5 Left-of relation

We define a left of relation that gives a partial order of vertices in Cilk POEG. The access history algorithm described in chapter 3 requires a labeling scheme in which a left-of relation is defined.

The notation of left of relation is given by $\prec$

Definition 5 *Given two threads T_i and T_j ,*
$T_i \prec T_j \Longleftrightarrow \alpha_i < \alpha_j$

Theorem 2 shows that the left-of relation establishes a canonical ordering of relatives with respect to their shallowest least common ancestor (LCA).

Theorem 2 *Given three threads T_i , T_j and T_k , $T_i \mid\mid T_j \mid\mid T_k$,*

$$T_i \prec T_j \prec T_k \Longrightarrow LCA(T_i, T_k) = LCA(T_i, T_j, T_k)$$

Proof Let's denote F_i , F_j and F_k as the function including T_i , T_j and T_k respectively. Since $T_i \mid\mid T_j \mid\mid T_k$, we know the three functions are different.

Let's denote F_x as the least common ancestor of F_i and F_k . Since $T_i \mid\mid T_k$, NR(T_i)$\diamond$ NR(T_k). Given T_i $\prec T_k$ and $NR(F_x) \diamond NR(T_k)$, we have $\beta_i < \alpha_k$. So $NR(F_x) \supseteq \langle \alpha_i, \beta_j \rangle$.

Given $T_i \prec T_j$, $\alpha_i < \alpha_j$. Given $T_j \mid\mid T_k$ and $T_j \prec T_k$, $\beta_i < \alpha_k \leq \beta_k$. Since $NR(F_x) \supseteq \langle \alpha_i, \beta_j \rangle$, $NR(F_x) \supseteq NR(T_j)$.

Assuming F_x is not ancestor of F_j , they can be in one of the three relationships based on function relationship property.

1. F_x is serial to F_j
   $F_x \rightarrow F_j \Longrightarrow$ $return(F_x) \rightarrow F_j$
   Since F_x is $LCA(F_i, F_j)$, T_i and T_j are in subgraph of F_x . Based on subgraph initial and return thread property $F_i \rightarrow$ $return(F_x)$ and $F_j \rightarrow$ $return(F_x)$. From the transitivity of serial operation,$F_i \rightarrow$ $return(F_x) \rightarrow F_k$. It is contradicted with the given condition $F_i \mid\mid F_k$. So F_x is not serial to F_j .

2. F_x is parallel with F_j

   $F_x \mid\mid F_j \Longrightarrow NR(F_x) \diamond NR(F_j) \Longrightarrow NR(F_x) \bigcap NR(T_j) = \varepsilon$. It contradicts with $NR(F_x) \supseteq NR(T_j)$, since $NR(T_j) \neq \varepsilon$

3. F_j is the ancestor of F_x
   Given F_j is the ancestor of F_x and any thread T_x in sub-graph of F_x , we have $T_j \parallel T_x \implies T_x \prec T_j$. Since F_x is common ancestor of F_i and F_j , T_i and T_k are in the sub-graph of F_x . $T_i \prec T_j$ and $T_k \prec T_j$. It contradicts with the given condition $T_j \prec T_k$.

The assumption is never true, F_x is ancestor of F_j . Since $F_x = LCA(F_i, F_k)$ and F_x is ancestor of F_j , $F_x = LCA(F_i, F_j, F_k)$.

□

There is an important indication from theorem 2.
Given there are k parallel accesses to the variable X and two locations $left_x$ and $right_x$ for record keeping. The program sees k access from $F_1 = Access_1(X)$ to $F_k = Access_k(X)$ in any order.

- Initialize $left_x$ and $right_x$ to the value of the first access on X. Let's assume the value is F_i .
- For any other $access_k$
  If $Access_k \prec left_x$, Do $left_x = Access_k$
  If $right_x \prec Access_k$, Do $right_x = Access_k$

Finally, applying theorem 2, we can get $LCA(left_x, right_x) = LCA(F_1, F_2, Fk)$. It means that we are able to keep two parallel records which have the shallowest LCA.

Chapter 3

ALGORITHM FOR ACCESS MONITORING AND RACE DETECTION

What is the smallest number of entries kept in access history of each location so that the data race in the location can be detected if and only if there is a determinacy race exists. Obviously two entries (one for read, one for write) are not enough. In simple parallel program without nested parallel loop, three entries (two for read, one for write) should be enough. The algorithm just keeps any two parallel read records of each variable once there are at least two parallel read records.

This model is not applicable to the language like Cilk. Keeping any two parallel read accesses may miss some data races. If we can always keep the two parallel read accesses, which have the shallowst LCA (least common ancestor) in parent child spawn tree, we are able to detect the data race with only two read access records. The idea here is to keep the left-most and right-most parallel read access records for each shared location. After we presented this idea in the project proposal, we found that the idea of using left-of relationship to keep three access records for each variable has been adoped in previous research such as OS labeling scheme [2]. We found that the checkread and checkwrite procedures in OS labeling scheme for record maintaining and race checking can be applied in our Parallel Nondeterminator. We present the readcheck and writecheck procedures in section 3.1 and prove its correctness in section 3.2.

3.1 Algorithm

The access history for each variable keeps three records - Read_Left(RL), Read_Right(RR) and Write(W). The monitoring procedure is shown below [2].

```
Checkread(access_history, tlabel)
    If access_history.W $\|$ tlabel then
        Report a write-read data race
    End if
    If tlabel $\prec$ access_history.RL or access_history.RL $\rightarrow$ tlabel
        access_history.RL=tlabel
    End if
    If access_history.RR $\prec$ tlabel or access_history,RR $\rightarrow$ tlabel
        access_history.RR=tlabel
    End if
End checkread

Checkwrite(access_history, tlabel)
    If access_history.W $\|$ tlabel then
        Report a write-read data race
    End if
    If access_history.RL $\|$ tlabel or access_history.RR $\|$ tlabel
        Report a read-write data race
    End if
    access_history.W=tlabel
End checkwrite
```

3.2 Race Detection Correctness Proof

There are two critical factors for the correctness of the race detection: 1) the concurrency relationship determination algorithm (MNR labeling) must be correct, and 2) the left-of relation must be able to establish a canonical ordering of relatives with respect to their LCA. In theorem 1, we proved that the MNR labeling is able to correctly decide the concurrency relationship given the labels of any two threads. In theorem 2, we proved that the left-of relation establishes a canonical ordering of relatives with respect to their shallowest least common ancestor.

Here we would show that at least one data race will be reported using the checkread and checkwrite access history algorithm if any data race is present for a shared variable. Because the proof of theorem 3 to 6 is conducted in a similar way as the ones in OS labeling [2] once the correctness of the two critical factors are guaranteed, we will just sketch the outline of the proof and focus on the place with differences.

Theorem 3 *In a checked access interleaving I_G^X for a variable X and a Cilk POEG $G(V, E)$, checkwrite will report a data race for a write in I_G^X if it is logically concurrent with some earlier read in I_G^X.*

Proof Outline
Proof is conducted by contradiction.
Suppose , $r \in I_G^X$ precedes w in I_G^X , T_r and T_w are the threads executing r and w operation respectively, and $T_r \|T_w$, but checkwrite fails to report a data race. Denote the access history of X by $OH(X)$ and the three access records by $OH(X).RL$, $OH(X).RR$, $OH(X).W$

No race is reported when access operation is w

$$\Rightarrow (OH(X).RR \rightarrow T_w) \wedge (OH(X).RL \rightarrow T_w) \qquad (I1)$$

If $(T_r \rightarrow OH(X).RL) \vee (T_r \rightarrow OH(X).RR), T_r \rightarrow T_w$. Contradicted with the supposition.

$$\Rightarrow \neg(T_r \rightarrow OH(X).RL) \wedge \neg(T_r \rightarrow OH(X).RR) \qquad (I2)$$

T_r is not present in the read access history

$$\begin{aligned}&\Rightarrow \neg(OH(X).RL \rightarrow T_r) \wedge \neg(OH(X).RR \rightarrow T_r)\\ &\wedge\neg(T_r \prec OH(X).RL) \wedge \neg(OH(X).RR \prec T_r)\\ &\wedge((OH(X).RL = OH(X).RR) \vee (OH(X).RL\|OH(X).RR)) \qquad (I3)\end{aligned}$$

Combining I2 and I3,

$$\begin{aligned}&\Rightarrow (T_r\|OH(X).RL) \wedge (T_r\|OH(X).RR) \wedge \neg(T_r \prec OH(X).RL) \wedge (OH(X).RR \prec T_r)\\ &\Rightarrow (OH(X).RL\|T_r\|OH(X).RL) \wedge (OH(X).RL \prec T_r \prec OH(X).RR) \qquad (I4)\end{aligned}$$

Let $F_x = LCA(F(OH(X).RL), F(OH(X).RR))$. Set T_{sync} to be the closest common thread which happen after $OH(X).RL$ and $OH(X).RR$. T_{sync} must be the sync thread in F_x . We have $(OH(X).RL \rightarrow T_{sync}) \wedge$ $(OH(X).RR \rightarrow T_{sync}) \wedge (T_{sync} \rightarrow T_w)$. Based on theorem 2, $F_x = LCA(F(OH(X).RL)$, $F(T_r)$, $F(OH(X).RR))$, F_x is also the ancestor of $F(T_r)$. Since T_{sync} is the sync thread in the ancestor of $F(T_r)$, $(T_{sync} \rightarrow T_r) \vee (T_r \rightarrow T_{sync})$. If $T_{sync} \rightarrow T_r$, $(OH(X).RL \rightarrow T_{sync}) \Longrightarrow (OH(X).RL \rightarrow T_r)$. It is contradicted with the derivation I4. If $T_r \rightarrow T_{sync}$, $(T_{sync} \rightarrow T_w) \Longrightarrow (T_r \rightarrow T_w)$. It is contradicted with the supposition $T_r \| T_w$.

By showing a contradiction in every case to the supposition, the theorem is proved. $\square$

Theorem 4 *In a checked access interleaving* $\in I_G^X$ *for a variable* X *and a Cilk POEG* $G(V, E)$*, if any two writes in* $\in I_G^X$ *are logically concurrent, then checkwrite will report a data race.*

Proof Outline

Suppose w_1 precedes w_k in $\in I_G^X$, T_{w1} and T_{w2} are the threads executing w_1 and w_2 operation respectively, and $T_{w1} \| T_{w2}$.

If T_{w1} is present in $OH(X)$ at the access check of w_2 operation, data race will be reported.

If T_{w1} is not present in $OH(X).W$ at the access check of w_2 operation, there is a sequence of wa_i (i=1...k) write operation which overwrite the value of $OH(X).W$. If the race is not reported from T_{w1} to T_{w2} , $T_{w1} \rightarrow T_{wa_1} \rightarrow T_{wa_2} ... \rightarrow T_{wa_k} \rightarrow T_{w2}$. It is contradicted with $T_{w1} \| T_{w2}$. So the race will be reported. $\square$

Theorem 5 *In a checked access interleaving* I_G^X *for a variable* X *and a Cilk POEG* $G(V, E)$*, a data race will be reported if a read in* I_G^X *is logically concurrent with some earlier write in* I_G^X*.*

Proof Outline

Suppose w precedes r in I_G^X , T_w and T_r are the threads executing w_1 and w_2 operation respectively, and $T_w \| T_r$.

If T_w is present in $OH(X)$ at the access check of r operation, data race will be reported.

If T_w is not present in $OH(X)$.W at the access check of r operation, there is a sequence of wa_i $(i = 1 \ldots k)$ write operation which overwrite the value of $OH(X).W$. If the race is not reported from w to wa_k ,$T_w \to T_{wa_1} \to T_{wa_2} ... \to T_{wa_k}$. If no race is reported at the access check of r operation $T_{wa_k} \to T_r$. We have $T_w \to T_r$. It is contradicted with $T_{w1} \| T_{w2}$. So there will be some race reports from w to r. □

Theorem 6 *In a checked access interleaving I_G^X for a variable X and a Cilk POEG $G(V, E)$, at least one data race will be reported if there are any conflicting, logically concurrent access in I_G^X .*

Proof Outline

There are three cases of conflicting accesses to consider:

1) Read after write access by parallel threads.
   Race will be reported based on theorem 5.

2) Write after read access by parallel threads.
   Race will be reported based on theorem 3.

3) Write after write access by parallel threads.
   Race will be reported based on theorem 4.

Until this point, we have completed the correctness proof of our Parallel Nondeterminator. Our parallel nondeterminator will report at least one data race for each variable if there are any concurrent and conflicting accesses in the variable. The next section will analyze its performance. □

Chapter 4

PERFORMANCE ANALYSIS

In this section, we examine the efficiency of our Parallel Nondeterminator. Its efficiency depends on the following parameters.

- V: the number of shared variables
- T: the maximum parallelism
- N: the nesting depth
- P: the number of processors

Time Complexity

In each spawn operation to create a child function, the parent needs to adjust its label with complexity O(1) and assigns a label of size O(N) to the its child function. So the work for each spawn is O(N).

In each sync operation, the parent only needs to get the λ value of the child, its time complexity is O(1).

The worst case time to verify if a shared variable access is involved in data race is dependent on the number of access history entries for each shared variable and the time to test concurrency between one label and one entry. There are three access history entries for each shared variable in our Parallel Nondeterminator. For each entry, when its λ value is greater than or equal to the λ value of the label, complexity is O(1). Otherwise, it needs to use binary search to find the nearest root of the label respect to the entry. Since the maximum length of the label history is O(N). The worst case complexity to check the concurrency relationship between a thread label and an entry is O(log_2N). Since there is constant number of entries in access history for each shared variable, the worst case complexity for each shared variable access is O(log_2N).

There are two factors in our parallel nondeterminitor, which affects the critical path of the cilk program. First, the work increases in the threads due to spawn, sync and race checking overhead. The critical path will increase correspondingly. But this increase generally does not affect the level of parallelism of the program. The second factor is that the history entries for each shared variable are required to be locked in readcheck and writecheck to guarantee the consistency of the access history. The lock will reduce the level of parallelism when two threads are accessing the history of the same variable concurrently. So it will increase the length of critical path. The increase is dependent on the number of concurrent readcheck and writecheck in the same shared variable. The influences on critical path and level of parallelism are further tested and discussed in the experiments.

Space Complexity

Space complexity includes the space for shared variable access history and the space for thread labels. The space for shared variable access history = the number of entries per variable $\times$ the space per entry $\times$ the number of shared variables. Since the number of entries per variable is O(1), the space per entry is O(1) and the number of shared variables is V, the space for shared variable access history is O(V). The space for thread labels = label space for each thread $\times$ number of concurrent threads=O(N)$\times$ min(P,T)=O(min(NP, NT)). So the space complexity is O(V+ min(NP, NT)).

In summary, the time complexity is O(N) per spawn, O(1) per sync and O(log_2N) per shared variable access. The space complexity is O(V+ min(NP, NT)).

Chapter 5

SIMULATION AND PERFORMANCE EVALUATION

In this section, we design Cilk simulation to implement our MNR labeling algorithm and evaluate its performance based on our testing cases.

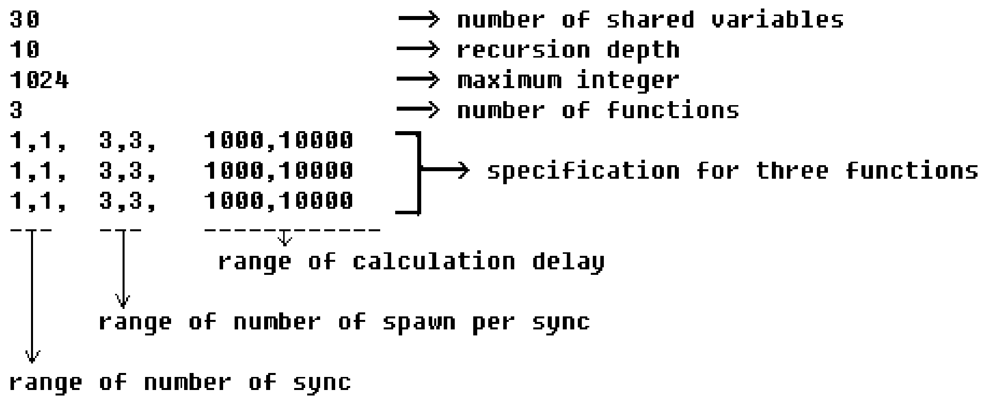
5.1 Simulation

We will introduce the simulation in this part. Our simulator has two main components: simulation program generator and nondeterminator runtime library. Generator is designed to automatically generate testing sample cilk programs. The nondeterminator runtime library implements the procedures of MNR labeling algorithm described in section 2 and race detection algorithm described in section 3. It supports labeling and read-write check operations.

5.1.1 Simulation Program Generator

Simulation Program Generator is designed to generate testing cases of cilk programs automatically. In each run, the generator will read the specification file and generate two cilk programs based on the specifications. One program is the normal cilk source, which is used as the reference in our performance evaluation. The other program is an instrumented cilk source which has the embedded nondeterminator routines. The two programs have the same source codes except that the latter one has additional routines to support parallel nondeterminator race checking. This design makes it possible to test the overhead introduced by race detection. Another important consideration for our generator design is to generate representative and meaningful testing cases. We try to generate source codes which cover a wide variety in terms of different function number, computation time, recursion depth, shared variable number, interleaving of operations, spawn and sync numbers.

Our specification file including the parameters is listed in Figure 5.1.



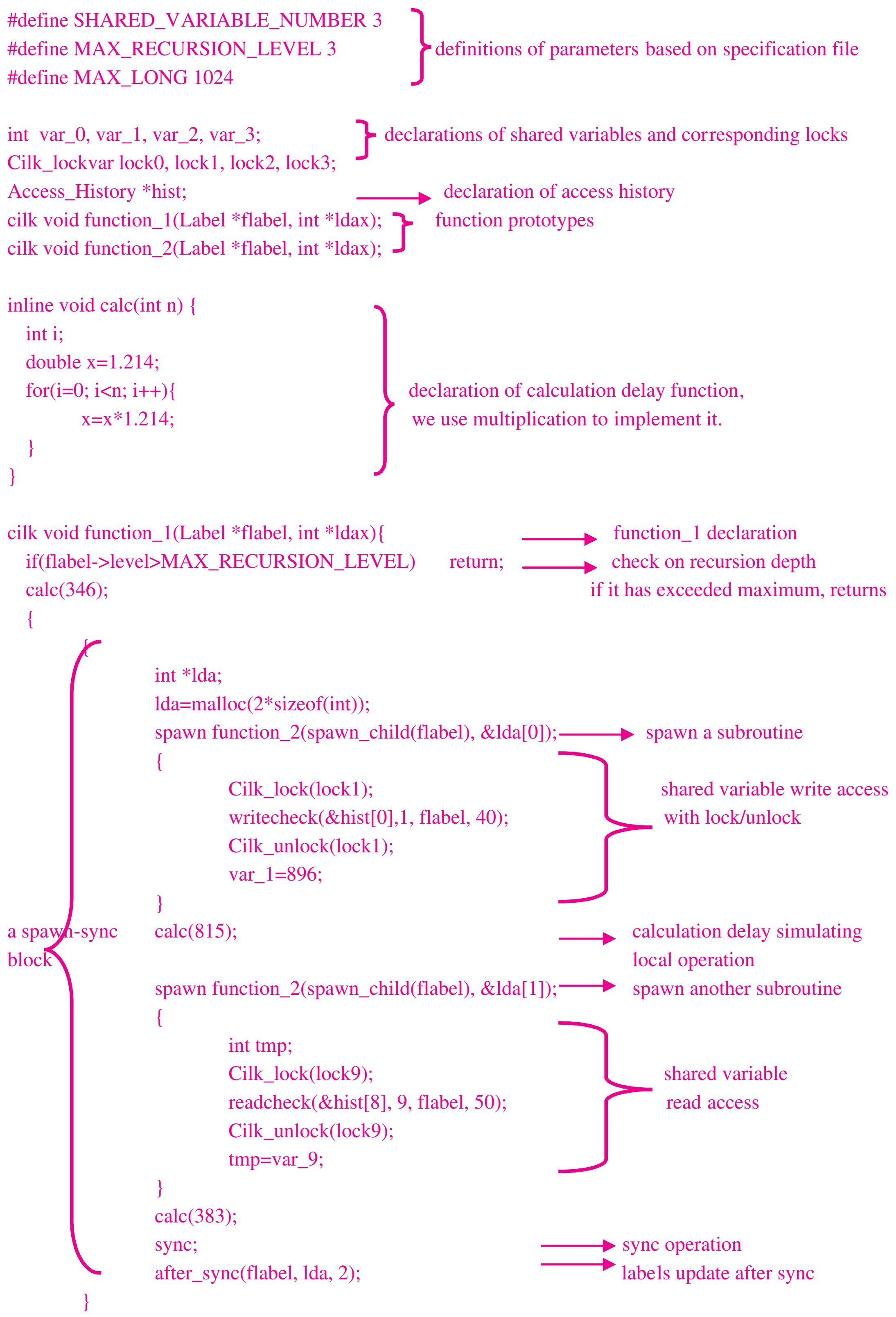
```
30
10
1024
3
1,1,  3,3,   1000,10000
1,1,  3,3,   1000,10000
1,1,  3,3,   1000,10000
```

number of shared variables
recursion depth
maximum integer
number of functions
specification for three functions
range of calculation delay
range of number of spawn per sync
range of number of sync

Figure 5.1: Sample specification file of Parallel nondeterminator

In the specification file, there are three parameters to define the features of the source program – the number of shared variables, the depth of recursion and the number of functions. In each function, there are three range values to define the features which are range of sync numbers, range of the number of spawns per sync and range of calculation delay. The calculation delay means to simulate the computation on the local variables. We use multiplication in our program to simulate the effect of local computation. In addition, d calculation delay stands for d-step of multiplications.

With the above specification, our generated file looks like: (the follows is the main part of the file)



```
#define SHARED_VARIABLE_NUMBER 3
#define MAX_RECURSION_LEVEL 3
#define MAX_LONG 1024

int  var_0, var_1, var_2, var_3;
Cilk_lockvar lock0, lock1, lock2, lock3;
Access_History *hist;
cilk void function_1(Label *flabel, int *ldax);
cilk void function_2(Label *flabel, int *ldax);

inline void calc(int n) {
   int i;
   double x=1.214;
   for(i=0; i<n; i++){
            x=x*1.214;
   }
}

cilk void function_1(Label *flabel, int *ldax){
   if(flabel->level>MAX_RECURSION_LEVEL)      return;
   calc(346);
   {
            {
                     int *lda;
                     lda=malloc(2*sizeof(int));
                     spawn function_2(spawn_child(flabel), &lda[0]);
                     {
                                 Cilk_lock(lock1);
                                 writecheck(&hist[0],1, flabel, 40);
                                 Cilk_unlock(lock1);
                                 var_1=896;
                     }
                     calc(815);

                     spawn function_2(spawn_child(flabel), &lda[1]);
                     {
                                 int tmp;
                                 Cilk_lock(lock9);
                                 readcheck(&hist[8], 9, flabel, 50);
                                 Cilk_unlock(lock9);
                                 tmp=var_9;
                     }
                     calc(383);
                     sync;
                     after_sync(flabel, lda, 2);
            }
```

definitions of parameters based on specification file
declarations of shared variables and corresponding locks
declaration of access history
function prototypes
declaration of calculation delay function,
we use multiplication to implement it.
function_1 declaration
check on recursion depth
if it has exceeded maximum, returns
spawn a subroutine
shared variable write access
with lock/unlock
a spawn-sync
block
calculation delay simulating
local operation
spawn another subroutine
shared variable
read access
sync operation
labels update after sync

```
        {
            int *lda;
            lda=malloc(2*sizeof(int));
            spawn function_2(spawn_child(flabel), &lda[0]);
            calc(665);
            spawn function_1(spawn_child(flabel), &lda[1]);
            {
                int tmp;
                Cilk_lock(lock2);                                 another spawn_sync block
                readcheck(&hist[1], 2, flabel, 68);
                Cilk_unlock(lock2);
                tmp=var_2;
            }
            calc(670);
            sync;
            after_sync(flabel, lda, 2);
        }
    }
    calc(373);
    *ldax=flabel->lda;
    free(flabel);
    return;                                          ——> function return
}
....
cilk int main(int argc, char *argv[]){               ——> declaration of main function
    Cilk_time tm_begin, tm_elapsed;
    Label *flabel;
    tm_begin=Cilk_get_wall_time();
    {
        int i;
        hist=malloc(SHARED_VARIABLE_NUMBER*sizeof(Access_History));
        for(i=0; i<SHARED_VARIABLE_NUMBER; i++){
            hist[i].rl.lda=1;
            hist[i].rl.alpha=1;
            hist[i].rl.belta=MAX_LONG;
            hist[i].rr.lda=1;
            hist[i].rr.alpha=1;                           initialize access history
            hist[i].rr.belta=MAX_LONG;
            hist[i].w.lda=1;
            hist[i].w.alpha=1;
            hist[i].w.belta=MAX_LONG;
        }
    }
```

```
   {
        flabel=malloc(sizeof(Label));
        flabel->alpha_o=1;
        flabel->belta_o=MAX_LONG;
        flabel->alpha=1;
        flabel->belta=MAX_LONG;
        flabel->level=1;
        flabel->lda=1;                                              initialize the labels
        flabel->rec_num=1;
        flabel->oh=malloc(sizeof(Record_History));
        flabel->oh->alpha=1;
        flabel->oh->belta=MAX_LONG;
        flabel->oh->lda=1;
        fprintf(stderr, "Complete label initialization\n");
   }
   {
        {
             int *lda;
             lda=malloc(2*sizeof(int));
             spawn function_1(spawn_child(flabel), &lda[0]);
             calc(513);                                               a spawn_sync block
             spawn function_1(spawn_child(flabel), &lda[1]);      the same as in the other functions
             calc(480);
             sync;
             after_sync(flabel, lda, 2);
        }
        ...
   }
   tm_elapsed=Cilk_get_wall_time()-tm_begin;
   printf( "time spent: %d \n",tm_elapsed);
   return 0;
 }
```

Basically, the generator will generate the source programs with the similar structure as shown above. Based on different specifications, the generated source programs will have different numbers of shared variables, functions and recursive depths. The functions will have different computation time, variable operations and spawn sync patterns. By means of that, our generated programs could cover a wide variety of cases in a random fashion.

5.1.2 Nondeterminator Runtime Library

Nondeterminator Runtime Library provides the runtime support of our race checking programs. It implements the procedures in MNR labeling algorithm such as label initialization, label update and concurrency relationship detection. It also implements the procedures in race detection algorithm as well, such as read check, write check and race report. During the execution of race checking Cilk program, those routines will be invoked to perform labeling and race detection operations.

5.2 Performance Evaluation

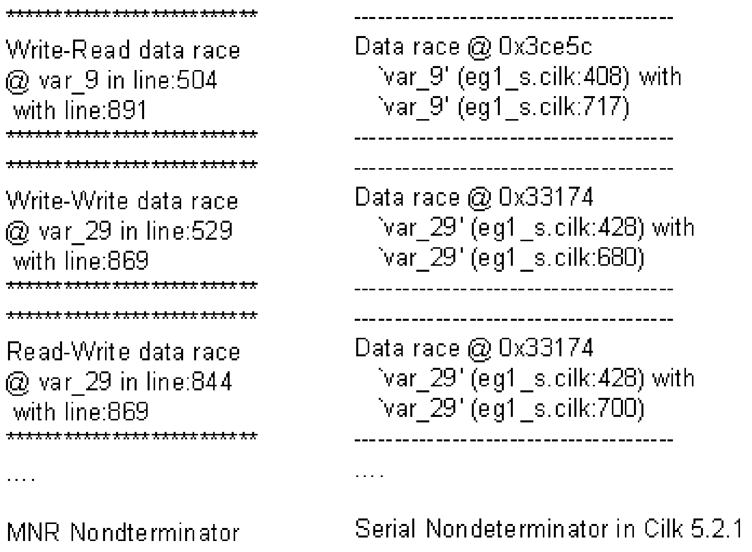
In this section, we will discuss the experimental results and evaluate the performance of our Parallel Nondeterminator. The testing is designed to fulfill two tasks - correctness testing and effectiveness testing.

5.2.1 Correctness Testing

The functionality of our Nondeterminator is to report the data races correctly during the parallel execution of the Cilk programs. So the first thing in the testing is to ensure the correctness of the implementation of our algorithms. We conducted a large number of testing for verifying the correctness. In smaller programs, we analyzed the reported races by hand. In larger programs, we compare the reported races with those returned by the serial nondeterminator in Cilk version 5.2.1. Our parallel nondeterminator performed correctly in all the testing cases. We will show one example of correctness testing as follows.

We conducted correctness test by comparing the results returned from our Parallel Nondeterminator simulator and from Cilk Nondeterminator in Figure 5.2. The left part of the results is obtained from our Parallel nondeterminator, and the right part is obtained from the serial Nondeterminator in Cilk version 5.2.1. In our Nondeterminator, race reports include three parameters: the type of data race, the name of shared variable and the line numbers of the conflicting variable access.

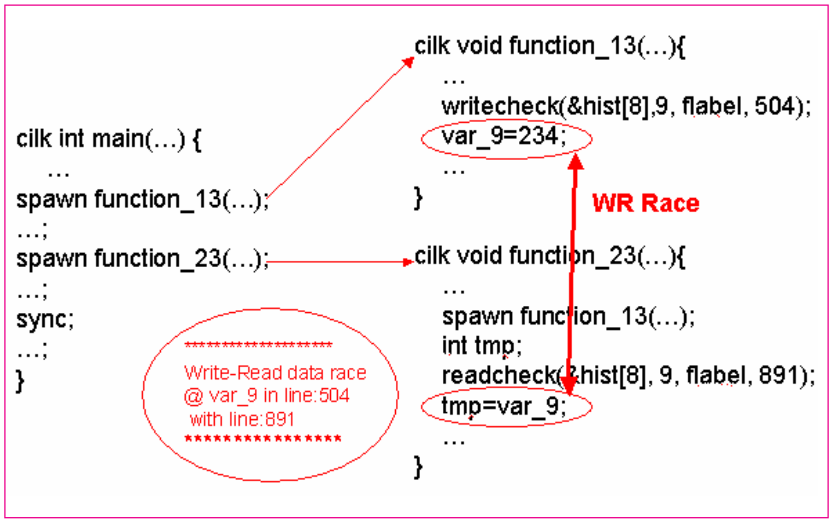
Figure 5.3 shows the testing program which generates the data race. The main function spawns to the function_13 first, in which there is a write operation on variable_9. Then the main function spawns the function_23, in which there exists a read operation on variable_9. We know there is a data race between these two operations. As expected, our Parallel Nondeterminator also reports a WR-type data race between line 504 and line 891 on variable_9.



```
***************************          ---------------------------------------
Write-Read data race                 Data race @ 0x3ce5c
@ var_9 in line:504                    `var_9' (eg1_s.cilk:408) with
 with line:891                         `var_9' (eg1_s.cilk:717)
***************************          ---------------------------------------
***************************          ---------------------------------------
Write-Write data race                Data race @ 0x33174
@ var_29 in line:529                   `var_29' (eg1_s.cilk:428) with
 with line:869                         `var_29' (eg1_s.cilk:680)
***************************          ---------------------------------------
***************************          ---------------------------------------
Read-Write data race                 Data race @ 0x33174
@ var_29 in line:844                   `var_29' (eg1_s.cilk:428) with
 with line:869                         `var_29' (eg1_s.cilk:700)
***************************          ---------------------------------------
....                                 ....

MNR Nondterminator                   Serial Nondeterminator in Cilk 5.2.1
```

Figure 5.2: Sample outputs of Parallel nondeterminator and Cilk Nondeterminator



```
cilk int main(...) {
    ...
spawn function_13(...);
...;
spawn function_23(...);
...;
sync;
...;
}

cilk void function_13(...){
    ...
    writecheck(&hist[8],9, flabel, 504);
    var_9=234;
    ...
}
cilk void function_23(...){
    ...
    spawn function_13(...);
    int tmp;
    readcheck(&hist[8], 9, flabel, 891);
    tmp=var_9;
    ...
}

********************
Write-Read data race
@ var_9 in line:504
 with line:891
****************
```

Figure 5.3: Sample Cilk Program with Parallel nondeterminator and potential data race

Figure 5.4 shows the running time of the cilk program in Figure 5.2. The program was tested with processor number from 1 to 32. According to the curve shown in the graph, the program with race check ran for about 0.225 seconds on 1 processor and ran for about 0.03 seconds on 12 processors. With parallel computation, its execution time reduced by around seven times of the time in serial. It introduces very small overhead compared with the program without race checking.

# of shared variables	30
recursion depth	5
# of functions	24
# of sync	1
# of spawn	3
range of calculation delay	1000 -10000 (steps)

Table 5.1: Specification of the test case

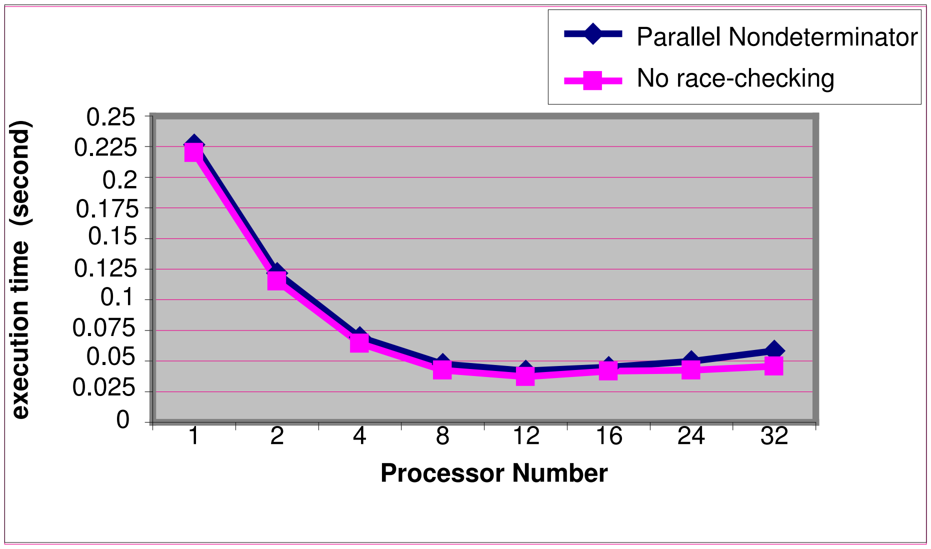


Figure 5.4: Sample execution time of Cilk program with parallel nondeterminator

5.2.2 Effectiveness Testing

Besides the correctness, another important factor is the effectiveness of the cilk programs with Parallel nondeterminator. We test the performance of our parallel nondeterminator in a wide variety of programs with the changed recursion depth and calculation delay. Furthermore, we also compare the critical path and parallelism of our program with the corresponding cilk program without race checking. All the tests were conducted on yggdrasil.lcs.mit.edu.

In our evaluation, we use execution time ratio as the basic criterion to evaluate the performance of Parallel nondeterminator. The ratio is defined as follows:

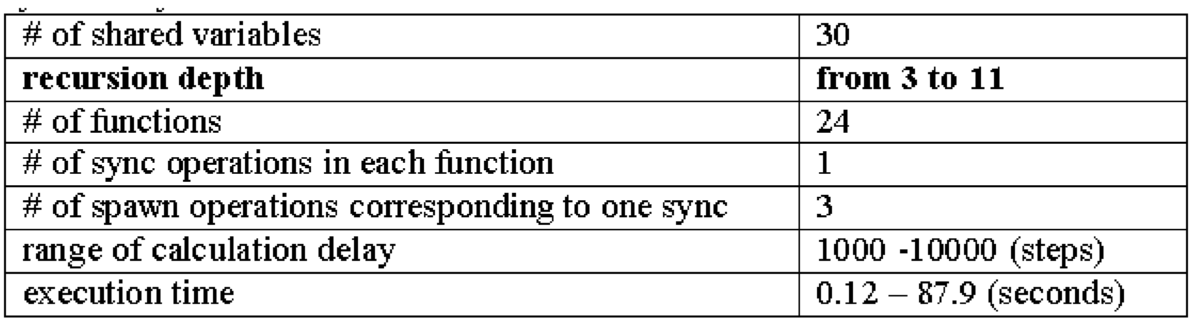
$$execution\ time\ ratio =$$

$$\frac{execution\ time\ of\ the\ Cilk\ program\ with\ race\ checking}{execution\ time\ of\ the\ corresponding\ Cilk\ program\ without\ race\ checking}$$

The program pairs in the comparison are generated from our simulation program generator. When the execution time ratio is close to 1, it indicates that the program with our race checking introduces very little overhead above the original program. It shows a good performance. Otherwise if the execution time ratio is much larger than 1, it indicates that a lot of overhead is resulted from the race checking.

Experiment on the change of recursion depth

Recursion depth is a critical factor which affects the performance of our MNR labeling algorithm based on the theoretical analysis. This set of experiments was designed to explore the influence of recursion depth on the performance of our nondeterminator from the aspect of experiments.



# of shared variables	30
recursion depth	**from 3 to 11**
# of functions	24
# of sync operations in each function	1
# of spawn operations corresponding to one sync	3
range of calculation delay	1000 -10000 (steps)
execution time	0.12 – 87.9 (seconds)

Table 5.2: Specification of the test cases

Figure 5.5 illustrates the performance curves under the change of recursion depth. X-axis is the number of processors and Y-axis is the execution time ratio. Seven curves represent seven different recursion depths from 3 to11 respectively.

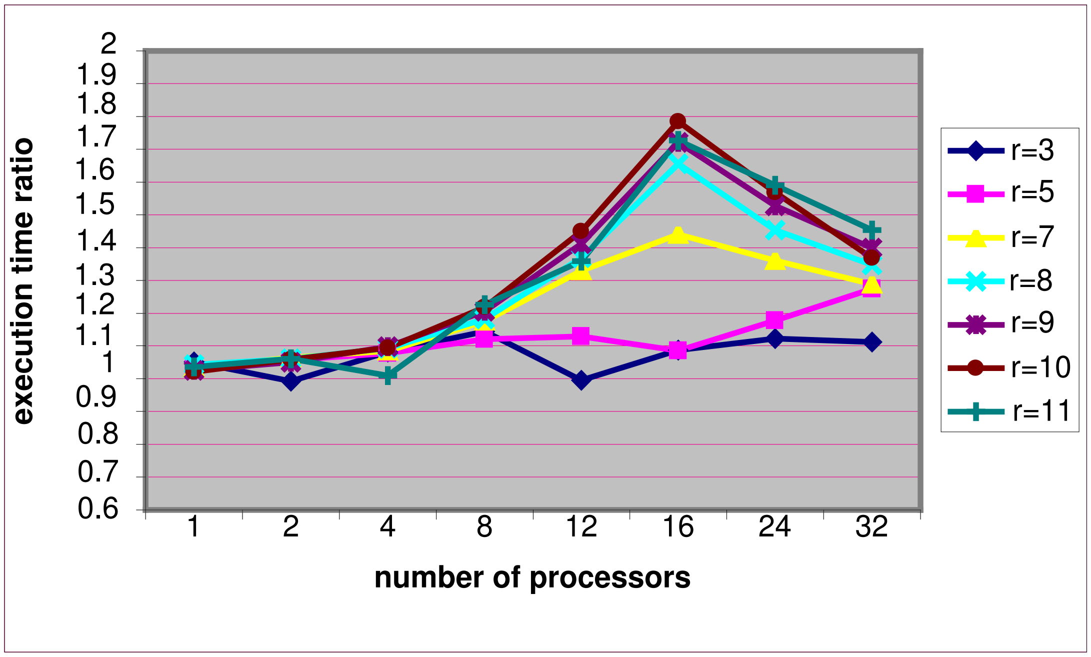


Figure 5.5: Performance ratios with change of recursion depth

In general, the performance ratios range from 1 to 1.8, and they are mainly located in the interval from 1 to 1.5. With small processor number (<8), the performance ratios are very close to 1. The increase of recursion depth does not have obvious influence on the performance. But when the processor number becomes larger (>=8), the increase of the recursion depth does make the performance drop.

The reasons can be explained as below:

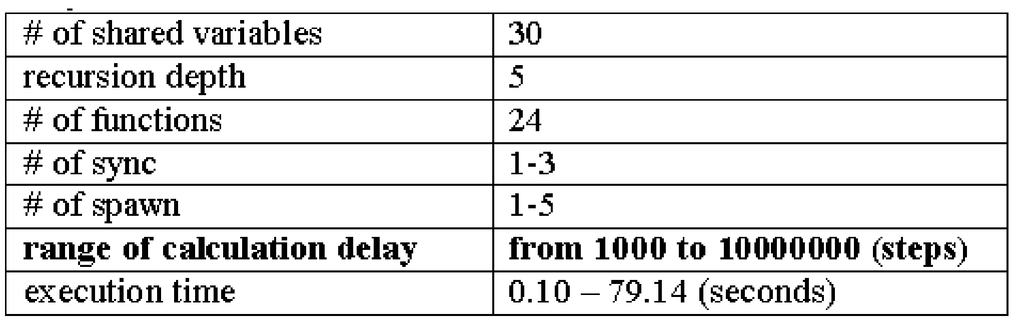
1) With the increase of recursion depth, the overhead introduced by race checking for each shared variable access increases correspondingly based on the performance analysis with worst case time complexity O(lgN). But in general, it only took one or two steps to find the relationship between each pair of labels as we tracked on the execution of the programs. So the performance of our parallel nondeterminator is still pretty good given the increase of the recursion depth.
2) When the processor number is large, the real parallelism tends to be large. As we showed in the sample program in section 5.1.1, the read and write check needs to lock the variable before the checking and unlock it after that. When the processor number is larger, the possibility

of conflict access on critical region of each variable becomes higher. The locks will reduce the parallelism of the race detection program and introduce some overheads. That is why the performance drops somewhat when the processor number becomes large.

In addition, there is a peak appearing around 16 processors. That is to say, the ratio comes to the largest at this point and the performance is worst at this point. Furthermore, it is not an occasional situation in our testing. In fact, a large section of our testing cases has the peak on 16 processors. It is pretty hard to explain why the worst case is in 16 processors but not 32 processors. It may be related to the interconnection of the ygg cluster.

Experiment on the change of calculation delay

The testing program uses calculation delay to simulate local operations. With the increase of calculation delay, the percentage of shared variable operations drops.



# of shared variables	30
recursion depth	5
# of functions	24
# of sync	1-3
# of spawn	1-5
range of calculation delay	**from 1000 to 10000000 (steps)**
execution time	0.10 – 79.14 (seconds)

Table 5.3: Specification of the test cases

Figure 5.6 displays the performance curves with different calculation delays. X-axis is the number of processors and Y-axis is the execution time ratio. Four curves represent four different ranges of calculation delay from 1000 to 1000000 respectively.

From Figure 5.6, we can see that the performance ratio is very close to 1 when the calculation delay is greater than 10000 steps of multiplication. When the calculation delay is smaller (<1000), the execution time ratio is higher, which indicates the drop of the performance.

Since calculation delay represents the local variable computations, its increase indicates that the percentage of operations on shared variables decreases. So the overhead introduced in the shared variable access takes less importance with respects to the total computation time. It explains why the performance is better when the computation delay is larger.

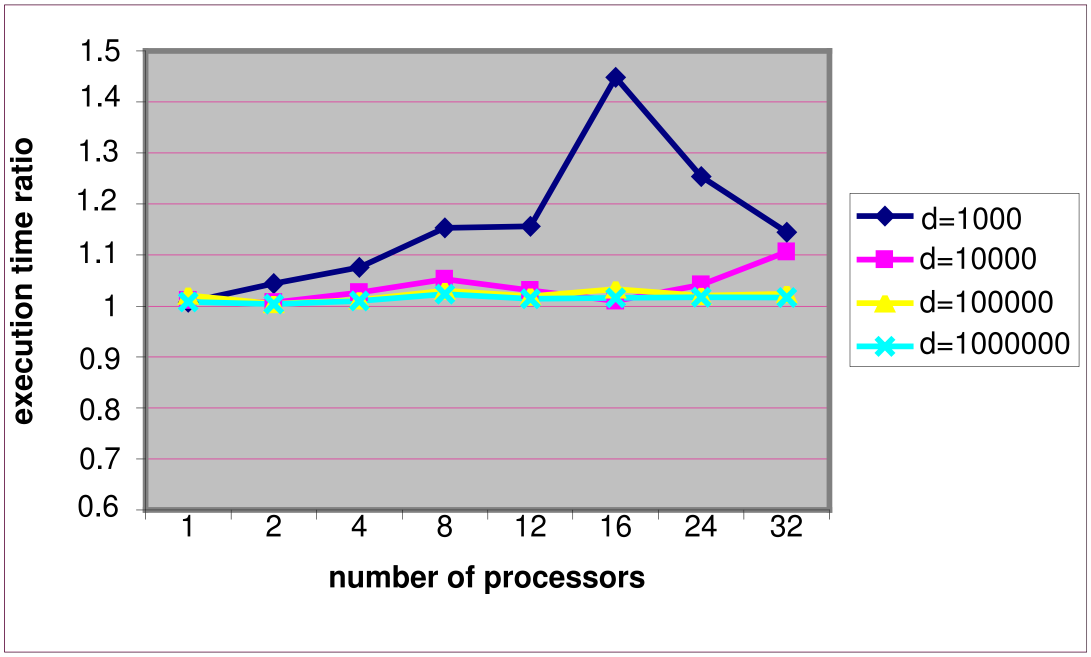


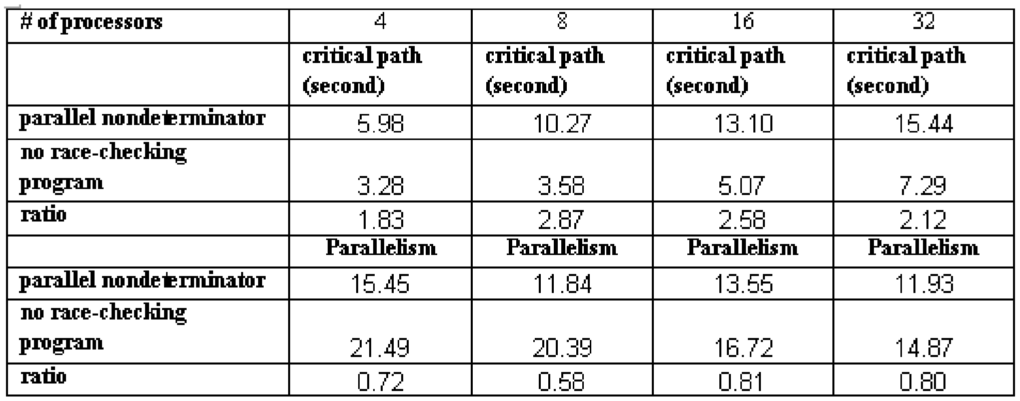
Figure 5.6: Performance ratios with change of calculation delay

Experiment on critical path and parallelism

This experiment is designed to compare the critical path and parallelism of the programs with and without race checking. The purpose is to explore the influence of race checking on the change of critical path and parallelism.

# of shared variables	30
recursion depth	5
# of functions	24
# of sync	1-5
# of spawn	1-8
range of calculation delay	1000 -10000 (steps)
# of processors	**from 4 to 32**

Table 5.4: Specification of the test cases



# of processors	4	8	16	32
	critical path (second)	critical path (second)	critical path (second)	critical path (second)
parallel nondeterminator	5.98	10.27	13.10	15.44
no race-checking program	3.28	3.58	5.07	7.29
ratio	1.83	2.87	2.58	2.12
	Parallelism	Parallelism	Parallelism	Parallelism
parallel nondeterminator	15.45	11.84	13.55	11.93
no race-checking program	21.49	20.39	16.72	14.87
ratio	0.72	0.58	0.81	0.80

Table 5.5: Testing results

Table 5.5 lists the experimental results of critical path and parallelism and the ratios between parallel nondeterminator and no race-checking program. We notice that the ratio of critical path ranges from 2 to 3, and the ratio of parallelism ranges from 0.5 to 1. The data tells us that the race check routines increase the critical path by one to two times of the original and the parallelism reduces to 0.7 times of the original in average.

There are two reasons for the increase of the critical path and the decrease of the parallelism level:

1) Our Parallel nondeterminator does some extra work than the no race-checking Cilk programs. For additional work - reporting data races, it needs to maintain an access history of the shared variables and check on every read or write operation for shared variables. The work increase on the threads along the critical path results in the increase of the critical path.
2) To ensure the consistency of the access history, we use locks to protect the shared variable access history in read check and write check. Those locks reduce the level of parallelism when several threads are doing read/write check concurrently on the same variable. Such a situation also results in the increase of the critical path.

Extreme case in experiments

This part shows an extreme case in our experiments. The race check introduces significant overhead to the parallel execution of the program.

Figure 5.7 illustrates the performance ratios on different numbers of processors. X-axis is the

# of shared variables	6
recursion depth	5
# of functions	10
# of sync	1
# of spawn	6
range of calculation delay	<10000 (steps)

Table 5.6: Specification of the test cases

number of processors and Y-axis is the execution time (second). Two curves stand for the execution time of the programs with race check and without race check.

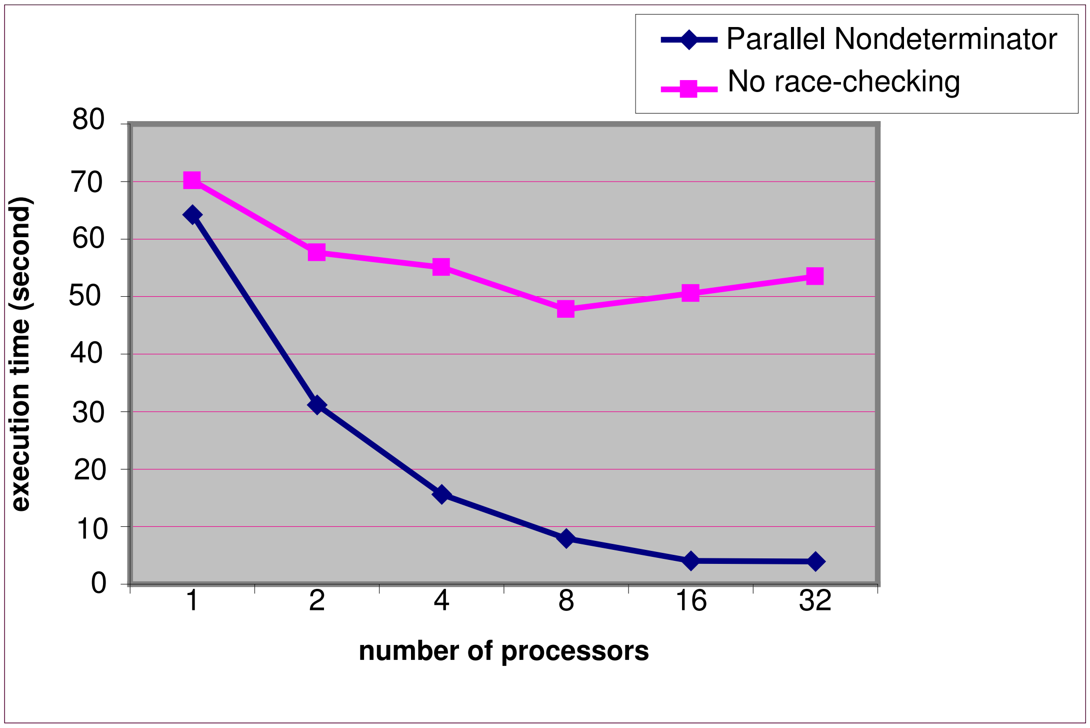


Figure 5.7: Bad case of the overhead on lock/unlock operations

From Figure 5.7, we can see that the execution time of the race check program in 32 processors is only two thirds of that in 1 processor. Furthermore, the race check program takes more than 10 times of execution time than the original program when both of them are running on 32 processors.

Why is the overhead of race check so large in this case? It is due to the combination effect of two important factors:

1) large number of shared variable access The local operation takes very small percentage of total computation time in the program of the example. The program has a lot of shared variable accesses and detects more than 200 data races.
2) Small number of shared variables As we declared before, our race check program adds lock and unlock before and after the read/write check to ensure the consistency of the access history. Since the lock for each share variable only allows one thread to use and update its history entries, the lock overhead will be extremely large when there are a lot of conflicting accesses on a small number of shared variables.

As a result, the performance of our parallel nondeterminator downgrades significantly.

5.2.3 Testing summary

The testing has successfully conducted the correctness and effectiveness test on our parallel nondeterminator.

In the correctness testing part, we analyze the data race reporting mechanism we used in our parallel nondeterminator and compare the results of data races between the cilk program with and without race check. In the meantime, the execution time of the cilk program with/without race check is shown in the figure. The results from the test case work as a sample show the correctness of our parallel nondeterminator.

In the effectiveness testing part, we evaluate the performance from three aspects: the change of recursion depth, the change of calculation delays and the influence on critical path and parallelism. We have listed and analyzed the results in details in the previous three sub sections. The check and lock operations in the shared variable access are the main sources to introduce overhead to our race check program. In most of our testing cases, the program with our race check only takes the execution time 1 to 1.8 times of the original program in both serial and parallel executions. It indicates that the overhead of our race check routines is pretty small and our parallel nondeterminator works well.

Chapter 6

CONCLUSIONS

Detecting data race is very important for debugging shared-memory parallel programs, because data races result in unintended nondeterministic execution of the program. Our MNR Labeling algorithm provides a dynamic on-the-fly mechanism to check for the determinacy races during the parallel execution of a Cilk program.

The efficiency of on-the-fly detection technique primarily depends on the time needed to determine whether a data race exists there and the space needed to store the necessary information. Our Parallel Nondeterminator has the time complexity – O(1) for each sync, O(N) for each spawn and O(lgN) for worst case access check. The space complexity is given by O(V + min (NT, NP)).

From the experimental results, we notice that regardless of the changes of the parameters, such as recursion depth, number of spawns, calculation delay and number of processors, the execution time ratio is generally within the range from 1 to 1.8 based on our testcases. These experimental results gave strong support for the efficiency of our Parallel Nondeterminator.

In summary, our Parallel Nondeterminator provides a provable and experimentally good performance for the race detection in the parallel execution of the programs written in Cilk.

Bibliography

[1] Y-K. Jun and K. Koh, "On-the-fly Detection of Access Anomalies in Nested Parallel Loops", in Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 107–117, 1993.

[2] John Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism, Supercomputing 91, pp 24-33, ACM/IEEE, Nov 1991

[3] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in Cilk programs," in Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97), (Newport, Rhode Island), June 22–25, 1997

[4] G.-I. CHENG, M. FENG, C. E. LEISERSON, K. H. RANDALL, AND A. F. STARK, Detecting data races in Cilk programs that use locks, in Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta, Mexico, June 1998

[5] Anne Dinning and Edith Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In Second ACM SIGPLAN Symposium on Principles Practice of Parallel Programming(PPOPP), pages 1–10, February 1990

[6] Dejan Perkovic and Peter J. Keleher. Online data-race detection via coherency guarantees. In Proceedings of the Second USENIX Symposium on Operating System Design and Implementation(OSDI'96), pages 47–58, October 1996

[7] Kim, J., and Y. Jun, "Scalable On-the-fly Detection of the First Races in Parallel Programs," 12nd Intl. Conf. on Supercomputing, pp. 345-352, ACM, July 1998

[8] Dejan Perkovic, Perter J. Keleher. A Protocol-Centric Approach to on-the-fly Race Detection. IEEE Transactions on Parallel and Distributed Systems,Vol 11, pp.58-69, 2000