Dynamic Scheduling Strategies for Shared-Memory Multiprocessors

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 shared-memory multiprocessor 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. To overlap the time required to dynamically schedule tasks with the execution of the tasks, we examine a class of Self-Adjusting Dynamic Scheduling (SADS) algorithms that centralizes the assignment of tasks to processors. These algorithms dedicate a single processor of the multiprocessor to perform a novel on-line branch-and-bound technique that dynamically computes partial schedules based on the loads of the other processors and the memory locality (affinity) of the tasks and the processors. Our simulation results show that this centralized scheduling outperforms self-scheduling algorithms even when using only a small number of processors.

1. Introduction

When scheduling the independent tasks of a single program on to the processors of a shared-memory multiprocessor system with the goal of minimizing the total execution time, the scheduling strategy must equitably distribute the computational load among the processors, and it must minimize the average memory delay by considering the locality of memory references. In addition, these factors must be balanced with the cost of performing the scheduling operation itself. Given perfect information at compile-time about the execution time and the memory referencing behavior of the individual tasks, a *static scheduling* strategy can precompute an optimal schedule. Since all scheduling decisions are made at compile-time, there is no run-time overhead associated with static scheduling. In many applications, however, the necessary information is not available at compile time, which may lead to load imbalances and longer 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 executing. Scheduling decisions are then adjusted to match the dynamically changing conditions encountered at run-time. Centralized [1][2][3][4] and distributed [5][6][7][8][9] dynamic scheduling strategies have been proposed. Typically, the distributed strategies distribute the scheduling operation to the processors so that idle processors assign work to themselves from a central or distributed queue of available tasks. The centralized strategies consider storing global information at a centralized location and using this information to make more comprehensive scheduling decisions. To do this they use one or more dedicated processor's computing and storage resources. A major issue to take into account in designing distributed or centralized dynamic scheduling algorithms is the overhead associated with executing such algorithms. In many of the existing techniques, the time required to perform this scheduling adds directly to the total execution time of the program.

To overlap the time required to dynamically schedule tasks with the execution of the tasks themselves, we propose the Self Adjusting Dynamic Scheduling (SADS) algorithm. SADS is a *centralized* scheduling strategy that assigns a single processor of the multiprocessor system to perform a novel on-line branch-and-bound algorithm to compute partial schedules based on the loads of the other processors and the memory locality (affinity) information of the tasks and the processors. In this paper, we introduce and compare several versions of SADS that use heuristic strategies to reduce the amount of work performed at each scheduling step, which can improve the resultant schedules.

The remainder of the paper is organized as follows. Section 2 describes the class of SADS algorithms and their heuristic strategies. Section 3 presents the experimental evaluations that we use to compare the performance of these new heuristics to an affinity scheduling strategy (AFS) similar to that proposed in[9]. The results and conclusions are summarized in Section 4.

2. Self-Adjusting Dynamic Scheduling Algorithms

A system consisting of a scheduling algorithm and a set of processors that execute the parallel tasks can be assumed to perform operations 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, while during the execution phase, the system executes the scheduled tasks. The scheduling and execution phases may be sequential, in which case the system statically plans a complete solution before executing a single task. Conversely, the 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. In the interleaved case, some processors remain idle while the tasks are scheduled. With completely overlapped scheduling and execution, the overhead of the scheduling effort is completely masked by the execution of previously scheduled tasks. The scheduling phase in an overlapped or an interleaved scheduling approach may be repeated several times until all of the tasks are scheduled. While the interleaved or overlapped scheduling and execution paradigms may reduce the scheduling overhead, they can lead to sub-optimal solutions[10].

In this section, we present a class of SADS algorithms that employ the overlapped-scheduling-and-execution paradigm. The algorithms differ in their utilization of heuristic information and in the order in which they search for partial schedules. The first subsection discusses a general version of SADS and its cost model. In the second subsection we introduce a heuristic function that facilitates the search for partial schedules by pruning the infeasible branches early on in the scheduling process. The third subsection discusses a version of SADS that employs a depth-bound strategy in searching for partial schedules.

2.1. Self-Adjusting Dynamic Scheduling (SADS)

SADS utilizes the completely overlapped scheduling and execution paradigm to schedule an independent set of tasks on to a set of processors. This technique performs partial task scheduling in repeated periods and places them 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 what is known on-line about the load on the working processors. During one scheduling phase, the algorithm continues to schedule tasks until the least-loaded 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. This method of allocating scheduling time reduces processor idle times by completely overlapping scheduling and execution. In this scheme, no processor will be idle while waiting for a task assignment.

SADS is an on-line branch and bound algorithm [11] that 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 making it suitable only for static scheduling. The branch and bound method has been investigated for static task scheduling in a specific model of a distributed system[12]. In our task scheduling problem, a space of all partial and complete schedules can be represented as a tree G(V,E) consisting of a set of nodes $v_j \varepsilon V$ and a set of edges $(v_j\ v_k) \varepsilon E$ connecting nodes v_j, v_k. Each node v_j in G represents a partial schedule $S_j = \{(t_l\ p_m) | t_l \varepsilon T_i, p_m \varepsilon P\}$ that assigns a set of tasks t_l 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 $S_k = S_j \cup \{(t_x\ p_y)\}$, where $p_y \varepsilon P$ 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 T. 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 this cost model is that it accounts for communication cost as well as the processing cost of tasks to tradeoff load balancing and communication overhead. The total execution cost of a task t_l on processor p_m in this cost model is $cc_{t_l p_m} + cp_{t_l p_m}$, where $cc_{t_l p_m}$ is the communication cost incurred by executing t_l on p_m, and $cp_{t_l p_m}$ is the processing cost of executing t_l on p_m. Note that $cc_{t_l p_m}$ is large if the data required by t_l is not placed on p_m's local memory at compile time (i.e. there is a low degree of affinity between t_l and p_m), and it is small if the data required by t_l is on p_m's local memory (i.e. there is a high degree of affinity between t_l and p_m). The total cost of S_k is $C_k = ce_{\max} = Max(ce_m | 1 \le m \le p)$, where $ce_m = \sum_{all\ (t_l\ p_m) \varepsilon S_k} (CE_m(i-1) - CS(i) + cc_{t_l p_m} + cp_{t_l p_m})$, p is the number of processors, $CE_m(i-1)$ is the total execution cost of processor p_m at the end of phase $i-1$, and $CS(i)$ is the time allocated to scheduling in phase i.

Each scheduling phase in SADS consists of one or more iterations. In each iteration, SADS expands (generates the successor nodes of) the node with the least cost in that iteration. This node expansion continues until either all tasks are scheduled or until the time allocated to the scheduling period i expires. During scheduling phase $i+1$, a new search through the space of the remaining tasks begins and leads to additional task-to-processor assignments. In the above expression, $CE_m(i-1) = ce_m$ if ce_m is the total execution cost of processor p_m after the last 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 iteration of that phase (i.e. *CE*).

Scheduling phase i is terminated when the time allocated to schedule in that phase expires; that is, when $CS(i) \geq \alpha CE_{\min}(i-1)$. This stopping criterion controls the scheduling time allocated to the current phase, $CS(i)$, as the fraction α of the total execution cost, $CE_{\min}(i-1)$, of the least-loaded processor at the end of phase $i-1$, or at the beginning of phase i. Suppose v_k is the least-cost node after the last node expansion of phase $i-1$. $CE_{\min}(i-1)$ is calculated as the cost of the least-loaded processor in S_k giving $CE_{\min}(i-1) = Min(ce_m | 1 \leq m \leq p)$, where $ce_m = \sum_{all\ (t_l\ p_m) \epsilon S_k} (CE_m(i-2) - CS(i-1) + cc_{t_l p_m} + cp_{t_l p_m})$. Figure 1 shows the pseudo-code for the SADS algorithm.

At the beginning of phase 1, $CE_{\min}(0)$ maybe zero. Thus, some mechanism must initially assign an appropriate positive value to $CE_{\min}(0)$, or another simple scheduling scheme must preschedule a few tasks onto processors 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 $CE_{\min}(i-1)$ may allow adequate scheduling time to compute a partial schedule with a large number of tasks. Small values of $CE_{\min}(i-1)$ may restrict the scheduling time and lead to a few task assignments in phase i, thus reducing SADS to a greedy algorithm. $CS(i)$ may represent elapsed time and may depend on the number of nodes expanded during the ith phase. Parameter α, in that case, can be regarded as the inverse of the cost σ 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 $CE_{\min}(i-1)$ is at least as large as σ.

We have shown that SADS finds the optimal complete schedule in terms of total execution cost (sum of communication costs and processing costs) in a special asymptotic case where $CE_{\min}$ is large. In general, processors will finish within the total execution time of the largest task; that is, $ct_{\max} = cc_{\max} + cp_{\max}$. This bound on load-imbalance of SADS is at least as low as that of AFS, which can be as high as N/p[9].

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

It has been shown[11] that an extension of the original branch and bound algorithm, A*, can utilize heuristic estimates of the remaining costs from a node to the destination to reach solutions while expanding

```
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 1: Pseudo Code for SADS

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. Given a monotone heuristic, A* will find the optimal solution while expanding a minimum number of nodes[11].

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 at each 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 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 $(t_1, \cdots t_n)$, assume that $S_j = \{(t_1\ p_x), \cdots, (t_k\ p_y)\}$ 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, $h(S_j) = (\sum_{i=k}^{n} cp_i)/p$, where cp_i is the minimum processing cost of task i on a 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 $CE_{\min}$ is large. We show experimentally in Section 4 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 2 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
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);
    x.heuristic := estimated_remaining_execution(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 2: Pseudo Code for MESADS

2.3. 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 online 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 partial schedules with fewer tasks which naturally incur lower costs.

After each node expansion, 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 whole 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. In Section 4, 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 3 shows the pseudo-code for the DBSADS algorithm.

3. Experimental Evaluation

3.1. Methodology

In our simulated experiments, we compared the performance of the basic SADS algorithm with that of the affinity scheduling (AFS) algorithm on a set of parallel tasks. The comparison studies of basic SADS and AFS are aimed at providing a study of centralized scheduling algorithms and self-scheduling. This study provides insight into the tradeoff of dedicating a processor to scheduling and the quality of the schedules.

In another set of experiments, we compared the performance of the SADS, MESADS, and DBSADS algorithms in terms of their scheduling efforts and in terms of the quality of schedules that were formed by these algorithms. We would like to 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 the scheduling and execution. This cost can be regarded as the average

```
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;
```

Figure 3: Pseudo Code for DBSADS

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 SADS and AFS under different locality (affinity) patterns among tasks and processors, in order to demonstrate the capability of SADS to adjust to different patterns of locality. A major point of observation in our experiments were to find out at what number of processors the tradeoff of dedicating a processor to scheduling pays off by producing better overall schedules despite absence of one of the potentially working processors.

Each problem instance in the experiments consists of a set of tasks, 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.

An important point to note here is that the term "degree of affinity" in our experiments is not used in a general sense and has a specific algorithm-dependent meaning. Degrees of affinity ranging from high to low are defined in the context of the affinity requirements assigned by the AFS algorithm. A high degree of affinity, for example, refers to the situation where the task-processor affinities perfectly match that of AFS's initial task-to-processor assignment (i.e. all of the first $\lceil N/p \rceil$ iterations have affinity with only the first processor; all of the second $\lceil N/p \rceil$ iterations have affinity with only the second processor and so on). A low degree of affinity, on the other hand, means that none of the initially assigned tasks in AFS were local to their corresponding processors, and thus would incur non-local memory access costs if they were processed on those processors. In high affinity, AFS can afford to ignore locality information and compute good-quality schedules. In low affinity, however, AFS continues to ignore locality information which results in poorer-quality schedules.

For each of our experiments we generated 500 independent tasks. The number of processors ranged from 2 to 20. The mean processing time of each task was chosen to be 100 CPU clock cycles and the standard deviation was chosen to be 30. The ratio of local to non-local memory access time was 100 clock cycles. These parameter values are based on values reported elsewhere in the literature [13]. 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 nodes examined during that scheduling phase. Measuring the actual scheduling effort in units of time is difficult and implementation dependent. The number of examined nodes provides an implementation-independent metric for measuring scheduling effort and is used widely in the literature. Schedule quality is measured as the total execution time of the complete schedules.

3.2. Effect of Centralized Scheduling on Execution Times

Figure 4 shows the results of comparing AFS and SADS under high degrees of affinity. A high degree of affinity implies that all of the initially scheduled tasks by AFS are local only to their corresponding processors (i.e. they are local to no other processor than the one they are assigned to initially). As is shown in the figure, under a high degree of affinity, AFS outperforms SADS for a small number of processors. This result is due partly to the fact that SADS is more heavily penalized for dedicating a processor to scheduling when the total number of such processors is small. However, as the number of processors increases, the performance of the two algorithms converge. For a larger task set on a

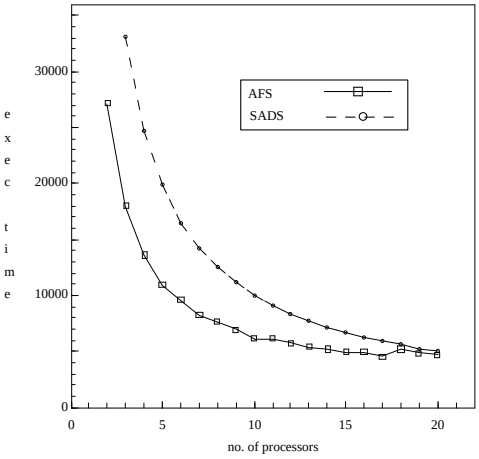

Figure 4.: Execution times of SADS and AFS in High Affinity

larger number of processors, SADS is expected to outperform AFS even at high degrees of affinity.

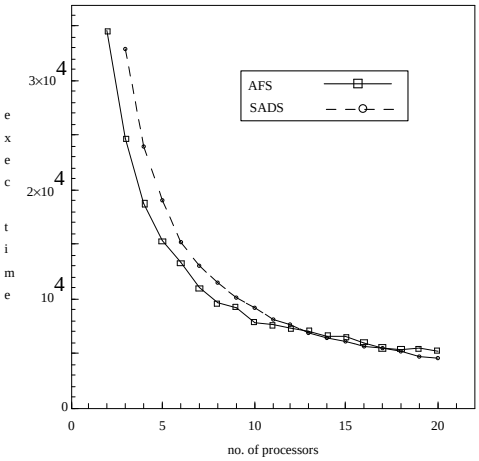

Figure 5.: Execution times of SADS and AFS in Medium Affinity

Figure 5 shows the results of comparing AFS and SADS under medium degrees of affinity. A medium degree of affinity implies that about half of the initially scheduled tasks by AFS are local to their corresponding processors. As is shown in the figure, the performance of SADS is close to that of AFS even for small numbers of processors. SADS starts outperforming AFS at 12 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, figure 6 shows the results of comparing AFS and SADS under a low degree of affinity. A low degree of affinity implies that none of the initially scheduled tasks by AFS are local to their corresponding processors (i.e. they are local to a processor other than the one to which they are initially assigned). As is shown in the figure, 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

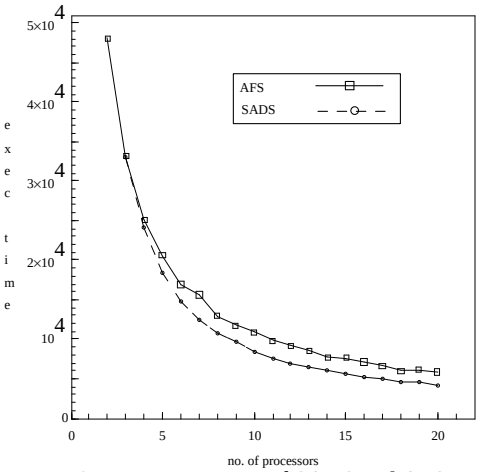

Figure 6.: Execution times of SADS and AFS in Low Affinity

SADS is absent due to its dedicated scheduling. The results of these experiments demonstrate the reasonable tradeoffs of employing sophisticated scheduling techniques even in small- to medium-sized machines.

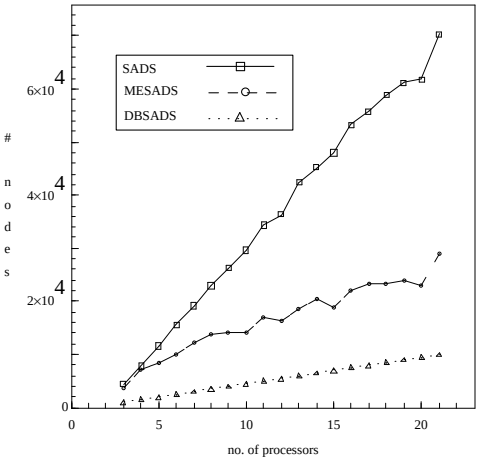

Figure 7.: Scheduling Efforts of SADS, MESADS, and DB-SADS in High Affinity

3.3. Effect of Heuristics on Scheduling Effort and Execution Times

Figures 7 and 9 show the results of comparing the scheduling effort of the three algorithms for different numbers of processors in high and low task-to-processor affinities, 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 predictive behavior of its scheduling costs. Such a smooth curve facilitates predicting scheduling costs during each phase and the prediction of the load (aver-

age 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.

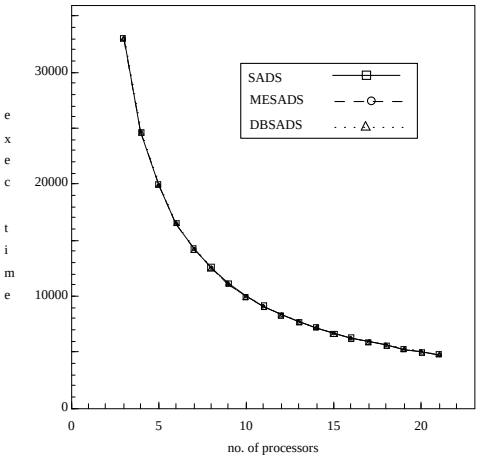

Figure 8.: Execution times of SADS, MESADS, and DBSADS in High Affinity

Figures 8 and 10 compare the total execution costs of SADS, MESADS, and DBSADS for different numbers of processors in high and low task-to-processor affinities, respectively. As shown in the figures, the execution costs of schedules resulting from different algorithms do not vary significantly. 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.

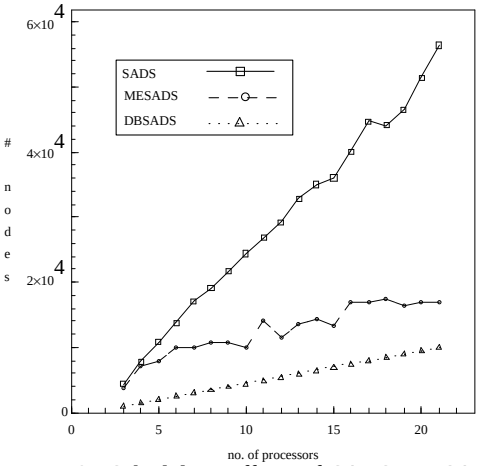

Figure 9.: Scheduling Efforts of SADS, MESADS, and DBSADS in Low Affinity

We observe that in most situations the execution time of the schedules found by MESADS is slightly lower than those found by DBSADS. In some situations the execution time of the schedules found by MESADS and DBSADS is better than those found by the SADS algorithm. This, we believe, is due to the fact that given an equal amount of time to schedule, MESADS and DBSADS focus their scheduling efforts on scheduling more tasks without loss of schedule quality. This leads to scheduling a larger number of tasks in each scheduling phase which produces fewer scheduling phases. With fewer scheduling phases (1 scheduling phase is the limit when CE_{min} is large), the higher the quality of the schedules will be, since information about the complete set of tasks is considered.

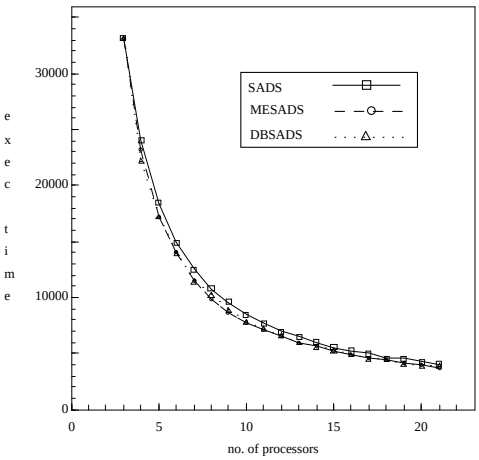

Figure 10.: Execution times of SADS, MESADS, and DBSADS in Low Affinity

4. Conclusions and Future Research

In this paper, we have introduced a class of centralized scheduling algorithms referred to as SADS which are capable of improving the performance of programs executed on shared-memory multiprocessor architectures. The algorithms of the SADS technique perform partial scheduling in repeated scheduling periods, using a novel on-line optimization technique running on a dedicated processor. The SADS 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. One such heuristic strategy, used in one of our algorithms referred to as MESADS, utilizes the information about an underestimate of the execution cost of the remaining tasks to be scheduled to produce an optimal solution while incurring much smaller scheduling costs than the basic SADS algorithm without any heuristics. Another heuristic strategy, used in another one of our algorithms referred to as DBSADS, gives higher priority to the partial schedules with more task-to-processor assignments thereby producing a combination of depth-first and branch-and-bound techniques in DBSADS.

The results of our experiments show significant improvements in program execution times over existing self-scheduling 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. 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. Our results show that 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. 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 algorithms, as well as the processing times and the memory delay times. We expect even greater performance improvement, compared to existing approaches, when using SADS, MESADS, and DBSADS algorithms with the scheduling costs taken into account.

References

1. H.C. Lin and C.S. Raghavendra, ‘‘A Dynamic Load-Balancing Policy With a Central Job Dispatcher (LBC),’’ *IEEE Transactions on Software Engineering*, vol. 18, no. 2, pp. 148-158, February 1992.
2. D.L. Eager, E.D. Lazowska, and J. Zahorjan, ‘‘Adaptive Load Sharing in Homogeneous Distributed Systems,’’ *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, pp. 662-675, May 1986.
3. M. Al-Mouhamed, ‘‘Analysis of Macro-Dataflow Dynamic Scheduling on Nonuniform Memory Access Architectures,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 8, pp. 875-887, August 1993.
4. P. Krueger and M. Livny, ‘‘Load Balancing, Load Sharing and Performance in Distributed Systems,’’ *Technical Report TR 700*, Computer Science Department, University of Wisconsin at Madison, 1987.
5. R. Vaswani and J. Zahorjan, ‘‘The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors,’’ *Operating Systems Review*, vol. 25, no. 5, pp. 26-40, 1991.
6. M. Willebeek-LeMair and A.P. Reeves, ‘‘Strategies for Dynamic Load Balancing on Highly Parallel Computers,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 4, pp. 979-993, September 1993.
7. M.S. Squillante and E.D. Lazowska, ‘‘Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 131-143, February 1993.
8. S. Subramaniam and D.L. Eager, ‘‘Affinity Scheduling of Unbalanced Workloads,’’ *Proceedings Supercomputing ’94*, pp. 214-26, 1994.
9. E.P. Markatos and T.J. LeBlanc, ‘‘Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 4, pp. 379-400, April 1994.
10. R. M. Karp and J. Pearl, ‘‘Searching for an Optimal Path in a Tree with Random Costs,’’ *Artificial Intelligence*, vol. 21, pp. 99-116, Elsevier, 1983.
11. N.J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.
12. C.C. Shen and W.H. Tsai, ‘‘A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion,’’ *Transactions on Computers*, IEEE, March, 1985.
13. C. J. Beckmann and C. D. Polychronopoulos, ‘‘The Effect of Barrier Synchronization and Scheduling Overhead on Parallel Loops,’’ *International Conference on Parallel Processing, Vol. II: Software*, pp. 200-204, 1989.