Monitoring of Programs with Nested Parallelism using Efficient Thread Labeling

Ok-Kyoon Ha and Yong-Kee Jun
Department of Informatics
Gyeongsang National University
Jinju, Republic of Korea
{jassmin, jun}@gnu.ac.kr

Abstract

It is difficult and troublesome to detect data races occurred in an execution of parallel programs. On-the-fly race detection techniques use monitoring to threads and events, which access to shared variables. Any of them using Lamport's happened-before relation needs a thread labeling scheme for generating unique identifiers which maintain logical concurrency information of the parallel threads. Previous labeling schemes for on-the-fly race detection in parallel programs with nested parallelism depend on the nesting depth on every fork and join operation. This paper presents e-NR (Extended Nest Region) labeling in which every thread generates and maintains its label in a constant amount of time and space. Experimental results using OpenMP programs show that presented labeling detects data races more efficiently, because it requires reduced time overhead about 10% and it 3.2 times smaller than previous labeling scheme in average space for race detection.

Keywords: *thread labeling, data races, parallel programming, nested parallelism.*

1. Introduction

Data races [1]-[2] in parallel programs [3] is a kind of concurrency bugs that can be occurred when two parallel threads access a shard memory location without proper inter-thread coordination, and at least one of these accesses is a write. The races must be detected for debugging because they may lead to unpredictable and mysterious results. However, it is difficult and troublesome to detect data races in an execution of the parallel program. On-the-fly race detection techniques use monitoring method about threads and events, which relate to shared variables. Any of them [4]-[5] needs a representation of Lampot's happened-before relation [6] for generating unique identifiers which maintain logical concurrency information of parallel threads.

NR (*Nest Region*) *labeling* [7]-[8] which is an efficient thread labeling scheme supports the fork-join model of parallel execution with nested parallelism and generates concurrency information using nest regions for nesting threads. The efficiency of NR labeling depends only on the nesting depth N of a parallel program, because it creates and maintains a list of ancestors information for a thread on every fork and join operation. Thus, NR labeling requires $O(N)$ time complexity for creating and maintaining thread labels, and the storage space for the concurrency information is $O(V+NT)$ in worst case, where V is the number of shared variables in a parallel program.

Generally, the time required for the race detection techniques to be consumed consists of two factors. One is to determine races and maintain access histories in each access to shared variables. The other is to generate a label for each thread to determine races occurred in an execution of the parallel program. And the space also consists of two factors. One is to store access histories for all shared variables. The other is to maintain the concurrent thread labels. The time and space complexity of the thread labeling schemes participate actively in the efficiency of on-the-fly race detection. Therefore, we can reduce the time and space cost for race detection by improving the efficiency of the thread labeling scheme.

This paper presents an improved NR labeling, called e-NR (Extended Nest Region) labeling, which does not depend on the nesting depth of a parallel program. Thus, it requires a constant amount of time and space complexity. The basic idea is using a pointer for a thread label to refer to its ancestor list by inheritance or update. For the reference, we change the list of a thread label into a pointer which point to ancestor list to create thread labels. The storage space for the concurrency information is O(V+T), and the time to generate a unique identifier of each thread is O(1) in the worst case. Some experiments were performed on OpenMP programs with nesting depths of three or four and maximum parallelisms varying from 10,000 to 1,000,000. The results show that e-NR labeling is 5 times faster than the original NR labeling and 4.3 times faster than OS labeling in the average time for creating and maintaining the thread labels. In average space required for labeling, it is 3.5 times smaller than the original NR labeling and 3 times smaller than OS labeling.

This paper is organized as follows. Section 2 illustrates notion of nested parallel programs and our motivation. Section 3 presents e-NR labeling for the parallel programs with nested parallelism. In Section 4, we analyze the efficiency of the labeling scheme used for on-the-fly race detection in OpenMP programs. In the last section, we conclude the paper and present the future work.

2. Background

Parallel or multi-thread programming is a natural consequence of the fact that multi-processor and multi-core systems are already ubiquitous. This section illustrates the notion of nested parallelism and introduces our motivation which generates thread concurrency information in parallel programs for on-the-fly race detection.

2.1. Programs with Nested Parallelism

Parallel programs using OpenMP[9]-[10] which is a typical model for scalable and portable parallel execution contain loop level parallelism. It employs the simple fork-join execution model that makes the program efficiently with lower overhead for parallel threads. Nested parallelism is a nestable fork-join model of parallel execution in which a parallel region encounters another parallel region. This paper considers nested parallelism without inter-thread coordination in the OpenMP programs.

```
#pragma omp parallel for private( i )
for (i = 1; i <= 2; i++) {

    do something

    if (i % 2 == 0) {
#pragma omp parallel for private( j )
        for ( j = 1; j <= i; j++) {

            do something

        }
#pragma omp parallel for private( j )
        for ( j = 1; j <= i; j++) {
#pragma omp parallel for private( k )
            for ( k = 1; k <= j; k++) {

                do something

            }
        }
    }
}
```

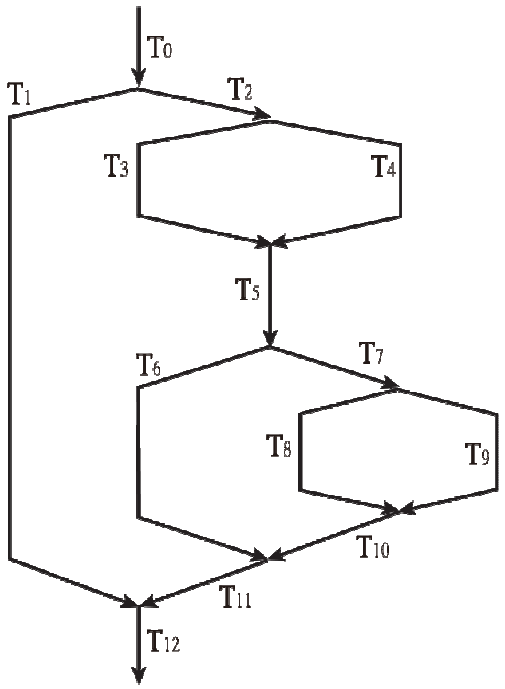


Figure 1. A nested parallel loop program and its POEG

An OpenMP program with nested parallelism can be contained many loop levels of a parallel region. In a nested parallel level L, the level is called an inner-most level if there is no other loop contained in a loop body. Otherwise, it is called an outer level. An individual loop can be enclosed by many outer levels in L. The nesting level of an individual loop is equal to one plus the number of eh enclosing outer levels. The nesting depth of L is the maximum nesting level of L. In a one-way nested loop of depth N, there is exactly one loop at each nesting level n, (n=1, 2, … , N). A loop is multi-way or m-way nested if there exists m disjoint loops in a nesting level, ($m \geq 1$).

For example, figure 1 shows a two-way nested parallel program of nesting depth three. If we remove the first nested loop indexed by j in the figure, the program becomes a one-way nested parallel program of nesting depth three. Let Ii denote the loop index of a loop Li, li and Ui denote the lower and upper bound of Ii respectively, and si denotes the increment of Ii. A loop Li is normalized if the values of both li and si are one. The si is optional when it is equal to one. To make our presentation simple, we assume that all parallel loops are normalized loops. Figure 1 shows a normalized loop with index j for which the lower bound is one, the upper bound is i, and the increment is one.

The graph in figure 1 represents an execution of the program shown in the figure through a directed acyclic graph called POEG (Partial Order Execution Graph). In this graph, a vertex means fork or join operation for parallel threads and an arc started from a vertex represents a thread started from the vertex. Using such a graph, we can easily figure out happened-before relation between any pair of threads, For example, the POEG in figure 1 shows a partial order on the threads in an execution instance of the two-way nested loop.

```
#pragma omp parallel for private( i )
for (i = 1; i <= 2; i++) {

    do something

    if (i % 2 == 0) {
#pragma omp parallel for private( j )
        for ( j = 1; j <= i; j++) {
            ...
            S++;
            ...
        }
    ...
}
```

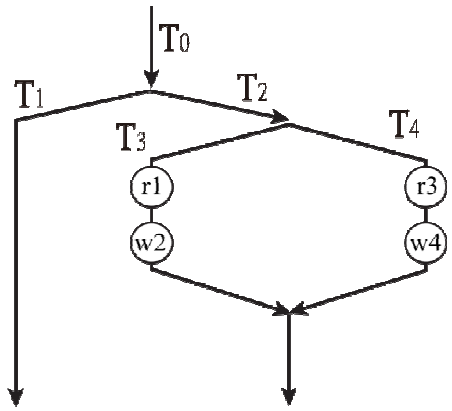


Figure 2. Example of data races in a nested parallel program

Data races in parallel programs are a kind of concurrency bugs that may occur when two parallel threads access a shared memory location without proper inter-thread coordination, and at least one of the accesses is a write. The parallel program may not have the same execution order with the same input, because thread execution order is timing-dependent. It is difficult for programmer to figure out when the program runs into a race. Unpredictable and mysterious results due to data races may be reported to the programmer, thus the races must be detected for debugging.

Figure 2 represents an execution of parallel program with nested parallelism. In this graph, the accesses r and w drawn in small disks upon the arcs name a read and a write access to a shared variable respectively. The numbers attached to accesses names indicate an observed order. Using such a graph, we can easily figure out happened-before relationship between a pair of accesses. For example, “r1” happened-before “w2” in Figure 2, because a path exists between these accesses. An access “r1” is concurrent with “r3” and “w4”, because any path does not exist between any pair of accesses. There exists a race, if any pair of concurrent accesses ei, ej includes at least one write without proper inter-thread coordination. We denote the race ei-ej. Thus, figure 2 includes three races: {r1-w4, w2-r3, w2-w4}.

2.2. On-the-fly Race Detection

We have much previous work which progresses some techniques for race detection. These techniques have focused on static and dynamic analysis. Static analysis techniques may lead too many false positives, because they report unfeasible races by analyzing only source code without any execution. Moreover, using these techniques for detecting feasible races in a program with dynamically allocated data is NP-hard [1].

Dynamic analysis includes trace based post-mortem techniques and protocol based on-the-fly techniques, which report only feasible races occurred in an execution of the parallel program. Post-mortem technique collects occurred accesses information into a trace file during an execution of a program. Then, this technique analyzes the traced information and reports feasible races. However, this approach is inefficient because it requires large time and space cost for tracing accesses and threads.

On-the-fly techniques based on two different methods: lock-set analysis and happened-before relation analysis. Lock-set analysis reports races of monitored program by checking violation of locks. This method is simple and can be implemented with low overhead. However, lock-set analysis techniques may lead to many false positives, because they ignores synchronization primitives which are non-common lock such as signal/wait, fork/join, and barriers [5]. Happened-before relation analysis uses a logical time stamp and a protocol for race detection. This method reports on-the-fly races between current access and maintained previous accesses by comparing their happened-before relation. Thus this method reports more precise results than lock-set analysis method. Mellor-crummey's protocol [4] and Dinning's protocol [14] are commonly used for race detection. This paper uses only the former protocol due to considering our program model.

Mellor-crummey's protocol guarantees to detect at least one race for each shared variable, if any exists. The protocol defines the structure of access history and maintaining policy. The access history consists of only three components: Left-Read, Right-Read, and Write. Each entry of the access history is a <label> for every monitored access. Left-Read and Right-Read components guarantee that maintained a read is occurred in left-most thread and right-most thread respectively. The concept of the left-most or right-most is well indexed for threads in fork-join execution model.

This protocol maintains the label to detect races by checking and updating the access history. When an access occurs to a shared variable, the policy for each entry is the following:

- ***Left-Read***: reports races with an access in Write compared using their label. A label of a read access is maintained, if the access is left-most read or ordered with a previous access in the Left-Read.
- ***Right-Read***: reports races with an access in Write compared using their label. A label of a read access is maintained, if the access is right-most or ordered with a previous access in the Right-Read.
- ***Write***: reports races with an access in all entries compared using their label. Only the last write access is kept in the Write.

In figure 2, we can detect that the races occur between T3 thread and T4 thread using this protocol. For example, when “r1” happens as the first read access to the shared variable in T3, the label is recorded into the two read entries of the access history for the shared variable. The access “w2” is recorded into the Write entry with its label. A race is reported by comparing with “w2” in the Write entry when the access “r3” occurs, because they do not satisfy happened-before relation. When “w4” of T4 occurs, three races {w2-r3, r1-w4, w2-w4} are finally detected between accesses {r1, w2} in T3 thread and accesses {r3, w4} in T4 thread.

2.3. Motivation

Lamport defined the notion of the happened-before relation [6]. The definition is applied to a partial ordering of threads that make up an execution instance of a parallel program.

This definition is the following:

- *If a thread T_i happened at an earlier time than a thread T_j, T_i happened before T_j, denoted by $T_i \rightarrow T_j$.*
- *Otherwise, a thread T_i concurrent with a thread T_j, denoted by $T_i \parallel T_j$.*

Happened-before relation analysis reports races of monitored program using a race detection protocol [4] which determines the logical concurrency between the current access and the previous conflicting accesses. For generating the concurrency information, the variations of vector timestamps [11]-[13] are presented to be used in happened-before relation analysis.

NR labeling [7]-[8] generates a unique identifier for every thread during the monitored execution of a parallel program. The labeling supports program model with nested parallelism and provides a constant-size label for each entry of an access history. This labeling generates concurrency information using nest regions for nesting threads, so that the complexity does not depend on the maximum parallelism of program. Because the labeling maintains an ordered list of ancestor information for each thread, a binary search method can be employed for the comparison of concurrency information. The efficiency of NR labeling depends only on the nesting depth N of a monitored parallel program. Thus, NR labeling requires O(N) time and storage space complexity for creating and maintaining thread labels.

In this paper, we present an improved NR labeling, called e-NR labeling, which does not depend on nesting depth of a parallel program. Our approach focused on improving the list of ancestor information, called one-way history OH. The basic idea is using a pointer for a thread label to refer its OH from the parent threads by inheritance or update, but does not regenerate. Thus, e-NR labeling provides a constant amount of time and storage space complexity for creating and maintaining thread labels. The efficiency of e-NR labeling can make on-the-fly race detection more practical.

3. Extended NR Labeling

The e-NR labeling consists of two main components: the one-way region *OR* and the one-way history *OH*. In this section, we present the one-way region which gives unique identifier to concurrent threads using the nest region and join counter. We also describe how our approach applied to the *OH*, and makes e-NR labeling efficiently for nested parallel programs.

3.1. NR Labels

The one-way region *OR* of a thread label is a pair of a join counter λ and a nest region $<\alpha, \beta>$, denoted by $[\lambda, <\alpha, \beta>]$ altogether. The join counter λ is the number of the joined ancestors of the thread in the critical path from the initial thread. The nest region consists of two integers $<\alpha, \beta>$ which mean a range of number space which has been divided by thread fork operations and concatenated by thread join operations. This one-way region can be used as thread identifier.

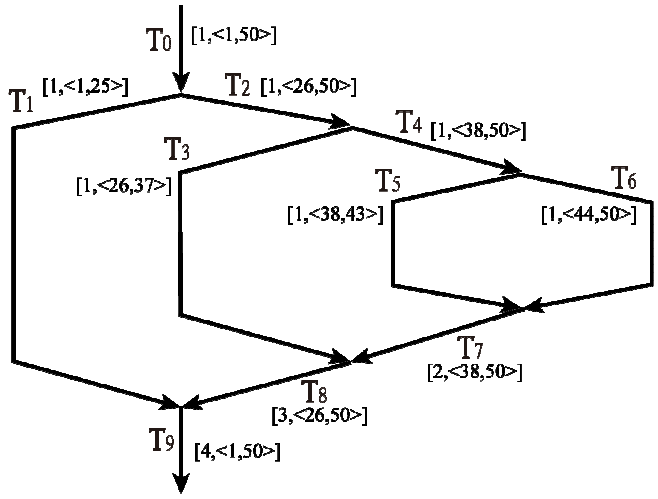


Figure 3. Example of NR labels for a one-way nested parallel loop program

Figure 3 shows an example of NR labels for one-way nested loop programs. The nest region of the initial thread T_0 is <1, 50> just for readability, although it is initialized with <1, *maxint*> in general where *maxint* is the maximum integer that can be represented in a machine. Because T_0 is assumed to be a joined thread, its joined counter λ_0 is one and then OR_0 becomes [1, <1, 50>]. The initial nest region is divided into two regions for the two child threads: <1, 25> and <26, 50>. The forked threads T_1 and T_2 copy the value of their λ from T_0. When a join operation occurs, the value of λ of the current *OR* is only changed to the maximum number of joined ancestors until the current thread. For example, OR_8 of the joined thread T_8 is [3, <26, 50>]. The nest region of T_8 is inherited from a previous thread in the same level under the same loop index. The join counter of T_8 is three, because its maximum joined critical path is $T_0 \rightarrow T_7 \rightarrow T_8$.

Definition 1 *Any labeled thread T_i has a nest region $<\alpha_i, \beta_i>$, denoted by $NR(Ti)$. If there exists two labeled threads T_i and T_j,*

$$NR(T_i) \sqsupseteq NR(T_j) \equiv (\alpha_i \leq \beta_j \wedge \alpha_j \leq \beta_i)$$
$$NR(T_i) \lesseqgtr NR(T_j) \equiv (\beta_i < \alpha_j \wedge \beta_j < \alpha_i) .$$

$NR(T_i) \sqsupseteq NR(T_j)$ means that $NR(T_i)$ overlaps with $NR(T_j)$. $NR(T_i) \lesseqgtr NR(T_j)$ means that $NR(T_i)$ does not overlap with $NR(T_j)$. In Figure 3, for example, it satisfies $NR(T_2) \sqsupseteq NR(T_7)$ and $NR(T_3) \lesseqgtr NR(T_7)$. We can see that any two threads are placed in the happened-before relation if they satisfy the overlapped relation.

Lemma 1 *Given two labeled threads T_i and T_j, $T_i \rightarrow T_j$ implies $NR(T_i) \sqsupseteq NR(T_j)$.*

Theorem 1 *Given two labeled threads T_i and T_j, $T_i \,||\, T_j$ is equivalent to $NR(T_i) \lesseqgtr NR(T_j)$.*

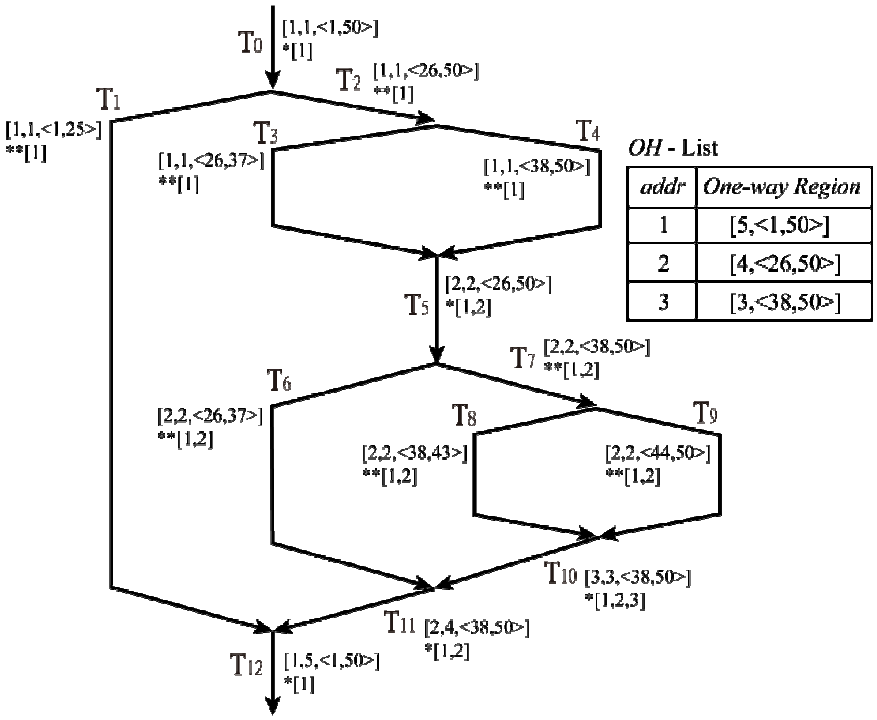


Figure 4. Example of e-NR labeling for program in figure 1

3.2. e-NR Labeling

The one-way history OH of a thread is an ordered list of the joined ancestor information of the thread which is represented by their one-way regions, and also is an ascending order of their join counters. A multi-way loop can have a joined thread which happens before forked threads. The joined ancestors of a joined thread Ti are the joined threads at an earlier time than Ti. A one-way root of Ti is the most recent joined ancestor of Ti in each loop level. In figure 1, a joined thread T9 has one-way root T5, because the joined thread T5 is the most recent joined ancestor of T9 among its two joined ancestors T0 and T5.

e-NR labeling is designed using only one OH-list which maintains one-way roots for all thread labels. For each OH of a thread label called thread OH, this labeling uses a pointer which refers to a selective list from the OH-list. Figure 4 shows an example of e-NR labeling using OH-list and pointers for the program shown in figure 1. In this figure, the symbol "*" means an OH of joined thread to reference entries of OH-list, and the symbol "**" means that forked threads only refer to the thread OH of their one-way root.

For the OH of a thread label, e-NR labeling uses the one-way root counter ξ in the OR of the thread, denoted by [ξ, λ, <α, β>]. Thus, a joined thread Ti has a label that consists of OR(Ti) and OH(Ti), denoted by OR(Ti)*OH(Ti). Similarly, a forked thread Tj has a label denoted by OR(Tj)** OH(Tj).

```
0 e-NR Init( )
1  <α, β> := <1, maxint>;
2  λ := 1;
3  OH(1) := [λ, <α, β>];
4  ρ := Loc(OH(1));
5  ξ := 1;
6 End e-NR Init

0 e-NR Join( )
1  λ := λ'+1;
2  ξ := ξ +1;
3  OH_add(OH_p(a), [λ, <α, β>]);
4  ρ := Loc(OH(1));
5  λ'_p := max{ λ'_p , λ};
6 End e-NR Join

0 e-NR Fork( )
1  stride := (β_p − α_p+1) / U;
2  α := α_p + (I−1) × stride;
3  if (I<U) then
4    β := α + stride−1;
5  else β := β_p;
6  endif
7  λ := λ_p;
8  ρ := ρ_p;
9  ξ := ξ_p;
10 λ'':= λ;
11 End e-NR Fork
```

Figure 5. e-NR labeling algorithms

Labeling threads in a multi-way nested loop, called multi-way labeling or e-NR labeling, require thread fork and join operations. We use three algorithms for e-NR labeling shown in figure 5: *e-NR Init*(), *e-NR Fork*(), and *e-NR Join*(). In the algorithms, each data structure with a subscript p such as α_p represents the corresponding data structure of the ancestor thread which forked the current thread. The pointer variable ρ is for the thread *OH* which refers to the *OH*-list. The counter variable λ' is a mirror variable which is locally shared by the sibling of the current thread to help maintain λ in the thread. The function *OH_add*() in e-NR Join() manages the *OH*-list. If the $OR(T_i)$ of a joined thread T_i already exists in the *OH*-list, the λ of the one-way root is changed to λ_i. Otherwise, the $OR(T_i)$ is added as new one-way root in the *OH*-list.

Property 1 *Given two threads T_i and T_j ($i < j$), the one-way root T_x of T_j for comparing with T_i is an ancestor thread which has the smallest join counter in the $OH(T_j)$ such that the join counter λ_x is greater than join counter λ_i.*

For example, consider two threads T_3 and T_{10} in figure 4. Thread T_5 is the one-way root of T_{10} and its *OR* is the second entry of $OH(T_{10})$. Note that the λ_3 is smaller than λ_{10}. Now, we can efficiently compare logical concurrency information of any two threads in a nested parallel program using e-NR labeling.

Lemma 2 *Given three threads T_x, T_i, and T_j, if T_x is one-way root of T_j and $\lambda_i < \lambda_j$, $T_i \rightarrow T_j$ is equivalent to*

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

Theorem 2 *Given three threads T_x, T_i, and T_j, if T_x is one-way root of and $\lambda_i < \lambda_j$, $T_i \parallel T_j$ is equivalent to*

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

4. Evaluation

In this section, we evaluate the efficiency of e-NR labeling with others such as the original NR labeling and OS labeling using a set of synthetic program, and we analyze the logical complexity of three labeling schemes and the experimental results of the efficiency for labeling and on-the-fly race detection.

4.1. Experimentation

To evaluate the efficiency of e-NR labeling, our experiments are carried on a system with two Intel Quad-core CPUs and 8GB of memory under the Ubuntu Linux operating system. We installed gcc 4.4.3 for OpenMP 3.0 which supports nested parallelism on the system. e-NR labeling is implemented as run-time libraries written in C language to be inserted into the target program. Then, we compiled and executed the instrumented program for the labeling and race detection.

We performed the following two steps for comparing the efficiency: labeling overhead and race detection overhead. We use the original NR labeling and OS labeling [4] for the comparison with the e-NR labeling, because it is well known that both two labeling schemes are effective for fork-join parallel programs with nested parallelism. Thus, we also implemented two labeling schemes with run-time libraries which are used to instrument OpenMP programs. For on-the-fly race detection with the three labeling schemes, we employ Mellor-Crummey's protocol because the protocol exploits a simple technique to compare any two threads, called left-of relation, and requires small overhead for detecting races during an execution of a parallel program. To evaluate the efficiency of the three labeling schemes, we developed a set of synthetic programs considering nesting depth, maximum parallelisms, and multi-way loops.

4.2. OpenMP Synthetic Programs

OpenMP programs with directives based on C/C++ language have a parallel computing program model and an activity management program model based on parallel threads. A parallel computing program divides single computation job into several jobs, and these jobs have the same kind of data structures and variables. An activity management program creates parent and child threads that have the allocated nestable thread teams in a program, and these thread teams have different kinds of data structures which may be shared. We evaluate the efficiency of three labeling schemes using a set of synthetic programs which based on the activity management program model.

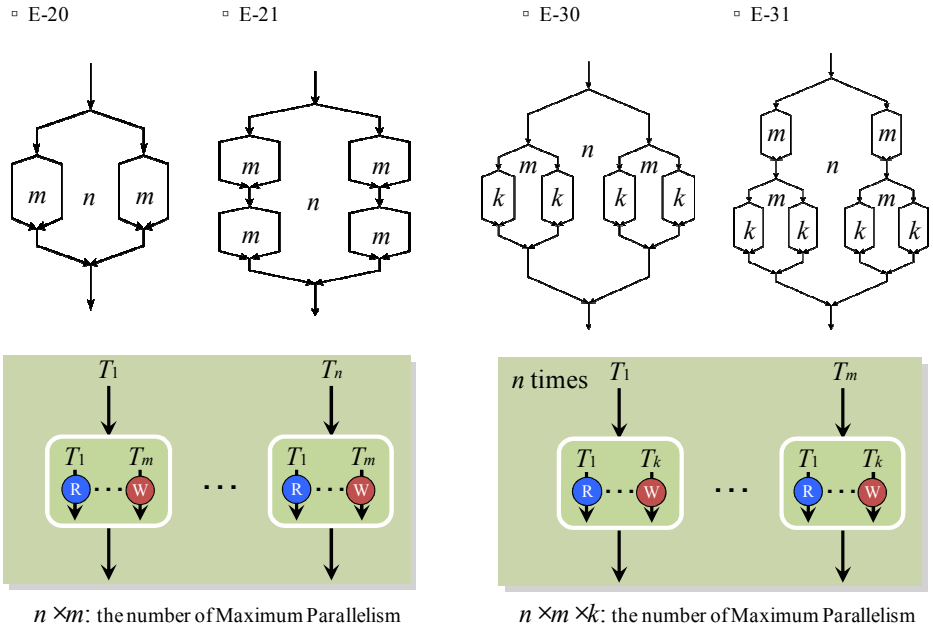


Figure 6. Design of synthetic programs to experiment three labeling scheme's efficiency

The synthesizes for the accuracy designed to consider general cases of parallel loop programs which include nested parallelism, multi-way loop, and maximum parallelism. We developed the synthetic programs that the nesting depths of the programs are maximally four for the efficiency, and we increase the maximum parallelism of the programs varying from 10,000 to 1,000,000. Figure 6 shows the models of designed synthetic programs which use to experiment the efficiency of three labeling schemes. In the figure, "E" represents the program for testing the efficiency of three labeling schemes. The first number means the nesting depth of the program, and the second number for multi-way loop program. The maximum parallelisms of each synthetic program are divided the total number of threads among each loop level. For example, "E-20" model allocates 100 threads to outer loop (n) and 100 threads to inner-most loop (m) for 10,000 threads which is the maximum parallelism of the program. Through the synthetic programs, we can experiment on most case of nested parallel programs written by programmer.

4.3. Results and Analysis

The efficiency of the three labeling schemes depends on three parameters: V for the number of monitored shared variables, T for the maximum parallelism, and N for the nesting depth of the monitored program. When a fork or join operation occurred, the time complexity to generate and maintain a thread label is $O(1)$ for e-NR labeling since it assigns a label that consists of a one-way region of size $O(1)$ and a pointer to one-way history of size $O(1)$. However, the original NR labeling consumes $O(N)$ time in the worst case, because it generates a label that consists of a one-way region of size $O(1)$

and a one-way history of size $O(N)$. OS labeling also consumes $O(N)$ time for maintaining thread labels in the worst case.

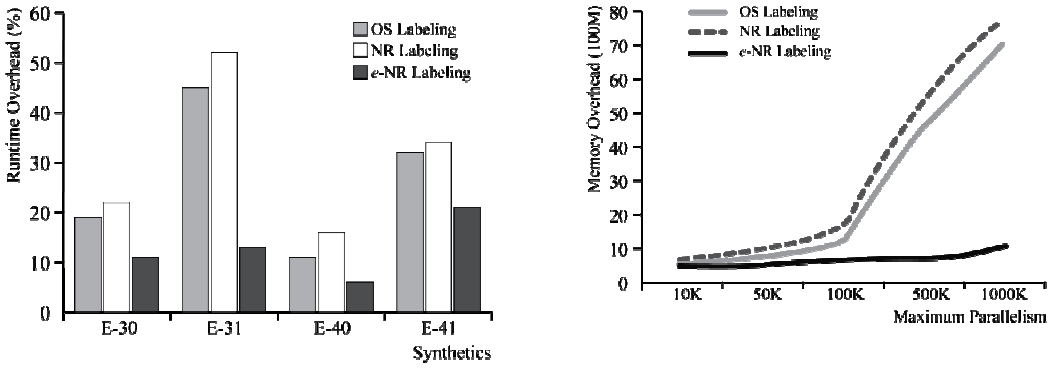
Table 1. Results for the efficiency of three labeling schemes

Programs	Max. parallelism	10,000		100,000		1,000,000	
		User	Sys	User	Sys	User	Sys
E-30	OS labeling	0.65	0.01	1.22	0.08	6.57	13.39
	NR labeling	0.76	0.08	1.67	5.51	4.57	36.66
	e-NR labeling	0.44	0.01	0.56	0.03	1.08	0.44
E-31	OS labeling	0.83	0.16	2.35	10.32	17.33	96.37
	NR labeling	0.93	0.18	2.65	11.28	18.27	98.49
	e-NR labeling	0.71	0.01	1.62	2.68	7.96	15.06
E-40	OS labeling	0.87	0.04	1.03	1.04	6.39	18.81
	NR labeling	0.76	0.04	0.93	0.95	7.27	19.69
	e-NR labeling	0.48	0.01	0.63	0.06	1.48	1.57
E-41	OS labeling	0.63	0.06	1.07	1.06	13.81	58.33
	NR labeling	0.72	0.50	1.06	1.01	14.49	62.51
	e-NR labeling	0.46	0.01	0.77	0.05	9.67	12.51

These labeling are effective for maintaining the concurrency information during race detection, and then make the efficiencies dependent on the size of thread labels. The space complexity for the original NR labeling is O(V+NT) in worst case. To maintain the concurrency information for all shared variables, the required space is O(NT). To store constant-sized labels of simultaneously active threads, the space depends on the nesting depth and the maximum parallelism. By OS labeling, the space complexity is O(VN) in the worst case, since concurrent information for logical threads requires the space of O(N). The space complexity by e-NR labeling is O(V+T), because it does not depend on nesting depth N.

We evaluate the efficiency of e-NR labeling by analyzing the time and space overhead. Table 1 shows the result of measurement of the required time for labeling with the synthesized programs.

The results show that e-NR labeling is 5 times faster than original NR labeling and 4.3 times faster than OS labeling in the average time for creating and maintaining thread labels.



(a) Time overhead (b) Space overhead

Figure 7. Result for the efficiency of race detection using three labeling schemes

Figure 7 shows that the results of measurement of the time and space overhead for on-the-fly race detection using the three labeling schemes. Figure 7(a) compares the time overhead of the three labeling Schemes, the slowdown of e-NR labeling is about only 13% in average case, but original NR labeling and OS labeling slowdown about 32% and 27% respectively. In figure 7(b), the space overhead of the original NR labeling and OS labeling increase dramatically, because the increasing maximum parallelism affects to execution overhead with the nesting depth. However, e-NR labeling shows small space overhead for on-the-fly race detection. In average, the space required for labeling is 3.5 times smaller than original NR labeling and 3 times smaller than OS labeling. Thus, the efficiency of e-NR labeling makes on-the-fly race detection more practical.

5. Conclusion

Data races in parallel programs can be occurred when two parallel threads access a shared memory location without proper inter-thread coordination, and at least one of these accesses is a write. Race detection is important for the debugging of parallel programs, but it is difficult and cumbersome. Any on-the-fly race detection techniques using Lamport's happened-before relation needs a thread labeling scheme for generating unique identifiers which maintain logical concurrency information on the parallel threads. In general, it is difficult to implement efficiently these methods, because they require expensive costs for time and storage space for on-the-fly race detection.

This paper presents e-NR labeling which has a time and space complexity of O(1) for creating and updating parallel thread labels. This improves original NR labeling which

is an efficient labeling scheme for the fork-join program model with nested parallelism. Because e-NR labeling removes the dependency on the nesting depth N of the previous labeling on every thread operation, the labeling requires a constant amount of time and space complexity. Empirical comparison were performed on OpenMP synthetic programs which have nesting depths of three or four and maximum parallelisms between 10,000 and 1,000,000. The results show that e-NR is 5 times faster than original NR labeling and 4.3 times faster than OS labeling for creating and maintaining in thread labels. In average, the space required for labeling is 3.5 times smaller than original NR labeling and 3 times smaller than OS labeling.

This improvement makes on-the-fly race detection technique more practical. It guarantees moderate enough time and space overhead for large parallel programs. Future work includes additional improvement of e-NR to compare the concurrency of parallel threads in a constant amount of time and extending it to handle more general execution models with inter-thread coordination.

References

[1] Netzer, R. H. B., and B. P. Miller, "What Are Race Conditions? Some Issues and Formalizations", *ACM Lett. Program. Lang. Syst.*, ACM, Vol. 1, No. 1, pp. 74-88, 1992.

[2] Banerjee, U., B. Bliss, Z. Ma, and P. Petersen, "A Theory of Data Race Detection", *In Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (*PADTAD*), ACM, pp. 69-78, 2006.

[3] Rinard, M., "Analysis of Multithreaded Programs", Edited by Cousot, P., *SAS 2001*. LNCS, Vol. 2126, pp. 1-19, 2001.

[4] J.M. Mellor-Crummey, "On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism", *In Proceedings of the ACM/IEEE conference on Supercomputing*, ACM/IEEE, pp. 24-33, 1991.

[5] Jannesari, A., and W. F. Tichy, "On-the-fly Race Detection in Multi-threaded Programs", *In Proceedings of the 2008 workshop on Parallel and distributed systems: Testing, Analysis, and Debugging* (*PADTAD*), ACM, pp. 1-10, 2008.

[6] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, ACM, Vol. 21, No. 7, pp. 558-565, 1978.

[7] Jun, Y., and K. Koh, "On-the-fly Detection of Access Anomalies in Nested Parallel Loops", *In Proceedings of 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, ACM, pp. 107-117, 1993.

[8] Park, S., M. Park, and Y. Jun, "A Comparison of Scalable Labeling Schemes for Detecting Races in OpneMP Programs", Edited by Eigenmann, R. and Voss, M.J., *WOMPAT 2001*. LNCS, Vol.2104, pp. 68-80, 2001.

[9] Petersen, P., Shah, S, "OpenMP Support in the Intel Thread Checker", Edited by Voss, M.J., *WOMPAT 2003*. LNCS, Vol.2716, pp. 1-12, 2003.

[10] *The OpenMP API specification for parallel programming*, http://www.openmp.org

[11] C. J. Fidge, "Logical Time in distributed Computing Systems", *Computer*, ACM, pp. 28-33, 1991.

[12] Baldoni, R. and M. Raynal, "Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems", *IEEE Distributed Systems Online* 3(2), IEEE Computer Society, 2002.

[13] Audenaert, K, "Clock Tree: Logical Clocks for programs with Nested Parallelism", *IEEE Transactions on Software Engineering*, IEEE Computer Society, Vol. 23, No. 10, pp. 646-658, 1997.

[14] Dinning, A. and E. Schonberg, "Detecting Access Anomalies in Programs with Critical Sections", *In Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, ACM, pp. 85-96, 1991.

Authors

Ok-Kyoon Ha received the BS degree in Computer Science under the Bachelor's Degree Examination Law for Self-Education from National Institute for Lifelong Education and the Master of Science from Gyeongsang National University (GNU), South Korea. He is currently enrolled as a PhD candidate with the Department of Informatics in GNU. He worked as the manager of IT department in Korea industry for several years. His research interests include parallel/distributed programming and its debugging, embedded system programs, and operating system. Mr. Ha is a member of Korean Institute of Information Technology (KIIT) and Korea Institute of Information Scientist and Engineers (KIISE).

Dr. Yong-Kee Jun received the BS degree in Computer Engineering from Kyungpook National University, and the MS and PhD degree in Computer Science from Seoul National University. He is now a full professor in the Department of Informatics, Gyeongsang National University, where he had served as the first director of GNU Research Institute of Computer and Information Communication (RICIC), and as the first operating director of GNU Virtual College. He is now the head of GNU Computer Science Division and the director of the GNU Embedded Software Center for Avionics (GESCA), a national IT Research Center (ITRC) in South Korea. As a scholar, he has produced both domestic and international publications developed by some professional interests including parallel/distributed computing, embedded systems, and systems software. Prof. Jun is a member of Association for Computing Machinery (ACM) and IEEE Computer Society.