6.895 Theory of Parallel Systems October 9, 2003

Project Proposal: Parallel Nondeterminator

He Yuxiong

1 Background

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 to write the location. Data race detection mechanisms are grouped by static and dynamic race detection. Dynamic race detection also has on-the-fly and post-mortem approach. The parallel nondeterminator I propose is a dynamic race detection algorithm with on-the-fly approach.

2 Project Objective and Primitive Ideas

I propose the parallel nondeterminator to check the determinacy race in the computation in parallel execution of the program written in the language like Cilk on a given input. The nondeterminator are supposed to guarantee the race is detected in location L if and only if a determinacy race exists in location L. The optimization of parallel nondeterminator mainly depends on two aspects - how quickly the test of concurrency can be made and how many entries are in the access history.

The parallel nondeterminator has increased complexity than serial one in the two aspects. Serial nondeterminator [1] only needs to identify the relationship between the running thread with other functions by taking the advantage of depth first serial execution. But parallel nondeterminator needs to identify the relationship between the running thread and other threads in the functions since the running thread may be parallel with some threads in function F and be serial to other threads in the same function. Serial nondeterminator only need to keep one access record for each shared location. Parallel nondeterminator may need the access history of each location as large as the maximum level of parallelism. The goal of the project is to make fast test of concurrency and reduce the number of entries in the access history so that the parallel nondeterminator runs efficiently regarding to time and space.

2.1 Primitive Idea of Concurrency Test

The related work [3] uses thread labeling - English-Hebrew labeling and task recycling algorithm to test the currency. English-Hebrew labeling is only applicable when the parent function knows the total number of children it is going to spawn. Task recycling algorithm is only applicable when the maximum number of parallel functions is known in the program. Both of them can not be used in cilk-like language. It is an alternative to design an efficient thread labeling scheme applicable to cilk-like language.

Another alternative is to use set operations for concurrency test. Each thread is represented by (fid, tid). Fid is a unique identifier for each function. Tid of each function starts from 1, and increases by 1 after each of the spawn or sync statement.

Each function f keeps two sets. The parallel set PS(f) stores the pairs {(fid, tid) | function fid is parallel with the running thread when thread number $\geq$ tid}. The child set CS(f) stores {fid | fid is the descendant of current function}. The root function is initialized with empty parallel set and child set. Following operations are performed when the program is executed:

Spawn: Thread T_x of function F_i spawn function F_j

Operations on child F_j

1. $PS(F_j) \leftarrow PS(F_i) \cup \{(F_i, T_x + 1)\}$
2. $CS(F_j) \leftarrow \{\}$

Operations on parent Fi

1. $\forall F_p, \exists tid, ((F_p, tid) \in PS(F_i)) \rightarrow ((PS(F_p) = PS(F_p) \cup \{(F_j, 1)\}$
2. $PS(F_i) = PS(F_i) \cup \{(F_j, 1)\}$
3. $CS(F_i) = CS(F_i) \cup \{F_j\}$

Sync: Function F_i executes sync

$PS(F_i) = PS(F_i) - CS(F_i)$

Return: Function F_j returns to Function F_i

$CS(F_i) = CS(F_i) \cup CS(F_j)$

Release PS(F_j) and CS(F_j)

Concurrency Test:

Check if (fx, tx) is parallel with the current running thread (fc, tc):

$\exists tid, ((fid, tid) \in PS(fc)) \wedge (tid \leq tx) \rightarrow (fx, tx)$ is parallel with (fc, tc)

I haven't proved out that the algorithm can decide the concurrency relation correctly. And I haven't found efficient data structure to perform the operations. The main overhead is on spawn operation. It needs to propagate the new child information to all the functions which are parallel with the current thread of the parent. Other set operations are also required to be revised in order to find an efficient algorithm.

2.2 Primitive Idea of Access History

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 cilk-like language structure.

In simple parallel program without nested parallel loop, three entries (two for read, one for write) should be enough. The rule for recording read access history is shown as below:

T read location l:

1. Tw=write(l)

   If Tw is parallel with T, report data race in location l

2. Tr1=read1(l) and Tr2=read2(l)

   If T is parallel with Tr1, set read2(l)=T ;

   Else if T is parallel with Tr2, set read1(l)=T ;

   Else set read1(l)=T, read2(l)=null

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 highest level of LCA (least common ancestor) in parent child spawn tree, we may be able to detect the data race with only two read access records. But checking LCA in parent child spawn tree is also costly. Should we just record all the parallel read accesses or record only two read accesses through more complex preprocessing? It requires further analysis and comparison.

3 Backup plan

I think I may have some difficulties to think out very efficient algorithms for parallel nondeterminator. If I still can't find efficient algorithm with my best efforts, I will have to look for sub-optimal solutions. Even when I think out some algorithms, I may meet difficulties in their theoretical analysis on time and space complexity. In such a situation, I may choose to give a description on the algorithm and show its performance in more generall way.

4 Related Works

I list some of the related works and show the expected new features of the proposed Parallel Nondeterminator.

- The Nondeterminator-1 of Cilk [1] checks determinacy race in the computation generated by a serial execution of the program on a given input with bounded execution time in O(Ta(v,v)). The Nondeterminator-2 [2] checks the data race in the locked critical section. Both of them are running the instrumented parallel program in serial execution. The proposed nondeterminator will detect the race in the parallel execution of the program.

- The paper [3] provides two labeling algorithms for access anomaly detection. As mentioned in the previous part, the two algorithms are not applicable to the cilk-like language, in which the parent function doesn't know how many children it is going to spawn.

- The paper [4, 6] detects data race in the parallel execution of the program with the support of modified version of the Coherent Virtual Memory. The nondeterminator will detect not only the data race encountered but also determinacy races.

- Paper [5] works on the first race detection in parallel programs. It detects all the first races in the simple programs without nested parallel loop.

- There are some other related works that I am going to read.

5 References

1. Mingdong Feng, and Charles E. Leiserson. "Efficient Detection of Determinacy Races in Cilk Programs" ACM Symposium on Parallel Algorithms and Architectures 1997.

2. Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall,and Andrew F. Stark. "Detecting Data Races in Cilk Programs that Use Locks" Annual ACM Symposium on Parallel Algorithms and Architectures 1998.

3. Anne Dinning, and Edith Schonberg. "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection" 2nd Symp. On Principles adn Practices of Parallel Programming, pp. 1-10, ACM March 1990.

4. Dejan Perkovic, and Perter J Keleher. "Online Data-Race Detection via Coherency Guarantees" Proc. of the 2nd Symp. on Operating Systems Design and Implementation.

5. Jeong-Si Kim, and Yong-Kee Jun. "Scalable On-the-fly Detection of the First Races in Parallel Programs" International Conference on Supercomputing 1998.

6. Dejan Perkovic, and Perter J. Keleher. "A Protocol-Centric Approach to on-the-fly Race Detection" IEEE Transactions on Parallel and Distributed Systems.

7. J. Mellor-Crummey, "Compile-Time Support for Efficient Data Race Detection in Shared-Memory Parallel Program" Technical Report CRPC-TR92232 Rice Univ., Sept 1992.

8. R.H.B. Netzer and B.P. Miller, "Improving the Accuracy of Data Race Detection", Proc. 1991 Conf. Principles and Practice of Parallel Programming. Apr, 1991.

9. J. Choi adn S.L. Min, "Race Frontier: Reproducing Data Races in Parallel Programming", Proc. 1991 Conf. Principles and Practice of Parallel Programming, Apr. 1991.