An adaptive task creation strategy for work-stealing scheduling

Conference Paper · April 2010

DOI: 10.1145/1772954.1772992 · Source: DBLP

An Adaptive Task Creation Strategy for Work-Stealing Scheduling

Lei Wang
[1]Institute of Computing Technology, Chinese Academy of Sciences
[2]Graduate University of Chinese Academy of Sciences
Beijing, China
wlei@ict.ac.cn

Huimin Cui
[1]Institute of Computing Technology, Chinese Academy of Sciences
[2]Graduate University of Chinese Academy of Sciences
Beijing, China
cuihm@ict.ac.cn

Yuelu Duan
Department of Computer Science, University of Illinois at Urbana-Champaign
duan11@illinois.edu

Fang Lu
[1]Institute of Computing Technology, Chinese Academy of Sciences
[2]Graduate University of Chinese Academy of Sciences
Beijing, China
flv@ict.ac.cn

Xiaobing Feng
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
fxb@ict.ac.cn

Pen-Chung Yew
[1]Department of Computer Science and Engineering, University of Minnesota, MN 55455 U.S.A
[2]Institute of Information Science, Academia Sinica, Taiwan
yew@cs.umn.edu

Abstract

Work-stealing is a key technique in many multi-threading programming languages to get good load balancing. The current work-stealing techniques have a high implementation overhead in some applications and require a large amount of memory space for data copying to assure correctness. They also cannot handle many application programs that have an unbalanced call tree or have no definitive working sets.

In this paper, we propose a new adaptive task creation strategy, called AdaptiveTC, which supports effective work-stealing schemes and also handles the above mentioned problems effectively. As shown in some experimental results, AdaptiveTC runs 2.71x faster than Cilk and 1.72x faster than Tascell for the 16-queen problem with 8 threads.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.3.4 [*Processors*]: Compilers, Run-time environments

General Terms Design, Languages, Management, Performance

Keywords adaptive, work-stealing, task granularity, backtracking search

1. Introduction

With the wide adoption of multi-threading techniques, many parallel programming languages such as Cilk [4] [10], X10 [5], and OpenMP3.0 [1], have provided their support for task-level parallelism. They define conceptually similar concurrent constructs, that include Cilk's *spawn-sync*, X10's *asyn-finish* and OpenMP3.0's *omp task-taskwait*. In their support, work-stealing is one of the key techniques used in the runtime system to help load balancing. Generally, in work-stealing, each thread maintains a double-ended queue (called *d-e-que*, in this paper) for ready tasks. An owner thread pushes and pops ready tasks to and from its own d-e-que's ***tail*** end. Each thread steals tasks from the ***head*** of the d-e-que in other threads when its own d-e-que is empty. Hence, when stealing tasks, the thief thread can run in parallel with the victim thread's execution. A thread could also suspend a waiting task to execute other ready tasks. With this scheme, work-stealing achieves good load balancing [4] [10] [3].

However, there are still problems in the current work-stealing techniques. Firstly, the overhead of task creation and d-e-que management could be very high in some applications. Secondly, in some popular applications such as backtracking search, branch-and-bound search and game trees, the overhead of allocating and copying workspaces for each child task to assure correctness, called *workspace copying* [13], could be quite high, and it could badly hurt the performance. Finally, the d-e-que is often implemented as a fixed-size array in Cilk, which is prone to overflow.

Tascell [13] uses an improved scheduling technique based on backtracking to solve some of these problems. In Tascell, the task is stored in a thread's execution stack instead of in a d-e-que. When a thread receives a task request from an idle thread, it backtracks through the chain of nested function calls, and creates a task for the requesting thread. It then returns to the top frame of its execution stack and resume its own execution. Hence, when responding to a request, the responding thread cannot run in parallel with the request thread. Tascell also delays workspace copying as much as possible. Its copying overhead could thus be significantly reduced, and it could often achieve a higher performance than Cilk in some important applications [13].

CGO'10 April 24–28, 2010, Toronto, Ontario, Canada.

However, Tascell still could not achieve good load balancing in some applications. For example, Tascell cannot suspend a waiting task (and has to wait for its child tasks to complete) because it uses its execution stack to store the task information. If a waiting task is suspended and starts to run other ready tasks, the stack frame of the waiting task will be destroyed and cannot be resumed. Taking 16-queens as an example, the waiting time for child tasks could be as high as 16.73% of the total execution time with 8 threads (see section 5.2).

In this paper, we proposed a new adaptive task creation strategy, called AdaptiveTC, to support work-stealing. When executing a *spawn* statement, AdaptiveTC can generate a *task*, a function call (a *fake task*, refer to Section 3), or a *special task*. The *task* is responsible for keeping idle threads busy; the *fake task* is responsible for improving performance; and the *special task* is used to switch a thread from a *fake task* to a *task* for good load balancing. In addition, AdaptiveTC introduces a new data attribute, call *taskprivate*, for *workspace variables* common in applications such as backtracking search, branch-and-bound search and game trees. Allocating and copying a new *taskprivate* variable for a child task is only performed in the *task*, not in the *fake task*. AdaptiveTC can adaptively switch between *tasks* and *fake tasks* to get a better performance.

In AdaptiveTC, a specified number of *tasks* are created initially to keep all threads busy, and then a *fake task* is executed in each thread. During the execution, except when some thread becomes idle, at which point a busy thread generates a *special task* to transition back from the *fake task* to a *task*, each busy thread would avoid creating more *tasks* into its d-e-que. As a result, the number of tasks created is smaller than that of Cilk. Hence, it reduces the overhead of task creation, d-e-que management, and workspace copying, without sacrificing good load balancing, and is less prone to d-e-que overflow. The cost of managing d-e-que in AdaptiveTC is thus much less than that of managing nested functions on the execution stack in Tascell.

Our experiments show that AdaptiveTC outperforms Cilk and Tascell in many common applications. For example, it runs 2.71 times faster than Cilk and 1.72 times faster than Tascell for the 16-queen problem with 8 threads (see section 5.1).

The contributions of this paper are:

- An adaptive task creation strategy is proposed to support work-stealing techniques for better load balancing and lower system implementation overhead. It reduces the number of tasks created with a better control of the task granularities, hence, could significantly reduced the overhead of task creation. It is also less prone to d-e-que overflow. AdaptiveTC is very suitable for many applications that have no definitive working sets, and could achieve a much better load balancing for applications with unbalanced call trees.
- A new data attribute *taskprivate* is introduced for workspace variables to improve the programmability and to further reduce the cost of workspace copying, and thus achieving a higher performance.

The rest of the paper is organized as follows. We first present some related work in Section 2. In Section 3, we introduce our adaptive task creation strategy. In Section 4, we describe the implementation of our approach. Some experimental studies are presented in Section 5, and in Section 6, we conclude our paper.

2. Related work

Cut-off strategies: several prior studies [16] [14] [9] [8] [7] used cut-off strategies to control the recursion depth of function calls during the task generation, and thus could reduce the overhead of task creation. These strategies could also control the task granularities by reducing the number of small tasks. A basic cut-off strategy usually specifies a depth of recursion in a *computation tree* (or *call tree*) beyond which no tasks could be created (see Figure 1.a). It has been found that such strategies work very well for balanced computation trees. However, for unbalanced computation trees, such cut-off strategies are known to cause starvation, i.e. some threads might be forced to become idle for lack of tasks to work on [14].

Three approaches were generally used to implement a cut-off strategy. The first is to ask the programmer to provide a cut-off depth for the recursion [16] [14], or using the runtime system to set a common default depth, for all applications [9]. Both are very simple, but cannot adapt to a changing environment. The second approach is *batching*, i.e. to set the cut-off depth according to the current size of the d-e-que and adaptively control the granularity of parallel tasks [8]. However, this approach needs the programmers to set a sequential processing threshold, and to carry out performance tuning manually. The third approach is *profiling* [7]. It adopts a working set profiling algorithm, and then uses the profiling information to perform cut-offs. It works well for some divide-and-conquer applications in which all parallel tasks deal with different parts of a working set. However, it becomes less effective in some important applications such as backtracking search, branch-and-bound search and game trees, in which there are no definitive working sets during the execution.

The AdaptiveTC can adaptively create tasks to keep all threads busy, and also adaptively control the task granularities to reduce the overhead of task creation.

Workspace copying: This problem is introduced by work-stealing scheduling. In some popular applications such as backtracking search, branch-and-bound search and game trees, solution space variables and states of nodes, such as chessboards and pieces, are all stored in workspaces. In order to assure correctness, programmer needs to allocate memory space, and copy the value of the parent's workspace variables to each child task.

The work proposed in [2] pointed out that workspace variables increase programming difficulties. Workspace variables are usually C arrays or pointers of data structure, if a pointer is used in an OpenMP 3.0's *firstprivate* directive, only the pointer is captured. In order to capture the value of the data structure, the programmer must deal with them inside each task, including proper synchronization, and it could become quite complicated to write such parallel programs. We found that by supporting such workspace variables in a programming language such as providing a special attribute for those workspace variables, it significantly improves the programmability of those applications.

Cilk supports SYNCHED variables to conserve space resources [11]. A SYNCHED variable has a value of 1 if the scheduler can guarantee that there is no stolen child task in the current task and 0 otherwise. By testing the SYNCHED variable, it would allow some child tasks to reuse the same memory space and store their private data so that the *space overhead* could be drastically reduced. However, all child tasks still have to copy the data from their parent tasks, and hence, the *time overhead* is not reduced.

In AdaptiveTC, we propose a new data attribute ***taskprivate*** that works with the controlled task granularities to reduce both space and time overhead, and also improves the programmability as mentioned before.

D-e-que: In [6], it presents a work-stealing d-e-que using a buffer pool that does not have the overflow problem. In [15], it proposes techniques to expand the size of a d-e-que with automatic garbage collection. As AdaptiveTC pushes fewer tasks into d-e-ques, it is less prone to overflow.

Adaptive work-stealing scheduler: SLAW [12] adaptively switches between work-first and help-first scheduling policies, which has the possibility of running parallel programs to completion when the sequential version overflows stack. In contrast, AdaptiveTC adaptively switches between tasks and fake tasks to get a better performance.

3. An adaptive task creation strategy for work-stealing

As mentioned in Section 1, when executing a *spawn* statement, AdaptiveTC can generate a *task*, a function call (a *fake task*), or a *special task*. The *task* is pushed to the d-e-que's *tail* end and can be stolen by idle threads; the *fake task* is only a plain recursive function and is never pushed into the d-e-que; the *special task* is pushed into the *tail* of d-e-que and marks a transition point from the *fake task* back to the *task*. Allocating and copying a new *taskprivate* variable for a child task is only performed in a *task*, not in a *fake task*. AdaptiveTC can adaptively switch between *tasks* and *fake tasks* to get a better performance. In AdaptiveTC, a specified number of tasks are created initially to keep all threads busy. During the execution, except when some thread becomes idle, all busy threads would avoid creating additional tasks into their d-e-ques. A randomized work-stealing algorithm with our adaptive task creation strategy is described in more detail as follows.

If the number of active threads is capped at N, the *cut-off* of a recursive call tree beyond which no tasks should be created, is initially set to $\lceil \log N \rceil$ by the runtime system. The depth of the recursive call chain for the original task is considered to be 0.

At the beginning, all d-e-ques are empty. Then, the root task is placed in one thread's d-e-que, while other threads start work stealing. A thread obtains work by popping the task from the d-e-que's *tail* end, and continues executing this task's instructions until this task spawns, terminates, or reaches a synchronization point, in which case, it performs according to the following rules.

Each active *worker/victim thread* (a *victim thread* is a thread whose tasks in its d-e-que have been stolen by other worker threads) will use the following scheme:

Spawn: (a *spawn* statement, task α spawns a child task β)

1. As shown in Figure 1.a, when the depth of task α (the depth of the recursive call tree) is smaller than the *cut-off*, a thread will push task α into the *tail* of the d-e-que, generate a new task β, and begin to execute task β. If the *cut-off* has been reached, a thread will not push task α into the *tail* of the d-e-que, but continue the main execution of recursive functions down the call tree, called the *fake task* (because no real task was generated for its execution), without creating new tasks, thus will not incur any task creation overhead.
2. However, before the *fake task* continues to execute down the recursive call tree, it will first check whether there is an idle thread waiting to steal tasks. If *not*, no new tasks will be generated; if *yes*, it creates a *special task* for itself to resume, and pushes the *special task* into the *tail* of its d-e-que, and then continues its own execution. The depth of the special task's child will be set to 0. As the depth of the special task's child task is 0, it generates and pushes more child tasks into its own d-e-que later on. Other threads could steal the descendant tasks. The *special task* in the d-e-que marks a transition point from the main fake task to its child tasks. It stores all of the task information of the main fake task, and thus cannot be stolen. It also has to wait for its child tasks to complete before its resumption for execution; otherwise, we will not be able to resume the main fake task when we complete the child tasks.

Terminate: (a *return* statement, task α terminates and returns to its parent task γ)

- The task α is popped from the d-e-que's *tail* end first. If task α is the root task, the schedule ends. Otherwise, a thread checks its d-e-que. (1) If the d-e-que contains any task, the thread will pop a task γ from the d-e-que's *tail* end and begin to execute task γ. (2) If the d-e-que is empty, and task α is spawned by the thread, i.e. the parent task γ is stolen by another thread, the thread will return immediately. (3) If the d-e-que is empty, and task α is stolen by the thread, i.e. the thread returns to the runtime system code, the thread will inform the parent task γ that task α is completed, and check the status of the parent task γ. If the parent task γ is suspended, and all the child nodes of task γ are completed, the thread will begin to execute task γ; otherwise, the thread will begin its work stealing.

Reaching a synchronization point: (a *sync* statement, task α reaches a synchronization point)

- A thread checks whether all the child nodes of task α are completed. If *yes*, it will execute the instruction following the synchronization point in task α. If *no*, (1) if task α is a *task*, the thread will pop task α from the d-e-que's tail end, **suspend** task α, and then start stealing other task; (2) if task α is a *special task*, the thread will **wait** for its child tasks to complete before its resumption of execution.

Each *thief thread* will use the following scheme:

Steal: (when a thread begins work stealing)

1. A thread randomly selects a victim thread, and tries to steal task from the victim thread. If it succeeds, the thread will execute the new stolen task; if not, it will inform the victim thread that it needs a task, and try again, picking another victim thread at random. When a thief thread is attempting to steal a *special task*, it will steal the special task's child task instead, if there is any, to avoid the problem mentioned above (i.e. the resumption of the original fake task).
2. The thread executes the new stolen task, restores the task's state first, and then goes to the point after a spawn or synchronization instruction according to the new stolen task's state and executes the task's instructions.

In Figure1, there are 4 threads (p0, p1, p2, p3) that execute nodes in the *computation tree* (i.e. *call tree*), and the default *cut-off* is 2. Note that not all of the nodes in the tree are generated as tasks by the threads. Figure 1.a illustrates the starting stage, in which each thread executes a sub-tree, respectively, from nodes 2, 41, 7 and 44. During the execution, p3 steals task 0 from p1, and suspends task 0 as neither child task 1 nor 40 is completed. p3 then steals task 40 from p1, and continues to execute node 44 (the second child of task 40), but not node 41 (the first child of task 40), because node 41 was already under execution by p1. Each thread will then execute nodes down its respective sub-tree sequentially.

Then, at the beginning stage of Figure 1.b, p0, p1 and p3 have finished these sub-trees, respectively, from nodes 2, 41 and 44; p2 is executing a certain node in the sub-tree rooted at node 7, and there is only task 1 in p2's d-e-que. When p1 steals task 1 from p2, it suspends task 1 as its child node 7 is not completed yet. There are no tasks to be stolen at this time. As the sub-tree rooted at the node 7 is larger than the other sub-trees, it is very likely that when p2 is executing a node in the sub-tree, say node 12, it could find that some other thread needs a task. As described in worker thread Spawn's step 2, p2 will create a *special task* 12 for node 12, and push it into the *tail* of its d-e-que. p2 will then create a task and push it into the *tail* of d-e-que for nodes 13 and 14 sequentially. In one scenario, p0 would steal task 13 from p2, and p1 would steal task 14 from p2. At this point, as $H >= T$ in the p2's d-e-que, there is no task in the p2's d-e-que to steal. p3 would steal task 13 from p0, suspend it as neither child tasks 14 nor 24 is completed, and then steal task 24 from p0 (illustrated in Figure 1.b). Now the threads p0, p1, p2 and p3 execute sub-trees 25, 18, 15 and 30.

Also, as illustrated in Figure 1.c, p2 finishes the sub-tree rooted at node 15 and returns to node 14. As task 14 is stolen by p1, p2 will return to node 13 immediately. Also, because task 13 is stolen as well, p2 will return to node 12 immediately and execute the sub-tree rooted at node 35. In this way, our adaptive task creation strategy only generates 20 tasks, while Cilk generates 49 tasks. Even though, Cilk could use its cut-off strategy to reduce the number of tasks generated, it could be at the risk of creating an unbalanced call tree.

With the same starting stage in Figure 1.a, Tascell would make p0 wait on task 1 for its child task 7's completion after it finishes its own sub-tree rooted at node 2. Thus, only p1, p2 and p3 could expect to execute the sub-tree rooted at node 7. However, as illustrated in Figure 1.b and 1.c, with the strategy of AdaptiveTC, all the four threads could expect to execute the sub-tree rooted at node 7 together, which would achieve better dynamic load balancing than Tascell.

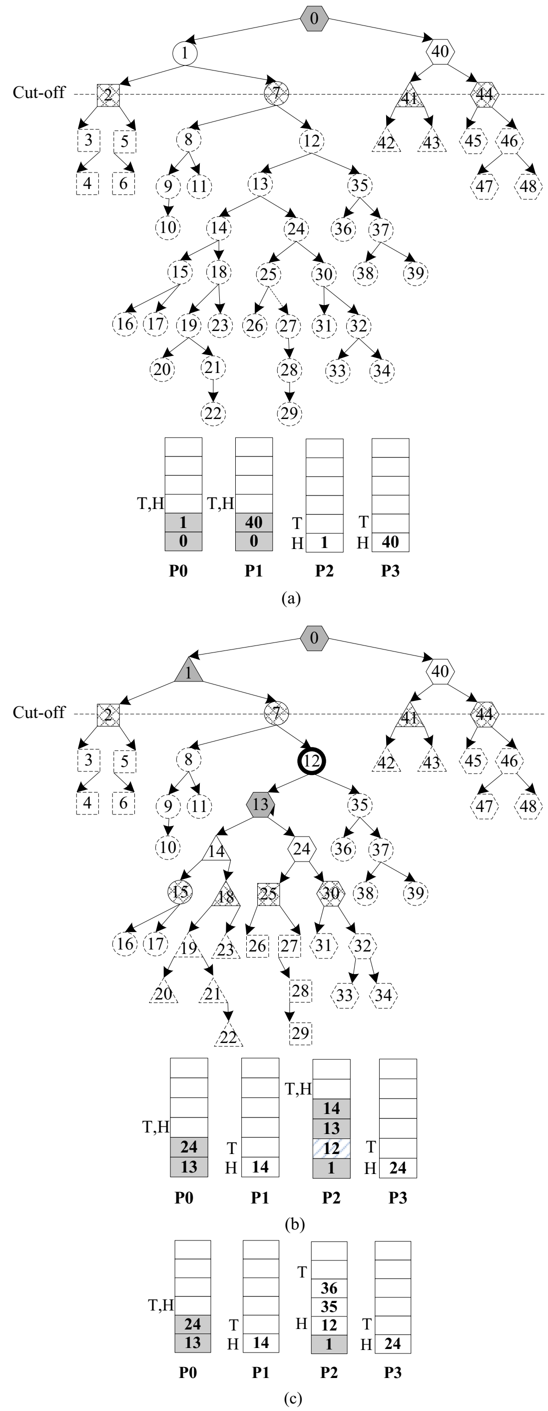


Figure 1. The status of a call tree and d-e-ques in active threads. In the call tree, nodes with a dotted boundary are executed sequentially in a thread and are not created as tasks. Solid boundary nodes are created as tasks, among which the grid-shaded ones are not pushed into d-e-ques and the non-shaded ones are pushed into d-e-ques. The grey ones are suspended. Node 12 is special task. The square nodes are executed in thread p0, the triangle ones are in p1, the circle ones in p2, and the hexagon ones in p3. In the d-e-ques, T indicates the *tail* of the d-e-que, H indicates the *head* of the d-e-que. Figure 1 (a) shows the starting stage. In Figure 1 (b), the special task 12 can be pushed into p2's d-e-que. Figure 1 (c) shows the next stage of Figure 1 (b).

4. AdaptiveTC - A comprehensive parallel programming environment

AdaptiveTC is a comprehensive parallel programming environment that includes a parallel programming language, a compiler and a runtime system. The parallel language is an extended Cilk. The key features of the Cilk language include the inclusion of parallelism and synchronization semantics through the *spawn* and *sync* keywords. AdaptiveTC extends the Cilk language further by providing the ***taskprivate*** keyword to specify data storage. Our compiler translates the extended Cilk program to a C program that could take advantage of an improved runtime library. The compiler generates five different versions of the code for each task, and these five versions will each generate a *task*, a function call (a *fake task*), or a *special task*. AdaptiveTC uses a finite state machine (FSM) in each thread, and executes a different version of the code depending on the state the thread is in. Transition from one state to another only requires a few concise steps followed by a transition to a different version of the code. This FSM implementation makes it easier to switch a thread from fake tasks to tasks, and then generate more tasks for other threads to steal, while at the same time minimizing the number of tasks stored in the d-e-que and the amount of workspace copying.

4.1 A new data attribute – *taskprivate*

A variable with a *taskprivate* attribute will have **no** storage association with the same named variable in other tasks. A *taskprivate* variable inherits the value of its parent task's *taskprivate* variable. Only parameters or local variables can be declared as *taskprivate*, and *taskprivate* could be declared on a pointer or an array. For example,

taskprivate: (*address) (an expression to calculate the size of the taskprivate variable);

In the n-queens problem, it computes the number of all possible placements for n queens in a chessboard with only one queen in any vertical, horizontal and diagonal line. It is a typical backtracking search problem. The implementation of the problem needs to maintain a chessboard that indicates all current positions of the queens. The chessboard variable can be declared as a *taskprivate* in AdaptiveTC as follows:

// $depth$ is the numerical ID of a queen; n is the total number of queens; *x[]* is the chessboard.

cilk int nqueens(int $depth$, int n, char* x)
taskprivate: (*x) (n * sizeof(char));

As the chessboard variable could be accessed and modified by multiple tasks concurrently, in Cilk, the programmer needs to allocate memory space, and copy the value of the parent's chessboard variable to each child task in order to assure correctness. There are two ways to do it: one is to use Cilk_alloca() function to allocate a new chessboard variable for each child task; the other is to allocate a new chessboard variable using malloc function, and free it at the end of the child task. In either way, the programmer must take special care to the chessboard variable. Cilk also provides SYNCHED variables to conserve memory space [11]. Hence, the *taskprivate* data attribute we proposed significantly improves the programmability of those applications.

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9*9 grid so that each column, each row, and each of the nine 3*3 blocks contains the digits from 1 to 9 only one time each. Appendix A is an AdaptiveTC program for Sudoku. Here, the program Sudoku finds all solutions for a given grid. The parameter *st* is a *taskprivate* variable.

In AdaptiveTC, *fake tasks* and *tasks* handle *taskprivate* variable in different ways. In *fake tasks*, the *taskprivate* keyword is ignored. But in *tasks*, allocating and copying a new *taskprivate* variable for a child task is performed in order to assure correctness. The chessboard variable is handled as follows:

In a *fake task*,
  x[*depth*] = *j*;
  sn += nqueens(*depth* + 1, *n*, *x*);
And in a *task*,
  char **tmp_x*;
  x[*depth*] = *j*;
  tmp_x = Cilk_alloca(*n* * sizeof(char));
  memcpy(*tmp_x*, *x*, *n* * sizeof(char));
  sn += nqueens(*depth* + 1, *n*, *tmp_x*);

In AdaptiveTC, as the number of tasks created is very small, it reduces the cost of workspace copying, and thus achieves a higher performance.

4.2 AdaptiveTC compilation strategy

To support the adaptive task creation strategy and to achieve a high performance, the AdaptiveTC compiler generates five different versions of the code for each task: a *fast* version, a *check* version, a *fast_2* version, a *sequence* version and a *slow* version. These five different versions provide the support of various work required at different stages of the execution. Figure 2 shows the relationship of these five versions at runtime during the adaptive task generation. The *fast*, *fast_2* and *slow* versions generate *tasks*. The *sequence* version generates *fake tasks*. The *check* version is similar to the *sequence* version when no other thread needs to steal a task. However, when any other thread needs a new task, it will generate a *special task* for its current sequential execution, and push it into the *tail* of d-e-que, so it could generate its child tasks into the d-e-que. The AdaptiveTC compiler ignores the *taskprivate* keyword in the *sequence* version and the *fake tasks* part of the *check* version, but allocates and copies a new *taskprivate* variable for a child task in the *fast*, *fast_2*, *slow* versions and the *special task* part of the *check* version (see section 4.1). The runtime system links together the actions of the five versions to produce a complete AdaptiveTC implementation with a high performance.

Appendix B shows a *fast* version of a Sudoku task in AdaptiveTC. When the *fast* version runs for the first time, the depth of the recursive call tree is 0. A task is created at the entry of the fast version and is freed at its exit. 1) When the depth is smaller than the *cut-off*, the state of the *fast* version is saved, and the task is pushed to the *tail* of the d-e-que. Then, the *fast* version of the child task is called with the depth incremented by 1. After the child task returns, it pops the saved task from the *tail* of the d-e-que, and

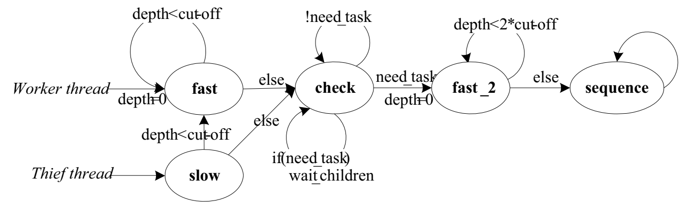


Figure 2. The relationship of the five versions in AdaptiveTC

check whether the task has been stolen. If *yes*, the *fast* version returns with a dummy value immediately. If *not*, it continues to run the next child task. 2) When the depth reaches *cut-off*, the *fast* version will call the *check* version without pushing the task into the d-e-que. In Figure 1, nodes 0, 1 and 40 use one of the fast versions before the *cut-off*, and nodes 2, 41, 7 and 44 use the fast versions beyond the *cut-off*.

In the *fast* version, all sync statements are translated to no-ops. Except for a *special task*, only parent tasks are allowed to be stolen, therefore all child nodes have completed when executing *sync* statements in the *fast* version. No operations are thus required for a *sync* statement.

Appendix C is a *check* version of a Sudoku task in AdaptiveTC. The *check* version checks whether other threads need tasks. If *not*, it calls its child task's *check* version recursively. If *yes*, it generates a *special task*, pushes the task into the tail of the d-e-que, and calls the child task's *fast_2* version with its depth set to 0. After the child task's *fast_2* version returns, it pops the *special task* and check whether its child task has been stolen. If *yes*, the *stolen_flag* variable is set to true. The *check* version continues to run the next child task's *fast_2* version until all child tasks are executed (using their *fast_2* version). At the synchronization point, if the *stolen_flag* variable is true, the *special task* will wait until all its child tasks are completed. In Figure 1, node 3, 5, 4, 6, 8, 9, 11, 10, 42, 43, 45, 46, 47 and 48 will use their *check* versions. The *special task* is node 12.

The *fast_2* version is a variant of the *fast* version with two differences. One is that the *cut-off* in *fast_2* is twice of that in the *fast* version. The other is that when the *cut-off* is reached, the *fast_2* version will call the *sequence* version, but not the *check* version as the *fast* version does. When the *fast_2* version is executed, the number of tasks generated by the *fast* version is not enough to keep all threads busy, so more tasks are generated in the *fast_2* version. The *sequence* version is a regular recursive function. In Figure 1, nodes 13, 14, 35, 36, 37 and 24 use their *fast_2* version before the *cut-off*, and nodes 15, 18, 25, 30, 38 and 39 use the *fast_2* version beyond the *cut-off*. Other nodes use their *sequence* version.

The *slow* version is used at the start of all stolen tasks. When a *thief thread* steals a task, the *slow* version of the task will be executed. It restores its program counter using a *goto* statement, and also restores its local variables and the depth for the task. Depending on whether the depth reaches the *cut-off* yet, the *slow* version will call either the *fast* version or the *check* version. At the synchronization point, a call to the runtime system, which checks whether all the child nodes of the task are completed, is inserted by compiler. If all the child nodes are completed, the thread will execute the next instructions of a synchronization point. If *not*, the thread will pop the task from the d-e-que's *tail* end, suspend the task, and then start stealing other task.

4.3 The runtime system

Cilk's work-stealing mechanism is based on a Dijkstra-like, shared-memory, mutual exclusive protocol called the THE protocol [10]. As both victim and thief operate directly on the victim's d-e-que,

```
Worker / Victim 0

push(){  T++;  }
pop(){
   T--;
   MEMBAR;
   if(H > T){
      T++;
      lock(worker.L);
      T--;
      if(H > T){
         unlock(worker.L);
         return FAILURE;
      }
      unlock(worker.L);
   }
   return SUCCESS;
}
```

(a)

```
Worker / Victim  1

pop_specialtask() {
   lock(worker.L);
   T--;
   if(H > T){
      H  = T;
      unlock(worker.L);
      return FAILURE;
   }
   unlock(worker.L);
   return SUCCESS;
}
```

(b)

```
Worker / Victim  2

sync_specialtask(){
   while(1){
      lock(worker.L);
      if (!children(f)) {
         unlock(worker.L);
         return;
      }
      unlock(worker.L);
      usleep(100);
   }
}
```

(c)

```
Thief 0

steal(){
   lock(victim.L);
   H++;
   MEMBAR;
   if(H > T){
      H--;
      stolen_num ++;
      if(stolen_num > max_stolen_num)
         need_task = 1;
      unlock(victim.L);
      return FAILURE;
   }
   stolen_num = 0;
   need_task = 0;
   unlock(victim.L);
   return SUCCESS;
}
```

(d)

```
Thief  1

steal_specialtask(){
   lock(victim.L);
   H += 2;
   MEMBAR;
   if(H > T){
      H -= 2;
      stolen_num ++;
      if(stolen_num > max_stolen_num)
         need_task = 1;
      unlock(victim.L);
      return FAILURE;
   }
   stolen_num = 0;
   need_task = 0;
   unlock(victim.L);
   return SUCCESS;
}
```

(e)

Figure 3. Pseudo code of a simplified THE protocol. (a), (b) and (c) show the action performed by a victim thread; (d) and (e) show the action of a thief thread.

race conditions will arise when a thief tries to steal the same task that its victim is attempting to pop. The THE protocol resolves such a race condition, and AdaptiveTC follows the THE protocol to implement the *special task* in the d-e-que.

Figure 3 shows the pseudo code of a simplified THE protocol used in AdaptiveTC. The code assumes that the d-e-que is implemented as a task array. T is the *tail* of the d-e-que, the first unused element in the array, and H is the *head* of the d-e-que, the first task in the array. Indices grow from the *head* to the *tail* so that under normal conditions, we have $T >= H$.

In *fast*, *fast_2* and *slow* versions, the *worker* thread uses a *push* operation to push a task into the *tail* of d-e-que before calling a *parallel* version. It also uses a *pop* operation to pop the task after calling the *parallel* version. In the *check* version, the worker thread uses a *push* operation to push a *special task* into the *tail* of the d-e-que before calling the *fast_2* version. It performs a *pop_specialtask* operation to pop the *special task* after calling the *fast_2* version, and a *sync_specialtask* operation to wait for the child tasks to complete at the synchronization point. In a *pop_specialtask* operation, when the *special task*'s child task is stolen, H is reset to T. The intention of this reset is to ensure the *special task* to be the *head* of the d-e-que because the *special task* could never be stolen.

A thief needs to get *victim.Lock* before attempting to steal the task at the *head* of d-e-que. Hence, only one thief may steal from the d-e-que at a time. When a thief attempts to steal a *special task*, it will steal the *special task*'s child task.

To notify a busy thread that some other idle thread needs tasks, the thief thread (an idle thread) increases the *stolen_num* of the victim thread (a busy thread). When the *stolen_num* exceeds the *max_stolen_num*, the *need_task* in the victim thread is set to *true*.

Nqueen-array(n)	The n-queens problem. It uses an array to record whether conflicts occur, and is more time efficient.
Nqueen-compute(n)	The n-queens problem. It traverses the chessboard to find out whether conflicts occur, and is more memory efficient.
Strimko	A logic puzzle. The objective is to fill in the given 7*7 grid so that each column, each row, and each stream contain the digits from 1 to 7 only once.
Knight's Tour	To find all solutions on a 6*6 chessboard. The knight is placed on an empty chessboard and moving according to the rules of the chess. It needs to visit each square on the chessboard exactly once.
Sudoku	To find all solutions for a given grid.
Pentomino(n)	To find all solutions to the Pentomino problem with n pieces (using additional pieces and an expanded board for $n > 12$).
Fib(n)	To compute recursively the n-th Fibonacci number.
Comp(n)	To compare array elements a_i and b_j for all $0 <= i, j < n$.

Table 1. Benchmark programs

As a result, the victim thread would notice that other threads need tasks. When the thief thread succeeds in stealing a task, it clears the victim thread's *stolen_num* and *need_task*. The default *max_stolen_num* is set to *20* in our runtime system.

pop_specialtask (in Figure 3.b): When $H < T$, no child task of the *special task* is stolen; otherwise, the child task is stolen, and H is reset to T. The intention of this reset is to ensure the *special task* to be the head of the d-e-que as the *special task* is never stolen.

sync_specialtask (in Figure 3.c): the *special task* awaits its child task to complete.

steal_specialtask (in Figure 3.e): the *special task* can never be stolen, and an attempt to steal it will lead to its child tasks being stolen.

5. Experimental results

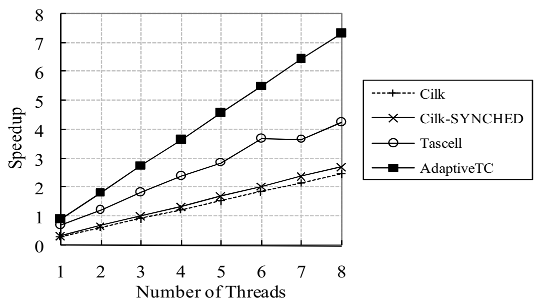
In this section, we present some experimental results and try to compare the performance of our AdaptiveTC with those in Cilk-5.4.6 and Tascell. We first give detailed experimental results, and then analyze the overheads of three systems, finally give the performance in unbalanced trees to evaluate the dynamic load balancing.

We perform such measurements on Intel multi-core SMP, 2-processor quad core Intel Xeon E5520 (2.26GHz, 8G memory). We compile all parallel benchmark programs with the Cilk-5.4.6 compiler using gcc with option -O3. All serial benchmark programs are compiled with gcc -O3 as well. The speedup is computed using the serial execution time as the baseline, and using the median execution time of 3 successive executions of its corresponding parallel version. We evaluate the performance of our AdaptiveTC using the benchmark programs in Table 1.

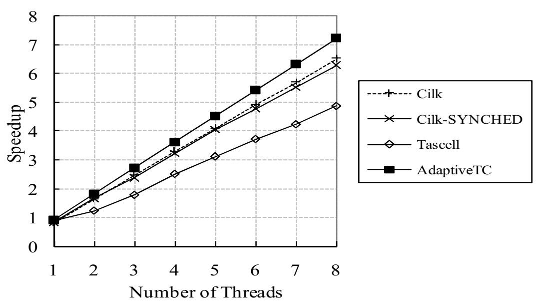
5.1 Detailed experimental results

The results in Figure 4 and Figure 5 show a significant performance improvement of the AdaptiveTC over Cilk in the range of 1.15x to 2.78x using 8 threads. In addition, from Figure 4 we can see that AdaptiveTC has a good scalability when the threads number increases. In *Nqueen-array*, *Strimko*, *Knight's tour*, *Sudoku* and *Pentomino*, reducing the cost of the workspace copying is the major performance contributor. In *Nqueen-compute*, *fib* and *comp*, reducing the cost of creating tasks and managing d-e-ques is another major performance contributor. It shows that the proposed adaptive task creation strategy in AdaptiveTC could be very efficient and effective in the implementation of work-stealing strategy.

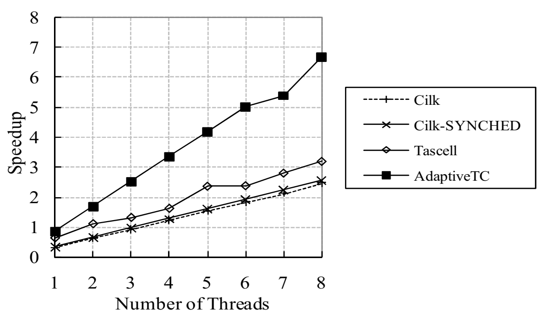
AdaptiveTC also achieves a higher performance than Tascell for most benchmarks. One reason is that the cost of creating tasks and managing d-e-ques in AdaptiveTC is much less than that of managing nested functions in Tascell; the other reason is that AdaptiveTC performs better dynamic load balancing than Tascell does(see sec-



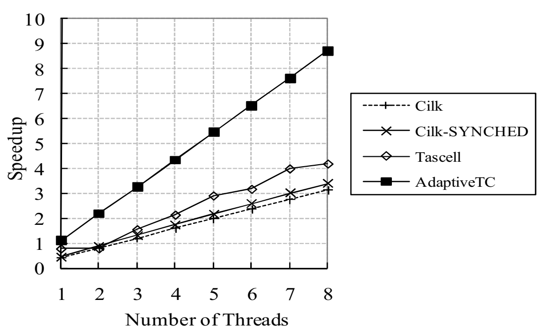
(a) Nqueen-array(16)



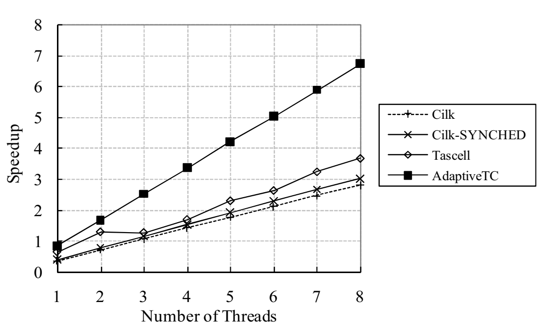
(b) Nqueen-compute(16)



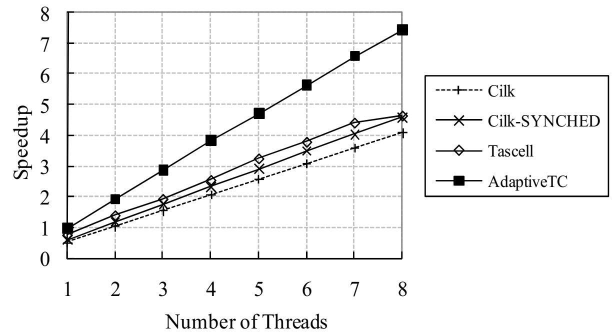
(c) Strimko



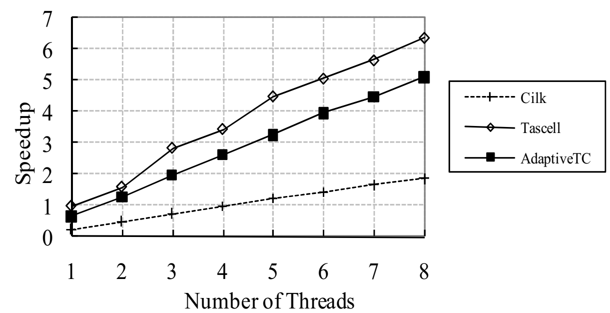
(d) Knight's Tour (6*6)



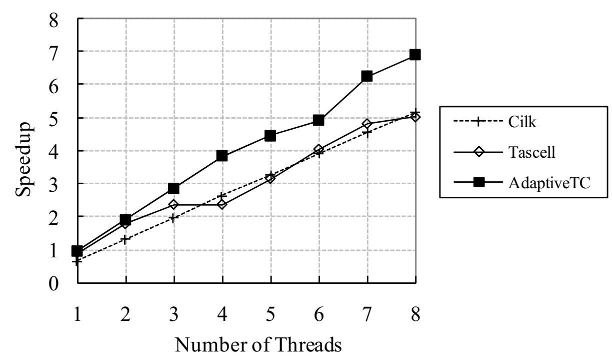
(e) Sudoku (input_balance tree)



(f) Pentomino (13)



(g) Fig(45)



(h) Comp (60000)

Figure 4. Speedup comparisons. Fib and Comp don't have *taskprivate* variables, therefore the speedup in (g) and (h) are against Cilk and Tascell only.

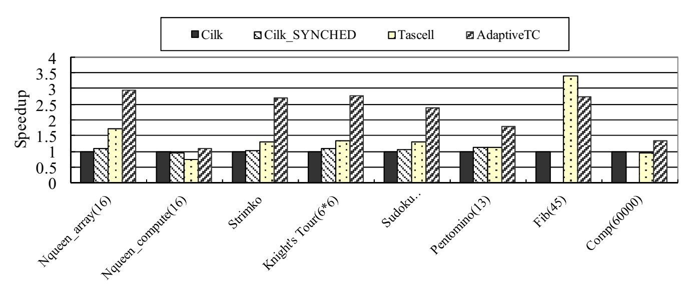


Figure 5. Speedup with 8 threads, baseline is Cilk's execution time.

	C	Tascell	Cilk	Cilk_SYNCHED	AdaptiveTC
Nqueen-array(16)	60.81	85.33 (1.4)	197.69 (3.25)	184.26 (3.03)	66.04 (1.09)
Nqueen-compute(16)	554.04	627.15 (1.13)	669.22 (1.21)	661.16 (1.19)	612.24 (1.11)
Strimko	262.97	423.24 (1.61)	839.03 (3.19)	813.01 (3.09)	315.55 (1.2)
Knight's Tour (6*6)	1322.51	1713.54 (1.3)	3307.32 (2.5)	3038.8 (2.3)	1217.56 (0.92)
Sudoku (balance_tree)	614.74	943.99 (1.54)	1717.09 (2.79)	1632.57 (2.66)	731.13 (1.19)
Pentomino (13)	8.74	11.64 (1.33)	16.73 (1.91)	14.83 (1.7)	9.176 (1.05)
Fib(45)	16.57	16.8 (1.01)	66.46 (4.01)	–	25.14 (1.52)
Comp(60000)	12.59	14.13 (1.12)	19.03 (1.51)	–	13.08 (1.04)

Table 2. Execution time in seconds (and relative time to sequential C programs) with one thread.

tion 5.2). The performance improvement over Tascell is in the range of 1.37x to 2.093x using 8 threads.

The only exception is *fib*. As shown in Figure 7.c, the cost of managing nested functions in Tascell is only 1.4% of the total execution time, while the cost of creating tasks and managing d-e-ques in AdaptiveTC is 51.7%; Tascell is thus 1.24x faster than AdaptiveTC. The main reason is that, in *fib*, there is almost no actual computation workload in each function. Hence, it increases the proportion of task creations and the d-e-que management cost substantially.

5.2 Overhead breakdown

We could basically break down the overheads of the three systems, AdaptiveTC, Cilk and Tascell as follows:

1. Overhead of AdaptiveTC = management of d-e-ques and task creations + *taskprivate* variables + THE protocol + waiting of child tasks to complete + task stealing overhead;
2. Overhead of Cilk = management of d-e-ques and task creations + workspace copying + THE protocol + task stealing overhead;
3. Overhead of Tascell = nested functions overhead + polling overhead+ waiting of child tasks to complete;

The cost of managing d-e-ques and creating tasks, workspace copying, *taskprivate* variables, and nested functions overhead could be measured by using only one thread, and the other costs need to be measured by running multiple threads.

From Table 2 and Figure 6, the overhead incurred in AdaptiveTC is lower than that in Cilk, and that is the main reason why AdaptiveTC could achieve a higher performance than Cilk for most benchmarks. However, in *fib*, the overhead in Tascell is much lower than that in the other two, thus Tascell gets the best performance on *fib*.

However, as shown in Figure 7, using Tascell, the waiting time for child tasks to complete takes 16.73%, 20.84%, and 11.31% of the total execution time in Nqueen-array, Nqueen-compute and fib, respectively, using 8 threads. The busy time in Cilk and AdaptiveTC is about 99% of the total execution time. Thus Nqueen-compute in AdaptiveTC is 1.485x faster than in Tascell, even though the cost of managing d-e-ques and creating tasks in AdaptiveTC is almost the same as the cost of managing nested functions in Tascell.

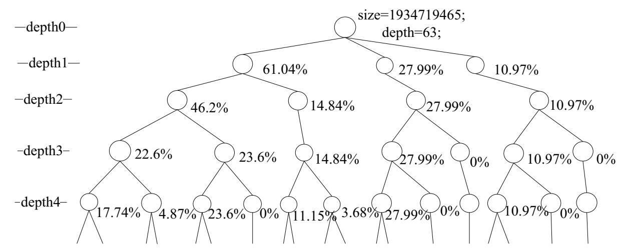


Figure 8. An unbalanced tree (input1)

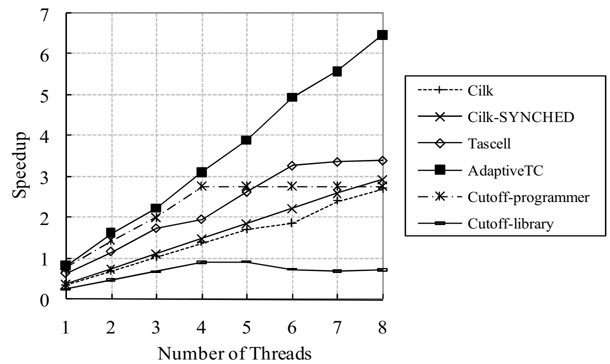


Figure 9. Speedup of Sodoku (input1)

5.3 The performance of unbalanced trees

5.3.1 The performance of AdaptiveTC and the cutoff strategy

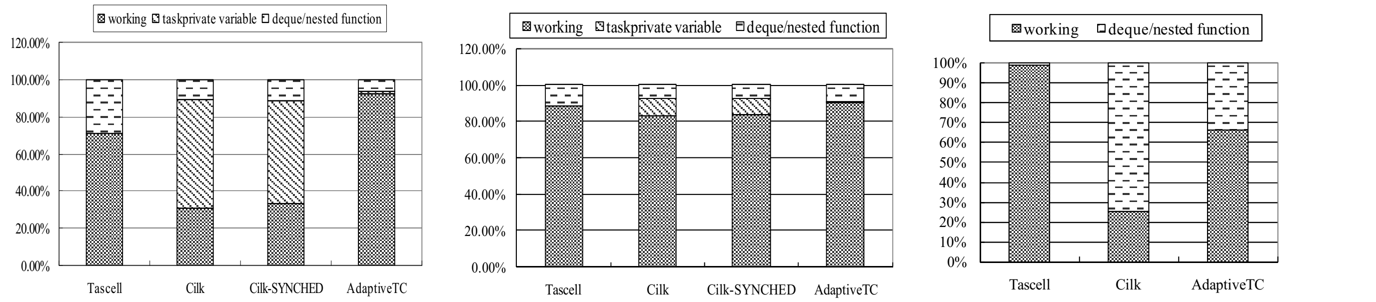
Figure 8 shows a part of an unbalanced tree. The tree has a total of 1,934,719,465 nodes, and a depth of 63. The percentage on each node shows the size of the sub-tree rooted on the node compared to the entire tree. This unbalanced tree is dynamically generated by one of the inputs to Sudoku.

We implemented two cutoff strategies. In one strategy (Cutoff-programmer), the cutoff is assigned by the programmer, and in the other (Cutoff-library) the cutoff is assigned by the runtime system. The *cut-off* is $\lceil \log N \rceil$ in AdaptiveTC. In both Cutoff-programmer and Cutoff-library, some threads are in starvation when the numbers of threads are larger than 4, as shown in Figure 9. In Cutoff-library, the cost of workspace copying cannot be reduced as mentioned before. In comparison, AdaptiveTC gets a better speedup in an unbalanced tree than the other two strategies.

5.3.2 The dynamic load balancing in Cilk, Tascell and AdaptiveTC

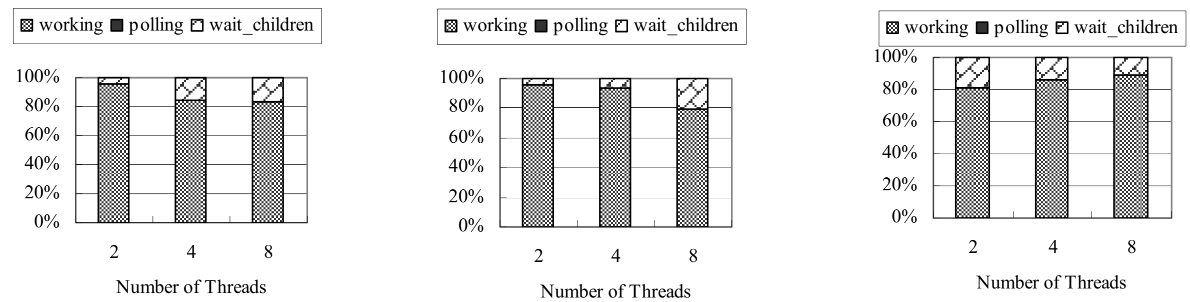
The three systems use different tradeoff strategies between dynamic load balancing and system implementation overhead to get a high performance. Cilk can suspend a waiting task (to avoid its waiting time) and execute other ready tasks because it keeps each task's information in the d-e-que. Tascell cannot suspend a waiting task and has to wait for all its child tasks to complete because Tascell uses the execution stack to keep the task information. AdaptiveTC can suspend a waiting task to execute other ready tasks, except the *special task* which it has to wait for all its child tasks to complete.

Figure 10 shows the speedups of 4 unbalanced trees. In Figure 10(a), it uses the tree shown in Figure 8 and its reversed tree. In Figures 10(b), 10(c) and 10(d), it uses three randomly generated unbalance trees and their reversed trees.



(a) The overhead of Nqueen-array(16) **(b)** The overhead of Nqueen-compute(16) **(c)** The overhead of Fib(45)

Figure 6. Breakdown of overheads with one thread. The overheads in AdaptiveTC are lower than Cilk for the three benchmarks and lower than Tascell for Nqueens, but higher than Tascell for fib.



(a) The overhead of Nqueen-array(16) **(b)** The overhead of Nqueen-compute(16) **(c)** The overhead of Fib(45)

Figure 7. Breakdown of overheads with multiple threads in Tascell.

Input	Size(the number of nodes)	Leaf nodes	Depth	The percent numbers shows size of the depth 1 sub-tree comparing with the entire tree. (%)
Tree1L	1961025791	1245356982	48	42.512, 25.362, 13.019, 4.936, 0.416, 11.771, 1.984
Tree1R	1961025791	1245356982	48	1.984, 11.771, 0.416, 4.936, 13.019, 25.362, 42.512
Tree2L	1961025791	1192225858	52	74.492, 20.791, 1.106, 2.732, 0.637, 0.049, 0.193
Tree2R	1961025791	1192225858	52	0.193, 0.049, 0.637, 2.732, 1.106, 20.791, 74.492
Tree3L	1961025791	1182058030	51	89.675, 6.891, 1.836, 0.819, 0.645, 0.026, 0.108
Tree3R	1961025791	1182058030	51	0.108, 0.026, 0.645, 0.819, 1.836, 6.891, 89.675

Table 3. Randomly generated unbalanced trees. The six trees have the same size, but different shapes. Tree*L is a left-heavy tree. Tree*R is reversed of Tree*L and is a right-heavy tree.

We use a random function of $x_i = (x_{i-1} * A + C) modeM$ to generate a fixed random sequence of numbers for a given x_0 (the initial seed). x_i is localized in each node and is used to get the size of each sub-tree. When the tree size and the initial seed are defined, the same unbalanced tree can be generated in multiple executions.

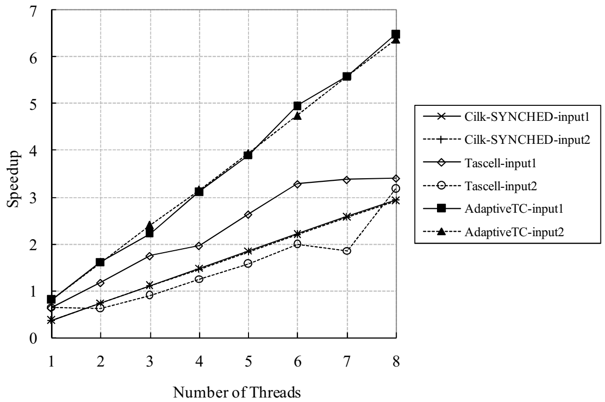
We set the execution time of each node to the average time of the task in the benchmarks shown in Figure 4. Table 3 presents the details of the unbalanced search trees. Tree3 is the most unbalanced one among these trees.

In the experimental results (Figures 10(b), 10(c), 10(d)), Cilk shows the best dynamic load balancing among the three systems because it gets almost the same speedup in all six trees. Figure 10 also shows Cilk achieves a slightly higher performance in right-heavy trees than in left-heavy trees.

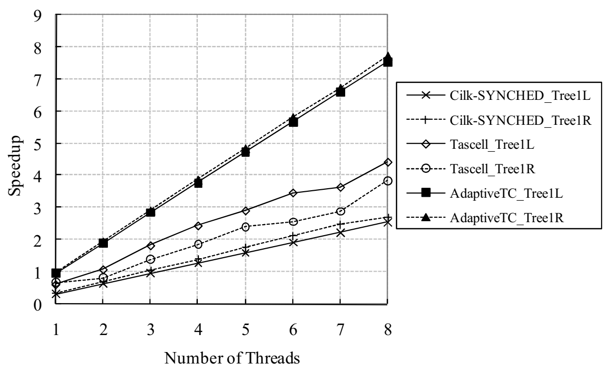
The performance of Tascell is impacted a lot by the shape of the tree. It gets worse performance on right-heavy trees than on left-heavy trees as the recursive call is a depth-first tree traversal. In Tascell, a *parallel-for* loop construct is implemented by spawning a half of the tasks for the requested threads. On a left-heavy tree, the first thread can run many tasks before waiting for its child tasks to complete. But on the right-heavy tree, the first thread could run fewer tasks before having to wait for its child tasks to complete. Therefore, it spends more time waiting on a right-heavy tree than on a left-heavy tree. For example, Tascell with 8 threads spends 8.08% of the total execution time in waiting on Tree3L, but almost 51.99% on Tree3R.

AdaptiveTC performs better dynamic load balancing than Tascell, but not always as good as Cilk. In Figures 10(a), 10(b) and 10(c), AdaptiveTC gets almost the same speedup on right-heavy and left-heavy trees. But in Figure 10(d), AdaptiveTC with 4, 7 and 8 threads, it gets worse load balancing on left-heavy tree than on right-heavy tree. AdaptiveTC with 4 threads in Tree3L spends 14.44% of total execution time waiting due to *steal fails* (i.e. it fails to steal a task) and 0.56% in waiting for child tasks to complete. About 2/3 of the *steal fails* are due to encountering an empty d-e-que. Hence, in AdaptiveTC with 4, 7 and 8 threads in Tree3L, the number of tasks generated is not sufficient to keep all threads busy, and this leads to the runtime starvation. But in Tree3R with 4 threads, AdaptiveTC spends 0.95% of total execution time waiting due to *steal fail* and 1.46% due to waiting for child tasks to complete.

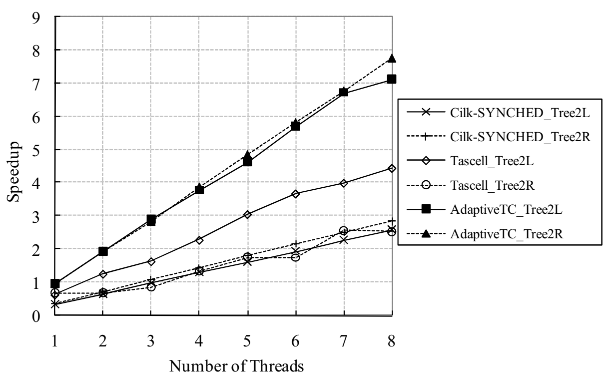
In the future, we will compare the number of steals in Cilk, the number of steals in AdaptiveTC and the number of respond-



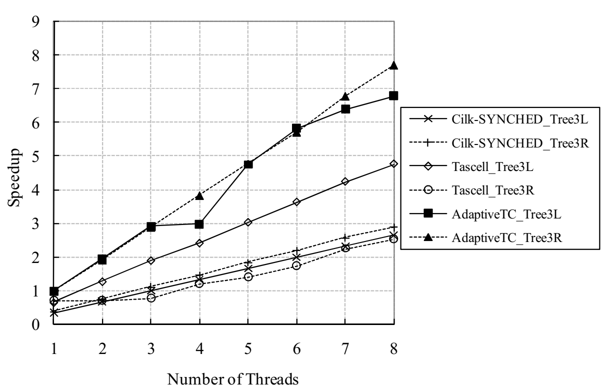
(a) Sodoku (input1/input2)



(b) Random unbalanced tree1L /tree1R



(c) Random unbalanced tree2L /tree2R



(d) Random unbalanced tree3L /tree3R

Figure 10. Speedup of unbalanced trees

ing requests in Tascell to analyze and evaluate the dynamic load balancing.

6. Conclusions

In this paper, we proposed an adaptive task creation strategy, called AdaptiveTC, to support work-stealing that could outperform Cilk and Tascell in several aspects. AdaptiveTC could adaptively create tasks to keep all threads busy most of the time, reduce the number of tasks created, and control the tasks granularity. It also introduced a new data attribute *taskprivate* for workspace variables that could reduce the workspace copying overhead in many important applications such as backtracking search, branch-and-bound search and game tree. As the result, it could reduce the overhead of managing the d-e-ques and creating tasks, the cost of workspace copying, and the chances of d-e-que overflow. Further, by using an adaptive task creation strategy, it improves load balancing on unbalanced call trees, and it is applicable to applications with or without definitive working set.

Acknowledgments

This research was supported in part by the National Basic Research Program of China (2005CB321602), the National Natural Science Foundation of China (60970024, 60633040), the National Science and Technology Major Project of China (2009ZX01036-001-002), the Innovation Research Group of NSFC (60921002), the U.S. National Science Foundation under the grant CNS-0834599 and a gift grant from Intel.

We would like to thank the reviewers for valuable comments and suggestions. We would like to thank Professor Zhiyuan Li at Purdue University for his many valuable suggestions and Tasuku Hiraishi for providing the Tascell system. Finally, we would like to thank Professor Vivek Sarkar for shepherding the paper.

References

[1] OpenMP Application Program Interface. Version 3.0, 2008.

[2] E. Ayguade, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. An Experimental Evaluation of the New OpenMP Tasking Model. In *the 20th International Workshop on Languages and Compilers for Parallel Computing*, pages 63–77, 2007.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5): 720C–748, 1999.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 207–216, 1995.

[5] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *the Twentieth Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 519–538, 2005.

[6] David Chase and Yossi Lev. Dynamic circular work-stealing d-e-que. In *the seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 21C–28, 2005.

[7] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsofi, Limor Fix, Nikos Hardavellas, Tod C. Mowry, and Chris Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *the nineteenth annual ACM symposium on Parallel algorithms and architectures, SPAA*, pages 105–115, 2007.

[8] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph

problems using adaptive work-stealing. In *the 2008 37th International Conference on Parallel Processing*, pages 536–545, 2008.

[9] Alejandro Duran, Julita Corbaln, and Eduard Ayguad. Evaluation of Openmp Task Scheduling Strategies. In *the 4th International Workshop on OpenMP, IWOMP*, pages 100–110, 2008.

[10] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 212C–223, 1998.

[11] Supercomputing Technologies Group. Cilk 5.4.6 Reference Manual. Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA.

[12] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler. In *the 24th IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2010. (To appear).

[13] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based Load Balancing. In *the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 55–64, 2009.

[14] Hans Wolfgang Loidl, Kevin Hammond, Hans Wolfgang, and Loidl Kevin Hammond. On the Granularity of Divide-and-Conquer Parallelism. In *Glasgow Workshop on Functional Programming*, 1995.

[15] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent Work Stealing. In *the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 45–54, 2009.

[16] Eric Mohr, David A. Kranz, and Jr Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2 (3):264C–280, 1991.

A. An AdaptiveTC program for Sudoku

```
typedef struct {
unsigned char board[9][9];            // the chess board
unsigned char placed_block[9][9];  // whether a piece is placed
unsigned char placed_row[9][9];
unsigned char placed_col[9][9];
} Status_t;

cilk int search(int next_row, int next_col, Status_t *st)
  taskprivate: (*st) (sizeof(Status_t));
{
  int sn = 0;                               // the number of solutions
                                            // find the first free row and col.
  if(!find_free_cell(next_row, next_col, &free_row, &free_col)){
    sn++;       return sn;              // a solution found
  }

  for(x = 1; x <= 9; x++){              // iterate through all numbers
    if(conflict(st, free_row, free_col, x)) // check whether conflict
      continue;
    set(st, free_row, free_col, x);          // set the board and placed arrays
    sn += spawn search( free_row, free_col+1, st);
    undo(st, free_row, free_col, x);  // undo the board and placed arrays
  }
  sync;
  return sn;
}
```

An AdaptiveTC program for Sudoku

B. A fast version of a Sudoku task in AdaptiveTC

```
typedef struct {
unsigned char board[9][9];            // the chess board
unsigned char placed_block[9][9];  //  whether a piece is placed
unsigned char placed_row[9][9];
unsigned char placed_col[9][9];
} Status_t;

int search(CilkWorkerState*const _cilk_ws,  int _adpTC_dp,
              int next_row,  int next_col, Status_t *st){
  search_info *f;                         //  task_infor pointer
  f = alloc(sizeof(*f));                  // allocate task_infor
  f->sig = search_sig;                    // initialize task_infor
                                          // find the first free row and col.
  if(!find_free_cell(next_row, next_col, &free_row, &free_col)){
    sn++;                                 // a solution found
    free(f);                              //  free task_info
    return sn;
  }
  f->sn = sn;
  for(x = 1; x <= 9; x++){               // iterate through all numbers
    if(conflict(st, free_row, free_col, x))   // check whether conflict
      continue;
    set(st, free_row, free_col, x);   // set the board and placed arrays
   if(_adpTC_dp < cut-off){
    tmp_st = Cilk_alloca(sizeof(Status_t));   // alloca a new space
    memcpy(tmp_st, st, sizeof(Status_t));     // copy parent status
    f->entry = 1;                         // save PC
    f->st = st;                           // save live vars
    f->mt = mt; f->depth = 0; f->x = x; f->sn = sn;
    f->free_row = free_row; f->free_col = free_col;
    *T = f;                                  // store task_infor pointer
    push();                                  // push task_infor into deque
    sn += search(_cilk_ws, _adpTC_dp+1, free_row,
                    free_col+1, tmp_st);
    if(pop(sn) == FAILURE)              //  check task_info
      return 0;                            //   child task stolen
   }else{
      sn += search_check(_cilk_ws, free_row, free_col+1, st);
   }
   undo(st, free_row, free_col, x);   // undo board and placed arrays
  }

  free(f);
  return sn;
}
```

A fast version of a Sudoku task in AdaptiveTC

C. A check version of a Sudoku task in AdaptiveTC

```
typedef struct {
unsigned char board[9][9];              // the chess board
unsigned char placed_block[9][9];  // pieces whether placed
unsigned char placed_row[9][9];
unsigned char placed_col[9][9];
} Status_t;

int search_check(CilkWorkerState*const _cilk_ws, int next_row,
                        int next_col, Status_t *st){
                                            // find the first free row and col.
   if(!find_free_cell(next_row, next_col, &free_row, &free_col)){
      sn++;                        // a solution found
      return sn;
   }

{    search_info *f = NULL;                         //  task_infor pointer
     int _adpTC_stolen = 0;
     int _adpTC_need_task = _cilk_ws->need_task;
   for(x = 1; x <= 9; x++){              // iterate through all numbers
      if(conflict(st, free_row, free_col, x))   // check whether conflict
         continue;
      set(st, free_row, free_col, x);   // set the board and placed arrays
      if(!_adpTC_need_task){
          sn += search_check(_cilk_ws, free_row, free_col+1, st);
      }else{
          if(!f){
             f = alloc(sizeof(*f));                  // allocate task_infor
             f->sig = search_sig;                   // initialize task_infor
             f->status = SPECIAL_TASK;
             f->sn = sn;
          }
          tmp_st = Cilk_alloca(sizeof(Status_t));      // alloca a new space
          memcpy(tmp_st, st, sizeof(Status_t));       // copy the parent status
          f->entry = 1;                          // save PC
          f->st = st;                            // save live vars
          f->depth = 0; f->x = x;
          f->free_row = free_row; f->free_col = free_col;
          *T = f;                                 // store task_infor pointer
          push();                                 // push task_infor into d-e-que
          sn += search_2(_cilk_ws, 0, free_row, free_col+1, tmp_st);
          if(pop_specialtask() == FAILURE)       // pop and check special task_info
              _adpTC_stolen = 1;                 // child task stolen
     }
     undo(st, free_row, free_col, x);      // undo the board and placed arrays
   }

   if(_adpTC_stolen){
      f->sn += sn;
      sync_specialtask();                     // wait children tasks
      sn = f->sn;                             // update the result
   }
   if(f)  free(f);
   return sn;
   }
}
```

A check version of a Sudoku task in AdaptiveTC