On the Benefits of Work Stealing in Shared-Memory Multiprocessors

Daniel Neill and Adam Wierman
Department of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
{neill,acw}@cs.cmu.edu

Abstract

Load balancing is one of the key techniques exploited to improve the performance of parallel programs. However, load balancing is a difficult task for the programmer. Work stealing is an architectural mechanism that provides improved performance by instantaneously balancing the load among processors in a multiprocessor system. In this work, we develop a queueing model of a shared-memory multiprocessor system in order to show that work stealing can ease the burden of the programmer by eliminating the need to manually load balance.

1 Introduction

Load balancing is one of the key techniques exploited to improve the performance of parallel programs. In this work, we focus on applying load balancing to a wide range of parallel programs including fork-and-join programs, divide-and-conquer programs, and search programs. Across all these types, the most difficult task for the parallel programmer is partitioning the program onto the multiprocessor system so that the computation load is balanced among the processors. Thus, it is important for the underlying architecture to provide help to the programmer in order to ease this burden.

In partitioning the program tasks onto the multiserver system, the programmer typically statically assigns tasks to the multiprocessor system. Thus, any load balancing performed by the programmer must be performed at a fairly coarse level — specifically, the programmer can only hope to balance the average loads on each processor. Optimal performance, however, can only come by balancing the instantaneous load at each processor. One important architectural technique commonly employed to attempt to accomplish this instantaneous load balancing is *work stealing*. Work stealing allows idle processors to steal tasks from the queues of processors that are overloaded. In order to understand the benefits of work stealing, consider the following example. Suppose that, as the programmer, we have succeeded in balancing the average load among the processors of our system. There will, however, be some cases where one processor will be forced to do a lot of computation while another is left with very little computation. For example, random events occurring in one spatial region of a simulation, but not in others (e.g. a gust of wind hitting one part of an aircraft's wing) may differentially increase some processors' workloads. In these cases, work stealing allows an idle processor to perform some of the tasks in an overloaded processor's queue. This seems like a win-win situation; however some real world constraints complicate the issue. In particular, there are costs associated with stealing a job from another processor's queue. First, there are the communication costs involved in transferring the job - these costs come from both the extra contention for the system bus and the latency of the transfer. Second is the fact that jobs have some affinity for the queue where they are assigned. This affinity results from the fact that most multiprocessor systems have *non-uniform memory accesses* (NUMA) and thus accessing a processor's local memory is much faster than using memory that is local to some other processor. Further, each processor has a local cache. When we steal a task from another processor we can no longer take advantage of locally cached computations, and this results in an initial burst of cache misses while the cache is reloaded. These two costs raise the question of whether work stealing will improve the performance of the microprocessor system. In particular, we ask:

> If the programmer has done a good job of load balancing the workload, can work stealing still help performance?

In the absence of extreme communication costs or very strong affinity of jobs for the processors they are assigned to, we show that work stealing results in improved performance as a consequence of the fact that it balances the load on a more instantaneous level than the user is able to. This motivates another very interesting question:

> Can work stealing compensate for an imbalanced load, and thus reduce the burden placed on the programmer?

In other words, can work stealing substitute for manual load balancing? We show that, if the architecture performs work stealing and the programmer does a reasonable, but not optimal, job of balancing the load, e.g. the average loads differ by at most b, the system performance compares favorably to that of a perfectly balanced system.

Our approach in this work is to model a NUMA shared-memory multiprocessor system using a queueing system and then use simulations of this queueing system under complex workloads in order to investigate these questions. The concept of work stealing is not new, and there has been a considerable amount of previous work on modeling the performance of these systems. We will describe this work and how our framework adds to the literature in Section 2. Our model is then described in detail in Section 3 and includes a wide array of possible program structures and work stealing algorithms. The simulation results are shown in Section 4. In these simulations, we study the effects of varying the number of processors in the system, the affinity costs, the program structure, the task length variability, and the level of system imbalance on the performance of our work stealing variations.

2 Background

The idea of work stealing can be traced back to the work of Burton and Sleep [6] on parallel execution of functional programs, as well as the implementation of Multilisp (a multiprocessor extension of the Lisp programming language) by Halstead [10]. Work stealing gained popularity in the mid-1990s, with the creation of the Cilk multithreaded programming language [5]. Blumofe and Leiserson [4] presented the first provably good work stealing scheduler for multithreaded computations with dependencies; they prove existentially optimal worst-case bounds on expected execution time, space required, and total communication cost, and show that work stealing has much lower communication cost than work sharing. Their simple work stealing algorithm (WS) assumes that a processor will work on its own tasks if possible, but attempt to steal from a randomly chosen processor if its queue is empty. Many variants of this algorithm are possible, such as those analyzed by Mitzenmacher [17], Acar et al [1], and Squillante et al [22]. Work stealing has also been investigated in a variety of other contexts, including load balancing independent jobs on a parallel computer [19], parallel backtrack search and branch-and-bound computations [13], parallel divide-and-conquer [25], and game tree search [9].

We focus on work stealing models that take *affinity* into account. As discussed in Section 1, in a shared memory multiprocessor system, it may be more efficient to schedule a task on one processor than another. Squillante and Nelson [21] argue that the major cost of task migration is a larger service demand at the processor that migrated the task, reflecting the time needed to establish the cache's working set at this processor. This is an additive cost based on the size of the working set. In this work, we also model a multiplicative cost due to non-uniform memory accesses: on a NUMA shared memory multiprocessor, remote memory accesses take significantly longer than local memory accesses. We discuss these components of our affinity model in more detail in Section 3.

Squillante and Lazowska [20] present a quite complex model of processor-cache affinity. When a task returns for execution and is scheduled on a processor, it experiences an initial burst of cache misses. However, if a significant portion of the task's working set is already in the cache, this penalty is reduced. Thus, we have a tradeoff between load balancing (assigning tasks to processors with low loads) and using locality (assigning tasks to processors with high affinity). Squillante and Lazowska propose and compare various scheduling algorithms which trade off load balancing and affinity, using a detailed cache model to examine cache reload time. They assume a centralized pool of tasks, but allow processors to search the pool and choose tasks for which they have affinity. Their affinity scheduler outperforms both a centralized queue (poor affinity) and distributed queues without work stealing (poor load balancing). However, the comparison ignores contention at the central pool of tasks, and neglects the potentially large time needed to search and remove tasks from the pool. We believe that work stealing is a better way to combine affinity scheduling with load balancing, without creating a single point of contention, and without this significant overhead.

The work most closely related to our present research is that of Squillante and Nelson [21]. Squillante and Nelson present a threshold-based queueing model of shared memory multiprocessor scheduling, and conclude from this model that work stealing can be beneficial even when migration costs are large. The main difference between this model and ours is that Squillante and Nelson assume an idealized workload and affinity model. They assume a streaming workload with Poisson arrivals and exponential processing time; processor-task affinity is taken into account by assuming that the processing time for a migrated task is exponentially distributed with a larger mean than for an unmigrated task. The advantage of these idealized assumptions is that their model can solved analytically as a discrete state space, continuous time Markov process. Similarly, Mitzenmacher [17] uses a differential equations approach to analyze a very similar (idealized workload) model of work stealing. Our workload models are not tractable for these types of analysis, and thus we must rely on discrete-event simulation. The major advantage of our simulation approach is its flexibility and

realism: we are no longer limited to streaming Poisson workloads, but can analyze a variety of more complex and realistic workloads based on program structure. Additionally, we present a more detailed model of the (workload-dependent) factors which influence processor-task affinity, allowing us to more precisely model the effects of affinity on the performance of the work stealing algorithm under a variety of workloads.

Finally, Acar et al [1] present a work stealing algorithm that uses processor-task affinity information to guide its decisions as to which task to steal. Each process maintains a queue of pointers to threads that have affinity for it, and attempts to steal these first; if a processor fails to steal any of the threads for which it has affinity, it then polls other processors as in the standard work stealing algorithm. Acar et al demonstrate that their work stealing algorithm outperforms the standard algorithm on a suite of benchmarks. In this paper, we consider a simpler affinity-guided work stealing algorithm, in which an idle processor polls other processors as before, but looks at the task at the back of each queue, and when possible chooses a queue for which it has affinity with that task.

What separates the current work from these prior approaches is that our focus is not on optimizing a particular work stealing algorithm, since the differences between particular algorithms pale in comparison to the improvement gained simply by doing some form of work stealing. Instead, we focus on understanding the benefits the mechanism provides under *realistic workloads* and a *realistic affinity model*.

2.1 Other approaches

The converse of the work stealing approach is *work sharing*, in which heavily loaded processors attempt to move some of their threads to under-utilized processors. Eager et al [8] argue that work sharing is preferable to work stealing for distributed systems, but this is unlikely to be true for shared memory multiprocessors [21]. In the simulation results of Eager et al, work sharing outperforms work stealing at light to moderate system loads, while work stealing outperforms work sharing at high loads, if the costs of task transfer under the two strategies are comparable. However, they argue that costs are likely to be greater under work stealing, making work sharing preferable. This is because work stealing policies must transfer tasks which have already started executing, while work sharing policies can transfer prior to beginning execution. This argument does not apply to shared memory multiprocessors, which have no difficulty migrating tasks that have started execution [21].

Perhaps more importantly, Squillante and Nelson note that in shared memory systems the cost of task migration is incurred by the migrating processor (the "thief") rather than the processor from which the task is migrated (the "victim"). In a shared memory system, an idle processor can search the queues of other processors and remove a task without disturbing the busy processors; the time required for probing and removal of processes is small. Thus, the direct costs of migration are shifted from the busy processor to the idle processor, and work stealing has much greater potential benefits. Additionally, migration of threads occurs more frequently with work sharing than work stealing, especially when the load is high [4]. Hence, work sharing results in higher overhead due to migration, and lower processor-task affinity. For these reasons, it is likely that work stealing will outperform work sharing for shared memory multiprocessor systems.

Various other approaches to load balancing have been investigated. One possibility is a centralized (rather than distributed) scheduling strategy, as in Hamidzadeh and Lilja [11]. Centralized scheduling uses a dedicated processor to gather state information and decide how to distribute the load. Because the coordinating node has complete or near-complete information about the entire system state, the load can be more evenly distributed than in a distributed scheduling strategy. The main disadvantage of centralized scheduling is contention at the coordinating node, which limits scalability; additional weaknesses include reduced fault-tolerance, and reduced resources (i.e. one less processor available for use). Thus, Lo and Dandamudi [16] attempt to obtain both the benefits of centralized scheduling (better load sharing) and distributed scheduling (better scalability) through a hierarchical load sharing approach. This approach appears promising, though maintaining the hierarchy adds complexity and overhead to the scheduling process.

3 Model Description

In this section, we will build a detailed model of task execution and scheduling in a NUMA shared-memory multiprocessor system. Our goal is to analyze the performance of five variants of the work stealing algorithm [4] for a variety of complex workloads. We focus on the case of highly parallelizable jobs that can be divided into a number of independent tasks, of which some can be parallelized and others must be performed sequentially. For our purposes, it is only interesting to consider the case where the number of tasks is larger than the available number of processors; otherwise each processor is assigned at most one subtask, and (assuming a homogeneous system) no benefits can be gained by work stealing. Our measure of performance is the total time to completion of all tasks of a job, averaged over a long sequence of (independent) jobs. Performance is measured for a wide variety of program structures. For each workload, we ask two main questions:

1. If the programmer has done a good job of balancing the workload, can work stealing still help performance?

2. Can work stealing compensate for an imbalanced load, and thus reduce the burden placed on the parallel programmer?

To answer these questions we will model the multiprocessor system using a set of parallel queues. Each processor will be modeled as a server with an associated queue. A variety of algorithms for work stealing between queues will be investigated, and the model for each of these is described in Section 3.1. Because of local caches and the NUMA architecture of the system, each task will have an affinity for a certain processor. A model of this affinity is described in Section 3.2. Each queue contains tasks generated by the current program being run on the system. The subtasks of each program have a particular dependency structure (see Section 3.3) depending on the type of program. This dependency structure results in new subtasks entering queues upon the completion of a task, as well as synchronization between tasks. The performance of the multiprocessor system will be evaluated under varying workloads, each of which will be formed using a series of jobs with a given dependency structure. These workloads are described in Section 4.

In summary, there are four main pieces to our model: the work stealing algorithm, the affinity of tasks to processors, the structure of programs in terms of their subtasks, and how programs are put together to form workloads. We will now describe in detail how we model each of these pieces.

3.1 Work stealing algorithms

The overriding goal of this work is to show that work stealing has the capability of (i) improving system throughput even when average load among processors is balanced and (ii) easing the burden on the programmer by compensating for imbalanced loads. To test these hypotheses, we examine five variants of the work stealing algorithm: WS_FIRST, WS_BEST, WS_NEIGHBOR, WS_FIRST_AFFINITY, and WS_BEST_AFFINITY. These are compared to two baseline algorithms: distributed queues without work stealing (NO_WS) and a single centralized task queue (CENTRAL_QUEUE). We first discuss the two baseline algorithms, then each variant of work stealing, in turn.

NO_WS

Under the NO_WS policy, each processor maintains a local FIFO queue of tasks. Whenever a processor finishes the task on which it is currently working, it takes the task from the head of its local queue, and begins working on that task; if a processor has no tasks in its queue, it remains idle until new tasks arrive. Also, if a task spawns more subtasks, these are placed on the head of that processor's queue (as in [4]). The clear disadvantage of this policy is its inefficient use of resources: unless the workload is perfectly balanced, it is likely that some processors will sit idle even though other processors are heavily loaded with waiting tasks. However, there is also a major advantage to NO_WS: tasks are always executed on the processor for which they are most likely to have affinity. Because all memory accesses are local rather than remote, no penalty is paid for non-uniform memory accesses; also, it is very likely that a related task will already have been executed on that processor, and thus no cache reload penalty must be paid.

CENTRAL_QUEUE

Under the CENTRAL_QUEUE policy, we keep one centralized FIFO queue of all waiting tasks. Whenever a processor becomes available, the task at the head of the queue is scheduled to that processor, regardless of whether the task has affinity for that processor. Again, newly spawned subtasks are placed at the head of the queue, while newly arriving tasks are placed at the tail, as in [4]. The advantages and disadvantages of the centralized queue policy are clear: better load balancing (since a processor will never be idle while there are tasks waiting to be processed), and poorer use of affinity (since the assignment of tasks to processors is arbitrary, it is unlikely that a task will be assigned to a processor for which it has affinity). Another disadvantage of the centralized queue is contention for the single queue, which may dramatically impact performance and limit scalability. In our model we neglect the effects of contention, but even so the centralized queue's failure to use processor-task affinity results in poor performance.

Variants of work stealing

Under the various work stealing policies, each processor maintains a local FIFO queue of tasks, as in the NO_WS policy. Whenever a processor finishes the task on which it is currently working, it takes the task from the head of its local queue, and begins working on that task. However, if a processor has no tasks in its queue, it attempts to steal a task from another queue, and execute this task. We refer to the processor which steals a task as the "thief," and the processor from which the task is stolen as the "victim." The work stealing policy is determined by which other queues the thief searches, and (if there are multiple potential victims) which victim is chosen for stealing; we discuss the various policies below. Note that tasks are stolen from the back of the victim's queue; this helps to prevent contention between the victim (who is taking tasks from the head of his queue) and the potential thief. It is possible for contention to arise between multiple (potential) thieves, but this is fairly rare, and thus is neglected in our model. This issue must be dealt with in real implementations of work stealing; see, for example, the non-blocking

implementation of work stealing by Arora et al [2]. The goal of work stealing is to combine good load balancing with the benefits of processor-task affinity; since tasks are only stolen when the thief's queue is empty, processors give priority to tasks for which they are most likely to have affinity. Also, since a processor only sits idle if it fails to steal a task, fewer resources are wasted than in the NO_WS policy. However, there is some risk inherent in stealing a task: since the task is likely to run significantly slower on the thief processor than it would have run on the victim (see section 3.2), stealing a task has the potential to actually hurt overall performance.

We now consider the five variants of the work stealing policy which we implement in our model. In the WS_FIRST policy, a processor whose queue is empty will sequentially poll the other processors' queues; as soon as it finds a non-empty queue, it will stop polling, and steal a task from that queue. In the WS_BEST policy, a processor whose queue is empty will first poll all the other processors' queues, and then choose its victim according to which queue has the highest load. Since the thief does not know how long each task will take to execute, it simply chooses the queue with the largest number of tasks, breaking ties arbitrarily. For both the WS_FIRST and WS_BEST policies, if a processor polls all other processors' queues without finding a task, it will sit idle until new tasks arrive; other policies, such as exponential backoff, are also possible but not investigated here. For the WS_NEIGHBOR policy, a processor only polls the processors directly connected to it (its "local neighborhood"). For simplicity, we assume that processor i polls only processors $i-1$ and $i+1$; a variety of other policies based on different local neighborhoods are also possible. The WS_NEIGHBOR policy is much more likely to fail to steal a task, since only two other processors are polled. In spite of this, we find that WS_NEIGHBOR performs nearly as well as WS_FIRST for most workloads. The advantage of the WS_NEIGHBOR policy is that it reduces the complexity and overhead of the polling process; we compare the average number of processors polled (for each attempted steal) in the WS_FIRST, WS_BEST, and WS_NEIGHBOR policies below.

Variants of work stealing with affinity

Finally, we consider two variants of the work stealing policy which take processor-task affinity into account, and attempt to explicitly steal tasks for which they have affinity. These are simpler schemes than that of Acar et al [1] discussed above; their scheme allows processors to steal any task for which they have affinity from a queue, while we only allow stealing from the back of each queue. In the WS_FIRST_AFFINITY policy, a processor whose queue is empty sequentially polls the other processors' queues, looking at the task at the back of each queue. If it finds a task for which it has affinity, it will stop polling and steal that task. Otherwise, after all processors are polled, it will steal a task from the first non-empty queue; if all other queues are empty, the processor will remain idle until new tasks arrive. In the WS_BEST_AFFINITY policy, a processor whose queue is empty polls all other processors' queues, looking at the task at the back of each queue. If it has affinity for any of these tasks, it will steal a task for which it has affinity, in particular the one with the most heavily loaded queue. If it does not have affinity for any of these tasks, it will simply steal from the most heavily loaded queue. Again, the processor will remain idle only if all other queues are empty.

3.2 Affinity characterization

Characterizing the affinity of tasks for processors is essential to any model of a NUMA shared-memory multiprocessor system. There are two dominant causes of processor-task affinity in such systems. First, because of non-uniform memory accesses, any task that is executed on a remote processor will perform at least some of its memory accesses remotely; since remote memory accesses are significantly slower than local memory accesses, this will slow down the overall speed of execution. Second, since each processor has a local cache, any task that is executed on a remote processor must pay the cost both of running on a cold cache and of communicating the task to and from the local processor. As argued by Squillante and Nelson [21], communication costs are low in a shared memory system (as opposed to a distributed system, where communication time may be very large), and the cache cost dominates. However, the cache cost need not always be paid: if a remote processor is working on data which it has already cached (for example, because it has recently executed a task which also relies on that data), the cache will not be cold. Typically, this will occur when a remote processor steals a task from a processor that it has recently stolen other work from.

To model these two types of affinity we use a linear transformation of the original execution time of the task. Suppose a task requires an execution time of x on its local processor. Further, suppose that proportion p of the execution time results from local memory accesses that, *if the task was not run on its local processor*, would have to be performed remotely. Note that this proportion p will be different depending on the program structure; different types of programs will require smaller or larger proportions of remote memory accesses when run remotely. Thus, we assume that the parameter p is workload-dependent. Additionally, assume that a remote memory access takes α times as long as a local memory access, where α is a constant fixed across workloads. Thus the total time to run the task remotely is $(\alpha p + 1(1-p))x$. We set $A = \alpha p + 1(1-p)$, and thus the remote execution time is Ax, where we call A the *memory affinity*. More precisely, A is the multiplica-

tive penalty on execution time for tasks without affinity, due to non-uniform memory accesses. However, we must also consider the costs of remote execution due to reloading of a cold cache, and due to communication with the local processor. The cache reload time is assumed to be a workload-dependent parameter β, resulting from the large number of cache misses occurring when a task returns for execution on a cold cache [20]. This cost is workload-dependent because it depends on the size of the data structures being used for a given task; as discussed above, this cost does not need to be paid if the cache is warm. Also, communication costs are assumed to be a small (workload-dependent) parameter γ. Setting $B = \beta + \gamma$, we obtain a remote execution time of $Ax + B$ on a processor with a cold cache, or $Ax + \gamma$ on a processor with a warm cache. For simplicity, and because of the arguments of Squillante and Nelson [21] that communication costs can be neglected, we assume $\gamma = 0$ for all models. Thus, the remote execution time becomes a linear function $Ax + B$, where A denotes the memory affinity and B denotes the cache affinity. A processor is said to have affinity for a task on another processor's queue when at least one of the following is true: (i) its cache is warm for that task, and additive cost B does not need to be paid, or (ii) the task is local to the thief processor (i.e. because it is a subtask of some task originally stolen from the thief processor), and thus the multiplicative cost A does not need to be paid. For each of the program structures below, we discuss under what circumstances each of these conditions holds.

3.3 Job Characterization

As mentioned above, the jobs in the queues of our model represent subtasks of parallel programs executed on the multiserver system. Thus, the structure of the dependencies between these jobs is key to the accuracy of the model. In this section, we describe varying program frameworks that cover a wide spectrum of parallel program structures. The frameworks described here cover the key interactions between subtasks, including sequential subtasks, parallelizable subtasks, and synchronization between subtasks. For each program framework we provide a description of typical uses of programs that fall within the framework and then describe an abstraction of the framework into sequential and parallelizable tasks. The frameworks we describe in this section build on the models of Au and Dandamudi [3], and Dandamudi and Ayachi [7]. We focus on five different program structures: fork and join (e.g. a parallelizable loop), streaming data (e.g. video processing), exhaustive search (e.g. multiresolution spatial decomposition), pruned search (e.g. game tree search), and divide and conquer (e.g. merge sort). We discuss each of these program structures in detail in the following subsections.

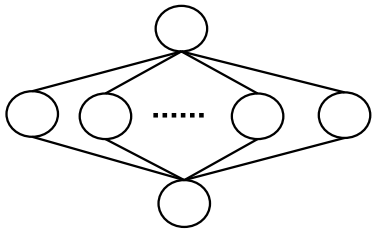


Figure 1: *The fork and join job structure.*

3.3.1 Fork and join

Fork and join programs are those where the work can be decomposed into parallelizable subtasks. A simple example of this type of program is a single loop when the computations performed in each iteration are independent. Thus, each iteration of the loop can be performed in parallel. More generally, it is possible for a parallelizable loop to be run inside of a sequential loop, thus creating a chain of fork and join subtasks. A more complicated example of this type of program structure is the n-body problem, where the movement of n stellar bodies is simulated and thus the gravitational forces created by each body must be calculated. In this situation, the movement of each body is affected by the gravitational force of the whole system (see, for instance, Jones and Murta [12]). Fork and join workloads have been investigated by Leutenegger and Vernon [15], Towsley et al [24], and many others.

Following Au and Dandamudi [3], we abstract the fork and join structure in Figure 1. This shows one phase of a typical fork and join; the computation is decomposed into a large number of independent tasks, and once all of these tasks complete, the program moves on to the next phase. Some preprocessing and/or postprocessing may be necessary, but this can be neglected in our model; the important thing to note is that all of the tasks in a phase have a single point of synchronization. If we are considering a multi-phase computation, with a sequential outer loop and a parallelizable inner loop, the parallel computation phases correspond to the work done inside the inner loop, and the synchronization points correspond to the communication and synchronization done in the outer loop. For simplicity, however, we assume that each job in our workload has only one phase of fork and join since all phases are stochastically equivalent; this assumption is also made in [3].

To model affinity of tasks for processors, we need to make further assumptions about the program structure. In particular, we assume that the computation has been decomposed into nP parallel tasks, where P is the number of processors and n is the number of tasks assigned to each processor. We assume that each task is *local*

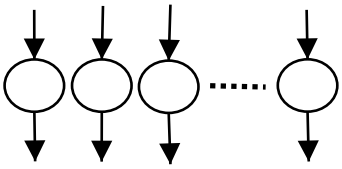


Figure 2: *The streaming job structure.*

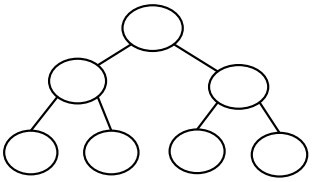


Figure 3: *The exhaustive search job structure.*

to the processor on which it is originally assigned; thus any remote execution on another processor will have to pay the multiplicative penalty for remote access, $A = \alpha p + 1(1-p)$. We fix α, the penalty for a remote access, across all workloads, assuming $\alpha = 9$ (i.e. a remote access takes 9x as long as a local access). Furthermore, we assume that the proportion of remote memory accesses is $p = \frac{1}{16}$ (this is a simple order-of-magnitude estimate; we will also allow p to vary in our experiments). These values of α and p give us $A = 1.5$. Also, we assume that a processor's cache is warm for a particular task if that processor has previously executed any task operating on the same data structure. From here, there are several assumptions we could make. One possibility is that each task relies on a different data structure, thus the cache penalty must be paid for every task a processor executes (this can be incorporated into the task execution time, allowing us to treat this case as if $B = 0$). Another possibility is that each processor is initially assigned a set of tasks which operate on the same data structure. This could result from a program such as the following:

```
divide space into P regions
for i=1 to n
  for j=1 to P
    perform task i on region j
```

In this case, a processor's cache is warm for a task if that processor has previously executed any task originating on the same processor. If the cache is not warm, then we assume that an additive penalty of $B = 0.5$ must be paid in order to load the data structure into the cache. (Since our choice of B is also somewhat arbitrary, we will examine performance over a wide range of affinity parameters). We will also get similar affinity behavior if the data structures are small (i.e. multiple structures fit into the same cache line), and adjacent elements can be assigned to the same processor.

3.3.2 Streaming

One important use for multiprocessor systems is to process data for streaming applications. Applications such as audio and video streams as well as real-time applications depend on multiprocessor systems to efficiently handle streaming data. To model this sort of application, we consider that each processor has an independent stream of n incoming subtasks that it must process. This is pictured in Figure 2. Similar models have been presented in [17, 21], though they make the idealized assumption of Poisson interarrival times. Instead we model interarrival times with a Coxian distribution. This approach is suggested by the work of [23], which indicates that interarrival times have higher variance than can be modeled by a simple Poisson distribution; the Coxian allows us to adjust the *coefficient of variation* (CV) accordingly. We assume that arrival times are distributed with a CV of 4; this falls within the suggested range of 2.5-6.

Again, we must make more detailed assumptions about the program structure in order to model processor-task affinity. We assume that the domain has been decomposed into P regions (one per processor), where each region produces events at random intervals and these events must be processed. As in the fork and join workload, any task is local to the processor on which it is originally assigned; we assume the proportion of local accesses is identical ($p = \frac{1}{16}$) giving us a multiplicative penalty of $A = 1.5$ as before. Also, we assume that the information for each region is loaded into the cache of its local processor; remote executions must also load this data (and take a number of cache misses in order to do so), resulting in an additive penalty of $B = 0.5$ as above.

3.3.3 Exhaustive search

The exhaustive search model is perhaps the simplest case in which tasks spawn other subtasks. In this case, some large data structure must be searched in order to find a substructure which meets certain criteria. For example, one major application is in epidemiology, where the goal is to search for a spatial cluster of disease cases (which may be indicative of an emerging epidemic). In this case, we are searching for the spatial region that maximizes some density measure D, dependent on the count (number of disease cases) and underlying population of the region. One of the most widely used algorithms for detection of disease clusters is the *spatial scan statistic* [14], which

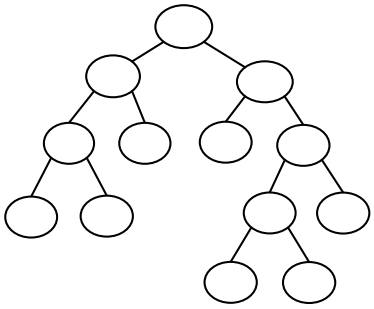


Figure 4: *The pruned search job structure.*

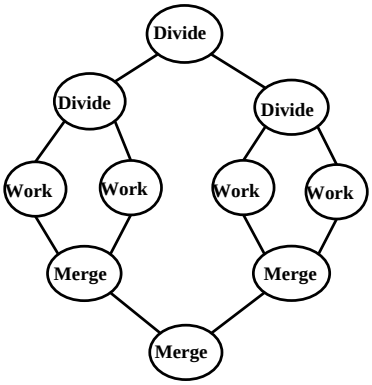


Figure 5: *The divide and conquer job structure.*

searches all possible circular regions and finds the most significant spatial disease cluster. One possible way to implement this is a top-down search: we search first over very large regions, then decompose the domain into subdomains and search over the smaller regions contained in each subdomain. This allows the search to be easily parallelized, where each subdomain is assigned to a different processor; subdomains are further divided for load balancing purposes.

We consider a very simple version of the scan statistic, where the domain is divided into P subdomains, and one subdomain is assigned to each of the P processors. In order to search a subdomain, we not only search the largest regions in that subdomain (the parent task) but continue to spatially divide the region. Thus, we must search not only the parent subdomain (i.e. execute the parent task) but also progressively smaller subregions of each. We assume that after searching each subdomain, we divide it into quarters (spawning four child tasks) and search each quarter. Similarly, each child task spawns four tasks, and so on. This is continued until we reach the smallest resolution we are interested in (assumed to be the great-grandchildren of the original subdomain), at which point the tasks complete without spawning subtasks. Note that because we are performing an exhaustive search, each subdomain spawns an identical number of descendent tasks (4 children + 16 grandchildren + 64 great-grandchildren); this will not be true in the pruned search case below. See Figure 3 for an abstract depiction of this task structure.

We now consider the affinity model as applied to the exhaustive search workload. Since one subdomain is assigned to each processor, that task and any subtasks spawned by that task are local to that processor. If any other processor steals one of these tasks, it must pay the multiplicative penalty $A = \alpha p + 1(1-p)$ for remote memory accesses on a NUMA system. We assume $\alpha = 9$ as before; however, the proportion of remote memory accesses p is likely to be higher. This is because search tends to rely heavily on large numbers of memory accesses, as compared to signal processing where a larger proportion of the time may be spent computing complicated mathematical functions. Thus, we assume $p = \frac{1}{8}$ for the exhaustive search workload, giving us a multiplicative penalty $A = 2$. Additionally, a processor must pay the additive penalty for cache reload for a given subtask if it has not previously executed an ancestor task (parent, grandparent, or great-grandparent) of that subtask. For simplicity, we assume $B = 0.5$ as above.

3.3.4 Pruned Search

While exhaustive searches are used in many cases, often the sheer size of the domain being searched makes an exhaustive search impractical or impossible. In this case, our goal is to search over the entire domain, but not to explicitly search every single subdomain; instead, we use methods which allow us to conclude that certain subdomains do not need to be searched, and to *prune* these subdomains from our search tree. Pruned search includes a variety of different techniques, many of which can be efficiently parallelized: for example, parallelized branch-and-bound computations have been investigated by Karp and Zhang [13], and parallelized alpha-beta game tree search has been studied by Feldmann et al [9]. In all of these cases, the goal of pruned search is typically to search only a small fraction of the total number of possible subdomains, resulting in a much faster search. For example, Neill and Moore [18] present a multiresolution branch-and-bound algorithm for computing the spatial scan statistic, and demonstrate that their method finds spatial disease clusters 20 to 150 times as fast as the standard (exhaustive search) approach.

Returning to our simple version of the spatial scan statistic, we now consider a pruned search approach to the problem. We decompose the domain exactly as before: P tasks are created, one per subdomain, and are

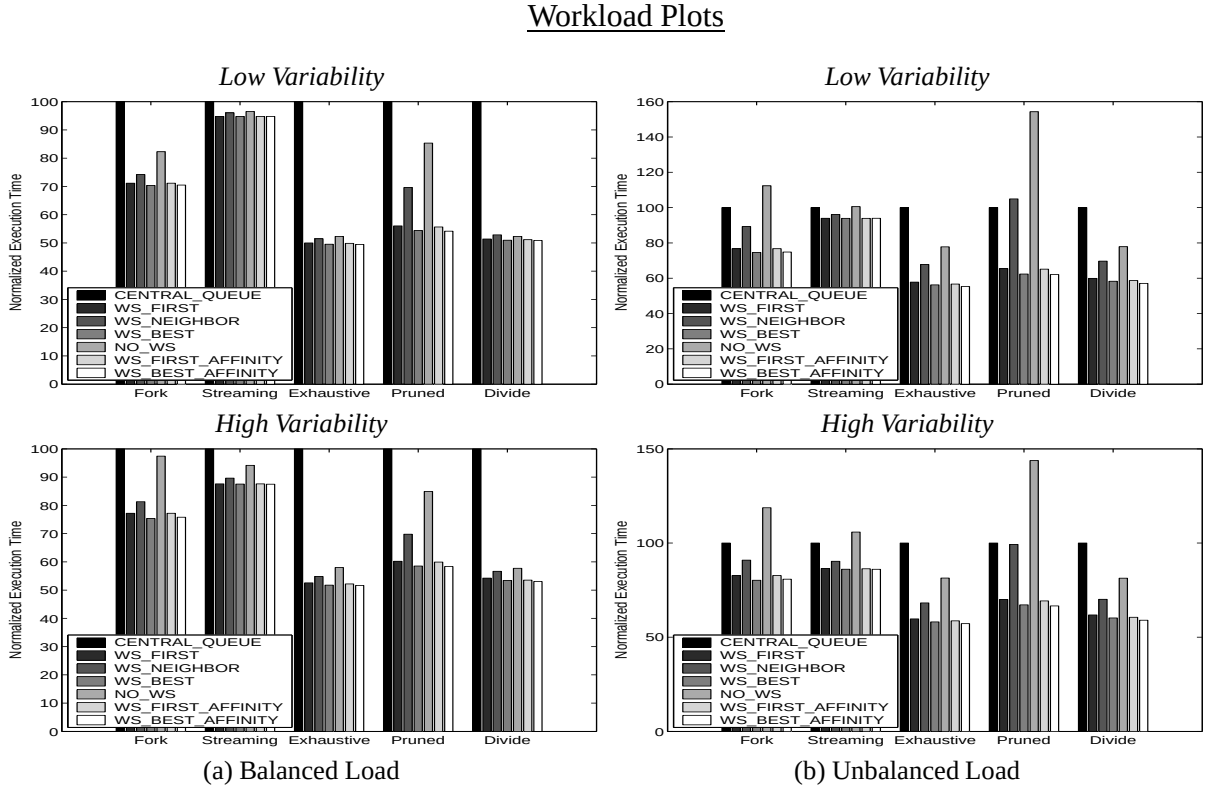


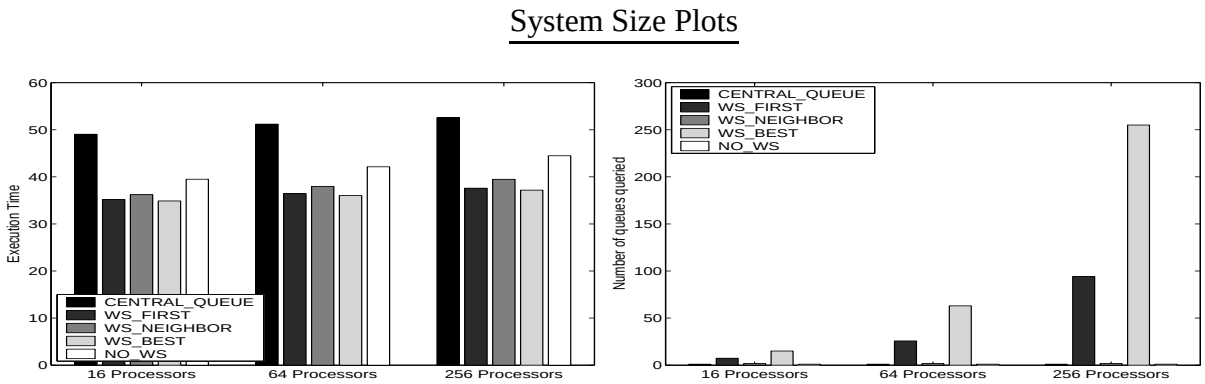
Figure 6: *Comparison of work stealing algorithms across workloads. The metric shown is the average execution time of a job normalized to the performance of CENTRAL_QUEUE. In all plots the system has 64 processors. In the top row the tasks have $C^2 = 1$; in the bottom row the tasks have $C^2 = 4$. In (a) the average load at all processors is equal ($b = 0$). In (b) the average loads differ by at most $b = 0.6$.*

spatially divided (quartered) into child, grandchild, and great-grandchild regions in turn. The difference is that we may be able to prune some tasks or subtasks, allowing us not to search any descendent of that task or subtask. Rather than precisely modeling the search scheme and pruning, we assume simply that there is a given probability p_{prune} that we will be able to prune any given subtask. Thus a task will spawn four children (of the next higher resolution) upon completion with probability $1 - p_{prune}$, and spawn no children with probability p_{prune}. It is clear from this model that different subdomains may spawn different numbers of descendent tasks: for each subdomain, anywhere from 1 task (the parent region only) to 85 tasks (parent, children, grandchildren, great-grandchildren) must be executed. See Figure 4 for an abstract depiction of this task structure.

In some cases, child tasks may require significantly less work than their parents, but this is not the case for the spatial scan statistic; thanks to a handy computational trick (the "cumulative density" trick) it requires the same amount of time to compute the statistic for any region regardless of size, and we assume that all task sizes are drawn from the same distribution. This means that the total amount of computation done for a subdomain may vary by a huge factor, depending on how many of the 85 possible subtasks must be performed. As a result, even when the expected loads are balanced for the pruned search case, the actual loads may vary wildly, and it is particularly important to dynamically balance the actual loads. Finally, we note that the affinity model is identical in the exhaustive search and pruned search cases.

3.3.5 Divide and conquer

The final workload we consider is divide and conquer, a common algorithmic approach that has been applied to database management, circuit design, numerical integration, and many other areas. Parallel algorithms for divide and conquer have been investigated by Wu and Kung [25] and others. This approach consists of three types of computation. First, the approach divides the computation task into easier to manage subtasks; these may in



(a) Execution Time (b) Queues Queried

Figure 7: *Comparison of algorithms across systems of differing size. In all cases the average load at each of the processors is the same ($b = 0$), the task sizes have $C^2 = 1$, and the fork and join workload is used. (a) shows the average execution time per job. (b) shows the average number of queries made per attempted steal under each of the scheduling algorithms.*

turn be further divided, as in the exhaustive search workload. Once this division has occurred, each of the subtasks has work performed on it. Finally, each of these subtasks are merged; typically the structure of merges mirrors the structure of task divisions. Generally, the execution times of the tasks decreases level by level during the divide stages and increases level by level during the merge stages.

As in Au and Dandamudi [3], we focus on the binary case, where each non-leaf task is divided into two subtasks, which will later be merged; the parallel structure of this approach is depicted in Figure 5. One example application is merge sort: in order to sort a list of words, we divide the list in half, recursively merge sort each half of the list, and merge the two halves. Note that the merges in this case are non-trivial; this distinguishes the divide and conquer structure from other structures which partition the domain, such as exhaustive search. The most interesting feature of the divide and conquer framework is its reliance on synchronization of subtasks. As opposed to the other frameworks, where a single point of synchronization (i.e. the completion of all subtasks) is assumed, divide and conquer has multiple points of synchronization, each of which synchronize a subset of the tasks. In particular, synchronization is performed in each merge stage, because a merge stage cannot occur before all subtasks being merged complete execution. We implement this synchronization explicitly by dynamically generating a new merge task whenever its two parent tasks have both completed; the merge task is assigned to whichever processor completes the last of its parent tasks (this way no persistent state is needed other than the number of completed parents).

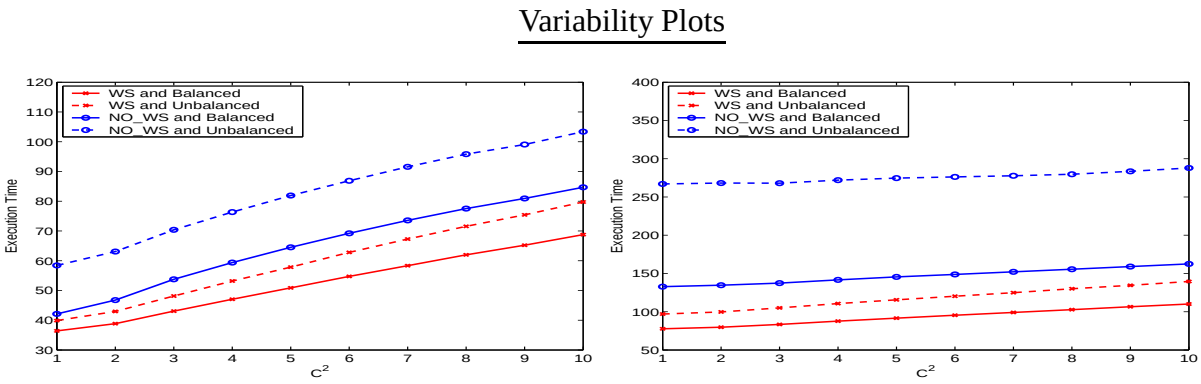
Affinity is modeled as before: the domain is split into P subdomains, one per processor, and the subtasks of each have an independent divide and conquer structure. All divide, work, and merge tasks originating from a subdomain are assumed to be local to the processor for which that subdomain was assigned; if any of these tasks are executed remotely, a multiplicative penalty of $A = 2$ (as in exhaustive search) must be paid. Finally, we simplify by assuming that a processor's cache is warm for a given task if any ancestor task has been executed on that processor: this includes, for example, if one of the two parents of a merge task has been executed. For a more precise model, we could consider cache affinity as a real (rather than binary) quantity, attempting to compute the proportion of the data structure which is currently resident in the processor's cache; one possible approach would be a model similar to [20].

4 Simulation Results

Using the complex model described in Section 3, we are able to study the benefits provided by work stealing algorithms in multiprocessor systems. Before discussing the results of this study, we will first describe the setup of our simulations.

4.1 Simulation Setup

Each of the program frameworks described in Section 3.3 will be the basis of a series of workloads used in this section. Each workload can be characterized by a tuple as follows: $W = (\Omega, S, P, b, C^2, A, B)$, where Ω is the pro-



(a) Fork and join workload

(b) Pruned search workload

Figure 8: *Comparison of algorithms across differing task size variabilities. The number of processors is fixed at 64. The work stealing algorithm used is WS_BEST. Under the balanced workloads all processor loads are equal ($b = 0$). Under the unbalanced workloads, processor loads may differ by up to $b = 0.6$.*

gram workload, S is the scheduling algorithm, P is number of processors, b is how balanced the server loads are, C^2 is the variability of the task sizes, and A,B are the affinity constants. Note that A,B depend on the given program structure Ω, and unless otherwise stated are defined as described in Section 3.2. The average load in these simulations is proportional to the average task size, since we are looking at the execution time of one job at a time and there is never more then one job (having a large number of subtasks) in the system.

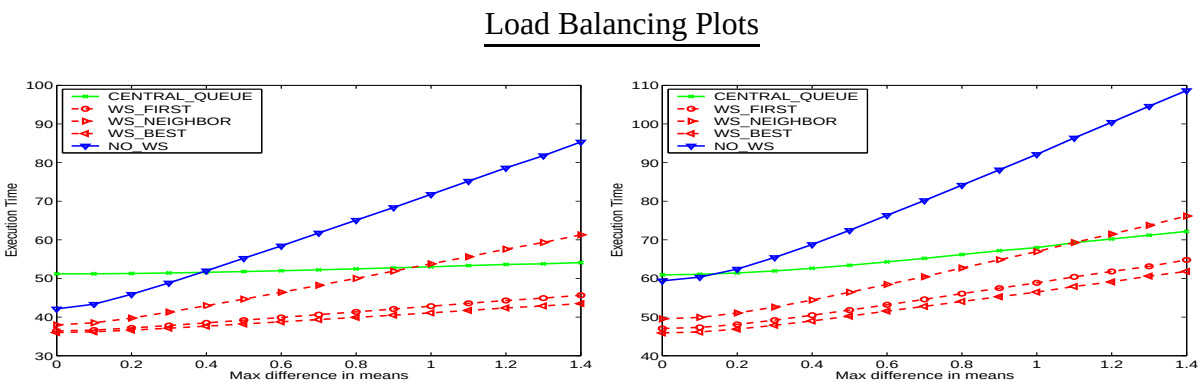
In order to answer our first question, whether work stealing can improve performance even when average load per processor is balanced, we consider the baseline case where each processor has the same initial number of tasks and the same average task size. For simplicity, we assume that average task size is normalized to 1; thus actual task sizes are generated from a Coxian distribution with mean 1 and coefficient of variation C^2. Also, the number of tasks per processor n is randomly selected for each job, ranging from 1-4 for the computation-intensive workloads (exhaustive search and divide-and-conquer), and 8-32 for the other three workloads.

In order to answer our second question, whether work stealing can compensate for poor manual load balancing, we also investigate workloads where the average load per processor is initially unbalanced. To do this, we still assume that the same number of tasks n are initially assigned to each processor. The average task sizes, however, are assumed to be larger for some processors than for others. We accomplish this by letting each processor's average task size be chosen uniformly from the interval $[1 - b/2, 1 + b/2]$, where b is the "system balance parameter," or the maximum allowable difference in processor loads. Actual task sizes are chosen from Coxian distributions with these means, and with coefficient of variation C^2, as before.

We run simulations that investigate how each of $\Omega, S, P, b, C^2, A, B$ affect the performance of multiprocessor systems. The simulations are each run for a series of 10,000 jobs (each job having more subtasks than there are processors), and we calculate the average execution time per job over these 10,000 jobs.

4.2 Results

We begin our analysis with a comparison of all scheduling algorithms across all program structures. This comparison is shown in Figure 6. In this figure, in addition to varying the program structure, we vary whether the system is load balanced and how variable the task lengths are. Starting with the upper left plot in Figure 6, the first point to notice is that work stealing helps. All of the work stealing algorithms provide a significant improvement over both CENTRAL_QUEUE and NO_WS. Further, there is very little difference between the performance of any of the work stealing algorithms. The only algorithm with noticeably different performance is WS_NEIGHBOR, which, especially in the case of the pruned search workload, does not perform as well as the other work stealing variants. It is interesting that the pruned workload exhibits such different behavior than the other workloads. This difference results from the fact that the pruned workload has extremely unbalanced instantaneous load even when average load is precisely balanced. That is, by trimming computation branches, pruning leads to some processors having more work to do than other processors. We can further see the effect of unbalanced system load by considering the top right plot in Figure 6. Here we see that



(a) Low Variability ($C^2 = 1$) (b) High Variability ($C^2 = 4$)

Figure 9: *Comparison of system load imbalance. In all situations, the fork and join workload is used, and the number of processors in the system is 64. In (a) the subtask sizes have low variability and in (b) the subtask sizes have high variability. The system imbalance b is varied from 0 (balanced workload) to 1.4 (extremely unbalanced workload).*

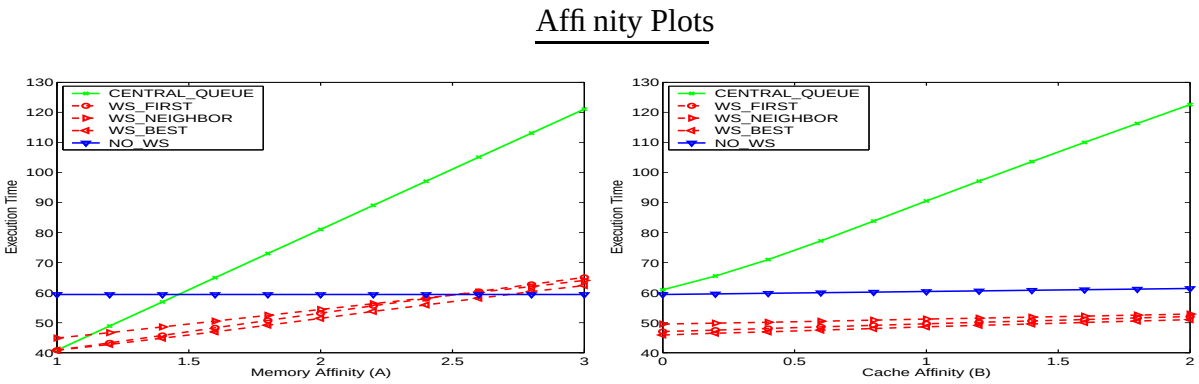
the relative performance of CENTRAL_QUEUE is better when the loads are unbalanced. The reason for this is that CENTRAL_QUEUE naturally balances the load among processors, at the expense of paying some affinity costs. NO_WS and WS_NEIGHBOR, on the other hand, are significantly hurt by the imbalanced load. Intuitively, this is because, when load is imbalanced, NO_WS can end up with idle processors unable to help overloaded processors; similarly, under WS_NEIGHBOR processors can only help their neighbors, which does not provide enough help when the load is imbalanced. A last point to take away from Figure 6 is that changing the variability of task sizes does not fundamentally affect the system performance. This can be observed by comparing the top and bottom rows of the figure.

Given our analysis of Figure 6, we can conclude that the fork and join workload is a representative workload. Thus, for the rest of the section we will focus on this particular workload. Further, we can notice that the work stealing algorithms that try to steal jobs with affinity do not perform significantly better than the ones that ignore the affinity of jobs being stolen. Thus, for the remainder of the section we will only consider WS_FIRST, WS_NEIGHBOR, and WS_BEST.

In Figure 6, the number of processors in the system was held constant at 64. Figure 7 illustrates that the conclusions we made looking at Figure 6 are unaffected by the number of processors in the system. In Figure 7 (a) we show the execution time of the fork and join workload on systems of 16, 64, and 256 processors. It is evident that the execution times are relatively unaffected by the size of the system. The (small) relative performance advantage of WS_BEST over WS_NEIGHBOR does increase slightly with the number of processors, from 3.7% with 16 processors, up to 5.9% with 256 processors. However, as shown in Figure 7 (b), the overhead of WS_FIRST and WS_BEST (as measured by number of queues queried per attempted steal) increases proportional to the number of processors in the system, while the number of queues polled under the CENTRAL_QUEUE, NO_WS, and WS_NEIGHBOR algorithms stays constant.

We now return to the motivational question for this paper: can work stealing ease the burden on the programmer by eliminating the need for load balancing. We answer this question in Figure 8. This shows the performance of WS_BEST and NO_WS under balanced and unbalanced system loads, as a function of the variability in task sizes C^2. This figure shows that work stealing on an unbalanced system load can outperform a system with balanced average load that does not perform work stealing. Thus, better performance can be attained with lesser burden to the programmer when the system architecture performs work stealing. Further, when a programmer does make the effort to load balance a program, the performance obtained in an architecture that performs work stealing is far better than when no work stealing is performed.

So far, we have considered just balanced and unbalanced systems. Figure 9 illustrates the effect of the level of imbalance in the system, by varying the balance parameter b. In these plots we can see that NO_WS is significantly hurt by an increasing load imbalance in the system. As discussed earlier, WS_NEIGHBOR is also negatively affected by an increasing load imbalance, though to a lesser extent than NO_WS. CENTRAL_QUEUE, on the other hand, insulates the system from any load imbalance. Interestingly, the work stealing algorithms also insulate the system from imbalanced load. Further, they provide significantly improved performance over CEN-



(a) Memory affinity (b) Cache affinity

Figure 10: *Comparison of performance with varying affinity costs. In all situations the fork and join workload is used, the task sizes have $C^2 = 4$, and the number of processors in the system is 64. In (a) we vary the memory affinity cost A and fix $B = 0$. In (b) we vary the cache affinity cost B and fix $A = 1.5$.*

TRAL_QUEUE. The fact that work stealing algorithms provide insulation against load imbalance is not surprising. This is in fact the primary goal of work stealing techniques; they strive to distribute work from overloaded processors to processors with lower loads.

Finally, we investigate the effect of the strength of task affinity on the benefits of work stealing. In Figure 10, we vary both the A and B parameters of the affinity model described above. In these plots, we can immediately see that CENTRAL_QUEUE is significantly hurt by increasing affinity costs. On the other hand, NO_WS is insulated from changes in affinity costs because every task is scheduled on the processor where it has affinity. The work stealing algorithms also provide insulation from changing affinity costs; and, as long as the affinity costs are not outrageous, work stealing algorithms provide improved performance over NO_WS. The fact that work stealing algorithms provide insulation against increasing affinity costs is a bit surprising. However, there is some intuition behind this finding. When affinity costs are high, a processor is unlikely to steal more than a handful of tasks because, when a task is stolen, its execution time becomes large. Thus, a work stealing system will not differ too much from a NO_WS system. On the other hand, when affinity costs are small, tasks that are stolen are only slightly larger than normal tasks. Thus, it is possible that a large number of tasks will be stolen, which provides a significant improvement for work stealing algorithms over NO_WS.

5 Conclusion

Work stealing is a powerful architectural tool, the effects of which are not well understood. In this work, we develop a complex queueing model of a NUMA multiprocessor system in order to study the benefits of work stealing in modern computer systems. Prior to this work, the performance of work stealing had not been thoroughly studied under workloads with real-world applicability in systems with real-world affinity models. We model the dependencies that result from realistic program structures, and take into account the affinity of tasks to processors that is present in NUMA multiprocessor systems.

Using simulations, we then investigate the benefits provided by work stealing. We find that work stealing insulates system performance from the negative effects of both high levels of affinity and imbalanced system load. In particular, many common policies, such as CENTRAL_QUEUE, are significantly hurt by increasing processor affinity costs. Other common policies, such as NO_WS are significantly hurt by increasing load imbalance in the system. Work stealing algorithms are the unique in that they are able to insulate system performance from both of these effects. As a result, we find that work stealing outperforms both CENTRAL_QUEUE and NO_WS over a wide range of conditions.

The insulation provided by work stealing policies has another added benefit. One of the most difficult tasks of parallel programming is that of balancing the load across the system. Thus, because work stealing insulates the system from the effect of load imbalance, work stealing is an architectural technique that eases the burden on the programmer. Specifically, when programming in a system that performs work stealing, the programmer need not be concerned about load balancing, and will pay a very minor performance cost for a significant reduction in programming time.

One important aspect of this research is that it considers not only the streaming workload, which is typically used in queueing models, but also workloads with more complicated program structures. Interestingly, the work stealing algorithms showed the smallest proportional improvement for the streaming workloads, and significantly larger performance gains for all other workloads tested. The impact of this observation is that previous analytical models of work stealing may actually underestimate the gains provided by this technique.

Because the focus of this work is on showing the benefits provided by work stealing in general, little time is spent optimizing the performance of the work stealing variants. An important future topic of work is to understand how to perform work stealing in a way that provides the maximal performance benefits while minimizing the system overhead necessary.

References

[1] U. Acar, G. Blelloch, and R. Blumofe. The data locality of work stealing. In *ACM Symp. Parallel Algorithms and Architectures*, pages 1–12, 2000.

[2] N. Arora, R. Blumofe, and C. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[3] S. Au and S. Dandamudi. The impact of program structure on the performance of scheduling policies in multiprocessor systems. *Int. J. Computers and Their Applications*, 3(1):17–30, 1996.

[4] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *IEEE Conference on Foundations of Computer Science*, pages 356–368, 1994.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmal, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel and Distributed Computing*, 37(1):55–69, 1996.

[6] F. Burton and M. Sleep. Executing functional programs on a virtual tree of processors. In *Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, 1981.

[7] S. P. Dandamudi and S. Ayachi. Performance of hierarchical processor scheduling in shared-memory multiprocessor systems. *IEEE Transactions on Computers*, 48(11):1202–1213, 1996.

[8] D. Eager, E. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated load sharing. *Performance Evaluation*, 6:53–68, 1986.

[9] R. Feldmann, P. Mysliwietz, and B. Monien. A fully distributed chess program. Technical Report 79, University of Paderborn, 1991.

[10] R. Halstead. Implementation of multilisp: Lisp on a multiprocessor. In *Symposium on Lisp and Functional Programming*, pages 9–17, 1984.

[11] B. Hamidzadeh and D. Lilja. Dynamic scheduling strategies for multiprocessors. In *Conference on Distributed Computing*, pages 208–215, 1996.

[12] P. Jones and A. Murta. Practical experience of run-time link configuration in a multi-transputer machine. *Concurrency: Practice and Experience*, 1(2):171–189, 1989.

[13] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch and bound computation. *J. ACM*, 40(3):765–789, 1993.

[14] M. Kulldorff. A spatial scan statistic. *Comm. in Stat. Theory and Methods*, 26(6):1481–1486, 1997.

[15] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *ACM Sigmetrics*, pages 226–236, 1990.

[16] M. Lo and S. Dandamudi. Performance of hierarchical load sharing in heterogeneous systems. In *Conf. Parallel and Distributed Computing Systems*, pages 370–377, 1996.

[17] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 212–221, 1998.

[18] D. B. Neill and A. W. Moore. A fast multi-resolution method for detection of significant spatial disease clusters. *Adv. in Neural Information Processing Systems*, 16, 2004.

[19] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[20] M. Squillante and E. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel and Distributed Systems*, 4:131–143, 1993.

[21] M. Squillante and R. Nelson. Analysis of task migration in shared-memory multiprocessor scheduling. In *ACM Conference on the Measurement and Modeling of Computer Systems*, pages 143–155, 1991.

[22] M. Squillante, C. Xia, D. Yao, and L. Zhang. Threshold-based priority policies for parallel-server systems with affinity scheduling. In *American Control Conference*, pages 2992–2999, 2001.

[23] M. Squillante, D. D. Yao, and L. Zhang. The impact of job arrival patterns on parallel scheduling. *Performance Evaluation Review*, 26(4):52–59, 1999.

[24] D. Towsley, C. G. Rommel, and J. A. Stankovic. Analysis of fork-join program response times on multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 1:286–303, 1990.

[25] I. Wu and H. T. Kung. Communication complexity for parallel divide and conquer. In *IEEE Symposium on Parallel and Distributed Processing*, pages 151–162, 1991.