The Data Locality of Work Stealing

Umut A. Acar
School of Computer Science
Carnegie Mellon University
umut@cs.cmu.edu

Guy E. Blelloch
School of Computer Science
Carnegie Mellon University
guyb@cs.cmu.edu

Robert D. Blumofe
Department of Computer Sciences
University of Texas at Austin
rdb@cs.utexas.edu

Abstract

This paper studies the data locality of the work-stealing scheduling algorithm on hardware-controlled shared-memory machines. We present lower and upper bounds on the number of cache misses using work stealing, and introduce a locality-guided work-stealing algorithm along with experimental validation.

As a lower bound, we show that there is a family of multithreaded computations G_n each member of which requires $\Theta(n)$ total instructions (work), for which when using work-stealing the number of cache misses on one processor is constant, while even on two processors the total number of cache misses is $\Omega(n)$. This implies that for general computations there is no useful bound relating multiprocessor to uninprocessor cache misses. For nested-parallel computations, however, we show that on P processors the expected additional number of cache misses beyond those on a single processor is bounded by $O(C\lceil \frac{m}{s} \rceil P T_\infty)$, where m is the execution time of an instruction incurring a cache miss, s is the steal time, C is the size of cache, and T_∞ is the number of nodes on the longest chain of dependences. Based on this we give strong bounds on the total running time of nested-parallel computations using work stealing.

For the second part of our results, we present a locality-guided work stealing algorithm that improves the data locality of multithreaded computations by allowing a thread to have an affinity for a processor. Our initial experiments on iterative data-parallel applications show that the algorithm matches the performance of static-partitioning under traditional work loads but improves the performance up to 50% over static partitioning under multiprogrammed work loads. Furthermore, the locality-guided work stealing improves the performance of work-stealing up to 80%.

1 Introduction

Many of today's parallel applications use sophisticated, adaptive algorithms which are best realized with parallel programming systems that support dynamic, lightweight threads such as Cilk [8], Nesl [5], Hood [10], and many others [3, 16, 17, 21, 32]. The core of these systems is a thread scheduler that balances load among the processes. In addition to a good load balance, however, good data locality is essential in obtaining high performance from modern parallel systems.

SPAA 2000, Bar Harbor, Maine USA

Several researches have studied techniques to improve the data locality of multithreaded programs. One class of such techniques is based on software-controlled distribution of data among the local memories of a distributed shared memory system [15, 22, 26]. Another class of techniques is based on hints supplied by the programmer so that "similar" tasks might be executed on the same processor [15, 31, 34]. Both these classes of techniques rely on the programmer or compiler to determine the data access patterns in the program, which may be very difficult when the program has complicated data access patterns. Perhaps the earliest class of techniques was to attempt to execute threads that are close in the computation graph on the same processor [1, 9, 20, 23, 26, 28]. The work-stealing algorithm is the most studied of these techniques [9, 11, 19, 20, 24, 36, 37]. Blumofe et al showed that fully-strict computations achieve a provably good data locality [7] when executed with the work-stealing algorithm on a dag-consistent distributed shared memory systems. In recent work, Narlikar showed that work stealing improves the performance of space-efficient multithreaded applications by increasing the data locality [29]. None of this previous work, however, has studied upper or lower bounds on the data locality of multithreaded computations executed on existing hardware-controlled shared memory systems.

In this paper, we present theoretical and experimental results on the data locality of work stealing on hardware-controlled shared memory systems (HSMSs). Our first set of results are upper and lower bounds on the number of cache misses in multithreaded computations executed by the work-stealing algorithm. Let $M_1(C)$ denote the number of cache misses in the uniprocessor execution and $M_P(C)$ denote the number of cache misses in a P-processor execution of a multithreaded computation by the work stealing algorithm on an HSMS with cache size C. Then, for a multithreaded computation with T_1 work (total number of instructions), T_∞ critical path (longest sequence of dependences), we show the following results for the work-stealing algorithm running on a HSMS.

- Lower bounds on the number of cache misses for general computations: We show that there is a family of computations G_n with $T_1 = \Theta(n)$ such that $M_1(C) = 3C$ while even on two processors the number of misses $M_2(C) = \Theta(n)$.
- Upper bounds on the number of cache misses for nested-parallel computations: For a nested-parallel computation, we show that $M_P \leq M_1(C) + 2C\tau$, where τ is the number of steals in the P-processor execution. We then show that the expected number of steals is $O(\lceil \frac{m}{s} \rceil P T_\infty)$, where m is the time for a cache miss and s is the time for a steal.
- Upper bound on the execution time of nested-parallel computations: We show that the expected execution time of a

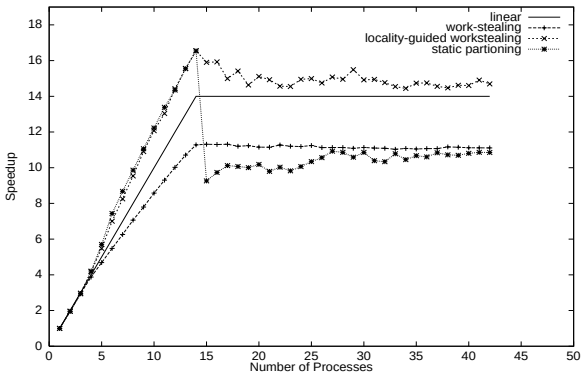

Figure 1: The speedup obtained by three different over-relaxation algorithms.

nested-parallel computation on P processors is $O(\frac{T_1(C)}{P} + m\lceil\frac{m}{s}\rceil C T_\infty + (m+s)T_\infty)$, where $T_1(C)$ is the uniprocessor execution time of the computation including cache misses.

As in previous work [6, 9], we represent a multithreaded computation as a directed, acyclic graph (***dag***) of instructions. Each node in the dag represents a single instruction and the edges represent ordering constraints. A nested-parallel computation [5, 6] is a race-free computation that can be represented with a series-parallel dag [33]. Nested-parallel computations include computations consisting of parallel loops and fork an joins and any nesting of them. This class includes most computations that can be expressed in Cilk [8], and all computations that can be expressed in Nesl [5]. Our results show that nested-parallel computations have much better locality characteristics under work stealing than do general computations. We also briefly consider another class of computations, computations with futures [12, 13, 14, 20, 25], and show that they can be as bad as general computations.

The second part of our results are on further improving the data locality of multithreaded computations with work stealing. In work stealing, a processor steals a thread from a randomly (with uniform distribution) chosen processor when it runs out of work. In certain applications, such as iterative data-parallel applications, random steals may cause poor data locality. The locality-guided work stealing is a heuristic modification to work stealing that allows a thread to have an affinity for a process. In locality-guided work stealing, when a process obtains work it gives priority to a thread that has affinity for the process. Locality-guided work stealing can be used to implement a number of techniques that researchers suggest to improve data locality. For example, the programmer can achieve an initial distribution of work among the processes or schedule threads based on hints by appropriately assigning affinities to threads in the computation.

Our preliminary experiments with locality-guided work stealing give encouraging results, showing that for certain applications the performance is very close to that of static partitioning in dedicated mode (i.e. when the user can lock down a fixed number of processors), but does not suffer a performance cliff problem [10] in multiprogrammed mode (i.e. when processors might be taken by other users or the OS). Figure 1 shows a graph comparing work stealing, locality-guided work stealing, and static partitioning for a simple over-relaxation algorithm on a 14 processor Sun Ultra Enterprise. The over-relaxation algorithm iterates over a 1 dimensional array performing a 3-point stencil computation on each step. The superlinear speedup for static partitioning and locality-guided work stealing is due to the fact that the data for each run does not fit into the $\mathrm{L}2$ cache of one processor but fits into the collective $\mathrm{L}2$ cache of 6 or more processors. For this benchmark the following can be seen from the graph.

1. Locality-guided work stealing does significantly better than standard work stealing since on each step the cache is prewarmed with the data it needs.

2. Locality-guided work stealing does approximately as well as static partitioning for up to 14 processes.

3. When trying to schedule more than 14 processes on 14 processors static partitioning has a serious performance drop. The initial drop is due to load imbalance caused by the coarse-grained partitioning. The performance then approaches that of work stealing as the partitioning gets more fine-grained.

We are interested in the performance of work-stealing computations on hardware-controlled shared memory (HSMSs). We model an HSMS as a group of identical processors each of which has its own cache and has a single shared memory. Each cache contains C blocks and is managed by the memory subsystem automatically. We allow for a variety of cache organizations and replacement policies, including both direct-mapped and associative caches. We assign a server process with each processor and associate the cache of a processor with process that the processor is assigned. One limitation of our work is that we assume that there is no false sharing.

2 Related Work

As mentioned in Section 1, there are three main classes of techniques that researchers have suggested to improve the data locality of multithreaded programs. In the first class, the program data is distributed among the nodes of a distributed shared-memory system by the programmer and a thread in the computation is scheduled on the node that holds the data that the thread accesses [15, 22, 26]. In the second class, data-locality hints supplied by the programmer are used in thread scheduling [15, 31, 34]. Techniques from both classes are employed in distributed shared memory systems such as COOL and Illinois Concert [15, 22] and also used to improve the data locality of sequential programs [31]. However, the first class of techniques do not apply directly to HSMSs, because HSMSs do not allow software controlled distribution of data among the caches. Furthermore, both classes of techniques rely on the programmer to determine the data access patterns in the application and thus, may not be appropriate for applications with complex data-access patterns.

The third class of techniques, which is based on execution of threads that are close in the computation graph on the same process, is applied in many scheduling algorithms including work stealing [1, 9, 23, 26, 28, 19]. Blumofe et al showed bounds on the number of cache misses in a fully-strict computation executed by the work-stealing algorithm under the dag-consistent distributed shared-memory of Cilk [7]. Dag consistency is a relaxed memory-consistency model that is employed in the distributed shared-memory implementation of the Cilk language. In a distributed Cilk application, processes maintain the dag consistency by means of the BACKER algorithm. In [7], Blumofe et al bound the number of shared-memory cache misses in a distributed Cilk application for caches that are maintained with the LRU replacement policy. They assumed that accesses to the shared memory are distributed uniformly and independently, which is not generally true because threads may concurrently access the same pages by algorithm design. Furthermore, they assumed that processes do

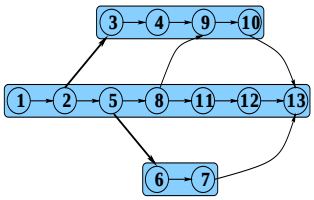

Figure 2: A dag (directed acyclic graph) for a multithreaded computation. Threads are shown as gray rectangles.

not generate steal attempts frequently by making processes do additional page transfers before they attempt to steal from another process.

3 The Model

In this section, we present a graph-theoretic model for multithreaded computations, describe the work-stealing algorithm, define series-parallel and nested-parallel computations and introduce our model of an HSMS (Hardware-controlled Shared-Memory System).

As with previous work [6, 9] we represent a multithreaded computation as a directed acyclic graph, a ***dag***, of instructions (see Figure 2). Each node in the dag represents an instruction and the edges represent ordering constraints. There are three types of edges, continuation, spawn, and dependency edges. A ***thread*** is a sequential ordering of instructions and the nodes that corresponds to the instructions are linked in a chain by ***continuation*** edges. A ***spawn*** edge represents the creation of a new thread and goes from the node representing the instruction that spawns the new thread to the node representing the first instruction of the new thread. A ***dependency*** edge from instruction i of a thread to instruction j of some other thread represents a synchronization between two instructions such that instruction j must be executed after i. We draw spawn edges with thick straight arrows, dependency edges with curly arrows and continuation edges with thick straight arrows throughout this paper. Also we show paths with wavy lines.

For a computation with an associated dag G, we define the ***computational work***, T_1, as the number of nodes in G and the ***critical path***, T_∞, as the number of nodes on the longest path of G.

Let u and v be any two nodes in a dag. Then we call u an ***ancestor*** of v, and v a ***descendant*** of u if there is a path from u to v. Any node is its descendant and ancestor. We say that two nodes are ***relatives*** if there is a path from one to the other, otherwise we say that the nodes are ***independent***. The children of a node are independent because otherwise the edge from the node to one child is redundant. We call a common descendant y of u and v a ***merger*** of u and v if the paths from u to y and v to y have only y in common. We define the ***depth*** of a node u as the number of edges on the shortest path from the root node to u. We define the ***least common ancestor*** of u and v as the ancestor of both u and v with maximum depth. Similarly, we define the ***greatest common descendant*** of u and v, as the descendant of both u and v with minimum depth. An edge (u, v) is ***redundant*** if there is a path between u and v that does not contain the edge (u, v). The ***transitive reduction*** of a dag is the dag with all the redundant edges removed.

In this paper we are only concerned with the transitive reduction of the computational dags. We also require that the dags have a single node with in-degree 0, the ***root***, and a single node with out-degree 0, the ***final*** node.

In a multiprocess execution of a multithreaded computation, independent nodes can execute at the same time. If two independent nodes read or modify the same data, we say that they are ***RR*** or ***WW*** sharing respectively. If one node is reading and the other is modifying the data we say they are ***RW*** sharing. RW or WW sharing can cause data races, and the output of a computation with such races usually depends on the scheduling of nodes. Such races are typically indicative of a bug [18]. We refer to computations that do not have any RW or WW sharing as ***race-free*** computations. In this paper we consider only race-free computations.

The work-stealing algorithm is a thread scheduling algorithm for multithreaded computations. The idea of work-stealing dates back to the research of Burton and Sleep [11] and has been studied extensively since then [2, 9, 19, 20, 24, 36, 37]. In the work-stealing algorithm, each process maintains a pool of ready threads and obtains work from its pool. When a process spawns a new thread the process adds the thread into its pool. When a process runs out of work and finds its pool empty, it chooses a random process as its victim and tries to steal work from the victim's pool.

In our analysis, we imagine the work-stealing algorithm operating on individual nodes in the computation dag rather than on the threads. Consider a multithreaded computation and its execution by the work-stealing algorithm. We divide the execution into discrete time steps such that at each step, each process is either working on a node, which we call the ***assigned node***, or is trying to steal work. The execution of a node takes 1 time step if the node does not incur a cache miss and m steps otherwise. We say that a node is ***executed*** at the time step that a process completes executing the node. The ***execution time*** of a computation is the number of time steps that elapse between the time step that a process starts executing the root node to the time step that the final node is executed. The ***execution schedule*** specifies the activity of each process at each time step.

During the execution, each process maintains a deque (doubly ended queue) of ready nodes; we call the ends of a deque the ***top*** and the ***bottom***. When a node, u, is executed, it enables some other node v if u is the last parent of v that is executed. We call the edge (u, v) an ***enabling edge*** and u the designated parent of v. When a process executes a node that enables other nodes, one of the enabled nodes become the assigned node and the process pushes the rest onto the bottom of its deque. If no node is enabled, then the process obtains work from its deque by removing a node from the bottom of the deque. If a process finds its deque empty, it becomes a ***thief*** and steals from a randomly chosen process, the ***victim***. This is a ***steal attempt*** and takes at least s and at most ks time steps for some constant $k \geq 1$ to complete. A thief process might make multiple steal attempts before succeeding, or might never succeed. When a steal succeeds, the thief process starts working on the stolen node at the step following the completion of the steal. We say that a steal attempt ***occurs*** at the step it completes.

The work-stealing algorithm can be implemented in various ways. We say that an implementation of work stealing is ***deterministic*** if, whenever a process enables other nodes, the implementation always chooses the same node as the assigned node for then next step on that process, and the remaining nodes are always placed in the deque in the same order. This must be true for both multiprocess and uniprocess executions. We refer to a deterministic implementation of the work-stealing algorithm together with the HSMS that runs the implementation as a ***work stealer***. For brevity, we refer to an execution of a multithreaded computation with a work stealer as an ***execution***. We define the ***total work*** as the number of steps taken by a uniprocess execution, including the cache misses, and denote it by $T_1(C)$, where C is the cache size. We denote the number of cache misses in a P-process execution with C-block caches as $M_P(C)$. We define the ***cache overhead*** of a P-process execution as $M_P(C) - M_1(C)$, where $M_1(C)$ is the number of misses in the uniprocess execution on the same work stealer.

We refer to a multithreaded computation for which the transi-

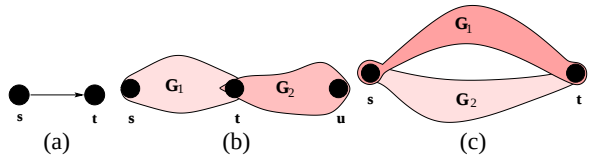

Figure 3: Illustrates the recursive definition for series-parallel dags. Figure (a) is the base case, figure (b) depicts the serial, and figure (c) depicts the parallel composition.

tive reduction of the corresponding dag is series-parallel [33] as a ***series-parallel computation***. A series-parallel dag $G(V, E)$ is a dag with two distinguished vertices, a ***source***, $s \in V$ and a ***sink***, $t \in V$ and can be defined recursively as follows (see Figure 3).

- *Base:* G consists of a single edge connecting s to t.
- *Series Composition:* G consists of two series-parallel dags $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ with disjoint edge sets such that s is the source of G_1, u is the sink of G_1 and the source of G_2, and t is the sink of G_2. Moreover $V_1 \cap V_2 = \{u\}$.
- *Parallel Composition:* The graph consists of two series-parallel dags $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ with disjoint edges sets such that s and t are the source and the sink of both G_1 and G_2. Moreover $V_1 \cap V_2 = \{s, t\}$.

A ***nested-parallel*** computation is a race-free series-parallel computation [6].

We also consider multithreaded computations that use futures [12, 13, 14, 20, 25]. The dag structures of computations with futures are defined elsewhere [4]. This is a superclass of nested-parallel computations, but still much more restrictive than general computations. The work-stealing algorithm for futures is a restricted form of work-stealing algorithm, where a process starts executing a newly created thread immediately, putting its assigned thread onto its deque.

In our analysis, we consider several cache organization and replacement policies for an HSMS. We model a cache as a set of (cache) ***lines***, each of which can hold the data belonging to a memory ***block*** (a consecutive, typically small, region of memory). One instruction can operate on at most one memory block. We say that an instruction ***accesses*** a block or the line that contains the block when the instruction reads or modifies the block. We say that an instruction ***overwrites*** a line that contains the block b when the instruction accesses some other block that replaces b in the cache. We say that a cache replacement policy is ***simple*** if it satisfies two conditions. First the policy is deterministic. Second whenever the policy decides to overwrite a cache line, l, it makes the decision to overwrite l by only using information pertaining to the accesses that are made after the last access to l. We refer to a cache managed with a simple cache-replacement policy as a simple cache. Simple caches and replacement policies are common in practice. For example, least-recently used (LRU) replacement policy, direct mapped caches and set associative caches where each set is maintained by a simple cache replacement policy are simple.

In regards to the definition of RW or WW sharing, we assume that reads and writes pertain to the whole block. This means we do not allow for false sharing—when two processes accessing different portions of a block invalidate the block in each other' s caches. In practice, false sharing is an issue, but can often be avoided by a knowledge of underlying memory system and appropriately padding the shared data to prevent two processes from accessing different portions of the same block.

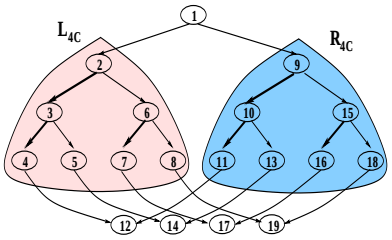

Figure 4: The structure for dag of a computation with a large cache overhead.

4 General Computations

In this section, we show that the cache overhead of a multiprocess execution of a general computation and a computation with futures can be large even though the uniprocess execution incurs a small number of misses.

Theorem 1 *There is a family of computations*

$$\{G_n : n = kC, for\ k \in \mathcal{Z}^+\}$$

with $O(n)$ computational work, whose uniprocess execution incurs $3C$ misses while any 2-process execution of the computation incurs $\Omega(n)$ misses on a work stealer with a cache size of C, assuming that $S = O(C)$, where S is the maximum steal time.

Proof: Figure 4 shows the structure of a dag, G_{4C} for $n = 4C$. Each node except the root node represents a sequence of C instructions accessing a set of C distinct memory blocks. The root node represents $C+S$ instructions that accesses C distinct memory blocks. The graph has two symmetric components L_{4C} and R_{4C}, which corresponds to the left and the right subtree of the root excluding the leaves. We partition the nodes in G_{4C} into three classes, such that all nodes in a class access the same memory blocks while nodes from different classes access mutually disjoint set of memory blocks. The first class contains the root node only, the second class contains all the nodes in L_{4C}, and the third class contains the rest of the nodes, which are the nodes in R_{4C} and the leaves of G_{4C}. For general $n = kC$, G_n can be partitioned into L_n, R_n and the k leaves of G_n and the root similarly. Each of L_n and R_n contains $2\lceil \frac{k}{2} \rceil - 1$ nodes and has the structure of a complete binary tree with additional k leaves at the lowest level. There is a dependency edge from the leaves of both L_n and R_n to the leaves of G_n.

Consider a work stealer that executes the nodes of G_n in the order that they are numbered in a uniprocess execution. In the uniprocess execution, no node in L_n incurs a cache miss except the root node, since all nodes in L_n access the same memory blocks as the root of L_n. The same argument holds for R_n and the k leaves of G_n. Hence the execution of the nodes in L_n, R_n, and the leaves causes $2C$ misses. Since the root node causes C misses, the total number of misses in the uniprocess execution is $3C$. Now, consider a 2-process execution with the same work stealer and call the processes, process 0 and 1. At time step 1, process 0 starts executing the root node, which enables the root of R_n no later than time step m. Since process 0 starts stealing immediately and there are no other processes to steal from, process 1 steals and starts working on the root of R_n, no later than time step $m + S$. Hence, the root of R_n executes before the root of L_n and thus, all the nodes in L_n execute before the corresponding symmetric node in R_n. Therefore, for any leaf of G_n, the parent that is in R_n executes before the parent in L_n. Therefore a leaf node of G_n is executed immediately after its parent in L_n and thus, causes C cache misses. Thus, the total number of cache misses is $\Omega(kC) = \Omega(n)$. ■

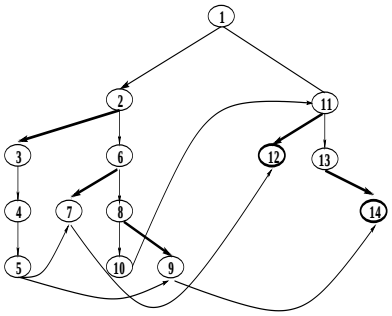

Figure 5: The structure for dag of a computation with futures that can incur a large cache overhead.

There exists computations similar to the computation in Figure 4 that generalizes Theorem 1 for arbitrary number of processes by making sure that all the processes but 2 steal throughout any multiprocess execution. Even in the general case, however, where the average parallelism is higher than the number of processes, Theorem 1 can be generalized with the same bound on expected number of cache misses by exploiting the symmetry in G_n and by assuming a symmetrically distributed steal-time. With a symmetrically distributed steal-time, for any ϵ, a steal that takes ϵ steps more than mean steal-time is equally likely to happen as a steal that takes ϵ less steps than the mean. Theorem 1 holds for computations with futures as well. Multithreaded computing with futures is a fairly restricted form of multithreaded computing compared to computing with events such as synchronization variables. The graph F in Figure 5 shows the structure of a dag, whose 2-process execution causes large number of cache misses. In a 2-process execution of F, the enabling parent of the leaf nodes in the right subtree of the root are in the left subtree and therefore the execution of each such leaf node causes C misses.

5 Nested-Parallel Computations

In this section, we show that the cache overhead of an execution of a nested-parallel computation with a work stealer is at most twice the product of the number of steals and the cache size. Our proof has two steps. First, we show that the cache overhead is bounded by the product of the cache size and the number of nodes that are executed "out of order" with respect to the uniprocess execution order. Second, we prove that the number of such out-of-order executions is at most twice the number of steals.

Consider a computation G and its P-process execution, X_P, with a work stealer and the uniprocess execution, X_1 with the same work stealer. Let v be a node in G and node u be the node that executes immediately before v in X_1. Then we say that v is ***drifted*** in X_P if node u is not executed immediately before v by the process that executes v in X_P.

Lemma 2 establishes a key property of an execution with simple caches.

Lemma 2 *Consider a process with a simple cache of C blocks. Let X_1 denote the execution of a sequence of instructions on the process starting with cache state S_1 and let X_2 denote the execution of the same sequence of instructions starting with cache state S_2. Then X_1 incurs at most C more misses than X_2.*

Proof: We construct a one-to-one mapping between the cache lines in X_1 and X_2 such that an instruction that accesses a line l_1 in X_1 accesses the entry l_2 in X_2, if and only if l_1 is mapped to l_2. Consider X_1 and let l_1 be a cache line. Let i be the first instruction that accesses or overwrites l_1. Let l_2 be the cache line that the same instruction accesses or overwrites in X_2 and map l_1 to l_2. Since the caches are simple, an instruction that overwrites l_1 in X_1 overwrites l_2 in X_2. Therefore the number of misses that overwrites l_1 in X_1 is equal to the number of misses that overwrites l_2 in X_2 after instruction i. Since i itself can cause 1 miss, the number of misses that overwrites l_1 in X_1 is at most 1 more than the number of misses that overwrites l_2 in X_2. We construct the mapping for each cache line in X_1 in the same way. Now, let us show that the mapping is one-to-one. For the sake of contradiction, assume that two cache lines, l_1 and l_2, in X_1 map to the same line in X_2. Let i_1 and i_2 be the first instructions accessing the cache lines in X_1 such that i_1 is executed before i_2. Since i_1 and i_2 map to the same line in X_2 and the caches are simple, i_2 accesses the line that i_1 accesses in X_1 but then $l_1 = l_2$, a contradiction. Hence, the total number of cache misses in X_1 is at most C more than the misses in X_2. ■

Theorem 3 *Let D denote the total number of drifted nodes in an execution of a nested-parallel computation with a work stealer on P processes, each of which has a simple cache with C words. Then the cache overhead of the execution is at most CD.*

Proof: Let X_P denote the P-process execution and let X_1 be the uniprocess execution of the same computation with the same work stealer. We divide the multiprocess computation into D pieces each of which can incur at most C more misses than in the uniprocess execution. Let u be a drifted node let q be the process that executes u. Let v be the next drifted node executed on q (or the final node of the computation). Let the ordered set O represent the execution order of all the nodes that are executed after u (u is included) and before v (v is excluded if it is drifted, included otherwise) on q in X_P. Then nodes in O are executed on the same process and in the same order in both X_1 and X_P.

Now consider the number of cache misses during the execution of the nodes in O in X_1 and X_P. Since the computation is nested parallel and therefore race free, a process that executes in parallel with q does not cause q to incur cache misses due to sharing. Therefore by Lemma 2 during the execution of the nodes in O the number of cache misses in X_P is at most C more than the number of misses in X_1. This bound holds for each of the D sequence of such instructions O corresponding to D drifted nodes. Since the sequence starting at the root node and ending at the first drifted node incurs the same number of misses in X_1 and X_P X_P takes at most CD more misses than X_1 and the cache overhead is at most CD. ■

Lemma 2 (and thus Theorem 3) does not hold for caches that are not simple. For example, consider the execution of a sequence of instructions on a cache with least-frequently-used replacement policy starting at two cache states. In the first cache state, the blocks that are frequently accessed by the instructions are in the cache with high frequencies, whereas in the second cache state, the blocks that are in the cache are not accessed by the instruction and have low frequencies. The execution with the second cache state, therefore, incurs many more misses than the size of the cache compared to the execution with the second cache state.

Now we show that the number of drifted nodes in an execution of a series-parallel computation with a work stealer is at most twice the number of steals. The proof is based on the representation of series-parallel computations as sp-dags. We call a node with out-degree of at least 2 a ***fork node*** and partition the nodes of an sp-dag except the root into three categories: join nodes, stable nodes and nomadic nodes . We call a node that has an in-degree of at least 2 a ***join node*** and partition all the nodes that have in-degree 1 into

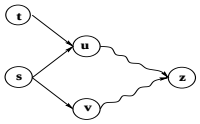

Figure 6: Children of s and their merger.

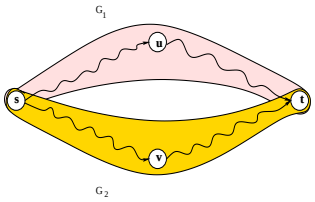

Figure 7: The joint embedding of u and v.

two classes: a ***nomadic*** node has a parent that is a fork node, and a ***stable*** node has a parent that has out-degree 1. The root node has in-degree 0 and it does not belong to any of these categories. Lemma 4 lists two fundamental properties of sp-dags; one can prove both properties by induction on the number of edges in an sp-dag.

Lemma 4 *Let G be an sp-dag. Then G has the following properties.*

1. *The least common ancestor of any two nodes in G is unique.*
2. *The greatest common descendant of any two nodes in G is unique and is equal to their unique merger.*

Lemma 5 *Let s be a fork node. Then no child of s is a join node.*

Proof: Let u and v denote two children of s and suppose u is a join node as in Figure 6. Let t denote some other parent of u and z denote the unique merger of u and v. Then both z and u are mergers for s and t, which is a contradiction of Lemma 5. Hence u is not a join node. ■

Corollary 6 *Only nomadic nodes can be stolen in an execution of a series-parallel computation by the work-stealing algorithm.*

Proof: Let u be a stolen node in an execution. Then u is pushed on a deque and thus the enabling parent of u is a fork node. By Lemma 5, u is not a join node and has an incoming degree 1. Therefore u is nomadic. ■

Consider a series-parallel computation and let G be its sp-dag. Let u and v be two independent nodes in G and let s and t denote their least common ancestor and greatest common descendant respectively as shown in Figure 7. Let G_1 denote the graph that is induced by the relatives of u that are descendants of s and also ancestors of t. Similarly, let G_2 denote the graph that is induced by the relatives of v that are descendants of s and ancestors of t. Then we call G_1 the ***embedding*** of u with respect to v and G_2 the embedding of v with respect to u. We call the graph that is the union of G_1 and G_2 the ***joint embedding*** of u and v with source s and sink t. Now consider an execution of G and y and z be the children of s such that y is executed before z. Then we call y the ***leader*** and z the ***guard*** of the joint embedding.

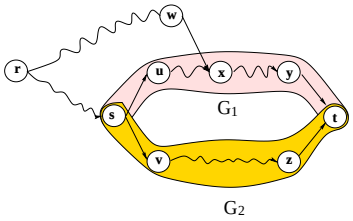

Figure 8: The join node s is the least common ancestor of y and z. Node u and v are the children of s.

Lemma 7 *Let $G(V, E)$ be an sp-dag and let y and z be two parents of a join node t in G. Let G_1 denote the embedding of y with respect to z and G_2 denote the embedding of z with respect to y. Let s denote the source and t denote the sink of the joint embedding. Then the parents of any node in G_1 except for s and t is in G_1 and the parents of any node in G_2 except for s and t is in G_2.*

Proof: Since y and z are independent, both of s and t are different from y and z (see Figure 8). First, we show that there is not an edge that starts at a node in G_1 except at s and ends at a node in G_2 except at t and vice versa. For the sake of contradiction, assume there is an edge (m, n) such that $m \neq s$ is in G_1 and $n \neq t$ is in G_2. Then m is the least common ancestor of y and z; hence no such (m, n) exists. A similar argument holds when m is in G_2 and n is in G_1.

Second, we show that there does not exists an edge that originates from a node outside of G_1 or G_2 and ends at a node at G_1 or G_2. For the sake of contradiction, let (w, x) be an edge such that x is in G_1 and w is not in G_1 or G_2. Then x is the unique merger for the two children of the least common ancestor of w and s, which we denote with r. But then t is also a merger for the children of r. The children of r are independent and have a unique merger, hence there is no such edge (w, x). A similar argument holds when x is in G_2. Therefore we conclude that the parents of any node in G_1 except s and t is in G_1 and the parents of any node in G_2 except s and t is in G_2. ■

Lemma 8 *Let G be an sp-dag and let y and z be two parents of a join node t in G. Consider the joint embedding of y and z and let u be the guard node of the embedding. Then y and z are executed in the same respective order in a multiprocess execution as they are executed in the uniprocess execution if the guard node u is not stolen.*

Proof: Let s be the source, t the sink, and v the leader of the joint embedding. Since u is not stolen, v is not stolen. Hence, by Lemma 7, before it starts working on u, the process that executes s executed v and all its descendants in the embedding except for t Hence, z is executed before u and y is executed after u as in the uniprocess execution. Therefore, y and z are executed in the same respective order as they execute in the uniprocess execution. ■

Lemma 9 *A nomadic node is drifted in an execution only if it is stolen.*

Proof: Let u be a nomadic and drifted node. Then, by Lemma 5, u has a single parent s that enables u. If u is the first child of s to execute in the uniprocess execution then u is not drifted in the multiprocess execution. Hence, u is not the first child to execute. Let v be the last child of s that is executed before u in the uniprocess execution. Now, consider the multiprocess execution and let q be the

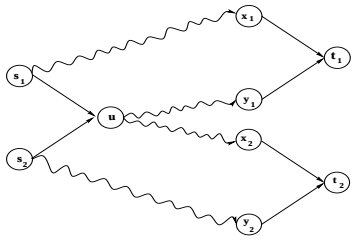

Figure 9: Nodes t_1 and t_2 are two join nodes with the common guard u.

process that executes v. For the sake of contradiction, assume that u is not stolen. Consider the joint embedding of u and v as shown in Figure 8. Since all parents of the nodes in G_2 except for s and t are in G_2 by Lemma 7, q executes all the nodes in G_2 before it executes u and thus, z precedes u on q. But then u is not drifted, because z is the node that is executed immediately before u in the uniprocess computation. Hence u is stolen. ■

Let us define the ***cover*** of a join node t in an execution as the set of all the guard nodes of the joint embedding of all possible pairs of parents of t in the execution. The following lemma shows that a join node is drifted only if a node in its cover is stolen.

Lemma 10 *A join node is drifted in an execution only if a node in its cover is stolen in the execution.*

Proof: Consider the execution and let t be a join node that is drifted. Assume, for the sake of contradiction, that no node in the cover of t, $C(t)$, is stolen. Let y and z be any two parents of t as in Figure 8. Then y and z are executed in the same order as in the uniprocess execution by Lemma 8. But then all parents of t execute in the same order as in the uniprocess execution. Hence, the enabling parent of t in the execution is the same as in the uniprocess execution. Furthermore, the enabling parent of t has out-degree 1, because otherwise t is not a join node by Lemma 5 and thus, the process that enables t executes t. Therefore, t is not drifted. A contradiction, hence a node in the cover of t is stolen. ■

Lemma 11 *The number of drifted nodes in an execution of a series-parallel computation is at most twice the number of steals in the execution.*

Proof: We associate each drifted node in the execution with a steal such that no steal has more than 2 drifted nodes associated with it. Consider a drifted node, u. Then u is not the root node of the computation and it is not stable either. Hence, u is either a nomadic or join node. If u is nomadic, then u is stolen by Lemma 9 and we associate u with the steal that steals u. Otherwise, u is a join node and there is a node in its cover $C(u)$ that is stolen by Lemma 10. We associate u with the steal that steals a node in its cover. Now, assume there are more than 2 nodes associated with a steal that steals node u. Then there are at least two join nodes t_1 and t_2 that are associated with u. Therefore, node u is in the joint embedding of two parents of t_1 and also t_2. Let x_1, y_1 be these parents of t_1 and x_2, y_2 be the parents of t_2, as shown in Figure 9. But then u has parent that is a fork node and is a joint node, which contradicts Lemma 5. Hence no such u exists. ■

Theorem 12 *The cache overhead of an execution of a nested-parallel computation with simple caches is at most twice the product of the number of misses in the execution and the cache size.*

Proof: Follows from Theorem 3 and Lemma 11. ■

6 An Analysis of Nonblocking Work Stealing

The non-blocking implementation of the work-stealing algorithm delivers provably good performance under traditional and multi-programmed workloads. A description of the implementation and its analysis is presented in [2]; an experimental evaluation is given in [10]. In this section, we extend the analysis of the non-blocking work-stealing algorithm for classical workloads and bound the execution time of a nested-parallel, computation with a work stealer to include the number of cache misses, the cache-miss penalty and the steal time. First, we bound the number of steal attempts in an execution of a general computation by the work-stealing algorithm. Then we bound the execution time of a nested-parallel computation with a work stealer using results from Section 5. The analysis that we present here is similar to the analysis given in [2] and uses the same potential function technique.

We associate a nonnegative potential with nodes in a computation' s dag and show that the potential decreases as the execution proceeds. We assume that a node in a computation dag has out-degree at most 2. This is consistent with the assumption that each node represents on instruction. Consider an execution of a computation with its dag, $G(V, E)$ with the work-stealing algorithm. The execution grows a tree, the ***enabling tree***, that contains each node in the computation and its enabling edge. We define the ***distance*** of a node $u \in V$, $d(u)$, as $T_\infty - depth(u)$, where $depth(u)$ is the depth of u in the enabling tree of the computation. Intuitively, the distance of a node indicates how far the node is away from end of the computation. We define the potential function in terms of distances. At any given step i, we assign a positive potential to each ready node, all other nodes have 0 potential. A node is ***ready*** if it is enabled and not yet executed to completion. Let u denote a ready node at time step i. Then we define, $\phi_i(u)$, the potential of u at time step i as

$$\phi_i(u) = \begin{cases} 3^{2d(u)-1} & \text{if } u \text{ is assigned;} \\ 3^{2d(u)} & \text{otherwise.} \end{cases}$$

The potential at step i, Φ_i, is the sum of the potential of each ready node at step i. When an execution begins, the only ready node is the root node which has distance T_∞ and is assigned to some process, so we start with $\Phi_0 = 3^{2T_\infty - 1}$. As the execution proceeds, nodes that are deeper in the dag become ready and the potential decreases. There are no ready nodes at the end of an execution and the potential is 0.

Let us give a few more definitions that enable us to associate a potential with each process. Let $R_i(q)$ denote the set of ready nodes that are in the deque of process q along with q's assigned node, if any, at the beginning of step i. We say that each node u in $R_i(q)$ ***belongs*** to process q. Then we define the potential of q's deque as

$$\Phi_i(q) = \sum_{u \in R_i(q)} \phi_i(u) \ .$$

In addition, let A_i denote the set of processes whose deque is empty at the beginning of step i, and let D_i denote the set of all other processes. We partition the potential Φ_i into two parts

$$\Phi_i = \Phi_i(A_i) + \Phi_i(D_i) \ ,$$

where

$$\Phi_i(A_i) = \sum_{q \in A_i} \Phi_i(q) \qquad \text{and} \qquad \Phi_i(D_i) = \sum_{q \in D_i} \Phi_i(q) \ ,$$

and we analyze the two parts separately.

Lemma 13 lists four basic properties of the potential that we use frequently. The proofs for these properties are given in [2] and the listed properties are correct independent of the time that execution of a node or a steal takes. Therefore, we give a short proof sketch.

Lemma 13 *The potential function satisfies the following properties.*

1. *Suppose node u is assigned to a process at step i. Then the potential decreases by at least $(2/3)\phi_i(u)$.*
2. *Suppose a node u is executed at step i. Then the potential decreases by at least $(5/9)\phi_i(u)$ at step i.*
3. *Consider any step i and any process q in D_i. The topmost node u in q's deque contributes at least $3/4$ of the potential associated with q. That is, we have $\phi_i(u) \geq (3/4)\Phi_i(q)$.*
4. *Suppose a process p chooses process q in D_i as its victim at time step i (a steal attempt of p targeting q occurs at step i). Then the potential decreases by at least $(1/2)\Phi_i(q)$ due to the assignment or execution of a node belonging to q at the end of step i.*

Property 1 follows directly from the definition of the potential function. Property 2 holds because a node enables at most two children with smaller potential, one of which becomes assigned. Specifically, the potential after the execution of node u decreases by at least $\phi(u)(1-\frac{1}{3}-\frac{1}{9}) = \frac{5}{9}\phi(u)$. Property 3 follows from a structural property of the nodes in a deque. The distance of the nodes in a process' deque decrease monotonically from the top of the deque to bottom. Therefore, the potential in the deque is the sum of geometrically decreasing terms and dominated by the potential of the top node. The last property holds because when a process chooses process q in D_i as its victim, the node at the top of q' s deque is assigned at the next step. Therefore, the potential decreases by $2/3\phi_i(u)$ by property 1. Moreover, $\phi_i(u) \geq (3/4)\Phi_i(q)$ by property 3 and the result follows.

Lemma 16 shows that the potential decreases as a computation proceeds. The proof for Lemma 16 utilizes balls and bins game bound from Lemma 14.

Lemma 14 (Balls and Weighted Bins) *Suppose that at least P balls are thrown independently and uniformly at random into P bins, where bin i has a weight W_i, for $i = 1, \ldots, P$. The total weight is $W = \sum_{i=1}^{P} W_i$. For each bin i, define the random variable X_i as*

$$X_i = \begin{cases} W_i & \text{if some ball lands in bin } i; \\ 0 & \text{otherwise.} \end{cases}$$

If $X = \sum_{i=1}^{P} X_i$, then for any β in the range $0 < \beta < 1$, we have $\Pr\{X \geq \beta W\} > 1 - 1/((1-\beta)e)$.

This lemma can be proven with an application of Markov' s inequality. The proof of a weaker version of this lemma for the case of exactly P throws is similar and given in [2]. Lemma 14 also follows from the weaker lemma because X does not decrease with more throws.

We now show that whenever P or more steal attempts occur, the potential decreases by a constant fraction of $\Phi_i(D_i)$ with constant probability.

Lemma 15 *Consider any step i and any later step j such that at least P steal attempts occur at steps from i (inclusive) to j (exclusive). Then we have*

$$\Pr\left\{\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i(D_i)\right\} > \frac{1}{4}\,.$$

Moreover the potential decrease is because of the execution or assignment of nodes belonging to a process in D_i.

Proof: Consider all P processes and P steal attempts that occur at or after step i. For each process q in D_i, if one or more of the P attempts target q as the victim, then the potential decreases by $(1/2)\Phi_i(q)$ due to the execution or assignment of nodes that belong to q by property 4 in Lemma 13. If we think of each attempt as a ball toss, then we have an instance of the Balls and Weighted Bins Lemma (Lemma 14). For each process q in D_i, we assign a weight $W_q = (1/2)\Phi_i(q)$, and for each other process q in A_i, we assign a weight $W_q = 0$. The weights sum to $W = (1/2)\Phi_i(D_i)$. Using $\beta = 1/2$ in Lemma 14, we conclude that the potential decreases by at least $\beta W = (1/4)\Phi_i(D_i)$ with probability greater than $1 - 1/((1-\beta)e) > 1/4$ due to the execution or assignment of nodes that belong to a process in D_i. ■

We now bound the number of steal attempts in a work-stealing computation.

Lemma 16 *Consider a P-process execution of a multithreaded computation with the work-stealing algorithm. Let T_1 and T_∞ denote the computational work and the critical path of the computation. Then the expected number of steal attempts in the execution is $O(\lceil \frac{m}{s} \rceil P T_\infty)$. Moreover, for any $\varepsilon > 0$, the number of steal attempts is $O(\lceil \frac{m}{s} \rceil P T_\infty + \lg(1/\varepsilon))$ with probability at least $1 - \varepsilon$.*

Proof: We analyze the number of steal attempts by breaking the execution into ***phases*** of $\lceil \frac{m}{s} \rceil P$ steal attempts. We show that with constant probability, a phase causes the potential to drop by a constant factor. The first phase begins at step $t_1 = 1$ and ends at the first step t_1' such that at least $\lceil \frac{m}{s} \rceil P$ steal attempts occur during the interval of steps $[t_1, t_1']$. The second phase begins at step $t_2 = t_1' + 1$, and so on. Let us first show that there are at least m steps in a phase. A process has at most 1 outstanding steal attempt at any time and a steal attempt takes at least s steps to complete. Therefore, at most P steal attempts occur in a period of s time steps. Hence a phase of steal attempts takes at least $\lceil (\lceil \frac{m}{s} \rceil) P)/P \rceil \cdot s \geq m$ time units.

Consider a phase beginning at step i, and let j be the step at which the next phase begins. Then $i + m \leq j$. We will show that we have $\Pr\{\Phi_j \leq (3/4)\Phi_i\} > 1/4$. Recall that the potential can be partitioned as $\Phi_i = \Phi_i(A_i) + \Phi_i(D_i)$. Since the phase contains $\lceil \frac{m}{s} \rceil P$ steal attempts, $\Pr\{\Phi_i - \Phi_j \geq (1/4)\Phi_i(D_i)\} > 1/4$ due to execution or assignment of nodes that belong to a process in D_i, by Lemma 15. Now we show that the potential also drops by a constant fraction of $\Phi_i(A_i)$ due to the execution of assigned nodes that are assigned to the processes in A_i. Consider a process, say q in A_i. If q does not have an assigned node, then $\Phi_i(q) = 0$. If q has an assigned node u, then $\Phi_i(q) = \phi_i(u)$. In this case, process q completes executing node u at step $i + m - 1 < j$ at the latest and the potential drops by at least $(5/9)\phi_i(u)$ by property 2 of Lemma 13. Summing over each process q in A_i, we have $\Phi_i - \Phi_j \geq (5/9)\Phi_i(A_i)$. Thus, we have shown that the potential decreases at least by a quarter of $\Phi_i(A_i)$ and $\Phi_i(D_i)$. Therefore no matter how the total potential is distributed over A_i and D_i, the total potential decreases by a quarter with probability more than $1/4$, that is, $\Pr\{\Phi_i - \Phi_j \geq (1/4)\Phi_i\} > 1/4$.

We say that a phase is ***successful*** if it causes the potential to drop by at least a $1/4$ fraction. A phase is successful with probability at least $1/4$. Since the potential starts at $\Phi_0 = 3^{2T_\infty - 1}$ and ends at 0 (and is always an integer), the number of successful phases is at most $(2T_\infty - 1)\log_{4/3} 3 < 8T_\infty$. The expected number of phases needed to obtain $8T_\infty$ successful phases is at most $32T_\infty$. Thus, the expected number of phases is $O(T_\infty)$, and because each phase contains $\lceil \frac{m}{s} \rceil P$ steal attempts, the expected number of steal attempts is $O(\lceil \frac{m}{s} \rceil P T_\infty)$. The high probability bound follows by an application of the Chernoff bound. ■

Theorem 17 *Let $M_P(C)$ be the number of cache misses in a P-process execution of a nested-parallel computation with a work-stealer that has simple caches of C blocks each. Let $M_1(C)$ be the number of cache misses in the uniprocess execution Then*

$$M_P(C) = M_1(C) + O(\lceil \frac{m}{s} \rceil C P T_\infty + \lceil \frac{m}{s} \rceil C P \ln(1/\varepsilon))$$

with probability at least $1-\varepsilon$. The expected number of cache misses is

$$M_1(C) + O(\lceil \frac{m}{s} \rceil C P T_\infty)$$

Proof: Theorem 12 shows that the cache overhead of a nested-parallel computation is at most twice the product of the number of steals and the cache size. Lemma 16 shows that the number of steal attempts is $O(\lceil \frac{m}{s} \rceil P(T_\infty + \ln(1/\varepsilon)))$ with probability at least $1-\varepsilon$ and the expected number of steals is $O(\lceil \frac{m}{s} \rceil P T_\infty)$. The number of steals is not greater than the number of steal attempts. Therefore the bounds follow. ■

Theorem 18 *Consider a P-process, nested-parallel, work-stealing computation with simple caches of C blocks. Then, for any $\varepsilon > 0$, the execution time is*

$$O(\frac{T_1(C)}{P} + m\lceil \frac{m}{s} \rceil C (T_\infty + \ln(1/\varepsilon)) + (m+s)(T_\infty + \ln(1/\varepsilon)))$$

with probability at least $(1-\varepsilon)$. Moreover, the expected running time is

$$O(\frac{T_1(C)}{P} + m\lceil \frac{m}{s} \rceil C T_\infty + (m+s)T_\infty) .$$

Proof: We use an accounting argument to bound the running time. At each step in the computation, each process puts a dollar into one of two buckets that matches its activity at that step. We name the two buckets as the work and the steal bucket. A process puts a dollar into the work bucket at a step if it is working on a node in the step. The execution of a node in the dag adds either 1 or m dollars to the work bucket. Similarly, a process puts a dollar into the steal bucket for each step that it spends stealing. Each steal attempt takes $O(s)$ steps. Therefore, each steal adds $O(s)$ dollars to the steal bucket. The number of dollars in the work bucket at the end of execution is at most $O(T_1 + (m-1) M_P(C))$, which is

$$O(T_1(C) + (m-1) \left\lceil \frac{m}{s} \right\rceil C P (T_\infty + \ln(1/\varepsilon')))$$

with probability at least $1-\varepsilon'$.

The total number of dollars in steal bucket is the total number of steal attempts multiplied by the number of dollars added to the steal bucket for each steal attempt, which is $O(s)$. Therefore total number of dollars in the steal bucket is

$$O(s \left\lceil \frac{m}{s} \right\rceil P (T_\infty + \ln(1/\varepsilon')))$$

with probability at least $1-\varepsilon'$. Each process adds exactly one dollar to a bucket at each step so we divide the total number of dollars by P to get the high probability bound in the theorem. A similar argument holds for the expected time bound. ■

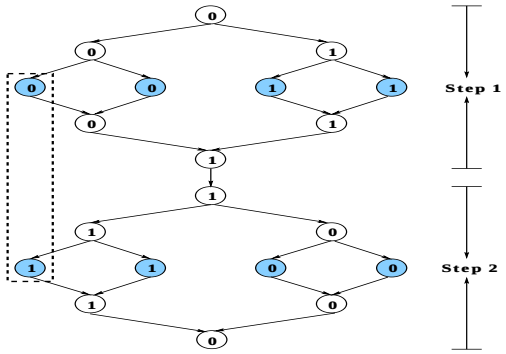

Figure 10: The tree of threads created in a data-parallel work-stealing application.

7 Locality-Guided Work Stealing

The work-stealing algorithm achieves good data locality by executing nodes that are close in the computation graph on the same process. For certain applications, however, regions of the program that access the same data are not close in the computational graph. As an example, consider an application that takes a sequence of steps each of which operates in parallel over a set or array of values. We will call such an application an ***iterative data-parallel application***. Such an application can be implemented using work-stealing by forking a tree of threads on each step, in which each leaf of the tree updates a region of the data (typically disjoint). Figure 10 shows an example of the trees of threads created in two steps. Each node represents a thread and is labeled with the process that executes it. The gray nodes are the leaves. The threads synchronize in the same order as they fork. The first and second steps are structurally identical, and each pair of corresponding gray nodes update the same region, often using much of the same input data. The dashed rectangle in Figure 10, for example, shows a pair of such gray nodes. To get good locality for this application, threads that update the same data on different steps ideally should run on the same processor, even though they are not "close" in the dag. In work stealing, however, this is highly unlikely to happen due to the random steals. Figure 10, for example, shows an execution where all pairs of corresponding gray nodes run on different processes.

In this section, we describe and evaluate ***locality-guided work stealing***, a heuristic modification to work stealing which is designed to allow locality between nodes that are distant in the computational graph. In locality-guided work stealing, each thread can be given an affinity for a process, and when a process obtains work it gives priority to threads with affinity for it. To enable this, in addition to a deque each process maintains a ***mailbox***: a first-in-first-out (FIFO) queue of pointers to threads that have affinity for the process. There are then two differences between the locality-guided work-stealing and work-stealing algorithms. First, when creating a thread, a process will push the thread onto both the deque, as in normal work stealing, and also onto the tail of the mailbox of the process that the thread has affinity for. Second, a process will first try to obtain work from its mailbox before attempting a steal. Because threads can appear twice, once in a mailbox and once on a deque, there needs to be some form of synchronization between the two copies to make sure the thread is not executed twice.

A number of techniques that have been suggested to improve the data locality of multithreaded programs can be realized by the locality-guided work-stealing algorithm together with an appropriate policy to determine the affinities of threads. For example, an

initial distribution of work among processes can be enforced by setting the affinities of a thread to the process that it will be assigned at the beginning of the computation. We call this locality-guided work-stealing with ***initial placements***. Likewise, techniques that rely on hints from the programmer can be realized by setting the affinity of threads based on the hints. In the next section, we describe an implementation of locality-guided work stealing for iterative data-parallel applications. The implementation described can be modified easily to implement other techniques mentioned.

7.1 Implementation

We built locality-guided work stealing into Hood. Hood is a multithreaded programming library with a nonblocking implementation of work stealing that delivers provably good performance under both traditional and multiprogrammed workloads [2, 10, 30].

In Hood, the programmer defines a thread as a C++ class, which we refer to as the ***thread definition***. A thread definition has a method named `run` that defines the code that the thread executes. The `run` method is a C++ function which can call Hood library functions to create and synchronize with other threads. A ***rope*** is an object that is an instance of a thread definition class. Each time the `run` method of a rope is executed, it creates a new thread. A rope can have an affinity for a process, and when the Hood run-time system executes such a rope, the system passes this affinity to the thread. If the thread does not run on the process for which it has affinity, the affinity of the rope is updated to the new process.

Iterative data-parallel applications can effectively use ropes by making sure all "corresponding" threads (threads that update the same region across different steps) are generated from the same rope. A thread will therefore always have an affinity for the process on which it' s corresponding thread ran on the previous step. The dashed rectangle in Figure 10, for example, represents two threads that are generated in two executions of one rope. To initialize the ropes, the programmer needs to create a tree of ropes before the first step. This tree is then used on each step when forking the threads.

To implement locality-guided work stealing in Hood, we use a nonblocking queue for each mailbox. Since a thread is put to a mailbox and to a deque, one issue is making sure that the thread is not executed twice, once from the mailbox and once from the deque. One solution is to remove the other copy of a thread when a process starts executing it. In practice, this is not efficient because it has a large synchronization overhead. In our implementation, we do this lazily: when a process starts executing a thread, it sets a flag using an atomic update operation such as test-and-set or compare-and-swap to mark the thread. When executing a thread, a process identifies a marked thread with the atomic update and discards the thread. The second issue comes up when one wants to reuse the thread data structures, typically those from the previous step. When a thread' s structure is reused in a step, the copies from the previous step, which can be in a mailbox or a deque needs to be marked invalid. One can implement this by invalidating all the multiple copies of threads at the end of a step and synchronizing all processes before the next step start. In multiprogrammed workloads, however, the kernel can swap a process out, preventing it from participating to the current step. Such a swapped out process prevents all the other processes from proceeding to the next step. In our implementation, to avoid the synchronization at the end of each step, we time-stamp thread data structures such that each process closely follows the time of the computation and ignores a thread that is "out-of-date".

Benchmark	Work (T_1)	Overhead ($\frac{T_1}{T_s}$)	Critical Path Length (T_∞)	Average Par. ($\frac{T_1}{T_\infty}$)
staticHeat	15.95	1.10		
heat	16.25	1.12	0.045	361.11
lgHeat	16.37	1.12	0.044	372.05
ipHeat	16.37	1.12	0.044	372.05
staticRelax	44.15	1.08		
relax	43.93	1.08	0.039	1126.41
lgRelax	44.22	1.08	0.039	1133.84
ipRelax	44.22	1.08	0.039	1133.84

Table 1: Measured benchmark characteristics. We compiled all applications with Sun `CC` compiler using `-xarch=v8plus -O5 -dalign` flags. All times are given in seconds. T_s denotes the execution time of the sequential algorithm for the application and T_s is 14.54 for `Heat` and 40.99 for `Relax`.

7.2 Experimental Results

In this section, we present the results of our preliminary experiments with locality-guided work stealing on two small applications. The experiments were run on a 14 processor Sun Ultra Enterprise with 400 MHz processors and 4M byte L2 cache each, and running Solaris 2.7. We used the `processor_bind` system call of Solaris 2.7 to bind processes to processors to prevent Solaris kernel from migrating a process among processors, causing the process to loose its cache state. When the number of processes is less than number of processors we bind one process to each processor, otherwise we bind processes to processors such that processes are distributed among processors as evenly as possible.

We use the applications `Heat` and `Relax` in our evaluation. `Heat` is a Jacobi over-relaxation that simulates heat propagation on a 2 dimensional grid for a number of steps. This benchmark was derived from similar Cilk [27] and SPLASH [35] benchmarks. The main data structures are two equal-sized arrays. The algorithm runs in steps each of which updates the entries in one array using the data in the other array, which was updated in the previous step. `Relax` is a Gauss-Seidel over-relaxation algorithm that iterates over one a 1 dimensional array updating each element by a weighted average of its value and that of its two neighbors. We implemented each application with four strategies, static partitioning, work stealing, locality-guided work stealing, and locality guided work stealing with initial placements. The static partitioning benchmarks divide the total work equally among the number of processes and makes sure that each process accesses the same data elements in all the steps. It is implemented directly with Solaris threads. The three work-stealing strategies are all implemented in Hood. The plain work-stealing version uses threads directly, and the two locality-guided versions use ropes by building a tree of ropes at the beginning of the computation. The initial placement strategy assigns initial affinities to the ropes near the top of the tree to achieve a good initial load balance. We use the following prefixes in the names of the benchmarks: `static` (static partitioning), none, (work stealing), `lg` (locality guided work stealing), and `lg` (lg with initial placement).

We ran all `Heat` benchmarks with `-x 8K -y 128 -s 100` parameters. With these parameters each `Heat` benchmark allocates two arrays of double precision floating point numbers of 8192 columns and 128 rows and does relaxation for 100 steps. We ran all `Relax` benchmarks with the parameters `-n 3M -s 100`. With these parameters each `Relax` benchmark allocates one array of 3 million double-precision floating points numbers and does relaxation for 100 steps. With the specified input parameters, a `Relax`

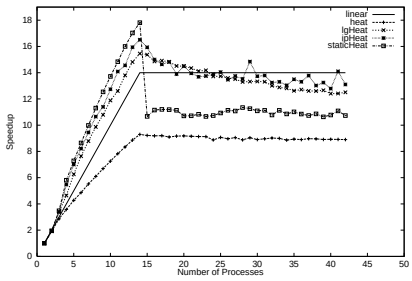

Figure 11: Speedup of `heat` benchmarks on 14 processors.

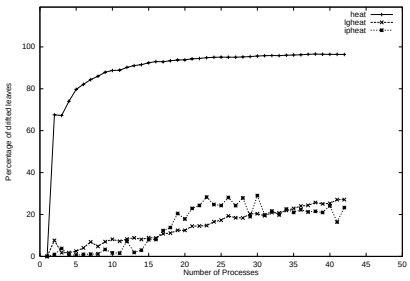

Figure 12: Percentage of bad updates for the `Heat` benchmarks.

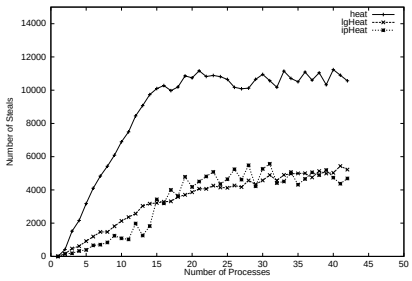

Figure 13: Number of steals in the `Heat` benchmarks.

benchmark allocates 16 Megabytes and a `Heat` benchmark allocates 24 Megabytes of memory for the main data structures. Hence, the main data structures for `Heat` benchmarks fit into the collective L2 cache space of 4 or more processes and the data structures for `Relax` benchmarks fit into that of 6 or more processes. The data for no benchmark fits into the collective L1 cache space of the Ultra Enterprise. We observe superlinear speedups with some of our benchmarks when the collective caches of the processes hold a significant amount of frequently accessed data. Table 1 shows characteristics of our benchmarks. Neither the work-stealing benchmarks nor the locality-guided work-stealing benchmarks have significant overhead compared to the serial implementation of the corresponding algorithms.

Figures 11 and Figure 1 show the speedup of the `Heat` and `Relax` benchmarks, respectively, as a function of the number of processes. The static partitioning benchmarks deliver superlinear speedups under traditional workloads but suffer from the performance cliff problem and deliver poor performance under multiprogramming workloads. The work-stealing benchmarks deliver poor performance with almost any number of processes. the locality-guided work-stealing benchmarks with or without initial placements, however, matches the static partitioning benchmarks under traditional workloads and delivers superior performance under multiprogramming workloads. The initial placement strategy improves the performance under traditional work loads, but it does not perform consistently better under multiprogrammed workloads. This is an artifact of binding processes to processors. The initial placement strategy distributes the load among the processes equally at the beginning of the computation but binding creates a load imbalance between processors and increases the number of steals. Indeed, the benchmarks that employ the initial-placement strategy does worse only when the number of processes is slightly greater than the number of processors.

The locality-guided work-stealing delivers good performance by achieving good data locality. To substantiate this, we counted the average number of times that an element is updated by two different processes in two consecutive steps, which we call a bad update. Figure 12 shows the percentage of bad updates in our `Heat` benchmarks with work stealing and locality-guided work-stealing. The work-stealing benchmarks incur a high percentage of bad updates, whereas the locality-guided work-stealing benchmarks achieve a very low percentage. Figure 13 shows the number of random steals for the same benchmarks for varying number of processes. The graph is similar to the graph for bad updates, because it is the random steals that causes the bad updates. The figures for the `Relax` application are similar.

References

[1] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998.

[3] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):113–121, August 1996.

[4] Guy Blelloch and Margaret Reid-Miller. Pipelining with futures. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 249–259, Newport, RI, June 1997.

[5] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[6] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–12, Santa Barbara, California, July 1995.

[7] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[10] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. Technical Report TR-98-13, The University of Texas at Austin, Department of Computer Sciences, May 1998.

[11] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, October 1981.

[12] David Callahan and Burton Smith. A future-based parallel language for a general-purpose highly-parallel computer. In David Padua, David Gelernter, and Alexandru Nicolau, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 95–113. MIT Press, 1990.

[13] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with OLDEN (parallel programming). In *Proceedings 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20. Springer-Verlag, August 1993.

[14] Rohit Chandra, Anoop Gupta, and John Hennessy. COOL: A Language for Parallel Programming. In David Padua, David Gelernter, and Alexandru Nicolau, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 126–148. MIT Press, 1990.

[15] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 249–259, San Diego, California, May 1993.

[16] D. E. Culler and G. Arvind. Resource requirements of dataflow programs. In *Proceedings of the International Symposium on Computer Architecture*, pages 141–151, 1988.

[17] Dawson R. Engler, David K. Lowenthal, and Gregory R. Andrews. Shared Filaments: Efficient fine-grain parallelism on shared-memory multiprocessors. Technical Report TR 93-13a, Department of Computer Science, The University of Arizona, April 1993.

[18] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

[19] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 1984.

[20] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[21] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[22] Vijay Karamcheti and Andrew A. Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *Proceedings of ACM/IEEE SC98: 10th Anniversary. High Performance Networking and Computing Conference*, 1998.

[23] Richard M. Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC)*, pages 290–300, Chicago, Illinois, May 1988.

[24] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[25] David A. Krantz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[26] Evangelos Markatos and Thomas LeBlanc. Locality-based scheduling for shared-memory multiprocessors. Technical Report TR-094, Institute of Computer Science, F.O.R.T.H., Crete, Greece, 1994.

[27] MIT Laboratory for Computer Science. *Cilk 5.2 Reference Manual*, July 1998.

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

[29] Grija J. Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1999.

[30] Dionysios Papadopoulos. Hood: A user-level thread library for multiprogrammed multiprocessors. Master's thesis, Department of Computer Sciences, University of Texas at Austin, August 1998.

[31] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread scheduling for cache locality. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 60–71, Cambridge, Massachusetts, October 1996.

[32] Dan Stein and Devang Shah. Implementing lightweight threads. In *Proceedings of the USENIX 1992 Summer Conference*, San Antonio, Texas, June 1992.

[33] Jacobo Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, December 1978.

[34] B. Weissman. Performance counters and state sharing annotations: a unified aproach to thread locality. In *International Conference on Architectural Support for Programming Languages and Operating Systems.*, pages 262–273, October 1998.

[35] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[36] Y. Zhang and A. Ortynski. The efficiency of randomized parallel backtrack search. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, October 1994.

[37] Yanjun Zhang. *Parallel Algorithms for Combinatorial Search Problems*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, November 1989. Also: University of California at Berkeley, Computer Science Division, Technical Report UCB/CSD 89/543.