Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 274 (2011) 83–96

www.elsevier.com/locate/entcs

A Parallel Approach to Concolic Testing with Low-cost Synchronization

Xiao Yu[1], Shuai Sun[2], Geguang Pu [1,4], Siyuan Jiang[3] and Zheng Wang[1]

Shanghai Key Laboratory of Trustworthy Computing
East China Normal University
Shanghai, China

Abstract

This paper presents a practical approach to parallelize the test data generation algorithm by which computing resources can be fully used. The test data generation approach that we are using is based on the dynamic symbolic execution (*concolic testing*). The basic idea of parallelizing the algorithm is to distribute analysis processes of different paths to different computing units. Although a centralized scheduler with several sub processes can directly achieve the goal of parallelism, it may cause global idle time when parallel processes frequently end at same time. In our approach, a runtime deterministic scheduler is introduced to reduce the potential global idle time. Our experiments show some notable results when using a proper scheduling function. Compared with the sequential concolic testing, our approach can save nearly 70% computing time in some cases on a system with eight CPU cores from our experiments.

Keywords: Parallel Algorithm, Automatic Test Generation, Symbolic Execution

1 Introduction and Motivation

Software testing is a popular methodology to find bugs. The most crucial step of software testing is designating proper test inputs. Some tools pay attention to automate this step. Pex [16] is an automated unit testing tool that can automatically generate test inputs for .NET applications. CUTE [15] is another tool that can analyze and generate test data for C programs. The underlying techniques for automated test data generation have experienced a long time of development. Symbolic execution [11], initially introduced in 1970s, uses symbols to represent concrete input values of programs and provides alternative execution semantics over

[1] {lesteryu, ggpu, wangzheng}@sei.ecnu.edu.cn
[2] marksun1988@gmail.com
[3] qin1537@hotmail.com
[4] Correspondence Author

doi:10.1016/j.entcs.2011.07.008

these symbols. By relying on the advance of constraint solvers, it is possible to obtain precise concrete inputs to guide program executions from program states which are represented over symbols. This idea has inspired many works on automated testing [8,13,18,17,20,21]. Recently, the concept of *concolic testing* [10,15,7] has been proposed. Concolic testing is a variant of symbolic execution and aiming at generating test data automatically, with the advantage that concrete program states are also adopted to guide the process of symbolic execution. Constraints of program paths are incrementally collected while some of those are replaced by concrete states. It is an enhanced dynamic symbolic execution [12] technique. The constraints collected along one path can be simplified with this technique. It is a novel way to improve the usability and the performance of pure symbolic execution. Based on this improvement, many other techniques [14,6,5,9,4] have been proposed to further improve the usability of concolic testing.

Test data generation techniques, like concolic testing, face with the path-state explosion problem since program paths increase radically with the growth of program scale. For example, the following code fragment has only 35 lines, it tries to find the word 'web' and the word 'ebay' in a given string with 5 characters. It uses a typical method of state machine. Every loop iteration contains 6 first-level branches, each of which contains 3 or 4 second-level branches. Because the first-level branches are not overlapped with each other, in all, there are 23 branch conditions in one single loop iteration. Considering the loop condition, the feasible paths can be deep and the number of feasible paths can be enormous. We tested this function on an *Intel Core i7* platform. The result showed that the sequential concolic test cost about 84.6 seconds to complete the searching of about 3400 feasible paths.

```
void foo(char c[]) {
    int state = 0, idx = 0;
    while (c[idx] > 0) {
        if (state == 0) {
            if (c[idx] == 'w')    state = 1;
            else if (c[idx] == 'e')     state = 2;
            else    state = 0;
        }
        else if (state == 1) {
            if (c[idx] == 'w')    state = 1;
            else if (c[idx] == 'e')     state = 3;
            else    state = 0;
        }
        else if (state == 2) {
            if (c[idx] == 'w')    state = 1;
            else if (c[idx] == 'e')     state = 2;
            else if (c[idx] == 'b')     state = 4;
            else    state = 0;
        }
        else if (state == 3) {
            if (c[idx] == 'w')    state = 1;
            else if (c[idx] == 'e')     state = 2;
            else if (c[idx] == 'b')     state = 6;
            else    state = 0;
        }
        else if (state == 4) {
            if (c[idx] == 'w')    state = 1;
            else if (c[idx] == 'e')     state = 2;
            else if (c[idx] == 'a')     state = 5;
            else    state = 0;
        }
        else if (state == 5) {
            if (c[idx] == 'w')    state = 1;
            else if (c[idx] == 'e')     state = 2;
            else if (c[idx] == 'y')     state = 7;
            else    state = 0;
        }
        else if (state > 5)  {
            printf ("\nHit\n");
            break;
        }
        idx++;
    }
}
```

To overcome this problem, some researchers proposed their approaches. For instance, Boonstoppel *et al.* proposed a simple algorithm called *RWset* [5] to reduce the number of traversed code paths by means of side-effects analysis among variables appeared in paths. Godefroid proposed the *SMART* [9,4] algorithm to reduce the cost of compositional state explosion problem brought by compositional units in programs. In this paper, we propose a parallel approach to concolic testing. It is a different view of improving usability from other previous work [5,9,4]. Our parallel approach contributes a way to reduce the time cost of test data generation. The general idea of a parallel approach is to schedule tasks to different computation units, and to maintain a global task queue. In our approach, the executions of program under test along with the computation of conditional choices are treated as the tasks that are scheduled to each computation unit on the parallel system by certain scheduling policy. Every iteration of execution generates a certain number of alternative path prefix candidates, which are subsequent tasks. The path selecting and distributing method makes sure that all tasks of path computation are processed and no task is repeated. It also tries to equalize the computation time over different cores, in order to make the best use of the computational units. Although in the scenario of parallel concolic testing the pattern that consists of a centralized scheduler and several sub processes can directly achieve the goal of parallelism, it may cause global idle time when parallel processes frequently end simultaneously.

Rather than building the centralized scheduler, we introduce a deterministic scheduler on each working unit, which considerably reduces the synchronization time cost. The details of our approach will be expanded in the following sections. The main contributions of our work are:

- Enhancing parallel capability to traditional concolic testing
- Introducing the concept of runtime deterministic scheduler in order to reduce synchronization time
- Implementing a parallel concolic testing framework with positive experimental results.

In this paper, Section 2 describes the basis of concolic testing and details about how to achieve parallelism. Section 3 shows the results of experiments and the discussion. The last section gives the conclusion.

2 Parallel Approach

The parallel approach for test data generation is introduced in this section.

2.1 Background: Concolic Testing

Concolic testing [15], which derives from dynamic test data generation [12], is a variant of symbolic execution. It combines symbolic execution and executing program under test concretely to dynamically generate test inputs and cover all feasible paths. When executing the program under test, it monitors the choices of branches along execution paths, then uses backtracking to collect path constraints and represents path constraints by a set of formula with constants and symbols which are related to input variables. In order to explore new paths and generate test data, the concolic testing algorithm modifies the collected path constraints slightly to satisfy some other expected paths, and uses a constraint solver to solve modified path constraints. The constraint solver returns a solution, which forms the inputs of a new expected path and for the execution of next iteration. This process is iterated until no new path can be generated, which indicates that all feasible paths of the program under test are fully explored.

During the concolic testing process, two main structures are maintained through iterations. One structure is the global path decision tree T, which contains the path (consists of a series of branch choices) recorded from every iteration and shares path information between each iteration. The other is a sequence of values M which provides concrete values for the sequence of input variables I. The whole process of sequential concolic testing is within a main iteration. In each iteration, it substitutes the input I with the value M into the target program P, and starts to execute P concretely and symbolically. The result from the execution of target program can be treated as a triple which consists of the execution trace t, the decision path p and the path feasibility, which is either *feasible* or *infeasible* indicating whether the execution of the target program goes through the expected path or the execution is aborted abnormally. The algorithm terminates when no more expected path prefix can be explored. Some technical details can be found in [19].

2.2 Parallel Model

This subsection presents the parallel algorithm of the concolic test data generation. We design a parallel model that makes the process of test data generation run concurrently. An interesting point in this parallel algorithm is that we divide the whole path space into different disjoint areas dynamically that can be managed and updated by different computing units. Thus, each computing unit can freely access and analyze the paths belonging to its own allocated area. This means the global synchronization can be fundamentally removed among parallel computing units. This technique can further improve our performance of parallelized concolic testing. The removal of global synchronization is implemented by a runtime task scheduler which allows each computing unit safely updates its own data on a shared global decision tree.

2.2.1 Architecture

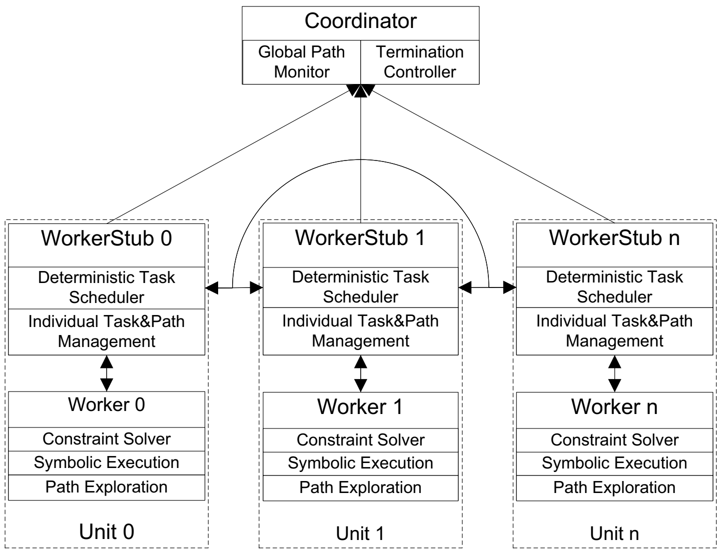

Fig. 1. Architecture

Figure 1 shows the basic architecture of our parallel model. There are three roles in the parallel model. There are *Worker*, *WorkerStub* and *Coordinator*, respectively. Instances of *Worker* perform actual concolic testing simultaneously on different computing units. Each of instances of *Worker* is managed by a corresponding instance of *WorkerStub*. The *WorkerStub* is the essential part of our parallel algorithm for the reduction of global synchronization. Instead of a centralized task scheduler, *WorkerStub* holds a runtime deterministic task scheduler, which will be explained later. Specifically, the *WorkerStub* takes charge of (1) starting one instance of *Worker* each iteration, (2) assigning path computing tasks to the *Worker*, (3) collecting feedback from the *Worker*, (4) pairwise exchanging computing tasks with other instances of *WorkerStub*, (5) managing a partial path decision tree and reporting the path decision tree it manages to the *Coordinator*.

The algorithm of *Worker* is shown in Algorithm 1. It is extended from the sequential process of concolic testing in order to support parallel exploration of program paths. A *Worker* takes (P, I,*choice*, *trace*) assigned by *WorkerStub* as inputs where P is the target program to be tested, I is the sequence of input variables, *choice* is the truth-value assignments for the expected path prefix and *trace* is the program trace which relates to the expected path prefix. After solving constraints of *choice* and executing the target program with the solving result (Line 14), the *Worker* returns (*feasible*, p, t, S_p) for a feasible path or *infeasible* for an infeasible path according to the current performance. If it is a feasible path, the complete executed path p (note that the prefix of p is the input *choice*) and the corresponding program trace t will be sent back to *WorkerStub* in order to build a partial path decision tree. If it is an infeasible path, the original expected path prefix will be marked as *infeasible*. Besides, the p-related path prefix set S_p is also returned to *WorkerStub* to create new tasks. The set S_p is computed by negating every constraint assignment by **FlipLastChoice** (Line 19-25) on the path p.

When one *WorkerStub* has been started up, it begins to maintain an individual task list and a partial decision tree which will be built from the complete tasks in the individual task list. If the started *WorkerStub* is the first instance and the task list is empty, it will start an instance of *Worker* with empty inputs in order to get the first path data from the target program. Otherwise, the *WorkerStub* will wait for some tasks sent from other *WorkerStub*s to the individual task list and then run a series of *Worker* iteratively to compute tasks. When an instance of *WorkerStub* has received a set S_p from its worker, it firstly uses the local deterministic task scheduler to decide for a specific path prefix in S_p which *WorkerStub* should receive it as an individual task. After the sorting, every item in S_p with related program trace will be sent as individual task to corresponding *WorkerStub* told by the scheduler.

The *Coordinator* maintains a global view of the path decision tree by periodically collecting and merging partial trees from all instances of *WorkerStub*. It initially starts several instances of *WorkerStub* (the exact number of instances is decided by the number of processors installed on the target computer). It terminates the whole testing process when the global path decision tree is full.

To the generalized view of the parallel model (see Figure 2), the *Worker*s with *WorkerStub*s are the parallelized units. The *Worker* only communicates with its own *WorkerStub* by which it receives computing tasks and sends results. The *Coordinator* mainly controls the termination. The key of low-cost synchronization is the *Deterministic Task Scheduler* that makes the task scheduling free from a global serialized task list. The shared tasks and tree are divided into a set of disjoint areas by the deterministic task scheduler, which will be explained in the following section.

Algorithm 1 The algorithm on the worker

```
Worker ($P$, $I$, $choice$, $path$, $trace$)
Inputs:
  $P$ is the target program to be tested.
  $I$ is the sequence of input variables.
  $choice$ is the truth-value assignments for the expected path prefix.
  $path$ is one path which relates to the prefix $c$ from the entire decision tree.
  $trace$ is the program trace which relates to the $path$.
Returns:
  ($feasible$, $p$, $t$, $S_p$) where $p$ is the result path of $P$,
  $t$ is the result trace and $S_t$ is the set of all prefix paths related to $p$.
  $infeasible$ when no solution satisfies the $choice$ on $path$.

1: $M := \langle\rangle$ {Let $M$ be the constraint solution values corresponding to the input $I$}
2: if $path \neq nil$ then
3:   let $c_1, c_2, ..., c_n$ be all non-leaf nodes of $path$ and $C$ be the final whole sequence of constraints
4:   for $i = 1$ to $n$ do
5:     $C := C^\frown$ CollectPathConstraints($c_i$, $trace$, $I$)
6:   end for
7:   $M$ :=SolveConstraints($C$, $choice$)
8: else
9:   $M$ :=GenerateRandomInput($I$)
10: end if
11: if $M = \langle\rangle$ then
12:   return $infeasible$
13: end if
14: $(t, p, s)$ :=ConcreteAndSymbolicExecution($P, I, M$)
15: if $s$ =$infeasible$ then
16:   return $infeasible$
17: end if
18: $S_p := \phi$
19: $c$ :=GetBranchChoice($p$)
20: while $c \neq 0$ do
21:   let $i_1, i_2, ...i_n$ be the branch choices in $c$
22:   $expected$:= FlipLastChoice($c$)
23:   $S_p := S_p \cup \{expected\}$
24:   $c := c - i_n$
25: end while
26: return ($feasible$, $p$, $t$, $S_p$)
```

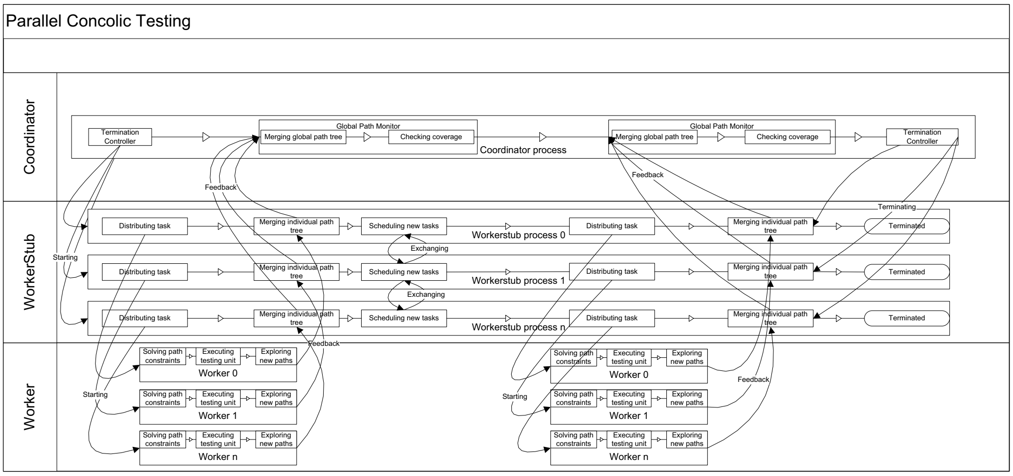

Fig. 2. Processes

2.2.2 Deterministic Task Scheduler

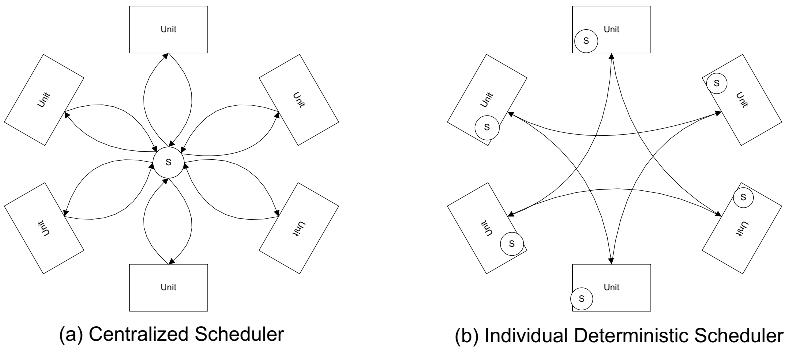
Unit
S
(a) Centralized Scheduler
(b) Individual Deterministic Scheduler

Fig. 3. Centralized and Deterministic Schedulers

A naive task scheduler (Figure 3a) usually maintains a centralized task list to schedule tasks. The centralized scheduler takes charge of assigning tasks to workers, which is shown as lines in Figure 3a. When there are more than one free workers waiting tasks, the scheduler has to serialize the assignment of tasks to avoid data races among workers (e.g avoiding two workers get the same task). The serialization means the workers have to wait in a queue, which leads to a waste of computing time. The underlying reason of the imperative serialization is the nondeterminism in the scheduling plan. It simply distributes tasks to any free worker. A task in the list can be accomplished by any workers.

In our parallel model introduced previously, dynamically generated paths can be scheduled among different computing units by a uniform deterministic task scheduler. The deterministic task scheduler (Figure 3b) is designed to overcome the disadvantages of the nondeterministic scheduler. As we can see in Figure 3b, the centralized structure along with its connections with the workers is eliminated. The workers connect directly with each other to exchange path records only when necessary(determined by the deterministic scheduler on each worker). The effect of the deterministic scheduler is that for each task generated by the testing process, the scheduler tells which worker should compute the task by a universal independent algorithm instead of randomly allowing some free worker to compute the task. Thus, the deterministic scheduler can be placed in each computing unit instead of a global one, which makes the serialized task list be separated to each computing unit. Finally, the global synchronization is eliminated.

Several observations should be kept to implement the deterministic task scheduler. The global path decision tree of a program is dynamically combined through concolic testing iterations. Each *Worker* iteration consumes only one task that consists of a path prefix which is corresponded to program history traces, and generates only one path from the prefix despite the path is feasible or infeasible. Each path along with its prefix on the tree is computed independently from others. By dividing and projecting paths and prefixes on the tree into the discrete space, the whole computation of the target program is naturally classified to several disjoint regions on the space. Every working unit takes charge of one region so that all working units have the knowledge of a specific task which working unit should take charge of.

Formally, the deterministic task scheduler is implemented by a function $(\mathcal{F} \circ \mathcal{H})(p)$ where the sub function $\mathcal{H}$ takes its domain as all possible paths on the binary decision tree, and its image is a bounded range of positive integer to present the abstract path space. Function $\mathcal{F}$ maps the image of $\mathcal{H}$ to integers in the range $[0, MAX\_UNIT\_CNT)$ to present regions on the space. The function must ensure that on different computing units it should get the same result for the same path p (namely deterministic). The sub function $\mathcal{H}$ can have many types of implementations. Different implementations have different effects to the parallel concolic testing.

A good design of the schedule function is a hard problem. One reason for its difficulty is that the condition of the path tree can be varied. Different programs, even different units in the same program, have different kinds of path distribution. This means there can hardly be a universal scheduler which performs the same well on every testing unit. Another reason for the difficulty is the inability in the prediction of the running time of every worker iteration. Even if we find a scheduler which can balance the number of paths on workers, the total time cost on different workers may be still unbalanced.

Although we have not got an excellent scheduler function, we come up with some standards which a good scheduler function is supposed to stick to. A good scheduler function should divide the space as uniformly as possible, each computing unit having almost the same number of tasks. Moreover, it is even better if the scheduler could balance the overall computing time on each computing unit. On the other hand, a poor scheduler function fails to equitably distribute the number of tasks and computing time on each computing unit. This could lead to that a certain number of computing units are extremely busy, while the others are just on waiting state

For instance, let p be the length of a path to be scheduled, and $\mathcal{H}$ be the hash policy used in the scheduler. Then, we have the following definitions for the scheduler function $(\mathcal{F} \circ \mathcal{H})(p)$:

$$\mathcal{H}(p) = p \ mod \ MAX\_UNIT\_CNT, \ \mathcal{F}(p) = \mathcal{H}(p).$$

This scheduler assigns a task to the working unit identified by the length of path prefix modulo the number of computing units. If the number of computing units is larger than the length of the longest path in the target program, some computing units will stay in starving state for a long time. Thus, the design of a fair function $\mathcal{H}$ is important to improve the whole testing performance.

2.2.3 Motivating Example Revisited

The motivating example in Section 1 can be efficiently processed. We tested the example on an *Intel Core i7* platform which is equipped with eight logical processors. Compared to the result of the sequential testing (84.6 seconds), the parallel testing cost only 21.4 seconds to complete the searching of about 3400 feasible paths. During the parallel testing in this case, eight processes are started at the same time to explore the path space. The CPU workload is fully utilized for solving constraints of long paths and exploring more paths from existing paths. The percentage of the performance improved is determined not only by the increasing of computing units but also by the complexity of the target program under test. In the extreme case by the example we present here, the performance improvement is huge. It shows that the parallel approach gains better advantage for large program involving longer paths and more complex path conditions.

3 Evaluation

We have implemented the parallel algorithm and integrated it into the unit testing toolkit CAUT [5] . In this section, some details on the experiments will be given. Also, some typical results will be shown, and the explanations to them will be given. Our experiments are conducted on 2.66GHz Intel Core i7 CPU running Windows 7 with 6GB RAM, which provides eight logical processors.

3.1 Experiment preparation

The experimental examples mainly come from SIR [1], including *bash*, *flex*, *grep*, *make*, *printtoken2* and *schedule*. Other examples are *algebra linear* [2] and *micro OpenGL core* (*c00nGL*) [3]. We selected parts of those programs but not whole programs to ensure that the testing time of each experiment was less than 15 minutes in the single core mode. For each example, we tested every separated function one by one in the target program and then summed the data of every tested unit (such as feasible paths) up as the result data. The calling dependencies in the unit under test

[5] The tool is available upon request.

Program	Units	Lines	Feasible Paths	Single-Core(ms)	Dual-Core(ms)	Single:Dual
algebra linear	27	3240	1657	723598	553444	130.74%
bash	35	1170	2002	336139	257466	130.56%
c00nGL	26	1282	226	76242	60864	125.27%
flex	25	538	3150	758809	587220	129.22%
grep	19	1215	505	101050	95835	105.44%
make	26	786	1769	136716	128294	106.56 %
printtoken2	13	359	47	176574	132333	133.43 %
schedule	16	147	100	6140	5659	108.50%

Table 1
Experimental Results

were replaced by mock functions. Besides, environment inputs shall be transformed to arguments of the testing unit, as they may disturb program paths.

The scheduler function $\mathcal{H}$ we adopted is a general hash function with the range $[0, 2^{32})$, while the function $\mathcal{F}$ divides the range of the $\mathcal{H}$ equitable to every computing unit. The functions are defined as follows:

```
unsigned int H(p) {
    unsigned int hash = 0;
    for each node value n in p
        hash = (hash << 5) + hash + n;
    return hash;
}

unsigned int F(p) {
    unsigned int worker = H(p) mod MAX_UNIT_CNT;
    return worker;
}
```

The above scheduler we adopted is based on our experimental tries. As it is discussed in the last section, it is selected according to the observations, although this instance of scheduler is not guaranteed to be the best solution.

3.2 Results

The experimental result is shown in Table 1, where the fourth column shows all the feasible paths of each example we tested. The last three columns show the time cost ratio between sequential (Single Core) and parallel (Dual Core) mode. It is clearly demonstrated that time cost of the parallel is less than the one of sequential mode with the given hardware resources given. Because of the scheduler function behaves differently on different examples the accelerated percentage (the last column) floats in a wide range (the lowest for *grep* costs 105.44% while the highest for *printtoken2* costs 133.43%). In a good one, e. g. *printtoken2*, the scheduler function may assign the generated paths uniformly on the two processers. The statistic data also supports our reasoning. In *printtoken2*, 315 cross-cpu tasks were sent to the one processor while the other processor received 304 cross-cpu tasks. Taking *grep* as another example, the performance of our parallel algorithm behaves not well enough. The experimental data shows that the paths allocation is not average for two processors: one is assigned to 782 cross-cpu tasks, while the other even only has 208 cross-cpu tasks, and computing tasks from 14 of 19 functions completely cannot be parallelized. To explain this reason, we analyzed the source code of *grep*. We found that many paths in *grep* program are short, which leads to the bad performance of our adopted scheduler function, because it may map the short paths to the same processor. The other examples which behaves not well in the parallel algorithm have the same reason.

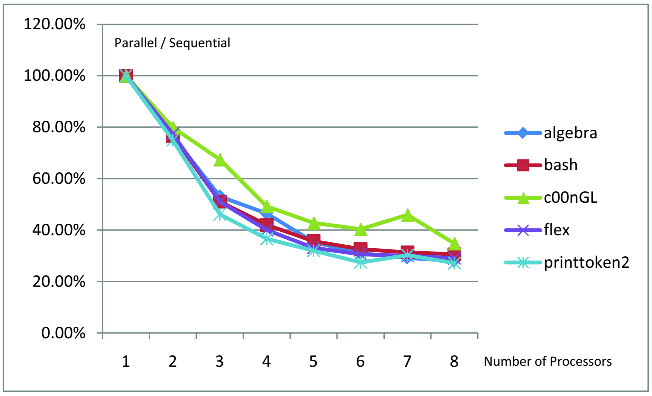

Fig. 4. Acceleration Ratio on Different Number of Processors

The other experiment shows the trend of performance boosting by increasing processors for five examples we listed previously. Figure 4 shows the result where the x axis presents the number of total processors and the y axis is the ratio between the time cost of parallel testing and the cost of sequential one. In Figure 4, we can easily see that the performance increases with the adding of processor numbers. The acceleration ratio can reach almost 30% with eight processors for those examples, which means that our parallel algorithm is very effective and can save nearly 70% time cost compared to the sequential concolic testing. Furthermore, the five curves also tell that the threshold of performance boosting may be arrived. It is obvious that some of curves drop more rapidly than others, but the trend of all of them tends to be flat. The reason is that in our model all the path schedulers are local and run in parallel, and they will send the paths to other processors to be handled. With the increasing number of processors, the communication cost (even it is asynchronous) will increase as well.

4 Conclusion

This paper gives a different perspective on the performance improvement of concolic testing technique by introducing a parallel algorithm. This kind of method is able to fully utilize resources of hardware, so the performance can be gained by increasing hardware processors or computation nodes in the distributed system. The contribution of our work is to apply parallel capability to traditional concolic testing with a low-synchronization framework. The parallel algorithm has been implemented and integrated into CAUT. The usability of the parallel approach is further confirmed by the application of CAUT. Comparing with other scalable test data generation techniques, the parallel model provides a practical approach for them.

Acknowledgement

Xiao Yu is partially supported by NFSC No. 61021004. Sun Shuai is partially supported by NFSC No. 90818024. Geguang Pu is partially supported by Fundamental Research Funds for the Central Universities and 863 Project No. 2009AA010313. Jiang Siyuan is supported by Project NFSC No. 61061130541. Zheng Wang is partially supported by Shanghai Leading Academic Discipline Project No. B412.

References

[1] Software-artifact infrastructure repository. http://sir.unl.edu/portal/index.html.

[2] Algebra linear, 2003. http://tech.groups.yahoo.com/group/mathc/.

[3] micro opengl core, 2005. http://sourceforge.net/projects/c00ngl/.

[4] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS'08/ETAPS'08: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: attacking path explosion in constraint-based test generation. In *TACAS'08/ETAPS'08: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 351–366, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[7] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. *SPIN*, 2005.

[8] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[9] P. Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[11] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[12] B. Korel. A dynamic approach of test data generation. *Software Maintenance*, pages 311–317, November 1990.

[13] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam. Using symbolic execution to guide test generation: Research articles. *Softw. Test. Verif. Reliab.*, 15(1):41–61, 2005.

[14] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[15] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[16] N. Tillmann and J. De Halleux. Pex: white box test generation for .net. In *TAP'08: Proceedings of the 2nd international conference on Tests and proofs*, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM.

[18] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 149, Washington, DC, USA, 2002. IEEE Computer Society.

[19] Z. Wang, X. Yu, T. Sun, G. Pu, Z. Ding, and J. Hu. Test data generation for derived types in c program. In *TASE '09: Proceedings of the 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 155–162, Washington, DC, USA, 2009. IEEE Computer Society.

[20] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 196–205, Washington, DC, USA, 2004. IEEE Computer Society.

[21] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, Edinburgh, UK, April 2005.