PAPER

Dynamic Multiple Work Stealing Strategy for Flexible Load Balancing

ADNAN[†a)], ***Nonmember and*** **Mitsuhisa SATO**[††], ***Member***

SUMMARY Lazy-task creation is an efficient method of overcoming the overhead of the grain-size problem in parallel computing. Work stealing is an effective load balancing strategy for parallel computing. In this paper, we present dynamic work stealing strategies in a lazy-task creation technique for efficient fine-grain task scheduling. The basic idea is to control load balancing granularity depending on the number of task parents in a stack. The dynamic-length strategy of work stealing uses run-time information, which is information on the load of the victim, to determine the number of tasks that a thief is allowed to steal. We compare it with the bottommost first work stealing strategy used in StackThread/MP, and the fixed-length strategy of work stealing, where a thief requests to steal a fixed number of tasks, as well as other multithreaded frameworks such as Cilk and OpenMP task implementations. The experiments show that the dynamic-length strategy of work stealing performs well in irregular workloads such as in UTS benchmarks, as well as in regular workloads such as Fibonacci, Strassen's matrix multiplication, FFT, and Sparse-LU factorization. The dynamic-length strategy works better than the fixed-length strategy because it is more flexible than the latter; this strategy can avoid load imbalance due to overstealing.

key words: *multiple steals, steal overhead, load imbalance, frame management, overhead reduction*

1. Introduction

For a scalable multithreaded computation on shared memory multicore processors, a computation requires a mechanism of efficient scheduling. To make the scheduling efficient, the overheads for scheduling a set of processors must be considerably less than the total amount of useful work. For instance, coarse-grain thread scheduling is efficient in many cases of regular parallelism. The other issue for efficient scheduling is load balancing. Even if, the total scheduling overhead is considerably less than the total work, load imbalance makes parallel computation unfavorable. The improvement by coarsening the task granularity may sometimes worsen load imbalance, that is, a set of processors cannot be divided into fully utilized processors resulting in underutilized processors.

Lazy-task creation [1] is an efficient method of overcoming the grain-size problem in parallel computing. By lazy-task creation, a parent task inlines its children, making the granularity of tasks coarse. While inlining its children, a victim makes parent continuations available for stealing by any other idle processors. In lazy-task creation, the cost of task creation is low, almost equal to that of a procedure call, which enables efficient parallel task capability by work stealing. Because of inlining tasks and stealing parent continuations, the granularity of tasks is coarsened without the loss of parallelism or load balancing.

The background of this research is a fine-grain-thread library, StackThreads/MP. This library adopts the lazy-task creation technique. In StackThreads/MP, idle workers steal tasks from the bottommost stack. We found that this strategy, which is a work stealing strategy of the original StackThreads/MP, results in a large steal overhead in the program of the Unbalanced Tree Search (UTS) [2], [3] benchmark. In such a case, the steal overhead increases rapidly when the number of processors is more than two. In Cilk, this problem does not occur because Cilk's workers steal work from the head of a queue efficiently by the THE [4] protocol.

To reduce the steal overhead, we propose a work stealing strategy that steals some threads from the bottom of a victim's logical stack. This strategy is to control the granularity of stolen threads by aggregating them. First, we have implemented a fixed-length strategy of work stealing in which a thief requests to steal a fixed number of tasks. We found that this strategy works more efficiently than the bottommost first strategy in the UTS benchmark with a long critical path. This strategy, however, does not perform better in some representatives of regular workloads such as Strassen's matrix multiplication, and even worse in Sparse-LU factorization, in which Sparse-LU is a non-recursive algorithm.

Another basic idea is to control the granularity for better load balancing depending on the number of task parents in the stack. The dynamic-length strategy of work stealing uses run-time information to determine the number of tasks that thieves can steal from the bottom of a logical stack. Because this strategy divides the logical stack fairly between the portion of the victim and the portion of the thief, it will work as the bottommost first strategy when the amount of sources in a logical stack is equal to or less than two tasks. According to experiments, we found that this work stealing performs well in irregular workloads such as in UTS benchmarks, as well as in the cases of regular workloads such as Strassen's matrix multiplication, FFT and Sparse-LU factorization.

In this paper, we present some experimental results of extended work stealing strategies for efficient fine-grain task

Manuscript received November 25, 2011.
Manuscript revised February 14, 2012.

[†] The author is with the Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba-shi, 305–8573 Japan.

[††] The author is with the Center for Computational Sciences, University of Tsukuba, Tsukuba-shi, 305–8577 Japan.

a) E-mail: adnantoo@hpcs.cs.tsukuba.ac.jp

DOI: 10.1587/transinf.E95.D.1565

scheduling. We report the experimental results of multiple work stealing strategies on both regular and irregular workloads.

Our contributions are as follows:

1. Although our proposed work stealing mechanism is rather simple, the experimental results show that it works reasonably well for both irregular workloads such as UTS and regular workloads such as Strassen's matrix multiplication, and FFT.
2. In our paper, we present a detailed experiment on and an analysis of our proposed work stealing mechanism. Although the proposed mechanism seems simple, to our best knowledge, no author has presented such detailed results yet.

The rest of this paper is organized as follows. In Sect. 2, we present some related research works. In Sect. 3, we discuss both the fixed-length and dynamic-length work stealing strategies. By using a model, we show that these strategies promote busy workers. We describe some experiments with some benchmark programs and present the results of evaluation in Sect. 4. We conclude this work in Sect. 5.

2. Related Works

Lazy-task creation [1] is an efficient method of overcoming the grain-size problem in parallel computing. By lazy-task creation, a child task is inlined by its parent, making the granularity of the task coarse. If the lazy-task creation technique is to increase the average run-time task granularity, proposed strategies should control load-balancing granularity.

We used the UTS [2], [3], [5] benchmark as the irregular and imbalance workloads in the experiments. Olivier and Prins developed the UTS benchmark using OpenMP [6] for shared memory computers and using UPC [7] for both shared memory and distributed memory computers.

In the UTS implementation by UPC, UPC partitions the memory-allocated stack into two regions. The first region is a region in which a thread pushes nodes created locally. The other region is a region in which a thread holds released nodes. These released nodes are available for work stealing [8].

UPC implemented work stealing so that stolen nodes are released nodes in the shared stack region. Because this region of a shared stack may be accessed concurrently by local and remote threads, the threads require locking so that we must introduce an additional overhead. Work aggregation [3], [5] and multiple work stealing strategies share the idea of paying off overheads. Work aggregation uses a task-chunking technique designed to increase granularity when creating tasks. The idea of work aggregation is similar to that of the fixed-length work stealing strategy. In both work-aggregation and fixed-length work stealing, we conducted manual tuning to determine the number of nodes/tasks stolen per steal operation. Moreover, work aggregation may lead to less or even no load balancing when the aggregated node is large, but the dynamic multiple work stealing strategy includes a work stealing mechanism for stealing half of the existing tasks from the victim's logical stack. This is for adjusting the number of stolen threads during the stealing time.

Cilk [4], [9] is a parallel language extension for the C language. It also works on the basis of lazy-task creation. Each worker in Cilk maintains a double-ended queue called a ready queue. When a parent task creates a child task, the parent inserts a continuation at the tail of its ready queue. Any other thread steals the continuation from the ready queue head using a randomized work stealing algorithm [10]. The work stealing scheduler of Cilk operates under the work-first principle that minimizes the work overhead by trading the work overhead with a longer critical path. This principle assumes that the average parallelism is larger than the number of processors available. The key optimization of Cilk is that it uses THE protocol to minimize the cost of mutual access to a queue; extended work stealing strategies minimize the steal overhead as an effect of multiple steals.

StackThreads/MP [11] is a fine-grain thread library that uses lazy-task creation similar to Cilk. Cilk uses the fast-clone and slow-clone compilation output approaches, and StackThreads/MP uses only a single compilation output and an efficient polling mechanism [12] to support work stealing. In the original strategy of StackThreads/MP, a victim selects the bottommost thread from a logical stack to be stolen by a thief. The proposed strategies increase the number of stolen tasks such that a victim attempts to select a number of chained tasks from the bottom of the logical stack for a single steal operation.

Tascell [13] is a parallel programming framework. It includes a multithreaded library and languages. The multithreaded framework of Tascell is similar to that of StackThreads/MP in that both of them use backtracking-based work stealing.

Although some authors have analyzed and suggested multiple steals for parallel computation, as we have observed recently, there are no real implementations and experimental results of multiple steals for lazy-task creation have been reported. For example, Berenbrink and Friedetzky [14] analyze the stability of natural work stealing, whereas Hendler and Shavit in [15] proposed a StealHalf work queue. In their paper, Hendler and Shavit also proved that StealHalf provides better load distribution. Although Hendler and Shavit presented stealHalf and an extended deque, they did not present real implementation and evaluation results. Hence, it still opens questions whether their proposal is realistic to be implemented and proven to improve performance in multithreaded computation. The dynamic multiple work stealing strategy differs from StealHalf in what they are working on. StealHalf works on an extended deque implemented with cyclic arrays.

Dinan et al. [16] presented scalable work stealing that implements a split task queue [17]. In a split task queue, Dinan et al. split a single-shared queue into local access only and shared access portions. By this split task queue, a thief steals a chunk of tasks from a shared queue. This work stealing mechanism is similar to the mechanism of work stealing in the UPC [8] above.

Dynamic multiple work stealing differs from StealHalf and the StealHalf in a split task queue in that we make use of a lazy-task queue in which we apply multiple work stealing.

3. Dynamic Multiple Work Stealing Strategy

In this section, we present a dynamic multiple work stealing strategy. The background and motivation of our research are StackThreads/MP. In the following subsection, we describe StackThreads/MP.

3.1 StackThreads/MP Fine-Grain Thread Library

StackThreads/MP is a thread library. It shares the same idea and intention as Pthreads in that both provide multithreaded programming [11]. Unlike Pthread that is suited to coarse-grain parallel programming, StackThreads/MP is intended for fine-grain parallel programming. As a thread library, StackThreads/MP provides a number of APIs to a StackThreads/MP run time system to create fine-grain threads and then multiplex these threads to a worker. Herein, we use *workers* to refer to OS threads/processes and *threads* to refer to threads created by StackThreads/MP [18].

In StackThreads/MP, workers are a bunch of OS threads or processes that have a shared memory address space. In StackThreads/MP, it is possible to arrange a number of workers into a small number of worker groups [19]. Such a group of workers enables work stealing between workers grouped together. In Fig. 1, we represent threads as circles, and workers as rounded rectangles.

Creating a new thread in StackThreads/MP is similar to calling a function. In addition, the workers of StackThreads/MP record information for created threads into a table. One important aspect of creating threads being similar to making a function call is that it is possible to create threads recursively in the same way that a C program may call a function recursively. When a worker creates threads recursively, we will find frames of threads chained within the stack, as shown in Fig. 2 (a).

Whenever a worker loses threads by having them stolen, its related stack frames are not removed from their original location. Figure 2 (b) shows that when a thread is created, a new stack frame is pushed on top of the stack, and when a thread is stolen, its stack frame remains in its original position; hence, the thief's Frame-Pointer should point to the original position. We call this original work stealing approach of StackThreads/MP as the bottommost first work stealing strategy. From the experimental results, we use st_org to refer to the bottommost first strategy.

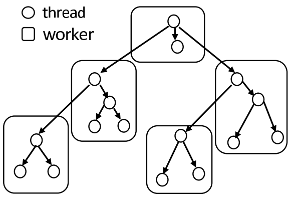


Fig. 1 Worker-thread relationship.

3.2 Extended Work Stealing Strategies

Now, we discuss extended work stealing strategies. In the fixed-length work stealing strategy, a thief attempts to steal a fixed number d of chained tasks from the bottom of a victim's logical stack. If the number of a victim's tasks is equal to or more than the specified number, the thief steals a fixed number of the stacked tasks at the same time. In the dynamic-length work stealing strategy, the number of threads that a thief steals is half the number of existing threads from a victim.

In the implementation, we made a simple modification to StackThreads/MP. The calling of the function **st_suspend()** is twice. This is similar to that of the original StackThreads/MP. But the modification is simply passing *n_total_threads* – d in the first call, and passing d in the second call. However, we should pay careful attention when computing d for a dynamic length. Computing the integer d, which might be rounded off, may not result in the victim incorrectly restarting half-top stack frames. The total number of threads at both the victim and the thief restart must be as large as that when the victim accepts a steal request.

Algorithm 1: A thief attempts to steal a bunch of tasks from any victim. It is performed in an infinite loop until the thief detects termination. First, the thief attempts to lock the steal port (*lock_steal_port*) of a randomly selected victim (*select_random_victim()*). If successful, the thief sends a steal request (*send_steal_request()*) and wait for a reply from the victim. The steal request, which is sent, contains an empty pointer to a *context list* r, which the victim must reply to after filling r with the context stolen thread. Whatever the reply from the victim is either a nonempty pointer of execution context or I_HAVE_NO_TASK, the thief unlocks the steal request port (*unlock_stealing_port*). If r is a nonempty pointer, the thief restarts r. Idle workers will sleep (*usleep()*) for a duration of $100\,\mu s$ after failing to obtain work from the victim.

Algorithm 2: The thread local storage *n_total_threads* denotes the number of threads in a victim's logical stack. Victims use it as run-time information. All victims compute

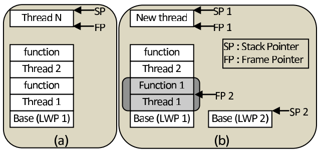


Fig. 2 (a) Chained threads in stack frames. (b) A new frame is pushed when a thread is created, and frames remain at the original position when stolen.

Algorithm 1 Extended work stealing strategy on thief side

```
COUNTER = 0
loop
    COUNTER++
    select_random_victim()
    lock_stealing_port()
    send_steal_request(r)
    if ($r \neq I\_HAVE\_NO\_TASKS$) then
        unlock_stealing_port()
        restart(r)
    else
        unlock_stealing_port()
    end if
    if ($COUNTER \geq LIMIT$) then
        usleep(sleepingduration)
        RESET COUNTER
    end if
    if (detected_termination) then
        exit()
    end if
end loop
```

Algorithm 2 Extended work stealing strategy on victim side

```
r = steal_received()
if (dynamic) then
    d = n_total_threads/2
    if ($d < 1$) then
        reply(I_HAVE_NO_TASK)
        return
    end if
else if (static) then
    d = constant
    if (n_total_threads -1 < d) then
        reply(I_HAVE_NO_TASK)
        return
    end if
end if
suspend(c,n_total_threads - d)
suspend(r,d)
reply(r)
restart(c)
```

the number of stolen threads, that a thief must steal, in line 3 for a dynamic length work stealing strategy, and in line 9 for a fixed-length work stealing strategy. For the dynamic-length work stealing strategy, $d = \lfloor n\_total\_threads/2 \rfloor$, whereas for the fixed-length work stealing strategy $d = constant$. As shown in algorithm 2 and Fig. 3, the extended work stealing strategies are as follows. A victim uses a polling mechanism to listen for a steal request. When a victim LWP1 receives a steal request from an idle worker in the first line of algorithm 2, the victim computes d and select d tasks from the bottom of its logical stack. If there are more than d tasks in the logical stack, then, in line 16 of algorithm 2, the victim LWP 1 suspends (*suspend()*) $n\_total\_threads - d$ tasks to the bottom of its logical stack; otherwise, the victim sends the thief an *I_HAVE_NO_TASK* message. As the result, the data structure c of *context_t* lists the execution context of all $n\_total\_threads - d$ suspended tasks, as shown in Fig. 3 (a). Next, in line 17 of algorithm 2, LWP 1 suspends d tasks. The 2^{nd} suspension lists the contexts of d suspended tasks in r, as shown in Fig. 3 (b). LWP 1 gives the execution context listed in r to the thief, i.e. LWP 2. Finally, LWP 1 restarts c and LWP 2 restarts r to continue the execution, as shown in Figs. 3 (c) and 3 (d).

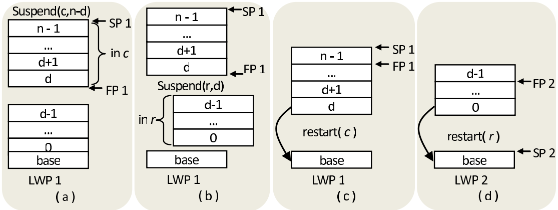


Fig. 3 Extended work stealing strategies. Assume that the worker has received a steal request. (a) The threads above the target threads are suspended and detached. (b) The target threads are suspended (c) After target threads have been stolen, the victim restarts c and (d) thief restarts r.

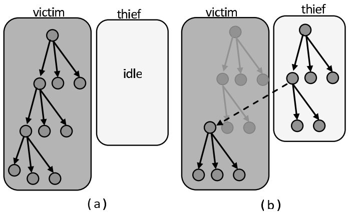


Fig. 4 Extended work stealing strategy in task graph (a) a thief sends a steal request to victim (b) thief has more tasks to execute.

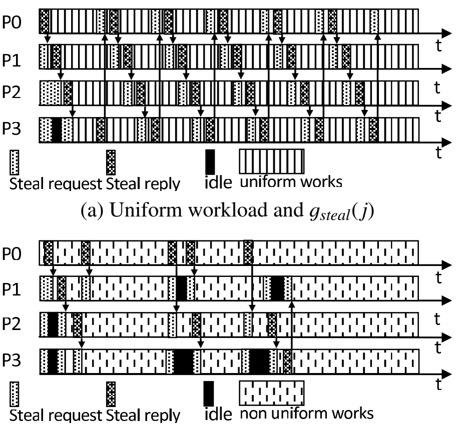
Using the extended work stealing strategies, a thief steals more tasks from a victim. In Fig. 4 (a), an idle thief sends a steal request. After work stealing is performed as shown in Fig. 4 (b), the thief has more tasks to execute. By this strategy, the time spent for work can be expected to be considerably larger than the time spent for stealing. In addition, we provide a differential equation model [20] to explain that our strategies promote busy workers in Sect. 3.4.

3.3 Controlling Load Balance Granularity

Controlling load-balancing granularity is important in that a small load-balancing granularity incurs a large steal overhead. Faxen introduced the load-balancing concept in his paper [21]. However, we redefine load-balancing granularity in Eq. (1). The load-balancing granularity $g_{steal}(j)$ defines the execution time of a bunch of tasks from j^{th} steal operation. We use Eq. (1) to emphasize that $g_{steal}(j)$ may vary. In Eq. (1), T_S is the total execution time without a steal overhead and an idle time. In Figs. 5 (a) and 5 (b), load balancing granularity can be measured as the distance between two steal requests. It is intuitively understood, that the number of steal operations, N_{steal}, is inversely proportional to load-balancing granularity. Therefore, the smaller the $g_{steal}(j)$, the larger the steal overhead.

$$T_S = \sum_{j=1}^{N_{steal}} g_{steal}(j) \tag{1}$$

The task granularity $g_{task}(i)$ in Eq. (2) has a relation to



(a) Uniform workload and $g_{steal}(j)$

(b) Non uniform workload and varying $g_{steal}(j)$

Fig. 5 Illustration of execution profile with work stealing.

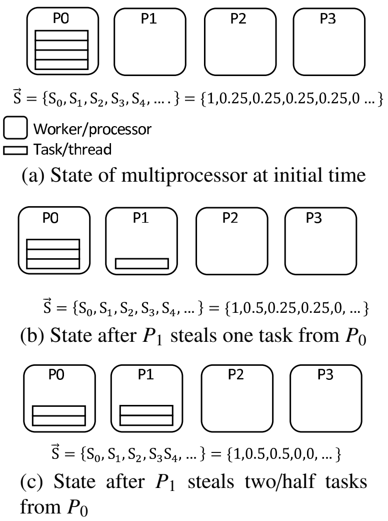
load-balancing granularity. We define task granularity as the execution time of a single task. We use Eq. (2) to emphasize that the workload can be distributed nonuniformly. The tasks shown in Fig. 5 (a), have a uniform workload. The tasks shown in Fig. 5 (b), have a nonuniform workload and a nonuniform $g_{steal}(j)$. In Figs. 5 (a) and 5 (b), idle occurs after a thief fails in several steal attempts. If the distribution of $g_{task}(i)$ were uniform, then it would be easy to estimate the ratio N_{task}/N_{steal} such that $g_{steal}(j) = N_{task}/N_{steal} \times g_{task}(i)$ would be ideal. However, $g_{task}(j)$ value can be markedly different from each other. Therefore, manual tuning to find the optimal N_{task}/N_{steal} is effective. Otherwise, the ratio should be decided dynamically at runtime for the ideal $g_{steal}(j)$. Our proposed dynamic-length work stealing is one of the solutions to this problem.

$$T_S = \sum_{i=1}^{N_{task}} g_{task}(i) \tag{2}$$

3.4 Analytical Model of Extended Work Stealing Strategies

In this subsection, we describe an analytical model of work stealing. We use the model presented by Mitzenmacher [20]. We use this analytical model to show that multiple steals increase the number of busy processors. Moreover, a dynamic multiple-steal strategy not only increases the number of busy processors but also doubles the number of prospective victims without increasing the steal threshold. Using the **steal threshold**, we specify that a victim is required to have i stacked threads such that a thief is allowed to steal $i - 1$ threads.

In his paper [20], Mitzenmacher defined $n_i(t)$ to be the number of processors with exactly i tasks at a time t. He also defined $p_i(t) = n_i(t)/n$ to be the fraction of processors with i tasks where n is the total number of processors, as well as the nonincreasing series $s_i(t) = \sum_{k=i}^{\infty} p_k(t)$ as a *state of processors*. This *state of processors* can be interpreted as $m_i(t)/n$; which is the fraction of processors that have at least i tasks at a dicrete time, where m_i is the number of processors with at least i tasks.


P0
P1
P2
P3
$\vec{S} = \{S_0, S_1, S_2, S_3, S_4, \ldots.\} = \{1, 0.25, 0.25, 0.25, 0.25, 0 \ldots\}$
Worker/processor
Task/thread
$\vec{S} = \{S_0, S_1, S_2, S_3, S_4, \ldots\} = \{1, 0.5, 0.25, 0.25, 0, \ldots\}$
$\vec{S} = \{S_0, S_1, S_2, S_3 S_4, \ldots\} = \{1, 0.5, 0.5, 0, 0, \ldots\}$

(a) State of multiprocessor at initial time

(b) State after P_1 steals one task from P_0

(c) State after P_1 steals two/half tasks from P_0

Fig. 6 Multiprocessor state. At the initial time, only P_0 has works, i.e., four works in its stack; other processors are empty. After P_1 steals one work from P_0, the fraction of busy processors S_1 increases; however S_2 remains 0.25. It does not add potential victims. By multiple steals, after P_1 steals two works from P_0, S_1 and S_2 increase up to 0.5; hence, this increases the probability of stealing from S_2 if the steal threshold remains at $i = 2$.

We now consider the first fraction, i.e. s_0. The fraction of s_0 is a fraction of those processors with at least 0 tasks, which means all processors. This value must be 1. Busy processors are all those included in s_1. However, only those processors in fraction s_2 can be prospective victims. In fact, fraction s_2 is the larger fraction but with the smallest steal-threshold. Now, we have the vector state $\overrightarrow{S} = \{s_0, s_1, s_2, \ldots\}$ that denotes the state of a multiprocessor system in a discrete time.

Figure 6 (a) shows an example of multiprocessors consisting of four processors. At the initial time, the system has only one processor with at least one task; hence, $s_1 = 0.25$. After a thief steals one thread/task from a victim, only s_1 increases. Unless workers create a lazy task, it would not increase s_2, as shown in Fig. 6 (b).

Let us consider the worst case of work stealing. The worst case is that after a worker steals a small-granularity thread, the worker does not create a lazy task. After a worker steals a small work and arrives at s_1, the worker does not go to s_2. Soon the worker becomes empty and goes back to performing repeatedly load balancing. This eventually results in excessive steals. In Cilk, whose workers steal works efficiently by THE protocol, efficiency is sometimes lost in critical path only. However, in StackThreads/MP whose victims are involved in unwinding stack frames, excessive steals increases steal overhead.

Fixed-length work stealing can be considered as multiple-stealing with a threshold of i tasks [20]. Here, although the fixed-length strategy promotes processors with a higher fraction $(s_2, s_3, \ldots)$, it does not increase the number of prospective victims because the steal threshold also increases. On the other hand, the workloads of threads can be extremely different from each other. In such a case, it is difficult to predict the appropriate number of stolen threads. A fixed-length strategy may sometime result in either oversteal that eventually increases the steals, or failure to steal that eventually increases the idle. Therefore, manual tuning is still effective to determine the optimal number of stolen threads.

In contrast to that in the dynamic-length work stealing strategy, the number of tasks that a thief steals from the fraction s_d is determined dynamically as the bottom-half of the number of active tasks in the victim's logical stack. In this case, the probability for a successful steal remains s_2, which can be larger than s_d. Because multiple steals are performed, by this strategy, s_i increases for $1 \leq i \leq d$, as shown in Fig. 6 (c). Hence, this strategy promotes busy workers, i.e. this increases the number of processors that busy as we show evidences in the performance evaluation result section. R_{busy} increases and at the same time R_{steal} decreases. Such extended work stealing strategies reduce the overhead in delivering the steal request significantly; it also reduces idle with the lack of polling [12], which is in contrast to the original work stealing strategy of StackThreads/MP.

Finally, there is no single strategy that suits all problems. In applications in which the tree is balanced such as the Fibonacci number, the dynamic length strategy has comparable performance to the bottommost strategy. Because of oversteal, the dynamic-length strategy increases a small steal, whereas stealing the bottommost thread is the most efficient strategy for increasing task granularity. Although the dynamic-length strategy may oversteal threads, the threads can be restolen. As long as all workers remain busy most of the execution time, steal and idle are reduced. This eventually will result in acceptable performance.

4. Experiment

In this section, we describe some experiments. As benchmarks in these evaluations, a subset of benchmarks from the Barcelona OpenMP Tasks suite [22] were adopted. Two binomial trees of the UTS were selected as the representatives of an irregular workload. We selected Strassen, Sparse-LU, and FFT as representatives of regular workloads. We added Strassen, Sparse-LU, and FFT just to see which strategy performs well or poorly on certain applications. We also evaluate the Fibonacci number. The evaluation for this benchmark is intented to evaluate overheads in depending on number of stolen threads.

In our evaluations, we used GCC 4.4.3, Intel C Compiler 11.1 and GCC 2.8.1. We used GCC 4.4.3 as a complete compiler for GCC OpenMP [6]. GCC 4.4.3 was used as the back end for Cilk. To compile all benchmarks with the StackThreads/MP, GCC 2.8.1 is used. We compiled all benchmarks with a -O3 compiler switch.

4.1 Experimental Configuration

We conducted some experiments on a machine with two 6168 AMD Opteron CPUs. Each CPU has 12 cores, 12 × 512 KB L2 Cache and 6 M shared L3 Cache. The machine is installed with Linux CentOS 5.3 as its operating system. The machine is configured with 12 GB RAM.

4.1.1 UTS Benchmark

The UTS problem is a problem of counting the number of nodes explored in an implicit tree. The use of execution threads of UTS makes tree exploration in a depth-first search manner. Since UTS generates an unbalanced tree, as benchmark, UTS makes a convincing model for analyzing load imbalance and irregular parallelism that exist in many applications.

UTS constructs a tree by generating nodes in a parallel and recursive way. The UTS benchmark applies the SHA-1 computation to a 20-byte descriptor of the parent node to obtain a new 20-byte descriptor for each generated child. UTS uses this 20-byte descriptor to calculate the probability function of nonleaf nodes that have m children. Hence, to construct a tree at some depth levels, UTS processes nodes recursively until no more nodes can be generated. In the UTS, each node corresponds to a task. Hence, nodes without children are fine-grain tasks, whereas nodes with children are non fine-grain tasks. As the result, load imbalance affected the distributions of these two types of node.

Load imbalance in the UTS benchmark depends on parameters m and q. In a binomial tree, these parameters specify that a node in an unbalanced tree has m children with a probability q. The parameters shown in Table 1 were used in our experiments.

The tree of UTS is much different from the tree of Fibonacci. In UTS, the granularity of nodes is not equally distributed in that some of the nodes are fine-grained and the remaining nodes are coarse-grained. The distribution of Fibonacci task granularity is a uniform distribution in that its individual nodes have a small granularity. When one executes a Fibonacci task using a single processor, one will find that a single processor overhead is large. It will be different when we execute UTS using a single processor. Then the average task granularity of the execution will not be too small. As the result, the single processor overhead in UTS will not be as large as in the Fibonacci. Hence, the issue of UTS is not the small-task granularity but the small load-balancing granularity. This small load-balancing granularity is due to a small subtree located at the bottom of a tree or a subtree.

Olivier performed some implementations of UTS. The implementations include OpenMP [5] and UPC [2], [7] implementations. We compare the performance of the OpenMP implementation of UTS to that of the lazy-task

Table 1 Parameters of binomial tree, root branching factors b_0, and parameters of a node that probaly has m children with a probability q.

Type	b_0	m	q	Tree size (in nodes)
A	2000	8	0.124875	4112897
B	2000	3	0.333333	30399116

creation implementation using StackThreads/MP and Cilk. In StackThreads/MP and Cilk implementations, we implemented UTS such that a thief steals nodes in logical stacks. For their C elision code, we replace each thread creation/task spawn with a procedure call. The difference between Cilk and StackThreads/MP is that we implemented inlet for Cilk to accumulate subtree sizes, whereas we use loop-for for StackThreads/MP to accumulate subtree sizes after the synchronization point.

4.1.2 Fibonacci Benchmark

The Fibonacci benchmark is a benchmark for computing the Fibonacci number. One algorithm for computing this is the recursive algorithm. A node that computes the Fibonacci n creates two children. The first node computes Fibonacci $n-1$ and the last node computes Fibonacci $n-2$. Each node in this benchmark has a small workload, and a large overhead. Therefore, it is difficult to achieve good scalability in this benchmark without cutting-off the task.

4.1.3 Strassen Matrix Multiplication

Strassen's matrix multiplication computes a matrix dot product $[C] = [A] \cdot [B]$ using Strassen's algorithm. Strassen's algorithm divides the matrices A, B, and C into four quadrants and then performs seven matrix dot products and four additions to submatrices. This algorithm can be recursively applied to four submatrices so that we can have recursive-parallel-task generation. In these experiments, the matrix sizes are 1024×1024, 2048×2048, and 4096×4096 of double precision. It is clear that the larger the problem size, the higher the degree of parallelism.

4.1.4 Sparse-LU Factorization

Sparse-LU computes sparse matrix LU factorization. Even though this is a matrix computation in which the parallelism is regular and uses a nonrecursive algorithm, because of sparseness in the matrix, the workloads among the processors are unbalanced. Because Sparse-LU lacks parallelism in a stack, and is also a coarse-grained regular loop parallelism, then we do not expect better performance with control of load-balancing granularity. The matrix size used in this experiment is the size of 50 blocks of 100 floats of submatrix.

4.1.5 Fast Fourier Transform

The fast Fourier transform algorithm computes one-dimensional discrete Fourier transform using the Cooley-Tukey algorithm recursively. At the first stage, the FFT algorithm precomputes the matrix coefficient W in a divide-and-conquer manner. After obtaining the matrix coefficient W, FFT computes the factors r of length n. At the final stage, recursively, the FFT divides DFT into r smaller DFTs of length n/r and multiply them by twiddle factors. FFT applies this algorithm to a vector of the complex data type. In these experiments, the vector sizes are 32 M and 64 M of the complex data type.

4.2 Methodology

We evaluated the five benchmarks discussed above. In the evaluation, we compared the fixed-length and dynamic-length work stealing strategies (st_sta and st_dyn) to the original bottommost first work stealing strategy (st_org) of StackThreads/MP. For the fixed-length strategy of work stealing for StackThreads/MP, we conducted manual tuning to determine the number of task tasks a thief steals. In the UTS benchmark, we found that 20 stolen threads are optimal, whereas in other benchmarks, we stopped at two threads since increasing this number will result in load imbalance, thereby giving a negative impact.

4.3 Performance Evaluation Result

4.3.1 Serial Program Execution Time and Work Overhead

We measured the sequential execution time as a baseline for scalability measurement; for the throughput measurement, we used the single-threaded execution time as its baseline. For the UTS benchmark, we made performance comparison between Cilk, the Intel OpenMP task, the GCC OpenMP task, and StackThreads/MP, whereas for the other benchmarks, we performed a performance comparison between StackThreads/MP and the Intel OpenMP task.

In these evaluations, we used the equivalent OpenMP task using StackThreads/MP. Figure 7 shows the sequential execution times and single processor overheads for different implementations and strategies. The single processor overhead of the Intel OpenMP task appears to be the highest. Nevertheless, note that the Intel OpenMP task has the shortest serial execution time. The GCC OpenMP task shows a low single processor overhead in that GCC 4.4.3 uses a technique limiting the number of queued tasks. StackThreads/MP with all strategies and Cilk showed a low single processor overhead. Note that, this uniprocessor overhead depends on the application. For example, the uniprocessor overhead of a Fibonacci is high, as shown in Fig. 8. In a parallel Fibonacci, the overhead is high in that the overhead of a task spawn (Cilk) and thread creation (StackThreads/MP) is much larger than the computation in individual nodes of a

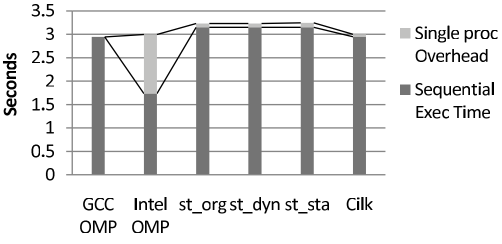


Fig. 7 Sequential execution times and single processor overheads in the UTS benchmark of different strategies and implementations.

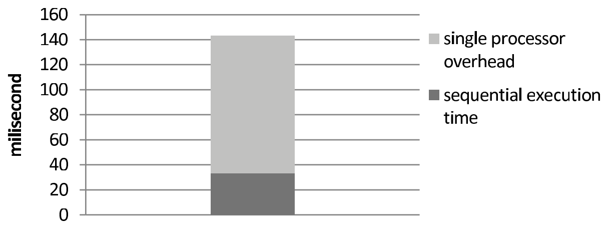


Fig. 8 Single processor overheads in Fibonacci number (n = 30). This overhead is larger than the computation core of a node in which the Fibonacci procedure involves only an addition operation.

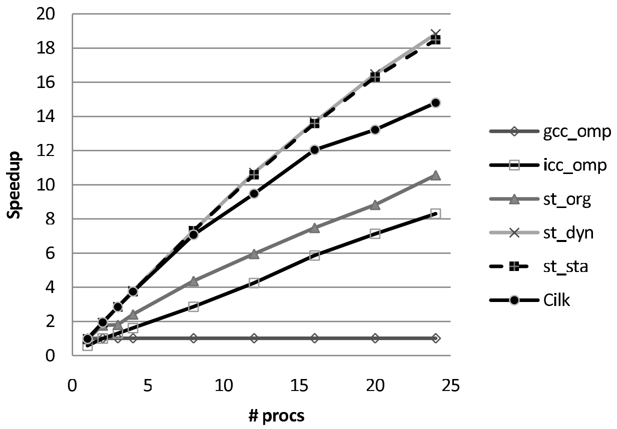


Fig. 9 Speedups of UTS benchmark of different strategies and implementations. In this evaluation, m = 8 q = 0.124875 is used.

Table 2 Execution profile of UTS type A for the bottommost first strategy, fixed-length, and dynamic-length work stealing strategies.

Strategy	Busy	Steals	Idle	Switch	frm_mgt
st_org	6758903	3623421	264845	497716	1445588
st_sta	4619911	403476	182164	339500	416425
st_dyn	4713164	379168	171128	437727	227515

Fibonacci.

4.3.2 Overheads and Scalability

We discuss the scalability comparison first. Figure 9 shows the speedups on UTS and the different types of work stealing strategy. Unless in the case of the GCC OpenMP task, only the bottommost first strategy in StackThreads/MP showed poor performance. Cilk, in which workers steal works efficiently from the head of a deque (bottommost task) using *THE* protocol, did not experience the problem as that in StackThreads/MP.

A high steal overhead had the original StackThreads/MP performed poorly. Not only is the steal overhead large, the overhead for managing stack frames also affects the performance of the original StackThreads/MP. We fixed the problem so that the overhead becomes low.

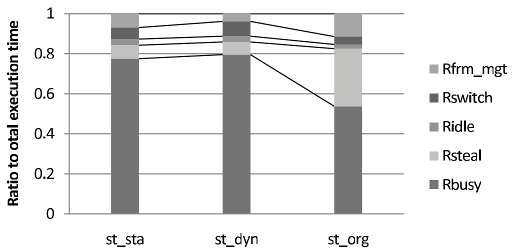


Fig. 10 Breakdown of total execution time for UTS in two AMD Opteron 6168 CPUs. All components are normalized the total running time of 24 workers.

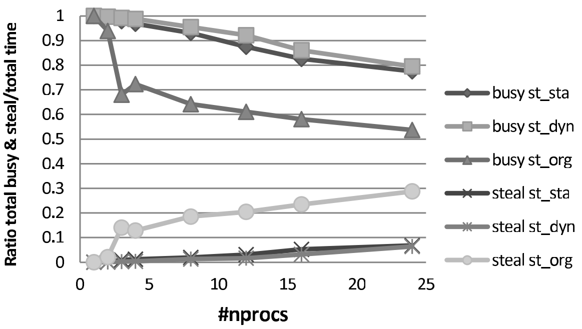


Fig. 11 Steal overhead trend line curves. As the number of workers is increased the steal overhead increases. The proposed strategies decrease this overhead significantly.

We conducted performance profiling and broke down the sum of the total execution time of 24 core processors, as shown in Table 2. We found that the steal overhead dominated the execution time in the original StackThreads/MP. Equations (3)–(7) show formulations of R_{busy}, R_{idle}, R_{frame_mgt}, R_{switch}, *and* R_{steal} respectively. In some equations, we used the symbol C to indicate that the time spent is an overhead, i.e., $C^{(i)}_{steal}$ to denote the steal overhead by the processor i.

$$T_{total} = \Sigma(T^{(i)}_{busy} + C^{(i)}_{steal} + T^{(i)}_{idle} + C^{(i)}_{frame\_mgt} + C^{(i)}_{switch})$$

$$R_{busy} = \Sigma T^{(i)}_{busy} / T_{total} \tag{3}$$

$$R_{idle} = \Sigma T^{(i)}_{idle} / T_{total} \tag{4}$$

$$R_{frame\_mgt} = \Sigma C^{(i)}_{frame\_mgt} / T_{total} \tag{5}$$

$$R_{switch} = \Sigma C^{(i)}_{switch} / T_{total} \tag{6}$$

$$R_{steal} = \Sigma C^{(i)}_{steal} / T_{total} \tag{7}$$

We clarify now that the cause of unsatisfactory performance by StackThreads/MP in UTS is the overheads being much larger than the granularity of a small subtree. In Eq. (7), $\Sigma C^{(i)}_{steal}$ increases linearly with increasing number of processors. Therefore, in Fig. 10, the steal overhead for st_org is 28.7%. Figure 11 shows the trend of the steal overhead. We observed that the steal overhead increased markedly as the number of workers increased beyond 2. It got worse because the probability q of a node that has m children is low, inducing the binomial tree to generate many small subtrees. Stealing a small subtree makes workers go away from s_1 and to become empty. This will reduce R_{busy} and increase R_{steal} in Eqs. (3) and (7). In addition, the overhead in managing stack frames was somewhat high. In st_org in Fig. 10, this overhead was 11.48%. This is because the space that the worker used for many stack frames was considerably large, in that UTS is a deep-recursive-task generation.

Extended work stealing strategies improved performance in which the busyness to work, steal overhead, and stack frames management overhead are improved significantly. In Fig. 10, the steal overhead dropped to 6.7% and 6.3% in st_sta and st_dyn, respectively; the frame management overhead dropped to 6.9% and 3.9% in st_sta and st_dyn, respectively. Overheads decreased because the victim did not unwind or manage long frames since the victim transferred them to the thieves.

Recall that the extended work stealing strategies increase s_i for $1 \leq i \leq d$. An increase in s_i means that the fraction of busy workers, s_1, increases, in contrast, the $1 - s_1$ decreases. As the result, $R_{busy} = \Sigma T^{(i)}_{busy}/T_{total}$ increases and $R_{steal} = \Sigma C^{(i)}_{steal}/T_{total}$ decreases following the trajectory of $1 - s_1$.

Figure 12 shows the speedup for the second binomial tree (type B) in Table 1. The tree is deeper than the first tree (type A) so that we can see its result in Fig. 9. However, the binomial tree B has more tasks generated than tree A. Comparing Figs. 9 and 12, the dynamic-length work stealing strategy shows an improvement in tree B, whereas the bottommost first work stealing strategy exhibits a degradation.

We report that StackThreads/MP with extended work stealing outperforms Cilk in UTS because Cilk experiences a large critical path overhead. Figures 13 and 14 respectively show the throughput for processing binomial tree A and binomial tree B of UTS by different scheduling schemes and implementations. Extended work stealing strategies show the best result in that they give good scalability. In this evaluation, throughput is achieved using 24 cores from two AMD Opteron 6168 CPUs. In those figures, we can see that the extended work stealing strategies have outperformed Cilk. Figure 9 shows that Cilk experienced deceleration as the number of workers increases from 8 to 24. In Fig. 12, in which UTS type B has more parallelism than UTS type A, Cilk exhibits better scalability. This is not surprising since the tree lacks parallelism. In a case that lacks parallelism, Cilk spends most of its time on the slow clones which contributes to the high critical path overhead. It is likely that the dynamic-length work stealing strategy will perform better as the number of processor cores increases.

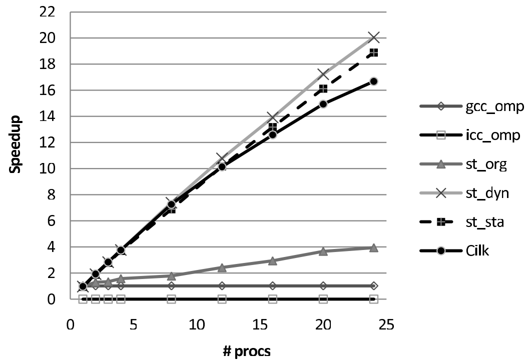


Fig. 12 Speedup of UTS benchmark for different strategies and implementations. In this evaluation, $m = 3$ $q = 0.333333$.

We present the overhead and performance evaluation on the Fibonacci number benchmark. To obtain a long execution time, the input parameter n is 40. In this evaluation, the length of stolen threads was varied. As shown in Fig. 15, busy, steal and idle were measured in accordance with the length of stolen threads. In the figure, steal and idle increase, and busy decreases so that busy was lower than steal and idle. When the stolen threads are 16, busy, steal

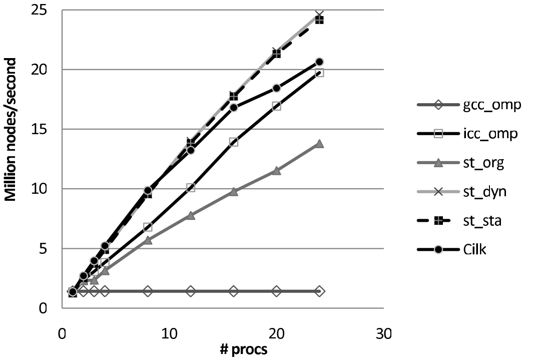


Fig. 13 Throughput of different strategies and implementations on UTS. In this result, binomial tree A is used, which generates more than 4 million nodes.

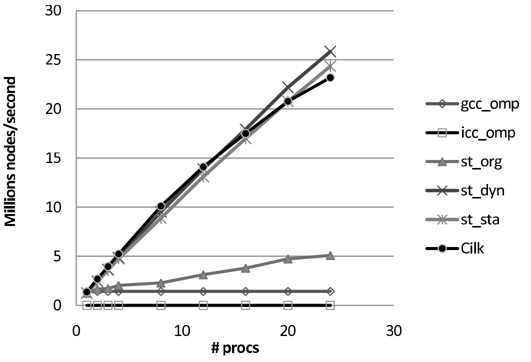


Fig. 14 Throughput of different strategies and implementations on UTS. In this result, binomial tree B is used, which generates more nodes.

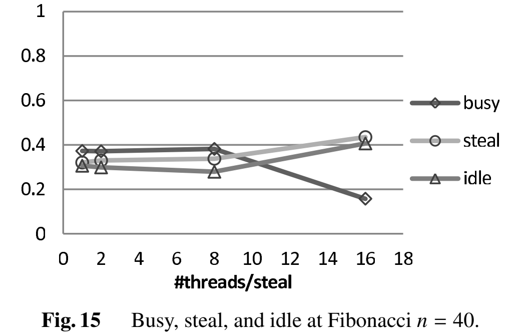


Fig. 15 Busy, steal, and idle at Fibonacci $n = 40$.

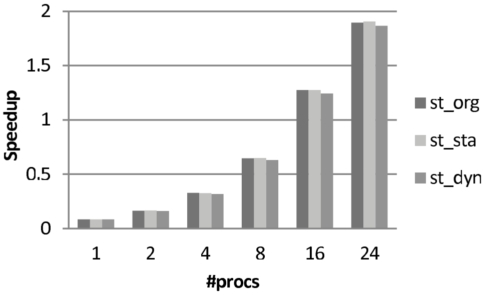


Fig. 16 Speedup at Fibonacci n = 40, in st_sta, d = 2.

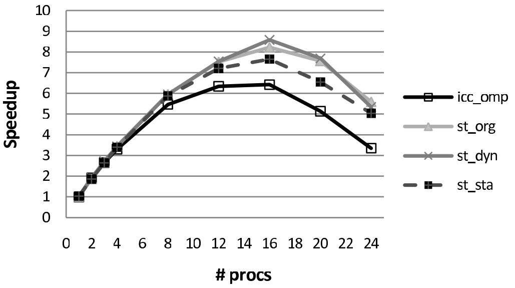


Fig. 17 Speedups of different work stealing strategies and implementations in Strassen's algorithm relative to serial program execution time.

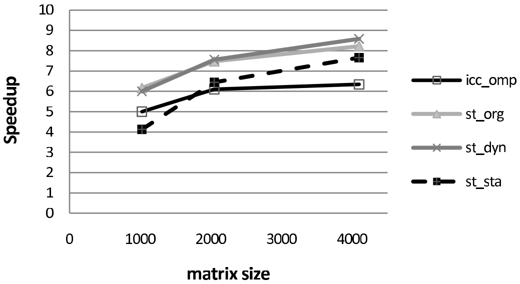


Fig. 18 If the problem size is increased the parallelism of recursive Strassen's computation is increased. This will improve the scalability of the extended strategy. In this figure, the number of stolen threads is two.

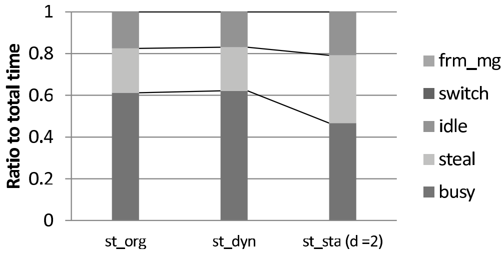


Fig. 19 Data on busy, steal, idle, switch, and frame management overhead of different strategies in Strassen's matrix multiplication. The obtained value is normalized the total time of 16 processors.

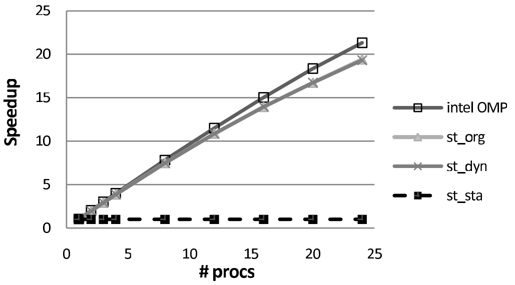


Fig. 20 Speedups of different strategies and implementations in Sparse LU relative to serial program execution time.

and idle are 0.158, 0.43 and 0.4, respectively. In Fig. 16, we show the result of scalability evaluation on this benchmark. With 24 processor cores and bottommost first scheduling, the speedup in the Fibonacci number is 1.90. If the number of stolen threads is increased up to two, the performance is still the same. According to data in Fig. 15 the performance will be lost when the number of stolen threads is increased further. However, in dynamic-length work stealing, the speedup in the Fibonacci benchmark is 1.86.

4.3.3 Evaluation in Other Benchmarks

In the following evaluations, speedups are also measured on the basis of serial program execution time. Figure 17 shows the speedup of Strassen's algorithm using different strategies of work stealing and Intel OpenMP task. From the curves, we found that the dynamic-length work stealing strategy performs as well as the bottommost first strategy of work stealing. However, the fixed-length strategy has a lower performance than the bottommost first strategy in the original StackThreads/MP. In this evaluation, the number of stolen threads is two. The performance will even worsen if we increase the number of stolen threads statically up to two because of the small s_3. Figure 18 shows evidence of this. In the results, we stop at 16 of the 24 available workers. Improvement can be achieved by increasing the matrix size. Increasing the matrix size of Strassen will increase not only s_1 but also s_i for $i \geq 2$. In other words, it generates more parallelism in stacks. Figure 19 shows the data on busy, idle, steal, switch, and frame management overhead. Overhead increases only in st_sta, in which the number of stolen threads is two. Note that busy in Fig. 19 includes the work overhead.

Now, we discuss the performance comparison in Sparse-LU matrix factorization. As in Fig. 20, the fixed-length work stealing strategy fails to scale performance. This is because Sparse-LU did not create many tasks in stacks so that $s_i = 0$ for $i > 2$, where it is impossible to steal more than one thread. Nonetheless, the dynamic-length work stealing strategy could adopt many number of existing threads in a victim. This strategy showed comparable performance to the bottommost first strategy used in the original StackThreads/MP. Figure 21 shows the data on busy, idles, steals, switch, and frame management overhead. It is not surprising that the idle and steal overhead in st_sta are large. Finally, the Intel OpenMP task showed the best result in this Sparse-LU matrix factorization in that the Intel Compiler is well known as a robust compiler. However, this result of Intel has nothing to do with the issues discussed here. Note that busy in Fig. 21 includes work overhead.

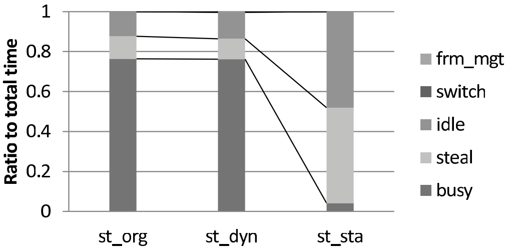


Fig. 21 Data on busy, steal, idle, switch, dan frame management overhead of different strategies in Sparse-LU factorization. The obtained value is normalized to the total time of 20 processors.

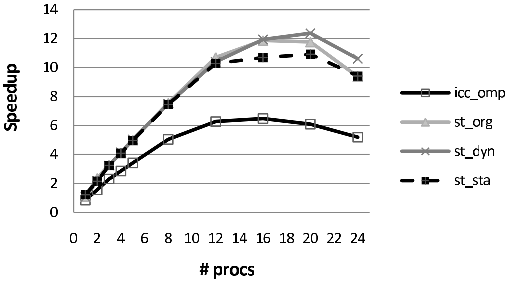


Fig. 22 Speedups of different work stealing strategies and implementations in fast Fourier transform. In this evaluation, the problem size is 64 M of the complex data type.

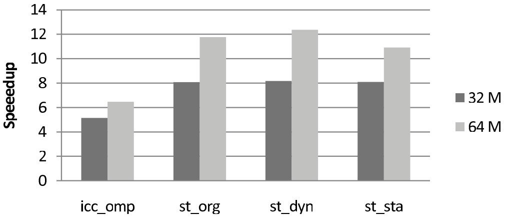


Fig. 23 Speedups of different work stealing strategies and implementations in fast Fourier transform relative to serial program execution time. In this evaluation, the problem sizes are 32 M & 64 M of the complex data type.

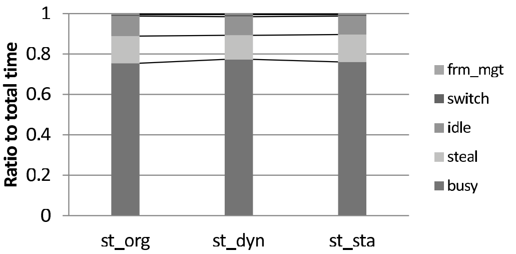


Fig. 24 Data on busy, steal, idle, switch, and frame management overhead of different strategies in fast Fourier transform. The obtained value is normalized to the total time of 20 processors.

Finally, Fig. 22 shows the results of evaluating the fast Fourier transform benchmark. The results look like the those of Strassen in that they are both recursive algorithms. Improvement can be achieved by increasing the problem size. This is because doing so increases not only s_1 but also s_i for $i \geq 2$. The dynamic work stealing strategy achieves comparable performance on 20 processor cores. Figure 23 shows the scalability improvement because the average parallelism increases as the problem size increases. Figure 24 shows the data on busy, idles, steal, and frame management overhead. In the st_sta case, in which the number of stolen threads is two, the steal overhead is 13.5%. This overhead will increase if the number of stolen threads increases. In the st_dyn case, the overhead is 11.9%; the steal overhead in st_sta is 13.4%. Note that busy in Fig. 24 includes the work overhead.

5. Conclusion

By using the bottommost first strategy in the case of deep-recursive and unbalanced tree such as the UTS benchmark, the steal overhead in StackThreads/MP increased. In addition, the frame management strategy, in which StackThreads/MP uses a heap sort, contributed to a high overhead. We fixed these problems using extended work stealing strategies to steal multiple tasks at a time, so that the steal overhead decreased from 28.7% to 6.7%–6.3%; the frame management overhead decreased from 11% to 6.9%–3.9%.

We demonstrated the performance of dynamic-length and fixed-length strategies of work stealing, and compared it to that of the bottommost first strategy used in the original StackThreads/MP. We showed that both the dynamic-length and fixed-length strategies brought more improvements to the UTS throughput than the bottommost first strategy. StackThreads/MP with extended work stealing strategies were more scalable than Cilk performed. In UTS, Cilk runs many slow clones that must increase the critical path overhead.

In Strassen's matrix multiplication, we showed that the fixed-length work stealing strategy experienced performance degradation as the number of the stolen tasks is increased. Moreover, even more in the case of Sparse LU matrix factorization, this strategy did not scale performance at all. We recommend to use the dynamic-length work stealing strategy.

We showed that both the above mentioned extended work stealing strategies performed better in a UTS case, which highly demands load balancing. We prove that the dynamic-length strategy performed better in a recursive/nested parallelism algorithm. Unfortunately, the fixed-length strategy failed to perform in a nonrecursive algorithm, whereas the dynamic-length work stealing strategy showed satisfactory performance as the bottommost first strategy. Hence, controlling load-balancing granularity automatically through a dynamic-length strategy is useful for handling the two types of algorithm mentioned above.

References

[1] E. Mohr, D.A. Kranz, and R.H. Halstead, Jr., "Lazy task creation: A technique for increasing the granularity of parallel programs," IEEE Trans. Parallel Distrib. Syst., vol.2, pp.264–280, July 1991.

[2] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.W. Tseng, "UTS: An unbalanced tree search benchmark," Proc.

19th International Conference on Languages and Compilers for Parallel Computing, LCPC'06, pp.235–250, Berlin, Heidelberg, 2007.

[3] S. Olivier and J.F. Prins, "Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs," Int. J. Parallel Programming, vol.38, no.5-6, pp.341–360, 2010.

[4] M. Frigo, C.E. Leiserson, and K.H. Randall, "The implementation of the cilk-5 multithreaded language," SIGPLAN Not., vol.33, pp.212–223, May 1998.

[5] S.L. Olivier and J.F. Prins, "Evaluating OpenMP 3.0 run time systems on unbalanced task graphs," Proc. 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism, IWOMP '09, pp.63–78, Berlin, Heidelberg, 2009.

[6] OpenMP ARB, "OpenMP application program interface, v.3.0.," Online, 2008.

[7] T.A. El-Ghazawi and L. Smith, "UPC - UPC: unified parallel C," SC, p.27, 2006.

[8] S. Olivier and J. Prins, "Scalable dynamic load balancing using UPC," Proc. 2008 37th International Conference on Parallel Processing, ICPP '08, pp.123–131, Washington, DC, USA, 2008.

[9] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," SIGPLAN Not., vol.30, pp.207–216, Aug. 1995.

[10] R.D. Blumofe and C.E. Leiserson, "Scheduling multithreaded computations by work stealing," J. ACM, vol.46, pp.720–748, Sept. 1999.

[11] K. Taura, K. Tabata, and A. Yonezawa, "Stackthreads/mp: integrating futures into calling standards," SIGPLAN, vol.34, pp.60–71, May 1999.

[12] M. Feeley, "Polling efficiently on stock hardware," Proc. Conference on Functional Programming Languages and Computer Architecture, FPCA '93, pp.179–187, New York, NY, USA, 1993.

[13] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based load balancing," Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (14th PPOPP'09), pp.55–64, Raleigh, NC, USA, Feb. 2009.

[14] P. Berenbrink, T. Friedetzky, and L.A. Goldberg, "The natural work-stealing algorithm is stable," SIAM J. Comput., vol.32, no.5, pp.1260–1279, May 2003.

[15] D. Hendler and N. Shavit, "Non-blocking steal-half work queues," PODC, pp.280–289, 2002.

[16] J. Dinan, D.B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," Proc. Conference on High Performance Computing Networking, Storage and Analysis, SC '09, pp.53:1–53:11, New York, NY, USA, 2009.

[17] J. Dinan, S. Krishnamoorthy, D.B. Larkins, J. Nieplocha, and P. Sadayappan, "Scioto: A framework for global-view task parallelism," ICPP, pp.586–593, 2008.

[18] K. Taura, K. Tabata, and A. Yonezawa, "Integrating futures into calling standar," Technical Report, University of Tokyo, 1999.

[19] K. Taura, StackThreads/MP User's Manual. University of Tokyo, http://venus.is.s.u-tokyo.ac.jp/sthreads/, 2010.

[20] M. Mitzenmacher, "Analyses of load stealing models based on differential equations," SPAA, pp.212–221, 1998.

[21] K.F. Faxen, "Efficient work stealing for fine grained parallelism," Proc. 2010 International Conference on Parallel Processing (39th ICPP'10) CD-ROM, San Diego, CA, pp.313–322, Sept. 2010.

[22] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openMP," Proc. 2009 International Conference on Parallel Processing (38th ICPP'09) CD-ROM, Vienna, Austria, Sept. 2009.

Adnan received his M.T. (M.Eng.) in Electrical Engineering from the Institut Teknologi Bandung in 2002. Currently, he is on leave from his job at Hasanuddin University and is pursuing Ph.D. in the University of Tsukuba.

Mitsuhisa Sato received his M.S. degree and Ph.D. degree in Information Science from the University of Tokyo in 1984 and 1990 respectively. He was a senior researcher at Electrotechnical Laboratory from 1991 to 1996, and a chief of the Parallel and Distributed System Performance Laboratory of Real World Computing Partnership, Japan, from 1996 to 2001. Currently, he is a professor of the Graduate School of Systems and Information Engineering, University of Tsukuba. He has been a director of the Center for Computational Sciences, University of Tsukuba since 2007, and a team leader of the programming environment research team of the Advanced Institute for Computational Science, RIKEN since 2010. His research interests include computer architecture, compilers, and performance evaluation for parallel computer systems, OpenMP, and parallel programming. He is a member of IEEE CS, IPSJ, and JSIAM.