Parallel Graph Traversals using Work-Stealing Frameworks for Many-core Platforms

MASAHIRO YASUGI1,a) TASUKU HIRAISHI2 SEIJI UMATANI1 TAICHI YUASA1

Received: April 11, 2011, Accepted: September 12, 2011

Abstract: Parallel programming/execution frameworks for many/multi-core platforms should support as many applications as possible. In general, work-stealing frameworks provide efficient load balancing even for irregular parallel applications. Unfortunately, naïve parallel programs which traverse graph-based data structures (e.g., for constructing spanning trees) cause stack overflow or unacceptable load imbalance. In this study, we develop parallel programs to perform probabilistically balanced divide-and-conquer graph traversals. We propose a programming technique for accumulating overflowed calls for the next iteration of repeated parallel stages. In an emerging backtracking-based work-stealing framework called “Tascell,” which features on-demand concurrency, we propose a programming technique for long-term exclusive use of workspaces, leading to a similar technique also in the Cilk framework.

Keywords: work stealing, graph traversals, spanning tree, many-core

1. Introduction

With the growing popularity of parallel architectures including many/multi-cores, it has become important to ensure easy parallel programming for efficient parallel computing. An ultimate goal of programming/computing system research is to allow users to describe the computation at a higher level of abstraction and to automatically determine the details of how to perform the computation.

In many irregular applications, static partitioning of work with sufficient concurrency into parallel tasks (for present cores), each with an equal amount of work, is impossible. In such cases, dynamic load balancing, where a task (a piece of work) is dynamically allocated to an idle core, is effective. Work stealing is a randomized technique that implements load balancing. An idle core (thief) steals a task from another randomly selected loaded core (victim).

In practice, users should employ good work-stealing frameworks and not write their own work-stealing protocols by hand, because work stealing involves intentional subtle races. By “subtle,” we mean that things tend to be incorrect without careful handling by experts. For example, some techniques require an elaborate protocol with store-load memory barrier (fence) instructions on modern parallel architectures for ensuring that each potential task is extracted exactly once by the owner or a thief. Therefore, even for a simple application, writing work-stealing code by hand is impractical and error-prone. Termination detection in the presence of work stealing is another subtle issue for general users.

In general, work-stealing frameworks [2], [3], [5], [6], [7], [10], [11] work well with parallel divide-and-conquer (tree-recursive) algorithms, where workers, if necessary, exchange relatively large subdivided tasks near the root of the invocation tree in order to reduce the total work-stealing costs. It is well known that (nested) FORALL-style parallel algorithms can be converted into parallel divide-and-conquer (recursive) algorithms easily (automatically), thereby affording better cache locality in some cases.

Cilk [3] is a parallel programming language. It provides good load balancing for many applications including irregular ones; that is, it keeps all workers busy by creating plenty of “logical” threads and adopting the oldest-first work-stealing strategy. Usually, the number of workers does not exceed the number of underlying computing resources, such as cores and “hardware” threads (afforded by Simultaneous Multithreading (SMT) and Hyper-threading), so that workers actually run in parallel. Cilk employs an implementation technique called LTC (Lazy Task Creation) [11], in which each worker spawns plenty of logical threads and schedules them internally and thus efficiently. An idle worker (thief) may steal (the continuation of) a logical thread from another worker (victim). That is, logical threads are used as tasks dynamically allocated to idle workers. When a logical thread recursively spawns offspring logical threads, the oldest-first work-stealing strategy is generally effective in making tasks larger.

Recently, we proposed a “logical thread”-free parallel programming/execution framework called Tascell as an efficient work-stealing framework [7]. Tascell implements backtracking-based load balancing with on-demand concurrency. A worker performs a computation sequentially unless it receives a task request with polling. When requested, the worker spawns a “real” task by temporarily “backtracking” and restoring its oldest task-spawnable state. Because no logical threads are created as po-

---

1 Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
2 Academic Center for Computing and Media Studies, Kyoto University, Kyoto 606–8501, Japan
a) yasugi@kuis.kyoto-u.ac.jp
tentative tasks, the cost of managing a queue for them can be eliminated. Tascell also promotes the long-term (re)use of workspaces (such as arrays and other mutable data structures) and improves the locality of reference since it does not have to prepare a workspace for each concurrently runnable logical thread.

Unfortunately, naive parallel programs which traverse graph-based data structures (e.g., for constructing spanning trees) cause stack overflow or unacceptable load imbalance. In this research [15], we expand the applicability of work-stealing framework works by the following three steps of our proposals for “parallel search” [15], we expand the applicability of work-stealing frame-

works:
1. Bounding the depth of a call/spawn chain and accumulating overflowed calls/spawns for the next iteration of repeated parallel stages. This prevents stack overflow and also achieves higher performance by realizing probabilistically balanced divide-and-conquer graph traversals.
2. Using workspaces (mutable data structures) for as long a time as possible for accumulating overflowed calls in Tascell. This achieves higher efficiency and good parallel speedups.
3. Enabling long-term reuse of a workspace for multiple logical threads in Cilk by passing around the ownership of the workspace.

In this paper, we employ the parallel spanning tree construction as a running example from Section 4 and then compare and evaluate the performance on the latest many-core platforms.

2. Parallel Programming Languages

2.1 Cilk

In Cilk [3], the programmer specifies parallel functions (cilk procedures). The spawning of a parallel function is written as a C call with an additional spawn keyword. At the language level, a logical thread that executes the parallel function is created. At the implementation level, this child thread is executed immediately (prior to the parent), and (the continuation of) the parent thread becomes stealable for dynamic load balancing. The programmer writes a sync statement so that the parent thread waits for the completion of all spawned child threads. Note that sync statements are compiled away for fast clones [13] at the implementation level. Since each parallel function has sync as its implicit last statement, the child threads cannot survive longer than the parent thread. Thus, the termination of a parallel algorithm is simply detected as the completion of the corresponding parallel function invocation.

Cilk employs a Dijkstra-like (and Dekker-like) protocol called the “THE” protocol for work stealing. When this protocol is implemented on modern parallel architectures that does not provide sequential consistency for shared memory, the owner (the potential victim) is forced to execute store-load memory barrier (fence) instructions when extracting its own potential tasks (logical threads in Cilk); this results in substantial overheads.

2.2 Tascell

The Tascell framework [7] is a load-balancing framework that consists of a compiler for the Tascell language and a runtime system. Although this framework supports both distributed and shared memory environments, we consider only shared memory environments in this paper.

Idle workers request tasks from loaded workers. When receiving a task request, a loaded worker creates a new task by dividing the current running task, and returns the new task to the idle worker. When an idle worker receives a task, it executes the task by calling worker functions and returns the result of the task. A task and its result are represented by a task object.

In worker functions, which are specified by the keyword worker (like cilk procedures in Cilk), we can use Tascell’s task division constructs. A parallel for loop construct can be used for dividing an iterative computation. It is syntactically denoted by:

```
for(int identifier : expression_from, expression_to) statement_body
```

handles task-name (int identifier_from, int identifier_to)

{ statement_put statement_get }
```

This iterates statement_body over integers from expression_from (inclusive) to expression_to (exclusive). When the implicit task-request handler (available during the iterative execution of statement_body) is invoked, the upper half of the remaining iterations are spawned as a new task-name task, whose object is initialized by statement_put. In statement_put, the actual assigned range can be referred to by identifier_from and identifier_to.

The worker handles the result of the spawned task by executing statement_get. Note that a worker performs iterations for a parallel for loop sequentially unless requested; the worker does not create any logical threads and can (re)use a single workspace (such as a worker-local array) for a long time.

Parallel for statements may be nested dynamically in their statement_body. Therefore, multiple task-request handlers may be available at the same time. Each worker attempts to detect a task request by polling at every parallel for statement without heavy memory barrier (fence) instructions. When the worker detects a task request, it performs temporary backtracking in order to spawn a larger task by invoking as old a handler as possible (see Fig. 1).

The current implementation of Tascell, when a worker waits for the result of a stolen task, it tries to steal (and executes) another task of the task requester until the result is returned (see Fig. 2)2.

The Tascell compiler employs an extended C language as the intermediate language. In the first compilation phase, a Tascell program is translated into an extended C program with nested function definitions in order to implement task-request handlers [7]. In the second compilation phase, the extended C program with nested functions is compiled by an enhanced version of GCC [16] or by a translator into standard C [8].

---

1. This saves the execution stack as in Leaptrogging [14]. TBB [9] employs a more general technique for saving the execution stack.

2. Specifying a task definition and several statements to handle task objects makes Tascell programs more verbose than Cilk programs. These costs are necessary for more exact control of workspaces and distributed memory environment support.
3. Parallel Graph Traversals

In some parallel graph applications, multiple workers may traverse graph-based shared data structures in parallel. They may visit a vertex at almost the same time. When a worker visits a vertex, the worker may perform some computation and modification. A worker may choose an edge to follow and arrive at the adjacent vertex and leave the other edges as potential tasks. Another worker may steal a task to follow such an edge.

In order to make such parallel graph traversals correct and efficient on real many/multi-core architectures, we must address the following three issues:

(1) concurrent access to vertices and related data structures may involve races among workers,

(2) work stealing among workers involves subtle races for potential tasks, and

(3) termination detection is difficult.

In general, emerging transactional memory may solve the races in issue (1), but lighter-weight solutions may exist for each “concurrent access” problem.

In order to control subtle races in issue (2) and write correct programs, existing common work-stealing frameworks should be employed. Writing work-stealing code for each application is impractical and error-prone. For example, XWS [2] (X10 Work Stealing framework) is such an open-source framework. In this paper, we employ and compare parallel programming languages such as Cilk [3] and Tascell [7] for correct work stealing.

In order to correctly detect termination in issue (3), each worker (core) must participate in some additional protocol. Otherwise, all workers may keep trying to steal a task from one another forever, or some workers may finish earlier and use an incomplete final answer. Thus, this issue has to be addressed in real implementations; for example, XWS provides a mechanism to detect global quiescence (“all work queues are empty”). Again, in this paper, we simply employ parallel programming languages for termination detection.

4. Parallel Spanning Tree Construction

Some parallel graph problems are well described as parallel graph traversals. Finding a spanning tree of a given connected graph is such a problem. We take a closer look at this simple graph problem as a running example hereafter. Note that the connectivity test is essentially the same problem. Since connectivity is an important property of any kind of network, attaining fast, parallel spanning-tree construction is important. Parallel garbage collectors also employ similar traversal algorithms for identifying live objects.

Many theoretically elaborate, parallel (PRAM) connectivity (spanning-tree) algorithms are known; they are mostly based on the Shiloach-Vishkin algorithm [12] with $O(\log n)$ time and $O(m + n \log n)$ work for any graph with $n$ vertices and $m$ undirected edges.

Bader and Cong [1] proposed a fast, parallel spanning-tree algorithm for SMPs with $O((n + m)/p)$ time and $O(n + m)$ work, where each worker performs a simple, serial, depth-first algorithm with work stealing. They reported that their parallel spanning-tree algorithm achieved parallel speedups on SMPs with $p$ processors for the first time even when the graph was sparse.

A basic, fast, serial, depth-first $O(n + m)$ algorithm can be implemented as a C program, as shown in Fig. 3. Even when implementing this simple algorithm on real many/multi-core architectures, we must address the three issues mentioned in Section 3. The first issue would be as follows: (1) stores in line 13 are races among processors to win an unvisited vertex. Bader and Cong proved that the races in issue (1) do not cause a problem, such as cycle construction if the sequential consistency is assumed. However, modern relaxed memory models require more accurate implementations. Furthermore, we address the remaining two issues (2) races for work stealing and (3) termination detection) with
higher confidence of safety/correctness by using existing carefully implemented work-stealing frameworks.

5. Our Proposals

In order to implement the running example in parallel work-stealing programming languages, we propose the use of a simple technique for bounded execution stacks. That is, we can simply bound the depth of a call chain and accumulate overflowed calls for the next iteration. Since the depth-first traversal may become extremely deep, without this technique, the worst-case space complexity of the execution stack is $O(n)$, resulting in stack overflow [2].

The C program in Fig. 4 illustrates a preliminary to our proposals; it constructs a spanning tree in a manner similar to Fig. 3. Instead of a vertex stack, the program uses recursive calls (on an execution stack) of serial_search_call in line 10 for the depth-first traversal. To prevent execution stack overflow, the depth of the recursive calls is limited by a constant (C_LIMIT, in Fig. 4).

We set C_LIMIT to 30 because sufficient work with $2^{30}$ calls is covered if the resultant invocation tree is well-balanced. This C_LIMIT value is also sufficiently small to prevent stack overflow. Note that the limitation of depth also improves the cache locality of an execution stack. The overflowed vertices are saved in line 12. The next stage with a collection of overflowed vertices is executed by search_main in lines 22–30 repeatedly until the collection becomes empty. At the beginning of each stage, v_list is used as the input collection of vertices from the previous stage; the pointer to the collection and its length are used to iterate over the collection in a doubly nested manner (with variables v_array and len) and to free the memory (with variable v_list0) after use while the variable v_list is reinitialized to accumulate newly overflowed vertices.

We use the program in Fig. 4 as the starting point of parallel programming. First, we manage the tractable “races” to mark unvisited vertices "visited." Figure 5 shows the possible solutions obtained by using (a) a compare-and-swap (cas) primitive and (b) a store-load memory barrier. For x86, 64, cas_int is implemented as a test followed by the lock cmpxchg instruction (aka compare-and-compare-and-swap), and a store-load memory barrier is implemented with the mfence instruction. For SPARC-V9, cas_int is implemented as a test followed by the cas instruction, and a store-load memory barrier is implemented with the membar#StoreLoad instruction.

Now, we focus on the details of parallel programming with Cilk and Tascell. In these languages, the termination of a parallel
algorithm is simply detected (or expressed) as the completion of
the corresponding (parallel) function invocation.

5.1 Depth Limiting in Cilk

In Fig. 6, the sequential calls of serial_search_call (in
lines 10 and 28 in Fig. 4) are replaced with the spawns of a par-
allel function parallel_search_spawn. In the Cilk implementa-
tion, this means the depth-first traversal (in the same order of
the recursive calls on serial_search_call) with stealable con-
tinuations just after the spawning of parallel_search_spawn.
When stolen, the remaining iterations over the edges are moved
to the thief worker.
This function is simple, but it needs properly synchronized
concurrent access to a vertex accumulator v_list (as a concur-
rent workspace) in line 11 in Fig. 6. In order to address this prob-
lem, we prepare an individual vertex accumulator for each logical
thread for the synchronization-free exclusive use.

Figure 7 shows the revised parallel function parallel_search_spawn. In Fig. 7, a parent thread adds
its children’s accumulated vertices into its own vertex accumu-
lator by using an inlet concatenate (which performs destructive list
concatenation with simple arithmetic/pounter manipulations) in
lines 6–8. We should also modify the internal representation of
vertex accumulators; namely, we should use very small arrays
(as list elements) of size V_ARRAY_SIZE for space efficiency.

In order to complete a parallel program in Cilk, we must par-
allelize the C function search_main (in lines 17–31 in Fig. 4).
After the first depth-limited traversal in line 21 or after each
stage with depth-limited traversals in line 28, overflowed ver-
tices are collected in a list of arrays v_list. After the par-
allelization shown in Fig. 7, (part of) this list is returned by
parallel_search_spawn and concatenated with an additional
concat inlet for search_main (code omitted). We have a dou-
bly nested loop in lines 26–28 in Fig. 4. The inner loop is sim-
ple, but the outer loop iterates over a list. We address this prob-
lem by converting the list into an array before entering the dou-
bly nested loop. Now, we have an array of arrays (of vertices
with irregular work), over which we can perform depth-limited
traversals in parallel by applying the standard, dividing-into-two-
halves-recursively, parallel divide-and-conquer technique.

Now, as the first step of our proposals, we obtain a parallel
program that does not suffer from stack overflow. As will be
shown in Section 6, this program can stably be used for larger
graphs and achieves higher performance than a parallel program
without depth limiting since the depth limiting serves as a guide
and realizes probabilistically balanced divide-and-conquer graph
traversals.

5.2 Tascell (Long-term Use of Workspaces)

As the second step of our proposals, we address the inefficien-
ty of the short-term use of vertex accumulators discussed in the
previous section.

First, the sequential calls of serial_search_call (in
lines 10 and 28 in Fig. 4) are replaced with the calls of a
Tascell worker function parallel_search_call in Fig. 8. The worker function parallel_search_call is based on
the C function serial_search_call in Fig. 4. Note that
parallel_search_call is not spawned (as a logical thread)
in line 21 in Fig. 8; rather, it is called sequentially just like
serial_search_call. This function employs a parallel for
In order to parallelize the C function `search_main` (in lines 17–31 in Fig. 4) using Tascell, we can simply employ doubly nested parallel for statements for the doubly nested loop in lines 26–28. In Tascell, we can use the parallel for construct for the outer loop that iterates over a list without list-to-array conversion by adopting the skipping method as in lines 14–15 in Fig. 8.

Now, as the second step of our proposals, we obtain a parallel program that exclusively uses updatable workspaces for a long time. As will be shown in Section 6, this program works well even when a large number of overflowed vertexes are collected.

5.3 Long-term Reuse of Workspaces in Cilk

From the Tascell program in the previous section, it is natural to look for a better Cilk program that emulates the long-term use of the workspace for a vertex accumulator as the third step of our proposals. In fact, this is possible by tricky use of the pseudovariable `SYNCHED`, although `SYNCHED` was originally introduced to promote the reuse of a workspace among child logical threads [13] and usually it cannot be used for the reuse of a workspace between parent and child logical threads.

In Fig. 9, the tricky parallel function `parallel_search_spawn_synched` is shown. In Cilk, the pseudovariable `SYNCHED` is true if all previously spawned child logical threads are completed. When a steal does not occur, the parent and child threads can pass around the ownership of a workspace for their own non-overlapping period. When the parent thread is stolen, it can realize the steal by checking `SYNCHED` just after spawning and can allocate its own workspace for a new vertex accumulator. Notice that, in Fig. 9, the parallel function passes the workspace pointed to by `v_list_p` to its child logical threads recursively; the ownership of a workspace is passed around recursively when a steal does not occur. Consequently, a much smaller number of allocations are performed, and a larger
array size can be employed.

Although this approach is tricky, it would be effective if performance enhancement is preferred over clarity in the algorithm’s description.

6. Evaluation

In this section, we compare and evaluate the performance of spanning-tree construction programs written in C, Cilk, and Tascell using various graphs to evaluate (1) our depth-limiting approach and (2) the long-term use of workspaces.

We use two many/multi-core machines as evaluation platforms (see Table 1). The first is a Linux PC server (x86-64) with two 2.67-GHz Intel Xeon X5650 processors featuring 12 cores (24 hardware threads afforded by Hyper-threading) in total. The second is an UltraSPARC T2 Plus server (SPARC) with two 1.4 GHz UltraSPARC T2 Plus processors featuring 16 cores (128 hardware threads afforded by SMT) in total.

By referring to previous studies [1], [2], [4], [10], we employ the following graph types:

- **Random**($n, M$) has $n$ vertices and approximately $2Mn$ edges, where each vertex picks $M$ neighbors at random (uniformly). The average degree is approximately $2M$. This type is a generalization of “tertiary” graphs$^4$ [4].

- **Hypercube**($N$) is a hyper-cube graph that has $2^N$ vertices with degree $N$.

- **2D-torus**($N$) is a two-dimensional torus that has $N \times N$ vertices with degree 4.

- **Bintree**($N$) is a binary tree graph that has $2^N - 1$ vertices and $2^N - 2$ edges. The degree is mostly 1 or 3. Its spanning tree is unique (up to direction reversal), but the parallel programs should also support this type.

In order to alleviate the positive or negative performance impact caused by an accidental affinity between an inherently free but unintentionally biased program behavior (such as a traversal order over neighbors) and the underlying memory subsystem (cache hierarchy) as much as possible, we simply randomized the order of edges for each vertex during the graph construction.

6.1 Effects of Depth Limiting

The primary effect of depth limiting appears as the prevention of stack overflow. The elapsed time of the simple parallel depth-first Cilk program without depth limit cannot be shown for the large Hypercube($N$) with $N \geq 19$ (or 2D-torus($N$) with $N \geq 1024$) graphs in Fig. 10. In contrast, depth-limited programs can accept much larger graphs.

Furthermore, the depth-limited programs run significantly faster than the simple parallel depth-first Cilk program since the depth limiting serves as a guide and provides probabilistically balanced divide-and-conquer graph traversals.

6.2 Effects of Long-term Use of Workspaces

Figure 11 illustrates the runtime overheads caused by code transformation or parallelization for four graphs: Random(4000000,2), Hypercube(21), 2D-torus(4000), and Bintree(24). In Fig. 11, the elapsed time on a single core is normalized to the elapsed time of the “serial” C program with a large vertex stack (in Fig. 3). As is often the case, the transformed
C program for depth-limited recursive calls (in Fig. 4) exhibits shorter elapsed time (as “depth limiting (serial)” for two out of four graphs. Table 2 shows the measured absolute elapsed time and the number of stages in lines 22–30 in Fig. 4, as well as some profiles of graphs.

The overheads accompanying the use of (a) a compare-and-swap (“cas”) primitive or (b) a store-load memory barrier (“membar”) (illustrated in Fig. 5) are substantial, which are shown as the differences between “depth limiting” cases with or without “cas” or “membar.” From these results, we can say that the cost of (b) a store-load memory barrier is higher (or lower) than that of (a) a compare-and-swap primitive on our x86-64 (or SPARC) machine.

Short-term use of workspaces in Cilk (illustrated in Fig. 7 in Section 5.1) requires some overheads. On the other hand, each Tascell worker can employ a single worker-local workspace for a long time for collecting overflowed calls (vertices) thanks to the semantics of sequential execution with on-demand concurrency, thereby achieving higher performance. Long-term reuse of workspaces among multiple logical threads in Cilk (illustrated in Section 5.3) also improves the overheads, mostly when a large number of overflowed vertices are collected. Note that in the short-term use of workspaces (shown in Fig. 7), the spawned thread for the parallel function on every visited vertex v executes some instructions to initialize (in line 5), concatenate to (in line 7), and return (in line 21) my_v_list of type struct v_list (v_list_t), while the long-term (re)use of workspaces does not require such instructions for every vertex.

In Fig. 12, we show parallel speedups to the serial C program (in Fig. 3) for four graphs. In general, long-term (re)use of workspaces achieves good speedups (both in Tascell and Cilk).

The speedup differences between long-term use and short-term use are significant when a large number of overflowed vertexes are collected (namely, in order, for 2D-torus, Hypercube, and Random graphs). Note that the 2D-torus is the most difficult problem to parallelize because newly found neighbors are often “visited.”

Table 3 shows the number of memory allocations on twelve workers (x86-64). For Random, Hypercube, and 2D-torus, the long-term (re)use of workspaces (both in Cilk and Tascell) shows the much (two orders of magnitude) smaller number of allocations than the short-term use, showing the effectiveness of the long-term (re)use.

Table 3 also shows the space overheads (the ratio of the unused part to the used part), since there is a trade-off between the number of allocations and the space overheads when choosing the allocated array size: for Random and Hypercube, the long-term use shows much lower space overheads than the short-term use. For 2D-torus, the space overheads of the long-term use are less than twice of the short-term use.

The remaining performance gaps between Tascell and Cilk both with long-term use of workspaces can be attributed to the difference between the basic performance of these systems, as presented in previous work [7]; for example, the “THE” protocol for work stealing in Cilk requires an expensive memory barrier other than for tractable races for vertices.

7. Discussion

7.1 Related Work

Cong et al. [2] proposed a strategy for the adaptive control of the granularity of parallel tasks, on the basis of the instantaneous size of the work queue in the X10 Work Stealing framework. Their schemes (adaptive batching schemes) alleviate the overheads of the “THE” protocol. In essence, their queue-sensing adaptation belongs to the polling-based approach, suggesting that the polling-based approach can eliminate the cost of spawning/managing logical threads.

Michael et al. [10] proposed new protocols based on the novel concept of idempotent work stealing, which provides relaxed work-stealing semantics where each potential task is extracted at least once instead of exactly once. Unlike the “THE” protocol, this work-stealing technique frees the owner from heavy memory barrier (fence) instructions. However, this scheme cannot be employed directly to improve Cilk’s performance because duplicated (continuations of) logical threads would have unclear semantics.

Yi Guo et al. [5] proposed an X10 runtime, which provides multiple implementations for work-first and help-first scheduling policies. They showed that the help-first scheduling policy is useful for the simple parallel depth-first traversals (without depth limiting). They also proposed a way to automatically and adaptively change the scheduling policies at runtime [6].

The abovementioned series of studies focused on improvement on system implementations (mostly based on Java). In this study, we rather increase the applicability of existing work-stealing frameworks by using practical programming techniques.
7.2 System-level Approaches

The language systems we employed in this paper are based on C’s stacks. Some systems may use a heap to manage function/thread frames. An obvious drawback of such systems is the high frame management cost, although we can reduce some cost by using stacks as bounded temporary spaces other than heaps [17]. In addition, even if stack overflow can be avoided, extremely deep recursion seems to cause performance degradation as Fig. 10 shows in the range of problem sizes without stack overflow. Since Fig. 10 uses Cilk as the stack overflow example and Cilk does not use “stack walk” for the work stealing, using v_list as an vertex accumulator and a recursively divisible input collection of vertices at the next stage enables probabilistically balanced divide-and-conquer graph traversals, resulting in good load balancing.

In Tascell, we can use worker-local workspaces effectively with its steal awareness. In Cilk, we can use the pseudovariable SYNCHED to write a steal-aware program where the long-term reuse of a single worker-local workspace can be realized indirectly. Note that the direct use of worker-local storage in work-stealing multithreaded languages like Cilk does not work without steal awareness in theory because the other worker may steal a logical thread that has performed some of an atomic series of operations on a worker-local workspace and wants to complete the rest of the operations on the same worker-local workspace.

7.3 Other Programming Techniques

Stack overflow can be avoided if we stop the recursive work-stealing (parallel) calls/spawns at some depth limit and perform further search sequentially as in Fig. 3.

Unfortunately, there are cases (input graphs) where the loss of parallelism is quite unacceptable. If (1) the input graph consists of two meshes with a link as shown in Fig. 13, (2) the depth limit is 10, and (3) the root of the spanning tree is the left-most bottom-most vertex in Fig. 13, then the work-stealing (parallel) tree construction is performed only with the black vertices (within the depth limit) in Fig. 13; multiple workers/threads will compete...
for white vertices in the left hand side mesh, but only a single worker/thread can visit white vertices in the right hand side mesh after following the single link. Since the single worker/thread visits all white vertices in the right hand side mesh sequentially, we severely lose a big parallel speedup opportunity when visiting the right hand side mesh. If the right hand side is a larger (e.g., 10 times larger) subgraph, the loss of parallelism is more severe.

7.4 Other Applications

Figure 14 illustrates how we can perform concurrent rewriting for “general computations,” including $\lambda$-calculus, graph/term rewriting systems, garbage collectors, and so on.

By employing appropriate graph-based data structures as configurations in Fig. 14, we can regard parallel graph traversals for constructing spanning trees as special cases of “general computations.” Conversely, the techniques proposed here can be applied...
Table 3  Numbers of allocations in typical runs on twelve workers (x86-64).

<table>
<thead>
<tr>
<th>graph</th>
<th>proposal</th>
<th># of overflown allocations</th>
<th># of allocations</th>
<th>space overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random(4000000,2)</td>
<td>short-term (Cilk)</td>
<td>350,812</td>
<td>136,339</td>
<td>17%</td>
</tr>
<tr>
<td></td>
<td>long-term (Cilk)</td>
<td>356,145</td>
<td>599</td>
<td>0.91%</td>
</tr>
<tr>
<td></td>
<td>long-term (Tascell)</td>
<td>706,505</td>
<td>1,183</td>
<td>0.47%</td>
</tr>
<tr>
<td>Hypercube(21)</td>
<td>short-term (Cilk)</td>
<td>1,024,891</td>
<td>437,266</td>
<td>28%</td>
</tr>
<tr>
<td></td>
<td>long-term (Cilk)</td>
<td>1,025,006</td>
<td>1,747</td>
<td>2.3%</td>
</tr>
<tr>
<td></td>
<td>long-term (Tascell)</td>
<td>1,025,064</td>
<td>1,724</td>
<td>0.91%</td>
</tr>
<tr>
<td>2D-torus(4000)</td>
<td>short-term (Cilk)</td>
<td>4,019,435</td>
<td>2,163,449</td>
<td>61%</td>
</tr>
<tr>
<td></td>
<td>long-term (Cilk)</td>
<td>4,014,040</td>
<td>14,397</td>
<td>115%</td>
</tr>
<tr>
<td></td>
<td>long-term (Tascell)</td>
<td>4,012,193</td>
<td>12,165</td>
<td>82%</td>
</tr>
<tr>
<td>Bintree(24)</td>
<td>short-term (Cilk)</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>long-term (Cilk)</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>long-term (Tascell)</td>
<td>0</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Fig. 13  An input graph example where the loss of parallelism is quite unacceptable in limited work-stealing approaches.

(a) a computation with rewrite steps \( r \) over configurations \( C_i \);
C = initial configuration \( C_i \);
while (there exists a redex (rule and position) in \( C \))
C = apply arbitrarily chosen redex of \( C \), \( C_i \);

(b) an abstract machine for the above computation.
C = initial configuration \( C_i \);
S = initial redex set of \( C \);
while (S is not empty) { // elements of \( S \) are steerable.
  redex \( r \) = pop (S);
  atomically:
  ( if \( r \) is applicable)
    \( (C, S') = apply(r, C) \);
  ( // \( C \) the next configuration
    \( S = S : S' \) // \( S \) may be a multiset.
}

(c) a more efficient event-driven abstract machine that may run in parallel if the atomicity of concurrent rewrites is preserved. Concurrent rewrites are permitted if independent parts of \( C \) are rewritten.

Fig. 14  Concurrent rewriting.

to general computations. Since the number of rewrite steps corresponds to the depth of the depth-first traversal, our techniques are quite important for preventing stack overflow or unacceptable load imbalance on real machines.

8. Summary and Future Work

Writing work-stealing code or termination-detection code correctly by hand is impractical/error-prone, irrespective of the simplicity of applications. Work-stealing frameworks, including the XWS framework [2], are intended to solve such issues. In this paper, we argued that parallel languages for work stealing are more suitable for solving such issues since we can write efficient programs instead of adjusting framework parameters.

The unlimited depth-first traversal is not a divide-and-conquer application for work-stealing frameworks. In this research[15], we address this issue by adopting depth limitation, thereby realizing more balanced divide-and-conquer patterns and enabling us to use parallel languages/systems with reasonable stack sizes.

In our proposals, long-term (re)use of a single workspace is very important for achieving good efficiency by avoiding costs of memory management, mutual exclusion, and poor locality. Depth limitation requires inexpensive workspaces for collecting overflowed calls (potential tasks). We showed how efficiently we can use workspaces in Tascell. In Cilk, we realized long-term reuse of a workspace for multiple logical threads by passing around the ownership of a workspace.

For more general “concurrent rewriting,” every concurrent rewrite should be committed as a single transaction. Thus, integration of transactional memory frameworks and work-stealing frameworks might be an interesting future research area.

Acknowledgments  This work was supported in part by a MEXT Grant-in-Aid for Scientific Research on Priority Area (21013027 Area #456), a JSPS Grant-in-Aid for Scientific Research (B) (21300008), and a MEXT Grant-in-Aid for Young Scientists (B) (22700030).

Reference


Masahiro Yasugi was born in 1967. He received a B.E. in electronic engineering, an M.E. in electrical engineering, and a Ph.D. in information science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. Since 1998, he has been working at Kyoto University as an associate professor. In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of the ACM and the Japan Society for Software Science and Technology. He was awarded the 2009 IPSJ Transactions on Programming Outstanding Paper Award.

Tasuku Hiraishi was born in 1981. He received a B.E. in informatics and mathematical science in 2003, an M.E. in informatics in 2005, and a Ph.D. in informatics in 2008, all from Kyoto University. In 2007–2008, he was a fellow of the JSPS (at Kyoto University). Since 2008, he has been working at Kyoto University as an assistant professor at the Supercomputing Research Laboratory, Academic Center for Computing and Media Studies, Kyoto University. His research interests include parallel programming languages and high performance computing. He was awarded the 2009 IPSJ Transactions on Programming Outstanding Paper Award. He is a member of the Japan Society for Software Science and Technology (JSSST).

Seiji Umatani was born in 1974, and received a B.E. degree in information science, and M.E. and Ph.D. degrees in informatics from Kyoto University, Kyoto, Japan, in 1999, 2001, and 2004, respectively. In 2004–2005, he was a research staff member in the Graduate School of Informatics at Kyoto University, and he was appointed to an assistant professor position in 2005. His current research interest includes programming languages, compilers, and parallel/distributed systems. He is a member of the ACM and the Japan Society for Software Science and Technology.

Taiichi Yuasa received the Bachelor of Mathematics degree in 1977, the Master of Mathematical Sciences degree in 1979, and the Doctor of Science degree in 1987, all from Kyoto University, Kyoto, Japan. He joined the faculty of the Research Institute for Mathematical Sciences, Kyoto University, in 1982. He is currently a professor at the Graduate School of Informatics, Kyoto University, Kyoto, Japan. His current area of interest include symbolic computation and programming language systems. Dr. Yuasa is a member of ACM, IEEE, the Institute of Electronics, Information and Communication Engineers, and the Japan Society for Software Science and Technology.