SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler

Yi Guo Jisheng Zhao Vincent Cave Vivek Sarkar
Department of Computer Science, Rice University, Houston, Texas 77005
Email: {yguo,jz10,vc8,vsarkar}@rice.edu

***Abstract*—This paper introduces SLAW, a Scalable Locality-aware Adaptive Work-stealing scheduler. The SLAW scheduler is designed to address two common limitations in current *work-stealing* schedulers: use of a fixed task scheduling policy and locality-obliviousness due to randomized stealing.**

Past work has demonstrated the pros and cons of using fixed scheduling policies, such as *work-first* and *help-first*, in different cases without a clear win for one policy over the other. The SLAW scheduler addresses this limitation by supporting both work-first and help-first policies simultaneously. It does so by using an *adaptive* approach that selects a scheduling policy on a per-task basis at runtime. The SLAW scheduler also establishes bounds on the stack and heap space needed to store tasks. The experimental results for the benchmarks studied in this paper show that SLAW's adaptive scheduler achieves 0.98× to 9.2× speedup over the help-first scheduler and 0.97× to 4.5× speedup over the work-first scheduler for 64-thread executions, thereby establishing the robustness of using an adaptive approach instead of a fixed policy. In contrast, the help-first policy is 9.2× slower than work-first in the worst case for a fixed help-first policy, and the work-first policy is 3.7× slower than help-first in the worst case for a fixed work-first policy. Further, for large irregular recursive parallel computations, the adaptive scheduler runs with bounded stack usage and achieves performance (and supports data sizes) that cannot be delivered by the use of any single fixed policy.

It is also known that work-stealing schedulers can be cache-unfriendly for some applications due to randomized stealing. The SLAW scheduler is designed for programming models where locality hints are provided to the runtime by the programmer or compiler, and achieves *locality-awareness* by grouping workers into *places*. Locality awareness can lead to improved performance by increasing temporal data reuse within a worker and among workers in the same place. Our experimental results show that locality-aware scheduling can achieve up to 2.6× speedup over locality-oblivious scheduling, for the benchmarks studied in this paper.

***Index Terms*—Task Parallelism; Work-stealing; Adaptive Scheduling; Locality-aware Scheduling.**

I. Introduction

As the computer industry is entering an era of mainstream parallel processing, 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 [1], Fortress [2], X10 [3]) all identified dynamic lightweight task parallelism as one of the prerequisites for success. Dynamic task parallelism is also being introduced in existing programming models for shared-memory parallelism such as OpenMP 3.0. In dynamic task parallelism, computations are dynamically created and the runtime scheduler is responsible for scheduling the computations across the cores. An efficient and scalable runtime scheduler is critical to achieving good performance for dynamic task parallelism.

Work-stealing runtime schedulers are increasing in popularity as scheduling algorithms for dynamic task parallelism, as evidenced by past work e.g., Cilk [4], Cilk++ [5], Intel Threading Building Blocks (TBB) [6], Java's Fork-join Framework [7], Microsoft Task Parallel Library [8], and StackThreads/MP [9]. A work-stealing scheduler employs a fixed number of threads called workers. Each worker has a local deque to store tasks. When a worker is suspended at a synchronization point and there is no local task, the worker will try to steal the task from the top of other busy workers' deques. For a busy worker, tasks are pushed and popped from the bottom of the deque and these operations are synchronization-free.

This paper introduces SLAW, a Scalable Locality-aware Adaptive Work-stealing scheduler. SLAW identifies and addresses two important issues that limit the scalability in current work-stealing schedulers: use of a fixed task scheduling policy, and locality-obliviousness due to randomized stealing.

Scheduling Policies: Task scheduling is critical for a work-stealing scheduler. Work-first and help-first are two commonly used task scheduling policy used when spawning a task [10]. Under the work-first policy, the worker will execute the spawned task eagerly and leave the continuation to be stolen. Under the help-first policy, the worker will make the spawned task available for stealing and itself will continue execution on the parent task. Work-first and help-first policies are usually implemented with different stack and memory bounds and also exhibit performance limitations in different scenarios. The work-first policy is good for scenarios when stealing is rare. It has a provable memory bound but its implementation may overflow the stack for large irregular computations. The help-first policy is good for scenarios when stealing is frequent. It can be implemented with low stack usage but is not space-efficient in general. These differences make it hard to determine a priori which policy should be used.

SLAW's adaptive scheduler is designed to achieve the best of both policies, while ensuring bounded stack usage. The bound is referred to as the *stack threshold*, and is set as a parameter by the programmer. SLAW's adaptive scheduler is provably space-efficient if the serial depth-first execution of the computation does not exceed the stack threshold. If it exceeds the threshold, the adaptive scheduler will move the stack pressure to the heap, since the heap is usually much larger than the stack. To obtain good scalability in both scenarios when steals are rare vs. frequent, SLAW has a heuristic to adjust scheduling policies according to the stealing rate. The adaptiveness in scheduling policy and bounded stack usage make SLAW capable of handling large irregular computations.

The experimental results from a variety of benchmarks show that: SLAW's adaptive scheduler achieves 0.98× - 9.2× speedup over the help-first scheduler and 0.97×-4.5× speedup over the work-first scheduler for 64-thread executions. In contrast, the help-first policy is 9.2× slower than work-first in the worst case for a fixed help-first policy, and the work-first policy is 3.7× slower than help-first in the worst case for a fixed work-first policy. Further, for large irregular recursive parallel computations, the adaptive scheduler runs with bounded stack usage and achieves performance (and supports data sizes) that cannot be delivered by the use of any single fixed policy.

Locality-aware scheduling: Work-stealing has also been known to be cache-unfriendly for some applications due to randomized stealing [11]. For tasks that share the same memory footprints, randomized locality-oblivious work-stealing schedulers do nothing to ensure scheduling of these tasks on workers that share a cache. This significantly limits the scalability for some memory-bandwidth bounded applications on machines that have separate caches.

SLAW is designed for programming models in which locality hints are provided by the programmer or the compiler, such as X10's places [3] and Chapel's locales [1]. In SLAW, workers are grouped by *places* and our current implementation restricts stealing to only occur within places. Our experimental results show that locality-aware scheduling can achieve up to 2.6× speedup over locality-oblivious scheduling by increasing temporal data reuse. We believe that the performance benefits of locality-aware scheduling will continue to increase in future systems.

Organization of the paper: Section II describes the context for the SLAW scheduler, including work-first and help-first scheduling polices and the Habanero-Java (HJ) language [12]. Section III describes the SLAW adaptive scheduling algorithm, and Section IV describes extensions to SLAW for locality-awareness. Sections V to VII contain our experimental results, related work discussion and conclusions respectively.

II. Background: SLAW context

A. Work-stealing Scheduling Policies

Work-first and help-first are two commonly used task scheduling policies used when spawning a task. Under the work-first policy, the worker will execute the spawned task eagerly and leave the continuation to be stolen. Under the help-first policy, the worker will make the spawned task available for stealing and itself will continue execution on the parent task.

Both work-first and help-first policies will yield coarse-grain tasks to the thief because the stealing is from the root of the spawn tree towards the leaves.

The work-first policy and help-first policy have different stack and memory requirements and have performance issues in complementary scenarios.

1) Performance Issues: Context switches represent a major source of overhead in a work-stealing scheduler [13]. A context switch is performed when the worker cannot continue normal sequential execution. This happens in two situations: a) When the current task terminates, and the worker cannot resume execution of its parent task. This can happen when the continuation of the parent task is stolen or the current task is not spawned under the work-first policy. b) When the current task reaches a synchronization point, and there are still unfinished descendant tasks. In both situations, the worker will return to the scheduler to request a new task. In the second situation, the task is suspended and the context switch will also save the activation frame of the current task to the heap, so that it can be resumed later. Compared to normal sequential execution, the overhead of context switches primarily arises from bookkeeping overhead in the scheduler and from cache misses.

Under the work-first policy, if there is only 1 worker thread, it will execute all tasks in the same order as the equivalent sequential program, thereby requiring no context switch at task synchronization points. In general, the work-first policy works well for situations where steals are infrequent [10]. However, for applications whose spawn trees are shallow and wide, the steals can become frequent because there each steal yields a limited amount of work. Consider a scenario in which one busy worker creates N tasks consecutively and the other $N-1$ workers are idle and looking for tasks. In order to distribute N tasks to the $N-1$ idle workers under the work-first policy, continuations must be passed from the victim worker to the thief through stealing and there will be $N-1$ such steals of continuations. More importantly, these steals must be serialized thereby limiting scalability and contributing to a large steal overhead.

On the other hand, under the help-first policy, the steals are performed more efficiently. The steal contains two parts: task retrieval from the victim and the following context switch to execute the task. Under the help-first policy, although the tasks have to be popped from the victim's queue in order, the context switches can be done in parallel. However, if steals are rare, using the help-first policy may increase the 1-thread execution time relative to sequential execution, because of the context switches that can occur at every task synchronization point.

In general, the work-first policy works better for recursive parallelism in which the task spawn tree is deep and the help-first policy works better for flat parallelism in which the task spawn tree is shallow and wide. In the extreme case, our experimental results show that, for 64 threads, the help-first policy can be 9.2× slower than the work-first policy on Fib micro-benchmark, and the work-first policy can be

```
class V  {
  V [] neighbors;
  V parent;
  V (int i) {super(i); }
  boolean tryLabeling(V n) {
      isolated { if (parent == null)
                  parent = n; }
      return parent == n;
  }
  void compute() {
     for (int i=0; i<neighbors.length; i++) {
        V e = neighbors[i];
        if (e.tryLabeling(this))
           async e.compute();
     }
  }
  void DFS() {
     parent = this;
     finish async compute();
  }}
```

Fig. 1. HJ code for a parallel-DFS (PDFS) spanning tree algorithm

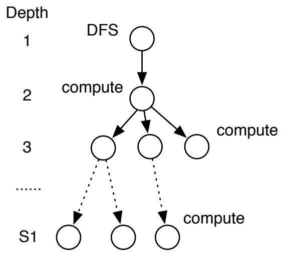


Fig. 2. A possible spawn tree for parallel DFS program (PDFS) in Figure 1. Solid edge is spawn edge and dash edge indicates a chain of spawn edges. For PDFS, the depth of the spawn tree (S_1) is only bounded by the size of graph in the worse case.

3.7× slower than the help-first policy for the Fork-Join micro-benchmark.

2) Stack and Memory Bounds: The work-first and help-first policies face different challenges with respect to stack and memory bounds.

It is well known that work-stealing schedulers that use the work-first policy are provably space-efficient [14]. If the serial depth-first execution of parallel application uses S_1 memory, the memory usage of the P-processor execution is bounded by S_1P. However, because the work-first policy is usually implemented as a sequential call to the spawn task to minimize the 1-thread execution time, the stack usage of the work-first scheduler is the same as the depth-first scheduler. This becomes a significant disadvantage for computations, such as the Parallel Depth First Search (PDFS) benchmark shown in Figure 1, where *the depth of the task spawn tree is proportional to the program's data size*. Work-stealing schedulers using the work-first policy will terminate prematurely due to stack overflow for large graphs because of the stack limit in current environment. Although it is possible to instead allocate stack frames in the heap, these solutions are mostly OS and architecture dependent, and may incur overhead compared to the simple function call.

Figure 1 shows a parallel recursive program to build a spanning tree on a graph (the semantics of the async, finish, and isolated constructs are explained in Section II-B) and Figure 2 shows the spawn tree of the application. Although the program expresses the parallelism correctly and elegantly, such recursive programs are not guaranteed to run successfully on many current work-stealing runtimes that use the work-first policy, because the sequential version of PDFS may overflow the stack for large applications.

On the other hand, under the help-first policy, the worker will not execute the spawned task eagerly; instead, it will continue executing the parent task until it is suspended or it terminates. When the task is suspended, the scheduler will switch context to a new task. The whole process is implemented by throwing an exception that be caught by the scheduler, which then calls the run method of the new task. The unnecessary stack pressure of ancestor tasks are released.

Despite their stack bound, work-scheduling schedulers that use the help-first policy are not provably space-efficient. In the worst case, if the parent task spawns an unbounded number of child tasks, all these child tasks will be saved to a deque on the heap, which may lead to heap overflow. However, the serial depth-first execution of the same program runs with bounded memory.

B. Habanero-Java Language (HJ)

This section describes the Habanero-Java (HJ) language currently supported by the SLAW scheduler, and used for the experimental results. HJ is a Java-based dynamic task parallel language derived from X10 v1.5 [3].

The `async` construct is used to create (fork) a new asynchronous task. The statement, `async` ⟨*stmt*⟩, causes the parent task to create a new child task that executes ⟨*stmt*⟩. The parent task and the child task can run in parallel. An `async` statement can optionally include a *schedule* clause and/or a *place* clause as follows: `async` *[schedule(wf|hf|dyn)] [place(p)]* ⟨*stmt*⟩.

The *schedule* clause can take one of three possible values: *wf*, *hf* or *dyn*, which correspond to the *work-first*, *help-first* and *dynamic*(default) scheduling policies respectively. For *dyn*, the scheduling policy is selected adaptively on a per-task basis at runtime as described later in Section III. The *place* clause takes a place expression p, and serves as an affinity hint to constrain the execution of the *async* to a designated subset of workers as described below.

The statement *finish* ⟨*stmt*⟩ causes the parent task to execute ⟨*stmt*⟩ and then wait until all sub-tasks created within ⟨*stmt*⟩ have terminated (including transitively spawned tasks). The main function is enclosed by an implicit finish scope.

A *place* is a virtual partition of tasks. Each `async` task executes in a designated place for its lifetime, though it can switch among multiple workers in the same place. A task obtains a reference to the current place by evaluating the constant *here*, which is also the default value for the place clause in the async statement. The number of places is fixed when the program is launched; there is no construct to create a new place. This is consistent with current programming models, such as MPI, UPC, and OpenMP, that require the number of processes/threads to be specified when an application is launched. For the purpose of this paper, a place is implemented as a set of worker threads in a work-stealing scheduler such that no stealing is permitted across places. Note that places are multi-threaded in general and stealing can occur within a place. The mapping of worker threads to places is specified when the program is launched, and is also referred to as the *deployment*. This mapping includes a binding of worker threads to processor cores. Unlike X10, the HJ subset used in this paper only associates places with tasks(`async`'s) and not with data. Data locality is instead achieved indirectly by assigning `async`'s with data affinity to execute in the same place.

The *isolated* construct is HJ's renaming of X10's *atomic* construct. As stated in [3], an atomic block in X10 is intended to be "executed by a task as if in a single step during which all other concurrent tasks in the same place are suspended". This definition implies *strong atomicity* semantics for the atomic construct. However, all X10 implementations that we are aware of are lock-based and do not enforce any mutual exclusion guarantees between computations within and outside an atomic block. As advocated in [15], we use the *isolated* keyword instead of *atomic* to make explicit the fact that the construct supports weak isolation rather than strong atomicity.

III. Scalable Adaptive Scheduling

This section presents SLAW's adaptive scheduling algorithm for a single place. The extension to multiple places is summarized in Section IV. We first informally give an overview of the adaptive scheduler, followed by the scheduling model and taxonomy assumed in this paper. We then present our adaptive scheduling algorithm, along with theoretical worst-case space bounds. This section concludes with a summary of the compiler support and the runtime implementation for the adaptive scheduling algorithm.

A. Overview of the Adaptive Scheduler

There are two major scalability concerns when designing the adaptive scheduler: (1) establishing space bounds which include stack space for worker threads as well as the total memory space; and (2) selection of help-first and work-first policy in different scenarios for better scalability.

As described in Section II-A2, a stack-based implementation of the work-first policy increases stack pressure, where as the help-first policy can be used to reduce stack pressure (but at the expense of additional context switches). Let us assume that S is the space limit (or *threshold*) for a worker's stack. If the input program has a spawn tree depth greater than S, then it is necessary at some point to use the help-first policy to ensure that a worker's stack space does not exceed threshold S. This decision is presented as *stack condition* in the spawn rule for Algorithm 1 discussed later.

Besides the stack bound, we also consider the total memory bound. The total memory bound is determined by the memory usage of both started and fresh tasks. Started tasks are those that have been executed by some processor; fresh tasks have been spawned but never executed. When only spawning under work-first policy, there will be no fresh tasks and the total memory bound of started tasks has been established by past research on work-stealing schedulers [14], [16]. However, under the help-first policy, all child tasks will be created as fresh tasks and saved on the heap. In order to provide a total memory guarantee for the adaptive work-stealing scheduler: the scheduler must switch to the work-first policy when the number of fresh tasks exceeds a threshold; this ensures that the total memory used by fresh tasks are bounded. The threshold is called the *fresh task threshold* denoted as F. This decision is presented as *fresh task condition* in the spawn rule for Algorithm 1.

These two conditions are enough to establish the stack and total memory bounds for the adaptive scheduling algorithm. One thing that is important to notice is that the adaptive scheduler treats stack bound as a hard bound and gives the stack condition higher priority than the fresh task condition. When the stack threshold is reached, help-first policy will always be used to avoid stack overflow regardless of the number of fresh tasks created.

SLAW employs a runtime heuristic to select the policy if neither of these two conditions is met. This heuristic is not required to establish the worst-case stack and memory space bound, but is designed to achieve better scalability and performance in practice. For this reason, the heuristic is described below but is not presented in the algorithm.

Before describing the heuristic, we first discuss two techniques used to reduce the overhead of adaptation and evaluation of the task spawning policy. First, each worker maintains its own spawning policy and the heuristic used to evaluate the spawning policy consists only of thread-local operations. We show in the Section V-B1 that the overhead is lower than 5%. Second, SLAW does not re-evaluate the spawning policy at every spawning point. Instead, it starts with the help-first policy at the beginning and re-evaluate the spawning policy periodically at an interval for every INT spawned tasks. The reason that it starts with the help-first policy is because steals are usually frequent at the beginning of the application, and help-first policy can raise parallelism when stealing is frequent. Evaluating the policy periodically further amortized the overhead.

The heuristic used by SLAW is based on a simple estimation on the likelihood of the new spawned task being stolen. It computes the number of tasks that were stolen from the worker during the last interval. If the number of steals is greater than INT, this implies the steal rate is higher than the task creation rate. The scheduler will use the help-first policy for the new task in the next interval to increase the rate of distributing tasks to other workers. Otherwise, the scheduler assumes the new task will not be stolen and thus uses work-first policy for the next interval to reduce the overhead of context switches.

In summary, the thresholds S and F for the *stack condition* and the *fresh task* condition are used to bound the algorithm's stack and total space requirements respectively. The third parameter INT is used to control the policy re-evaluation interval in SLAW to reduce overhead. Later in Section V-B, we present experimental results on the sensitivity of performance to these parameters and also discuss selection of their default values.

B. Scheduling Model

As in past work [14], [16], we model the execution of a multi-threaded program as a dag of dynamic instructions connected by dependency edges. The instructions are connected by *continue edges* into tasks, and tasks are connected into a *spawn tree* using spawn edges. Given a task γ, we use $ST_{spawn}(\gamma)$ to denote the set of tasks in γ's subtree (including γ) in the spawn tree of the dag and $PR_{spawn}(\gamma)$ to denote γ's spawn tree parent. Tasks in the dag are also connected by *sync edges* for task synchronization. If there is no sync edge from the last instruction of the task, an implicit sync edge to the root task is assumed (corresponding to the implicit finish in the main program). In a strict computation, sync edges can only go from a task to its ancestor in the spawn tree. In a fully-strict computation, sync edges can only go from a task to spawn tree parent. In terminally-strict computations created by async-finish constructs, there is only one sync edge starting from the last instruction of a task, but it can go to any spawn tree ancestor [16].

The *sync tree* for terminally-strict computations can be defined as follows. Each node in the sync tree corresponds to a task. There is an edge from task γ_a to γ_b in the sync tree if there is a *sync edge* in the dag from γ_b to γ_a. Given a task γ, we use $ST_{sync}(\gamma)$ to denote the set of tasks in γ's subtree (including γ) in the sync tree and use $PR_{sync}(\gamma)$ to denote γ's parent in the sync tree.

```
     //T1
S1:  finish {
        //T1
S2:     async {
           //T2
S3:        async T3;
S4:        finish {
              //T2
S5:           async T4
S6:        }
S7:     }
S8:     finish {
          //T1
S9:       async T5
S10:  }}
```

Fig. 3. HJ Code

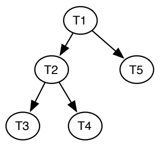


Fig. 4. Spawn Tree for the HJ Code Snippet in Figure 3

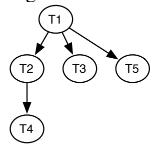


Fig. 5. Sync Tree for the HJ Code Snippet in Figure 3

In fully-strict computations, like those created by Cilk's spawn-sync constructs, the spawn tree is same as the sync tree. This is not the case for terminally-strict computations created by async-finish constructs. As an example, Figures 4 and 5 show the spawn tree and the sync tree respectively for the HJ code in Figure 3. Notice that T_2 spawns T_3 at S_3, so T_2 is T_3's parent in the spawn tree. However, both T_2 and T_3 have the same sync tree parent, T_1.

C. Taxonomy

We use the notation $G(V, E, S_1)$ to represent a computation *dag* with set of tasks V tasks, set of edges E, and stack depth S_1 which is defined as the total memory space required for serial depth-first execution.

A P-processor schedule X of a dag is defined as a sequence of steps, where each step consists of at most P instructions, one for each processor. For a given dag, $X(t, p)$ denotes the task of the instruction executed by processor p at step t.

1) **Task terminates:** a task γ terminates after step t, if its last instruction is executed at step t by processor p. We also say processor p finishes/terminates γ at step t.
2) **Suspended task:** A task γ is suspended before step t if γ cannot be run at step t due to dependency. We say processor p suspends a task γ at step t if $\gamma = X(t, p)$ and γ is suspended before step t+1.
3) **Task ready:** A task is ready if it is not suspended. A task *becomes* ready before step t if it is suspended before step t-1 and it is ready before step t.
4) **Fresh task:** A task γ is said to be *fresh* before step t if it is has been spawned by processor p before step t, but is never executed by any processor. Processor p is called γ's owner. Fresh task is saved on the heap. We assume the size of any fresh task is $O(1)$.
5) **Preempted task:** Task γ is said to be *preempted* by processor p *at* step t if $\gamma = X(t-1, p)$ and γ is not suspended before step t and $\gamma \neq X(t, p)$. No task is preempted at step 0.
   Task γ is *preempted and owned* by processor p before step t if:
   a) $\gamma \neq X(t-1, q)$ for any processor q
   b) γ is either preempted and owned by p before step t-1 or preempted by processor p at step t-1 .

   In the adaptive algorithm, a task is preempted only when it performs a work-first spawn (Action 1 in Algorithm 1).
6) **Making progress:** A task γ is making progress due to processor p at step t if $X(t, p) \in ST_{spawn}(\gamma)$. A task is said to be making progress at step t if it is making progress due to any processor at step t.
7) **Progressive schedule:** We say a schedule is progressive if all non-fresh tasks are making progress for every step.

If all non-fresh tasks are removed from the spawn tree, a *progressive* schedule has the *busy leaf property* [14] and has the following memory bound.

Theorem 3.1: For any P-processor progressive schedule X for a dag $G(V, E, S_1)$, for any step, the memory usage of all non-fresh tasks is bounded by S_1P.

Theorem 3.1 provides the bound for non-fresh tasks in a progressive schedule [1]. If the memory usage of fresh tasks is also bounded, the total memory space will be bounded. In the following subsection, we present our adaptive scheduling algorithm which is progressive and has a bound for non-fresh tasks if the stack threshold is not exceeded.

[1]Due to space limit, we omit the proof of all theorem.

Subroutine 1 Idle Subroutine for Processor p Before Step t

```
1) If $p$ owns any fresh task or preempted task, remove one. If multiple
   such tasks exist, the following tie breaker is used:
     a) Return one that is the deepest in sync tree.
     b) Return preempted tasks before fresh task
     c) Return one that is the deepest in spawn tree
   If success, goto 4.
2) In this case, processor $p$ does not own any task. It will go stealing.
   It will attempt to remove task $\gamma$ in the pool that meets one of the
   following stealing restrictions.
     (a) if $\gamma$ is fresh and created by some processor $q$, $\gamma$ is the one that
         was created earliest among all fresh tasks created by $q$.
     (b) if $\gamma$ is preempted and owned by some processor $q$, $\gamma$ is the
         one that was preempted earliest among all tasks preempted and
         owned by $q$.
   If success, goto 4.
3) Processor $p$ remains idle. Goto 1.
4) Processor $p$ returns the task for execution at step $t$.
```

D. Scheduling Algorithm

We use P-ADP(S, F) to denote a P-processor adaptive schedule that can be generated by the adaptive work-stealing algorithm shown in Algorithm 1. As mentioned earlier, S and F denote the stack threshold and fresh-task threshold respectively.

To abstract the runtime call stack, some tasks are flagged *on-stack-p* where p is a processor id. The activation frame of those tasks flagged on-stack-p is considered to be on processor p's runtime call stack. The algorithm always flags a task γ as *on-stack-p* if processor p starts executing γ. This flag will not be cleared when γ spawns a new task under the work-first policy, since work-first task spawn is implemented as a sequential call in SLAW. However, when a processor p does a context switch to start executing a fresh-task, or resume a suspended task, all *on-stack-p* flags for that particular processor p are cleared.

Actions 1-5 in the algorithm and the idle subroutine, guarantee that all adaptive schedules execute tasks in a depth-first order when a task is suspended or terminates. However, when many tasks have the same depth, the tie breakers in the idle subroutine are important to ensure progress-ness of the schedule, which leads the space bound of the algorithm.

Theorem 3.2: All adaptive schedules are progressive.

E. Theoretical Space Bound

Given a dag $G(V, E, S_1)$, the following theorem presents the space bound for any P-processor adaptive schedule.

Theorem 3.3: If $S_1 <= S$, the memory space of any $P - ADP(S, F)$ schedule is bounded by $S_1P + O(FP)$. If $S_1 > S$, the memory space of any $P - ADP(S, F)$ schedule is bounded by $S_1P + O(V)$.

Theorem 3.3 establishes the memory space bound for the adaptive schedule. If $S_1 <= S$, this is the case that work-first can run successfully without exceeding the stack bound. The work-first work-stealing scheduler's memory bound in this case is S_1P. The memory space of the adaptive scheduler is bounded by $S_1P + O(FP)$. If $S_1 > S$, this is the case that work-first will overflow the stack. The adaptive schedule will never overflow the stack and the memory space is bounded by $S_1P + O(V)$.

Algorithm 1 Adaptive Work-Stealing Algorithm - ADP(S,F)

```
1) Environment: There are $P$ processors and a shared task pool where
   every processor can remove and put tasks. All operations are assumed
   to be atomic.
   The algorithm proceeds step by step. Note that both the spawn tree
   and the sync tree are unfolded online as the algorithm progresses.
   When a processor $p$ is idle before step $t$, it will call the idle subroutine
   in Subroutine 1 to attempt to remove and execute task for the step $t$.
2) Step 0: At step 0, one processor will start executing the root task. All
   other processors are idle.
3) Step $t$+1: For step $t$, the task will decide the task to execute for step
   $t$+1. If processor $p$ executes task $\gamma_a$ at step $t$, it will execute the next
   instruction in task $\gamma_a$ unless $\gamma_a$ spawns, suspended or terminates. In
   these cases, the following rules a)-c) are followed respectively:
     a) Spawn: Let $\gamma_a$ spawns $\gamma_b$. Processor $p$ will use the following
        rule to decide the spawn policy:
          i) If the space of the activation frames of all tasks marked on-
             stack-p $\geq S$, use help-first (stack condition);
         ii) Otherwise if the number of fresh tasks currently owned by
             $p$ before $t$ is $\geq F$, use work-first (fresh-task condition);
        iii) Otherwise free to use any heuristic. See Section III-A for
             SLAW's heuristic.
        • Action 1: If the spawn is under work-first policy, return $\gamma_a$
          to the pool and execute $\gamma_b$ for step $t$+1.
        • Action 2: If the spawn is under help-first policy, put $\gamma_b$ to
          the pool and continue to execute next instruction of $\gamma_a$ for
          step $t$+1.
     b) Suspended: If the task $\gamma_a$ is suspended, processor $p$ will return
        $\gamma_a$ to the pool, do a context switch and clears all on-stack-p
        flags on tasks. Then
        • Action 3: processor $p$ will remove any fresh task it created
          in $ST_{sync}(\gamma_a)$.
        If not success, $p$ becomes idle.
     c) Terminates: If the task $\gamma_a$ terminates and if $\gamma_a$ is the root task,
        then the schedule ends. Otherwise, let $T_1$ be $PR_{spawn}(\gamma_a)$ and
        $T_n$ be $PR_{sync}(\gamma_a)$. Processor $p$ will:
        • Action 4: If $T_1$ is preempted and owned by $p$, remove $T_1$
          and execute it for step $t$.
          If Action 4 is not taken, the processor $p$ will do a context
          switch and clears all on-stack-p flags on tasks. Then it will
        • Action 5: Check if $T_n$ becomes ready before step $t$+1 If yes,
          processor $p$ will attempt to remove $T_n$ from the pool and
          execute $T_n$ for step $t + 1$.
          If the Action 5 is not taken, processor $p$ will become idle.
```

It is important to notice that the S_1 is related to the input data size [17] because it is the space requirement of serial depth-first execution. However, F is a preset parameter and is not related to the input data size. The constant of the $O(FP)$ is the size of the holder of the fresh task on the heap, which is usually small. In SLAW, the task holder contains only the value of input parameters to the task function and a few bookkeeping fields.

F. Runtime Implementation and Compiler Support

Many work-stealing schedulers use the deque data structure to store tasks. Deque is a double-ended queue. The steal operations are performed at the top-end of the deque; only the owner of the deque will push and pop tasks at the bottom-end of the deque. SLAW's deque implementation is based on the dynamic circular deque implementation described in [18].

SLAW implements the adaptive scheduling algorithm using two deques per worker: one for preempted tasks owned by the worker; the other for fresh tasks created by the worker. When a task is preempted at a work-first spawn, it is pushed to the bottom-end of the preempted task deque. When a fresh task is created using the help-first policy, it is pushed to the bottom-end of the fresh task deque. When a thief is stealing, it steals from the top-end of other workers' deques. For computations created by async-finish constructs, this implementation guarantees that tasks in the preempted task deque from top-end to bottom-end are from shallow to deep in the spawn tree, (also shallow to deep in the sync tree). The tasks in the fresh-task deque, from top-end to bottom-end, are from shallow to deep in the sync tree. This property simplifies the implementations for Action 2, 4 and the idle subroutine.

HJ compiler takes HJ language as input and performs automatic code generation to support execution on the SLAW work-stealing runtime. To enable adaptive work-stealing at runtime, the HJ compiler generates both work-first and help-first code under the two branches of a conditional branch. The branch taken at runtime is decided by a query to the runtime for the scheduling policy.

IV. Locality-aware Scheduling

SLAW is designed for locality-aware scheduling in combination with adaptive scheduling, using locality hints provided by the programmer. SLAW scheduler groups workers into *places*. Each place has a single *mailbox* to receive tasks from workers in other places. When a remote-place task is created, it is sent to the mailbox of the destination place. All place-local tasks are scheduled according to the adaptive scheduling algorithm described in Section III. The tasks in the mailbox of a place have lower priority than the tasks in the deques for workers executing in the same place. When a worker becomes idle, it will first check tasks in its local deque, then tasks in other workers' deque within the same place, and finally it will check the mailbox.

In our current implementation, cross-place steals are disabled to avoid counter-productive steals, which can occur in the following situations for places p_1 and p_2: 1) a task from p_1 is stolen by a worker in p_2 when there are still idle workers in p_1. 2) two tasks for p_1 and p_2 are created consecutively and all workers in p_1 are busy. In this case, if the worker in p_2 steals and executes the first task, the cache locality for p_2's other tasks may be polluted.

V. Experimental Results

A. Setup

In this paper, we report performance results obtained on the following two machines:

1) **Niagara 2**: This system includes a 8-core 64-thread 1.2GHz UltraSPARC T2 processor with 32GB main memory. All cores share a single 4MB L2 cache, thus it is not interesting to locality-aware scheduling. Only locality-oblivious deployment is used (1 place for all workers) on Niagara 2.
2) **Xeon SMP**: This system includes four Quad-Core Intel E7330 processors running at 2.40GHz with 32GB main memory. Each Quad-core processor has two core-pairs and each core-pair share a 3MB L2 cache. The locality-aware deployment provided for the SLAW scheduler has 8 places, with 1 or 2 workers per place.

The implementation used to evaluate SLAW in this paper is based on Java to facilitate portability across the above systems. Each worker is implemented as a separate Java thread. The JVM used on both machines is Sun Hotspot JDK 1.6. In both cases, the JVM was invoked with the following parameters: "-Xmx2g -Xms2g -Xmn1g -server -Xss8M -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:+UseBiasedLocking -XX:+AggressiveOpts". The experiment also includes some Cilk++ results. The Cilk++ release used is based on gcc v4.2.4. Both Cilk++ code and the serial C code were compiled using the -O2 option.

We evaluate the SLAW work-stealing scheduler on a variety of benchmarks listed in Table I. To reduce the impact of JVM overheads in the evaluation, including JIT compilation and garbage collection, the execution time reported is the average of the three best benchmark iterations from three separate VM invocations. Each VM invocation performs 10 benchmark iterations.

B. Sensitivity Analysis of Parameters in SLAW Scheduler

Figures 6(a) - 6(f) contain performance results obtained on the Niagara 2 machine to analyze the sensitivity of the INT, F, and S parameters on the performance of the SLAW scheduler, as discussed in the following subsections. Based on this sensitivity analysis, the default parameter value of SLAW is presented in Table II.

1) Impact of INT Parameter: Figures 6(a) - 6(d) study the impact of the policy evaluation interval, INT, on the performance of the SLAW scheduler.

Figure 6(a) shows the impact of INT on the execution time of the Fib(35) microbenchmark on 1 worker, with S and F set to their default values of 256 and 128 respectively. Since no stealing occurs in the 1-worker case, the adaptive heuristic will switch very quickly from help-first to work-first, and each subsequent re-evaluation will keep the policy as work-first for this case. The largest overhead is incurred when INT=1, since the spawning policy is re-evaluated for every spawned task. This suggests that INT should not be made too small. However, even in the INT=1 worst case, the overhead of the adaptive policy is only about 5% compared to the work-first policy. The overhead rapidly approaches zero with increasing values of INT. The execution time for the help-first policy is too big to fit in Figure 6(a) (about 9× slower than the work-first policy due to the context switching incurred at every task synchronization point).

Figure 6(b) repeats the evaluation in Figure 6(a), but with 32 workers instead of 1 worker. In this case, we see a negative performance impact of selecting an INT value that's too large. If the interval is too large, the performance degrades as shown in Figure 6(b) and 6(d). This is because, for a recursive benchmark like Fib, work-first is the best spawning policy. The SLAW scheduler will start with the help-first policy at the beginning and then switch to work-first. However, if INT is too

Benchmark	Type	Description	Source
Fib(35)	Micro Recursive	Two-way recursive Fibonacci for n=35, with no sequential threshold/cutoff	Cilk++
FJ(1024)	Micro Flat	Create 1024 dummy tasks and join them	JGF
SOR	Loop	2D Successive Over-Relaxation algorithm on a 2000 $\times$ 2000 float array	JGF
CG.A	Loop	Conjugate Gradient, size A	NPB 3.0
MG.A	Loop	Multi-Grid, size A	NPB 3.0
Sort	Recursive	Parallel Merge Sort on 50331648 random integers	BOTS [19]
Matmul	Recursive	Recursive Matrix Multiplication. (two 1500*1500 double matrix , Threshold=64)	Cilk++
LU	Recursive	Recursive LU Decomposition (2048*2048 double matrix, BlockSize=64)	JCilk
GC	Recursive	Graph Coloring using Parallel Constraint Satisfaction Search(CLIQUE_10,10 colors)	[20]
PDFS	Irregular Recursive	Parallel-DFS(Figure 1) on a Torus graph with 4M nodes	XWS [21]

TABLE I
LIST OF BENCHMARKS IMPLEMENTED IN HJ AND THEIR SOURCES

large, then some noticeable context switch overhead will be observed before the policy switch occurs. The same situation occurs for the PDFS benchmark in Figure 6(d). When we increase the INT value past 64, the throughput of the parallel depth first search benchmark declines.

Figure 6(c) shows the impact of INT on the execution time of SOR on 64 workers. In this fork-join version of SOR, 64 tasks are distributed among 64 workers in each outer (time-step) iteration, and these tasks are joined with a finish construct at the end of each iteration. Note that stealing is very frequent in this example, since 63 out of 64 tasks will be stolen in each iteration, thereby implying that the stealing rate is high and help-first policy is the best choice. . Because the adaptive schedule starts with help-first policy and re-evaluates the policy after every INT spawns, this experiment suggests that the INT should be set to be greater or equal the number of workers. Doing so will ensure that at least one task is spawned for each worker using the help-first policy before the worker switches to work-first policy. If a worker switches to the work-first policy too early, it will delay task creation and negatively affect the performance of the entire application. For the same reason, the fresh deque threshold should also be to be equal or greater than the number of workers to hold at least one task for each worker.

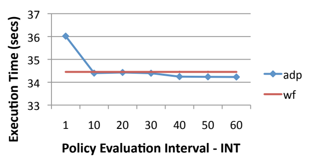
For the reasons described above, since the maximum number of workers is 64 (on Niagara 2), the default value of INT is set to 64 for the experiments presented in this paper.

2) Impact of Parameter F: Figure 6(e) shows how the throughput of the PDFS benchmark varies for different values of the fresh task threshold, F. The other two parameters INT and S are fixed at 50 and 256 respectively. INT is fixed at 50 for this example because (INT=50,S=256) was the best combination we found for PDFS after enumerating the parameter space. All other experimental results reported in this paper use the default parameter value unless otherwise specified.

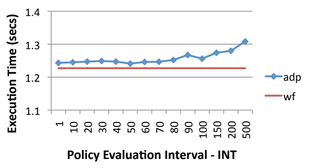
We do not find any correlation between the throughput and F. This is in part because F is a soft bound that has lower priority in the SLAW algorithm than the stack bound. The worker will always use the help-first policy to create fresh tasks regardless of the number of existing fresh tasks, if the stack bound prevents the use of the work-first policy.

In the previous discussion on the parameter INT, we also mentioned the reason why F should be equal or greater than the number of workers. The default value of F in SLAW is set to 128.

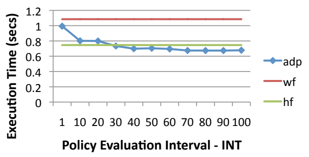
3) Impact of Parameter S: Figure 6(f) shows the throughput of the PDFS benchmark as a function of the stack threshold S. When S is set to 1, the adaptive schedule becomes equivalent to the work-first schedule. Interestingly, if the stack threshold is set too large, the performance also degrades. This is because if the stack becomes too deep, the number of memory pages spanned by the runtime call stack increases, which in turn leads to an increase in TLB misses. The default stack threshold in SLAW is set to 256 activation frames. Among the benchmarks used in this paper, only the PDFS benchmark requires a stack that grows proportionally with the input problem size. The stack requirement for other benchmarks is bounded by a small number; consequently they do not hit the stack bound.



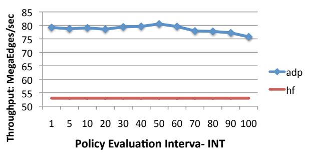
(a) Fib(35),S=256,F=128,W=1



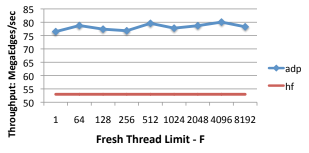
(b) Fib(35),S=256,F=128,W=32



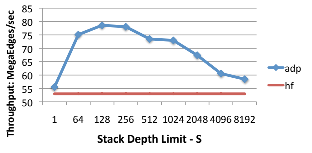
(c) SOR,S=256,F=128,W=64



(d) PDFS,S=256,F=128,W=64



(e) PDFS,S=256,INT=50,W=64



(f) PDFS,F=128,INT=50,W=64

Fig. 6. Analysis of Adaptive Schedule Parameter Sensitivity on the Niagara 2 system. The benchmark name, SLAW parameter values (S, F, INT), and the number of workers (W) are specified in the sub-figure captions. Better performance is indicated by smaller values in (a),(b),(c) and larger values in (d),(e),(f).

Parameter	INT	F	S
Default Value	64	128	256 activation frames

TABLE II
DEFAULT ADAPTIVE SCHEDULE PARAMETERS VALUE

C. Benchmark Results

Recursive parallelism and flat parallelism are two common patterns for task parallelism. In recursive parallelism, the parallelism is expressed recursively. Thus the task spawn tree is usually deep. On the other hand, flat parallelism corresponds to those do-cross loops and pointer-chasing cases, where asynchronous tasks are iteratively spawned. In flat parallelism, the depth of the task spawn tree is usually wide and shallow.

In work-stealing, each steal will yield the whole sub-tree for the thief to work on. Intuitively, for a spawn tree with large depth, it implies that each worker will get more work compared to a shallow spawn tree. As a result, stealing is generally considered rare in recursive task parallelism but frequent in flat parallelism.

One optimization that has been studied in previous research is to transform some flat loops to recursive style [22], [5]. This optimization requires compiler support, and only applies to do-all loops on a divisible region and does not apply to do-cross loops or pointer-chasing programs. As the main contribution of this paper is to show the robustness of the runtime, we do not apply such optimizations. However, for the FJ microbenchmark, we do show the performance results of with and without the recursive loop optimization.

We use two microbenchmarks, Fib and FJ, to show the extreme cases where work-first policy is better than help-first policy and the help-first policy is better than the work-first policy respectively. Fib is used as the extreme case for recursive parallelism and FJ without the recursive loop optimization is used as the extreme case for flat parallelism. Table III shows the execution time of Fib on Niagara 2. Figure 7 shows the number of fork-joins performed per second. For Fib, the work-first policy is 10.2× faster than the help-first policy due to fewer context switches. In FJ, steals are frequent and the help-first policy is 4.6× faster than the work-first policy. In both micro-benchmarks, the performance of the adaptive scheduler is close to that of the better policy.

Figure 8 shows the number of fork-joins performed per second in a recursive-style fork-join (`fj-rec`), and compares the number to the iterative fork-join (`fj`). In `fj-rec`, the tasks are recursively spawned. In order to spawn 1024 parallel tasks, the depth of the task spawn tree is 11. When the number of threads is small ($<= 8$), the work-first policy performs than better than the help-first policy. This is because the number of steals is infrequent compared to the task spawned. As the number of threads increase, the steal becomes more frequent (considering the depth of the task spawn is 10 for 1024 tasks), and the help-first policy performs better than the work-first policy. This example is interesting because it shows that the best choice of scheduling policy is more a dynamic choice than a static choice, although the shape of a program can probably give some clue. The experiment also confirms that `fj-rec` is

Wrks	1	2	4	8	16	32	64
hf	334.14	173.64	79.43	39.71	21.43	11.04	8.04
wf	34.45	17.13	8.65	4.31	2.24	1.23	0.87
adp	34.25	16.99	8.54	4.36	2.25	1.25	0.90

TABLE III
PERFORMANCE RESULTS FOR FIB(35) MICROBENCHMARK ON NIAGARA 2 USING 1 TO 64 WORKERS. EXECUTION TIME (IN SECONDS) IS REPORTED. (SMALLER IS BETTER.)

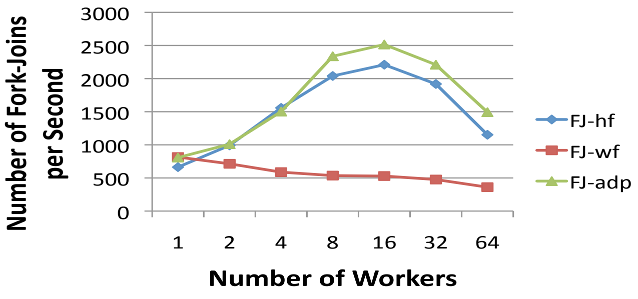


Fig. 7. Performance results for FJ(1024) microbenchmark (tasks are spawned iteratively) on Niagara 2 using 1 to 64 workers. Number of fork-joins performed per second is reported. (Bigger is better.)

more scalable than `fj`, as the task spawns are now performed in parallel as well. However, the sequential overhead of the `fj-rec` is higher than the iterative `fj`, which explains the lower performance when the number of threads is small.

Figure 9 shows the speedup of the SLAW scheduler on Niagara 2 over the Java-serial version with one exception for PDFS, whose speedup is based on 1-thread help-first execution. Both the serial version and the work-first schedule of PDFS will overflow the stack as described in Section II-A2. This is why there is no bar for the *wf* in the figure. This exception also applies to Figure 10.

Three scheduling policies are compared: help-first only, work-first only and the adaptive scheduling algorithm described in Section III. As all cores on Niagara 2 share the same L2 cache, this experiment uses the locality-oblivious deployment, which specifies only 1 place with all 64 workers. No processor binding is used.

CG.A, MG.A and SOR are flat, loop-based parallel benchmarks. In these benchmarks, the help-first policy performs better than the work-first policy. The results in Figure 9 show that the adaptive scheduling algorithm matches or exceeds the performance of the help-first policy for these benchmarks.

Sort, Matmul, LU and GC are task recursive parallel benchmarks. In these benchmarks, the work-first policy is better than or almost the same as the help-first policy. The result shows the adaptive scheduling algorithm matches or exceeds the performance of work-first policy in those benchmarks.

PDFS is an irregular graph computation. Irregular graph computations are interesting because the structure of the spawn tree depends on the order in which nodes are visited (labeled) in parallel. We used the Parallel Depth First Search benchmark (PDFS) studied in [21] (kernel code shown in Figure 1), and applied it to a two-dimensional 2000 × 2000 torus graph consisting of 4 million nodes. Our results show that the

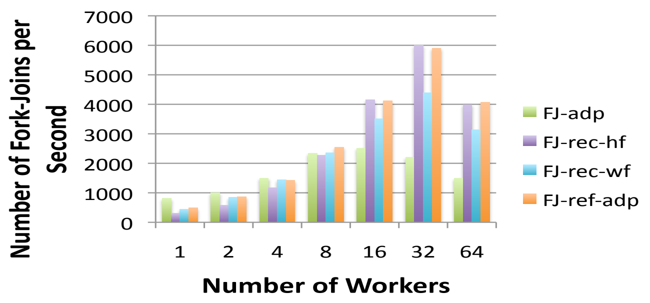


Fig. 8. Performance results for FJ(1024) microbenchmark (in FJ-rec, tasks are spawned recursively.) on Niagara 2 using 1 to 64 workers. Number of fork-joins performed per second is reported. (Bigger is better.)

adaptive approach outperformed the help-first policy for this benchmark because of its ability to combine help-first and work-first policies. At the beginning, all workers are idle and stealing is frequent thereby making help-first the more desirable policy. After each worker gets some work, they begin traversing the graph and stealing becomes less frequent, thus causing the adaptive runtime to switch to the work-first policy. The work-first policy incurs no synchronization overhead and executes the tasks as if they are sequential calls. Finally, according to the stack condition in the adaptive scheduling algorithm, the runtime will switch back to the help-first policy when it becomes necessary to avoid overflowing the stack size limit.

Figure 10 shows the speedup of the SLAW scheduler on Xeon SMP using the three scheduling policies. For reference, we report also Cilk++'s speedup for those benchmarks for which Cilk version is available(SORT, MATMUL, LU). We also translate the JGF SOR to Cilk++ and use the *cilk_for* to parallelize the loop. To factor out uniprocessor performance differences between Java and C, the speedup for SLAW in this figure is based on the Java-serial version and Cilk++'s speedup is based on the C-serial version. This experiment uses the locality-oblivious deployment. The experimental results of the locality-aware scheduling are presented later in the Section V-D.

For Sort, Matmul and LU, SLAW achieves over 10× speedup on Xeon SMP. SLAW scales almost linearly on Matmul. Cilk++ also scales almost linearly from 1 worker to 16 workers, but its speedup looks smaller because Cilk++'s 1-worker case is 2.4× slower than the C-serial version due to some optimizations that are disabled by the Cilk++ compiler. CG, MG and SOR do not scale beyond 4× hit a memory-bandwidth wall. The scalability of CG and SOR can be significantly improved by locality-aware scheduling as shown next in Section V-D. We also discuss the scalability issue of GC and PDFS in Section V-E,

D. Locality-aware Scheduling Results

We show the performance improvement on memory-bandwidth bound applications by locality-aware work-stealing scheduling with the locality hints provided by the programmer on Xeon SMP. With 4 quad-core processors in Xeon SMP, there are in total of eight(8) L2 shared caches on the machine with a total size of 24M. Thus, the locality-aware deployment provided for the SLAW scheduler has 8-places, with 1 or 2 workers per place. The SLAW scheduler will bind workers to virtual processors.

Figure 11 shows the performance results of the SLAW locality-aware scheduler using two locality-aware deployments: one with 8-places and 1 worker per place with a total of 8 workers; the other has 8-places and 2 workers per place with a total of 16 workers. The scheduling policy used for task scheduling with each place is the adaptive schedule. The speedup reported for Cilk++ is also based on the Java-serial version in order to compare the execution time.

For SOR, the total data size is 2000 × 2000 float matrix. Divided by 8 places, the data for each place fits into the 3M L2 cache. The 8-place-8-worker locality-aware scheduling is 2.1× faster than the locality-oblivious scheduling using the adaptive scheduling and the speedup for 8-place-16-worker locality-aware scheduling is 2.6×.

For CG.A, the total data size is 2198000 double sparse matrix elements. Divided by 8 places, the data set for each place also fits the L2 cache, thus enable temporal data-reuse. Experimental results of the 8-place-8-worker locality-aware scheduling is 1.7× faster than the locality-oblivious scheduling. We do not get improvement from 1 worker per place to 2 workers per place. This is due to the cache contention between two workers under the same place.

MG.A cannot benefit from temporal reuse on Xeon SMP because the memory footprint of each place exceeds the capacity of the L2 cache.

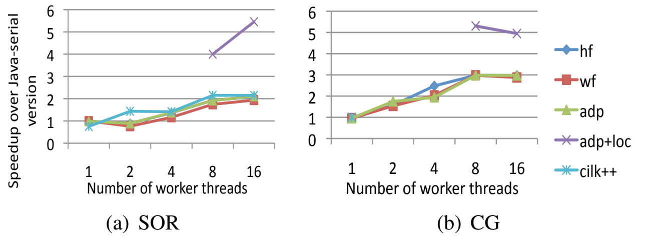


Fig. 11. Comparing Locality-aware scheduler with locality-oblivious scheduler on SOR and CG on Intel Xeon SMP. The locality-aware deployment for adp+locality has 8 places with 1 or 2 workers per place. The workers are binded to virtual processors.

E. Other Scalability Issues

Benchmarks like PDFS and GC scale well on Niagara 2 with one shared L2 cache, but not on Xeon SMP with separate L2 caches. This trend has also been observed by other researchers and some explanations are proposed, such as the allocation wall theory [23]. Besides allocation-wall, we also observe the false-sharing between application objects accessed from different worker threads. For parallel graph algorithms, there are lots of small objects(e.g. graph nodes) accessed by multiple worker threads. Without padding these objects, this may cause two objects accessed by two different threads lay in the same cache line and thus will cause false sharing. But automatic padding for application objects in Java is not

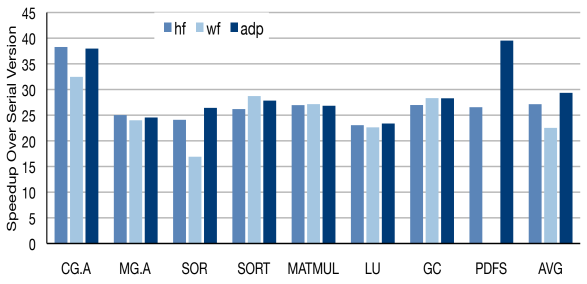


Fig. 9. Performance results on Niagara 2. Deployment is locality-oblivious(1-place, 64 workers) with no processor binding.

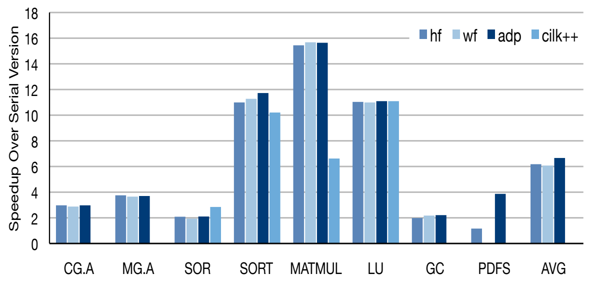


Fig. 10. Performance results on Xeon SMP. Deployment is locality-oblivious.(1-place, 16 workers) with no processor binding.

straightforward. The full investigation of the scalability issue for these benchmarks involves the JVM memory management system and is beyond the scope of this paper.

VI. Related Work

Following the popular Cilk-5 work-stealing runtime [4], many task parallel runtime systems have been developed in the last 10 years including Cilk++ [5], Intel Threading Building Blocks [22], Java's ForkJoin Framework [7], Microsoft's Task Parallel Library [8], StackThreads/MP [9], and XWS [21]. These runtimes are either exposed as library routines, or serve as implementations of new language constructs, or some combination thereof. SLAW is a work-stealing runtime designed for a task parallel language called HJ, which is derived from X10 v1.5. In our implementation, the HJ compiler generates calls to the SLAW runtime. However, it is possible to use the techniques introduced in this paper to support a library approach as well.

Some work-stealing runtime schedulers use the work-first policy inspired by Cilk [4], [5], [9] while others use the help-first policy [22], [8], [7]. Especially, Intel TBB uses the "depth-first execution and breadth-first theft" principle to raise parallelism as well as to maintain locality [6], which is similar to the help-first policy. XWS exposes low-level deque operations, but does not have compiler support for either policy. SLAW includes compiler and runtime support for both work-first and help-first policies. The programmer has the option of selecting a fixed scheduling policy in HJ or using SLAW's default adaptive scheduler that selects the policy on a per-task basis at runtime.

Two approaches in past work have been shown to be provably space-efficient. One category consists of work-stealing schedulers with the work-first policy, which were first proven to be space-efficient for fully-strict computations [14]. The same result was later extended for terminally-strict computations [16] in languages like X10 and HJ. Another category of techniques is based on depth-first schedulers [17] such as DFDeques [24], which can use less memory than work-stealing schedulers. All these scheduling techniques focus on the memory space usage without bounding the stack pressure of individual processors, because all models assume that a serial depth-first schedule can run successfully. SLAW's adaptive scheduling algorithm addresses this problem by tracking the stack pressure and generating schedules with bounded stack usage, even in cases when a sequential execution cannot run successfully.

Previous research has compared the pros and cons of different scheduling policies [10]. Work-first works better for situations when steals are rare, while help-first works better when steals are frequent. [25] compared depth-first and breadth-first task spawning policy in OpenMP task scheduling and found that depth-first performed slightly better than the breadth-first policy. Their breadth-first policy is different from the help-first policy in work-stealing because it uses a global task pool to store all untied tasks, while work-stealing uses local pool per worker. Second, the benchmarks they used to evaluate the performance are mostly task recursive parallel programs where steals are rare.

Another area of research aims to reduce scheduling overhead by increasing task granularity. This can be done by chunking a parallel loop or by using a cut-off technique to run the task sequentially if the task is too fine-grained [26], [22]. These techniques are applicable to SLAW as well. [27] proposed a back-tracked scheduling approach in which the program runs normally in sequential mode and back-tracks upon a steal request. This approach requires the programmer to explicitly write roll-back code at the language level.

The locality issue in work-stealing has also received a lot of attention in past work. Acar et al. [11] provide a theoretical bound on the number of cache misses for randomized work-stealing and gives a locality-guided work-stealing implementation on a single-core SMP. SLAW's locality-aware scheduling is designed for multicore SMPs and stealing occurs within a single place but not across places. Chen et al. [28] studied and compared the cache behavior between work-stealing and parallel depth-first scheduler on simulators for cores that share the L2 cache. They proposed approaches to control the task granularity and promote constructive cache sharing. In a multicore SMP environment, the concept of places in HJ can be used to enable constructive cache sharing on a single multicore processor.

Intel TBB has the `affinity_partitioner` structure to utilize temporal cache-reuse by binding the same iteration to the same worker thread that previously executed the iteration. TBB allows stealing regardless of the affinity and has a mechanism to reduce counter-productive stealing. SLAW currently disables cross-place stealing in order to avoid counter-productive stealing.

VII. Conclusions and Future Work

In this paper, we introduced SLAW, a scalable locality-aware adaptive work-stealing scheduler. SLAW uses a novel adaptive scheduling algorithm with guaranteed space bounds, and performs locality-aware scheduling in accordance with locality hints (place annotations) provided by the programmer. Experimental results show that the use of the adaptive scheduler delivered up to 4.5× speedup over the work-first policy on iterative loop parallelism (FJ) and 9.2× speedup over the help-first policy on recursive task parallelism (Fib). In addition, the adaptive algorithm can achieve scalable performance with support for large data sizes (as in the case of PDFS) that cannot be achieved by the use of a fixed policy. Finally, our results show that locality-aware scheduling can achieve up to 2.6× speedup over locality-oblivious scheduling, for the benchmarks and platforms studied in this paper.

For future work, we will extend the place-locality model and implementation to allow load balancing across places while avoiding counter-productive stealing. Another direction for future research is to extend the work-stealing runtime system presented in this paper to support additional language features such as futures and phasers. We also plan to investigate other scalability bottlenecks such as the impact of JVM memory management system on the work-stealing runtime.

Acknowledgments

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. We also gratefully acknowledge support from an IBM Open Collaborative Faculty Award. We would like to thank all Habanero team members for their contributions to the HJ software that served as the foundational infrastructure for this research. Finally, we would like to thank Doug Lea for access to the UltraSPARC T2 SMP system used to obtain the experimental results reported in this paper.

References

[1] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[2] Http://projectfortress.sun.com/Projects/Community.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05*. New York, NY, USA: ACM, 2005, pp. 519–538.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, 1998.

[5] Cilk Arts: http://www.cilk.com.

[6] *Intel(R) Threading Building Blocks*, Intel Corporation.

[7] D. Lea, "A java fork/join framework," in *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. New York, NY, USA: ACM, 2000, pp. 36–43.

[8] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," vol. 44, no. 10. New York, NY, USA: ACM, 2009, pp. 227–242.

[9] K. Taura, K. Tabata, and A. Yonezawa, "Stackthreads/mp: integrating futures into calling standards," *SIGPLAN Not.*, vol. 34, no. 8, pp. 60–71, 1999.

[10] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[11] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2000, pp. 1–12.

[12] "The Habanero Java (HJ) Programming Language." [Online]. Available: http://habanero.rice.edu/hj

[13] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper, "Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures," in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM, 2009, pp. 91–100.

[14] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[15] J. R. Larus and R. Rajwar, *Transactional Memory*. Morgan and Claypool, 2006.

[16] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, "Deadlock-free scheduling of x10 computations with bounded resources," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 229–240.

[17] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *J. ACM*, vol. 46, no. 2, pp. 281–321, 1999.

[18] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, 2005.

[19] 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," in *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131.

[20] R. Haralick and G. Elliott, "Improving tree search efficiency for constraint-satisfaction problems," *Artificial Intelligence*, 1980.

[21] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 536–545.

[22] A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in tbb," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–8.

[23] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao, "Allocation wall: a limiting factor of java applications on emerging multi-core platforms," in *OOPSLA '09*. New York, NY, USA: ACM, 2009, pp. 361–376.

[24] G. J. Narlikar, "Scheduling threads for low space requirement and good locality," in *SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1999, pp. 83–95.

[25] A. Duran, J. Corbaln, and E. Ayguad, "Evaluation of openmp task scheduling strategies," in *Proceeding of IWOMP 2008*.

[26] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[27] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based load balancing," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2009, pp. 55–64.

[28] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, "Scheduling threads for constructive cache sharing on cmps," in *SPAA '07*. New York, NY, USA: ACM, 2007, pp. 105–115.