On-Line Optimization Techniques for Task Scheduling in Multiprocessor Systems

Technical Report No: HPPC-95-07

Babak Hamidzadeh
David J. Lilja
On-Line Optimization Techniques for Task Scheduling in Multiprocessor Systems

Babak Hamidzadeh
(hamidzad@cs.ust.hk)
Department of Computer Science
University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong

David J. Lilja
(lilja@ee.umn.edu)
Department of Electrical Engineering
University of Minnesota
Minneapolis, MN 55455

Abstract

Efficiently scheduling parallel tasks on to the processors of a multiprocessor system is critical to achieving high performance. Given perfect information at compile-time, a static scheduling strategy can produce an assignment of tasks to processors that ideally balances the load among the processors while minimizing the run-time scheduling overhead and the average memory referencing delay. Since perfect information is seldom available, however, dynamic scheduling strategies distribute the task assignment function to the processors by having idle processors allocate work to themselves from a shared queue. While this approach can improve the load balancing compared to static scheduling, the time required to access the shared work queue adds directly to the overall execution time. In this paper, we introduce a class of algorithms (i.e. Self-Adjusting Dynamic Scheduling (SADS)) that undertake an on-line optimization strategy to dynamically compute partial schedules based on the loads of the other processors and the memory locality (affinity) of the tasks and the processors. We also introduce a general framework for classifying and characterizing different scheduling strategies. The paper provides analytical and empirical analyses of the SADS family of scheduling algorithms. Our results show that the SADS algorithms outperform existing dynamic scheduling algorithms by performing optimization techniques while explicitly controlling the scheduling time.

Key Words: dynamic scheduling, scheduling costs, load balancing, locality management.

<table>
<thead>
<tr>
<th>Name</th>
<th>email</th>
<th>phone</th>
<th>fax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Babak Hamidzadeh</td>
<td><a href="mailto:hamidzad@cs.ust.hk">hamidzad@cs.ust.hk</a></td>
<td>852-2358-7011</td>
<td>852-2358-1477</td>
</tr>
<tr>
<td>David J. Lilja</td>
<td><a href="mailto:lilja@ee.umn.edu">lilja@ee.umn.edu</a></td>
<td>612-625-5007</td>
<td>612-625-4583</td>
</tr>
</tbody>
</table>
1. Introduction

The efficient scheduling of the parallel tasks partitioned from an application program is essential to achieve high performance in large-scale multiprocessor systems. Many scheduling problems have been shown to be NP-complete[1] making it infeasible to compute optimal schedules when the number of parallel tasks to be scheduled or the number of available processors is large. The complexity of the multiprocessor scheduling problem stems from the existence of many interrelated factors that directly or indirectly contribute to the execution time of the program. To effectively schedule the parallel tasks, these competing factors must be considered in the scheduling cost function. Many different scheduling algorithms have been proposed which consider a few of these factors simultaneously. Such scheduling algorithms have been divided into many different categories such as local versus global scheduling, static versus dynamic scheduling, and centralized versus distributed scheduling[2,3].

The focus of this paper is on dynamic scheduling algorithms which aim at reducing total execution costs of tasks on a multiprocessor architecture by addressing all major cost factors simultaneously. Dynamic scheduling is an important area of resource management in operating systems, since it allows part or all of the scheduling task to be performed at run time when a large amount of useful information about the problem instances and their interaction with the underlying architecture becomes available. The dynamic scheduling problem in multiprocessor environments can be regarded as a class of problems to which we refer as on-line optimization problems. On-line optimization problems arise in many application areas such as adaptive control, communication networks and real-time computing. An example is event-driven applications such as real-time simulation where the occurrences of tasks and their invocation patterns become known only during problem solving. Another class of problems which can benefit from dynamic scheduling are numerical analysis applications where the input to the system consists of largely varying and unpredictable tasks. Fluid dynamics problems in which the movement behavior of the fluids and particles varies unpredictably and can be detected only at run time are examples of such applications.

Dynamic scheduling techniques search for appropriate schedules on line and thus cannot take a long time to execute. Existing dynamic scheduling techniques are efficient, but they may produce poor schedules. The complexity of such algorithms has been investigated extensively but the direct effect of scheduling complexity on the total execution time and its tradeoff with the quality of the resulting schedules has not been addressed.

In this paper, we introduce a class of dynamic scheduling algorithms, referred to as Self-Adjusting Dynamic Scheduling (SADS), which address load balancing and memory locality while explicitly accounting for the time spent on scheduling. The SADS algorithms are unique since they directly control the scheduling time and since they employ optimization techniques, at run time, to account for major cost factors which affect performance in multiprocessor architectures. The tradeoff between the scheduling costs and the quality of resulting schedules leads to interesting analytical and empirical results. SADS algorithms are dynamic approaches which take advantage of information about task execution times that is not available at compile-time. They can be regarded as a centralized strategy in which a single processor is dedicated to executing a more computationally complex scheduling function, although we note that the scheduling operation itself can be distributed among the processors. The proposed SADS algorithms overlap
the scheduling process with the execution of tasks to mask the overhead associated with the scheduling process. These algorithms reduce the bottleneck effect of centralized scheduling by pro-actively assigning tasks to working processors while these processors are performing their previously assigned tasks.

The remainder of this paper is organized as follows. In Section 2, we identify the important factors that can affect the execution time of a set of parallel tasks. As part of that section, we also specify the task model and a statement of the problem that we address in the remainder of the paper. Section 3 provides a taxonomy for categorizing and characterizing multiprocessor scheduling strategies. It then provides a discussion of a set of relevant scheduling strategies in the context of the taxonomy. Our on-line scheduling class of algorithms, SADS, is presented in Section 4. Analysis of the behavior of SADS and the expected quality of the schedules that result from performing these techniques are also provided in that section. Section 5 presents simulation results of evaluating this new scheduling strategy by comparing its performance with one of the well-known distributed scheduling techniques. Our results and conclusions are summarized in Section 6.

2. Task Scheduling in Multiprocessor Architectures

2.1. Performance Factors

Techniques for scheduling parallel tasks on to the processors of a multiprocessor system must trade-off four interrelated factors: 1) processor load balance; 2) memory locality; 3) scheduling and synchronization overhead; and 4) memory use and availability. The first three factors are the most important, since they contribute directly to the total execution time of the program, while the last factor has an indirect performance impact.

Since the total execution time of a parallel program is determined by the last processor to finish executing its assigned tasks, it is important to maintain a balanced load among the processors. A perfectly balanced load does not imply that all processors must execute the same number of parallel tasks since the tasks typically have variable execution times. Consequently, it is not sufficient to simply divide the total number of tasks evenly among all of the processors. Another consideration is that each task most likely references a different subset of the data objects declared by the application program. If these data objects are distributed among the local memories of a Non-Uniform Memory Access (NUMA) multiprocessor[4,5,6,7,8], for instance, each parallel task will have an affinity for a particular processor based on the locality of its memory references. Load balancing and memory locality must be traded-off since optimizing for memory locality tends to cluster tasks on a small number of processors, while optimizing for load balancing tends to spread the tasks among the processors.

In a similar fashion, the time required to actually schedule the tasks on to the processors can be in direct conflict with the desire to balance the computational load. For instance, using small task sizes can lead to good load balancing since any number of these small tasks can be used to fill in processor idle times. However, if some of the scheduling operations are performed at runtime, the time required to perform the scheduling will be added directly to the execution time of each task. Since this scheduling time generally is independent of the execution time of the
parallel task, the use of small task sizes can produce very high scheduling overheads, which can eliminate the advantages of using small tasks for load balancing.

The amount of memory available in the system can have an indirect impact on the total execution time by allowing the scheduler to modify and expand task-to-processor affinities. For example, in a NUMA machine with sufficient memory, heavily-shared data objects can be replicated in the memories of several processors. This replication expands the number of processors for which a particular task may have an affinity. This type of replication is exactly what occurs in systems with coherent private data caches[9]. Disadvantages of this replication are the cost of the additional memory, and, perhaps more importantly, the time required to maintain coherence.

2.2. Problem and Scope

In the remainder of this paper we examine some of the trade-offs involved in optimizing the performance factors discussed above in the context of dynamically scheduling a set of tasks, which arrive in the system over time, on to the processors of a NUMA multiprocessor[4,5,6,7,8]. In a NUMA architecture, memory is physically distributed among the processors in the system. A processor, in such architectures, can reference data in its own local memory more quickly than it can reference data in another processor’s memory. By a "task", we mean an atomic unit of computation. In this interpretation, a task can be regarded as a single sequential process. In this paper, we focus on scheduling of a set of independent tasks, since these types of tasks constitute a common class of tasks that occur frequently in dynamic environments. The proposed techniques can be extended, however, to schedule tasks with precedence constraints, and other constraints imposed on the task model.

The problem addressed in this paper can be stated as follows. Given a set of tasks \( t_l \in T \) \( (1 \leq l \leq n) \) arriving into the system over time, a set of processors \( p_m \in P \) \( (1 \leq m \leq p) \) on a NUMA multiprocessor architecture, task processing costs \( cp_{t_l|p_m} \), and task-to-processor memory access costs \( cc_{t_l|p_m} \), the objective is to find, in time \( CS \), a schedule \( S = \{(t_l \ p_m) | t_l \in T, \ p_m \in P \} \) such that the total execution time \( CE = CS + \sum_{(t_l \in T), \ (p_m \in P)} (cp_{t_l|p_m} + cc_{t_l|p_m}) \) is a minimum. The above formula defines the total execution time, \( CE \), as the sum of the scheduling time, \( CS \), and the total time required to execute each of the parallel tasks, which includes the time required for each task to read its necessary data values. Accounting for performance factors such as load balancing and memory locality simultaneously, at run time, in a multiprocessor system is a hard, on-line optimization problem. Solving such a problem faces the challenge of considering the trade-offs between scheduling complexity and schedule quality which we address in this paper.

3. Overview of Existing Work

3.1. A Taxonomy for Scheduling Algorithms

To help in understanding how different scheduling strategies can influence the different performance factors, we propose the following taxonomy for scheduling algorithms. This taxonomy categorizes and characterizes scheduling strategies along the dimensions of adaptability, locus of control, computational effort, memory usage, and task granularity. Each of these dimensions is described below. The taxonomy proposed here can be regarded as an extension to that proposed
Adaptability. Given perfect information at compile-time about the execution time and the memory referencing behavior of the parallel tasks, a static scheduling strategy can theoretically precompute an optimal schedule. Since all scheduling decisions are made at compile-time, there is no run-time overhead to add to the execution time of the program. In many applications, however, this detailed information is simply not available at compile-time. Statically estimating this information may lead to large load imbalances and long average memory delays. To compensate for this lack of a priori information, dynamic scheduling postpones the assignment of tasks to processors until the program is actually executing. The scheduling strategy can then adapt to the dynamically changing conditions encountered at run-time.

Locus of control. Depending on where scheduling decisions are made, scheduling strategies can be categorized as either centralized or distributed. Static strategies are inherently centralized since they locate all scheduling decisions in the compiler. Dynamic centralized strategies assign a single processor to perform all scheduling operations at run-time. With distributed scheduling strategies, on the other hand, each processor is responsible for assigning available tasks to itself when it finishes its previously assigned tasks. This type of distributed scheduling is often referred to as self-scheduling.

Computational effort. The third dimension of this scheduling taxonomy is the total amount of computational time, or effort, spent in assigning tasks to processors. Static scheduling strategies can afford to use complex, time-consuming strategies that produce high-quality schedules since the scheduling operation is performed off-line and thus does not add to the program’s execution time. These complex strategies can lead to unreasonably long compile-times, however. Since the scheduling time with dynamic scheduling directly adds to the task execution time, dynamic strategies tend to use simple, easy-to-compute heuristics at the risk of producing lower-quality schedules. More complex procedures can be used dynamically to improve the schedule quality, but at the cost of higher scheduling overhead.

Data partitioning. This aspect of the taxonomy refers to the effect of the data-partitioning scheme on the quality of the schedule, and it is dependent on the particular multiprocessor architecture being used. For instance, scheduling strategies for NUMA architectures are data-partition dependent in that the data may be completely replicated on all of the processors, it may be only partially replicated, or it may be completely distributed. Each different data partitioning will affect the scheduling strategy by producing an affinity between tasks that reference certain data objects, and the processor memories that contain those objects. Scheduling strategies can vary widely in the degree to which they take into account, or exploit, the data partitioning.

Granularity. This aspect of the framework refers to the size, in terms of execution time, of a task as an atomic unit of scheduling. The granularity in a task model can range from a fine task-grain (e.g. instruction-level granularity) to a coarse task-grain (e.g. program-level granularity). Based on the level of granularity that they address, scheduling algorithms can vary a great deal in their strategies and the heuristics they use.
3.2. Related Work

Scheduling strategies can be classified as either static or dynamic, depending on when the task assignment decisions are made. Static scheduling, also referred to as prescheduling, determines at compile-time which tasks should be executed by which processors. For example, using a simple static strategy, tasks can be assigned to processors in a round-robin fashion such that processor 0 executes tasks 1, p+1, 2p+1, ..., processor 1 executes tasks 2, p+2, 2p+2, ..., and so on for the remaining processors. Using this strategy, each processor executes approximately the same number of tasks, however, this strategy requires that all the tasks in the system be known prior to run time. The run-time scheduling overhead for static scheduling is negligible since processors know exactly which tasks they are to execute based on their processor numbers and the task identifiers.

If complete characteristics of all tasks (including their arrival time, execution costs, inter-dependencies, etc.) were known at compile time, static scheduling could use a complex algorithm to perfectly balance the computational load by appropriately assigning tasks to processors. It is generally impossible, however, to predict or collect complete task characteristics at compile time. An example of such task characteristics is the task execution time. Predicting the execution time of a task is difficult, due to nondeterministic events, such as cache misses, page faults, interprocessor communication delays, and data-dependent program branches. The unpredictability of these events limits the effectiveness of static scheduling.

Dynamic scheduling strategies move the locus of control from the compiler to one or more processors that allocate work to themselves and/or to other processors as needed at run-time to thereby balance the system’s load. In dynamic scheduling, the overhead associated with the task of scheduling can directly affect the performance of the system. Hence, it is important to address issues related to where scheduling is performed, where the information required for scheduling is stored, and how complex the scheduling algorithms can be. Two major approaches that address issues related to the scheduling task itself are distributed and centralized scheduling.

In distributed scheduling [10,11,12,13,14,15,16,17,18] the scheduling task and/or the scheduling information are distributed among the processors and their memories. A common approach to distributed dynamic scheduling is referred to as Self-Scheduling (SS). With self-scheduling, idle processors assign tasks to themselves at run-time by accessing a shared global queue. This shared queue indicates which task should be executed by the next available processor. To prevent more than one processor from executing the same task, access to this queue must be limited to one processor at a time using an appropriate synchronization mechanism. The time required to access this shared queue to remove a task from the common pool of waiting tasks introduces some run-time overhead, but the improved load balancing may compensate for this additional delay.

It is possible to reduce the number of accesses to the critical section (i.e. the global queue) by allocating a chunk of $c$ tasks at a time[19,20,21,22]. Allocating fewer large chunks reduces the scheduling overhead compared to allocating smaller chunks more often, but it also increases the load imbalance due to the coarser granularity of the tasks. The mechanisms proposed for trading off chunk size for scheduling overhead fail to account for scheduling cost explicitly, which may cause inefficient chunking for certain processor and task configurations. The above
and other existing distributed scheduling techniques generally concentrate on load balancing based on task processing times alone and ignore the effects of task-processor affinities. An obvious reason for ignoring the task-processor affinities in scheduling is that simultaneously accounting for both load-balancing and task-processor affinities increases the scheduling complexity. Another reason is that the distributed scheduling algorithms have limited capability in accessing information required for addressing task-processor affinities, particularly if these algorithms are designed to run on NUMA architectures. Despite these problems, distributed approaches have been investigated which employ heuristics for minimizing execution and communication costs [15,16,17,18].

The Affinity Scheduling (AFS) technique [18] can be regarded as an example of a distributed scheduling technique which combines static and dynamic scheduling to maintain good load balancing while also providing for greater memory locality than a purely dynamic scheme for some parallel task structures. In this approach, tasks are equitably assigned to processors initially. Each processor maintains a local work queue of the tasks it has been assigned. When a processor becomes idle, it executes $1/k$ of the tasks in its local queue. Once a processor's local queue becomes empty, it finds the processor with the largest number of tasks in its local queue and removes $\lceil 1/p \rceil$ of the tasks from that processor's local queue and executes them. Small values of $k$ incur small synchronization costs but create a larger potential for load imbalance. As $k$ approaches $p$, load balancing in AFS improves while the number of synchronization operations increases.

The task assignment scheme used by AFS makes assumptions about the memory referencing behavior of tasks which are not valid for many commonly used task models. AFS does not utilize the affinity information of data and instead assumes that data are accessed in a repeatedly occurring pattern. Furthermore, AFS does not consider affinity information when migrating tasks from their initially assigned processor to another processor that has requested additional tasks. These assumptions cause AFS to favor load balancing over affinity when making scheduling decisions. This approach does not account for scheduling costs explicitly and can incur large scheduling costs as the number of processors increases. Selection of the parameter $k$ in AFS is difficult, since it depends on the particular task set and processor configurations.

The self-scheduling techniques discussed earlier are designed for shared-memory multiprocessor architectures. Self-scheduling by different processors is possible in such architectures since every processor has direct access to the shared information. Scheduling in a distributed-memory multiprocessor architecture is a more difficult task than that in an architecture with a globally shared memory due to the cost of accessing, updating, and maintaining consistency of shared information used for scheduling.

Centralized techniques [23,24,25,26,27,28,29,30,31,32,33,34] can remedy some of the problems encountered by distributed techniques for scheduling in NUMA architectures. These techniques can store global information at a centralized location and can use this information to make more comprehensive scheduling decisions using one or more dedicated processor's computing and storage resources. Dynamic scheduling techniques[33,34] have been proposed to designate one or more processors' memory to store this shared information. In these approaches, the need to maintain consistency among several copies of the shared information is eliminated. One of the drawbacks of the existing centralized scheduling approaches is the contention caused by
attempts to access the shared information and to request tasks for execution. This contention may cause the designated processor to become a bottleneck. Hierarchical scheduling architectures have been proposed\[33\] to reduce this bottleneck effect. Another drawback of some of the existing centralized approaches is that they use only the designated processor’s local memory for storing shared information. In addition, these approaches do not exploit the full processing power of the designated processor for performing sophisticated optimization techniques.

All of these approaches employ scheduling techniques that are specific to a particular class of problems and work for a narrow range of parameters. Little research has been done on employing optimization techniques for dynamic task scheduling because of the perception that the prohibitive computational cost of these techniques will render them ineffective for dynamic scheduling. On-line optimization techniques need to be designed that take advantage of useful on-line information while keeping the scheduling costs at a minimum. This paper introduces an on-line optimization technique, SADS, for dynamic task scheduling on NUMA architectures. The proposed technique can be regarded as a centralized scheduling technique since it dedicates a processor for scheduling and for storing shared information. Overlapped scheduling and execution allows SADS to reduce the bottleneck effect of synchronization and data sharing by proactively assigning tasks to processors while the working processors are executing previously-assigned tasks, and by reducing the need to access the scheduling information stored on the scheduling processor’s memory. This is a unique feature of SADS since, in many of the existing approaches, task assignment is initiated by a request from idle processors. The important issue in centralizing SADS, in this paper, is to study the tradeoffs for gaining access to global information and for using the processing power of a processor to perform the scheduling algorithms. Centralization, however, is not an intrinsic characteristic of the algorithm itself. SADS can be parallelized and its computation can be distributed among processors. The decision as to the degree of centralization of SADS is an application-dependent one. The implementation of a distributed version of SADS and studying its effect on performance, however, is out of the scope of this paper and is part of our plan for future phases of this research.

A contribution of the paper is the representation scheme chosen for the specification of the proposed algorithms. This scheme regards scheduling as a search in the space of all possible task-to-processor mappings. An algorithm’s (e.g., SADS) scheduling strategy might be to incrementally choose among alternate mappings of tasks to processors. Another algorithm’s strategy might be to reduce the search complexity by using simple heuristics such as fixed-size chunking or variable-size chunking. Using the proposed taxonomy and our chosen representation scheme for algorithm specification, scheduling algorithms can be categorized and their salient features can be compared. These frameworks also allow consideration of hybrid strategies. One such strategy, for example, would be to have a hybrid between a strictly centralized scheduling and a strictly distributed scheduling. In such an approach, the task of exploring the space of mappings can be shared among a few centralized schedulers each responsible for a cluster of processors. Such a hybrid will be appropriate for considerations such as scalability and fault-tolerance. The study of such approaches is beyond the scope of this paper and is proposed as future work in the conclusion section.
4. On-Line Optimization Techniques for Multiprocessor Scheduling

A system consisting of a scheduling algorithm and a set of processors that execute the parallel tasks can be assumed to perform tasks in two phases, namely the scheduling phase and the execution phase. During the scheduling phase, the scheduling algorithm searches for a solution to a given problem instance. During the execution phase, the system carries out the scheduled tasks to achieve the desired results. The scheduling and execution phases may be sequential, in which case the system plans a complete schedule before executing a single task. This approach is generally more appropriate for static scheduling. Alternatively, scheduling and execution phases may be interleaved or overlapped, in which case partial task assignments are computed by the scheduling algorithm and are placed on processor queues for execution. The interleaved or overlapped scheduling approach is generally used by dynamic scheduling algorithms. In the interleaved case, some processors must remain idle while the scheduling is being performed. In completely overlapped scheduling and execution, the overhead of the scheduling effort is completely masked by execution of previously scheduled tasks. The scheduling phase in overlapped or interleaved scheduling may be repeated several times until all the tasks are scheduled. While the interleaved or overlapped scheduling and execution paradigms may reduce scheduling effort, they can lead to suboptimal schedules[35]. In this section, we introduce a family of on-line optimization techniques that schedule tasks dynamically on the processors of a NUMA architecture. The proposed techniques overlap scheduling and execution to mask the overhead of scheduling at run time.

4.1. Self-Adjusting Dynamic Scheduling (SADS)

SADS utilizes the completely overlapped scheduling and execution paradigm to schedule an independent set of tasks on a set of processors[36]. This technique performs partial task scheduling in repeated periods and places the scheduled tasks on working processors’ local queues until all tasks are scheduled. SADS uses a novel on-line parameter tuning technique to determine the duration of each scheduling phase. The time allocated to a single scheduling phase in this algorithm is self-adjusted based on an on-line, lower-bound estimate of the working processors’ load. During one scheduling phase, the algorithm continues to incrementally build a schedule of tasks until the least-loaded working processor has completed executing the tasks on its local queue. At that point, the scheduling algorithm assigns tasks to all processors based on the partial schedule that it computed during the scheduling phase. Building schedules in an incremental fashion is an important characteristic of SADS which allows the scheduling process to be interrupted at any time to provide a valid partial schedule of tasks. Since, in the last scheduling phase, scheduling is finished before all computational tasks are executed on working processors, the scheduling processor will be free to execute some computational tasks itself. This fact can be accounted for in the last scheduling phase, so that the scheduling processor can assign some tasks to itself. For simplicity, the results presented in this paper are on performance of those versions of SADS which do not incorporate this mechanism.

SADS is an on-line optimization algorithm that like the branch-and-bound algorithm [37] searches through a space of all possible partial and complete schedules. The original branch-and-bound algorithm is capable of finding an optimal solution, but it is computationally expensive. This algorithm has been investigated for static task scheduling in a specific model of a distributed system[38]. It is important to note that SADS differs from the original branch-and-bound
algorithm and that it is significantly faster to execute. This is mainly due to SADS’s ability to
divide the solution space into smaller clusters each of which is relevant only to the current
scheduling phase. In previous experiments, to compare the complexity of branch-and-bound
and SADS, we obtained results of scheduling 50 tasks on 10 processors in a few milliseconds. The
branch-and-bound algorithm failed to produce an answer to the same problem after a few hours
and was terminated. The results of our experiments show that, due to its controlled complexity,
the basic SADS algorithm does not become a bottleneck. The algorithm efficiently schedules
tasks on a large number of processors (i.e. it scales up) and produces improved performance
compared to other existing algorithms. Heuristic variations of SADS are even faster than the basic
SADS and do not show any sign of suffering from the bottleneck effect, in our experiments.

In our task scheduling problem, a space of all partial and complete schedules can be repre-
 sented as a tree G(V,E) consisting of a set of nodes v_j∈V and a set of edges (v_j v_k)∈E, connecting
nodes v_j, v_k∈V. Each node v_j in G represents a partial schedule S_j = {(t_1 p_m)|t_1∈T_i,p_m∈P} that
assigns a set of tasks t_1 in task set T_i of scheduling phase i to corresponding processors p_m of the
set of processors P. An edge (v_j v_k) in G represents the extension of the partial schedule of node
v_j by adding one more task-to-processor assignment to the partial schedule of v_j. The partial
schedule of a successor node v_k of v_j is thus S_k = S_j ∪{t_x p_y}, where p_y∈P and t_x is a task
that has not yet been scheduled as part of S_j. A cost C_k is associated with each node v_k in G. C_k
is calculated as the total execution cost of the partial schedule S_k of node v_k. This total cost is
calculated as the cost ce_max of the most-loaded processor in S_k.

One of the unique features of SADS’s cost model is that it accounts for memory access
 costs as well as processing cost of tasks on their corresponding processors. This allows addressing,
simultaneously, the tradeoffs between the two interacting factors of load balancing and memory
management. The total execution cost of a task t_i on processor p_m, in this cost model, is
ct_i,p_m = cc_{t_i,p_m} + cp_{t_i,p_m}, where cc_{t_i,p_m} is the memory access cost incurred by executing t_i on p_m,
and cp_{t_i,p_m} is the processing cost (CPU bound) of executing t_i on p_m. Note that cc_{t_i,p_m} is large if
the data required by t_i does not reside on p_m’s local memory (i.e. there is a low degree of affinity
between t_i and p_m), and it is small if the data required by t_i is on p_m’s local memory (i.e. there is
a high degree of affinity between t_i and p_m). The total cost of partial schedule S_k is:

C_k = ce_{max} = Max\{ce_m | 1 \leq m \leq p, and ce_m = CE_m(i - 1) + \sum_{all (t_i,p_m) \in S_k} (cc_{t_i,p_m} + cp_{t_i,p_m})\).

In the above formula, p is the number of processors, and CE_m(i - 1) is the total execution cost of
processor p_m at the end of phase i - 1.

Each scheduling phase i consists of one or more SADS iterations. In each SADS iteration,
the node with the least cost is expanded (i.e. its successor nodes are generated). This node expan-
sion continues until either all tasks are scheduled or until the time allocated to the scheduling
period i is consumed. During the scheduling phase i+1, a new search through the space of the
remaining tasks starts and leads to additional task-to-processor assignments. In the above for-

mula, CE_m(i - 1) = ce_m if ce_m is the total execution cost of the processor p_m after the last SADS
iteration of phase i - 1. The difference in notation (i.e. "CE" versus "ce") was adopted to signify
the difference between the cost of a processor’s load during a scheduling phase (i.e. "ce"), and the
cost of that processor’s load once the final schedule of a phase has been determined after the last
SADS iteration of that phase (i.e. "CE").
Figure 2 provides pseudo-code of the general SADS algorithm. The scheduling phase $i$ is terminated when allocated time to schedule in that phase runs out, i.e.

$$CS(i) \geq \alpha CE_{\text{min}}(i - 1),$$

This stopping criterion controls the scheduling time allocated to the current phase ($CS(i)$) as a fraction ($\alpha$) of the total execution cost ($CE_{\text{min}}(i - 1)$) of the least-loaded processor, at the end of phase $i-1$ or just at the beginning of phase $i$. Suppose $v_k$ is the least-cost node after the last node expansion of phase $i-1$. $CE_{\text{min}}(i - 1)$ is calculated as the cost of the least-loaded processor in $S_k$; i.e.

$$CE_{\text{min}}(i - 1) = \text{Min}(\{ce_m|1 \leq m \leq p, \text{ and } ce_m = CE_m(i - 2) + \sum_{\text{all } (t_l, p_m) \in S_k} (cc_{t_l, p_m} + cp_{t_l, p_m})\}).$$

At the beginning of phase 1, $CE_{\text{min}}(0)$ may be zero. Thus, a mechanism has to be adopted to initially assign an appropriate positive value to $CE_{\text{min}}(0)$, or another simple scheduling scheme (e.g. prescheduling) has to be adopted to schedule a few tasks onto processors initially and then set the termination criterion of SADS based on the value of the least-loaded processor in that initial partial schedule. In general, large values of $CE_{\text{min}}(i - 1)$ may allow adequate scheduling time to compute a partial schedule with a large number of tasks. Small values of $CE_{\text{min}}(i - 1)$ may restrict the scheduling time and lead to a few task assignments in phase $i$, reducing SADS to a greedy algorithm. $CS(i)$ may represent elapsed time and may depend on the number of partial schedules examined (i.e. number of tree nodes generated in the search space) during the $i$th phase. Parameter $\alpha$, in that case, can be regarded as the inverse of the cost $\sigma$ of expanding one node in the search space (i.e. the cost of adding one task to the current partial schedule). The expanded nodes are those whose descendants were generated during phase $i$ and were added, in the order of increasing costs, to the list of previously generated nodes. Throughout this paper we assume that $CE_{\text{min}}(i - 1)$ is at least as large as $\sigma$. The smallest amount of scheduling carried out by SADS in any scheduling phase is equal to that of a single branch-and-bound iteration.

4.1.1. Algorithm Characteristics

The following results formalize some characteristics of the SADS algorithm.

**Lemma 1:** SADS reduces to the branch-and-bound algorithm for large $CE_{\text{min}}$.

**Proof:** Suppose that $CE_{\text{min}}(0) = \infty$. This implies that the stopping criterion will never be satisfied to start the second phase, and that the search will terminate when a schedule consisting of a complete assignment of tasks is found. Since SADS consists of a set of branch-and-bound searches, each of which attempt to find a complete schedule from the start node during that phase, a one-phase SADS which terminates when a complete schedule is found is equivalent to the branch- and-bound algorithm.

Note that the result of Lemma 1 implies that SADS finds the optimal complete schedule in terms of total execution cost (sum of memory access costs and processing costs), in this special asymptotic case. It also implies that the expected quality of the schedules found by SADS improves as $CE_{\text{min}}$ increases.

**Lemma 2:** SADS reduces to a greedy local gradient descent algorithm, if $CE_{\text{min}}$ is small.
PROCEDURE SADS(task-set);
VAR
queue,succ_list : queue-of-nodes;
x,current_node: node;
WHILE NOT((solved(head(queue)) OR (stopping_criterion))) DO
BEGIN
   current_node := head(queue);
delete(current_node,queue);
succ_list := successors(current_node);
FOR each x IN succ_list DO
BEGIN
   x.cost := cost(current_node,x);
   IF not_member_of(x,queue) THEN
      insert(x,queue);
END
sort_queue_by_cost(queue);
END
IF no_more_tasks(head(queue)) THEN
announce_success;
ELSE
IF (stopping_criterion) THEN
BEGIN
   assign_partial_schedule(current_node);
   SADS(remaining_task_set);
END
ELSE announce_failure;

Figure 2: Pseudo Code for SADS

Proof This situation will result in satisfying the condition $CS(i) \geq CE_{\min}(i – 1)$ after the first SADS iteration in each phase leading to greedy scheduling phases and local gradient descent algorithm. Each scheduling phase of the greedy algorithm will consist of a single branch-and-bound iteration during which a single task is assigned to a processor.

Theorem 1: Within a single scheduling phase, SADS results in the partial schedule with the least cost, among all other partial schedules examined during that phase.

Proof Each scheduling phase of SADS starts by searching through the space of remaining tasks and updating the current processor cost matrix. The processor cost matrix simply indicates the cumulative load (cost) of each processor up to the current phase. Since during each iteration SADS expands the partial schedule with the least total cost so far, the partial schedule produced during the last iteration of the scheduling phase has the least cost of all other partial schedules investigated during that phase.

Theorem 2: Assume that the set of tasks to be scheduled have variable processing times, and that poor affinity can exist among the tasks and the processors to which they are assigned (e.g. all data is not replicated on all processors). Under SADS, all processors will finish within the time of the longest-running task.

Proof The proof of this theorem is by induction on scheduling phase $i$. We take the base step of the proof to be phase 1, at the beginning of which the processor queues are empty. In the asymptotic case where each scheduling phase of SADS results in only one task assignment (the greedy case), the load will be unbalanced by at most $ct_{\max}$ (i.e. the cost of the
task with the largest total execution time), since all processors are idle. In other cases where a scheduling phase results in more than one task-to-processor assignment, the resulting partial schedule in phase 1 is the one with the least cost, among all other schedules examined, due to Theorem 1.

In the induction step, we assume that during phase i the load is imbalanced by \( c_{t_{\text{max}}} \). In the asymptotic case in phase \( i+1 \), where each scheduling phase of SADS results in only one task assignment (the greedy case), SADS will put the task on a processor other than the one causing the imbalance as it always looks to minimize the cost of the current partial schedule with respect to the current processor costs. In other cases, where scheduling phase \( i+1 \) results in more than one task-to-processor assignment, SADS will again compute the least-cost partial schedule with respect to the current processor costs in that phase. This result also follows from Theorem 1. The worst-case scenario is when at the beginning of phase \( i+1 \), all processor loads are perfectly balanced, and the scheduling time allocated to that phase allows only one task to be scheduled on one of the processors. In that case, there will be imbalance in the resulting schedule as much as the size (cost) of a single task.

4.2. Variations of SADS

This section presents two variations of the basic SADS algorithm which reduce scheduling complexity by reducing the number of nodes to explore in the solution space[39].

4.2.1. Minimum-Remaining-Execution-Time Heuristic (MESADS)

It has been shown[37] that an extension of the original branch-and-bound algorithm, A*, can utilize heuristic estimates of the remaining costs from a node in the solution space to the destination to reach solutions while examining fewer nodes. The optimality of the solution found depends on a characteristic of the heuristic estimate referred to as monotonicity. A heuristic is monotone if it consistently produces an underestimate of the remaining execution cost of a node under examination in the solution space. Given a monotone heuristic, A* will find the optimal solution while expanding a minimum number of nodes[37].

Recall that the actual cost of a partial schedule in SADS can be calculated using the cost function provided in the previous section. A heuristic function for the SADS algorithm that will provide an estimate of the total execution cost of a complete schedule, given an intermediate partial schedule, is the sum of this actual cost and the estimated cost of the remaining task-to-processor assignments. Since for each partial schedule represented by a node the cost of executing the remaining tasks depends on the execution cost of each task on a processor and the degree of affinity of that task with the corresponding processor, a monotone heuristic estimate for the MESADS algorithm can be developed which calculates the estimated cost by considering the minimum task execution costs without taking into account any memory affinity information.

Specifically, given a set of \( n \) tasks \( (t_1, \ldots, t_n) \), assume that \( S_j = \{(t_1, p_x), \ldots, (t_k, p_y)\} \) is a partial schedule assigning the first \( k \) tasks to processors. The monotone heuristic estimate of the remaining tasks’ execution cost is calculated by dividing the sum of lower-estimates of execution times of the remaining tasks by the number of processors, that is, \( h(S_j) = (\sum_{i=k}^{n} c_{p_i}) / p \), where \( c_{p_i} \) is
the minimum processing cost of task \( i \) on any processor and \( p \) is the number of processors. This heuristic estimate is a lower bound on the actual cost and will allow MESADS to produce optimal solutions in the asymptotic case where \( CE_{\text{min}} \) is large. We show experimentally, in the following sections, the effect of this heuristic on the quality of solutions found by MESADS and its effect on the scheduling effort expended to find such solutions. Figure 3 shows the pseudo-code for the MESADS algorithm.

```
PROCEDURE MESADS(task-set);
VAR
  queue, succ_list : queue-of-nodes;
  x, current_node : node;
WHILE NOT((solved(head(queue)) OR (stopping_criterion))) DO
  current_node := head(queue);
  delete(current_node, queue);
  succ_list := successors(current_node);
  FOR each x IN succ_list DO
    BEGIN
      x.cost := cost(current_node, x);
      x.heuristic := estimated_remaining_execution_time(x);
      x.heuristic_function := x.cost + x.heuristic;
      IF not_member_of(x, queue) THEN
        insert(x, queue);
    END
  sort_queue_by_heuristic_function(queue);
END
IF no_more_tasks(head(queue)) THEN
  announce_success;
ELSE
  IF (stopping_criterion) THEN
    BEGIN
      assign_partial_schedule(current_node);
      MESADS(remaining_task_set);
    END
  ELSE
    announce_failure;
```

Figure 3: Pseudo Code for MESADS

4.2.2. Depth-Bound SADS (DBSADS)

In the previous subsection, we discussed the utilization of heuristic information to reduce the effort to search for solutions by examining fewer nodes (partial schedules) in the search space of the problem. Another method of reducing the number of nodes examined is to focus the scheduling effort on a particular part of the search space by changing the order in which nodes are examined. Another version of SADS, which we refer to as Depth-Bound SADS (DBSADS), combines an on-line version of depth-first search with the original branch-and-bound method to give priority to examining nodes that reside at a higher depth (i.e. closer to leaves). In effect, DBSADS gives higher priority to examining partial schedules in which a larger number of tasks have been assigned. This bias in the order of examining the nodes is aimed at reducing the branching ratio of the basic SADS algorithm to limit examination of partial schedules with fewer tasks which naturally incur lower costs.
After each node expansion in the solution space, the DBSADS algorithm first orders the children of the current node based on their cost values and then adds them to the front of the queue of nodes to be expanded. Note that the main difference compared to the order in which the basic SADS algorithm examines nodes is that in SADS, the children are first added to the queue and then the entire queue is sorted based on the costs of the children and the costs of the previously generated nodes. DBSADS is expected to generate significantly fewer nodes in the scheduling problem addressed in this paper. This algorithm, however, does not guarantee reaching an optimal solution, even within a single scheduling phase.

**Lemma 3:** Given $n$ tasks to be scheduled on $p$ processors, DBSADS generates and evaluates $n \times p$ nodes of the solution space, in order to schedule all $n$ tasks on the $p$ processors.

**Proof** Each level of the search tree $G(V, E)$ corresponds to assignment of a task to a processor. Thus, the tree has $n$ levels. Since at each level $p$ nodes corresponding to the $p$ processors are generated, DBSADS will have generated a total of $n \times p$ nodes by the time all tasks are scheduled.

In later sections, we experimentally study the effect of this node examination order on the quality of solutions found by DBSADS, and its effect on the total scheduling effort. Figure 4 shows the pseudo-code for the DBSADS algorithm.

```
PROCEDURE DBSADS(task-set);
    VAR
        queue,succ_list : queue-of-nodes;
        x,current_node: node;
    WHILE NOT((solved(head(queue)) OR (stopping_criterion))) DO
        BEGIN
            current_node := head(queue);
            delete(current_node,queue);
            succ_list := successors(current_node);
            FOR each x IN succ_list DO
                BEGIN
                    x.cost := cost(current_node,x);
                END
            sort_succ_list_by_cost(succ_list);
            IF not_member_of(x,queue) THEN
                concatenate(succ_list,queue);
            END
            IF no_more_tasks(head(queue)) THEN
                announce_success;
            ELSE
                IF (stopping_criterion) THEN
                    BEGIN
                        assign_partial_schedule(current_node);
                        DBSADS(remaining_task_set);
                    END
                ELSE announce_failure;
        END
```

Figure 4: Pseudo Code for DBSADS
5. Experimental Evaluation

In our simulated experiments, we compared the performance of the basic SADS algorithm with that of a distributed scheduling algorithm similar to the affinity scheduling (AFS) algorithm [18] on a set of parallel tasks. The comparison studies of basic SADS and AFS provide insight into the tradeoff of dedicating a processor to scheduling and the quality of the schedules relative to the performance of the distributed scheduling algorithms. We chose AFS for our comparison studies since AFS is one of the few distributed dynamic scheduling algorithms that attempts to address load balancing and memory access delays simultaneously. Note that, in all the results presented in this section, SADS, MESADS, and DBSADS have been penalized for dedicating a processor to scheduling, while AFS’s scheduling and synchronization costs have been ignored.

In another set of experiments, we compare the performance of the SADS, MESADS, and DBSADS algorithms in terms of their scheduling efforts and the quality of schedules that were produced by these algorithms. We remind the reader that the scheduling effort does not contribute to the total execution costs of the SADS class of algorithms, since this cost is masked by overlapping scheduling and execution. This cost can be regarded as the average length of the scheduling periods in each algorithm and can provide insight into the tradeoff between scheduling efforts and schedule quality in the comparison studies of SADS, MESADS, and DBSADS.

We compared the algorithms under different locality (affinity) patterns among tasks and processors to demonstrate the capability of SADS to adjust to different patterns of locality. We also compared the algorithms under different degrees of variance in task execution times to test the effect of regular and irregular task execution times on the performance of the two algorithms. We evaluated the degree of imbalance in SADS’s schedules to confirm the analytical results achieved in earlier sections. A major point of observation in our experiments was to find out at what number of processors the tradeoff of dedicating a processor to scheduling pays off by producing better overall schedules despite the absence of one of the potentially working processors.

5.1. Methodology

Each problem instance in the experiments consists of a set of tasks which are assumed to arrive into the system over time, a set of processors, a task-processor affinity matrix, and task processing times. The parameters of our experiments include the number of tasks, mean task processing times, standard deviation of task processing times, number of processors, local and non-local memory access times, and degree of affinity between tasks and processors reflected in the task-processor affinity matrix. Degree of affinity refers to the probability that a given task has memory affinity with a given processor. A low degree of affinity in our experiments signifies that the probability that a given task has affinity with a given processor is 1/p (i.e. the data resides on only one of the processors’ local memory). A medium degree of affinity signifies that the probability that a given task has affinity with a given processor is 50% (i.e. half of the p processors have affinity with any task). A high degree of affinity signifies total-replication of data on all processors’ local memories so that the probability that a task has affinity with a given processor is 100%.

For each of our experiments we generated 500 independent tasks. The arrival rate of tasks was chosen to be high enough to ensure that there were always enough tasks in the system, so that
the scheduler does not have to remain idle when waiting for additional tasks to arrive. The high arrival rates also help us to test the algorithms’ ability to keep up with high loads during bursty arrival of tasks. The number of processors ranged from 2 to 51. The mean processing time of each task was chosen to be 100 cycles and the standard deviations were chosen to be 10 and 400. The ratio of local to non-local memory access time was 100. These parameter values are based on values reported elsewhere in the literature [40]. The parameter $k$ of the AFS algorithm was chosen to be equal to $p$, the number of processors, in all of the experiments. Scheduling costs in AFS are assumed to be zero. The scheduling costs of SADS, MESADS, and DBSADS, on the other hand, are modeled by dedicating one processor to perform the scheduling, leaving $p - 1$ processors to execute the parallel tasks. The scheduling costs of SADS, MESADS, and DBSADS during each scheduling phase were measured by computing the number of partial schedules generated and examined during that scheduling phase. Measuring the actual scheduling effort in units of time is difficult and implementation-dependent. The number of generated nodes provides an implementation-independent metric for measuring scheduling effort and is used widely in the literature. Schedule quality is measured as the speed-up and the total execution time of the complete schedules. Degree of load imbalance is calculated by

$$\frac{CE_{\text{max}} - CE_{\text{min}}}{ct_{\text{max}}},$$

where $CE_{\text{max}}$ is the total cost of the most-loaded processor in the complete schedule, $CE_{\text{min}}$ is the total cost of the least-loaded processor in the complete schedule, and $ct_{\text{max}}$ is the task in the task set that has the highest total execution time. Note that according to the above formula, and the analytical results of the previous sections, the degree of imbalance of SADS is expected to range between zero and one.

5.2. Effect of SADS on Speed-Ups

Figures 5 through 10 present the results of the experiments comparing AFS, SADS, MESADS, and DBSADS under different degrees of affinity and different variabilities in task processing costs.

Figures 5 and 6 show the results of comparing AFS, SADS, MESADS, and DBSADS under high degrees of affinity with low and high variance in task processing times, respectively. As is shown in Figure 5, under a high degree of affinity with low variance in task execution times, AFS generally demonstrates better overall speedups for both small and large numbers of processors. This result is due mainly to the fact that the need for locality management is diminished when the data are totally replicated on all processors. With the low variance in task execution times, the static initial partitioning of tasks among processors in AFS also seems to result in reasonable speed-ups. As the variance in task execution times increases (as shown in Figure 6), the SADS family of algorithms outperforms AFS even at a small number of processors.

Figures 7 and 8 show the results of comparing AFS and SADS under medium degrees of affinity with low and high variance in task processing times, respectively. As is shown in Figure 7, the performance of SADS is close to that of AFS for small numbers of processors. SADS starts outperforming AFS at 25 processors. At high variance in task execution times, as shown in Figure 8, the SADS algorithms outperform AFS even with a small number of processors,
demonstrating the importance of producing high-quality, adaptive schedules, even at the expense of removing a single processor from performing the computational tasks of the application program.

Finally, Figures 9 and 10 show the results of comparing AFS and SADS under a low degree of affinity with low and high variance in task processing times, respectively. As is shown in the figures, SADS outperforms AFS starting from 3 processors onwards. This is a significant improvement considering the fact that at three processors, over 30% of the system’s processing power in SADS is used to perform the centralized scheduling. The results of these experiments demonstrate the reasonable tradeoffs of employing sophisticated scheduling techniques even in small- to medium-sized machines.

5.3. Sensitivity to Affinity and Variance

Figures 11 and 12 demonstrate the effect of affinity on AFS and on SADS. As shown in the figures, AFS is more sensitive to affinity, in that it performs more poorly as affinity degrades, whereas SADS does not seem to be affected as much by different degrees of affinity. Figures 13 and 14 demonstrate the effect of variance in task execution times on AFS and on SADS. As is shown in the figures, AFS is more sensitive to variance in task execution times than SADS. AFS performs more poorly than SADS when there exists a high degree of variance in execution times among tasks. Note also that at $k = p$, AFS incurs the smallest degree of load imbalance, and is expected to perform even more poorly for smaller values of parameter $k$, and large variances in task execution costs.

5.4. Effect of Heuristics on Scheduling Effort and Speed-Ups

Figures 15 through 20 show the results of comparing the scheduling effort of SADS, MESADS, and DBSADS under high degrees of affinity with low and high variance in task processing times, respectively. As shown in the figures, major improvements are obtained by using heuristic information, particularly for larger numbers of processors. The DBSADS algorithm outperforms SADS and MESADS in terms of scheduling costs. Improvements of up to 85% over SADS can be achieved by DBSADS with larger numbers of processors. This improvement is quite significant considering the exponential growth of the search space as the number of processors increases. Another attractive feature of DBSADS is the predictable behavior of its scheduling costs. Such a smooth curve facilitates predicting scheduling costs during each phase and the prediction of the load (average number of tasks per processor) on the system during each execution phase. Note, however, that DBSADS does not guarantee optimal solutions within a single scheduling phase or in general. In applications where a guarantee on the optimality of the execution cost of schedules is desirable, it maybe worthwhile to use MESADS rather than DBSADS, despite its higher scheduling costs.

As shown in Figures 5 through 10, the speed-ups of schedules resulting from SADS, MESADS, and DBSADS are within 10% of each other. This result indicates that, by using heuristics, such as the lower-bound estimate of remaining execution times and the depth-bound node ordering, significant savings in scheduling costs can be obtained without sacrificing the quality of the resulting schedule. In these experiments, some problem instances were found for which the speed-up of the schedules found by MESADS and DBSADS was higher than those
found by the SADS algorithm. This, we believe, can be due to the fact that given an equal amount of time to schedule, MESADS and DBSADS can reach deeper levels in the state space graph and can thus schedule a larger number of tasks in each phase, which in turn leads to producing fewer scheduling phases. With fewer scheduling phases the quality of the schedules will be higher, since information about a more complete set of tasks is considered.

5.5. Load Imbalance

The analytical results of the previous sections predict a maximum imbalance degree of 1 for SADS. This result is consistent with the data collected from all of our experiments, as demonstrated in Figure 21. As is shown in the figure, the degree of imbalance in SADS is always within the size (in terms of total execution time) of the largest task in the task set.

6. Conclusions and Future Research

In this paper, we have introduced a class of on-line optimization techniques, referred to as SADS, for scheduling parallel tasks on the processors of a NUMA architecture. As mentioned in earlier sections, these techniques have much lower computational and space complexities than other traditional optimization techniques (e.g. branch-and-bound). The proposed techniques are capable of improving the performance of programs executed on NUMA architectures. The SADS algorithms perform partial scheduling in repeated scheduling periods, while running on a dedicated processor. These algorithms self-adjust the amount of time allocated to each scheduling period to minimize the processor idle times. The algorithms employ a unified cost model to evaluate partial and complete schedules which accounts for memory delay times as well as CPU processing times. We have designed different heuristic strategies to control the scheduling cost in each scheduling period. The MESADS algorithm’s strategy uses an underestimate of the execution cost of the tasks to be scheduled. This algorithm incurs much smaller scheduling costs than the basic SADS algorithm, while producing schedules which result in similar total execution times as the basic SADS. The DBSADS algorithm’s strategy gives higher priority to the partial schedules with more task-to-processor assignments thereby producing a combination of an online depth-first search and the branch-and-bound techniques.

The results of our experiments show significant improvements in program speed-ups and execution times over existing dynamic distributed approaches even when the synchronization and scheduling costs of the existing approaches are ignored. This work demonstrates that, even with a small number of processors, performing a sophisticated scheduling technique on a dedicated processor can produce substantial improvements in total execution times. The dedicated processor will not become a bottleneck due to the overlapping of scheduling with other processing, and due to pro-active assignment of tasks to working processors. The pro-active assignment of tasks eliminates the need for working processors to send requests for more tasks to the scheduling processor when they (the working processors) become idle. Our experiments also show that using heuristic information can significantly reduce the scheduling costs in each scheduling period without increasing the execution time of the resulting schedules. DBSADS typically incurs the lowest scheduling costs while producing only slightly poorer schedules. MESADS incurs lower scheduling costs than the basic SADS algorithm while actually improving the average execution time slightly.
As part of our future work we plan to develop accurate models of synchronization costs and
to evaluate our technique by measuring the scheduling overhead incurred by the scheduling algo-
rithms, as well as the processing times and the memory delay times. We expect even greater per-
formance improvement, compared to existing approaches, when using SADS, MESADS, and
DBSADS algorithms with the scheduling costs taken into account. Our future work also involves
the study of scalability and fault-tolerance of a variation of SADS algorithms in which the task of
exploring the space of task-to-processor mappings can be shared among more than one central-
ized scheduler each responsible for scheduling tasks on a cluster of processors.

7. Acknowledgments

This work was supported in part by The Research Grant Council of Hong Kong Grant No.
DAG-93/94.EG03., by The Minnesota Supercomputer Institute, and by The National Science
Foundation Grant No. MIP-9221900. We would like to thank the referees for taking the time to
review this paper. We look forward to receiving their comments.

References

2. T.L. Casavant and J.G. Kuhl, “A Taxonomy of Scheduling in General-Purpose Distributed Comput-
3. M.G. Norman and P. Thanisch, “Models of Machines and Computation for Mapping in Multicom-
7. J.G. Lipovski and M. Malek, Parallel Computing: Theory and Comparisons, John Wiley and Sons,
1987.
9. D.J. Lilja, “Cache Coherence in Large-Scale Shared Memory Multiprocessors: Issues and Compar-
159-178, June 1991.
Computers,” IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 4, pp. 979-993,
September 1993.
15. M.S. Squillante and E.D. Lazowska, “Using Processor-Cache Affinity Information in Shared-
Figure 5.: Speedups of AFS and SADS Algorithms in High Affinity and Low Variance

Figure 6.: Speedups of AFS and SADS Algorithms in High Affinity and High Variance
Figure 7.: Speedups of AFS and SADS Algorithms in Medium Affinity and Low Variance

Figure 8.: Speedups of AFS and SADS Algorithms in Medium Affinity and High Variance
Figure 9.: Speedups of AFS and SADS Algorithms in Low Affinity and Low Variance

Figure 10.: Speedups of AFS and SADS Algorithms in Low Affinity and High Variance
Figure 11.: Execution times of AFS in Different Degrees of Affinity

Figure 12.: Execution times of SADS in Different Degrees of Affinity
Figure 13.: Execution times of AFS in Different Degrees of Variance

Figure 14.: Execution times of SADS in Different Degrees of Variance
Figure 15: Search Effort of SADS Algorithms in High Affinity and Low Variance

Figure 16: Search Effort of SADS Algorithms in High Affinity and High Variance
Figure 17.: Search Effort of SADS Algorithms in Medium Affinity and Low Variance

Figure 18.: Search Effort of SADS Algorithms in Medium Affinity and High Variance
Figure 19.: Search Effort of SADS Algorithms in Low Affinity and Low Variance

Figure 20.: Search Effort of SADS Algorithms in Low Affinity and High Variance
Figure 21.: Degree of Load Imbalance in SADS