Concurrency: Practice & Experience, October 1995.

Dynamic Scheduling Techniques for Heterogeneous Computing Systems

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

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

Yacine Atif
(zohra@cs.ust.hk)
Dept. of Computer Science
University of Science & Technology
Clearwater Bay,Kowloon,Hong Kong

Abstract

There has been a recent increase of interest in heterogeneous computing systems, due partly to the fact that a single parallel architecture may not be adequate for exploiting all of a program's available parallelism. In some cases, heterogeneous systems have been shown to produce higher performance for lower cost than a single large machine. However, there has been only limited work on developing techniques and frameworks for partitioning and scheduling applications across the components of a heterogeneous system. In this paper, we propose a general model for describing and evaluating heterogeneous systems that considers the degree of uniformity in the processing elements and the communication channels as a measure of the heterogeneity in the system. We also propose a class of dynamic scheduling algorithms for a heterogeneous computing system interconnected with an arbitrary communication network. These algorithms execute a novel optimization technique to dynamically compute schedules based on the potentially non-uniform computation and communication costs on the processors of a heterogeneous system. A unique aspect of these algorithms is that they easily adapt to different task granularities, to dynamically varying processor and system loads, and to systems with varying degrees of heterogeneity. Our simulations are designed to facilitate the evaluation of different scheduling algorithms under varying degrees of heterogeneity. The results show improved performance for our algorithms compared to the performance resulting from existing scheduling techniques.

Key Words: heterogeneous systems, dynamic scheduling, scheduling costs, communication cost, load balancing.

Name	email	phone	fax
Babak Hamidzadeh	hamidzad@cs.ust.hk	852-358-7011	852-358-1477
David J. Lilja	lilja@ee.umn.edu	612-625-5007	612-625-4583
Yacine Atif	zohra@cs.ust.hk	852-358-7028	852-358-1477

1. Introduction

Independent computational tasks are said to be executing in parallel when they are run concurrently on different functional units, processors, or systems. The instantaneous number of these tasks from a single program that can be executed simultaneously is a measure of the *parallelism* available in that program at that instant in time. Since a single application program may have parallelism available at several different *granularities* during different portions of its execution, and since there is a wide range of potential parallelism in actual application programs[1,2,3,4,5], a single parallel architecture may not be adequate for exploiting all of the program's available parallelism[3,6,7]. This dilemma has led to greater interest in heterogeneous computer systems that can produce higher performance for lower cost than a single large machine[8,9,10].

Both hardware and software systems have been proposed to interconnect a wide variety of different computers into a single, unified system[11,12,13,14,15,16]. In addition, there are several programming environments and tools for developing large application programs to be executed across a network of heterogeneous systems, such as the Parallel Virtual Machine (PVM)[17], the Meta-Systems approach[18], and Linda[19]. While these programming tools are useful for developing explicit heterogeneous application programs, they provide little or no assistance for partitioning or scheduling applications across the components of a heterogeneous system.

Multiprocessor scheduling strategies determine which computational tasks will be executed on which processors at what times, but there has been little work developing dynamic scheduling strategies for heterogeneous systems, or developing frameworks for evaluating such strategies. Menasce and Almeida[20] proposed two distinct forms of heterogeneity in high-performance computing systems. *Function-level* heterogeneity exists in systems that combine general-purpose processors with special-purpose processors, such as vector units, floating-point co-processors, and input/output processors. With this type of heterogeneity, not all of the tasks can be executed on all of the function units. *Performance-level* heterogeneity, on the other hand, exists in systems that contain several interconnected general-purpose processors of varying speeds. In these systems, a task can execute on any of the processors, but it will execute faster on some than on others. While this model is useful for describing different heterogeneous architectures, it ignores the fact that heterogeneity is itself a continuum of performance and communication. We propose a new model for describing heterogeneous systems that incorporates both computation and communication.

Our new model considers the degree of uniformity in the processing elements and the communication channels as a measure of the heterogeneity in the system. Uniformity in computation refers to the degree of variance in processing costs (i.e. execution time) of a set of tasks on a set of processors. Similarly, uniformity in communication refers to the degree of variance in communication time due to the allocation of tasks to specific processors. In the remainder of this paper, by a "task" we mean an atomic unit of computation. In this interpretation, a task can be regarded as a single sequential process. For example, in a completely homogeneous system, the computation and communication costs are uniform across the system, independent of any specific task-to-processor assignments provided by the scheduling operation. This means that the cost of executing a single task, characterized by the sum of communication cost and computation cost incurred in the system for executing that task, is the same no matter on what processor that task is executed. An example of this type of system is a uniform-memory access (UMA) shared-memory multiprocessor in which each processor is identical, and the time required to access the global shared-memory is independent of the address being referenced. A non-uniform memory access (NUMA) shared-memory architecture, such as the Stanford DASH machine[21], has high uniformity in computation costs since all of the processors are of the same type. However, it has low uniformity in communication costs since the time required to reference a global address that has its home in the local cluster would be less than the time required to access an address that has its home in a remote cluster. Thus, in this architecture, the scheduler should assign tasks to processors considering the memory locations referenced by the task. A completely heterogeneous system is one in which the degree of uniformity in both computation and communication is low. An example of this type of system might be a NUMA architecture with multiple types of processors.

Large degrees of heterogeneity in a system adds significant additional complexity to the scheduling problem. In a completely homogeneous system, for example, the assignment of a parallel task to any of the processors will not affect the total execution time of that task by adding extraneous computation or communication time. In a heterogeneous system, however, tasks typically can execute only on a subset of the available processors. Furthermore, the execution time of a task on the different processors can vary dramatically depending on how well matched the task is to the architecture of the processor. For instance, a task may execute very well on a multiple instruction stream, multiple data stream (MIMD) architecture, but the same task may require a very long execution time on a single instruction stream, multiple data stream (SIMD) architecture. In addition, there can be significant communication delays incurred when the results produced by a task are needed by another task executing on a different heterogeneous component of the system. These factors must be considered when scheduling parallel tasks on a heterogeneous system in addition to the scheduling overhead and load balancing factors that are part of the homogeneous scheduling problem. To minimize total execution time in a completely heterogeneous system, it is important for the scheduler to consider both the type of processor to which a task can be assigned, and the cost of the communication that results from assigning a task to a particular processor. In some cases, for instance, a lower overall execution time may be produced by assigning a task to a slower processor type, but one which will produce a lower communication cost.

This new model of heterogeneous systems using the degree of uniformity in computation and communication subsumes Menasce and Almeida's function- and performance-level model. Performance-level heterogeneity is described simply as a measure of the time required to execute a particular task on a particular processor. Function-level heterogeneity is incorporated by assigning an infinite computation cost value to a task if it were to be executed on a function unit of the wrong type. For instance, an integer-only task would take infinite (or at least a very long) time to execute on a floating-point-only function unit. As shown in the remainder of this paper, by incorporating the heterogeneous nature of both computation and communication, this new model is useful for guiding the development of scheduling strategies for heterogeneous systems.

Multiprocessor scheduling strategies can be categorized as either *static* or *dynamic*[22]. Static scheduling strategies preassign tasks to processors using compile-time estimates of the execution times of the tasks. Given perfect information, static scheduling can theoretically precompute an ideal schedule that will produce the shortest possible execution time. In most applications, however, such precise information is unavailable at compile-time. Static scheduling with imprecise estimates can lead to large computational load imbalances at run-time, which produce significantly longer execution times. Dynamic scheduling attempts to compensate for this lack of compile-time information by delaying the assignment of parallel tasks to processors until run-time. This postponement allows the program execution to adapt to the dynamically varying processor loads encountered at run-time. A major cost factor that a dynamic scheduling strategy must take into account is the run-time overhead associated with performing task-assignment on-line.

Our focus in this paper is on dynamic scheduling of tasks some or all of whose characteristics may not be known a priori. Examples of such task models are event-driven applications such as real-time simulation where the occurrences of tasks and their invocation patterns become known only during problem solving. The tasks in these applications are characterized by arbitrary arrival times. Another class of applications which can potentially benefit from our proposed dynamic scheduling techniques are numerical applications where the input to the application consists of largely varying tasks. An example of such a situation is matrix manipulation techniques where the input matrices vary largely in their sparseness[23]. Let us assume that a task in such applications consists of a binary operation (e.g. multiplication) on two vectors from two input matrices. In this situation, while the number and arrival times of tasks may be known a priori, their execution costs cannot be estimated accurately, off-line. More accurate estimates of the task execution costs can be obtained on-line when the number of non-zero elements in each vector becomes known. The ability to obtain, on-line, more accurate estimates of task execution times in these applications, lends them well to dynamic scheduling techniques which can, in turn, provide potentially significant performance improvements compared to static scheduling. In this paper, we propose a class of dynamic scheduling algorithms for a heterogeneous computing system interconnected with an arbitrary communication network. These algorithms, referred to as Self-Adjusting Scheduling for Heterogeneous systems (SASH), execute a novel optimization technique on-line to dynamically compute partial schedules based on the potentially non-uniform task computation costs on the processors of a heterogeneous system, the computational loads of the processors, and the potentially non-uniform communication costs. To reduce the performance impact of using these scheduling algorithms, we have investigated the use of a dedicated scheduling processor to overlap the execution of the parallel tasks with the computation of partial schedules. In many existing dynamic scheduling algorithms where processors request tasks from a globally-shared mechanism, the scheduling overhead is increased due to synchronization and bottleneck effects. SASH assigns tasks to working processors automatically to further reduce the overhead associated with such synchronization and bottleneck effects. Our results show that the performance gain of the better partial schedules, compared to those produced by simpler scheduling heuristics, can outweigh the loss in performance caused by removing a processor from executing the application program's parallel tasks, even in systems with a small number of processors.

A unique aspect of the SASH algorithms is that they can be applied to a variety of task models (e.g. models in which tasks are generated dynamically and have arbitrary arrival times) and can easily adapt to different task granularities, to dynamically varying processor and system loads, and to systems with varying degrees of both computation and communication heterogeneity. We use probabilistic simulations to evaluate these aspects of SASH based on the model of heterogeneity proposed above. These simulations are aimed at evaluating the performance of different scheduling algorithms under varying degrees of heterogeneity.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the SASH algorithm, while an evaluation of the performance of this algorithm is presented in Section 4. Finally, our results and conclusions are summarized in Section 5.

2. Background and Related Work

Several problems of optimally scheduling independent computational tasks to be executed on the processors of a multiprocessor system have been shown to be NP-complete[24]. The complexity of these problems is due to the large number of interrelated factors that directly and indirectly contribute to the execution time of the individual tasks. Consequently, a large number of heuristics have been developed to generate good, but perhaps suboptimal, schedules. The following subsections review and summarize several of these heuristic scheduling techniques.

2.1. Scheduling in Homogeneous Systems

Many heuristic strategies have been proposed for scheduling tasks for execution on the processors of a homogeneous multiprocessor[22,25] using task models that address a wide range of task granularities. In some models, a task refers to a single iteration of a parallel loop, while in other models, a task may refer to a module in a single program or to separate programs run by one or more users. The simplest scheduling heuristic statically assigns tasks to processors in a round-robin fashion such that processor 0 executes tasks 1, $p+1, 2p+1, \cdots$, processor 1 executes tasks 2, $p+2, 2p+2, \cdots$, and so on, where p is the number of processors in the system. Static scheduling generates essentially no run-time overhead, but, since the tasks may have non-uniform execution times, static scheduling can severely unbalance the computational load on the processors leading to excessively long execution times.

To more evenly distribute the load among the processors, dynamic self-scheduling forces processors to allocate independent tasks to themselves when they become idle. One of the primary differences among the various self-scheduling heuristics is the number of tasks a processor assigns to itself as a *chunk* of work each time it accesses the list of tasks ready to be executed. At one extreme, single-task scheduling allocates one task per scheduling step, while, at the other extreme, single-chunk self-scheduling allocates one large chunk of $\lceil N/p \rceil$ tasks to each of the p processors, where N is the total number of independent tasks. Since the time required to allocate a chunk of tasks to a processor is independent of the chunk size, the single-task strategy incurs significantly higher run-time overhead than the single-chunk strategy, and this overhead adds directly to the total execution time of the program. However, the single-chunk strategy can lead to load imbalances comparable to the simple static scheduling strategy.

Several heuristic loop scheduling strategies have been proposed to trade-off scheduling overhead and load balancing[25]. The common idea behind these strategies is to allocate large chunks of a parallel loop's independent iterations to the first several processors requesting work, and then to allocate successively smaller chunks to subsequent processors. Initially allocating large chunks is intended to reduce the total scheduling overhead, while the smaller chunks allocated towards the end of the loop's execution can be used to fill-in processor idle time to thereby balance the load. For example, Guided Self-Scheduling[26] allocates $\lceil N/p \rceil$ iterations to the first processor, $\lceil (N - \lceil N/p \rceil)/p \rceil$ iterations to the next processor, and so on. As a result, at each scheduling step, the requesting processor is allocated approximately $1/p$ of the remaining iterations. Several other variations of this declining chunk size strategy have proposed[27,28], including one that combines an initial static allocation with dynamic load rebalancing to improve the memory referencing locality of the processors[29]. While these loop scheduling heuristics have been developed for shared-memory multiprocessors, there have been related heuristics proposed for distributed memory multicomputers[30,31,32].

In addition to these heuristics for scheduling the iterations from parallel loops, there have been many algorithms developed for scheduling general parallel tasks[22,33,34]. The Distributed Self-Scheduling (DSS) technique[30,32], for instance, uses a combination of static and dynamic scheduling. During the initial static scheduling phase, p chunks of work are assigned to the p processors in the system. The first processor to finish executing its tasks from the static scheduling phase then designates itself as the centralized scheduling processor. This processor stores the shared information about which tasks are yet to be executed, and which processors have the data necessary to compute each of the tasks. It then dynamically distributes the tasks to the processors as they become idle to balance the total work among all of the processors. Several different data partitioning schemes have been evaluated for DSS, along with a hierarchical scheduling strategy to eliminate the potential bottleneck caused by the single scheduling processor.

The Self-Adjusting Dynamic Scheduling (SADS) algorithm[35] develops a detailed cost model to trade-off the interrelated performance effects of processor load balance, memory referencing locality (i.e. interprocessor communication delays), and scheduling and synchronization overhead. During a single scheduling period, a designated scheduling processor assigns tasks to the processors' local work queues using a novel algorithm, related to the branch-and-bound family of algorithms, to minimize the cost function. The time allocated to a single scheduling phase is determined by a lower-bound estimate of the load on the least-loaded processor. The scheduling processor continues to schedule tasks until the least-loaded processor finishes all of the assigned tasks in its queue. The scheduling processor then assigns tasks to all of the processors using the partial schedule it computed during the scheduling phase. Thus, the available scheduling time is dynamically adjusted on-line as the processor loads change, and the scheduling operation is completely overlapped with the execution of the parallel tasks by the remaining processors.

2.2. Scheduling in Heterogeneous Systems

The Optimal Selection Theory[9] has been developed to determine the optimal configuration of heterogeneous supercomputers that maximizes the performance of a given application program subject to some fixed constraint, such as cost. This approach has been extended to include the effects of executing code segments on non-optimal machines and the effects of non-uniform decomposition of code segments[36]. It has been further extended to include the effects of exploiting heterogeneous parallelism within a single task[37]. Several other scheduling strategies have been proposed that typically use heuristics to minimize some cost function based on static estimates of task execution times[38,39,40,41].

While these static scheduling strategies are quite interesting from a theoretical point-of-view, they typically have not been used in actual applications. Due to the practical difficulties of the heterogeneous scheduling problem, large-scale heterogeneous applications have tended to use a simple static partitioning or pipeline scheduling. For example, researchers at Carnegie-Mellon University implemented an iterative relaxation algorithm across a networked Cray Y-MP and a Connection Machine CM-2 by using the Cray to solve an initial small set of equations, and then transferring the larger set of equations produced by the iterative relaxation to the CM-2 after reaching a critical threshold[42,43]. A DNA sequencing application reversed the direction of the data communication by performing an initial, parallel calculation on the CM-2 and then transferring the intermediate results to the Y-MP for the less parallel recalculation phase[44]. While neither of these applications used both supercomputers concurrently, researchers at the University of Minnesota used a pipeline consisting of a Connection Machine CM-5, a Cray-2, and a Connection Machine CM-200 interconnected with a HiPPI network to perform a volumetric analysis of thermal convective mixing. The final image was displayed in real-time on a Silicon Graphics workstation[45].

The Automatic Heterogeneous Supercomputing (AHS) system[46] uses a quasi-dynamic scheduling strategy for minimizing the response time observed by a user when submitting an application program for execution. This system maintains an information file for each program that contains an estimate of the execution time of the program on each of the available machines. When a program is invoked by a user, AHS examines the load on each of the networked machines and executes the program on the machine that it estimates will produce the fastest turn-around time. Once the program is initiated on a specific processor, however, no further scheduling is performed.

These applications have demonstrated that simple static scheduling can be used to exploit heterogeneous computing systems, but perhaps not to their full potential. In the next section, we develop a dynamic heterogeneous scheduling strategy based on the SADS algorithm[35] described above. This strategy dynamically schedules the individual tasks on to the heterogeneous components of the system using an on-line optimization technique that minimizes the execution cost function in a series of repeated scheduling periods. The performance of this new strategy is compared to a heterogeneous extension of the original DSS algorithm[30,32].

3. Self-Adjusting Scheduling for Heterogeneous Systems (SASH)

The Self-Adjusting Scheduling for Heterogeneous Systems (SASH) algorithm is based on the SADS class of centralized dynamic scheduling techniques[35,47]. It utilizes a maximally overlapped scheduling and execution paradigm to schedule a set of independent tasks on to a set of heterogeneous processors. Overlapped scheduling and execution in SASH is accomplished by dedicating a processor to execute the scheduling algorithm. SASH performs repeated scheduling phases in which it generates partial schedules. At the end of each scheduling phase, the scheduling processor places the tasks scheduled in that phase on to the working processors' local queues.

The scheduling processor then schedules the next subset of tasks while the previously scheduled tasks are executed by the working processors.

A lower-bound estimate of the load on the working processors is used to determine the duration of each scheduling phase. During a single scheduling phase, the algorithm schedules tasks until the least-loaded working processor has completed executing all of the tasks on its local queue. At that point, the scheduling algorithm assigns tasks to all processors based on the partial schedule just computed. Note that the last scheduling phase will be completed before all of the computational tasks have finished executing. Thus, in the last scheduling phase, the scheduling processor can assign some tasks to itself. For simplicity, however, the results presented in this paper assume that the scheduling processor remains idle after assigning all of the tasks to the working processors.

The SASH algorithm, which is a variation of the family of branch-and-bound algorithms[48], searches through a space of all possible partial and complete schedules. In our scheduling problem, we use a tree to represent this space of schedules. Each node in this tree represents a partial schedule consisting of a set of tasks assigned to a corresponding set of processors. An edge from a node represents an extension of the partial schedule obtained by adding one more task-to-processor assignment to the partial schedule represented by that node. It is important to note that SASH differs from the original branch-and-bound algorithm and that it is significantly faster to execute. This is mainly due to the SASH'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 SASH, 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.

Each scheduling phase in the SASH algorithm consists of one or more SASH iterations. Within an iteration, the node with the lowest cost is expanded by extending the partial schedule with an additional edge; that is, by adding a task-to-processor assignment. These node expansions continue until all of the tasks have been scheduled, or until the time allocated to this scheduling phase, i, expires. At this point, the node with the least cost is used to allocate tasks to processors. Scheduling phase $i+1$ then proceeds by beginning a new search through the remaining tasks. Pseudo code for the SASH algorithm is shown in Figure 1.

The cost function used to estimate the total execution time produced by a given partial schedule is a critical component of the SASH algorithm. This function begins by estimating the cost of executing task t_l on processor p_m to be $(cp_{t_l p_m} + cc_{t_l p_m})$, where $cp_{t_l p_m}$ is the processor time required to execute task t_l on processor p_m, and $cc_{t_l p_m}$ is the additional communication delay required to transfer any data values needed by this task to the processor. The total

```
PROCEDURE SASH(task-set);

VAR
queue,succ_list : queue-of-nodes;
x,current_node: node;

WHILE NOT((solved(head(queue)) OR (scheduling_time ≤ 0))) 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);
    SASH(remaining_task_set);
  END
  ELSE announce_failure;
```

Figure 1: Pseudo code for the SASH algorithm.

execution time at scheduling phase i for processor p_m when task t_l is added to the current node then is

$$ce_m(i) = CE_m(i-1) + \sum_{all\ (t_l\ p_m)} (cc_{t_l p_m} + cp_{t_l p_m})$$

where $CE_m(i-1)$ is the total execution cost of processor p_m at the end of scheduling phase $(i-1)$. The notation $ce_m(j)$ signifies the cost of processor p_m's load *during* scheduling phase j, while the notation $CE_m(j)$ signifies the actual load on processor p_m *after* the final schedule of phase j has been determined.

Since the total execution time for the partial schedule is determined by the processor that finishes executing its tasks last, the execution time for the partial schedule represented by node v_k, which we denote C_k, is the maximum of the $ce_m(i)$ values for all of the p processors. Thus, the cost of executing the schedule defined by node v_k at scheduling step i is

$$C_k(i) = \max(ce_m(i) \mid 1 \le m \le p).$$

The time allocated for scheduling during phase i, $CS(i)$, is a fraction α of the total execution cost of the least-loaded processor at the end of scheduling phase $i-1$. That is, $CS(i) = \alpha CE_{\min}(i-1)$ where $CE_{\min}(j) = \min(ce_m(j) \mid 1 \le m \le p)$ is the minimum over all of the processors. Thus, the time allocated to scheduling in each phase is dynamically adjusted to completely overlap scheduling with the execution of the parallel tasks. Note that $CS(0)$ must be set to some appropriate positive value when the program is first loaded, or some other scheduling strategy, such as simple prescheduling, must be used to initially allocate a few tasks to the processors. Partial schedules with a large number of tasks can be computed when $CS(i)$ is large, while small values will limit the scheduling time and reduce SASH to a simple greedy algorithm. In this paper, we assume that $CS(i)$ is at least as large as the cost of expanding a single node in the search space so that the smallest amount of scheduling performed in any scheduling phase is a single iteration of the standard branch-and-bound.

One of the unique features of SASH's cost model is that it accounts for non-uniformity in the tasks' communication and processing costs when scheduled on specific processors. This model simultaneously addresses the tradeoffs between the two interacting factors of processor load balancing and communication cost management. Another important feature of SASH's cost model is its adjustability to the degree of heterogeneity in the system. For example, in a completely homogeneous system, SASH's cost model does not consider communication cost or processing cost on different processors as a major factor in evaluating the quality of a schedule. Instead, SASH's cost model effectively concentrates on balancing the processors' loads.

In a system with a high degree of non-uniformity in communication costs, on the other hand, SASH's cost model evaluates the quality of a schedule based on how well the load is balanced, while minimizing the total communication cost. In a distributed system whose processors are connected with a wide-area network rather than a local interconnection network, for instance, the cost model automatically emphasizes more heavily the need for scheduling tasks on those processors which have greater affinity with those tasks due to the much larger communication costs compared to a parallel computer with a local interconnection network. In a highly heterogeneous system, the cost model effectively accounts for the fact that load balancing does not merely depend on the number of tasks assigned to a processor, but that it also depends on how long it will take a processor to execute a specific task. In the case of a system with function-level heterogeneity, for instance, this consideration is crucial for achieving high system performance.

Some analytical properties of SASH can be obtained by generalizing the results obtained for this algorithm's predecessor designed for homogeneous systems[35]. We summarize some of these properties here and refer the interested reader to [47] for a detailed discussion and proof of these properties. An important property to consider is the bound on the optimality of SASH. Here we are interested in knowing how close to optimal SASH's schedules can be despite the major reduction in complexity of the algorithm compared to the traditional branch-and-bound algorithm. It can be shown that SASH is optimal within a single scheduling phase. Using this result, we can also show that, in general, SASH produces schedules whose total execution times are larger than the optimal by at most the time of the longest-running task. In the following section, we examine the performance of SASH in different systems with different degrees of heterogeneity.

4. Experimental Evaluation

The main objective of our experiments was to evaluate the performance of SASH in computing environments with varying degrees of heterogeneity. We also were interested in examining the gain in performance obtained by dedicating a processor to scheduling. Our experiments were designed to investigate the system configurations (e.g. the number of processors in the system) in which the performance gain justifies the loss of performance due to centralized scheduling. To evaluate these aspects of SASH, we have chosen a performance comparison methodology using probabilistic simulations. Controlling parameters in the simulation experiments facilitates evaluation of the algorithms' performance in environments with varying degrees of heterogeneity relative to an existing technique.

In our experiments, we compared the performance of SASH with that of a heterogeneous extension of Distributed Self-Scheduling (DSS). [30,32] In the original description of DSS, a working processor requests tasks from a server processor, which stores in its local memory global information about the task set. DSS balances the workload well if small chunks of tasks are assigned at each work request. However, the high frequency of mutually exclusive access to shared variables (such as the loop index in the case of parallel loops) may seriously degrade performance. In our experiments, we did not penalize DSS for such overhead cost. In other words in DSS, we assumed a chunk size of one without penalizing the algorithm with the additional scheduling overhead due to such a small chunk size. In the case of SASH, on the other hand, we incorporated a major part of the scheduling cost by taking into account the effect on performance of dedicating a processor to scheduling. In the following, we describe our metrics of performance, experimental parameters, the choice of problem instances, and the modeling of heterogeneity and interprocessor communication costs.

4.1. Experimental Methodology

The metric of performance in our experiments was chosen to be the total execution time of the complete schedules; that is, the total time required to complete the execution of the task set submitted to the scheduler. Each problem instance consists of a set of tasks, a set of processors, a task-processing-time matrix, and a task-communication-time matrix. The parameters of our experiments include the number of tasks, the mean task processing times, the standard deviation of task processing times, the mean of communication, the standard deviation of task communication, and the number of processors. For each of our experiments we generated 500 independent tasks. The number of processors ranged from 2 to 20.

The mean task processing time (μ_{cp}) was chosen to be 500 CPU clock cycles, and the values of the task-processing standard deviations (σ_{cp}) were 10, 50 and 100 cycles. The mean values of the communication times (μ_{cc}) were 500, 5000 and 50000 clock cycles. The standard deviation of communication time values (σ_{cc}) corresponding to a communication mean of 500 are 10, 50 and 100; those corresponding to a mean of 5000 are 100, 500 and 1000, and finally those associated with a mean of 50000 are 1000, 5000 and 10000. The processing costs of a task t_l on the processors of the system are uniformly distributed around the μ_{cp} values in the interval $[(\mu_{cp} - 3\sigma_{cp}), (\mu_{cp} + 3\sigma_{cp})]$. Similarly, the communication costs of tasks on different processors are uniformly distributed around the μ_{cc} values in the corresponding interval $[(\mu_{cc} - 3\sigma_{cc}), (\mu_{cc} + 3\sigma_{cc})]$. This distribution of communication cost of tasks on different processors can be interpreted as an indication of the degree of affinity or data locality of a task on a given processor.

The range of parameter values facilitated the study of the candidate algorithms' performance in environments with varying degrees of heterogeneity. Small σ_{cp} values represent a high degree of uniformity in processing costs, whereas large σ_{cp} values represent an environment with a low degree of uniformity in processing costs. Similarly, small σ_{cc} values represent a high degree of uniformity in processing costs, whereas large σ_{cc} values represent an environment with a low degree of uniformity in communication. Thus, small values of σ_{cp} and σ_{cc} represent highly homogeneous systems, while large values of σ_{cp} and σ_{cc} represent highly heterogeneous systems according to the model of heterogeneity proposed in Section 1.

In the experiments, we selected sets of parameter values that modeled degrees of heterogeneity in a set of existing architecture models. For example, uniform processing and uniform communication may represent a UMA shared-memory architecture such as the Sequent Symmetry S-81. Uniform processing costs with non-uniform communication costs may represent a NUMA distributed-memory architecture. Depending on how high the value of σ_{cc} may be, we can categorize the architecture as a distributed system or a parallel computer. Typically, the σ_{cc} value in the former is larger than that of the latter. Examples of NUMA architectures modeled by uniform processing costs and non-uniform communication costs are the BBN TC-2000, CRAY T3D, or Intel Paragon XP/S. Non-uniform processing and communication costs in our model represent a heterogeneous multicomputer, such as an arbitrarily-connected network of heterogeneous workstations and supercomputers.

The parameter values selected for the experiments also allowed us to evaluate the effect of different communication-to-computation ratios on our candidate algorithms' performance. This ratio is defined to be $R = \frac{\mu_{cc}}{\mu_{cp}}$ and can be interpreted in a variety of ways, one of which is the network size in terms of communication delay (i.e. the larger the R ratio, the more distributed the communication network). Some existing scheduling algorithms have been shown to fail in asymptotic cases when communication costs are high relative to CPU costs (i.e. when R is large) [49]. We note that, in our experiments, the scheduling cost of DSS is assumed to be zero while the scheduling cost of SASH is modeled by dedicating one processor to perform the scheduling, leaving $p - 1$ processors to execute the parallel tasks where p is the total number of processors in the system.

4.2. Simulation Results

In examining the data from our experiments, we are interested in observing the minimum number of processors required for a positive gain in performance when executing SASH on a dedicated processor. The gain in performance was measured with respect to the execution time of schedules produced by a version of DSS which uses a chunk size of one, for which scheduling overhead is ignored.

The results of the experiments are presented in Figures 2 through 13. The twelve figures can be viewed as a 3×4 matrix (3 figures in each column and 4 figures in each row). In this respect, Figures 2, 5, 8 and 11 would constitute the first row of the matrix and Figures 2, 3 and 4 would constitute the first column of the matrix. The degree of heterogeneity increases along the rows of the matrix from left to right and the ratio R increases along the columns from top to bottom.

Figures 2, 3, and 4 (first column) depict the total execution times of SASH and DSS in a highly homogeneous system with uniform computation costs and uniform communication costs. In these figures, even though the mean communication cost (i.e. μ_{cc}) increases, the degree of non-uniformity in communication costs (i.e. σ_{cc}) remains relatively low with respect to the magnitude of μ_{cc}. As is shown in the figures, DSS outperforms SASH due to the well-balanced loads that are produced by its small chunk size, and due to the fact that SASH is penalized for dedicating an entire processor to scheduling. We observe that as the number of processors increases the performance of the two algorithms converges.

Figures 5, 6, and 7 (second column) depict the total execution times of SASH and DSS in a system with uniform computation costs and increased non-uniformity in communication costs. In these figures, the σ_{cc} values are somewhat higher with respect to the magnitude of μ_{cc}. As is shown in the figures, SASH outperforms DSS at 13 processors when $\mu_{cc} = 500$, at 10 processors when $\mu_{cc} = 5000$, and at 8 processors when $\mu_{cc} = 50000$.

Figures 8, 9 and 10 (third column) depict the total execution times of SASH and DSS in a system with uniform computation costs and with high non-uniformity in communication costs. In these figures, the values of σ_{cc} are the highest with respect to the magnitude of μ_{cc}, among the values selected for our experiments. As is shown in the figures, SASH outperforms DSS at 8 processors when $\mu_{cc} = 500$, at 4 processors when $\mu_{cc} = 5000$, and at 4 processors when $\mu_{cc} = 50000$.

Figures 11, 12 and 13 (fourth column) depict the total execution times of SASH and DSS in a highly heterogeneous system with non-uniform computation costs and with high non-uniformity in communication costs. As is shown in the figures, SASH outperforms DSS at 5 processors when $\mu_{cc} = 500$, at 4 processors when $\mu_{cc} = 5000$, and at 4 processors when $\mu_{cc} = 50000$.

To summarize these results, we note that as the degree of heterogeneity in the system increases, SASH running on a dedicated processor outperforms DSS even in systems with only a few processors. Furthermore, as the communication-to-computation ratio (i.e. R) increases, SASH can outperform DSS with fewer processors in the system.

5. Conclusions and Future Research

In this paper, we have proposed a new model for characterizing heterogeneity in computer systems. The model characterizes heterogeneity in terms of the degree of non-uniformity in computation and communication. We have investigated the performance of a class of centralized scheduling algorithms referred to as SASH in computing environments with different degrees of heterogeneity. The degrees of heterogeneity in our experiments were chosen to represent a set of existing architecture models.

The SASH algorithms perform partial scheduling in repeated scheduling periods, using a novel on-line optimization technique running on a dedicated processor. As mentioned earlier, these algorithms have much lower computational and space complexities than other traditional optimization techniques such as the branch-and-bound algorithm. The SASH algorithms self-adjust the amount of time allocated to each scheduling period to minimize the processor idle times. An important aspect of these algorithms is the unified cost model they employ to evaluate partial and complete schedules. This cost model automatically adjusts to the degree of heterogeneity in the system. In a completely homogeneous system, SASH's cost model effectively concentrates only on balancing the processors' loads since communication costs and processing costs are highly uniform in such systems. In a highly heterogeneous system, the cost model accounts for the fact that good performance does not merely depend on an equally distributed processor load. Instead, SASH's cost model accounts for the length of time required for a processor to execute a specific task in addition to the communication time required to perform the task if it were assigned to that processor.

The results of our experiments show significant improvements in program execution times over the DSS approach in highly heterogeneous systems, even when the synchronization and scheduling costs of DSS 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.

As part of our future work, we plan to address the issue of selecting the appropriate scheduling processor from a set of heterogeneous processors. We also plan to investigate the effectiveness of the SASH technique in a hierarchical heterogeneous system in which the processors are divided into clusters with high degrees of inter-cluster heterogeneity and low degrees of intra-cluster heterogeneity. Implementation of the SASH techniques on heterogeneous architectures (e.g. hierarchical clusters of different workstations such as SUN Sparc stations and SGI workstations) is currently under way.

6. 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 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

1. M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, ‘‘Single Instruction Stream Parallelism is Greater than Two,’’ *International Symposium on Computer Architecture*, pp. 276-286, 1991.
2. M. Kumar, ‘‘Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications,’’ *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1088-1098, September 1988.
3. D. J. Lilja, ‘‘A Multiprocessor Architecture Combining Fine-Grained and Coarse-Grained Parallelism Strategies,’’ *Parallel Computing*, vol. 20, no. 5, pp. 729-751, May 1994.
4. D. J. Lilja (ed.), *Architectural Alternatives for Exploiting Parallelism*, IEEE Computer Society Press, Los Alamitos, CA, ISBN 0-8186-2642-9, 1992.
5. D. W. Wall, ‘‘Limits of Instruction-Level Parallelism,’’ *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176-188, 1991.
6. A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, ‘‘Heterogeneous Computing: Challenges and Opportunities,’’ *Computer*, vol. 26, no. 6, pp. 18-27, June 1993.
7. C. C. Weems, ‘‘Image Understanding: A Driving Application for Research in Heterogeneous Parallel Processing,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 119-126, 1993.
8. J. Andrews and C. Polychronopoulos, ‘‘An Analytical Approach to Performance/Cost Modeling of Parallel Computers,’’ *Journal of Parallel and Distributed Computing*, no. 12, pp. 343-356, 1991.
9. R. F. Freund, ‘‘Optimal Selection Theory for Superconcurrency,’’ *Supercomputing ’89*, pp. 699-703, 1989.
10. D. Menasce and V. Almeida, ‘‘Cost-Performance Analysis of Heterogeneity in Supercomputer Architectures,’’ *Proc. Supercomputing ’90*, pp. 169-177, 1990.
11. L. Wittie and C. Maples, ‘‘MERLIN: Massively Parallel Heterogeneous Computing,’’ *International Conference on Parallel Processing, Vol I: Architecture*, pp. 142-150, 1989.
12. E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste, ‘‘The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers,’’ *Third Intl. Conf. on*

Architectural Support for Programming Languages and Operating Systems, pp. 205-216, 1989.

13. R. Bisiani and A. Forin, ‘‘Multilanguage Parallel Programming of Heterogeneous Machines,’’ *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 930-945, August 1988.

14. J. Mahdavi, G. L. Huntoon, and M. B. Mathis, ‘‘Deployment of a HiPPI-based Distributed Supercomputing Environment at the Pittsburgh Supercomputing Center,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 93-96, 1992.

15. R. J. Vetter, D. H. C. Du, and A. E. Klietz, ‘‘Network Supercomputing: Experiments with a Cray-2 to CM-2 HiPPI Connection,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 87-92, 1992.

16. L. Smarr and C. E. Catlett, ‘‘Metacomputing,’’ *Communications of the ACM*, vol. 35, no. 6, pp. 45-52, June 1992.

17. V. S. Sunderam, ‘‘PVM: A Framework for Parallel Distributed Computing,’’ *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315-339, December 1990.

18. A. S. Grimshaw, ‘‘Meta-Systems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 54-59, 1992.

19. N. Carriero, D. Gelernter, and T. G. Mattson, ‘‘Linda in Heterogeneous Computing Environments,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 43-46, 1992.

20. D. Menasce and V. Almeida, ‘‘Heterogeneous Supercomputing: Is It Cost-Effective?,’’ *Supercomputing Review*, vol. 4, no. 8, pp. 39-41, August 1991.

21. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, ‘‘The Stanford DASH Multiprocessor,’’ *Computer*, vol. 25, no. 3, pp. 63-79, March 1992.

22. T. L. Casavant and J. G. Kuhl, ‘‘A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems,’’ *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141-154, February 1988.

23. V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, pp. 407-489, Benjamin/Cummings Publishing Company, Inc., 1994.

24. M. R. Garey and D. S. Johnson, *Computers and Intractability,* W. H. Freeman and Company, New York, 1979.

25. D. J. Lilja, ‘‘Exploiting the Parallelism Available in Loops,’’ *Computer*, vol. 27, no. 2, pp. 13-26, February 1994.

26. C. Polychronopoulos and D. Kuck, ‘‘Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,’’ *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1425-1439, December 1987.

27. S. F. Hummel, E. Schonberg, and L. E. Flynn, ‘‘Factoring: A Method for Scheduling Parallel Loops,’’ *Communications of the ACM*, vol. 35, no. 8, pp. 90-101, August 1992.

28. T. H. Tzen and L. M. Ni, ‘‘Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87-98, January 1993.

29. 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.

30. J. Liu and V. A. Saletore, ‘‘Self-Scheduling on Distributed-Memory Machines,’’ *Supercomputing ’93*, pp. 814-823, 1993.

31. D. C. Rudolph and C. Polychronopoulos, ‘‘An Efficient Message-Passing Scheduler Based on Guided Self Scheduling,’’ *ACM International Conference on Supercomputing*, pp. 50-60, 1989.

32. V. A. Saletore, J. Liu, and B. Y. Lam, ‘‘Scheduling Non-uniform Parallel Loops on Distributed Memory Machines,’’ *Hawaii International Conference on System Sciences, Vol. II*, pp. 516-525, January 1993.

33. I. Ahmad and A. Ghafoor, ‘‘Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems,’’ *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 987-1004, October 1991.

34. S. Chowdhury, ‘‘The Greedy Load Sharing Algorithm,’’ *Journal of Parallel and Distributed Computing*, no. 9, pp. 93-99, May 1990.

35. B. Hamidzadeh and D. J. Lilja, ‘‘Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing, Volume II: Software,’’ *International Conference on Parallel Processing*, pp. 39-46, 1994.

36. M.-C. Wang, S.-D. Kim, M. A. Nichols, R. F. Freund, H. J. Siegel, and W. G. Nation, ‘‘Augmenting the Optimal Selection Theory for Superconcurrency,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 13-21, 1992.

37. S. Chen, M. M. Eshaghian, A. Khokhar, and M. E. Shaaban, ‘‘A Selection Theory and Methodology for Heterogeneous Supercomputing,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 15-22, 1993.

38. M. A. Iqbal, ‘‘Partitioning Problems in Heterogeneous Computer Systems,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 23-28, 1993.

39. L. Tao, B. Narahari, and Y. C. Zhao, ‘‘Heuristics for Mapping Parallel Computation to Heterogeneous Parallel Architectures,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 36-41, 1993.

40. E. Haddad, ‘‘Load Distribution Optimization in Heterogeneous Multiple Processor Systems,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 42-47, 1993.

41. D. J. Lilja, ‘‘Experiments with a Task Partitioning Model for Heterogeneous Computing,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 29-35, April 1993.

42. D. DiMaria, S. Feldman, and J. Kellum, ‘‘Program notes: A Union of Superpowers,’’ *IEEE Spectrum*, p. 18, June 1991.

43. M. Schneider, ‘‘Tying the Knot Between Serial and Massively Parallel Computing: Pittsburgh’s Not-So-Odd Couple,’’ *Supercomputing Review*, vol. 4, no. 8, pp. 36-38, August 1991.

44. H. Nicholas, G. Giras, V. Hartonas-Garmhausen, M. Kopko, C. Maher, and A. Ropelewski, ‘‘Distributing the Comparison of DNA and Protein Sequences Across Heterogeneous Supercomputers,’’ *Proceedings of Supercomputing ’91*, pp. 139-146, 1991.

45. A. E. Klietz, A. V. Malevsky, and K. C. Purcell, ‘‘A Case Study in Metacomputing: Distributed Simulations of Mixing in Turbulent Convection,’’ *Workshop on Heterogeneous Processing, International Parallel Processing Symposium*, pp. 101-106, 1993.

46. H. G. Dietz, W. E. Cohen, and B. K. Grant, ‘‘Would You Run it Here... or There? (AHS: Automatic Heterogeneous Supercomputing),’’ *International Conference on Parallel Processing, Volume II: Software*, pp. 217-221, 1993.

47. B. Hamidzadeh and D. J. Lilja, ‘‘Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing,’’ *Minnesota Supercomputer Institute Technical Report*, February 1994.

48. N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

49. A. Q. Khan, C.L. McCreary, and M.S. Jones, ‘‘A Comparasion of Multiprocessor Scheduling Heuristics,’’ *Proc. International Conference on Parallel Processing*, vol. II, pp. 243-250, 1994.

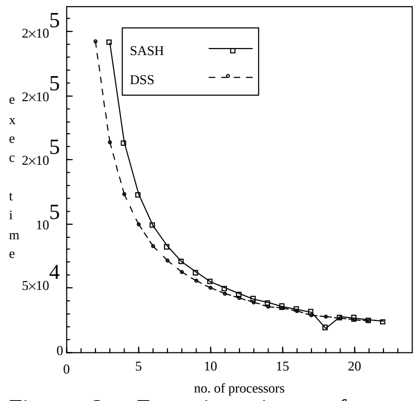

Figure 2.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 500, \sigma_{cc} = 10$

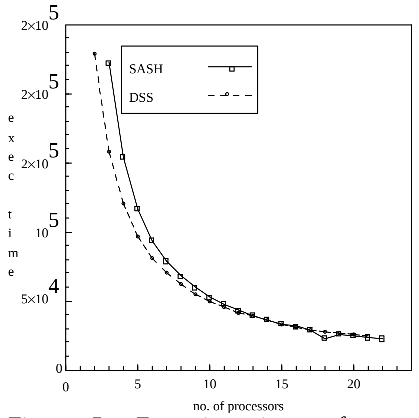

Figure 5.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 500, \sigma_{cc} = 50$

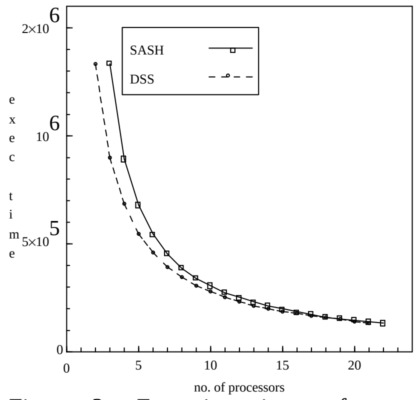

Figure 3.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 5000, \sigma_{cc} = 100$

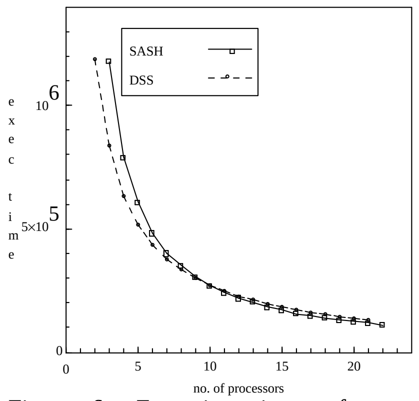

Figure 6.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 5000, \sigma_{cc} = 500$

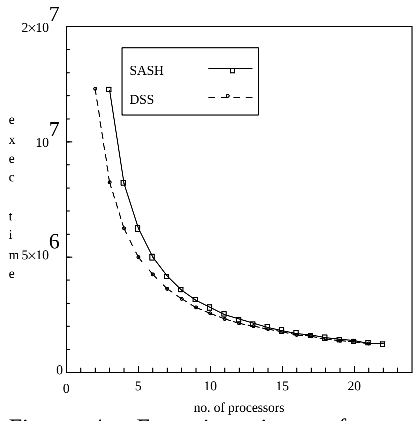

Figure 4.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 50000, \sigma_{cc} = 1000$

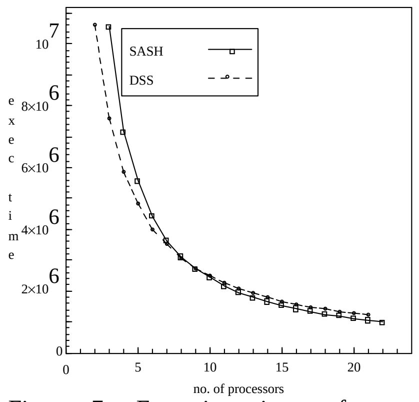

Figure 7.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 50000, \sigma_{cc} = 5000$

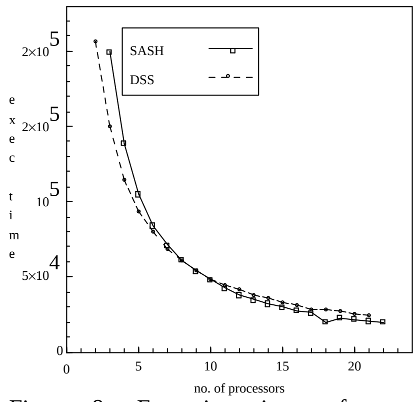

Figure 8.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 500, \sigma_{cc} = 100$

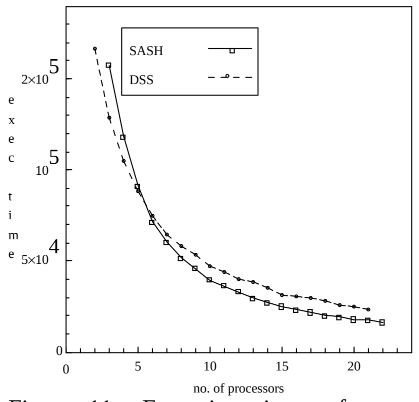

Figure 11.: Execution times of SASH and DSS for $\sigma_{cp} = 100, \mu_{cc} = 500, \sigma_{cc} = 100$

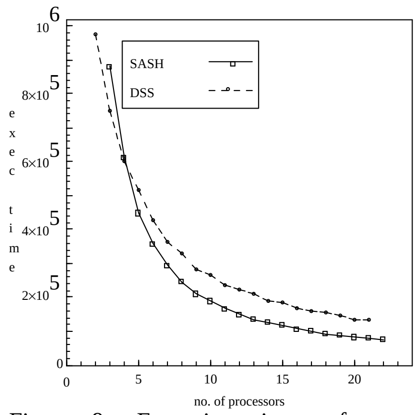

Figure 9.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 5000, \sigma_{cc} = 1000$

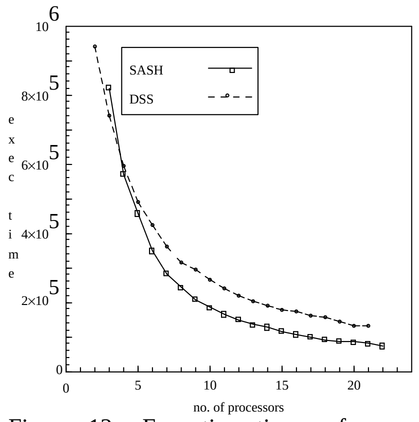

Figure 12.: Execution times of SASH and DSS for $\sigma_{cp} = 100, \mu_{cc} = 5000, \sigma_{cc} = 1000$.

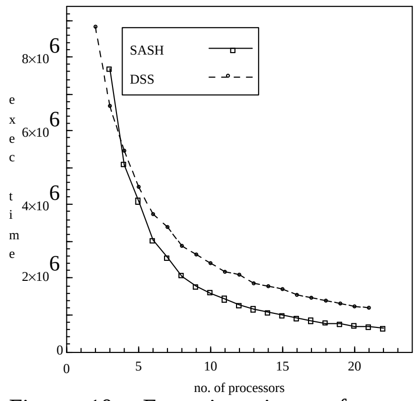

Figure 10.: Execution times of SASH and DSS for $\sigma_{cp} = 10, \mu_{cc} = 50000, \sigma_{cc} = 10000$

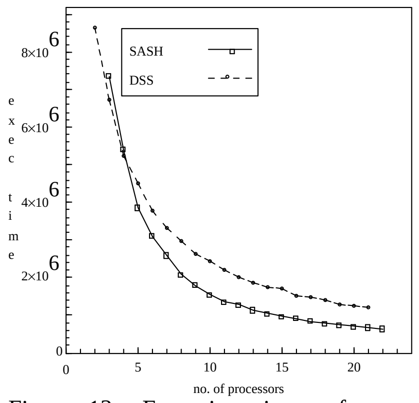

Figure 13.: Execution times of SASH and DSS for $\sigma_{cp} = 100, \mu_{cc} = 50000, \sigma_{cc} = 10000$

- 21 -