Work-First and Help-First Scheduling Policies for Terminally Strict Parallel Programs

Rajkishore Barik, Yi Guo, Raghavan Raman, Vivek Sarkar

Rice COMP TR08-07

November 2008
Work-First and Help-First Scheduling Policies for Terminally Strict Parallel Programs*

Yi Guo  Rajkishore Barik  Raghavan Raman  Vivek Sarkar
Department of Computer Science, Rice University
{yguo, rajbarik, raghav, vsarkar}@rice.edu

Abstract

Multiple programming models are emerging to address an increased need for dynamic task parallelism in applications for multicore processors and shared-address-space parallel computing. Examples include OpenMP 3.0, Java Concurrency Utilities, Microsoft Task Parallel Library, Intel Thread Building Blocks, Cilk, X10, Chapel, and Fortress. Scheduling algorithms based on work stealing, as embodied in Cilk’s implementation of dynamic spawn-sync parallelism, are gaining in popularity but also have inherent limitations. In this paper, we address the problem of efficient and scalable implementation of X10’s terminally strict async-finish task parallelism, which is more general than Cilk’s fully strict spawn-sync parallelism. We introduce a new work-stealing scheduler with compiler support for async-finish task parallelism that can accommodate both work-first and help-first scheduling policies. Performance results on two different multicore SMP platforms show significant improvements due to our new work-stealing algorithm compared to the existing work-sharing scheduler for X10, and also provide insight on scenarios in which the help-first policy yields better results than the work-first policy and vice versa.

1 Introduction

The computer industry is entering a new era of mainstream parallel processing due to current hardware trends and power efficiency limits. Now that all computers — embedded, mainstream, and high-end — are being built using multicore chips, the need for improved productivity in parallel programming has taken on a new urgency. The three programming languages developed as part of the DARPA HPCS program (Chapel, Fortress, X10) all identified dynamic lightweight task parallelism as one of the prerequisites for success. Dynamic task parallelism is also being included for mainstream use in many new programming models for multicore processors and shared-memory parallelism, such as Cilk, OpenMP 3.0, Java Concurrency Utilities, Intel Thread Building Blocks, and Microsoft Task Parallel

*This work was supported in part by the National Science Foundation under the HECURA program, award number CCF-0833166. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.
Library. In addition, dynamic data driven execution has been identified as an important trend for future multicore software, in contrast to past programming models based on the Bulk Synchronous Parallel (BSP) and Single Program Multiple Data (SPMD) paradigms. Scheduling algorithms based on work stealing, as embodied in Cilk’s implementation of spawn-sync parallelism, are gaining in popularity for dynamic lightweight task parallelism but also have inherent limitations.

In this paper, we introduce a new work-stealing scheduler with compiler support for async-finish task parallelism that can accommodate both work-first and help-first scheduling policies. The steal operations in the help-first scheduler are non-blocking, which is important for scalability in situations when steals can otherwise become a serial bottleneck. We study on the performance differences between work-first and help-first scheduling policies and shed insight on scenarios in which the help-first policy yields better results than work-first policy and vice versa. Performance results show significant improvements (up to 4.7x on a 16-way Power5+SMP and 22.9x on a 64-thread UltraSPARC II) for our work-stealing scheduler compared to the existing work-sharing scheduler for X10 [2].

The rest of the paper is organized as follows. Section 2 summarizes the differences between Cilk-style fully strict spawn-sync parallelism and X10-style terminally strict finish-async parallelism, as well as past work on the Cilk work-stealing scheduler. Section 3 summarizes two key limitations of Cilk-style work-stealing schedulers in handling finish-async task parallelism, namely escaping async’s and sequential calls to parallel functions, and our approach to addressing these limitations in a work-stealing framework. Section 4 introduces the help-first scheduling policy. Section 5 presents performance results on two different multicore SMP platforms. Section 6 discusses related work, and Section 7 contains our conclusions.

2 Background

In this section, we first compare fully strict computations as in Cilk-style spawn-sync task parallelism with terminally strict computations as in X10-style finish-async task parallelism and then briefly summarize the Cilk work-stealing scheduler for fully strict spawn-sync computations [7].

2.1 Fully Strict vs. Terminally Strict Parallelism

Blumofe et al. [3] defined the notion of fully strict computation for Cilk-style spawn-sync task parallelism as follows. Each multithreaded computation can be viewed as a dag of instructions connected by dependency edges. The instructions in a task are connected by continue edges, and tasks form a spawn tree with spawn edges. Join edges are introduced to enforce the ordering required by sync operations. A fully strict computation is one in which all join edges from a task go to its parent task in the spawn tree. All executions of Cilk programs are required to be fully strict.

This definition of fully strict computations was extended to terminally strict computations for X10-style finish-async task parallelism in [1] as follows. As in the case of Cilk, an X10 computation can also be represented as a dag in which each node corresponds to a dynamic execution instance of an X10 instruction/statement, and each edge defines a precedence constraint between two nodes. Figure 1 shows an example X10 code fragment and its
computation dag. The first instruction of the main activity \(^1\) serves as the root node of the dag (with no predecessors). Any instruction which spawns a new activity will create a child node in the dag with a spawn edge connecting the async instruction to the first instruction of that child activity. **X10** activities may wait on descendant activities by executing a finish statement. We model these dependencies by introducing start fin (l2start in Figure 1) and end fin (l2end in Figure 1) nodes in the dag for each instance of a finish construct and then create join edges from the last instruction of the spawned activities within the scope of finish to the corresponding end fin instruction.

While there are many similarities between an **X10** computation dag and a **Cilk** computation dag, there are also some interesting differences. For example, the direct join edge from activity \(\Gamma_2\) to activity \(\Gamma_0\) in Figure 1 is not allowed in **Cilk** because \(\Gamma_0\) is not \(\Gamma_2\)'s parent in the dag. The only way to establish such a dependence in **Cilk** is via \(\Gamma_1\). In **X10**, it is possible for a descendant activity (e.g., \(\Gamma_2\)) to continue executing even if its ancestor activity (e.g., \(\Gamma_1\)) has terminated. This degree of asynchrony can be useful in parallel divide-and-conquer algorithms so as to permit sub-computations at different levels of the divide-and-conquer tree to execute in parallel without forcing synchronization at the parent-child level.

The class of computation defined by **X10**'s finish-async style is called terminally strict computations since the source of the join edge can only be the last instruction of a task. Terminally strict computations are a non-trivial superset of **Cilk**'s fully strict computations. It has been proved that the **Cilk**'s space and time bounds hold for terminally strict computations \([1]\), but that work did not include an implementation of work stealing algorithms for terminally strict computations.

### 2.2 Cilk Work-Stealing Scheduler

The **Cilk** work stealing runtime comprises of a set of workers, typically one per CPU or core. Each worker maintains a local deque of frames which represent work. The runtime starts with one worker executing the main function and the rest being idle with empty deques. Whenever a worker is idle, it becomes a thief and attempts to steal work from another workers' deque. On a spawn, the continuation is saved in a frame which is pushed onto the worker's deque so that other workers can steal it. Whenever the worker returns from a spawned task, it will first check if the frame that it pushed before the spawn is stolen. If so, the fully-strict model guarantees that there is no other useful work on the deque for the worker and hence

---

\(^1\)The terms “activity” and “task” are used interchangeably in this paper.  

it becomes a thief. Otherwise, it just executes the continuation of the spawn. Every cilk function has two clones generated by the compiler: the fast clone and the slow clone. When the thief steals a frame from a victim worker, it starts executing the stolen continuation by invoking the slow clone, which performs the necessary operations to restore local variables for the continuation.

3 Work-Stealing Extensions for Terminally Strict Computation

As discussed in Section 2, the class of terminally strict computations is a non-trivial superset of the class of fully strict computations. In this section, we first identify the two new features of terminally strict computations that demand compiler and runtime support for work stealing, but have not been addressed by past work. Then we present our approach to supporting the two features. The approach described in this section is a common framework used by both the work-first and help-first policies described in the next section.

3.1 Escaping Asyncs

The first feature is escaping asyncs, which are defined as the tasks that may outlive their parents. For example, consider the parallel-DFS spanning tree graph algorithm [6] as shown in Figure 2. In a terminally strict language, like X10, a single finish scope at line 19 suffices for all descendant tasks spawned at line 14. The spawned tasks can still be alive after its parent (compute) returns. However, a fully strict computation implementation, like Cilk, will automatically insert an implicit sync operation at the end of each function thereby ensuring the parent waits for all its children. Semantically, it is equivalent to adding a finish scope to enclose the body of compute function. This adds unnecessary synchronization overhead between parents and their children. A limited form of escaping asyncs without nested finish
was supported in the form of “improperly nested procedures” in the XWS library [6].

3.2 Sequential Call to a Parallel Function

The second feature is that sequential calls to parallel functions are allowed in terminally strict computation language. Parallel functions are those that spawn tasks either directly or indirectly by calling other functions. In Cilk, sequential calls to a parallel functions (known as cilk function) are not allowed. As a result, all functions that may appear on a call path to a spawn operation must be designated as cilk functions and thus must be spawned. This restriction has a significant software engineering impact because it increases the effort involved in converting sequential code to parallel code, and prohibits the insertion of sequential code wrappers for parallel code. In contrast, X10 permits the same function to be invoked sequentially or via an async at different program points.

The program shown in Figure 5 is valid in X10 but cannot be directly translated to Cilk. In Cilk, C() and E() would be cilk functions because they may spawn tasks. Thus C() and E() cannot be called sequentially in function B() and D() respectively.

3.3 Our Approach

3.3.1 Escaping Asyncs

We support escaping asyncs by decoupling finish scopes from function/method boundaries. This decoupling cannot be achieved in Cilk because sync is basically a join of all child tasks and lacks an operation like startFinish to mark the beginning of a scope.

Every finish construct in the source program is translated by the compiler to a pair of runtime API calls startFinish and stopFinish, which mark the enter and exit of the finish scope respectively, and may be nested to an arbitrary depth. We dynamically create a finish node data structure in the runtime for each startFinish call. Various finish nodes are maintained in a tree-like structure with the parent pointer pointing to the node of its Immediately Enclosing Finish (IEF) scope. Apart from the parent pointer, finish nodes keep track of the number of workers that are working under its scope. When a worker is blocked at stopFinish, the continuation after the finish scope is saved in the finish node. This continuation is subsequently picked for execution by the last child in the finish scope (the child which decrements the worker counter to zero).

Figure 3 and 4 show the key pseudo-code that illustrate how the finish and the steal operation are implemented in our work stealing work-first policy. Work-first policy is the policy used by Cilk to schedule the spawned tasks. Section 4 compares the work-first and help-first policy and describes the help-first policy implementation.

Note that two versions of stopFinish are generated for fast and slow clones respectively under work-stealing work-first policy. Under the work-first policy, the worker will stay in the fast clone if no steal happened, and the stopFinishFast is just to set the current finish scope to its parent.

The stopFinishSlow operation incurs the slow path overhead. Apart from resetting the current finish scope to the parent, it checks if there are more workers still actively working for the finish scope. If the counter is not zero, the continuation after the finish scope is suspended on the finish scope. The suspended continuation will be later on picked up by the last worker that terminates in the finish scope for execution. Since stealing can interfere
public class FinishNode {
    private int numLiveChildren = 0;
    private Continuation suspendedContinuation;
    private FinishNode parent;
}

public void startFinish() {
    setCurrentFinishScope(new FinishNode(getCurrentFinishScope()));
}

public void stopFinishFast() {
    setCurrentFinishScope(getCurrentFinishScope().getParent());
}

public boolean stopFinishSlow() {
    FinishNode finish = getCurrentFinishScope();
    FinishNode parent = finish.getParent();
    synchronized(finish) {
        if (finish.getNumLiveChildren() > 0) {
            status = STATUS_SUSPENDED;
            finish.setSuspendedContinuation();
            return false;
        }
    }
    return true;
}

Figure 3: Finish Scope Handling for Work-Stealing with Work-First Policy

public Object steal() {
    int t = this.top.get();
    int b = this.bottom;
    CircularArray a = this.activeArray;
    int size = b - t;
    if (size <= 0) return Empty;
    Object o = a.get(t);
    Frame f = (Frame) o;
    FinishNode finish = f.getFinishScope();
    synchronized (finish) {
        if (!this.top.compareAndSet(t, t+1))
            return Abort;
        finish.incNumLiveChildren();
    }
    Closure c = new Closure(f, finish);
    return c;
}

private Continuation victimTerminating() {
    FinishNode finish = getCurrentFinishScope();
    synchronized (finish) {
        if (finish.decNumLiveChildren() == 0)
            return finish.getSuspendedContinuation();
    }
}

Figure 4: steal and victimTerminating operations for work-first policy
with the numLiveChildren counter check, grabbing a lock on the finish scope is required.

We have implemented a lock-free dynamic circular work-stealing deque proposed by Chase and Lev [5]. Their algorithm comprises of a dynamic cyclic array and its volatile top and bottom fields. The three main operations of the deque are described in details below: (1) The pushBottom operation pushes a frame and increments the bottom if the deque is not full. (2) The steal operation checks if the deque is empty or not. It then atomically tries to update the top field of the victim using a CAS operation. Successful CAS operation implies that no other concurrent process tried to remove the frame, so it returns the frame. In case CAS fails, the stealing operation is aborted. (3) The popBottom returns empty if the deque is already empty. In case the deque has more than one frame, it returns the successfully popped frame. However if the deque becomes empty by the current pop operation, then a CAS is performed on the top to see if it can win the competition with a concurrent steal operation. If CAS, succeeds the object is returned after updating the bottom. Otherwise if CAS fails, it returns empty.

The steal operation described in Figure 4 of the deque (presented below) differs from the original steal operation by adding functionality for updating the the number of outstanding workers for the finish scope of the continuation in the stolen frame. Note that the successful steal and increment of the outstanding children in the finish scope needs to be done atomically. We achieve this by first grabbing a lock on the finish scope and then checking if the steal succeeds or not using the CAS operation. If CAS fails, the steal operation is aborted by releasing the lock on the finish scope. If CAS succeeds, the number of outstanding children are updated. Finally we return a closure for the current frame. Note that in constrast to the help-first policy that will be described in Section 4, the steal operations under the same finish scope is serialized.

3.3.2 Sequential Call to a Parallel Function

When calling a parallel function sequentially, the continuation after the call has to be saved. This is because if stealing occurs in the parallel function, the thief is responsible for executing the continuation after the call. To support this feature, every continuation is extended to contain a stack of continuations up to the previous spawn in the call chain. This stack of continuations is managed at runtime at every sequential and parallel function call, but is only pushed to the deque at points where stealing may actually occur, i.e., the async program points. The thief that steals the frame is then responsible for unwinding the continuation stack. In Cilk, functions are distinguished as parallel or sequential using the cilk keyword. In our implementation, the compiler performs static interprocedural analysis to distinguish sequential and (potentially) parallel functions so as not to burden the programmer.

Consider the example in Figure 5. C1 and C2 label the points where stealing can actually occur. At C1, the frame pushed to the deque contains the continuation stack for C1, L2, L1 in order. The thief that steals the frame is responsible for starting the continuation at C1. Upon returning from function C, the thief will resume the execution at L2, which will return to L1. At each return, the thief will find the continuation required to resume the execution by popping the continuation stack. The continuation stack for C2 contains the continuations up to the previous spawn i.e., C2 and L3.

In terminally strict computations, a naive way to support a sequential call to a parallel
function is to enclose the sequential call in `finish-async`. This approach loses parallelism by disallowing the code after the sequential call to be running in parallel with the task that escapes the callee of the sequential call. In contrast, our approach does not lose any parallelism.

4 Work-first vs. Help-First Scheduling Policies

One of the design principles of Cilk is the work-first policy [7]. In a nutshell, the work-first policy dictates that a worker execute a spawned task (async) and leave the continuation to be stolen by another worker, and the help-first policy dictates that the worker execute the continuation and leaves the async to be stolen. The work-first policy is designed for scenarios in which stealing is a rare event. When the number of steals is low, the worker will mostly execute the fast clone for each spawn, in which case the `sync` operation becomes a no-op.

4.1 Limitations of Work-First Policy

In practice, we observe that the overhead of steals becomes increasingly significant as the number of workers increases. One example is iterative loop parallelism. Figure 6 shows the code structure of a wavefront algorithm implemented in an X10 version of the Java Grande SOR benchmark. It has a sequential outer loop and a parallel inner loop. The work in the inner parallel loop is divided evenly among $P$ workers. Figure 7 shows the speedup of this SOR benchmark relative to its serial version on a 64-thread UltraSPARC T2 Niagara2 machine. If the async’s are executed under the work-first policy, we observe a degradation in performance beyond 16 threads. The following analysis explains why.

Let the total amount of work in one instance of the inner parallel loop be $T$, the amount of time to migrate a task from one worker to another be $t_{\text{steal}}$ and the time to save the continuation for each iteration be $t_{\text{cont}}$. Under the work-first policy, one worker will save and push the continuation onto the local deque and another idle worker will steal it. Therefore, distributing the $P$ chunks among $P$ workers will require $P - 1$ steals and these steals must occur sequentially. The length of the critical path of the inner loop is $(t_{\text{steal}} + t_{\text{cont}}) * (P - 1) + T / P$. Thanks to the THE protocol [7], saving and pushing continuations are non-blocking and contain only thread-local operations, thus $t_{\text{cont}} << t_{\text{steal}}$. As $P$ increases, the actual work for each worker $T / P$ decreases and hence, the total time will be dominated by the time
for (i=1 to SOR_ITERATIONS) {
    finish for (p = 0 to P-1) {
        // parallel for.
        // work evenly divided
        // to P partition
        async {...}
    }
}

Figure 6: SOR code structure depicting iterative loop parallelism

```c
for (i=1 to SOR_ITERATIONS) {
    finish for (p = 0 to P-1) {
        // parallel for.
        // work evenly divided
        // to P partition
        async {...}
    }
}
```

Figure 7: Speedup of SOR

```c
// Work-first Policy
finish {
    push continuation after L1;
    S1; // Worker executes eagerly
    return if frame stolen
    push continuation after L2;
    S2; // Worker executes eagerly
    return if frame stolen;
    S3;
}

// Help-first Policy
finish {
    push task S1 to local deque;
    push task S2 to local deque;
    S3;
}
```

Figure 8: Handling async using work-first and help-first policy

4.2 Help-First Scheduling Policy

As mentioned earlier, our work-stealing runtime also supports a help-first policy for async’s. Under this policy, upon executing an async, the worker will create and push the task onto the deque and proceed to execute the async’s continuation. Once the task is pushed to the deque, it is available to be stolen by other workers and the stealing can be performed in parallel. Let \( t_{\text{task}} \) be the time to create and push the task to the deque. For the same reason mentioned in the last subsection, \( t_{\text{task}} \ll t_{\text{steal}} \). As the stealing overhead is parallelized, the overhead of the steal operation is not a bottleneck any more. When async’s are handled under help-first policy, the performance of the SOR benchmark scales well as shown in the Figure 7.
4.3 Non-blocking Steal Operation

As illustrated in the SOR example, steal operations for the same finish scope are serialized in a work-first policy. However, under the help-first policy, the steal operations can be performed in parallel by using a non-blocking algorithm. To support this, we extend the lock-free dynamic circular work-stealing deque proposed by Chase and Lev [5] and designed a non-blocking implementation for help-first steal operations that maintains a global worker counter for each finish scope. We also maintain distributed local task counters for each worker. The global worker counter counts the number of workers that have checked out while working on the tasks under the finish scope. The local task counter counts the number of tasks that are pushed onto the local deque of the worker, not including the task the worker is currently working on. When a worker finishes all its local tasks, it decrements the global worker counter. If the global worker count reaches 0, it indicates that all tasks under the finish scope have completed.

The pseudo code for the steal operation, the function called when the worker is suspended and the function called when the worker has completed it current task are shown in Appendix 7. Note that, as in other non-blocking protocols, the ordering of the statements is important. For example, in steal(), if the local counter is decremented before the global counter is incremented, another worker completing the task may find that all global worker counter and local counter is 0 and thus unexpectedly pass the finish barrier.

4.4 Discussion

An important theoretical advantage of the work-first policy is that it guarantees that the space used to run the parallel program is bound by a constant factor of the space used in its sequential version. In the help-first policy, since we are creating tasks eagerly, the space bound is not guaranteed. As part of our future work, we plan to explore a hybrid approach that adaptively switches from a help-first policy to a work-first policy depending on the size of the local deques so as to establish a guaranteed space bound.

Another approach to create a task eagerly is work-sharing. The current X10 runtime [2] falls in this category. Unlike work-stealing, work-sharing uses a global task pool. Whenever an async is encountered, the task is created and added to the global pool. All workers get their tasks from the global pool. The disadvantage of the work-sharing approach is that the global pool is likely to become the bottleneck as the number of workers grows. In contrast, our work-stealing scheduler with the help-first policy maintains local deques for workers and adds newly created tasks onto these deques locally.

5 Experimental Results

In this section, we present experimental results to compare the performance of our portable implementation of work-stealing schedulers based on work-first and help-first policies. We also compare the work-stealing implementation with the existing work-sharing X10 runtime system. Section 5.1 summarizes our experimental setup. Section 5.2 compares the performance of the schedulers on a set of benchmarks that includes eight Java Grande Forum (JGF) benchmarks, two NAS Parallel Benchmark (NPB) benchmarks, and the Fibonacci, and Spanning Tree micro-benchmarks. The Java Grande Forum and NAS Parallel benchmarks
Figure 9: Compiler and Runtime Infrastructure for Work-Sharing and Work-Stealing Schedulers

are more representative of iterative parallel algorithms rather than recursive divide-and-conquer algorithms that have been used in past work to evaluate work-stealing schedulers. We use Fibonacci and Spanning Tree as microbenchmarks to compare the performance of the work-first and help-first policies described in Section 4. There are several advantages in using a new language like X10 to evaluate new runtime techniques, but a major challenge is the lack of a large set of benchmarks available for evaluation. To that end, we focused on X10 benchmarks that have been used in past evaluations [4, 2, 10, 11, 6].

5.1 Experimental Setup

The compiler and runtime infrastructure used to obtain the performance results in this paper is summarized in Figure 9. A Polyglot-based front-end is used to parse the input X10 program and produce Java classfiles in which parallel constructs are transformed to X10 runtime calls. To support our work-stealing based runtime, the front-end generated code needs to be modified to produce fast and slow clones for every method [7]. We achieve this in the Code Gen components in Figure 9 by using the Soot infrastructure [12] to transform the class files. The work-sharing runtime in the current X10 system does not need any special code generation for fast and slow clones. In an effort to achieve as close to an “apples-to-apples” comparison as possible, all paths use the same X10 source program at the top of the figure and the same JVM at the bottom.

The work-sharing scheduler shown in the left hand side of Figure 9 represents the current scheduler in the open source X10 implementation. The work-sharing scheduler makes extensive use of the standard java.util.concurrent (JUC) library [9]. Details of the current X10 runtime can be found in [2].

The performance results were obtained on two multi-core SMP machines. The first is a 16-way 1.9 GHz Power5+ SMP with 64 GB main memory; the runs on this machine were performed using IBM’s J9 virtual machine for Java version 1.6.0. The second machine is a 64-thread 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory; the runs on this machine were performed using Sun’s Hotspot VM for Java version 1.6. All results were obtained using the -Xmx2000M -Xms2000M JVM options to limit the heap size to 2GB, thereby
Figure 10: Execution times of Fib(35) for HFP (Help First Policy) and WFP (Work First Policy) with thresholds = 0 and 5 on an UltraSparc T2 system

ensuring that the memory requirement for our experiments was well below the available memory on all three machines. The main program was extended with a three-iteration loop within the same Java process for all JVM runs, and the best of the three times was reported in each case. This approach was chosen to reduce the impact of JIT compilation time in the performance comparisons. We also used the \(-X\text{jit:count=0, optLevel=veryHot, ignoreIEEE}\), \(-\text{PRELOAD\_CLASSES=true}\) and \(-\text{BIND\_THREADS=true}\) options for the Power5+ SMP runs. The \(-X\text{jit:count=0}\) option ensures that each method is JIT-compiled on its first invocation. The \(-\text{PRELOAD\_CLASSES=true}\) option causes all X10 classes referenced by the application to be loaded before the main program is executed. This approach is permitted for X10 (but not for Java in general), and allows for better code to be generated when each method is compiled on its first invocation. The \(-\text{BIND\_THREADS=true}\) option prevents runtime migration of worker threads.

All JGF experiments were run with the largest data size provided for each benchmark except for the Series benchmark for which Size B was used instead of Size C. There are five available sizes for the NPB benchmarks (S, W, A, B, C), and we used the intermediate Size A for all runs in this paper. Since the focus of this paper is on task-level parallelism in an SMP, the \text{NUMBER\_OF\_LOCAL\_PLACES}\ runtime option was set to 1 for all runs.

5.2 Results

\text{Fib} is a classic example of recursive parallel divide and conquer algorithm, and is a useful micro-benchmark for work-stealing schedulers. Figure 10 compares the execution time of the work-first and help-first policies for Fib(35) by varying the number of workers from 1 to 64 on an UltraSparc T2 system. As predicted by past work \((e.g., [7])\), the work-first policy \((\text{WS-WFP})\) significantly outperforms the help-first policy \((\text{WS-HFP})\) for the threshold=0
Figure 11: Spanning Tree Speedups for HFP (Help First Policy) and WFP (Work First Policy) on an UltraSparc T2 system

This result re-establishes the fact that the work-first policy is well-suited for recursive parallel algorithms with abundant parallelism and small numbers of steals. However, if we increase the threshold to 5 (i.e., all calls to Fib() with parameters ≤ 5 are implemented as sequential calls) the gap between WS-HFP and WS-WFP is significantly reduced. The gaps can be further reduced if the Fib function is cloned into separate sequential and parallel versions (which was not done for these threshold results).

In Figure 11, we compare the speedups of the work-first and help-first policies on the spanning tree microbenchmark for irregular large graphs. These speedups were obtained by normalizing with respect to a single-thread execution of the corresponding policy. Spanning tree is an important kernel computation in many graph algorithms. In general, large-scale graph algorithms are challenging to solve in parallel due to their irregular and combinatorial nature. There are three observations that can be drawn from Figure 11. First, WS-HFP outperforms WS-WFP for a given input graph (62.5K nodes). Second, as also observed in [6], WS-WFP is unable to run on large graphs due to stack size limitations. In our experiments, we were unable to get WS-WFP results for a graph larger than 62.5K nodes. Third, not only can WS-HFP process larger graphs (with 9M nodes in Figure 11), but it also achieves better scalability when doing so.

The Fib and Spanning Tree microbenchmarks illustrated cases in which WS-WFP performs better than WS-HFP and vice versa, though the gap can be narrowed for Fib by using a small threshold value for parallelism. We now turn our attention to the JGF and NPB benchmarks, with results presented for two SMP machines in Figures 12 and 13. For convenience, the speedup measurements in both figures are normalized with respect to a single-processor execution of the X10 Work-Sharing scheduler [2]. In previous work [10], we showed that the Work-Sharing scheduler can deliver equal or better performance than the
Figure 12: Comparison of Work-Sharing, HFP (Help First Policy) and WFP (Work First Policy) on a 16-way Power5+ SMP

Figure 13: Comparison of Work-Sharing, HFP (Help First Policy) and WFP (Work First Policy) on a 64-thread UltraSparc T2 SMP
multithreaded Java version for these benchmarks. We observe that on average, Work-Stealing
schedulers outperform the Work-Sharing scheduler and the help-first policy performs better
than work-first policy for Work-Stealing. This is because the code for JGF and NPB use
iterative loop based parallelism. For SOR, CG, and LUFa ct benchmarks, the help-first policy
outperforms work-first policy by a large margin.

To summarize, we observe that both work-first and help-first policy based Work-Stealing
schedulers outperform the Work-Sharing scheduler. For most JGF and NPB benchmarks, the
Work-Stealing scheduler using a help-first policy performed better than the Work-Stealing
scheduler with a work-first policy. This is because these benchmarks are based on iterative
loop parallelism. We have also shown two extreme cases for comparing work-first and help-
first policies. In the Fib microbenchmark, the work-first policy wins (though a small
threshold can narrow the gap) where as in the Spanning-tree microbenchmark the help-first
policy wins.

6 Related Work

Work-stealing schedulers have a long history that includes lazy task creation [8] and the
theoretical and implementation results from the MIT Cilk project. Blumofe et al. defined
the fully strict computation model and proposed a randomized work stealing scheduler with
provable time and space bounds [3]. An implementation of this algorithm with compiler
support for Cilk was presented in [7]. In earlier work [1], we proved that terminally strict
parallel programs can be scheduled with a work-first policy so as to achieve the same time
and space bounds as fully strict programs. To the best of our knowledge, the work presented
in this paper is the first work stealing implementation for a terminally strict parallel language
like X10 with both compiler and runtime support.

The open source X10 v1.5 implementation [2] includes a work-sharing runtime scheduler
based on the Java 5 Concurrency Utilities. This approach implements X10 activities as
runnable tasks that are executed by a fixed number of Java threads in a ThreadPoolExecutor,
compared to creating a separate Java thread for each X10 activity. As shown in our
experimental results, the work-stealing based scheduler (for both work-first and help-first
policies) presented in this paper performs better than this work-sharing implementation.

The X10 Work Stealing framework (XWS) is a recently released library [6] that supports
help-first scheduling for a subset of terminally strict X10 programs in which sequential and
async calls to the same function and nesting of finish constructs are not permitted. The
single-level-finish restriction leads to a control flow structure of alternating sequential and
parallel regions as in OpenMP parallel regions, and enables the use of a simple and efficient
global termination detection algorithm to implement each finish construct in the sequence.
The library interface requires the user to provide code for saving and restoring local variables
in the absence of compiler support. With these restrictions and an additional optimization
for adaptive batching of tasks, the results in [6] show scalable speedup for solving large
irregular graph problems. In contrast, our approach provides a language interface with
compiler support for general nested finish-async parallelism, and our runtime system supports
both work-first and help-first policies. In addition, we have implemented nonblocking steal
operations for help-first scheduling that differs from the algorithm in [6]. An interesting
direction for future research is to extend our compiler support to generate calls to the XWS
library so as to enable performance comparisons for the same source code, and explore integration of the scheduling algorithms presented in this paper with the adaptive batching optimization from [6].

7 Conclusions and Future Work

In this paper, we addressed the problem of efficient and scalable implementation of X10’s terminally strict async-finish task parallelism, which is more general than Cilk’s fully strict spawn-sync parallelism. We introduced work-stealing schedulers with work-first and help-first policies for async-finish task parallelism, and compared it with the work-sharing scheduler that we previously implemented for X10. Performance results on two different multicore SMP platforms show that the work-stealing scheduler with either policy performs better than the work-sharing scheduler for terminally strict computations. They also shed insight on scenarios in which work-stealing scheduler with work-first policy outperforms the one with help-first policy and vice-versa.

There are many interesting directions for future research. We see opportunities for optimizing the book-keeping code necessary for maintaining continuation frames. It is also important to extend the work stealing algorithm presented in this paper to be locality-conscious so that it can support the affinity directives embodied in X10 places [4], and also support broader classes of X10 programs with coordination and synchronization mechanisms that go beyond async-finish, such as phasers [11].

References


Appendix: Pseudo Code for Nonblocking Steals in Help-First Scheduling Policy

```java
public Object help_first_deque_steal () {
    task = pop the task frame from the victim's deque or abort;
    finish = task's finish scope;
    finish.global_worker_counter++;
    finish.local_task_counter[victimID]--;
    return task;
}

private Object JobCompleting() {
    finish = completed task's finish scope;
    task = get some local task under the finish;
    if (task != null) {
        finish.local_task_counter[this.workerid]--;
        return task;
    }
    // this worker has completed all its task under the finish
    worker.global_worker_counter--;
    if (finish.local_task_counter[this.workerID] == 0) {
        if (worker.global_worker_counter == 0) {
            // all tasks under the finish has been completed
            // workers now competing for the continuation suspended on the finish
            // only one worker will execute the continuation others goStealing()
        }
        ....
    }
}

private task goStealing() {
    task = get local task frame from local deque;
    if (task != null) {
        task.finish.local_task_counter[this.workerID]--;
        return task;
    }
    scan other workers' deque until a task is successfully stolen
}

private pushJob(task) {
    finish = current finish scope;
    finish.local_task_counter[this.workerID]++;
    this.deque.pushBottom(task);
}
```