A Comparative Analysis of Static Parallel Schedulers Where Communication Costs Are Significant

Douglas Michael Pase
B.S., Northern Arizona University, 1982

A dissertation submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

July, 1989

The dissertation "A Comparative Analysis of Static Parallel Schedulers Where Communication Costs Are Significant" by Douglas Michael Pase has been examined and approved by the following Examination Committee:

Robert G. Babb II
Associate Professor, Thesis Advisor
Oregon Graduate Center

Michael Wolfe
Associate Professor
Oregon Graduate Center

Virginia Mary Lo
Assistant Professor
University of Oregon

Vivek Sarkar
Research Staff Member
IBM Thomas J. Watson Research Center

Dedication

To my God, who made this work possible,

To my wife Anne and daughters Kathryn and Kirja,

Who supported me while I did it,

And to my parents, who encouraged me when I was young.

Acknowledgements

I wish to acknowledge the support and encouragement of the faculty, staff, and students of the Department of Computer Science and Engineering of the Oregon Graduate Center. Their friendship and willingness to listen have been as invaluable as their ideas and enthusiasm. Particularly I would like to thank my advisor, Dr. Robert G. Babb II, and my friend David C. DiNucci for their thoughts and opinions on technical matters, and my committee members for their constructive comments and their patience.

Table of Contents

Abstract

A Comparative Analysis of Static Parallel Schedulers

Where Communication Costs Are Significant

Douglas M. Pase, Ph.D.

Oregon Graduate Center, 1989

Supervising Professor: Robert G. Babb II

Efficient multiprocessor scheduling where communication between processors is free has been studied for almost three decades. However, modern distributed architectures have communication channels for which communication is *not* free. Such channels have a non-zero latency and a finite capacity for communication. Previous work on parallel scheduling accounting for communication effects has assumed that the channels had sufficient capacity to service all transmissions without significant delay from contention. We show that the average schedule length can be significantly shortened by taking contention into account. We define families of static schedulers based on the strategy chosen for various phases, and present a performance analysis based on that classification. Because certain static schedulers are equivalent to dynamic schedulers for which perfect knowledge is available, parts of this work also apply to dynamic scheduling.

CHAPTER 1

Introduction

1.1. Background

Since the inception of modern mechanized computing a particular theme has occurred many times — how do we solve a given problem faster? This pursuit of greater speed has led to the development of better algorithms, more effective compiler optimization techniques, and faster hardware. Speeding up the hardware could mean improving the speed of a single processor, or it could mean replicating the processors and dividing the problem into smaller units (or tasks) which are then executed in parallel.

A particularly difficult part of executing tasks in parallel is arranging the execution of individual tasks so that the maximum benefit is gained from all the effort. It is easy to see that when there is no additional cost for executing tasks in parallel, doing so will never slow the computation down. However, when parallel execution incurs an additional expense, such as from communication, improper scheduling can actually lead to *slower* program execution.

A number of approaches have been proposed to solve the problem of multiprocessor scheduling considering communication costs. Among them are processor allocation, dynamic load balancing, and static scheduling. Processor allocation problems generally take the form of mapping a program graph $G_p = (V_p, E_p)$ to a network of processors $G_n = (V_n, E_n)$ such that some criterion is minimized [BeS87, Bok81a, Bok81b]. V_p is the set of vertices (tasks) in the program, and E_p is the set of edges (communication arcs). Similarly, V_n is the set of processors in the network, and E_n is the set of communication links which connect the processors. It is called the *mapping problem* if the objective is to minimize the number of

arcs in G_n spanned by arcs in G_p. This assumes that all communication is of equal cost, and the value to be minimized is the distance over which each message must travel. It also assumes that only one task may be mapped to a processor.

Another problem related to processor assignment is called the *quadratic assignment problem* [Bok81a]. In this case there are n objects and n locations. The affinity between objects is recorded in a matrix A, and distances are recorded in a matrix D. The element a_{ij} records the affinity between objects i and j, and element d_{ij} records the distance between locations i and j. The objective is to find a mapping function $p: V_p \rightarrow V_n$ such that the overall cost of communication, $\sum_{i,j} a_{ij} d_{p(i)p(j)}$, is minimized. The affinity between two objects can be thought of as a volume of communication between two tasks. A distance d_{ij} can be thought of as the the cost of communicating between processors i and j.

Dynamic load balancing deals with communication on an indirect level. As work becomes available, it is shipped to the processor which is best able to accept it [Cas87, Ham80, Sta84]. If a task has not received all of its input, it is not ready to be shipped. This is much like the "macro dataflow" model used in [SaH86, Sar87]. Processor selection is deferred until the task can be executed, and the best processor is selected at that moment. Processors are not left idle as long as work is available. A particularly important advantage to this approach is that the schedule adapts itself to the execution of the program as the execution takes place. Thus even programs whose execution are extremely data dependent can use this method of scheduling.

Dynamic load balancing approaches are generally classified as centralized or decentralized. Centralized load balancing has one processor (the master) which is responsible for all scheduling decisions. It tracks the work levels of all processors within the system, and supplies tasks whenever they are available to processors (workers) that need

them. As tasks are completed the worker informs the master who collects and records the information. When all of a task's inputs are available, the master places the task on a ready queue, or assigns it to a worker. Although this approach is simple and effective for small numbers of processors, it does not scale well. Loading of the master is proportional to the number of the processors in the system, so increasing the network size will eventually cause it to be saturated. In addition, as networks get larger there is an increase in the communication delay between the more distant processors and the master, which causes additional processors to be less effective.

Distributed load balancing attempts to remedy these problems by making decisions locally. This means that the ability to make decisions increases with network size. It also means that the distance between the unit which makes the decisions and the unit which executes those decisions is zero. However, because each processor must make decisions about whether to accept or forward tasks, and where, each processor must now have some idea of the system state. The system state must itself be communicated through messages which are subject to communication delay, so they may not reflect that state accurately when they are received or used.

Both centralized and distributed load balancing suffer somewhat from the fact that scheduling is done at runtime, and therefore the scheduling overhead is paid for every time a program is run. Little pre-execution program analysis is normally done to aid the scheduler in making its decisions, which prevents processors from planning the execution to minimize the overall processing time.

Static scheduling attempts to solve some of these problems by analyzing the program graph and scheduling it before execution begins. This is necessarily restricted to programs or sections of programs which have little varying dynamic behavior. Our approach to static

scheduling further restricts the problem to the scheduling of tasks with acyclic precedence constraints and heterogeneous task and communication weights. It is a superset of the *Precedence Constrained Scheduling Problem* (PCS) [GaJ79], in that it adds to PCS the additional problem of scheduling communication costs. In both problems the tasks have a finite lifetime and are executed once. All incoming communication must be received before a task may begin, and all outgoing transmissions are sent after the task has completed.

Because the problem in its general form is **NP**-complete [Ull75], solution approaches have taken two diverging paths, namely that of restricting the problem until polynomial solutions may be found, and of finding heuristic algorithms that may be computed more cheaply but still produce schedules that are frequently close to optimal.

Static scheduling may be done as the program is constructed, as a preprocessing phase prior to compilation, automatically or semi-automatically at compile time, or at the time the program is loaded onto the machine for execution. Tasks may represent individual instructions in a program, subroutines, program modules, or whole programs which are part of a script. Communication between these tasks might be the fetch of a datum from main memory, a structured message a few hundred or thousand bytes long, or the transfer of complete files between successive filters. We assume here that the only cost associated with communication is the message transmission time, which includes both the time required to transmit the message over the communication link, and the queuing delay which occurs because of competition from other messages in the system. No setup time in sending or receiving messages is included in this analysis, although there is no reason why it could not have been.

A scheduler is *preemptive* if execution of a given task may be interrupted and suspended to allow another task to execute. It is *nonpreemptive* if the reverse is true, that

is, once a given task is started it runs to completion without interruption.

Scheduling strategies may be further subdivided into optimal and heuristic approaches. Optimal schedulers may use branch-and-bound techniques [KaN84,Koh75] or linear, integer, or dynamic programming [ACD74,LaL78]. These approaches produce schedules from the equivalence class of shortest length schedules (there may be more than one possible shortest schedule), but the schedulers can require running times which are exponential in the number of tasks to be scheduled.

Heuristic schedulers are more difficult to classify because of the great diversity in approaches. However, a distinction can be made between *stubborn* and *non-stubborn* schedulers. A stubborn scheduler will not move or attempt to reschedule a task once it has been scheduled. Non-stubborn schedulers will generate an initial task schedule, then perturb it in different ways hoping to find a better schedule. *List* schedulers are a special class of stubborn schedulers. In this dissertation, a taxonomy of schedulers is developed and the performance of different types is considered.

1.2. Contributions of This Dissertation

This dissertation makes the following specific contributions to the study of parallel scheduling:

(1) We examine five variables in the program/architecture system for their effect on scheduler performance. The program variables are: the distribution of tasks within a program, the number of subtasks within a program, and the average parallelism. The architecture variables are: the average time (latency) required to communicate over empty links and the total number of processors available.

(2) We decompose static parallel scheduler algorithms into three basic parts and examine how different designs for the parts affect scheduler performance. The

subdivisions we consider are: task selection, processor selection, and schedule generation. The task selection strategies we consider include those used in critical path scheduling and in diffusion dynamic load balancing. Processor selection includes strategies where only processor load is considered, where processor load and empty channel communication latency are considered, and where load, latency, and contention are considered.

(3) Several of the static schedulers we examine resemble dynamic (diffusion type) load balancing schedulers. The static schedulers are similar in all important respects except (1) there was no runtime overhead for scheduling, and (2) the static scheduler has complete and accurate information about the entire system at each time a decision about task placement is made. As such the static schedulers delineate the best average performance that could be expected from similar dynamic schedulers.

1.3. Dissertation Outline

The remainder of this dissertation is organized as follows:

Chapter 2 summarizes much of the relevant work which has been done in static scheduling, and particularly in list scheduling. In Chapter 3 we present a precise definition of the multiprocessor scheduling problem. Chapter 4 describes a taxonomy of our 12 schedulers based on their modular decomposition. The construction of each of the 12 schedulers used in later chapters is also given, along with a worst case complexity analysis for each.

A complete description of the scheduler experiment setup, inputs, and environment is given in Chapter 5. We describe the five variables considered to be most relevant to scheduler performance, and the range of values used for each. Chapter 6 analyzes the

effects of each experimental variable on scheduler performance. Chapter 7 analyzes the effects of variables used in scheduler construction on scheduler performance. Our conclusions and recommendations are presented in Chapter 8, and Chapter 9 presents some ways in which this work might be extended.

Appendix A contains graphs of the different task distributions. The remaining appendixes contain the numerical results of the different experiments. In particular, Appendix B presents the experimental results in terms of frequency histograms of schedule length, bar charts of average schedule length, and tables of all seven performance measures. Results are grouped by problem characteristic to show the effect that task distribution, average parallelism, program size, etc., have on the different schedulers. Appendix C gives the same presentation grouped by scheduler to show the effect of different scheduling decisions on scheduler quality. Appendix D contains plots of relative parallelism vs. relative efficiency. Cumulative histograms of relative scheduler performance are given in Appendix E.

CHAPTER 2

Related Work

This chapter summarizes previous work in precedence constrained scheduling (PCS). Because of the scheduler strategies considered in this dissertation, we concentrate primarily on list and list related scheduling strategies. Ullman [Ull75] proved that if task execution times are not equal, or there are more than two processors, precedence constrained scheduling for arbitrary graphs is **NP**-complete. This, in turn, implies that our extended problem is **NP**-hard, because it is a superset of PCS.

2.1. PCS Without Communication Costs

A number of good heuristic solutions to PCS have been proposed in the literature. Although we are considering a more general problem, PCS with non-zero communication costs, these heuristics provide an excellent starting point for developing heuristic solutions to the extended form of PCS.

Much of the material discussing solutions to PCS is collected together into two works. The first is a survey article by M. J. Gonzalez [Gon77], the second is a book edited by E. G. Coffman [Cof76]. Gonzalez [Gon77] surveys some of the major results in scheduling theory known at that time. He classifies scheduling problems by number of processors, task duration, precedence graph structure, task interruptibility, job persistence or periodicity, presence or absence of deadlines, whether resources are limited, and whether processors are homogeneous or heterogeneous. A number of performance measures are also given, including minimum completion time, minimum mean flow time, and maximum processor utilization. Minimum completion time is an appropriate measure for scheduling large single jobs on

multiprocessor systems. Minimum mean flow time is appropriate for scheduling multiple independent jobs in a time sharing environment, where fast turn around time is desirable. Appropriate heuristics and measures are also given for hard and soft real-time environments. Several scheduling algorithms are described, including those in [ACD74], and performance bounds are given.

Coffman *et al* [Cof76] collect into a single work much of what is known about scheduling theory. This work is more varied and in some ways more detailed than [Gon77]. It includes polynomial algorithms for exact solutions to specific subclasses of the general scheduling problem. Solutions include tree-structured task systems, processors with different speeds, and preemptive and nonpreemptive approaches. The problem complexity (its NP-completeness) is shown, and bounds are derived on the performance of several scheduling problems. Lastly, several exact and near exact algorithms are given which use branch-and-bound and dynamic programming (see [HiL74]) techniques.

The earliest reference to PCS and a critical path solution is by Hu. Hu presents the original critical path scheduling algorithm and proves it is optimal if all tasks have equal execution times and the graph is a tree or forest [Hu61]. Coffman and Graham [CoG72] later present a level-by-level scheduling algorithm (CG) which has tighter bounds than does critical path scheduling. Furthermore, scheduling of arbitrary acyclic graphs is optimal using CG if all tasks have equal execution times and there are only two processors. These two scheduling algorithms provide the basic platform from which most of the scheduling heuristics are derived.

In CG as well as Hu's algorithm, the emphasis is on ordering the selection of tasks from which the schedule is generated. A pre-scheduling analysis is done on the program graph, and the tasks are ordered into a list. As a task is removed from the list for

scheduling, each processor schedule is examined and the processor with the earliest finishing schedule is selected. The task is placed at the end of that processor's schedule. A machine schedule contains only the order in which the tasks are executed, the processor on which each task is executed, and the finish time of each.

Many authors explore the advantages and limitations of this approach, among them:

Kaufman, in [Kau74], discusses a heuristic solution to the precedence constrained multiprocessor scheduling problem where the ordering relation forms a tree. Communication is considered insignificant and tasks are nonpreemptive, but tasks may have non-unit weights. Tight bounds are derived which relate his algorithm to an optimal preemptive schedule and to an optimal nonpreemptive schedule.

Adam *et al* [ACD74] compare the performance of five list scheduling algorithms. The schedulers are HLFET (Highest Levels First with Estimated Times), HLFNET (HLFET with equal task weights), RANDOM (task priorities are selected randomly), SCFET (Smallest Co-levels First with Estimated Times), and SCFNET. A dynamic programming preemptive scheduler is also used as a basis for comparison. There were 22 tests pulled from actual programs, mostly written in FORTRAN, and about 900 were generated stochastically. A statistical analysis of variance (AOV) concluded that HLFET performed best. A P=0.01 confidence level was used for the AOV. Tables from the text report that the largest variation between schedulers was about 31 percent. The tests considered 2, 3, and 5 processors.

Garey and Johnson present a solution to the two processor scheduling problem where there is arbitrary start times and deadlines for each of the tasks [GaJ77]. An $O(n^3)$ algorithm gives a schedule whenever one exists. This same algorithm can also be coupled with a binary search to find the shortest such schedule, or to minimize "tardiness". A

number of variations of this scheduling problem are shown to be **NP**-complete.

Bashir *et al* report the results of a statistical study in [BSV83]. In this study all tasks have unit weights, and graphs have between 20 and 48 tasks per graph. 700 graphs are generated at random, and the resulting sample is used to determine the probability that the critical path scheduling algorithm finds an optimal schedule.

Blazewicz *et al* [BWD84] discuss the variation on the scheduling problem where some tasks require two processors simultaneously. They present a general model for this type of scheduling, and an appropriate heuristic. Bounds on the performance of their heuristic are also developed.

Kasahara and Narita describe a fast branch-and-bound approximation scheme in [KaN84]. The initial selection for the branch-and-bound algorithm is determined by a modified critical path algorithm called CP/MISF (for Critical Path, Most Immediate Successors First). CP/MISF uses the standard critical path algorithm with the exception that ties are broken in favor of the task with the greatest number of successor tasks (i.e., tasks between it and the exit node). The approximation/optimization algorithm enumerates all possible solutions, pruning as early as possible any that are clearly inferior. The current best solution is replaced whenever a superior solution is found. Because the number of possible solutions is so large, a CPU time limit was imposed, which causes the solution to be only approximate. Tests were done for graphs with 5-200 tasks, and 2-10 processors, with no communication costs. Experimentation showed that in most cases this approach found an optimal solution within a few seconds. Kohler [Koh75] describes a slightly less refined version of the branch-and-bound algorithm used by Kasahara and Narita. His results strongly agree with those reported in the later article.

Ramamoorthy *et al* [RCG72] develop dynamic programming algorithms that determine (1) the minimum number of processors to process a graph in the smallest possible time, (2) the minimum time required to process a graph on k processors, and (3) whether a graph can be processed in the minimum time on k processors. Two heuristics are also presented, both of which are similar to load balancing. The heuristics are compared against an optimal algorithm for small graphs and two processors. The major thrust of this paper is intended to be the dynamic programming algorithms, but the two heuristics provide ideas on task selection strategies which we use in this dissertation. Ramamoorthy's task selection strategy is different from critical path scheduling in that tasks are scheduled in the order that they become available, which can be used both in a static or dynamic scheduling environment.

Sethi [Set76] discusses some results from [CoG72] which includes an optimal $O(n^2)$ algorithm for scheduling arbitrary directed acyclic graphs with unit weights on two identical processors. He presents a graph labeling function with $O(n+e)$ steps. He also presents a new optimal algorithm for the two processor problem which has complexity $O(n\alpha(n)+e)$, where $\alpha(n)$ is an almost constant function of n.

Graham [Gra69] and Fernandez and Bussell [FeB73] investigate the worst-case performance of a critical path scheduling algorithm. Graham derives bounds for several variations of the job-shop scheduling problem (i.e., PCS), including an upper bound on the schedule length given a fixed number of processors. Although Graham's work considers only a subset of the problem we consider, it does provide some justification for claiming that the distribution of parallelism does not have a major impact on scheduler performance, which we investigate empirically for the larger problem.

More detailed information about the limits of the Coffman-Graham and critical path scheduling algorithms are given in [LaS77], [Kun81], and [Llo82]:

Lam and Sethi [LaS77] discuss the worst case performance of preemptive and nonpreemptive versions of the Coffman-Graham (CG) scheduling algorithms. They show that both algorithms are bounded by $\omega/\omega_o \leq 2-2/m$, where ω is the length of the CG schedule, ω_o is the length of an optimal schedule, and m is the number of processors. Note that this accounts for the optimality of the special case where $m = 2$.

Kunde derives worst-case asymptotic bounds for the critical path scheduling heuristic in [Kun81]. The bounds are derived for the special cases where tasks have unequal weights. Three types of dependency structures are considered, namely trees $(2-2/(m+1))$, anti-trees (exact bounds are not given, but they are generally worse than for trees), and chains (5/3).

Lloyd [Llo82] investigates the worst-case performance of the critical path scheduling algorithm and the Coffman-Graham scheduling algorithm. This analysis presumes that there are a fixed number of available processors, and that additional resources exist. An upper bound is given which depends on the number of processors and non-processor resources in the system. This upper bound is the same for both scheduling algorithms, and is asymptotically the best possible worst-case upper bound.

2.1.1. PCS With Communication Costs

Several recent studies do consider communication costs in their analysis. However, none of the studies are as extensive as we have undertaken here. Three such studies are summarized here.

Kruatrachue considers the problem of communication for a precedence based scheduler in [KrL87,Kru87,KrL88a,KrL88b]. He defines the ISH and DSH schedulers; ISH is a modified version of Hu's scheduler [Hu61]. DSH is like ISH with an extra pass that

duplicates tasks whenever it is beneficial to do so. Task duplication can have the beneficial effect of using idle CPU time to reduce communication. A basic assumption underlying all his results is that contention has an insignificant effect on the performance of a scheduler. All scheduling decisions are made assuming that the only delay in communication comes from channel latency, and that messages rarely interfere with each other.

Granski *et al* [GKS87] present a critical path algorithm suitable for scheduling dataflow graphs on a dataflow machine. Their algorithm schedules conditional branches by transforming the graph into a set of deterministic subgraphs, each element of which represents a possible path of execution. A critical path algorithm is then used to schedule each of the subgraphs independently. Loops are scheduled by first multiplying the weight of each node within the loop body by the expected number of iterations. The loop body is then scheduled as if it were acyclic. Simulated performance of their algorithm shows their algorithm compares favorably with a random scheduling algorithm.

Chester Carroll *et al* discuss a solution based on critical path analysis in [CHA88]. The solution first schedules critical paths (see Section 3.1 or [HiL74]), then adds non-critical tasks later. Task selection for non-critical tasks is done by decreasing distance from the terminal node of the graph, and uses distance from the initial node to break ties. Only "processor rich" systems were used in scheduling, which never blocked task execution because of processor unavailability. Their study considers both latency and contention in communication. Latency is restricted to being no longer than the duration of the average task. Other aspects of communication were also modeled — in particular, both completely connected and star networks with a packet switching protocol were used, and communication buffer size was included in the feasibility constraints. No performance results were reported.

CHAPTER 3

Definitions And Terminology

Briefly stated, the general problem to be considered is: what are the characteristics of parallel schedulers, programs, and architectures which affect resulting performance? Of course this problem is so broad that one can only consider a very small portion of it in a work such as this. For the sake of simplicity we will restrict the problem to static acyclic program graphs and an idealized multiprocessor architecture. In doing so we restrict the problem to what we believe are its principal components. As mentioned before, much work has been done when the cost of communication between processors is zero [Cof76], and some work has been done when communication latency is important but the communication link bandwidth is effectively infinite [Kru87]. We consider architectures that have a finite communication capacity across links, so communication latency and contention may both affect scheduler performance. In this chapter we give definitions and introduce the terminology used in the remainder of the dissertation.

3.1. Task Graph Characteristics

DEFINITION: A *task graph* $G = (A,T)$ is a connected, directed, acyclic graph with heterogeneous non-negative weights on all nodes and arcs.

The set T of graph nodes represents tasks to be performed; node weights represent the computational resources (i.e. CPU time) required by the program to complete its execution. Arcs in A represent communication between tasks; arc weights represent the volume of communication between tasks. It does *not* represent the communication time of the arc — that is a function of the processor schedule. The direction of the arc indicates which task is

the sender and which is the receiver.

Intuitively, a task graph is a way of representing a program. Tasks within a graph are *strict* on all parameters, that is, they must receive all communication before they begin execution. Tasks are modeled as sending messages to other tasks only after the sending task has completed its execution. It is assumed for convenience that every graph begins with a single node and ends with a single node. Graphs which have more than one initial or terminal node may be easily modified to this form by adding special initial and terminal nodes.

Task graphs have no cycles nor conditional execution such as are found in dataflow graphs [Ack82, DaK82, Den80, Gur84]. The restriction on cycles is particularly severe for the representation of programs, as few useful programs are written without some form of iteration structure such as loops, recursion, or generators. Compiler technology in recent years, however, has progressed to the point where loop unrolling may take place as part of the optimizations a compiler is able to use [AlC72]. Loop unrolling partially or completely removes cycles from an otherwise cyclic graph. The unrolled portion of a loop may be represented as a task graph, or the whole loop may be represented as a single node.

DEFINITION: The *parent* relation of a task graph $G = (A,T)$ is the set A of arcs of G, that is, a is a parent of b iff $(a,b) \in A$. $Parent(a)$ denotes the set of tasks $\{ p : (p,a) \in A \}$, which are the parent tasks of a. $Child(a)$ is the set of tasks $\{ c : (a,c) \in A \}$, which are all children of a.

Intuitively, a is a parent of b if b receives a message directly from a; also, b is a child of a.

DEFINITION: The *ancestor* relation of a task graph $G = (A,T)$ is the transitive closure of the parent relation. In other words, a is an ancestor of b iff a is a parent of b, or

there exists some c such that a is a parent of c and c is an ancestor of b. The *descendant* relation is the transitive closure of the child relation.

The ancestor relation is both irreflexive and antisymmetric. Irreflexive in this case means a can never be its own ancestor, and antisymmetric means that it cannot be true that both a is b's ancestor, and b is a's ancestor.

DEFINITION: An *initial node* of a task graph G is a task $a \in T$ which has no parent in G, that is, a is an initial node iff $\forall\ b \in T\ \ (b,a) \notin A$. A *terminal node* of a task graph $G = (A,T)$ is a task a such that a is not an ancestor of any node. In other words, a is terminal iff $\forall b \in T\ \ (a,b) \notin A$.

DEFINITION: The *earliest starting time* (EST)[1] of a task a is $\max_{x \in parent(a)} (EST_x + w_x)$, where w_x is the weight of task x.

A task may begin execution only after all its parents have finished, so a task's EST is the estimated time of the latest parent's termination. EST ignores arc weights because the costs associated with arc weights depend on particulars of the task placement and schedule, which have not yet been determined. The EST of the initial node may be any finite value, positive, zero, or negative, but is usually chosen to be zero for convenience.

DEFINITION: The *latest starting time* (LST) of a task a is $\left(\min_{x \in child(a)} LST_x\right) - w_a$. The LST of the terminal node is its EST.

A task's slack is the difference between its LST and EST. Intuitively, slack measures the freedom available in scheduling the node.

---

[1] These definitions for EST, LST, and slack are equivalent to the classical definitions, such as are found in [HiL74].

DEFINITION: A *critical path* of a graph is a connected directed path, including initial and terminal nodes, for which the slack of each task is zero.

A graph may have more than one critical path, but all critical paths will have the same length. The EST of a task represents the length of the longest path from the initial node to the task. The LST is a linear function of the length of the longest path from the terminal node to the task. Any task scheduled for execution between its EST and LST will not adversely affect the execution of the graph. A task can be scheduled before its EST when the scheduling progresses from bottom to top, in exactly the same way that a task may be scheduled after its LST when the scheduling order is from top to bottom. The two activities are symmetrical[2]. Any task scheduled before its EST will increase the total execution time of the graph by at least the difference between the EST and the scheduled time. Similar results occur if a task is scheduled after its LST[3].

The definitions for EST, LST, slack, and critical path reflect the most optimistic execution possible which will not violate precedence constraints. They are optimistic in that they assume enough computational resources that no task is delayed due to processor unavailability, and that there is no penalty for communication. Even though these values are optimistic they serve a useful purpose as indicators for task priority.

---

[2] The idea of scheduling a task before its EST may be confusing to some readers. To understand how this may occur, one must recognize that the EST is only an estimator which measures the earliest time at which a task may be scheduled *without increasing the length of the schedule beyond the length of the critical path.* Some scheduler designs (e.g. SCFET [ACD74]) fix the termination time of the final task first, then schedule each parent task in succession. If the final task is given a start time which is its EST, then any task which is scheduled before its EST will increase the length of the schedule. Also, when scheduling proceeds backwards like this, it may be necessary to schedule a task before its EST in order to avoid violating precedence constraints.

[3] If more than one task is scheduled outside its EST-LST range, the execution time of the graph may or may not be augmented by the sum of the differences. This is because the schedule of one of the tasks may cause the critical path to change in such a way that the other does not affect execution. For example, suppose two parallel tasks each have LST = EST = 10. If task a is scheduled at time 15 and task b at 20, the execution will be increased by the maximum of the two, or by 10 time units. If on the other hand the tasks are sequential rather than parallel, the increase will be the sum, or 15 time units.

A number of task graph characteristics can affect the length of an optimal schedule. Among them are program size, average parallelism, task distribution, and the arity of its nodes. Program size affects schedule length by determining the total amount of work to be done. More work to be done generally means longer schedules.

DEFINITION: The *average parallelism* of a task graph is the ratio of the total task graph weight to the length[4] of its critical path.

Average parallelism measures the total amount of work that can be done in parallel over the life of the computation. It is also the ideal speedup, given an infinite number of processors with infinitely fast communication between them. Some programs are highly parallel, while others exhibit an average parallelism near unity (they are effectively sequential). An example of a nearly sequential program fragment would be the algorithm in Figure 3.1, which raises a value b to an integer power $y = b^p$ using the binary decomposition of p. This algorithm is very fast, but it has very little parallelism.

An example of a highly parallel program fragment is in Figure 3.2. Assuming + is an associative operation, this fragment may be decomposed into two equally reasonable task graphs, as shown in Figure 3.3 (a) and (b). Different task graphs are possible in this case because of the associativity. Both decompositions have the same number of operations — N additions in each case. However, because of the greater parallelism available in (b), one would expect it to have a parallel shorter schedule than (a) whenever multiple processors were available. The average parallelism of Figure 3.3 (a) is roughly 1; the average parallelism of (b) is $N/\log_2 N$.

[4] The length of a critical path is the sum of the weights of the tasks on the critical path.

```
b ← base
p ← power
y ← unity
while (p > 0) {
      if (p mod 2 = 1) y ← y * b
      b ← b * b
      p ← ⌊p/2⌋
}
```

Figure 3.1. — Program Fragment With Limited Parallelism

```
            DO 10 I = 1, N
                  S = S + A(I)
10          CONTINUE
```

Figure 3.2. — Potentially Parallel Reduction Operation

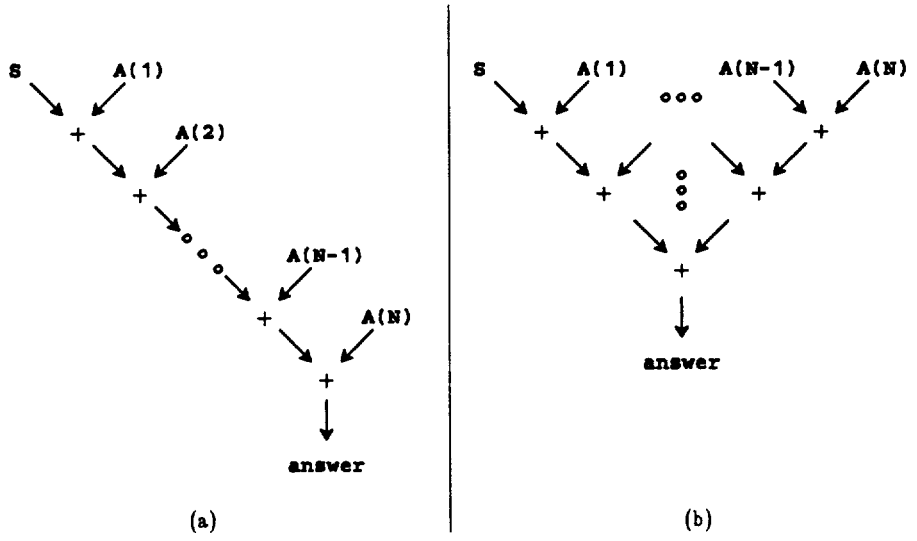


Figure 3.3. — Reduction Operation Task Graphs

The distribution of tasks within a task graph (or task distribution) can potentially affect the execution in several ways. The first and most obvious way is that it determines

the graph's average parallelism. The second is that it determines the amount of slack each task will have. Slack is a measure of how tightly constrained a task is in its schedule. A large slack means the task may be scheduled within a large range of times without directly impacting the overall length of the schedule.

Lastly, the shape of the distribution is capable of affecting the schedules as well. For example, if 90% of the potential parallelism occurs in the last 10% of the graph (as measured along the critical path), the execution will be essentially sequential up to the end of the program, after which the program will load up the processors until its done. If 90% of the parallelism occurs in the first 10% of the program, it might be possible to spread the parallelism across some or all of the execution of the critical path, and thus incur less additional expense. Whether this is realizable depends in no small part on the available slack.

The fan-in, or arity, of nodes in a task graph may affect parallel schedule length because each incoming arc to a task places a constraint on the execution of that task. Tasks must receive communication from all parents before they may begin execution. If the parent resides on the same processor as the child, the communication is free. If the parent does not, a certain penalty in delay and resource usage must be paid. As the arity increases, the likelihood that a parent will be scheduled on another processor, and thus that communication will be across links increases, forcing a tradeoff between communication delay and loss of parallelism. The communication delay can be reduced at the expense of reducing the exploited parallelism. This becomes very important when it forces delays in the execution of tasks along a critical path in the graph, since delays along the critical path cannot be hidden or masked — except by longer delays in parallel parts of the graph.

3.2. Architecture Characteristics

Although graph characteristics are important, they are not the only significant factors in scheduling. Various architectural considerations may also influence the length of parallel schedules. This study considers three of the most important: processor count, link latency, and link bandwidth.

There is a large number of multiprocessor architectures, each with characteristics that are unique, and each with characteristics that are common to other systems[5]. A majority of the systems can be classified as distributed memory, shared memory, or shared address space. For the purposes of this dissertation, it is assumed only that each processor may communicate with other processors via message passing over a network of communication links. This is quite natural to a distributed memory machine. A shared memory machine might also be used as a message passing machine, by using locks or semaphores to signal the arrival of messages. A shared memory system can be viewed as a distributed machine with a completely connected network that has near-zero communication latency.

DEFINITION: A *multiprocessor architecture* is a graph $M = (P,L)$, where P is the set of processor elements, and L is the set of communication links.

Communication links are some combination of uni- and bidirectional arcs, with labels on all arcs and nodes. The directionality of the arcs represents the possible flow of communication across the system. An arc label represents the bandwidth of the communication link which connects processors on both ends of the arc. (There is nothing inherent to our model which precludes communication startup costs from being used, but we

[5] A large number of texts have appeared on this topic in recent years, and three are mentioned here. For a thorough treatment of multiprocessor systems at the architectural level, see [HwB84]. Babb [Bab87] discusses programming different commercially available parallel processing systems, and Chambers *et al* [CDJ84] considers design ideas behind some of the more exotic experimental systems.

do not consider them in this study.) Node labels represent the different capabilities a processor may have, along with the speed with which it is able to perform the work.

DEFINITION: *Link latency* is the time delay per unit message incurred in transmitting a message over a single empty communication link. *Message latency* is the time delay of a given message.

Link latency is also called communication latency, or just latency. The units of latency are generally seconds per bit or seconds per byte, but here we are interested in the time relative to the execution of an average task so it is the average tasks executed per transmission of an average message. Message latency is a function of the link latency and the size of the message to be sent.

Message delay comes from two main sources: delay due to physical properties of the communication circuits (i.e. link latency), and queuing delay due to multiple messages competing for communication links. Queuing delay, or *contention*, depends on the total resources available, the resources used by each message, and the pattern of usage. Network resources are determined by the network size and topology. Message transmission patterns can be co-operative or interfering. If message patterns are (effectively) random, interference depends on the average distance a message is sent and the number of messages in transit. Contention is dependent in part on latency because longer latency means messages take longer to cross a link, and thus the link usage is higher. This in turn causes other messages waiting to use the link to be further delayed.

Not all communications cause an increase in schedule length. Since communication between two tasks on the same processor is "free", delay can sometimes be avoided. Communication between processors that occurs along the critical path will always affect the schedule length (unless something in parallel affects it more). However, if there is sufficient

slack between communicating tasks off the critical path, communication may have no effect at all. But to say that the effect of communication is indirect is *not* to say that it is insignificant. Its significance depends in large measure on the scheduler's ability to take advantage of opportunities to reduce its effects. This dissertation will examine in later chapters the effect message delay has on different scheduler strategies.

3.3. Scheduling Performance Metrics

DEFINITION: A *task assignment* $\alpha: T \rightarrow P$ is a mapping of tasks to a set P of processors.

If task duplication is allowed [Kru87], α is a relation, not a function, because a task may be assigned to more than one processor. The same is true if certain types of preemptive scheduling is used. If task duplication is not allowed and scheduling is nonpreemptive, α also induces a partition, and each task is assigned to exactly one processor.

DEFINITION: A *schedule* $\sigma: T \rightarrow Z$ is a function from a set T of tasks to Z, the set of integers[6]. A *multiprocessor schedule* is a task assignment of T to a set P of processors with a schedule for each processor. All tasks must be executed at least once, and no more than one task may execute at a time on a processor.

Intuitively, T is divided among the available processors, with some tasks possibly occurring on more than one processor. Then σ is a function which returns the start time of a task on a processor. Because the schedules are nonpreemptive, the finish time of a task is the sum of its start time and its weight. A schedule is valid if it obeys all of the constraints, such as precedence, which are imposed upon the task graph. Although we only

[6] In some formulations of the problem, σ is a function to Z^+ or Z^0, and negative integers are not included. We include negative integers in our definition in order to allow a scheduler to fix the termination time of the schedule and work backwards to the starting time. The usual order is to fix the start time at zero or one and schedule forward to the end.

consider precedence in our model, other constraints, such as memory usage limits, are certainly possible.

For reasons mentioned earlier, it is desirable to schedule tasks between their EST and LST. In critical path scheduling, tasks are assigned priorities to establish the order in which tasks will be scheduled. The highest priority goes to the task which will be scheduled furthest outside of the EST-LST range, in order to minimize its impact on the schedule. Scheduling lower priority tasks first can never increase the opportunities to schedule higher priority tasks — it can only fill time slots that higher priority tasks might have used.

On the other hand, always scheduling the highest priority task first will not guarantee an optimal schedule, or even a good one. Suppose there are two unscheduled tasks a and b, and task a has the highest priority of the two. Task a might have several slots which would be equally suitable, whereas because of communication constraints task b might have only one slot which does not adversely affect the schedule length. If task a is given the slot which also happens to be the best slot for b (because it is also the best slot for a by a small margin) the schedule suffers because the completion time for b suffers. If a were less aggressively scheduled, b could take its best slot, and a would be scheduled in a slot that is "almost" as good. This strategy would ultimately give the best overall schedule, but to implement it reliably requires a search of nearly all the possible task combinations.

DEFINITION: The *length* of a task graph's schedule is the difference between the start time of the earliest task, and the finish time of the latest task.

Task weights affect schedule length in an easily understood manner — tasks that are added to the beginning or end of a schedule increase the schedule length by the value of the task weight. Arc weights (i.e. message weights) do not have as direct an influence on the schedule length. If the sending and receiving tasks are both on the same processor, no

message is scheduled, and the schedule length is unaffected. If the tasks are on separate processors, the message must be scheduled on each communication link in the path which is used to transmit the message from the sending task's processor to the receiving task's processor. The amount of time reserved on each link (i.e the message latency) is proportional to the message weight and the link transmission rate. Thus the execution of the receiving task can be delayed by the latency of the message, and possibly more if other messages are competing for the links. Communication delays the execution of individual tasks, which in turn increases the schedule length.

Schedule length measures the total execution time of a given schedule. It is interesting to note that as a scheduler constructs a schedule, it attempts to model "reality" with some degree of accuracy. There may be some differences, perhaps insignificant, perhaps highly significant, between "reality" and a scheduler's perception of reality. Because of those differences, a scheduler's perception of a schedule and the actual schedule may be quite different. This is also true of the scheduler's perception of the schedule length and the actual schedule length.

DEFINITION: The *parallel efficiency* of a task graph schedule is the value $\frac{T_s}{n \times T_p}$, where T_s is the length of a sequential execution of the graph (or the sum of the weights of the individual tasks), n is the number of processors, and T_p is the length of the parallel schedule.

The parallel efficiency describes how effectively the machine is being utilized. Efficiencies near one show that near maximum speedup is being attained, while efficiencies near zero indicate that few of the available resources are being used efficiently. If tasks are not duplicated to increase parallel execution speed, low efficiencies also reflect a high processor idle time. If tasks are duplicated, all processors may be kept busy even when

parallel efficiency is low.

The processor count and average parallelism of the task graph together place an upper bound on the speedup any parallel schedule can display on that system. The bounds are calculated in the following way:

$$Speedup \leq \min(Average\ Parallelism,\ Processor\ Count).$$

Processor count and parallelism also place a limit on the best possible parallel efficiency attainable for a graph. The best parallel efficiency is bounded above by

$$Parallel\ Efficiency \leq \min(\frac{Average\ Parallelism}{Processor\ Count}, 1).$$

CHAPTER 4

Scheduler Components

Stubborn scheduling designs generally consist of three main phases: task selection, processor selection, and schedule generation. Both the length of schedule computed and the running time necessary to create the schedule will depend on the algorithms chosen for these phases. It is interesting that the *interaction between components* can also have a major effect on scheduler performance. For example, we discovered that combining a schedule generator that models communication latency and contention, with a processor selector that uses only latency can be very detrimental. Schedulers which include this pair can generate schedules that are more than 40 times as long as a corresponding sequential schedule. Dividing schedulers into phases gives us a simple taxonomy which will be used to compare schedulers in later chapters.

4.1. Task Selection

The task selection phases can be subdivided into task priority assignment and task selection. Priority assignment is an analysis of the (perhaps partially scheduled) task graph to determine the order in which (remaining) tasks will be scheduled. Tasks are then selected for scheduling according to the priorities assigned. Task priority may be a function of the task's distance from the top of the task graph, from the bottom of the graph, or both [Gon77]. Distance is measured as the sum of the running times of each task along the longest path from the graph start (finish) to the task. No additional distance is usually included because of communication.

```
While unscheduled tasks remain
    Assign task priorities
    Select a task to be scheduled
    Select a processor for the task
    Schedule the task on the processor
End
```

Figure 4.1. — Scheduler With Multiple Task Priority Calculation

The priority assignment may occur as infrequently as once or it may occur more often. A single priority assignment has the advantage of requiring less CPU time to generate a schedule, whereas multiple priority assignments allow the task priorities to adjust to the changing conditions that will occur during execution. For example, the assignment of a particular task to a particular processor could change the critical path of the task graph; reassigning task priorities would allow priorities to reflect such changes. Figure 4.1 shows the scheduling algorithm that recomputes the task priorities each time a task is selected. If the task priorities are computed only once, the priority computation can be moved out of the scheduling loop, as in Figure 4.2. This algorithm is used in all other schedulers.

```
Assign task priorities
While unscheduled tasks remain
    Select a task to be scheduled
    Select a processor for the task
    Schedule the task on the processor
End
```

Figure 4.2. — Scheduler With Single Task Priority Calculation

4.2. Processor Selection

Once a task has been selected for scheduling, a determination must be made as to which processor the task must be assigned to yield the most favorable result. Each processor must be examined to determine which processor assignment will yield the overall shortest schedule. This requires that the "important" aspects of the architecture be modeled. Some features worth considering are communication link latency and capacity, and processor load. Incorrect modeling of an architecture can lead to grossly inefficient schedules, while complete modeling can be prohibitively expensive.

This work considers several processor selection functions, namely random processor selection, selection based on processor load only, selection based on processor load and communication latency, and selection based on load, latency, and communication capacity (or contention). For random selection a processor is selected at random each time a task is to be scheduled. Each processor is equally likely to be selected, and no consideration is given to the architecture or to the schedule generated. When selection is based on load, each processor schedule is examined. The processor with the shortest schedule, i.e., the earliest completion time or lightest processor load, is selected to receive the task. When selection is based on load and latency, each schedule is examined as before but the time required to communicate results to other tasks is also included. It is assumed that each communication link is completely devoid of other traffic, i.e., interference between messages is not considered. This is a reasonable approximation if the average link utilization is generally quite light.

The last processor selection strategy considers processor load, link latency, and contention. Communication contention is modeled by individually scheduling messages on communication links. A separate schedule is maintained for each link in the system. Thus

if a previous message is scheduled to use a link at a given time and a second message would use the same link at the same time, the second message is delayed or, when possible, scheduled earlier than the first. In this way message contention is completely accounted for, and simultaneous transmission of multiple messages over a communication link is not allowed.

4.3. Schedule Generation

Schedule generation deals with the actual construction and recording of the various schedules. It is at this time that a task is actually assigned to the processor which has been selected for it. This can be done in several ways. For a given task assigned to a processor, the generator may place the new task at the top (or bottom) of the processor's schedule, or it may search the schedule for a suitable slot which would not increase the schedule length. Task insertion in parallel scheduling has been the basis of some study [KrL87,Kru87], and although it increases the time required to generate a schedule, it can shorten the schedule length by a modest amount.

Another degree of freedom in schedule generation is the level of architecture modeling undertaken. Just as processor selection may make certain assumptions about the environment in which a program will be executed, the schedule generator must also make assumptions about the environment. And although it is the same environment, each need not necessarily make the same assumptions. For example, it might be the case that only processor load is considered in selecting a processor, but the schedule generator might build schedules which explicitly account for all messages a task will send, and thus model both communication latency and contention. It is worth noting that the processor selector may only use those features modeled by the generator, though it may choose to use fewer. This is because the schedule generator is the mechanism that records the state of the execution

through each step of the scheduled computation.

4.4. Descriptions of Schedulers

Task selection, processor selection, and schedule generation define a taxonomy. Of the 160 schedulers this taxonomy defines 12 were selected for experimentation, for reasons explained below. The specific characteristics of the schedulers used are summarized in Table 4.1, and their relationship to other schedulers may be found in Figure 4.3. A complete description of each scheduler is included in the following sections. The designations in the table indicate how the scheduler was constructed. The two fields under "Priority Assignment" indicate the behavior of that part of the scheduler. The entry "Once" indicates that the scheduler assigns the task priorities once before the first task is scheduled, and they do not change after that point. "Many" indicates that the task

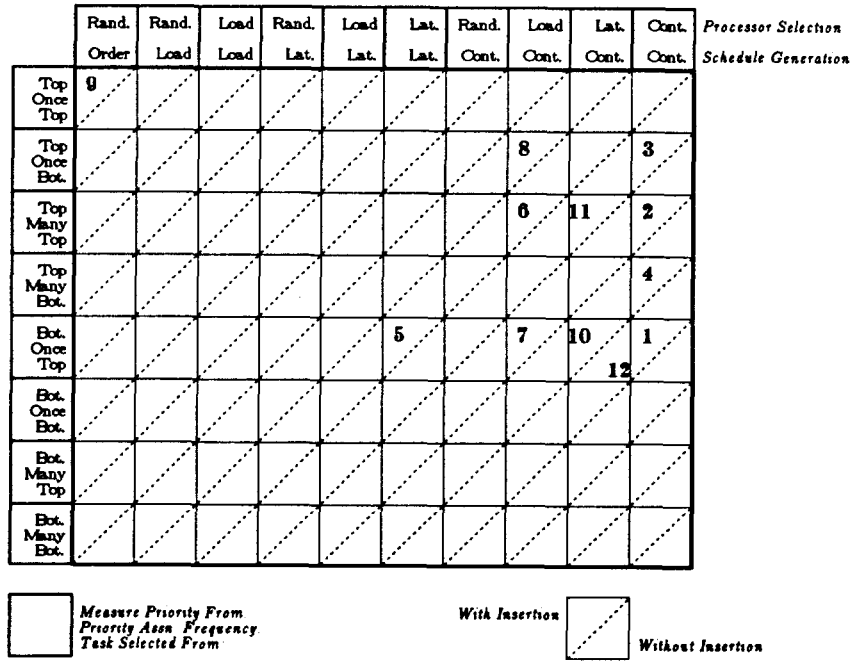


Figure 4.3. — Organization of Scheduler Design Space

Scheduler	Priority Assignment		Task Selection	Processor Selection	Schedule Generation		Complexity
#1	Once	Bottom	Top	Contention	Insertion	Full	$n^2p\ell$
#2	Many	Top	Top	Contention	Insertion	Full	$nm+n^2p\ell$
#3	Once	Top	Bottom	Contention	Insertion	Full	$n^2p\ell$
#4	Many	Top	Bottom	Contention	Insertion	Full	$nm+n^2p\ell$
#5	Once	Bottom	Top	Latency	Insertion	Lat.	n^2p
#6	Many	Top	Top	Load	Insertion	Full	$nm+np+n^2\ell$
#7	Once	Bottom	Top	Load	Insertion	Full	$np+n^2\ell$
#8	Once	Top	Bottom	Load	Insertion	Full	$np+n^2\ell$
#9	Once	Top	Top	Random	Insertion	Full	m
#10	Once	Bottom	Top	Latency	Insertion	Full	$np+n^2\ell$
#11	Many	Top	Top	Latency	Insertion	Full	$nm+np+n^2\ell$
#12	Once	Bottom	Top	Latency	No Insert.	Full	$np+n\ell$

Table 4.1. — Scheduler Design Parameters

priorities are re-calculated each time a task is scheduled. "Top" ("Bottom") indicates that the priority is measured as the distance from the top (bottom) of the task graph. "Top" ("Bottom") under "Task Selection" indicates that task selection occurs from the top (bottom) of the task graph.

Under "Processor Selection" there are four choices, namely, "Random", "Load", "Latency", and "Contention". These indicate the level of architecture modeling that occurs in the determination of each task's processor assignment. "Random" indicates that the processor is selected at random. "Load" indicates that only the processor load, or schedule length, is considered in selecting a processor. "Latency" means that both processor load and communication latency are modeled. An entry of "Contention" indicates that the task is successively scheduled on each processor, complete with task insertion and message scheduling, and the processor which gives the best task completion time is selected.

The "Schedule Generation" columns indicate whether task insertion was used in the final schedule, and at what level the scheduler modeled the architecture. "Full" indicates that both tasks and messages were scheduled on the various resources, i.e., that

communication contention was modeled. "Latency" indicates that tasks were scheduled with sufficient delay that communication could take place across the appropriate processors, but otherwise communication was not considered.

Figure 4.3 includes the entry "Order", to indicate that the order of tasks is maintained on the architecture, but no additional information is retained. A scheduler need not record task start and finish times only if they are not used in either task or processor selection. Any scheduler which selects a processor based on processor load, latency, or contention must record task start and finish times in addition to the order in which tasks are executed. Scheduler #9 selects processors at random, so no start or finish times are needed. In our experiments a separate task graph execution simulator is used to determine the actual schedule length, so all schedules are, in effect, "measured by the same yardstick". This is needed because, as discussed in Section 3.2, each scheduler's perception of task start and finish times may not be consistent with "reality".

This particular selection of schedulers was chosen to compare not only the costs and benefits of different scheduler phase designs, but also the relative importance of each scheduler phase. Several of the most promising phase designs were selected for each scheduler phase. A processor selection phase which selected processors at random was also used, primarily to provide a scheduler which could be used as a standard of reference for other schedulers.

The first four schedulers, #1, #2, #3, and #4, were designed to test the impact of task selection strategies on schedule length and CPU time. Each of these four schedulers are identical in every way, except for task selection strategy. Scheduler #1 uses the same task selection strategy used in Hu's scheduler [Hu61], and in critical path scheduling [Koh75].

Scheduler #2 uses a task selection strategy which is identical to that which is used in dynamic load balancing (cf. [ELZ86,LiK87]), namely earliest available task first, or first come, first served. Dynamic load balancing selects tasks in order of availability, that is, task priority is measured as the distance from the starting point, and the tasks closest to it are selected. In other words, both priority assignment and task selection occur from the top of the task graph. Because load balancing occurs at run time, the priority assignment is effectively recomputed each time a task is assigned.

One important difference between dynamic load balancing and the corresponding static approach is that the dynamic approach suffers from incomplete knowledge of the system load when decisions are made, due to the necessity of distributing load messages relatively infrequently. For this reason the static approach is an idealized version of the dynamic approach. It provides a lower bound on the schedule length, both because of the availability of complete information at the time the schedule is created, and because the overhead of generating the schedule does not interfere with the execution of the task graph.

In addition to the above approaches, Hu's task selection method was also inverted, that is, priority was measured as the distance from the top of the graph and tasks were selected from the bottom. Scheduler #3 measures the task priorities once, scheduler #4 measures the priorities each time a task is selected for scheduling. Scheduler #4 was chosen to test the importance of multiple passes in the task selection phase.

To understand more fully how the task selection mechanism works, consider the task graph in Figure 4.4. Scheduler #1 uses a task selection phase which measures task priority from the bottom of the task graph, so it uses the LST of each task to determine the task selection order. It selects tasks from the top, so a task with the *smallest* LST is selected first. Thus scheduler #1 would select tasks in the order *a*, *c*, *b*, *e*, *d*, *f*. Ties are not

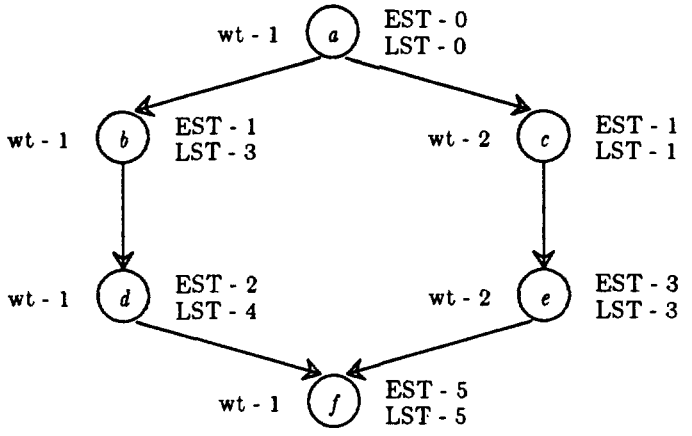


Figure 4.4. — Example Task Graph

explicitly resolved, so the order of tasks *b* and *e* might be reversed.

Scheduler #2 measures task priority from the top of the task graph, so it uses the EST value. It selects tasks from the top, so it would select them in order of smallest EST to largest. The first task to be selected is task *a*. Once scheduled, it would fix *a*'s EST at the actual time it was scheduled, and recompute the EST for all unscheduled tasks. In this way the task selection mechanism receives feedback from the actual schedule, at least as it is perceived by the scheduler. The order in which all remaining tasks are scheduled, therefore, depends on details of the architecture and other phases of the scheduler. After the EST's are recomputed, the next task is selected for scheduling.

Scheduler #3 measures task priority from the top, and selects tasks from the bottom, so it too uses the EST to determine task ordering, but it reverses the order used by scheduler #2. It first selects the final task and schedules it. It then selects another task and schedules it, subject to the constraint that the execution of the second task must complete *before* the final task, including any time needed for communication. Scheduling proceeds in that manner until all tasks are scheduled. The order in which tasks from Figure

4.4 would be scheduled is *f*, *d*, *e*, *b*, *c*, *a*. Scheduler #4 works in the same way as #3, but like scheduler #2 it fixes the EST of a task once it is scheduled, and recomputes all ESTs.

Schedulers #6, #7, and #8 were designed to discern the effect of less expensive processor selection strategies. A major portion of the scheduling expense comes from the processor selection phase of the scheduler, so a less expensive one — load balancing — was substituted. To select a processor, each processor schedule is examined, and the processor with the earliest completion time is selected. Since it was not known whether task selection would have an impact on the effectiveness of a processor selection strategy, three different task selection strategies were used. Scheduler #6 in particular was selected because its design is closest to that of dynamic diffusion scheduling.

Schedulers #10 and #11 were also designed as an attempt to find effective, but inexpensive, approaches to scheduling. The idea was to use load balancing, but include a cost for communication. The main constraint was that the communication cost had to be inexpensive to compute. Message latency was used, since the amount of computation required is proportional to the task arity, and the arity is usually a very small number.

To illustrate the different approaches to processor selection, consider the partial task graph in Figure 4.5 (a). (Only node precedence has been shown — node and arc weights have not been marked, for convenience.) If we are scheduling the graph for a two processor system, a partial result might be the schedule shown in Figure 4.5 (b). Assume that task *c* is the next task to be scheduled.

If the processor selection phase uses only load to select a processor for a task, then *c* will be scheduled on PE 1, because its schedule finishes first. If the communication requirements between tasks *a* and *c* are small, the result will be a tight schedule, as shown in Figure 4.6 (a). However, if the message is large, the results could be very poor, as shown

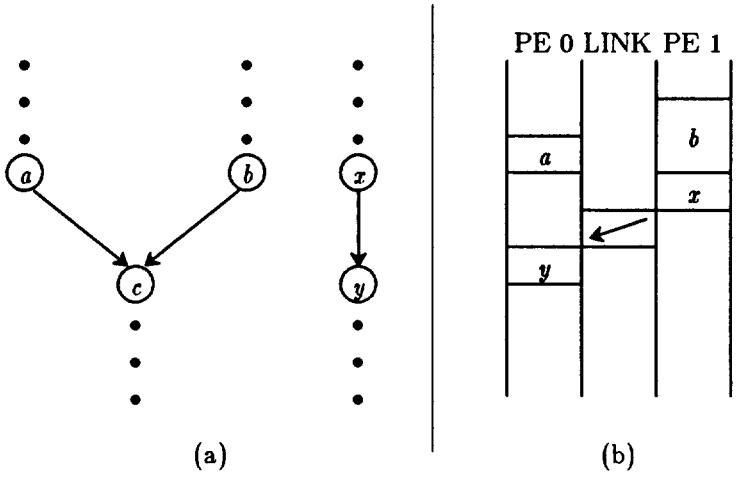


Figure 4.5. — Partially Scheduled Task Graph Fragment

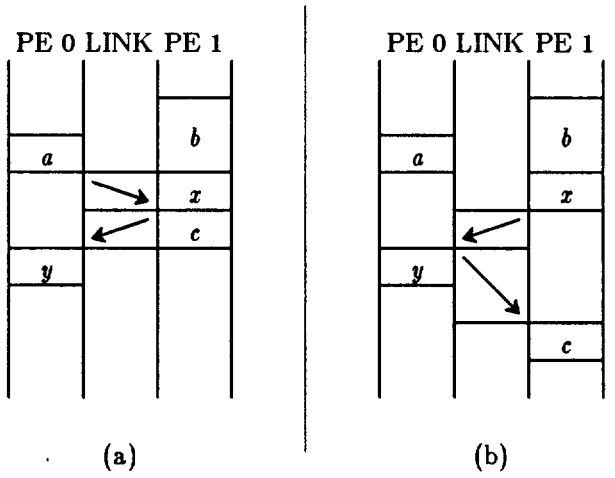


Figure 4.6. — Processor Selection Using Processor Load

in Figure 4.6 (b). It might be better to schedule c on PE 0, depending on the size of the message connecting b and c. The same type of problem can occur if load and latency, but not contention, are used in processor selection, although it would not occur in this example.

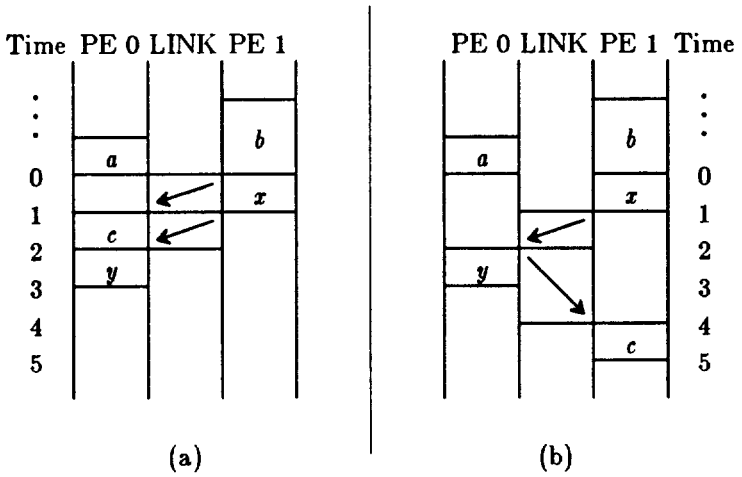


Figure 4.7. — Processor Selection Using Contention

Now consider the situation which occurs if the message from a to c is large, and the message from b to c is small. For concreteness, assume that the messages require empty link transmission times (i.e. latencies) of 2 and 1, respectively, and that the execution time of task c is also 1. The processor selector would first try c on PE 0, the result of which is shown in Figure 4.7 (a). It would then try c on PE 1, shown in Figure 4.7 (b). The first schedule clearly is the better schedule, so PE 0 would be selected.

This example also illustrates the concept of *task insertion*. There are unused spaces in PE 0's schedule which can be used by tasks such as c. When such spaces exist, they can often be filled by scheduling suitable tasks in them, as they were in Figure 4.7 (a). Schedules can be made more efficient with task insertion, because it uses what would otherwise be processor idle time.

The remaining schedulers, #5, #9, and #12, were each selected for different reasons. Schedulers #5 and #9 were chosen to provide a basis of comparison for the other schedulers. Scheduler #5 is Kruatrachue's ISH scheduler, which has been proposed as a solution to this problem [Kru87], and scheduler #9 schedules tasks at random. Scheduler

#12 was selected because poor performance was observed in scheduler #10, and it was thought that task insertion might be causing the problem. Scheduler #12 was designed to test that hypothesis. It differs from #10 in that #10 uses task insertion and #12 does not.

4.5. Scheduler #1

Scheduler #1 (Figure 4.8) is one of several variations on Hu's scheduler [Hu61]. Assigning task priority, as mentioned before, can be done using a standard PERT analysis routine and the tasks put on a heap (also called a priority queue, heaps are discussed in [Sed83]). PERT analysis has complexity $O(m+n)$, where m is the number of precedence arcs in the task graph and n is the number of tasks in the task graph, so task priority assignment is $O(m+n\log n)$, Checking for unscheduled tasks has complexity $O(1)$. Task selection, because of the heap structure, takes $O(\log n)$ time.

Processor selection and task scheduling are a bit more complicated. To select the best processor for a task t, t must be scheduled on each of the p available processors, and the best selection recorded. This involves finding the shortest available communication path from each parent of t to t itself. A standard $O(\ell)$ shortest-path graph search is used, such

---

```
Assign task priorities as distance from the graph bottom
While unscheduled tasks remain
    Select the task that is closest to the top
    Select the processor for which the task will finish
        at the earliest time (including communication)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.8. — Scheduler #1 Algorithm

---

as is found in [Sed83] — ℓ is the number of links connecting the p processors. For this scheduler, each message must be scheduled on each link over which the message is transmitted to avoid overloading the link. Overloading the link would cause a delay in the actual message transmission which would not be anticipated by the scheduler. To further shorten communication time, the link schedule is searched for the earliest time slot which will accommodate the new message. This allows the message to be sent across the link at the earliest possible time after the processor has initiated the transmission. Therefore, assuming the number of parents is bounded above by some small constant (i.e. task arity is independent of the total number of tasks in the graph), the complexity of finding the fastest communication path is $O(n\ell)$.

To insert t into the destination processor schedule, the schedule is searched for the earliest slot which occurs *after* the message arrives. Because this search is $O(n)$, it is overshadowed by the complexity of the communication algorithm, and can be ignored. Now because there are p processors on which to try each task, processor selection is an $O(p\ell n)$ operation. Task scheduling involves at most the same operation repeated once (rather than p times) and does not affect the overall complexity.

Collecting terms together and noting that the loop executes n times, the time complexity for scheduler #1 is $O(n^2 p\ell)$.

4.6. Scheduler #2

This algorithm (Figure 4.9) is very similar to that of scheduler #1, with the notable exceptions that task priority is measured from the top, and it is measured each time a task is selected for scheduling. This behavior is very much like that of a dynamic load balancing system, in that tasks are selected as soon as they become available on a first come, first served basis. It is also similar in that it locally minimizes the system load.

```
While unscheduled tasks remain
    Assign task priorities as distance from the graph top
    Select the task that is closest to the top
    Select the processor for which the task will finish
        at the earliest time (including communication)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.9. — Scheduler #2 Algorithm

Differences are that there is no runtime overhead associated with scheduling, and the scheduler has complete knowledge of the state of the entire system. Dynamic load balancing systems *do* have a runtime overhead associated with scheduling each task. Also, every processor's knowledge about the system is limited by the frequency with which it receives load messages. Because load messages are received relatively infrequently, a processor's knowledge is incorrect by the amount the system state has changed since the last load message was received.

Another, perhaps more subtle difference is that because the entire system state is known, the static scheduler can effectively minimize loading for communication links as well as the processors. It also avoids delays which can result because of the asynchronous nature of a dynamic load balancing system. For example, if two processors each have work to export and they both choose the same processor to receive the work, the receiving processor could end up with work to export — it might have been better if one of the original processors had given the work to a different processor.

The time complexity of this algorithm is very similar to that of scheduler #1. Differences are that the PERT analysis is done inside the loop, and the heap is no longer needed. This means the complexity for scheduler #2 is $O(nm+n^2p\ell)$.

```
Assign task priorities as distance from the graph top
While unscheduled tasks remain
    Select the task that is closest to the bottom
    Select the processor for which the task will finish
        at the latest time (including communication)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.10. — Scheduler #3 Algorithm

4.7. Scheduler #3

Scheduler #3 (Figure 4.10) is very similar to scheduler #1 in its construction. Task priority is assigned once before task selection begins and is not changed. Schedule generation records both processor and link schedules, and both are used in processor selection — communication link capacity is considered in processor selection. The difference is that scheduler #3 measures task priority as distance from the top of the task graph down, and schedules tasks from the bottom up. Its complexity is therefore the same as scheduler #1, or $O(n^2 p \ell)$.

4.8. Scheduler #4

Scheduler #4 (Figure 4.11) is much the same as scheduler #3. Priority assignment is done from the top down, while task scheduling is done from the bottom up. Again, processor selection and schedule generation both consider processor load as well as communication latency and contention. The difference is that scheduler #4 recomputes the task priorities each time a task is scheduled. The complexity for scheduler #4 is the same as that of scheduler #2, namely $O(nm+n^2 p \ell)$.

```
While unscheduled tasks remain
    Assign task priorities as distance from the graph top
    Select the task that is closest to the bottom
    Select the processor for which the task will finish
        at the latest time (including communication)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.11. — Scheduler #4 Algorithm

4.9. Scheduler #5

Scheduler #5 (Figure 4.12) is Kruatrachue's ISH scheduler [Kru87]. It uses the same approach to select tasks as is used by scheduler #1. Task priorities are measured from the bottom up, and tasks are selected from the top down. Each task is tried on every available processor, and the processor with the best finish time is selected. The difference is that scheduler #5 does not model the architecture as completely as does scheduler #1. Scheduler #5 examines the load on each processor, but it computes the communication delay as if the link had no competing traffic. It also ignores communication traffic when it generates the

```
Assign task priorities as distance from the graph bottom
While unscheduled tasks remain
    Select the task that is closest to the top
    Select the processor for which the task will finish
        at the earliest time (considering load and latency)
    Schedule the task on the selected processor
        (recording only processor schedules offset by latency)
End
```

Figure 4.12. — Scheduler #5 Algorithm

schedule.

This simplifies some complexity results for the scheduler. Task priority is $O(m+n\log n)$ and task selection is $O(\log n)$ as they were for scheduler #1. However, processor selection is $O(pn)$, and schedule generation is $O(n)$. Thus scheduler #5 is $O(pn^2)$.

4.10. Scheduler #6

Scheduler #6 (Figure 4.13) has more in common with dynamic load balancing than does scheduler #2. Scheduler #2 uses communication load in its calculations, which load balancing cannot usually do because of a lack of dynamic information. Task priorities are assigned each time a task is selected for scheduling on a first come, first serve basis, and a processor is selected based on its load only. When the schedule is generated, tasks are inserted into the earliest slots which will accommodate them.

In dynamic load balancing systems, this corresponds to the following behavior: Tasks are selected for possible distribution in the order that they are created, that is, earlier tasks are given higher priority than later ones. Tasks are selected for execution on a processor primarily in the order they are received by the processor, but if a task receives all its inputs

---

```
While unscheduled tasks remain
    Assign task priorities as distance from the graph top
    Select the task that is closest to the top
    Select the processor for which the task will finish
        at the earliest time (considering only processor load)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.13. — Scheduler #6 Algorithm

---

and is ready to run, it will begin execution before the earlier task. The processor will execute the later task rather than be idle.

The complexity of task assignment is $O(m)$, processor selection requires a comparison of p load values and thus is $O(p)$. The schedule generator schedules messages on links as well as tasks on processors, as do schedulers #1-#4, so its complexity is $O(n\ell)$. Thus the overall complexity of scheduler #6 is $O(nm+np+n^2\ell)$.

4.11. Scheduler #7

This scheduler (Figure 4.14) is another variant of Hu's scheduler, as are #1 and #5. Task priority is measured as distance from the bottom of the graph to the task; task selection starts at the top of the graph and works down. Processor selection considers only processor load; schedule generation schedules communication as well as processor tasks.

As described in previous sections, task priority assignment is $O(m+n\log n)$. Task selection is $O(\log n)$, processor selection is $O(p)$, and schedule generation is $O(n\ell)$. Overall the complexity of scheduler #7 is $O(np+n^2\ell)$.

```
Assign task priorities as distance from the graph bottom
While unscheduled tasks remain
    Select the task that is closest to the top
    Select the processor for which the task will finish
        at the earliest time (considering only processor load)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.14. — Scheduler #7 Algorithm

```
Assign task priorities as distance from the graph top
While unscheduled tasks remain
    Select the task that is closest to the bottom
    Select the processor for which the task will finish
        at the latest time (considering only processor load)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.15. — Scheduler #8 Algorithm

4.12. Scheduler #8

Scheduler #8 (Figure 4.15) is a variant on scheduler #3 which does not consider communication in processor selection — only processor load. Task priorities are measured from the top, and tasks are selected from the bottom. Processor selection takes place by selecting the processor with the lightest load. Schedule generation explicitly schedules communication. The complexity for this algorithm is the same as for scheduler #7, namely $O(np+n^2\ell)$.

```
Assign task priorities as distance from the graph top
While unscheduled tasks remain
    Select the task that is closest to the top
    Select the processor at random
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.16. — Scheduler #9 Algorithm

4.13. Scheduler #9

Scheduler #9 (Figure 4.16) is different from other schedulers in that it selects processors at random. Task selection is the same as in schedulers #2 and #6, that is, priority is measured, and tasks are selected, from the top. This scheduler roughly corresponds to a dynamic scheduler which selects processors at random. The probability distribution here is uniform for each processor — every processor is equally likely to receive each task. Because of this scheduler's extreme simplicity, the only significant phase is task priority assignment, which is required to maintain the topological ordering of the task graph. Task selection, processor selection, and schedule generation are all $O(1)$ operations, so the overall complexity is $O(m)$.

4.14. Scheduler #10

Scheduler #10 (Figure 4.17) is another variant of Hu's algorithm. It is similar to schedulers #1 and #7 in all respects but one: processor selection considers processor load and communication latency, but not the capacity of the individual communication links. It measures task priority from the top, and selects tasks from the bottom. Its complexity is

---

```
Assign task priorities as distance from the graph bottom
While unscheduled tasks remain
    Select the task that is closest to the top
    Select the processor for which the task will finish
        at the earliest time (considering load and latency)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.17. — Scheduler #10 Algorithm

---

the same as for scheduler #7, that is, $O(np+n^2\ell)$.

4.15. Scheduler #11

Scheduler #11 (Figure 4.18) is, like #2 and #6, similar to dynamic load balancing. In this variant, the processor selection mechanism accounts for communication latency as well as processor load. As before, this scheduler selects tasks from the top in a first available, first served manner. Processors are selected by examining the load of each processor and the cost of transmitting messages from all of the task's parents to the processor. Communication is assumed to be over empty channels. The processor which gives the earliest completion time is selected. The task is scheduled on the processor, and all messages from its parents are scheduled on the appropriate communication links.

Considering latency does not change the complexity of the processor selection mechanism, so the overall complexity of scheduler #9 is $O(nm+np+n^2\ell)$.

4.16. Scheduler #12

Scheduler #12 (Figure 4.19) is identical in most respects to scheduler #10. Scheduler #10, however, had unexpectedly poor performance. Scheduler #12 was created to test the

---

```
While unscheduled tasks remain
    Assign task priorities as distance from the graph top
    Select the task that is closest to the top
    Select the processor for which the task will finish
        at the earliest time (considering load and latency)
    Schedule the task on the selected processor
        (recording both processor and link schedules)
End
```

Figure 4.18. — Scheduler #11 Algorithm

```
Assign task priorities as distance from the graph bottom
While unscheduled tasks remain
    Select the task that is closest to the top
    Select the processor for which the task will finish
        at the earliest time (considering load and latency)
    Schedule the task on the selected processor
        (recording both processor and link schedules,
        but tasks are added to the ends of schedules,
        rather than inserting them into an earlier slot)
End
```

Figure 4.19. — Scheduler #12 Algorithm

hypothesis that the task insertion was hiding some of the communication costs from the processor selector. The difference between #10 and #12 is that scheduler #12 does not use task insertion in its schedule generation. As each new task is added, it is added to the end of the schedule even when earlier slots are available. The complexity for this algorithm is $O(np+n\ell)$.

CHAPTER 5

Experiment Description

Of all the variables that could have been used to generate task graphs and computer architectures, five were selected for examination. They were: task distribution, average parallelism, program size (task count), processor count, and communication (link) latency. Variables which were not examined include the task arity, average task slack, network topology, and communication switching technology and overhead.

Each of the 12 schedulers was tested on 6075 different cases. The cases consist of 63 different simulated programs with 2048 tasks each, 54 simulated programs with 1024 tasks each, 45 programs with 512 tasks, 36 programs with 256 tasks, and 27 programs with 128 tasks. Each of the groups was subdivided by the amount of parallelism, and by the distribution of parallelism within the program.

The average parallelism available in any program depends on the total task weight and the weight of the critical path. For these test cases, the average task weight was fixed at 10, and varied between 6 and 14. The number of tasks in the critical path started at 8 and increased by a factor of 2 up to one fourth the total task count. However, because all task weights were selected at random, the average parallelism varied from less than 3 to more than 256.

Nine distributions of parallelism were used to give a wide spread in the arithmetic mean and a moderate spread in the distribution variances. The distributions were all nonlinearly scaled normal distributions. (Many good statistical texts describe normal distributions and their properties, for example, [HoT77].) The nonlinear scaling compressed

$$tdf(t) = \frac{e^{-\left(\frac{a}{t}+\frac{b}{(t-1)}\right)^2}}{area}$$

Distribution	a	b	Area	Mean	Variance
0	0.10	0.10	0.7470	0.5000	0.0509
1	0.10	0.50	0.4226	0.3164	0.0259
2	0.10	1.00	0.1969	0.1877	0.0114
3	0.50	0.10	0.4226	0.6836	0.0260
4	0.50	0.50	0.3551	0.5000	0.0155
5	0.50	1.00	0.2432	0.3641	0.0086
6	1.00	0.10	0.1970	0.8121	0.0116
7	1.00	0.50	0.2432	0.6358	0.0086
8	1.00	1.00	0.2052	0.4999	0.0060

Table 5.1. — Task Distribution Function Characteristics

the distribution into a finite range without significantly altering its shape. The particulars for each distribution along with the task density function are given in Table 5.1. Graphs of the distributions may be found in Figure 5.1, and in Appendix A. In the graphs the initial node, or starting point of the program is at 0. The terminal node, or finishing point is at 1.

Each program was scheduled for machines with 4, 8, and 16 processors. In every case the processors were connected by a completely connected network — every processor had a communication channel to every other processor. Communication links between processor pairs were bi-directional, and only one message per link could be transmitted at a time. This network topology was selected, not because it is more or less realistic than another, but because it offers very low communication contention. If contention is a factor in scheduling tasks for this network, it will certainly be a factor in scheduling for any other.

Although this topology is very expensive for real systems of even moderate size, using it had a number of advantages. One advantage is that different message switching technologies such as circuit switching or packet switching do not affect either the latency or

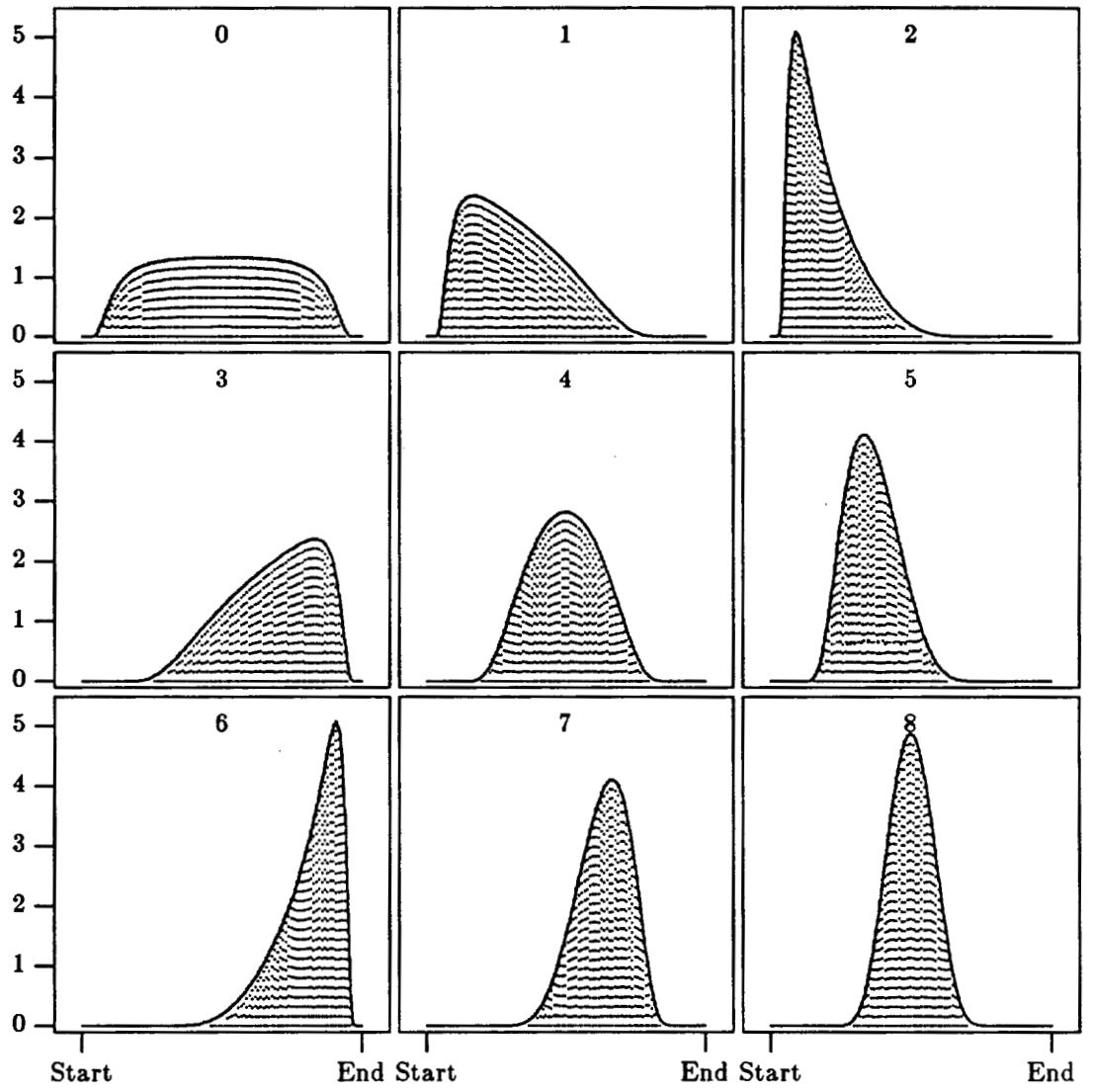


Figure 5.1. — Graphs of Task Distributions

the amount of contention in the network. In sparse networks such as a hypercube, the choice of switching technology and the packet size (when packet switching is used) can dramatically affect both message latency and contention that occurs within the network.

A completely connected network also has the advantage that it has the highest performance of any network. This is because every network can be trivially embedded into

a completely connected network of the same size. Thus negative findings of this study will also apply to other networks as well, although positive findings may not apply as universally.

Communication latency is measured here as the average time required to send a message over an empty communication link (see Chapter 3). We measure time in terms of the the average time required to execute a task, rather than in seconds. Latency is a function of both the average message size and the link speed. Message size is determined by the task graph, and link speed is determined by the machine architecture. The average task graph edge weight (message size) was fixed at 10, and communication latency was set by varying the architecture link speed. If the communication latency were set at 5 and a given message had a weight of 12, its transmission time (message latency) would be $5 \times 12/10 = 6$. In other words it would take the same amount of time to transmit that message over a single empty link as it would to execute 6 tasks.

Latency was varied in such a way that the communication time of an average message varied from 0 to 16 times the length of an average task. Nine tests were run in which the ratios of message transmission time to task execution time were 0, 1/8, 1/4, 1/2, 1, 2, 4, 8, and 16. This range varies the importance that communication plays in the execution of the program from insignificant to highly significant. A latency of 16 would occur in a system when the message size is large and the communication links slow, or when the task size is especially small. Such was the case in a distributed Prolog system developed for the Intel iPSC [Pas87], although the latency was not as high as 16.

This range of characteristics was selected for the test suite in an attempt to include those characteristics which would most likely be encountered in real systems. Characteristics which would most adversely affect the schedule length, and therefore

distinguish most clearly between schedulers, were also selected. Of those used, latency, parallelism, and processor count produced the largest differences, sometimes reaching a factor of 15 between the best scheduler and the worst. Varying the task distribution or the problem size gave only modest differentiation between schedulers.

Other variables were not considered for various reasons. The most compelling reason was the vast number of cases that would result if they were included. Instead, reasonable values were selected where possible and used for all tests. For example, average task arity was fixed between two and three for all tasks. The arity for individual tasks was allowed to vary randomly according to the requirements of the graph.

Slack was not expressly fixed, but it was not a controlled variable either. The average slack of a task in a task graph varied from 2 to 14, or 1/6 to a little less than 1½ times the average task weight. The overall average was 6, or 3/6 the weight of an average task.

Task graphs were generated in a very controlled manner in order to guarantee a specific set of characteristics. This tight control, however, removes a significant element of randomness, which weakens the interpretation of statistical tests. The validity of any conclusion rests heavily on the assumption that those variables which were controlled are, in fact, the factors which control the performance of the schedulers tested here for "real world" programs.

The task graph generation program created graphs by cutting the task distribution into slices — as many slices as there were tasks to be assigned to the critical path. The number of tasks allocated for each slice was proportional to the area under the slice, the total graph size, and was inversely proportional to the length of the critical path. Task weights were all generated at random using a distribution similar to Distribution 4 (see Appendix A) Task graph edge weights (message volume) were generated in exactly the same

way.

Each slice was "sewn" to the previous one, generating the arcs and arc weights at random. To do this, each task in the new slice was assigned two numbers, corresponding to two tasks in the previous slice, using a uniform distribution. Those two arcs were then established and the arc weights generated, again using Distribution 4. Although this process connected all tasks in the new slice to some task in the previous slice, it was not sufficient to guarantee that each task in the previous slice would be connected to some task in the new slice. A second pass searched the previous slice for unconnected tasks, which were then linked into the graph in the same way.

This approach limits the possible connection patterns between tasks to level graphs. In general, task graphs will have connections that span multiple levels. It is not known if more general connection patterns would affect the results uncovered by these experiments. It should be noted, however, that although the the number of tasks in any path to a given node will be the same, the sums of the task weights will be different because the task weights vary randomly.

Figure 5.2 shows a block diagram of the experimental system used to generate task graphs, schedules, and verify schedule lengths. Task graphs are generated according to the supplied parameters. Those graphs are fed into a scheduler, along with a description of the architecture for which the graph is to be scheduled. The schedule is then fed into a simulator which determines the actual schedule length, or how long it would take to execute that schedule on the architecture.

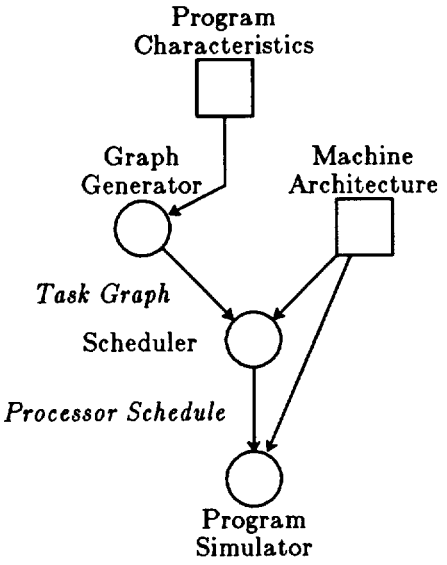


Figure 5.2. — Experimental System Setup

CHAPTER 6

Problem Characteristics and Scheduler Performance

The length of a parallel schedule and the CPU time required to generate that schedule depend on the specific characteristics of the problem being scheduled. This chapter explores the the effects of five problem characteristics on scheduler performance. Those characteristics are: task distribution, average parallelism, program size, communication latency, and processor count.

Scheduler performance is measured and compared here in seven different ways, each of which is marked with its own symbol. They are:

%P≤S The percentage of parallel schedules that were shorter than a sequential schedule.

S/P The speedup gained by the parallel schedule as compared with a sequential schedule, or $\frac{T_s}{T_p}$. T_s is the length of a sequential schedule and T_p is the length of the parallel schedule.

S/C The speedup gained by the corrected schedule as compared with a sequential schedule, or $\frac{T_s}{T_c}$. T_c is the length of the corrected schedule.

P/C The speedup gained by correcting the parallel schedule for schedules which are longer than a sequential schedule, or $\frac{T_p}{T_c}$.

P Eff The *parallel efficiency* of a schedule as defined in Chapter 3, or $\frac{T_s}{n \times T_p}$, where n

is the number of available processors.

C Eff The *corrected parallel efficiency* of a schedule, defined as $\frac{T_s}{n \times T_c}$, where n is the number of available processors.

CPU Sec The average number of CPU seconds on a Sequent Symmetry™ used to schedule the programs. (The theoretical worst case complexity for each scheduler may be found in Chapter 4, in Table 4.1.)

Appendix B gives tables of these seven values for each scheduler, projected over each problem characteristic. It also has frequency histograms for the schedule lengths, and bar charts for the average parallel schedule length and average corrected parallel schedule length, compared against a sequential schedule.

It is important to note that in treating each characteristic separately there is an implicit assumption of independence. If, as will be assumed, the effect of changing one characteristic is qualitatively independent of changes in other characteristics, this analysis will hold. To take the most general case and assume complete interdependence would necessitate the display and analysis of a 6075 point, 5 dimension surface, which is difficult for a discrete problem space such as this. By treating the characteristics as independent, the analysis becomes somewhat more tractable.

6.1. Distribution

The first problem characteristic to be considered is the distribution of parallelism within a task graph. As can be seen Appendix B section 1, each of the schedulers show very little variation in performance between distributions. Because the performance is very different between schedulers, a direct comparison is not possible. However, if the distribution has a minimal effect on performance, as claimed, then the ratio of performance

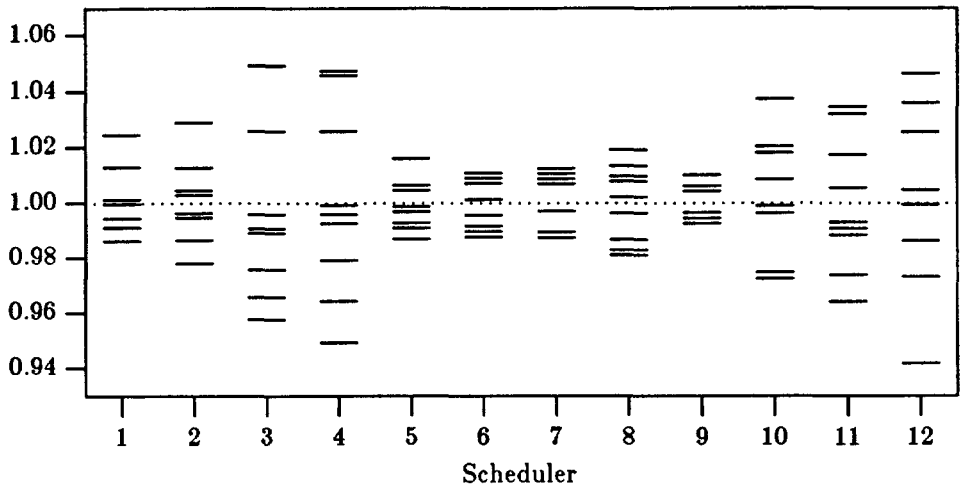


Figure 6.1. — Distribution %P≤S / Mean %P≤S

for each distribution to the average performance over all nine distributions will be close to 1.00.

Figure 6.1 plots the variation from the average for "%P≤S", the percentage of parallel schedules that are shorter than sequential schedules. The variation is figured over the different task distributions by dividing the value for each distribution by the average over all distributions. A large spread between distributions in this ratio would serve as a strong indication that scheduler performance is strongly dependent upon the distribution of parallelism within the task graph. Conversely, a small spread would be a strong indication of independence between scheduler performance and task distribution.

Figure 6.1 clearly shows that distribution has very little effect on the number of parallel schedules that are shorter than sequential schedules. (The widest range in variation is only about 10% of the average.) This holds true for schedulers which generate very short schedules, such as scheduler #1 as well as those whose schedules are almost all longer than the sequential, such as #12.

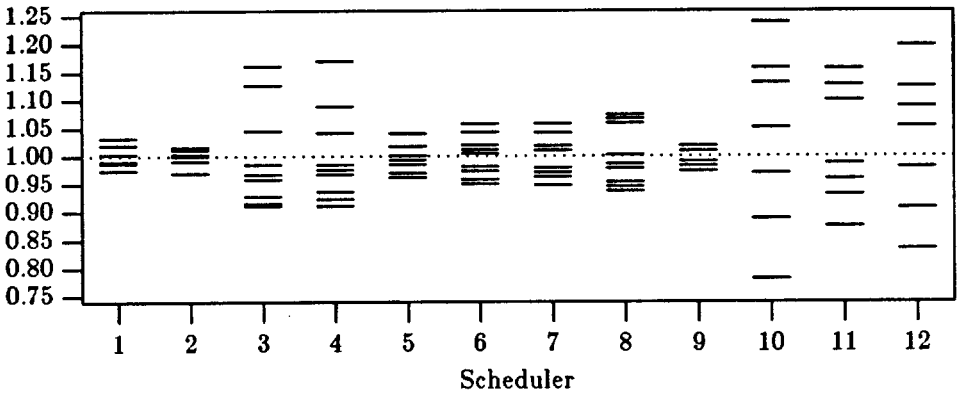


Figure 6.2. — Distribution S/P / Mean S/P

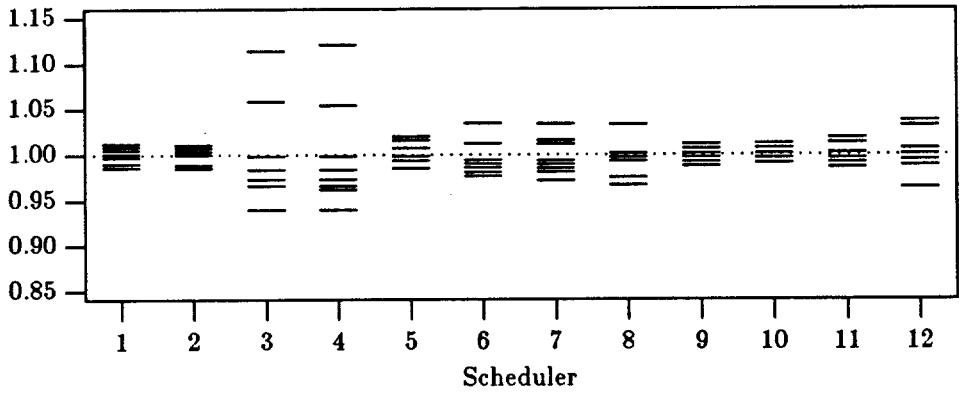


Figure 6.3. — Distribution S/C / Mean S/C

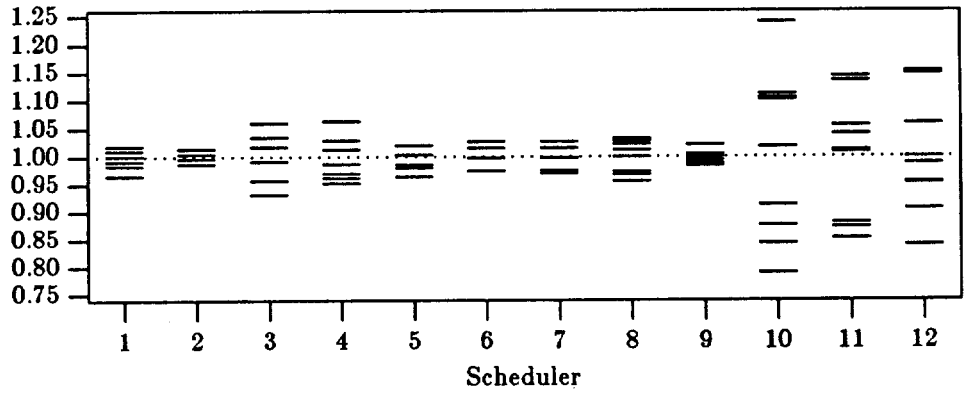


Figure 6.4. — Distribution P/C / Mean P/C

Figure 6.2 shows the ratios for "S/P", or the average speedup of a parallel schedule over a sequential schedule. The differences here are somewhat larger than for the previous graph, but they are still relatively small, so average speedup is also independent of task distribution. Plots for the other performance indicators are given in Figure 6.3 through Figure 6.7.

From these diagrams it is evident that all of the performance measures are independent of the distribution of parallelism within the graph being scheduled. However, some anomalies in the plots are worth pointing out. For instance, schedulers #10, #11, and #12 have the most variation in the different indicators, with the exception of Figure 6.3. This may be explained by referring to Appendix B Sections 1.10, 1.11, and 1.12. A large number of the parallel schedules are much longer than a corresponding sequential schedule, so the corrected schedule will use the sequential schedule a disproportionately high number of times. This means the comparison is more of sequential schedules against themselves than for other schedulers. To further illustrate this point, if the schedulers had always chosen parallel schedules longer than the sequential, then the correction process would always select the sequential schedule over the parallel, and the variation would be zero.

Of the schedulers which do a complete architecture simulation, namely #1, #2, #3, and #4, schedulers #3 and #4 have the widest variation in schedule speedup. Of the group #5 through #9, scheduler #8 has the widest variation. Schedulers #3, #4, and #8 have one thing in common — they measure task priority from the top of the task graph and select tasks for scheduling from the bottom. This pattern of task selection causes the schedulers to be marginally more sensitive to the distribution of parallelism than other schedulers.

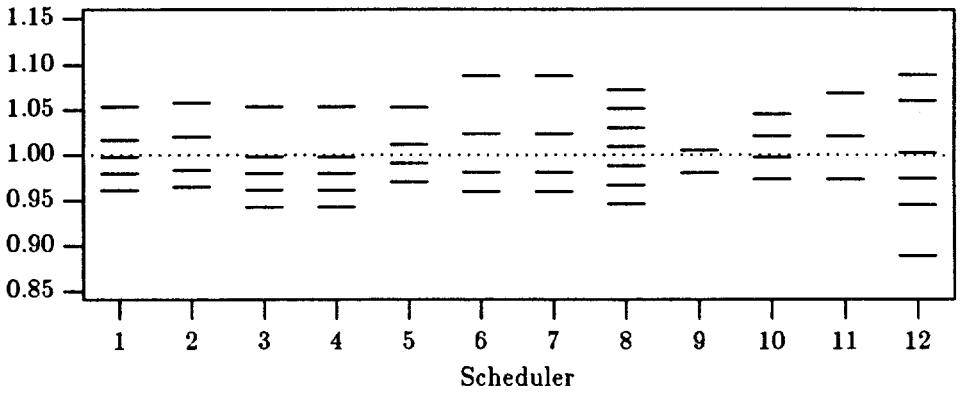


Figure 6.5. — Distribution P Eff / Mean P Eff

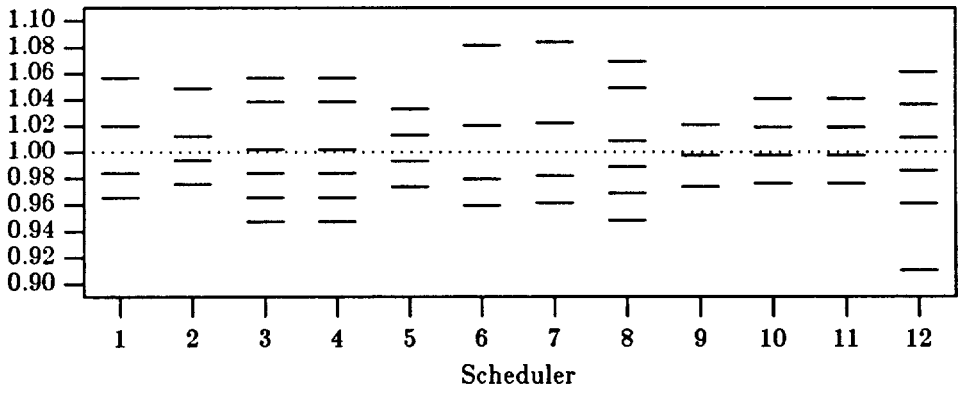


Figure 6.6. — Distribution C Eff / Mean C Eff

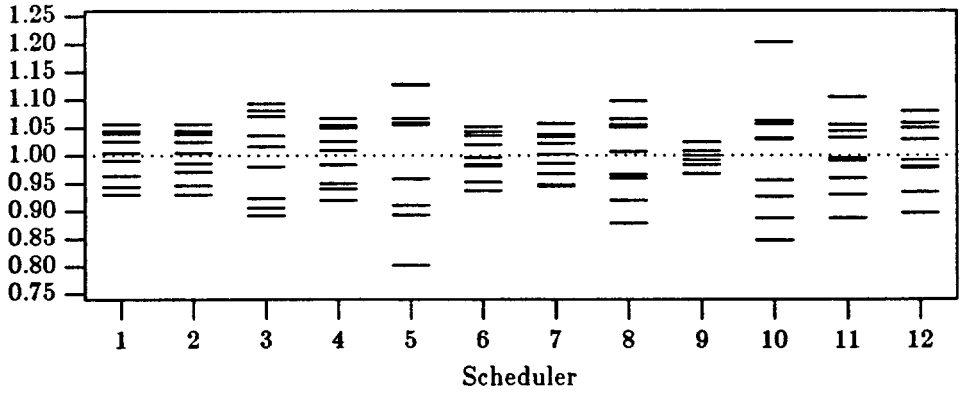


Figure 6.7. — Distribution CPU Sec / Mean CPU Sec

It is also interesting to note that in each of the previous plots the scheduler which showed the least variation was the random scheduler (scheduler #9). The variance was never more than 5% for any of the performance indicators. This shows that there was nothing inherent in the distributions which would cause the performance to be lower or higher for a particular distribution. Instead it was the way the scheduler reacted to the distribution that caused one distribution to fare better than another, even if only in a minor way.

6.2. Average Parallelism

The analysis of the effect of parallelism on scheduler performance is more complex than for the distribution of parallelism. This is because the cost of communication has a disproportionate effect on the schedule length, affecting graphs with high parallelism more severely than those with little or no parallelism.

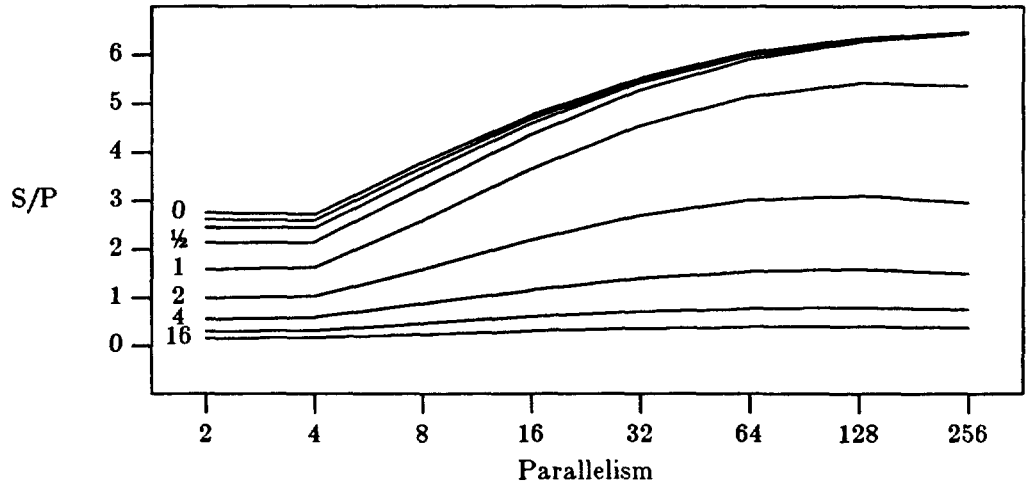


Figure 6.8. — Speedup for Scheduler #9 (Latencies 0-16)

A random scheduler such as #9 is unbiased in the sense that it doesn't use feedback from a partial schedule to determine where other tasks will be placed. Tasks are scattered at random, and performance is determined by the characteristics of the task graph rather than feedback within the scheduler. This is in contrast to schedulers which do use feedback, which can have vastly different schedules for task graphs that have only small differences. This lack of bias can be used to form a baseline for isolating the effects of latency from average parallelism. Figure 6.8 shows the speedup (S/P) for scheduler #9 over different values of average parallelism and latency. Each of the tests were scheduled for 4, 8, and 16 processors.

The average parallelism of a graph is bounded above by the total number of nodes in the graph. This means there is a bias in the average size of each category of parallelism — the average size increases as the parallelism increases. It will be shown later that the speedup is relatively unaffected by the size of the task graph, so graph size can be ignored here.

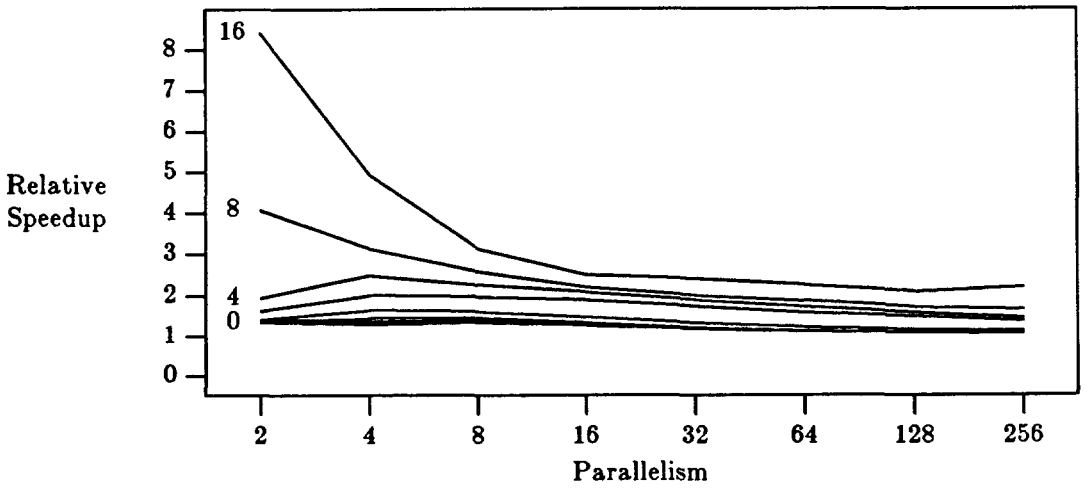


Figure 6.9. — Speedup of Scheduler #1 over #9 (T_9/T_1, Latencies 0-16)

Latency cannot be so easily ignored. To remove the effect of latency, the speedup for each level of parallelism and latency is divided by the corresponding value for the random scheduler. The result is the speedup relative to the random scheduler. Figure 6.9 shows the relative speedup of scheduler #1 to scheduler #9. As it turns out, each scheduler, when compared to scheduler #9, has the same general pattern, but perhaps with some vertical translation and scaling. Each scheduler does somewhat better with low parallelism, but asymptotically approaches a stable linear factor with respect to the random scheduler.

To demonstrate this more convincingly, the variance over a weighted average is shown in Figure 6.10. The weights were chosen to prefer increasing parallelism, and in fact are linearly proportional to the parallelism. From Figure 6.10 one sees that the largest variance from $\bar{y}$ (the sample mean) is approximately 0.35. Because the parallelism is sampled at exponentially growing intervals, this means approximately ½ the weight is placed on a parallelism of 256. From this one can see that if the variance were all to occur on the task graphs with the largest parallelism (which is the most one could violate the asymptotic

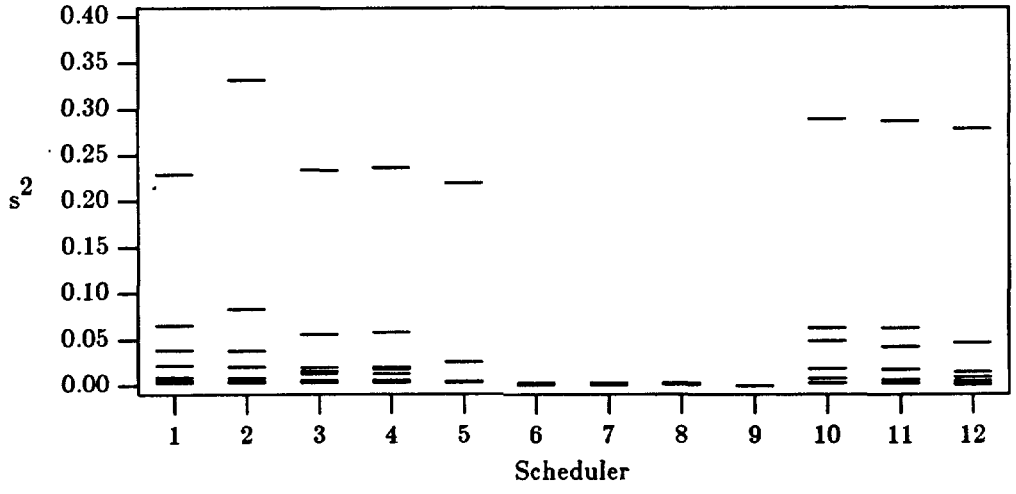


Figure 6.10. — Variances Plotted For Values of Parallelism

performance idea), then $\frac{1}{2}(y_{256} - \bar{y})^2 = 0.35$ This immediately reduces to $|y_{256} - \bar{y}| = 0.84$. In other words, the distance from the asymptotic mean is bounded above by 0.84 over these task graphs. In fact, because the vast majority of the variance in each case occurs in the low parallelism range (as illustrated in Figure 6.9), this analysis is exceedingly pessimistic. Even so it demonstrates that the speedup over the random scheduler approaches a constant factor as parallelism increases.

The weighted mean ($\bar{y}$) for each scheduler and latency are given in Figure 6.11. The weighting function used here is linearly proportional to the communication latency. To reemphasize its significance, Figure 6.11 represents an approximate factor of improvement each scheduler has over randomly scattering tasks across CPUs. Schedulers #1 through #4 outperform the random scheduler (#9) by as much as a factor of 2¼; #5 through #9 perform about the same, and #10, #11, and #12 do much worse.

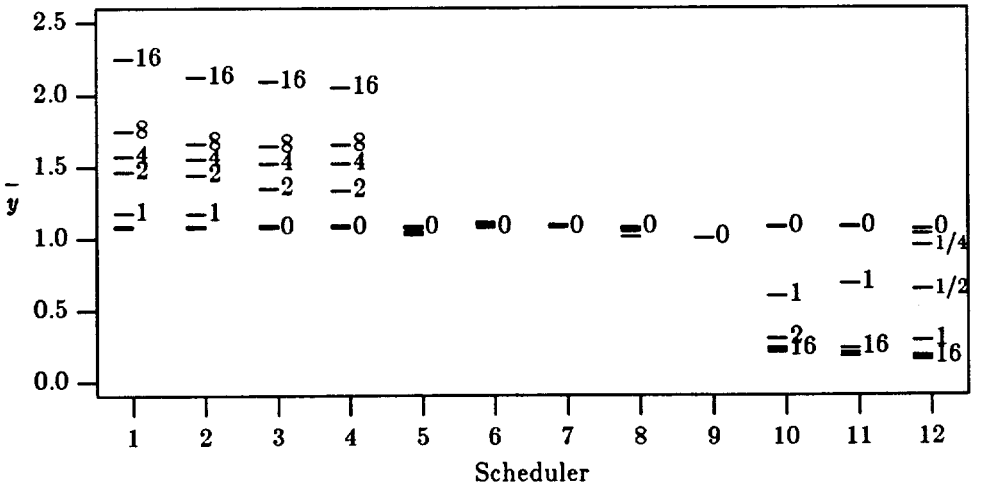


Figure 6.11 — Weighted Means For T_9/T_i (Latencies 0-16)

6.3. Program Size

From Appendix B Section 3 it appears that speedup improves with increasing program size. The differences in speedup between task graph sizes are displayed in Figure 6.12. This graph is obtained by plotting the difference in the average parallel speedup between adjacent graph sizes. Thus the y value for the square symbol (i.e. □) is obtained by subtracting the average speedup for graphs with 128 tasks from the average speedup for graphs with 256 tasks.

If the speedup were completely independent of program size, all differences would be exactly zero or, allowing for random variation, evenly scattered on both sides of the zero line. Figure 6.12 shows this is not what is happening here. Those schedulers which do well show an improvement as the size of the task graph increases. Schedulers #10, #11, and #12, which do poorly under many circumstances, tend to do worse as the size increases.

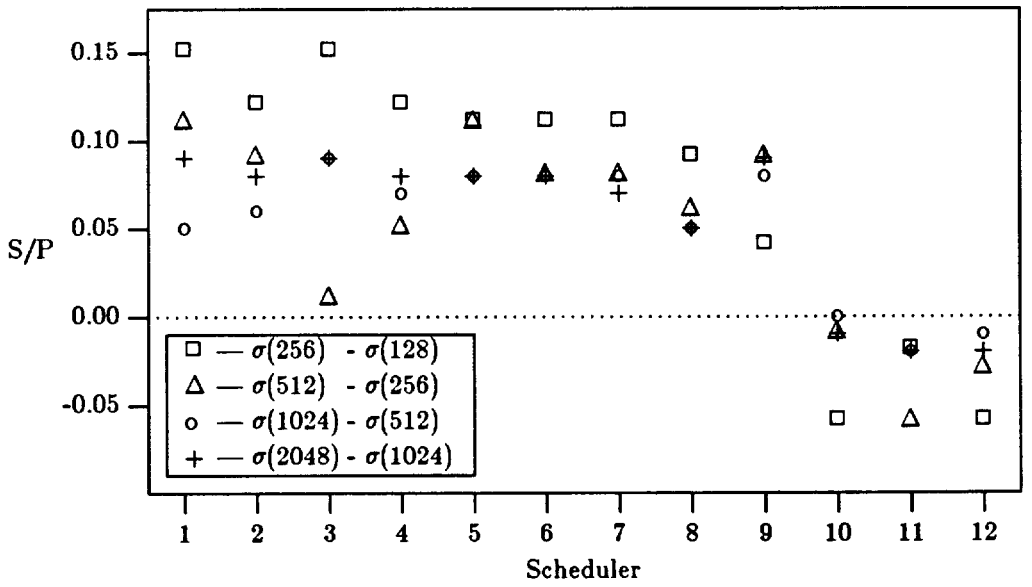


Figure 6.12. — Relative Increase in Speed w/ Increasing Size

Although qualitatively the trend is upward for most of the schedulers, quantitatively the trend is not very significant. For example, increasing the graph size by a factor of 16 only improves the speedup of scheduler #1 by 19 percent. It is expected that this trend diminishes as graphs grow larger (otherwise the parallel efficiency would eventually exceed 1), so increasing the graph size by another 16× would yield less improvement.

6.4. Communication Latency

It is very difficult to directly measure the message delay due to contention, since the delay itself modifies the graph execution. However, one can easily measure the effect of accounting for (or not accounting for) latency or contention in a scheduler. Scheduler #5 (the ISH scheduler) accounts only for the communication delay due to latency, while scheduler #1 accounts for both latency and contention delays. The schedulers are identical but for that detail. Figure 6.13 shows the performance improvement scheduler #1 enjoys over scheduler #5 for different latencies.

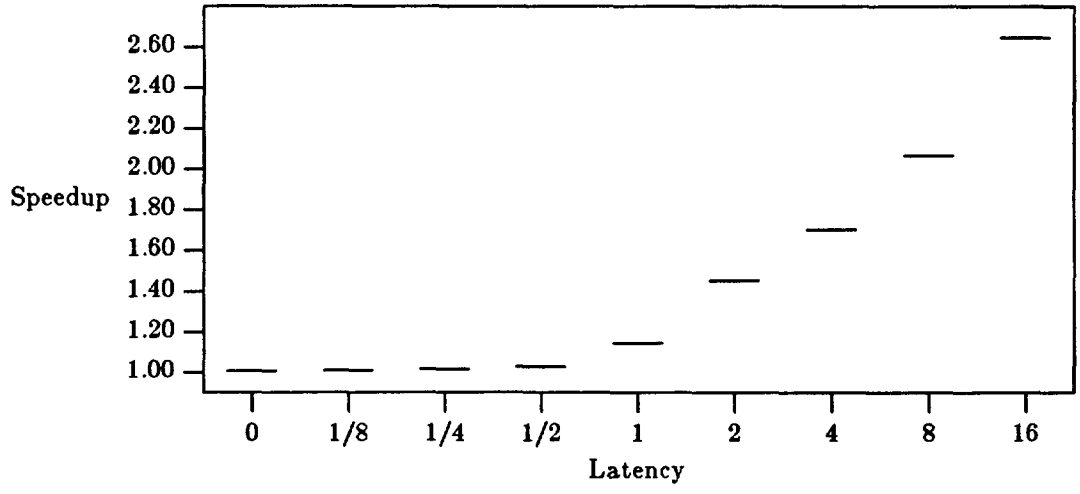


Figure 6.13. — Average Parallel Schedule Length: Scheduler #5 / Scheduler #1

When the cost of communication rises, whether due to latency or contention, the speedup will decline because of the higher cost of doing work in parallel. This is true of all parallel schedulers. The curves for schedulers #1 through #12 are given in Figure 6.14.

The important features of this graph are the backward "S" shape, that every curve dips below 1.00, and that the schedulers fall naturally into 3 main groups or families. The "S" shape indicates that increasing latency causes decreasing performance. Because speedup values can never be below zero, the curve levels out. It is important to note that in *every* case the curve approaches a value which is *below* 1.00. This shows that for high latencies, none of the schedulers tested here is able to consistently generate parallel schedules that are better than sequential schedules.

Although the curves appear to be asymptotic, such is not the case. If the latency were greater than the total task weight of the program, the processor selection phase would have no justification for using any parallelism whatsoever, and S/P would be exactly 1. The reason each scheduler has a region in which it does worse than sequential is that every time

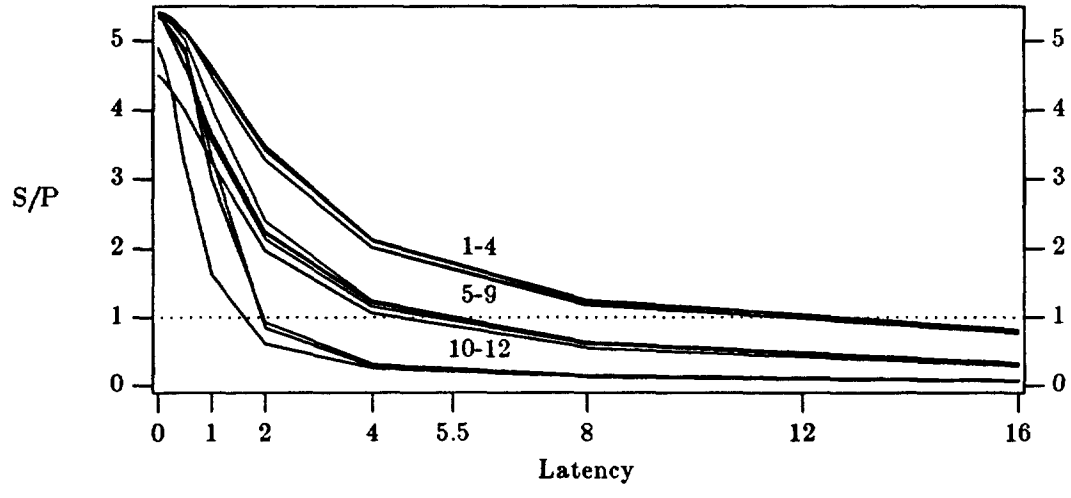


Figure 6.14. — Parallel Speedup vs. Latency

a task is scheduled, the processor selection phase looks only at the effects of past decisions on the execution of the current task. It does not look at what effect the current processor selection will have on future decisions. The schedulers do not consider how much of the program remains nor the expense of transmitting results to where they will eventually be needed. Instead they choose to execute a task in parallel whenever its completion time will be earlier, even if takes longer to get the information where it's needed.

The fact that every scheduler has some latency value for which a sequential schedule is better is a surprising result. For most of the schedulers used, it is a simple matter to compare the parallel schedule against a sequential schedule and select the shorter of the two. The two schedulers for which this comparison cannot be easily made are schedulers #5 and #9. Scheduler #5 is Kruatrachue's ISH, and #9 is the random scheduler. Neither of these schedulers record communication schedules, so neither has an accurate estimation of what the parallel schedule length actually is. It is because these two schedulers do not record communication that they require less than 1/10th the CPU time of other schedulers to generate a schedule.

From Figure 6.14 it is apparent that the schedulers tested here fall into 3 main groups. The group with the best performance consists of schedulers #1 through #4. This group has in common that schedules for the links are recorded along with the processor schedules, and that the scheduler uses communication schedules in its processor selection. The worst performing family (schedulers #10, #11, and #12) recorded communication schedules, but used only communication latency in processor selection. The third family contains a variety of different approaches, whose overall performance was very similar to the random scheduler.

6.5. Processor Count

With a good scheduler one would expect that throwing more processors at a problem would give shorter schedules, and indeed this is the case for the schedulers examined here. Ideally, one would hope that the speedup would be linear with respect to the increase in processors. Unfortunately a number of considerations prevent that from being easily achieved, such as whether there is sufficient parallelism to keep the processors busy, and whether there is sufficient slack in the task graph to allow the parallelism to be used to an advantage.

Comparing the effects of the number of available processors on performance is most easily done by examining the parallel efficiency. Parallel efficiency is the value $\frac{T_s}{n \times T_p}$, or the sequential schedule length divided by the number of processors times the parallel schedule length. It is bounded above by 1 and below by 0. In a sense, it represents the fraction of the machine in use over the execution of a program. In another sense it represents the speedup normalized by the greatest speedup the machine can offer.

However, even though a machine offers 16 processors, a 16× speedup is almost never possible. A more accurate observation is that the speedup cannot exceed the minimum of the average parallelism and the number of available processors [Jor87]. So, if the average parallelism is 2, then no matter how many processors are available, the speedup will never exceed 2. Note that this says nothing about the number of processors required to attain that speedup, except for the trivial observation that it requires at least 2 processors.

This suggests $\frac{T_s}{\min(p,n) \times T_p}$ as a measure of schedule effectiveness. This measure gives the fraction of usable machine which is actually put to use. It gives the parallel efficiency without the inherent penalty for having excess processors. Values close to 1 mean

that the schedule is as effective as possible, given the problem and the architecture. The measure does not account for other variables (such as latency) which can negatively affect the efficiency. Because of this, values not close to 1 may mean that the schedule doesn't use the machine effectively, or that it is affected by some unmeasured variable.

One may go about measuring the effect of processor count on performance in several ways. The most common approach is fix the problem to be scheduled and vary the number of processors directly, reporting any improvement. The disadvantage to this approach is that the measurement is relative to a particular problem, which has its own special set of characteristics. However, performance is dependent on the amount of parallelism relative to the number of processors, *not* the absolute number of processors. By fixing the problem and varying the number of processors one is indirectly measuring the effect of relative parallelism.

This gives a second approach, namely compare the relative efficiencies against the relative parallelism. This has the advantage that it measures scheduler performance against the best that could be done given the machine and the problem. It gives tighter bounds on what is optimal than simply measuring speedup against processor count. We also prefer this measure because it gives a result which is more generally applicable. In any case, we do both. Appendix B Section 5 shows a comparison of speedup and processor count. Appendix D compares relative efficiency against relative parallelism. It can be seen from Appendix D that qualitatively each scheduler performs very nearly the same — each pair of graphs with the same latency have very similar shapes, regardless of the scheduler that produced it. Quantitatively, of course, performance differs between schedulers in the ways described in previous sections.

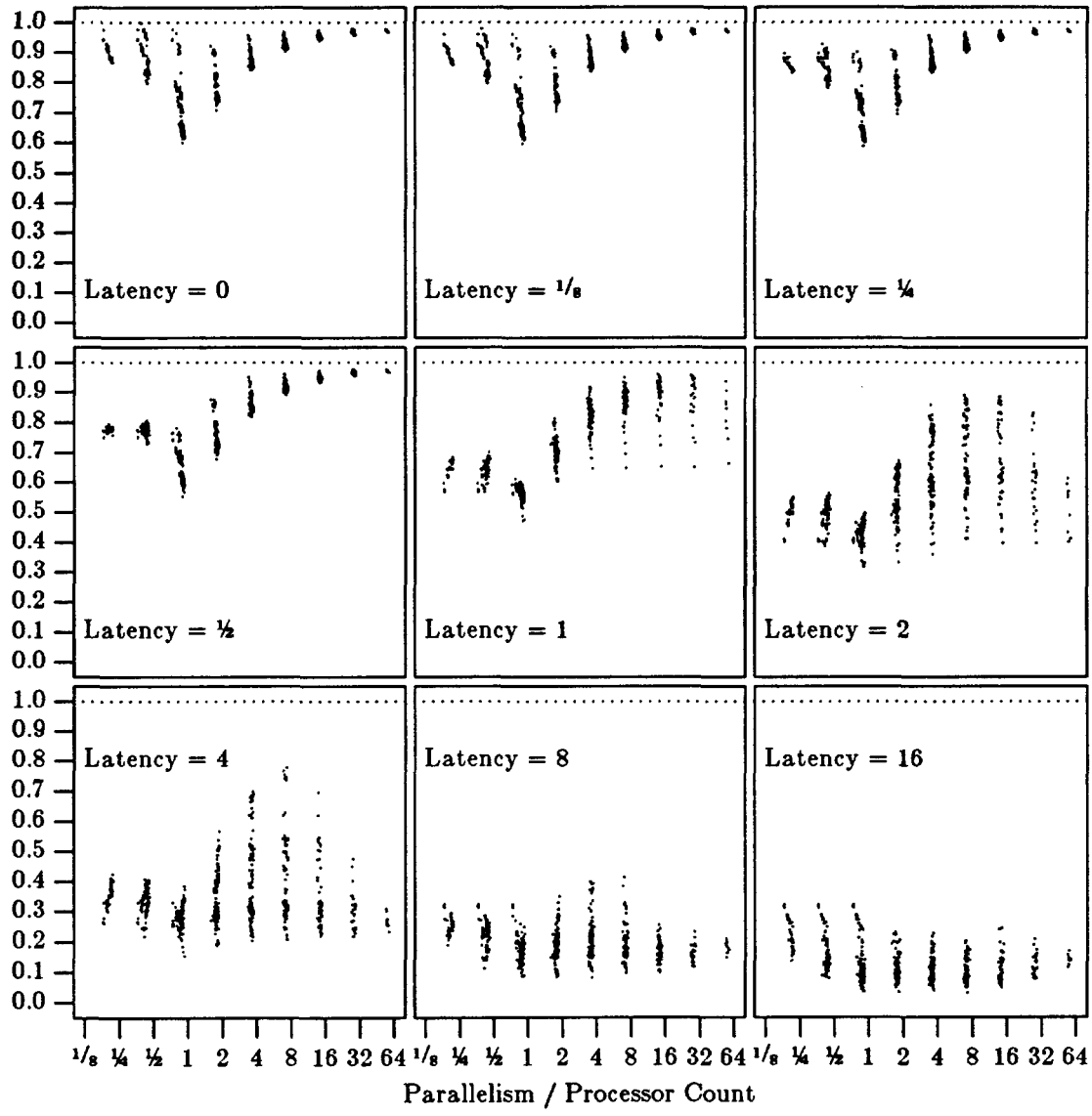


Figure 6.15. — Scheduler 1 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$

The plots in Appendix D show some fairly complex behavior in the scheduler/program/architecture system. For example, based on latency there are three phases that occur, with a gradual transition between them. Using scheduler #1 as the example (reproduced in Figure 6.15), the first phase includes latency values from 0 to ½. This phase is characterized by good performance when there is an excess of either

parallelism or processors. However, when the parallelism and processor count are very nearly the same, the relative efficiency fluctuates between ½ and 1.

The second phase includes latencies between 1 and 4. It is characterized by a wide variation in relative efficiency when the average parallelism exceeds the number of processors, with a relatively small variation when the processor count exceeds the average parallelism. The third phase includes latencies greater than 4, and is characterized by uniformly low efficiencies regardless of the parallelism or processor count.

CHAPTER 7

Comparison of Schedulers

Considering how the different schedulers were constructed, one might expect some similarities as well as some differences in their performance. This chapter explores some of those similarities and differences in terms of parallel schedule length and CPU time required to generate parallel schedules. In particular we examine how different choices in task selection, processor selection, and schedule generation affect scheduler performance.

7.1. Task Selection

As discussed in previous chapters, four task selection strategies were used. The strategy for schedulers #1 and #7 measured priority once as distance from the terminal node of the graph, and selected tasks from the start node, working towards the terminal node. Schedulers #2 and #6 measured priority as distance from the top each time a task was scheduled, and selected tasks from the top. Schedulers #3 and #8 measured priority once as distance from the top, but scheduled tasks from the bottom up. Scheduler #4 measured priority in the same way as #3 and #8, but the priority was recalculated each time a task was scheduled. In this section schedulers which are identical except for task selection strategy are compared across different problem space variables (i.e., task distribution, parallelism, program size, latency and processor count). This is done to determine 1) if the task selection has an effect on scheduler performance, and 2) how that effect is influenced by the different variables.

Schedulers are compared by selecting one scheduler from a group to be used as a reference. Each of the remaining schedulers are then compared against the reference

scheduler. A numerical value is obtained by dividing the average schedule length for the scheduler by the average schedule length for the reference. In this way, if task selection strategy has no effect on schedule length, the numerical values will be close to 1.00. If task selection *does* influence the schedule length, *some* value will be either much greater than or much less than 1.00. If the impact is independent of the problem space variable, all values will have approximately the same magnitude. On the other hand, if the impact is exacerbated by some aspect of the problem space, the numerical values will increase or decrease along with the variable that influences it.

We used two groups of schedulers for these tests. The first group was schedulers #1, #2, #3, and #4; the second was schedulers #6, #7, and #8. The reference schedulers were #1 and #7. Results from these tests show that differences in task selection strategy induce only minor variations in parallel schedule length. Communication latency and processor count emphasize those variations in clearly discernible patterns, but the differences are definitely of minor significance.

The effects caused by task selection may be seen in Figure 7.1 through Figure 7.3. Figure 7.1 shows that the performance of each task selection strategy is not completely independent of the distribution of tasks within a program. The relative performance of each strategy does change somewhat with task distribution, but not in any clear pattern. Performance is independent of average parallelism and program size.

Latency has a very clear effect on the efficacy of task selection strategies. For latencies less than 1 there is no appreciable difference between task selection strategies. As latency increases beyond 1, however, two different strategies take the lead. Which of the two is better depends on other details in scheduler construction. The first strategy is used by both critical path scheduling and scheduler #1. It shows a clear improvement over the

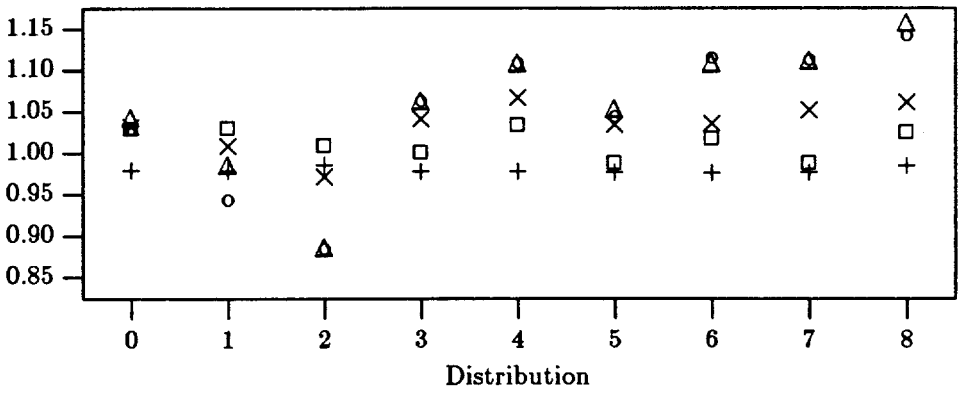


Figure 7.1. — Effects of Task Selection Strategy By Distribution

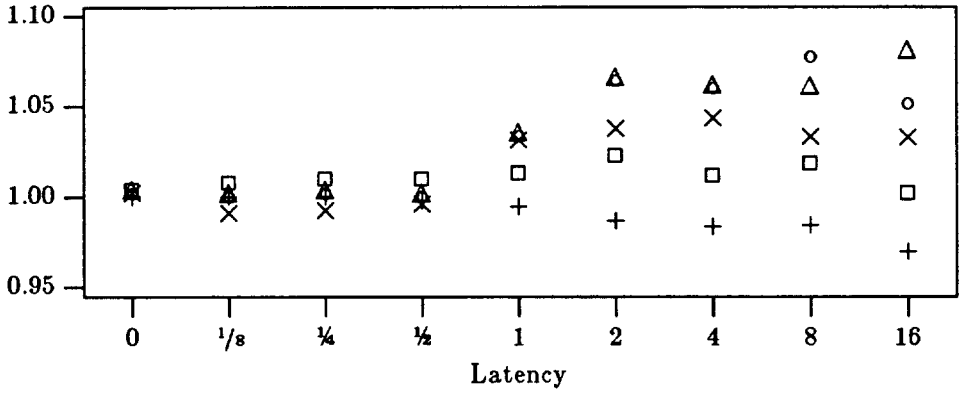


Figure 7.2. — Effects of Task Selection Strategy By Latency

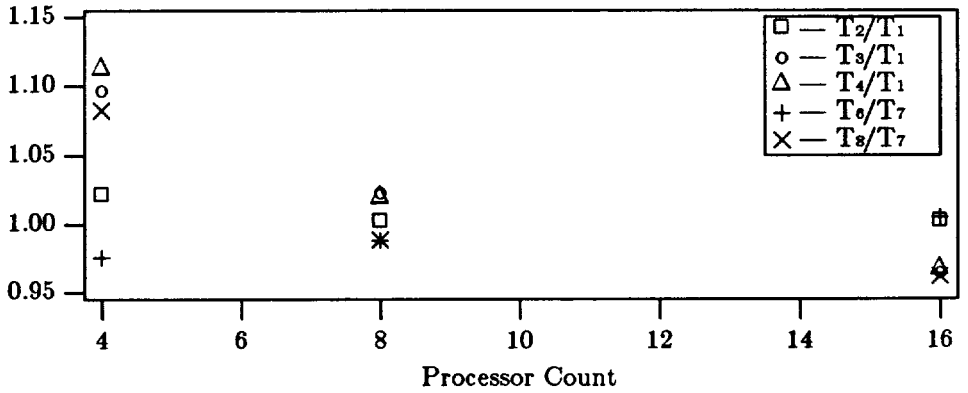


Figure 7.3. — Effects of Task Selection Strategy By Processor Count

strategies used by schedulers #3 and #4, and a minor but noticeable improvement over the strategy used by scheduler #2. This strategy is most successful when a full system simulation is used in processor selection. The strategy used by scheduler #2, which is similar to that of diffusion scheduling, holds a slight advantage when system load is the predominant criterion in processor selection. Processor count also has a clear effect on task selection performance. Interestingly enough, unlike latency the effect becomes less pronounced as the number of available processors is increased.

Although task selection has only a small impact on parallel schedule length, the same is not true of its effect on CPU time. The selection of task selection strategy can have a significant impact on the CPU time required to generate a parallel schedule. This is clearly brought out in Figure 7.4, which shows scheduler #4 using 2½ times as much CPU time as scheduler #1. It is worth noting that while multiple task priority calculation incurs considerable expense in scheduler #4, it does not induce the same expense in scheduler #2.

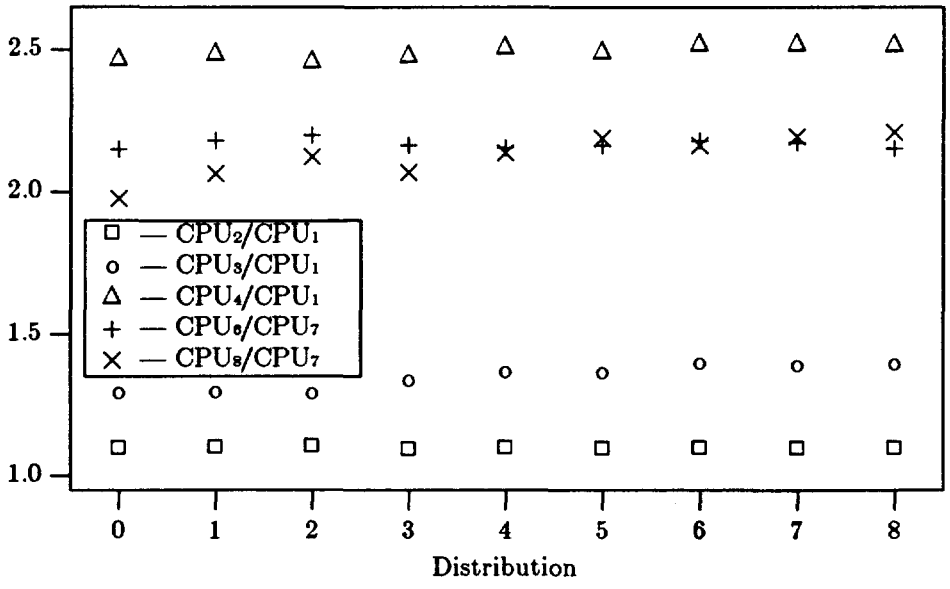


Figure 7.4. — Task Selection Expense By Distribution

So although multiple task priority determination *can* be an expensive option, it is not necessarily so.

The expense of task selection is most affected by program size, latency, and processor count. It is largely unaffected by task distribution and parallelism. Figure 7.5 shows that not only are the reference schedulers faster than the others, but the program size affects *how much* faster they are. It shows that the reference schedulers do better in proportion to the size of the program to be scheduled. According to the complexity analysis from chapter 4, as n increases the slope of the ratio should level off because the dominating term for n has the same power for each scheduler. We conclude that although the complexity analysis provides a prediction for the behavior of each scheduler for very large n, the similarity of the terms and the behavior of the decision algorithms internal to each scheduler preclude any accurate analytical comparison for small n. The best guide is the experimental results reported here and in Appendixes B and C.

Communication also has a significant effect on the relative cost of each scheduler. To understand this aspect of scheduler performance one must understand a little more about the decision algorithm, and in particular, how the communication was scheduled. Each scheduler attempts to minimize total execution time by scheduling each task so that it starts at the earliest possible time. (Schedulers #3, #4, and #8 do this in a round about way, by fixing the termination time and scheduling each task to establish the latest possible start time.) If two tasks must communicate, the cost of the communication will depend on, among other things, the relative placement of the two tasks. When both tasks are on the same processor, the communication is free. When the tasks are separated by heavily used communication links, the communication is expensive.

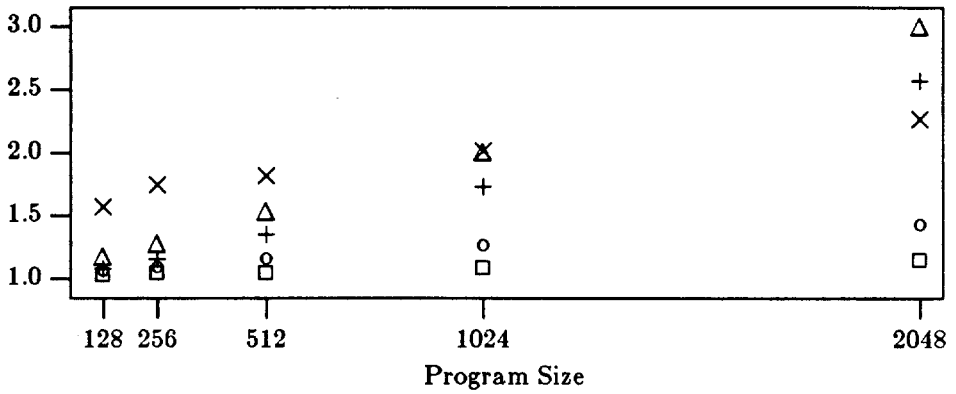


Figure 7.5. — Task Selection Expense By Program Size

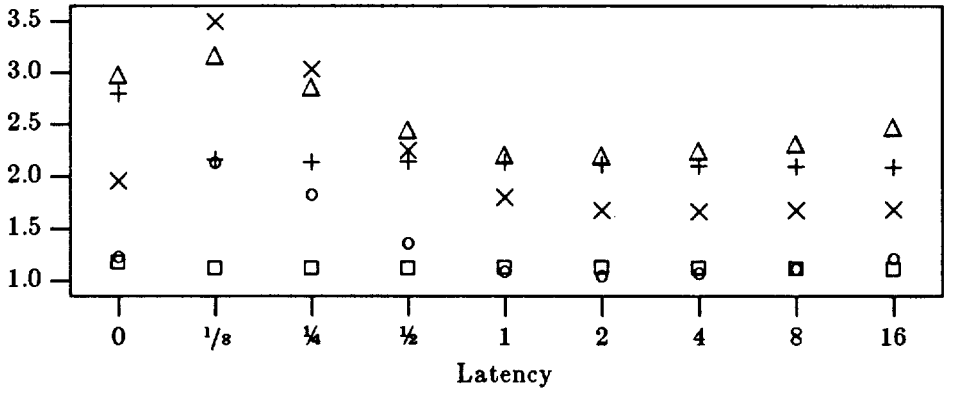


Figure 7.6. — Task Selection Expense By Latency

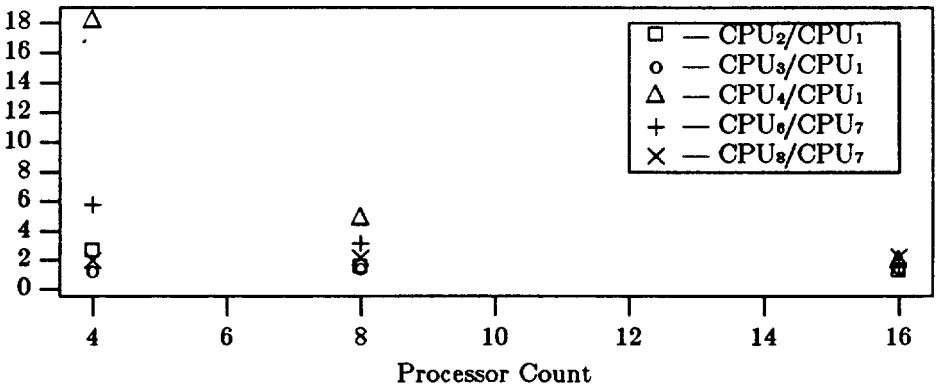


Figure 7.7. — Task Selection Expense By Processor Count

Latency affects this by increasing or decreasing the incentive to take advantage of available parallelism by scheduling communicating tasks on separate processors. Low latency encourages lots of parallelism, and lots of communication. This in turn means the CPU cost of scheduling communication will be high. On the other hand, large latencies discourage the use of parallelism, which causes longer delays from the time a task is available for execution to the time its execution begins. This in turn, because task insertion is used, causes an increase in the CPU time required to schedule a task. It is this combined effect which causes the psychiatrist-couch appearance of the CPU curves for latency.

Processor count has a really dramatic effect on the relative performance of different strategies. Figure 7.7 shows that when there are few processors to be scheduled, task selection can dominate the cost of scheduling by as much as a factor of 18. As the number of processors increases, the costs associated with task selection are overshadowed by those of processor selection and schedule generation. At 16 processors it made little difference in terms of cost what form of task selection was used.

7.2. Processor Selection

Processor selection strategy, unlike task selection, does have a significant effect on parallel schedule length. This effect was most noticeable when the parallelism was low, the program size was small, or the latency was high. The effect varied from different strategies being on par when communication latency was zero, to being faster by a factor of 2.6 when latency was 16.

Several strategies were used to select processors for tasks. The most successful, and the most costly, was employed by schedulers #1 through #4. These schedulers tried each task on each processor, fully scheduling the task and all associated communication for each trial. A second strategy measured only the load of each processor, by noting the finish time

of the latest task, and selecting the processor with the lightest load. The task and all associated communication were then irrevocably scheduled for the selected processor. This strategy was used by schedulers, #6, #7, and #8.

A third strategy, which is not discussed in this section, selected a processor by measuring the processor load and combining it with the required communication time, similar to the first strategy. However, to reduce the expense, the communication was assumed to be over empty channels. Communication contention, therefore, was not considered in processor selection. When a task was added to the selected processor's schedule, it and all its associated communication were properly scheduled as with other strategies. This approach was used by schedulers #10, #11, and #12, and its performance will be discussed further in section 7.3.

As mentioned earlier, the efficacy of one processor selection strategy over another was most influenced by parallelism, program size, and communication latency. As program size or parallelism increase, the advantage of the first processor selection strategy over the second decreases. The effect is reversed for latency. The effect is very pronounced with parallelism. As even moderate values of parallelism are used, the advantage of the first strategy are relatively minor. If the parallelism exceeds 16 the more expensive strategy is less than 1.8 times faster than the much cheaper approach.

The loss of advantage is still apparent, but less pronounced, with increasing program size. To some extent the two effects must reinforce each other, if only because increasing parallelism requires an increase in program size. However, there are sufficient discrepancies in the data to preclude either effect being only a reflection of the other. For example, programs with parallelism 256 all had 2048 tasks. If the two effects were different manifestations of the same effect, the ratio when parallelism reached 256 would be the same

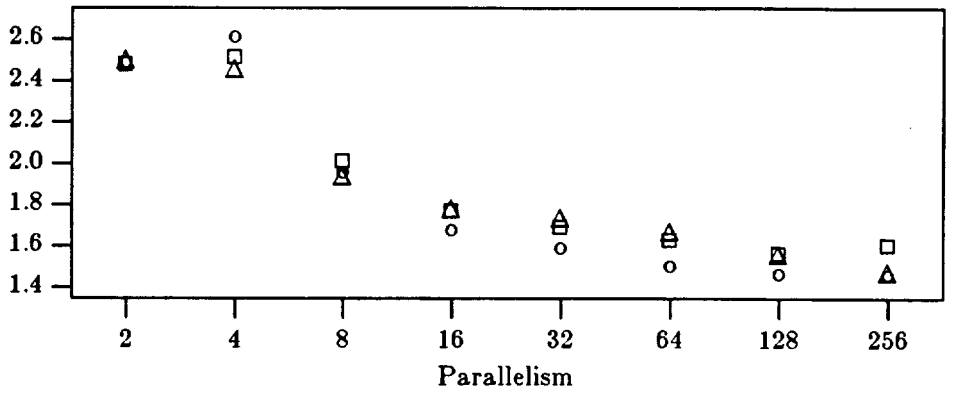


Figure 7.8. — Effects of Processor Selection Strategy By Average Parallelism

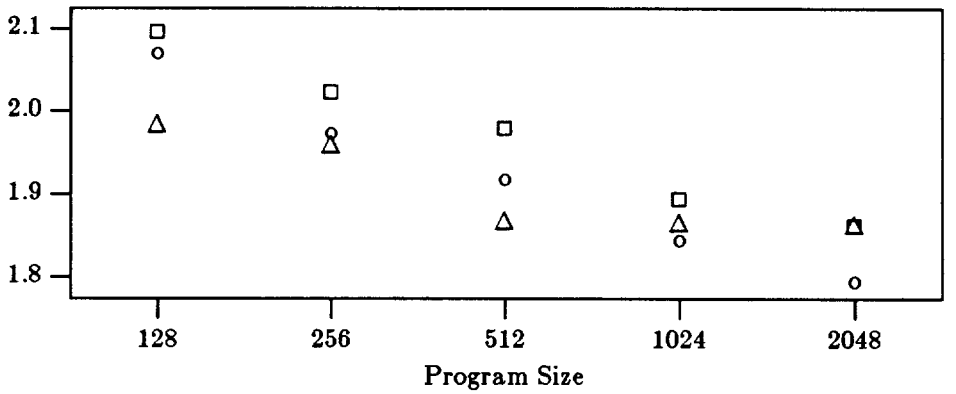


Figure 7.9. — Effects of Processor Selection Strategy By Program Size

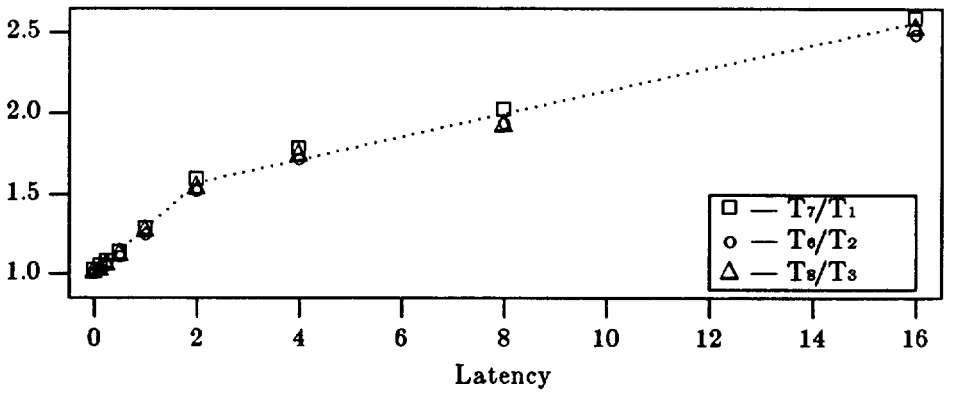


Figure 7.10. — Effects of Processor Selection Strategy By Latency

as when program size reached 2048, and they are not the same.

The first strategy's advantage increases as latency increases. As can be seen from Figure 7.10, the advantages are consistent regardless of other details in the scheduler design. The graph rises sharply between 0 and 2, then rises more gradually for higher latencies. It is interesting to note that Figure 7.10 is divided into two sections, both of which are straight lines. The first section has a slope of ≈ 0.3, the second, ≈ 0.07. The corner separating the two divisions occurs at latency 2, which is the point of inflection of the derivative, i.e, where the third derivative changes sign.

The effect of processor selection strategy on CPU time to generate a parallel schedule is very significant, and mostly independent of variables in the problem space. On the average, scheduler #1 requires about 13 times as much CPU time as scheduler #7, scheduler #2 requires 6½ times as much as scheduler #6, and #3 needs 8½ times as much as scheduler #8. This variance indicates that the amount of additional CPU time a strategy will require depends in part on other details of the scheduler design.

The effect program size has on how much faster the first strategy is than the second is shown in Figure 7.12. CPU_2/CPU_6 is strongly affected by program size, in that scheduler #2 becomes relatively cheaper as the size increases. Although the complexity analysis shows CPU_2/CPU_6 will eventually stabilize, it clearly does not do so in the range covered by these experiments. On the other hand, CPU_1/CPU_7 and CPU_3/CPU_8 appear to have stabilized at 13 and 8, respectively.

Increasing processor count emphasizes the differences in speed. As the processor count increases, both processor selection strategies require more CPU time. Additionally, the $n^2p\ell$ term (processor selection) in the complexity analysis represents a significant portion of the CPU time in schedulers #1, #2, and #3, so a small increase in p will be reflected as a linear

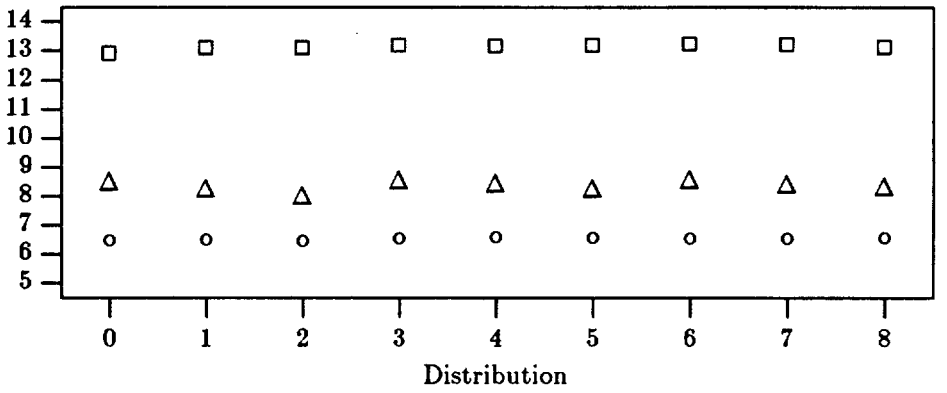


Figure 7.11. — Processor Selection Expense By Distribution

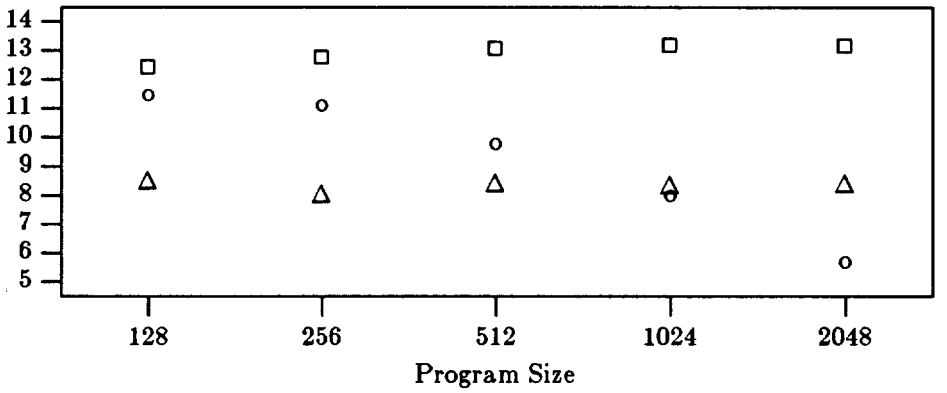


Figure 7.12. — Processor Selection Expense By Program Size

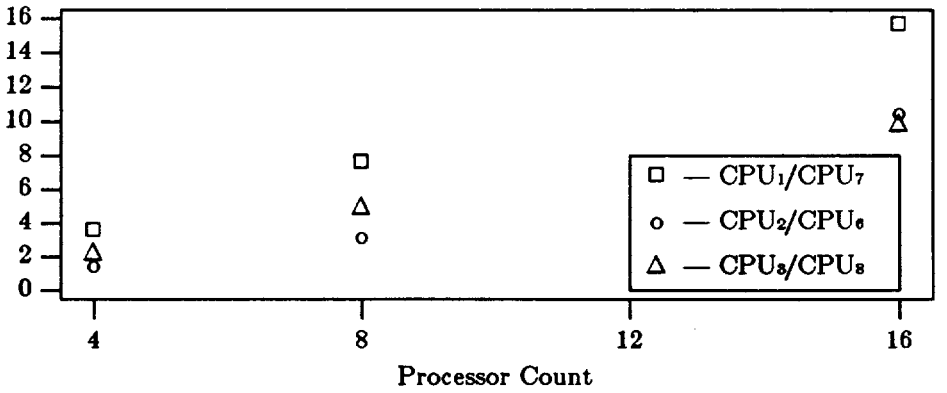


Figure 7.13. — Processor Selection Expense By Processor Count

increase in CPU time. On the other hand, the np (processor selection) term is a very small portion of the total time in schedulers #6, #7, and #8, so a small increase in p will yield a relatively small change in CPU time. The net result is that a small increase in the number of processors (p) gives an almost linear increase in the CPU ratios, as shown in Figure 7.13.

7.3. Processor Selection and Schedule Generation

One processor selection strategy which appeared promising was the third strategy described in the previous section. This strategy combined elements from both of the previous strategies, by taking into account the processor load and the communication time, but ignoring delays from contention with competing messages. It was anticipated that its CPU requirements would be only slightly higher than merely accounting for processor load, and that it would give some improvement in parallel schedule lengths. What happened was that because of several minor optimizations within inner loops, the CPU requirements were slightly lower. The minor increase expected was more than offset by improvements in the implementation. The parallel schedule lengths, however, were very much worse — not better — than the other strategies. It sometimes generated schedules that were 15 times as long as other parallel schedules, and nearly 40 times as long as the corresponding sequential schedules.

In an attempt to discover the precise combination of elements which caused this unexpected behavior, several additional schedulers were constructed. Scheduler #10 used the critical path scheduling task selection strategy, so scheduler #11 was designed with the diffusion scheduling task selection strategy. The result was not significantly different.

It was also conjectured that task insertion in the schedule generation phase was causing the problem, so scheduler #12 was constructed as a copy of scheduler #10 with the task insertion removed. This means the schedule generator placed each new task at the top

of the schedule rather than search the schedule for a better slot. Scheduler #12 performed about 30% worse than scheduler #10, which shows that task insertion was not the reason for the poor performance.

Scheduler #5 is also very similar in design to scheduler #10. Both use task insertion in the schedule generation, both use load and latency but not contention to select processors. The difference is that scheduler #5 records only processor schedules, where scheduler #10 records both processor and communication link schedules. Scheduler #10 records and schedules around contention where scheduler #5 pretends contention does not exist.

Strangely enough, scheduler #5 performed significantly *better* than scheduler #10, and cost much less because it did not schedule communication. Since scheduler #5 did not have the same performance as scheduler #10, the key elements to #10's performance problems were the use of latency without contention in processor selection, while using message contention in schedule generation.

7.4. Other Comparisons

Considering the parallel schedule lengths over all tests, the schedulers naturally formed three groups. The first group is schedulers #1, #2, #3, and #4. The second group is schedulers #10, #11, and #12. The third group is schedulers #5, #6, #7, #8, and #9. The division is quite easily seen in Figure 7.14 (cf. Figure 6.14.). Figure 7.14 is a frequency histogram of the schedule lengths, with the the horizontal axis representing the fraction of schedules that had the given y value.

The first group of schedulers performed significantly better than the other two, generating schedules which were shorter on the average by a factor of 1.8 or more. The principal characteristic that identifies this group is that each scheduler modeled contention

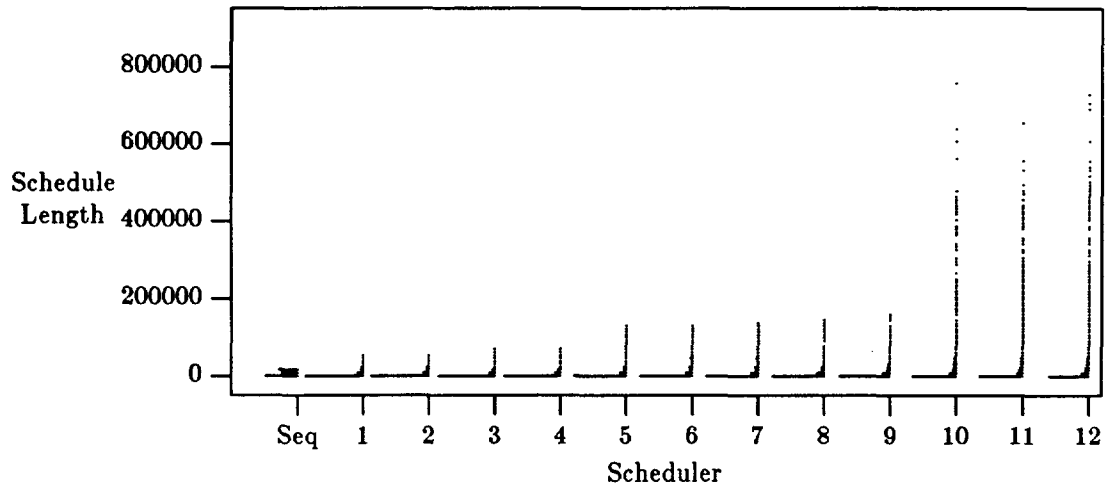


Figure 7.14. — All Tests (6075 Cases)

in both processor selection and in schedule generation. This group included all schedulers which did both of these things. It included no scheduler that did not do both.

The second group consists of schedulers that modeled latency in the processor selection, but modeled contention in the schedule generation. One practical example of this type of scheduler is a diffusion type load balancing strategy which attempts to account for communication in its task distribution. Performance of this group was the worst of the three because of an anti-synergistic effect.

The third group consists of all remaining schedulers. It includes, among others, random scheduling (scheduler #9) and load balancing (scheduler #6). It also includes Kruatrachue's ISH scheduler (scheduler #5), which forms the platform from which DSH is constructed [Kru87]. The schedule lengths of this group are substantially shorter than those of the second group, though not as good as those of the first. It is noteworthy that this group contains schedulers which work much more quickly than group one, sometimes by more than two orders of magnitude. Figure 7.15 shows a histogram of the CPU times for

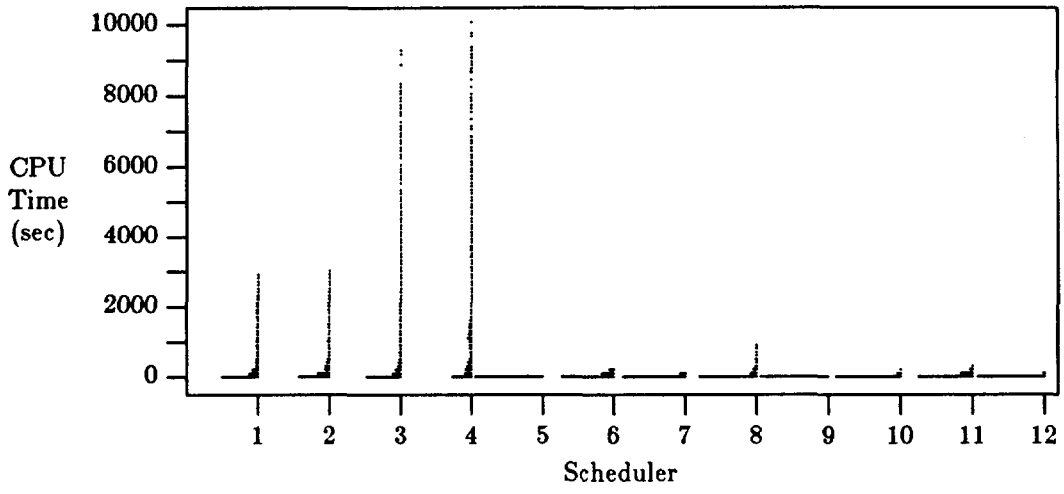


Figure 7.15. — All Tests (6075 Tests)

each of the 12 schedulers. Scheduler #5 (ISH) and scheduler #9 (random processor selection) used the least CPU time of those tested, primarily because neither incurs the expense of scheduling program communication.

Performance was also measured by the frequency with which a scheduler chose the best schedule. This was measured in two ways — by comparing all of the schedulers together, and by deriving an ordering through comparing each pair of schedulers. Figure 7.16 shows the results of comparing all schedulers together. It was obtained by pooling all of the scheduler results together and counting the number of test cases for which each scheduler created the best parallel schedule.

In addition to the group comparison, the schedulers were also compared pairwise. Every pair of schedulers was compared, experiment by experiment, for the number of shortest parallel schedules. Each scheduler was compared individually against the 11 other schedulers. When a scheduler had more shorter schedules than its opponent, it was given a point. The scheduler which had the largest number of points was given the rank of 1.

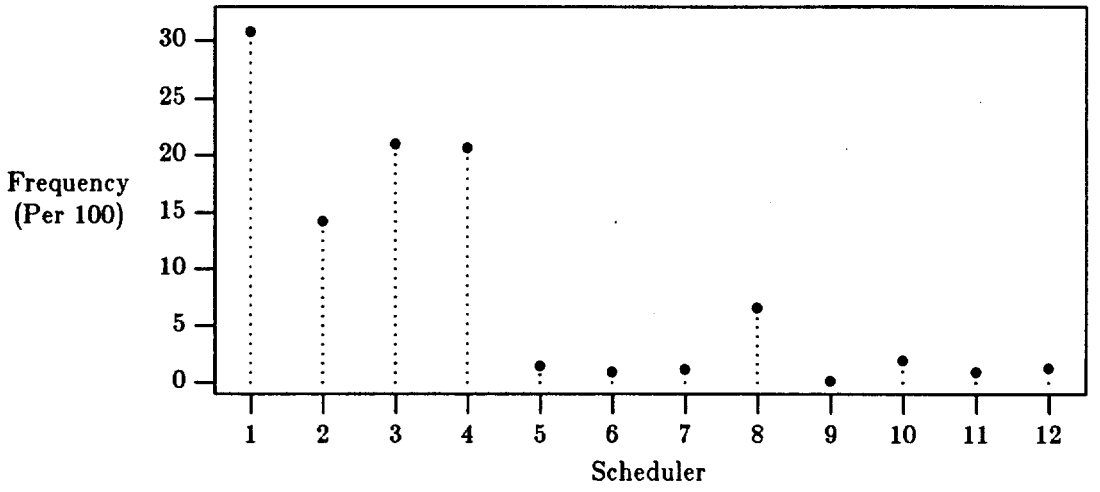


Figure 7.16. — Relative Frequency By Group Comparison

(Scheduler #1 had the most points at 11.) The scheduler with the next highest number of points was given the rank of 2, and so on. The results are given in Table 7.1. The ranks reflect the order in which schedulers found the shortest schedules most frequently, i.e., scheduler #1 is ranked number 1 because it found the shortest schedule more often than any other scheduler.

It is interesting to note that each scheduler generated the shortest parallel schedule for *some* test case, and that there was only a weak correlation between the average parallel schedule length, the average number of shortest schedules, and the pairwise ranking of all

Schedulers Ranked By Pairwise Comparison					
Scheduler	Rank	Scheduler	Rank	Scheduler	Rank
#1	1	#5	5	#9	10
#2	3	#6	8	#10	11
#3	4	#7	6	#11	9
#4	2	#8	7	#12	12

Table 7.1.

schedulers.

A third comparison was also done. For each experiment the best schedule was found; this included the 12 parallel schedules and a sequential schedule. Each schedule was then compared against the best schedule, by dividing the length of the schedule by the length of the shortest schedule for that experiment. A cumulative histogram was then constructed for each scheduler, which shows the number of schedules that are better than a certain factor of performance. The results are given in Appendix E. The histograms of all experiments taken together are also reproduced in Figure 7.17.

Figure 7.17 supports a number of claims made elsewhere in this dissertation. From it one may clearly see that scheduler #1 indeed is the better scheduler, and that #2, #3, and #4 have very similar performance. One may also distinguish between the three scheduler groups. It shows not only how the groups are different, but also how individual schedulers within each group differ.

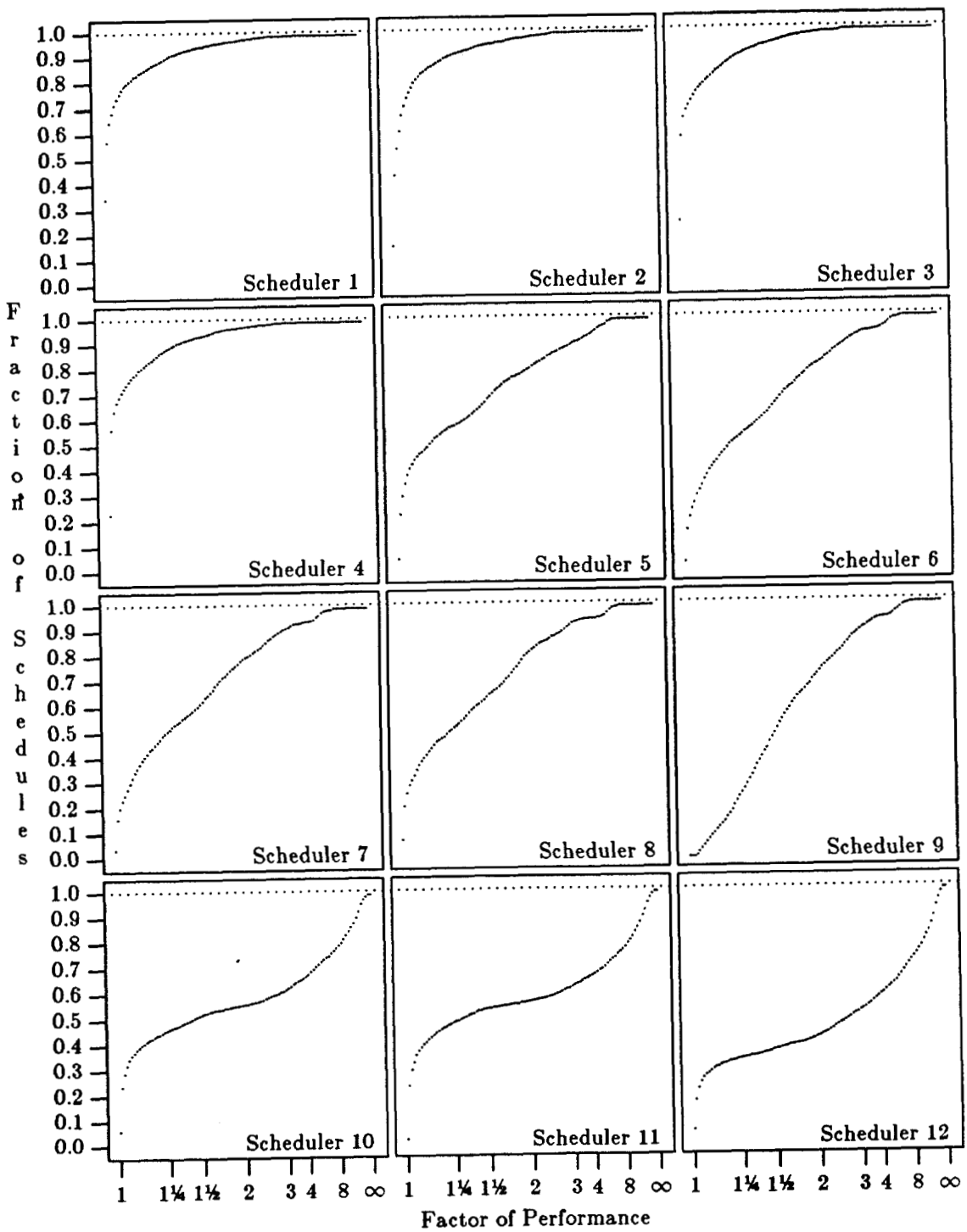


Figure 7.17. — Cumulative Histogram of Performance (All Tests)

CHAPTER 8

Conclusions

8.1. Scheduler Phase Effects

Static scheduling strategies can be divided into three phases — task selection, processor selection, and schedule generation. Task selection decides the order in which tasks will be scheduled. Processor selection chooses a processor which will yield the best overall schedule for the selected task. Schedule generation records the the selections and system resource usages such selections require.

Of the three phases, task selection affects the schedule length the least. Schedulers #1, #2, #3, and #4 were identical in every way except task selection strategy, as were schedulers #6, #7, and #8. The largest average difference in the first group was between schedulers #1 and #4, the difference being that schedules from #4 were 5.2% longer than those from #1. The average over each task distribution varied from about 13% in favor of scheduler #4, to 15% in favor of scheduler #1. Average differences over other variables (parallelism, size, latency, and processor count) yielded smaller differences. A 4.8% difference was measured between schedulers #6 and #8 over all tests; the variation over each variable was not much larger.

Although task selection had the least impact on the schedule length, it still had substantial impact on the CPU time required to generate a schedule. Measuring task priority from the top and scheduling from the bottom required more CPU time than measuring from the bottom and scheduling from the top. Scheduler #3 used 35% more CPU time than scheduler #1, and scheduler #8 used more than twice the CPU time of

scheduler #7. Re-measuring the priority each time a new task was selected increased the cost of scheduler #4 by 80% over scheduler #3. Each phase's direct contribution to the total CPU cost was not measured, but schedulers #1, #3, #4, #7, and #8 used nearly identical algorithms to select tasks. The only difference was in the *direction* of the priority measurement and task selection. This indicates that most of the cost was *indirect*, that is, that certain task selection strategies caused other phases to do more work.

Processor selection proved to be the phase which affected schedule length most profoundly. Schedulers #1, #7, and #10 were identical except for their processor selection phase. Scheduler #1 modeled processor load, latency, and contention, scheduler #7 considered only processor load, and #10 considered both load and latency. The difference in average schedule length was overwhelmingly in favor of scheduler #1. Scheduler #10 produced schedules which were 6.75 times as long as those produced by scheduler #1. Schedules from #7 were about 1.9 times as long as those from #1.

Latency brought out the difference in performance most clearly. When the latency was low (less than 1), the performance was nearly identical for the three schedulers. When latency was very high (e.g. 16) schedules from #10 were more than 10 times as long as those from #1. Scheduler #7 behaved similarly, generating schedules about 2.5 times as long as those from #1 when latency was high.

While processor selection was the most important phase for generating short schedules, it also affected the *cost* of generating schedules the most. Scheduler #1 required more than 15 times as much CPU time as scheduler #7 or #10.

The architecture model used by the schedule generator also affected the CPU time. Schedulers #5 and #10 were identical with the exception that the schedule generator in #5 modeled processor load and message latency, but not message contention. Scheduler #10

modeled load, latency, and contention. Scheduler #10 required about 6.4 times as much CPU time as scheduler #5. This factor did vary somewhat with communication latency because smaller latency encourages heavier usage of the communication system, which in turn requires more information to be recorded by the schedule generator. (Zero latency is a special case — it implies any message can get through at any time, so there is no need to record message transmissions.)

As an aside, schedule generation records information that is used by other phases. Information that is not recorded presumably is not available. Thus it does not make much sense for the other phases (particularly processor selection) to attempt to model the architecture in more detail than what is recorded by the schedule generator. On the other hand, information that is not used directly by other phases can still affect their behavior — scheduler #10 is an example of this. This scheduler used only communication latency in its processor selection phase, but it modeled contention in its schedule generator. Scheduler #5 modeled latency in both processor selection and schedule generation. Scheduler #5's schedules were 4.9 times faster than those of #10.

This latter case is also an example of pathological interactions that can occur between phases. It is clear from observing the behavior of schedulers #10, #11, and #12 that certain combinations of scheduler phase behaviors can cause unexpected results. In this case, #10, #11, and #12 all have the common characteristic that they model latency in processor selection, and contention in the schedule generation. As each task was assigned to a processor, only the latency was considered — the fact that the communication system was heavily overloaded was ignored. Thus moving one more task to another processor only served to load the communication system further, which in turn caused greater delays due to contention. It is not fully understood why this effect occurs when processor selection models latency, but it does not occur when only processor load is considered.

8.2. Scheduler Families

Because of the characteristics of the phases, as outlined in the previous section, the schedulers tested here naturally fall into three families. The first two families have very specific phase designs which identify them and are responsible for how they behave. The third family is less easily defined, except that each scheduler generates schedules of about the same length as every other scheduler in the family.

The first family consists of schedulers #1, #2, #3, and #4. In this family each scheduler modeled processor load, message latency, and contention in both processor selection and schedule generation phases. This family consistently generated the shortest schedules, especially when the latency was high (e.g., greater than 1). It also required significantly greater CPU time to generate the schedule than any other family.

The second family of schedulers consists of schedulers #10, #11, and #12. Family 2 schedulers modeled processor load and message latency in the processor selection phase, but modeled load, latency, and contention in the schedule generation phase. This family required about 1/6th the CPU time of family 1, and generated schedules that were more than 6 times as long. Schedule length worsened as latency increased — for the highest latency tested, average schedule length was 10 times that of family 1. The poor performance of schedulers in this family is not because task insertion is used. Inserting tasks into the middle of a schedule may increase contention by placing tasks where a lot of communication is already taking place, but removing the task insertion only degrades performance further (compare schedulers #10 and #12).

Family 3 consists of schedulers #5, #6, #7, #8, and #9. Scheduler #5 is Kruatrachue's ISH scheduler [KrL87,Kru87]. Schedulers #6, #7, and #8 are identical to schedulers #2, #1, and #3, respectively, with the exception that the processor selection

phase only considers the processor load. Scheduler #9 selects the processor at random. Even though the design of each scheduler varies substantially within this family, their overall performance is very similar. This is because there are enough details in the program execution that the scheduler does not explicitly account for, that *the processor assignment is effectively random.* This explains particularly why the performance of each scheduler is so similar to random processor assignment.

It is worth noting that *every* scheduler, regardless of its family, generated some schedules that were longer than a sequential schedule. This fact offers a possibility for an additional performance improvement. Sequential schedule lengths are easily determined (by summing the weights of the tasks). If the scheduler models the architecture sufficiently well that the length of the parallel schedule accurately represents the program's execution time, compare the parallel and sequential schedule lengths, and select whichever is shorter. This has the benefit of guaranteeing that no program will take longer than it would on a single processor machine. This improvement was effective for family 1 schedulers when the latency was above 4, and for family 2 schedulers when the latency was above 2.

This improvement may also add a substantial cost to the scheduler CPU time. In schedulers #5 and #9, where the architecture is not accurately modeled, the cost of scheduling a program may be multiplied many fold. These two schedulers, appropriately modified, would require approximately the same CPU time as scheduler #7.

8.3. Effects of Problem Characteristics

In this work we chose to consider five variables in the problem space, namely, task distribution, average parallelism, program size, communication latency, and processor count. Of those, task distribution had almost no effect on any aspect of the scheduling problem. In contrast, average parallelism had a fairly substantial impact on performance, generally

shortening parallel schedules as parallelism increased. Schedule length for schedulers #1 through #9 improved substantially. However, relative to the random scheduler all scheduler performance declined. So although each scheduler improved, in an important sense it was because the problem was easier.

Program size had a minimal, if consistently positive, effect on scheduler performance. Small improvements in schedule speedup were measured for each increase in program size.

As was expected, increased latency had a universally negative effect on schedule lengths. What was not expected, however, was that none of the schedulers could handle high latency very well. Each scheduler had a point above which it selected parallel schedules which were worse than the equivalent sequential schedules. The best schedulers were able to do better with high latency than were the other schedulers, but even the best scheduler had problems with some programs.

Increasing processor count, like increasing parallelism, improved the average parallel schedule length. It would be desirable, but unrealistic, to hope for a linear increase in speedup. However, a new measure of performance which we call the *relative efficiency* shows that the best schedulers were doing better than would be indicated by the parallel efficiency alone.

8.4. Implications For Dynamic Scheduling

Schedulers #2, #6, and #11 serve a special purpose in this work — they all emulate diffusion scheduling. Each represents an idealized form of this dynamic load balancing strategy. They are ideal in the sense that they do not suffer from two problems inherent to diffusion scheduling, namely, runtime scheduling overhead and incomplete knowledge of the system state.

Scheduler #6 is most similar to the common approach to diffusion scheduling [LiK87]. Scheduler #11 attempts to be only slightly more intelligent about its processor selection by adding communication latency into its calculation. Scheduler #1 tries to be very intelligent about processor selection, by considering communication link loading as well. A dynamic implementation of this last algorithm would require special hardware to provide accurate measures of link loading. Such hardware could not use the standard communication links, or reporting the load would alter it, causing the measure to be unreliable.

Because these algorithms closely match dynamic diffusion schedulers without some of their problems, it is unlikely that diffusion schedulers will perform better than either scheduler #6 or #1.

8.5. Recommendations

In systems where high performance is most desirable, schedulers must accurately model the communication system. A strategy which generates parallel schedules using scheduler #1 but selects a sequential schedule when it is shorter gives the overall best performance. However, this strategy can cost 100× more than a scheduler such as #5 (ISH). If the latency is guaranteed to be less than 2, scheduler #5 will yield excellent results at very low cost. If latency may be high, scheduler #7 with a comparison against a sequential schedule is a moderately low cost, high performance alternative.

CHAPTER 9

Future Work

9.1. Other Network Topologies

This dissertation has only considered the effect of a single topology — a completely connected network — on scheduler performance. Such networks are ideal in the sense that the diameter and average diameter are both minimal, namely 1. The *contention value*, which is the proportion of network resources an average message will use [Pas88], is also minimized, namely $1/n$ for n processors. Although it was appropriate to use the topology in this simulation study, completely connected networks are generally too expensive for use in real systems. The obvious reason is that the number of connections grows as the square of the number of processors in the network. This implies that even with a small number of processors (say, 16) the cost of the whole system is dominated by the cost of the interconnection network.

A number of topology families exist which have low cost and relatively high performance. (For a general survey see [Fen81] and Chapter 5 of [HwB84]. Other important articles on this topic include [AkK84,Dot84,PrV81,Von83].) Many of these topologies grow as a factor of $N\log N$ or better, where N is the number of processors. Even though the number of connections grows slowly — $N\log N$ is much less than N^2 for large N — both the diameter and average diameter are $\log N$ or better. For example, a binary n-cube (hypercube) has $n2^n$ connections for 2^n nodes. Its diameter is n, and its average diameter is $n/2$.

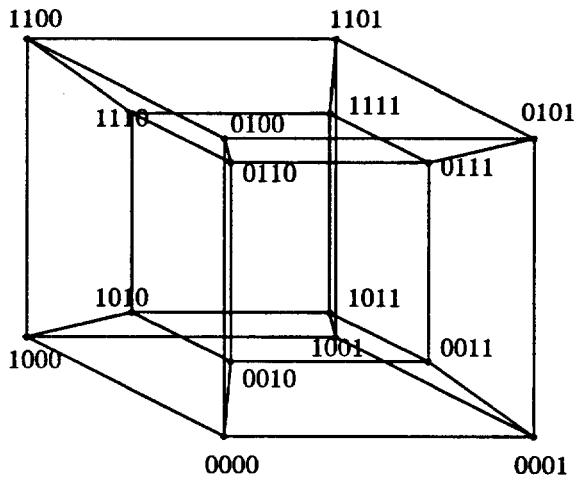


Figure 9.1. — D/4 Hypercube

If at any given instant every node is equally likely to send a message, the average resource usage for the network will be $\frac{rDN}{c\ell}$ where r is the rate of message transmission, D is the average distance a message must travel, N is the number of nodes, c is the capacity of a single network link, and ℓ is the number of links in the network. If the network is *regular* (i.e. the degree is the same for every node), then $\ell = Nd/2$, where d is the degree of the network. Substituting into the previous expression gives us $\frac{2rD}{cd}$. If every node is equally likely to be the recipient of a message, the average distance a message must travel is simply the average diameter. For the hypercube this equation reduces to $\frac{r}{c}$. This may be contrasted against the same value for a completely connected network, $\frac{2}{N}\frac{r}{c}$.

As was noted in previous chapters, scheduler performance is driven in some cases by the scheduler's ability to handle contention. Because different networks offer different abilities to deal with heavy message traffic, the network topology may affect scheduler performance as well. Message switching technology could also have an impact, because it also affects the performance of the network (see [DaS87,HRW85,MTH78]). Much work

remains to be done to measure the effect of more "realistic" network designs on scheduler performance. It would be especially interesting to compare the performance of star-graphs [AkK87] against that of hypercubes under automatic scheduling systems. It would also be valuable to examine the effects of packet and circuit switching on the system as well, particularly because static schedulers seem to be much more sensitive to contention than they are to latency.

9.2. Scheduling Loops and Conditionals

Static scheduling of acyclic task graphs by itself is not as generally useful as one might hope. Requiring task graphs to be acyclic is a fairly severe restriction — the vast majority of programs are expressed using some form of loop, and it is in those loops where most of the parallelism may be found. Special provisions must also be made for conditional expressions, which cannot be scheduled directly by these algorithms either.

Properly structured loops have a single entry and a single exit point. This characteristic can be exploited by scheduling the loop body as if it were acyclic (Figure 9.2). The loop execution time for this type of scheduling will be the number of loop iterations times the loop body schedule length. Thus a short schedule for the loop body will provide a short (but not necessarily optimal) execution for the whole loop.

Further improvements may be realized by unrolling the loop some number of iterations and scheduling the combined iterations. The difficulty with this solution is in guaranteeing

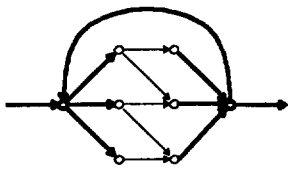


Figure 9.2. — Simple Loop

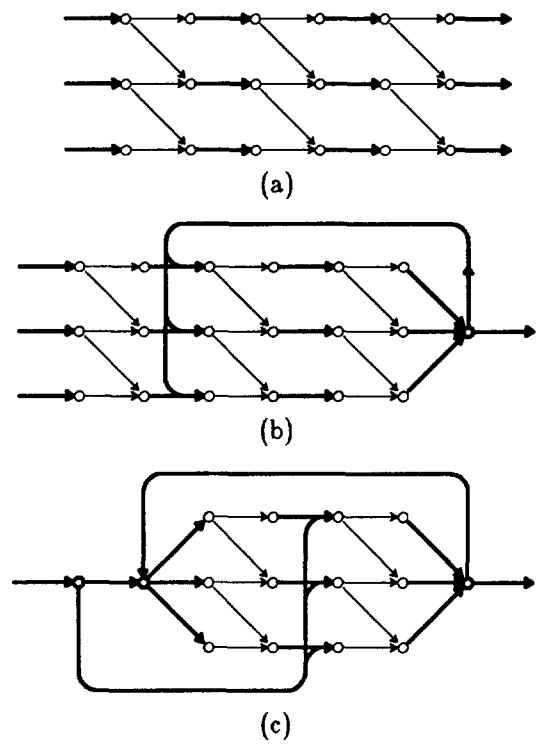


Figure 9.3. — Unrolled Loops

each part of the loop executes the correct number of times. If the number of loop iterations (N) is known when the schedule is generated, that number of iterations may be unrolled, giving an acyclic graph to schedule (Figure 9.3a). If the loop is too large to unroll completely, then some "reasonable" number of iterations (i) might be chosen. The schedule would then consist of (N mod i)+N unrolled iterations — N iterations inside the loop and N mod i iterations either before or after the loop (Figure 9.3b).

If the number of loop iterations is not known when the schedule is generated but is known before the loop begins execution, some loop unrolling can still take place (Figure 9.3c). The idea here is that the initial entry point of the loop causes execution to begin in such a way that when the final iteration of the unrolled loop is accomplished, the original loop has executed the correct number of iterations.

Further work in this area would allow researchers to use large graphs from real programs in comparing alternative systems. Most importantly, it could ultimately give compilers the ability to automatically schedule programs for efficient parallel execution in ways that humans are unable to do.

APPENDIX A

Task Density Functions

This appendix contains graphs which represent the density of tasks available for execution with respect to the critical path of a program. The x-axis represents the progression of time, the y-axis represents the the relative number of tasks whose earliest schedulable time (EST) corresponds with the x-axis value. The area under each curve has been normalized to 1, i.e., $\int_0^1 tdf(t)\,dt = 1$. The actual task distribution for a given program may be obtained by multiplying the function by the program's average parallelism (i.e. $\frac{total\ task\ weight}{critical\ path\ weight}$). Important values for each density function is given in Table A.1. The start time is fixed at time 0, finish time is fixed at time 1.

$$tdf = \frac{e^{-\left(\frac{a}{t}+\frac{b}{(t-1)}\right)^2}}{area}$$

Distribution	a	b	Area	Mean	Variance
0	0.10	0.10	0.7470	0.5000	0.0509
1	0.10	0.50	0.4226	0.3164	0.0259
2	0.10	1.00	0.1969	0.1877	0.0114
3	0.50	0.10	0.4226	0.6836	0.0260
4	0.50	0.50	0.3551	0.5000	0.0155
5	0.50	1.00	0.2432	0.3641	0.0086
6	1.00	0.10	0.1970	0.8121	0.0116
7	1.00	0.50	0.2432	0.6358	0.0086
8	1.00	1.00	0.2052	0.4999	0.0060

Table A.1. — Task Distribution Function Parameters

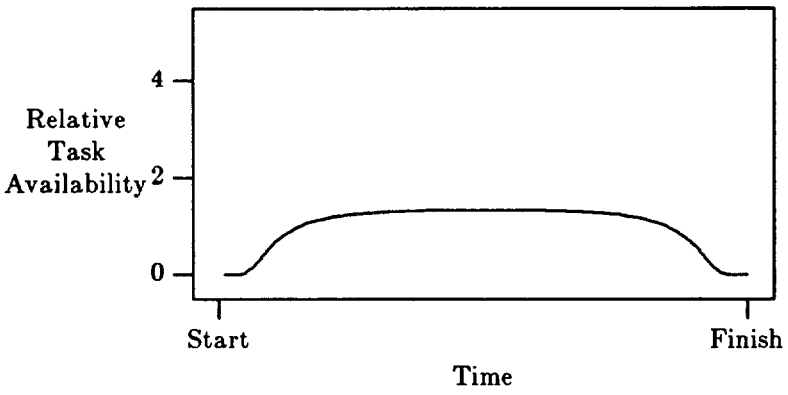


Figure A.1. — Distribution 0.

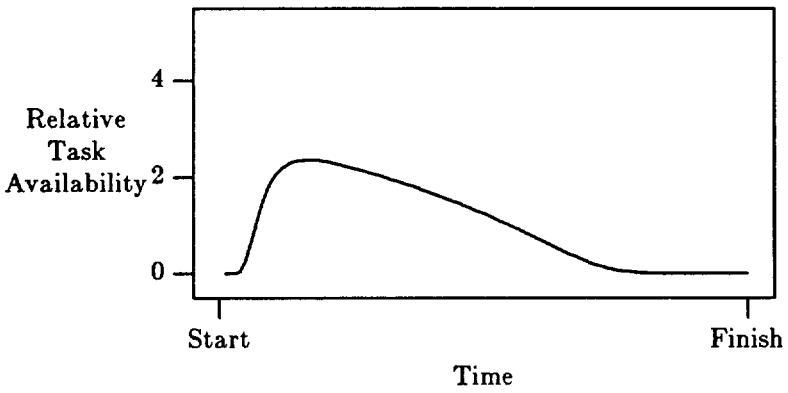


Figure A.2. — Distribution 1.

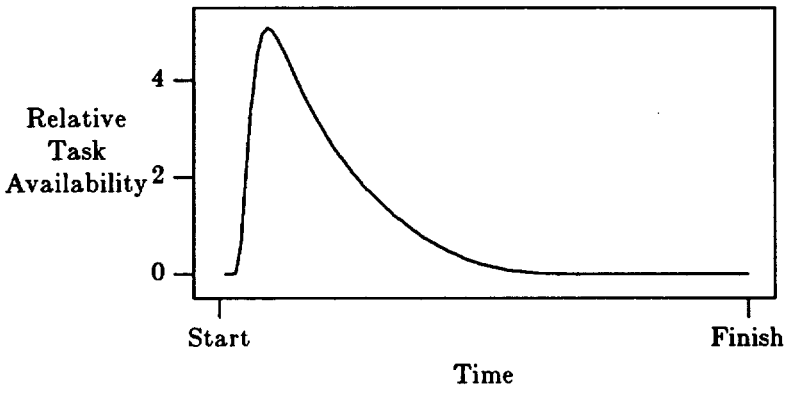


Figure A.3. — Distribution 2.

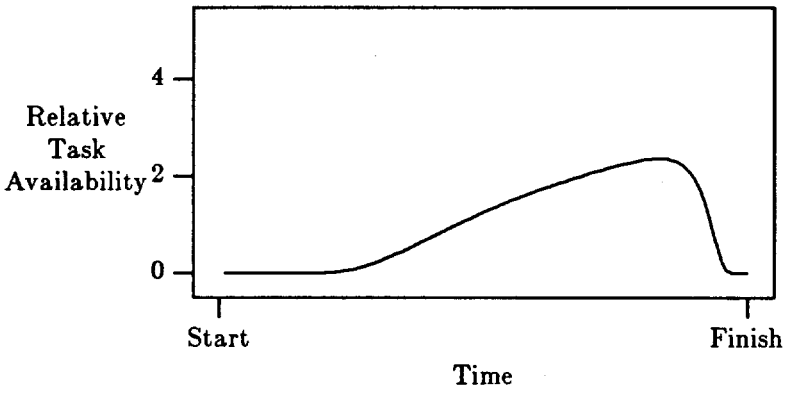


Figure A.4. — Distribution 3.

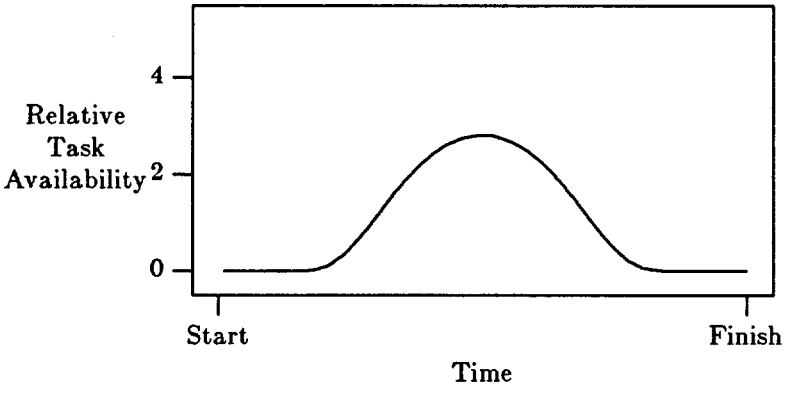


Figure A.5. — Distribution 4.

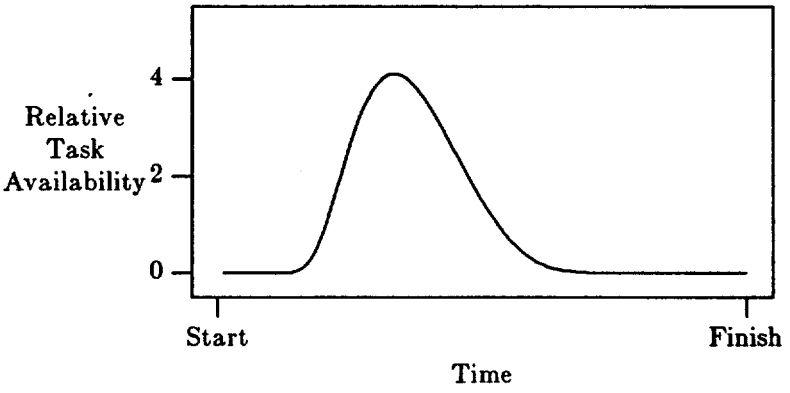


Figure A.6. — Distribution 5.

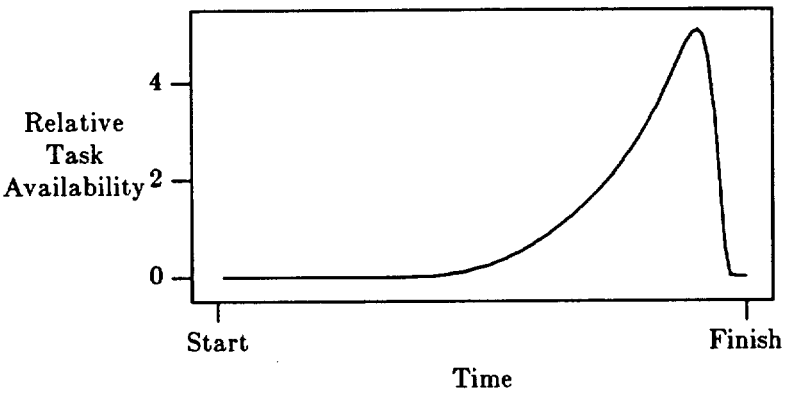


Figure A.7. — Distribution 6.

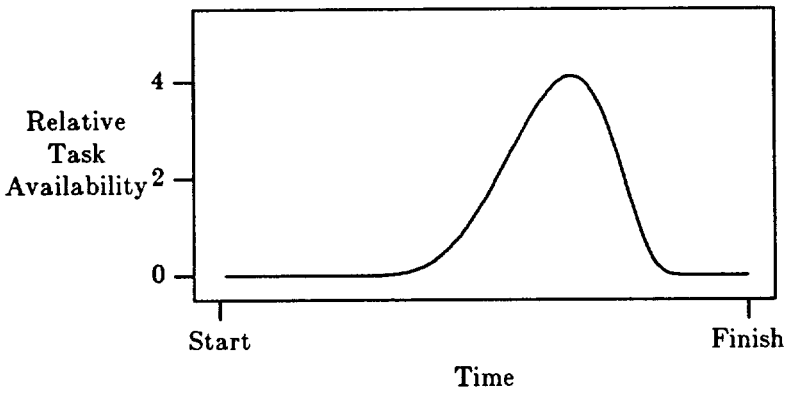


Figure A.8. — Distribution 7.

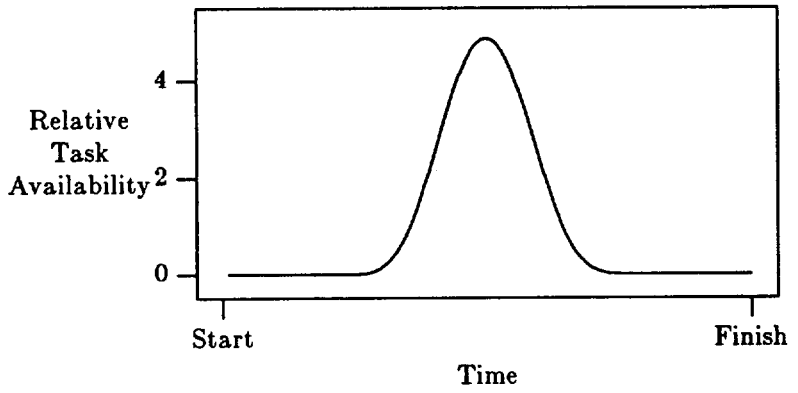


Figure A.9. — Distribution 8.

APPENDIX B

Scheduler Performance Characteristics

In this appendix the performance characteristics for each scheduler are displayed. The problem space is divided by task distribution, average parallelism, program size, communication latency, and processor count. It shows how changing a given characteristic, such as program size, can affect the performance of a given scheduler. Each scheduler/characteristic pair uses a histogram chart, a bar chart, and a table. The histogram chart shows histograms side-by-side, to compare the distributions of schedule lengths for a given scheduler. The bar chart shows the average sequential schedule length (dashed line), the average length of the parallel schedules, and the average length of min(*sequential schedule, parallel schedule*), which is referred to as the *corrected schedule length*. This third item recognizes that the parallel schedules generated by the different schedulers are not always shorter than a sequential schedule, and shows the effect of selecting the shorter of the two.

The table gives specific values of interest in a numerical form. The values are:

%P≤S Percentage of parallel schedules that were shorter than a sequential schedule.

S/P The speedup gained by the parallel schedule, or $\frac{T_s}{T_p}$, where T_s is the length of a sequential schedule and T_p is the length of the parallel schedule.

S/C The speedup gained by the corrected schedule, or $\frac{T_s}{T_c}$, where T_c is the length of the corrected schedule.

P/C The speedup gained by correcting the parallel schedule for schedules which are longer than a sequential schedule, or $\frac{T_p}{T_c}$.

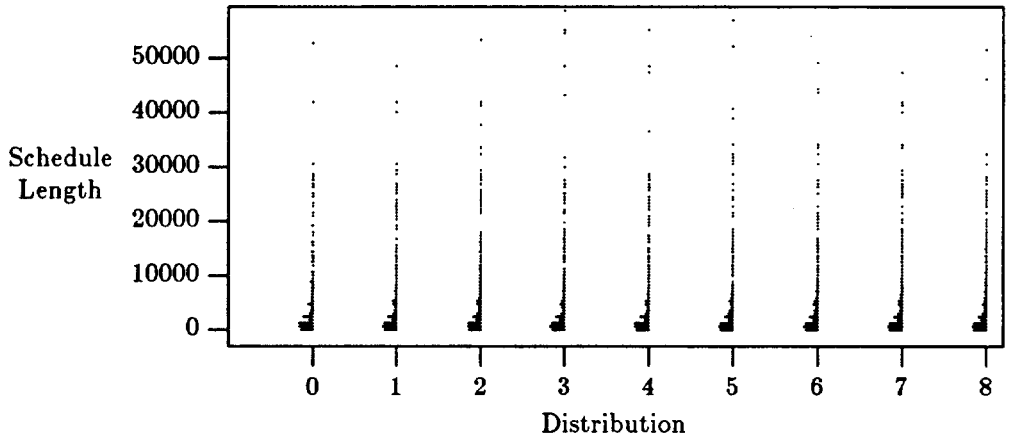
P Eff This entry gives the *parallel efficiency* of a schedule, defined as $\frac{T_s}{n \times T_p}$, where n is the number of available processors.

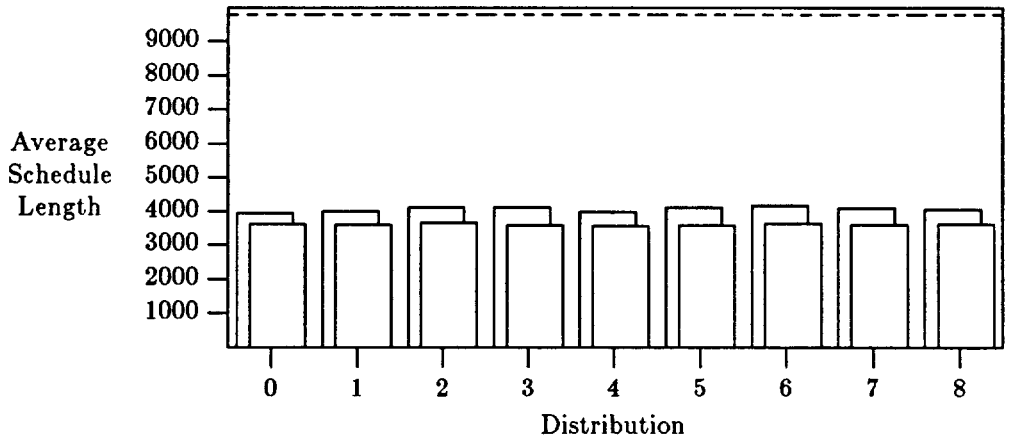
C Eff This entry gives the *corrected parallel efficiency* of a schedule, defined as $\frac{T_s}{n \times T_c}$, where n is the number of available processors.

CPU Sec This field gives the average number of CPU seconds on a Sequent Symmetry™ used to schedule the programs.

B.1. Scheduler Performance By Task Distribution

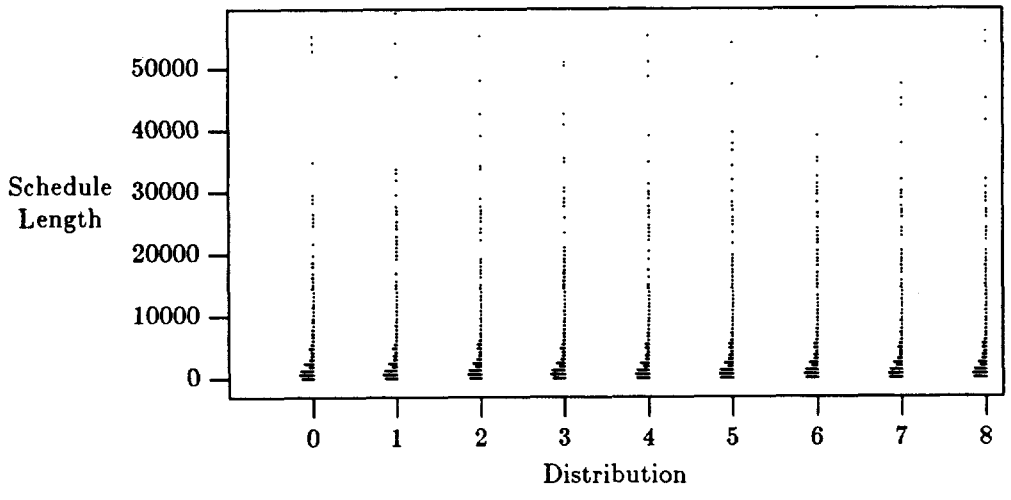
B.1.1. Figure B.1. — Scheduler 1

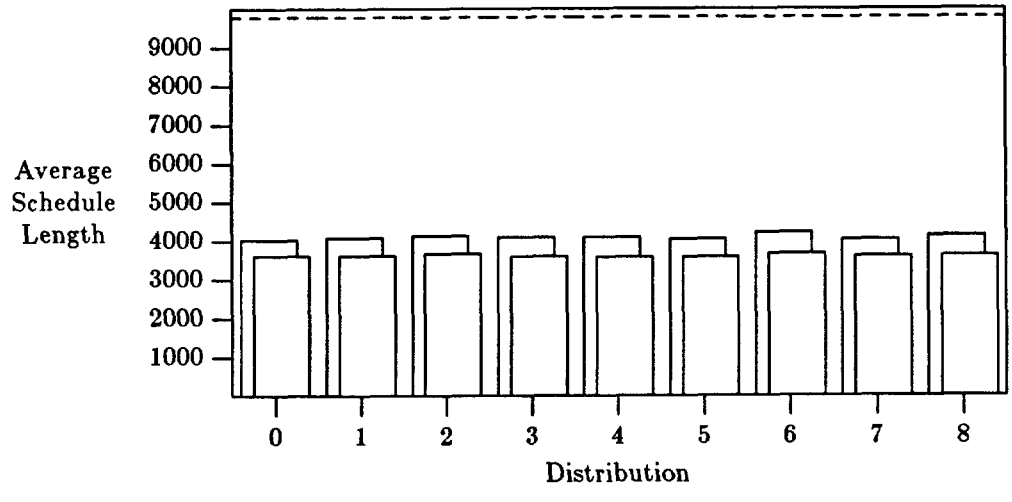




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	91.26	89.04	88.59	90.22	89.19	88.30	87.85	88.59	88.59
S/P	2.49	2.46	2.38	2.38	2.46	2.38	2.35	2.39	2.42
S/C	2.71	2.72	2.68	2.73	2.75	2.74	2.69	2.72	2.71
P/C	1.09	1.11	1.13	1.15	1.12	1.15	1.15	1.14	1.12
P Eff	0.57	0.55	0.53	0.55	0.55	0.54	0.52	0.53	0.53
C Eff	0.58	0.56	0.53	0.56	0.56	0.54	0.53	0.54	0.54
CPU Sec	324.47	329.20	336.16	345.11	349.93	357.54	363.99	362.10	368.47

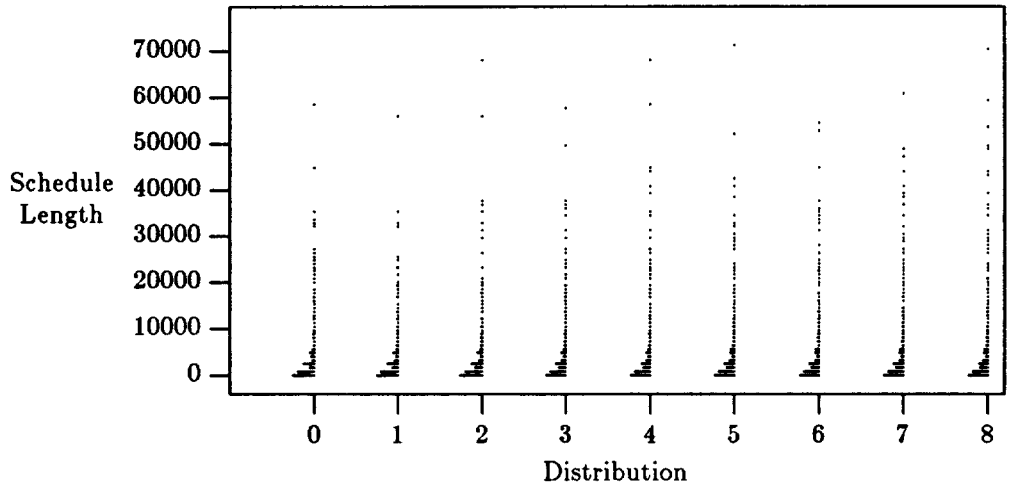
B.1.2. Figure B.2. — Scheduler 2

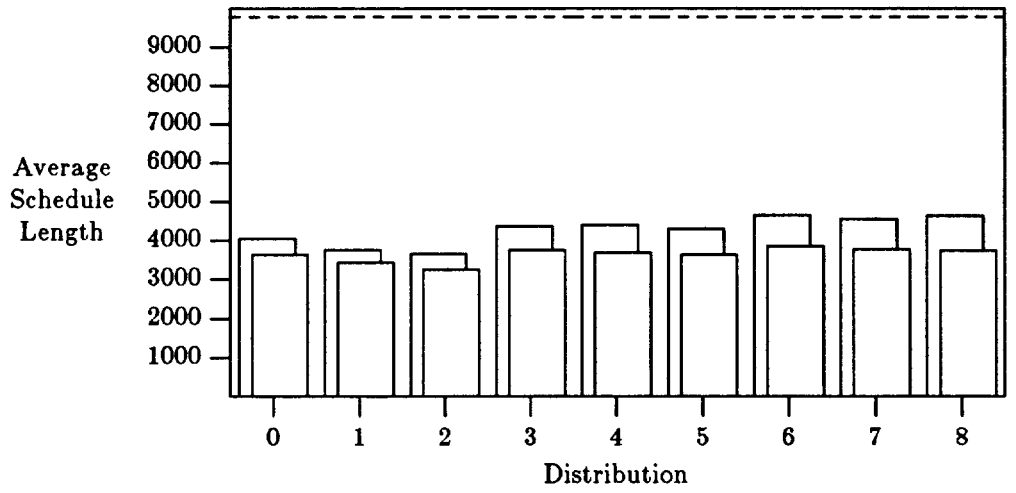




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	93.04	90.67	89.93	91.56	89.93	90.07	88.44	90.81	89.19
S/P	2.43	2.40	2.37	2.39	2.39	2.42	2.32	2.43	2.37
S/C	2.71	2.71	2.67	2.72	2.74	2.73	2.68	2.72	2.71
P/C	1.12	1.13	1.13	1.14	1.15	1.13	1.15	1.12	1.14
P Eff	0.57	0.55	0.52	0.55	0.55	0.53	0.52	0.53	0.53
C Eff	0.57	0.55	0.53	0.55	0.55	0.54	0.53	0.54	0.53
CPU Sec	352.66	358.93	368.04	373.73	380.61	388.22	395.58	393.50	400.46

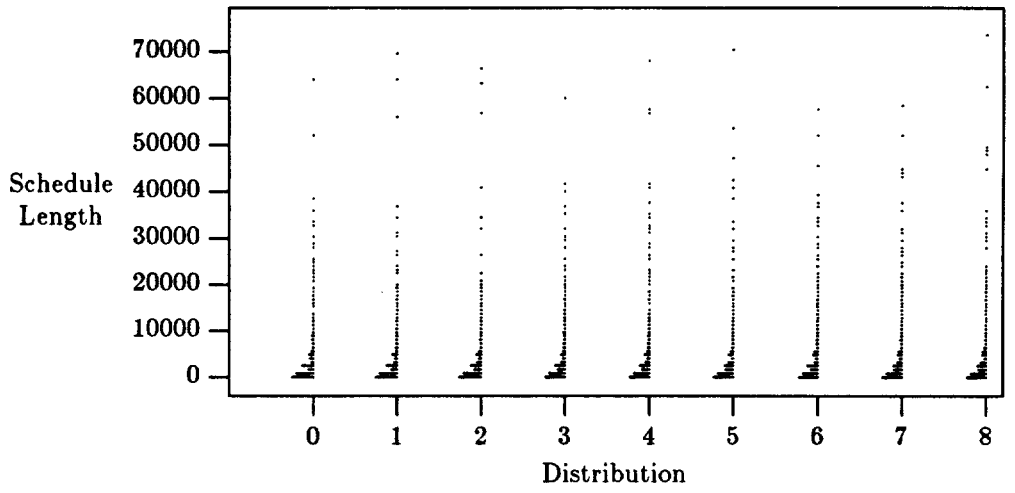
B.1.3. Figure B.3. — Scheduler 3

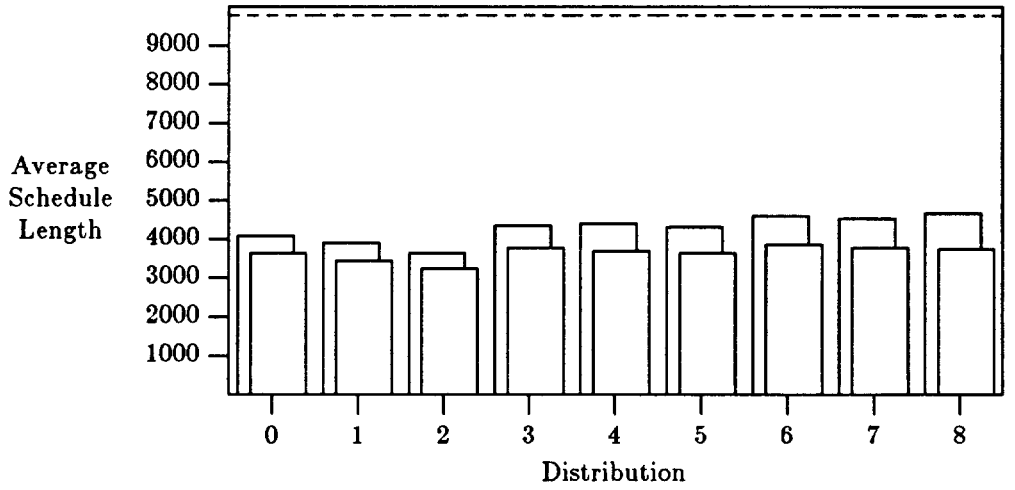




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	91.11	93.19	93.19	88.44	88.00	87.85	85.04	85.78	86.67
S/P	2.42	2.61	2.69	2.24	2.22	2.28	2.11	2.15	2.12
S/C	2.70	2.86	3.01	2.61	2.66	2.70	2.54	2.61	2.63
P/C	1.12	1.09	1.12	1.16	1.19	1.19	1.21	1.21	1.24
P Eff	0.57	0.57	0.57	0.54	0.54	0.53	0.51	0.52	0.52
C Eff	0.58	0.57	0.57	0.55	0.55	0.54	0.52	0.53	0.53
CPU Sec	419.27	425.95	433.74	460.77	477.16	486.70	507.85	503.09	513.46

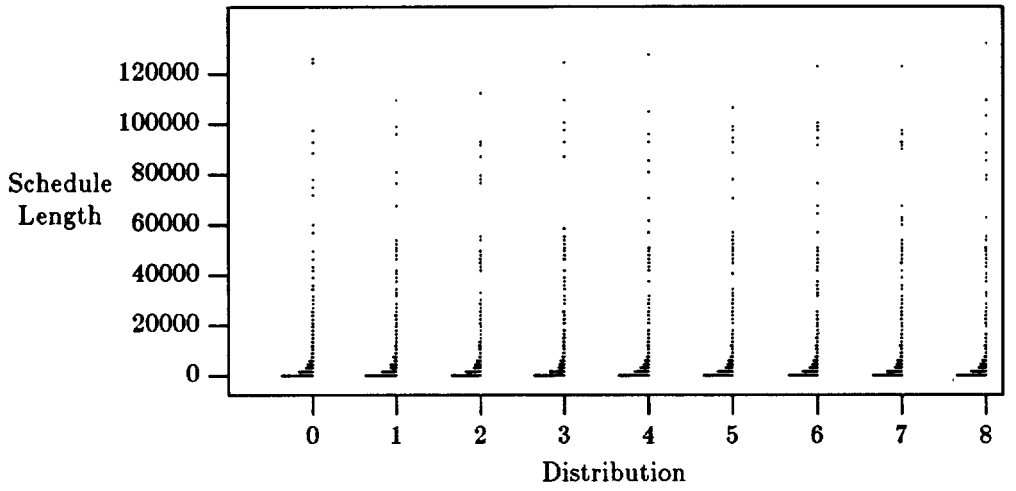
B.1.4. Figure B.4. — Scheduler 4

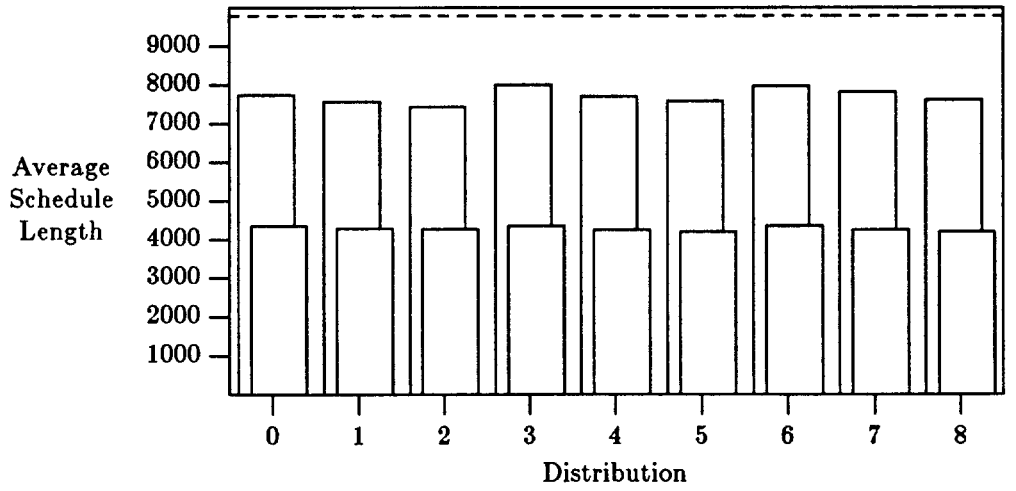




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	91.41	93.33	93.19	88.74	89.04	88.44	84.59	85.93	87.26
S/P	2.40	2.51	2.70	2.25	2.23	2.27	2.13	2.16	2.10
S/C	2.70	2.85	3.03	2.60	2.66	2.70	2.54	2.61	2.63
P/C	1.13	1.14	1.12	1.16	1.19	1.19	1.19	1.21	1.25
P Eff	0.57	0.57	0.57	0.54	0.54	0.53	0.51	0.52	0.52
C Eff	0.58	0.57	0.57	0.55	0.55	0.54	0.52	0.53	0.53
CPU Sec	798.10	815.98	824.09	853.34	875.43	888.79	914.43	909.83	924.85

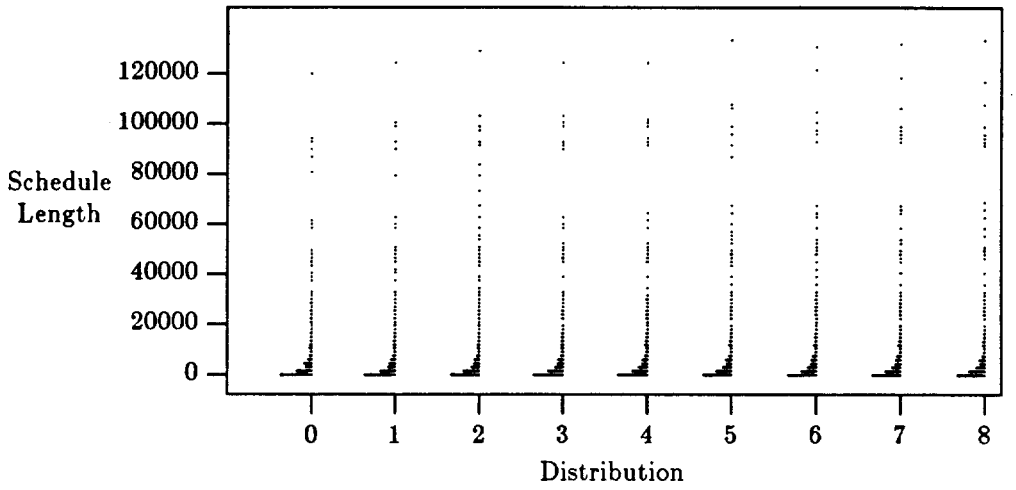
B.1.5. Figure B.5. — Scheduler 5

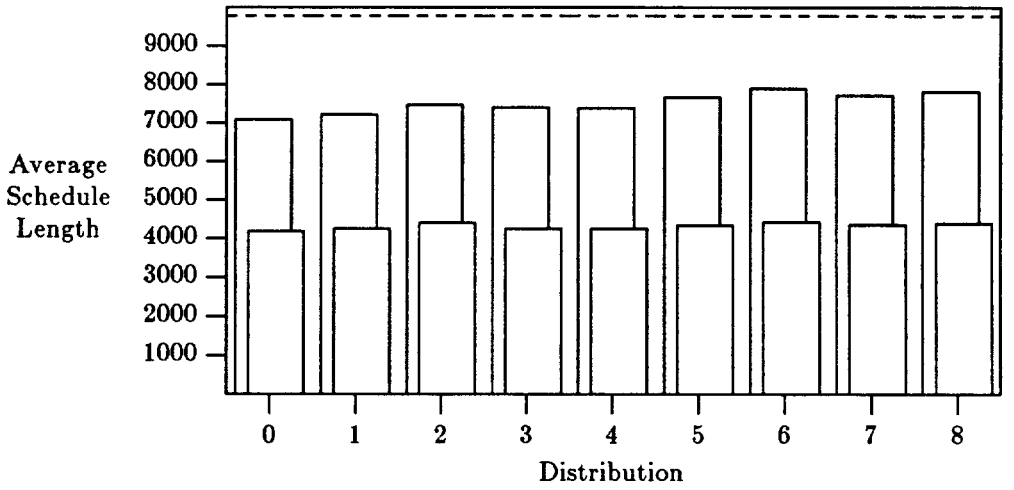




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	76.59	78.07	77.33	76.74	76.15	77.33	75.85	76.30	77.19
S/P	1.26	1.29	1.32	1.22	1.27	1.29	1.23	1.25	1.29
S/C	2.26	2.28	2.29	2.26	2.31	2.34	2.26	2.31	2.33
P/C	1.78	1.77	1.74	1.84	1.81	1.81	1.84	1.84	1.81
P Eff	0.51	0.49	0.47	0.49	0.49	0.48	0.47	0.48	0.48
C Eff	0.52	0.51	0.49	0.51	0.51	0.50	0.49	0.50	0.50
CPU Sec	2.98	3.38	4.18	3.32	3.56	3.92	3.93	3.96	4.19

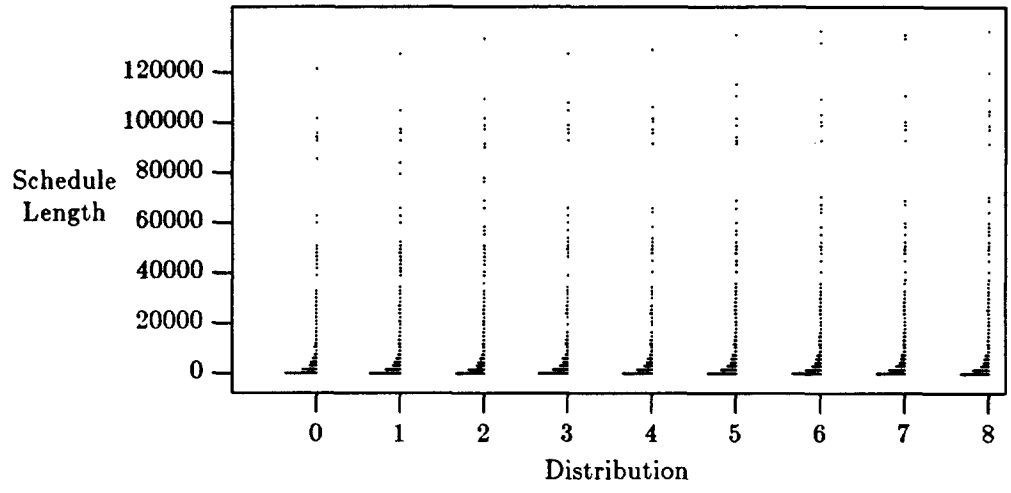
B.1.6. Figure B.6. — Scheduler 6

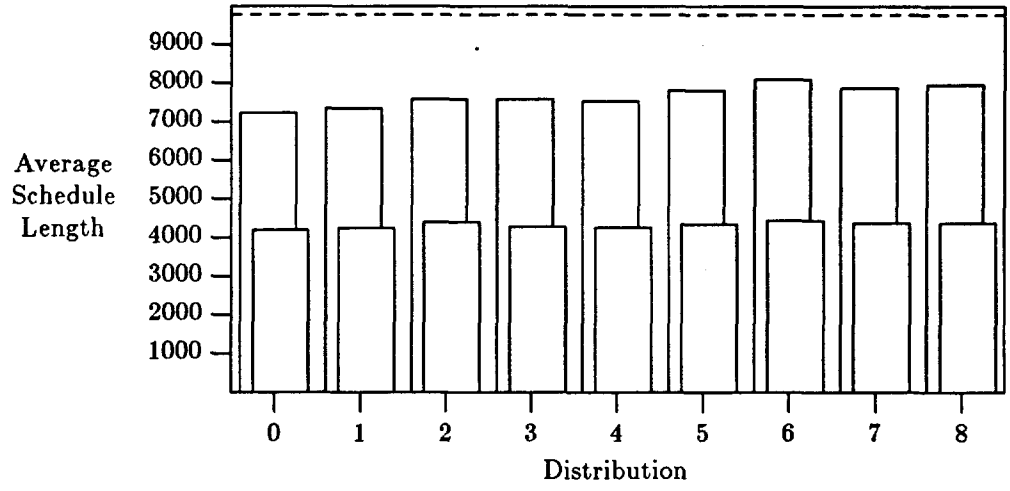




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	77.93	78.07	76.74	77.93	78.22	77.04	76.59	77.48	76.44
S/P	1.38	1.36	1.31	1.32	1.33	1.28	1.24	1.27	1.25
S/C	2.35	2.30	2.23	2.30	2.30	2.26	2.22	2.25	2.24
P/C	1.70	1.70	1.70	1.74	1.74	1.77	1.79	1.77	1.79
P Eff	0.51	0.48	0.45	0.48	0.48	0.46	0.45	0.46	0.45
C Eff	0.53	0.50	0.47	0.50	0.50	0.48	0.48	0.48	0.47
CPU Sec	54.50	55.39	57.03	57.21	57.89	59.25	60.57	60.19	61.12

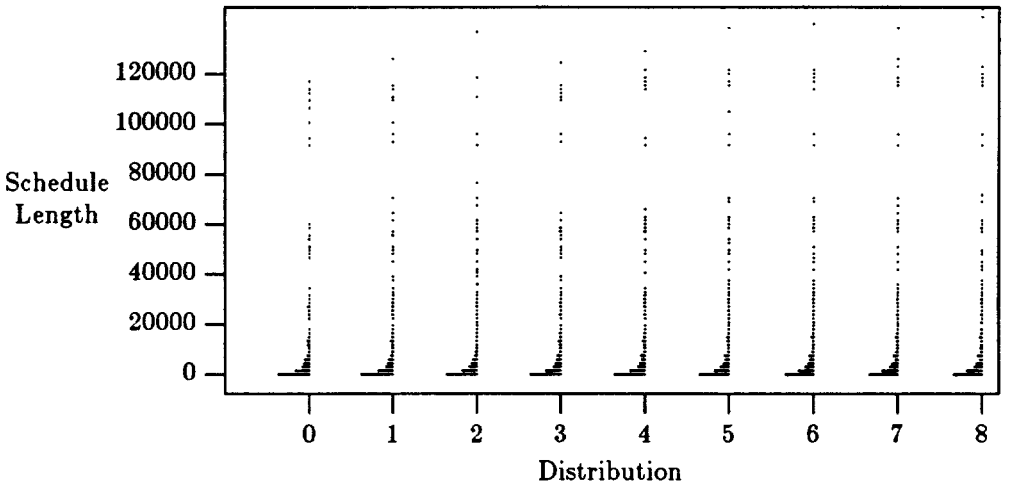
B.1.7. Figure B.7. — Scheduler 7

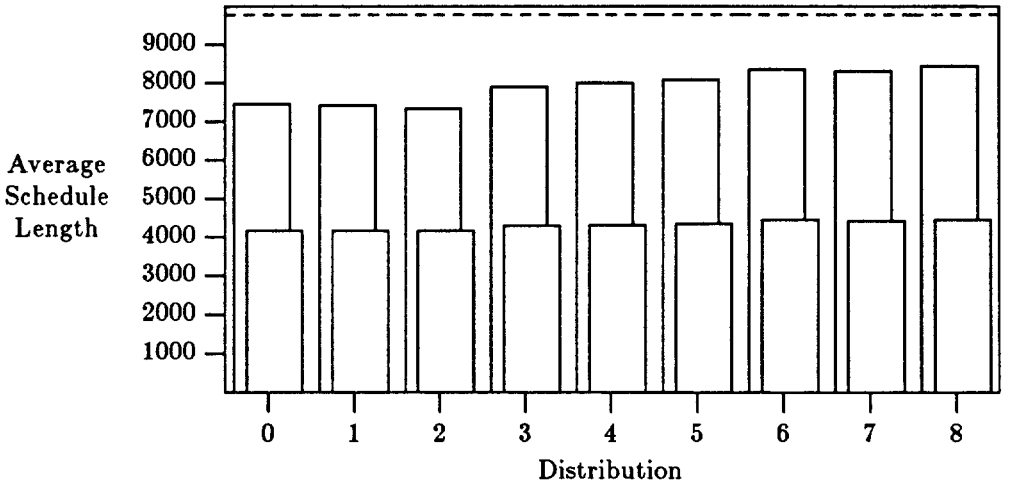




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	78.07	77.78	76.30	77.63	77.93	76.89	76.30	76.89	76.15
S/P	1.35	1.33	1.29	1.29	1.30	1.25	1.21	1.24	1.23
S/C	2.34	2.30	2.22	2.29	2.30	2.25	2.20	2.24	2.23
P/C	1.73	1.73	1.72	1.77	1.77	1.80	1.82	1.80	1.82
P Eff	0.51	0.48	0.45	0.48	0.48	0.46	0.45	0.46	0.45
C Eff	0.53	0.50	0.47	0.50	0.50	0.48	0.47	0.48	0.47
CPU Sec	25.37	25.42	25.95	26.46	26.89	27.42	27.83	27.71	28.38

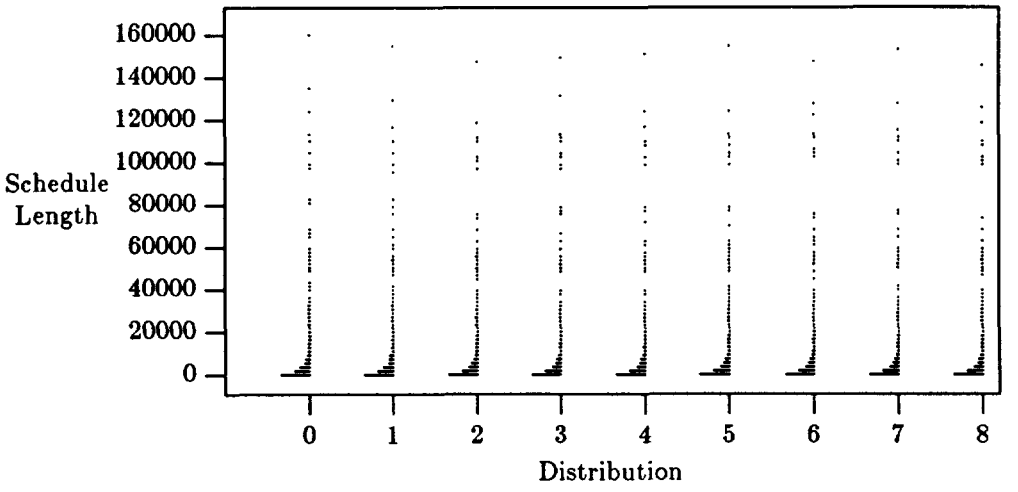
B.1.8. Figure B.8. — Scheduler 8

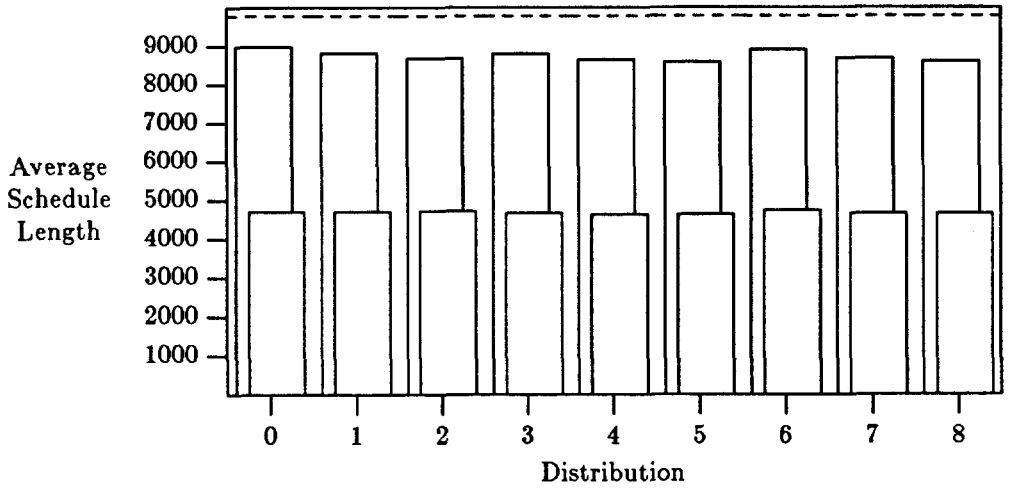




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	78.52	79.26	78.81	78.37	77.93	77.48	76.30	76.74	76.44
S/P	1.31	1.32	1.33	1.24	1.22	1.21	1.17	1.18	1.16
S/C	2.35	2.35	2.35	2.28	2.27	2.26	2.20	2.22	2.20
P/C	1.79	1.78	1.76	1.84	1.86	1.86	1.88	1.89	1.90
P Eff	0.51	0.50	0.49	0.48	0.47	0.47	0.45	0.46	0.45
C Eff	0.53	0.52	0.50	0.50	0.50	0.49	0.47	0.48	0.47
CPU Sec	50.05	52.41	55.03	54.65	57.34	59.87	60.13	60.70	62.60

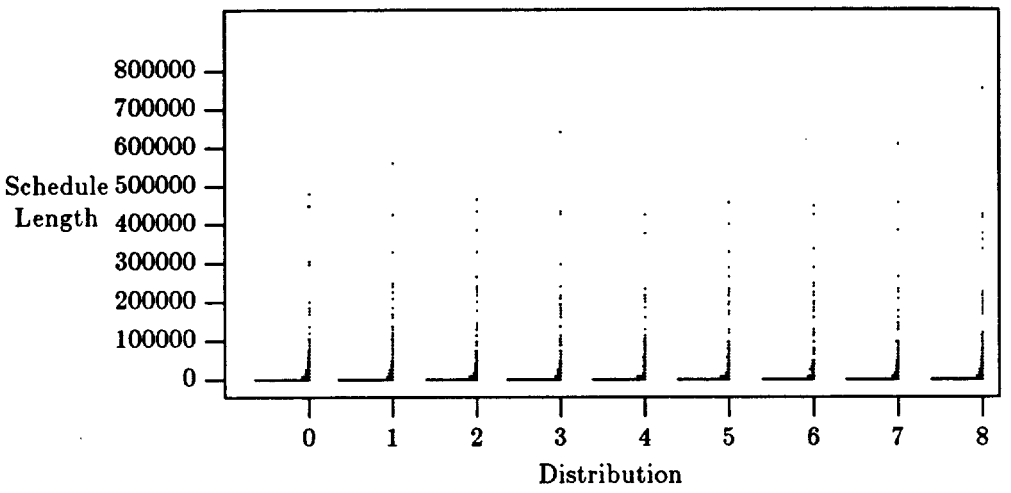
B.1.9. Figure B.9. — Scheduler 9

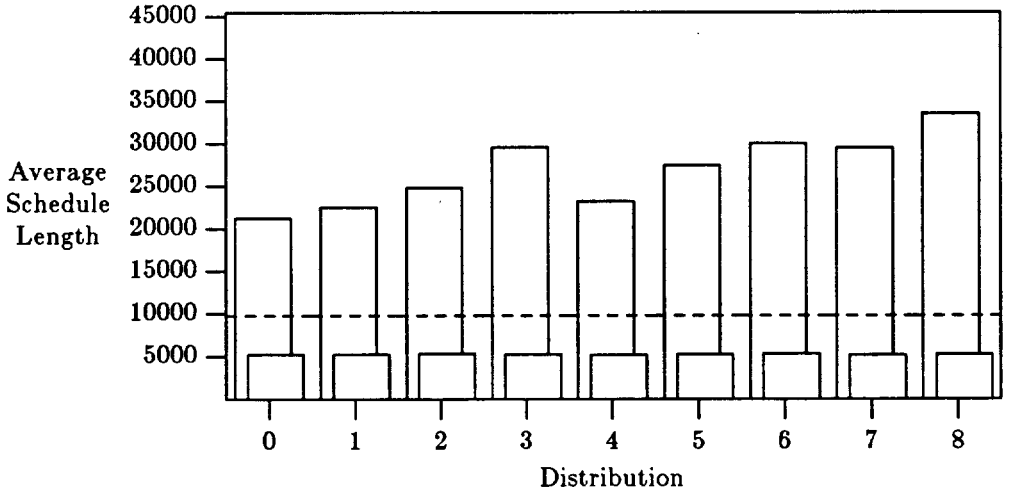




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	75.11	75.26	75.11	75.41	76.15	76.44	75.41	76.00	76.15
S/P	1.09	1.11	1.13	1.11	1.13	1.14	1.10	1.13	1.14
S/C	2.08	2.09	2.07	2.10	2.12	2.12	2.07	2.10	2.11
P/C	1.91	1.88	1.84	1.88	1.87	1.86	1.88	1.87	1.85
P Eff	0.40	0.40	0.39	0.40	0.40	0.40	0.39	0.40	0.40
C Eff	0.42	0.42	0.42	0.42	0.43	0.43	0.41	0.42	0.42
CPU Sec	1.19	1.21	1.24	1.22	1.23	1.24	1.24	1.24	1.26

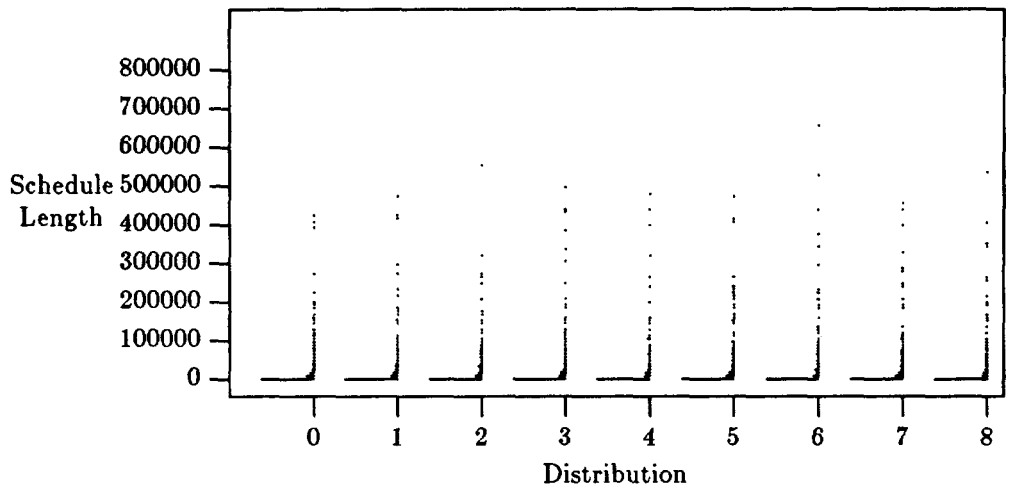
B.1.10. Figure B.10. — Scheduler 10

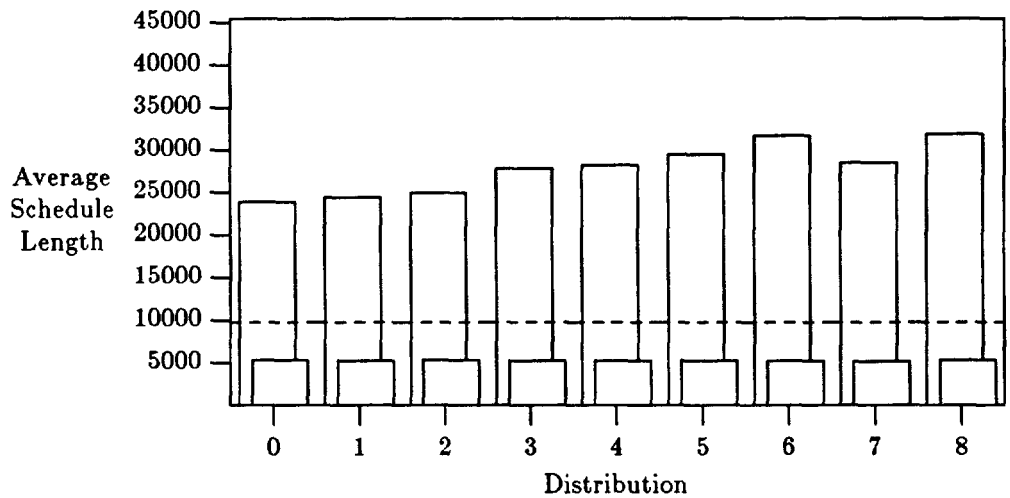




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	64.00	62.96	60.00	61.48	62.22	61.63	60.15	62.81	60.00
S/P	0.46	0.43	0.39	0.33	0.42	0.36	0.33	0.33	0.29
S/C	1.85	1.87	1.84	1.86	1.88	1.85	1.84	1.88	1.84
P/C	4.03	4.30	4.66	5.62	4.47	5.19	5.64	5.67	6.33
P Eff	0.44	0.43	0.41	0.43	0.43	0.41	0.41	0.42	0.41
C Eff	0.48	0.47	0.45	0.47	0.47	0.45	0.45	0.46	0.45
CPU Sec	19.86	21.73	24.74	20.81	22.38	24.88	24.10	24.16	28.19

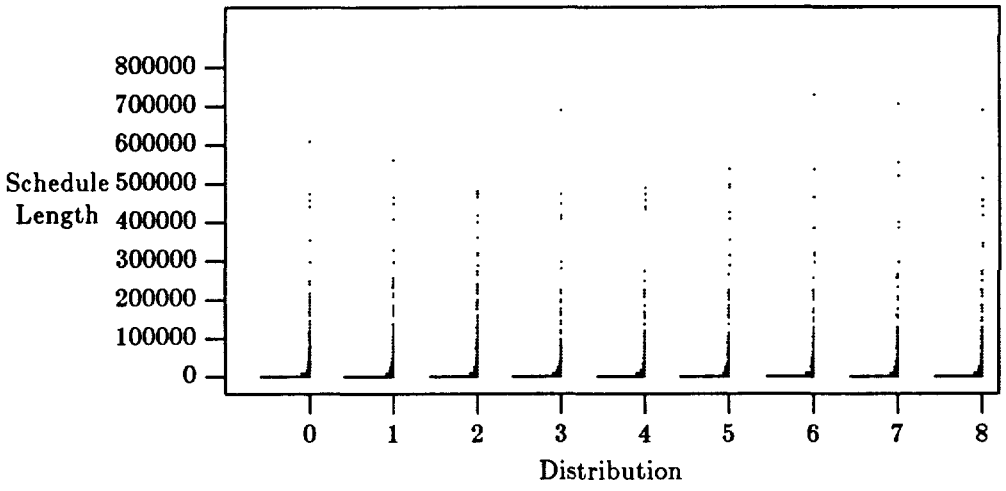
B.1.11. Figure B.11. — Scheduler 11

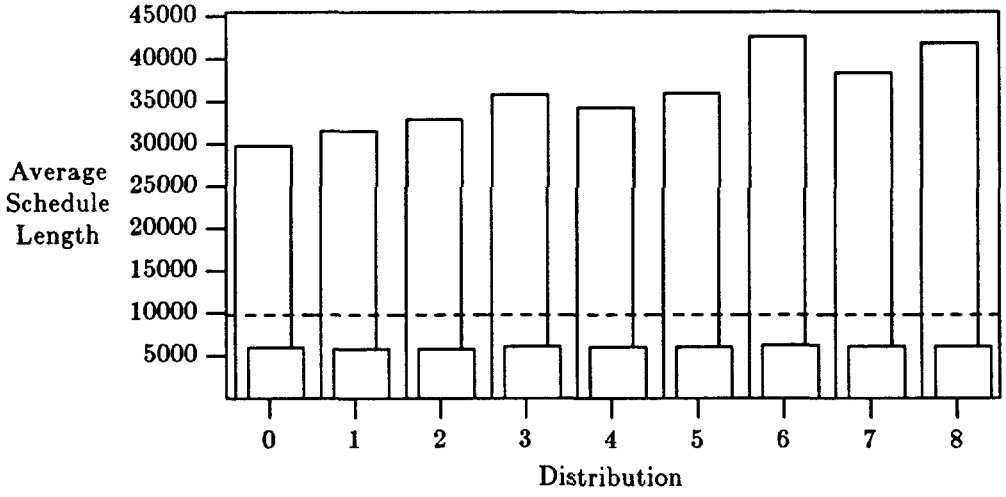




	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	63.11	62.96	58.81	62.07	61.33	59.41	60.44	60.59	60.30
S/P	0.41	0.40	0.39	0.35	0.35	0.33	0.31	0.34	0.31
S/C	1.84	1.84	1.82	1.87	1.84	1.84	1.85	1.88	1.83
P/C	4.49	4.60	4.64	5.33	5.31	5.56	6.02	5.48	5.98
P Eff	0.45	0.43	0.41	0.43	0.43	0.41	0.41	0.41	0.41
C Eff	0.48	0.47	0.45	0.47	0.47	0.45	0.45	0.46	0.45
CPU Sec	46.83	49.04	52.50	50.58	52.23	54.42	55.65	55.02	58.25

B.1.12. Figure B.12. — Scheduler 12

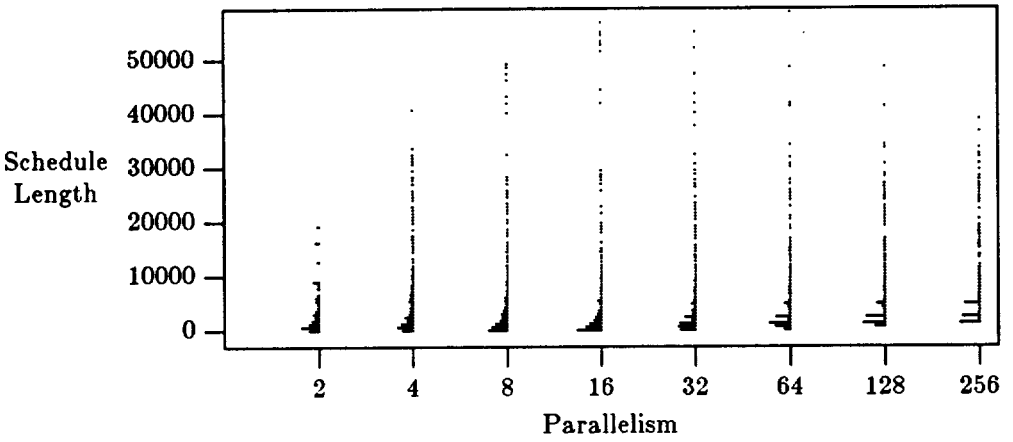


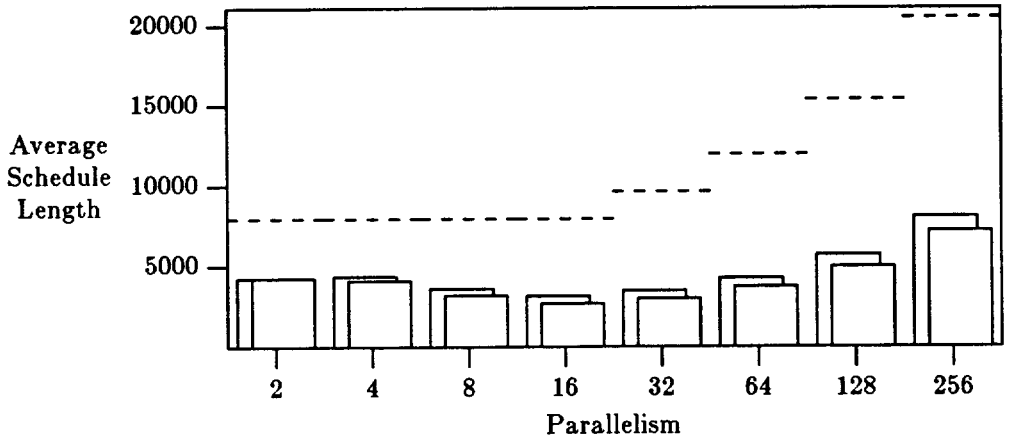


	Distribution								
	0	1	2	3	4	5	6	7	8
%P≤S	59.26	58.07	56.89	58.67	55.85	56.59	53.33	55.11	55.85
S/P	0.33	0.31	0.30	0.27	0.29	0.27	0.23	0.25	0.23
S/C	1.64	1.68	1.69	1.61	1.63	1.62	1.57	1.61	1.61
P/C	5.01	5.41	5.69	5.90	5.70	5.97	6.85	6.32	6.88
P Eff	0.37	0.38	0.37	0.34	0.35	0.35	0.31	0.33	0.34
C Eff	0.41	0.42	0.42	0.38	0.40	0.40	0.36	0.38	0.39
CPU Sec	14.30	15.60	17.22	14.91	15.82	16.73	15.63	16.41	16.89

B.2. Scheduler Performance By Parallelism

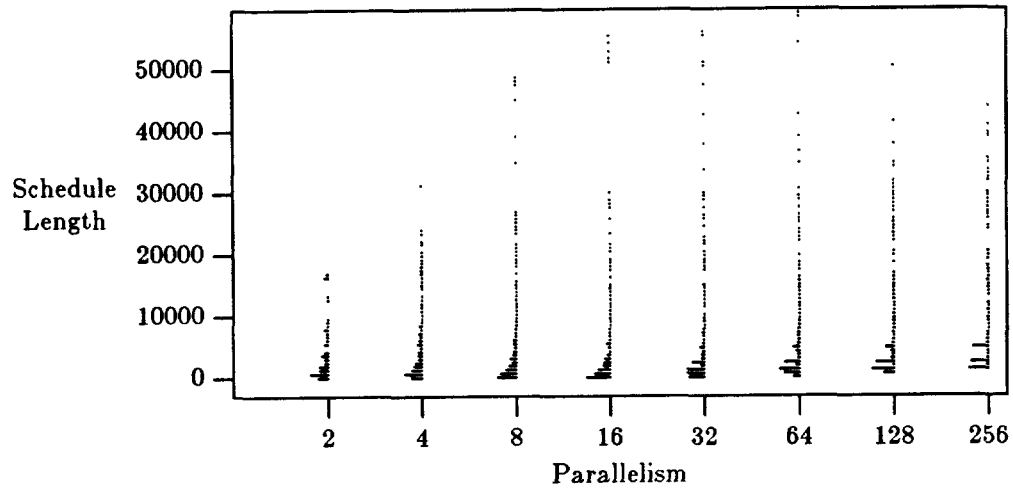
B.2.1. Figure B.13. — Scheduler 1

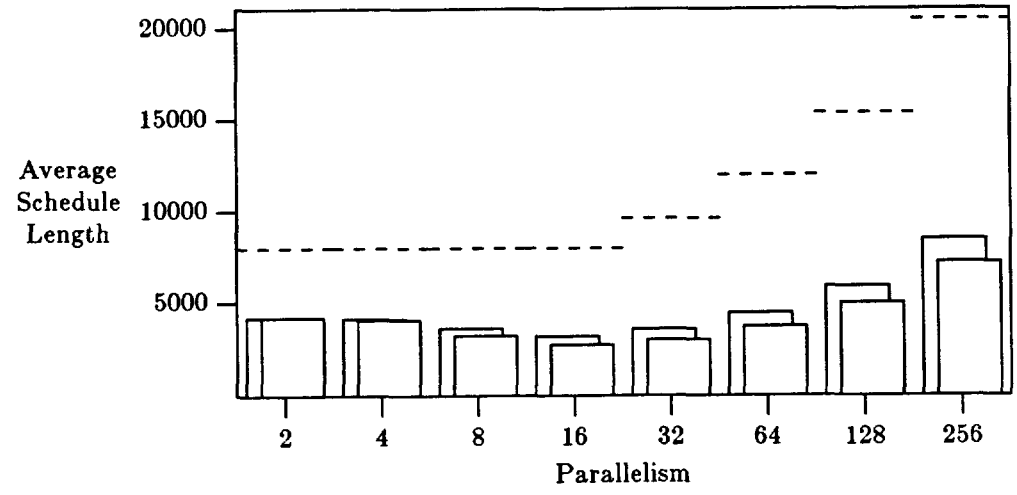




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	100.00	88.52	88.40	88.89	89.51	89.16	88.68	88.48
S/P	1.87	1.82	2.21	2.52	2.76	2.82	2.70	2.55
S/C	1.87	1.94	2.49	2.97	3.21	3.22	3.10	2.86
P/C	1.00	1.07	1.13	1.18	1.16	1.14	1.14	1.12
P Eff	0.34	0.33	0.46	0.57	0.65	0.68	0.70	0.68
C Eff	0.34	0.34	0.47	0.58	0.65	0.69	0.70	0.69
CPU Sec	171.53	203.84	250.10	280.26	363.91	473.72	637.22	909.51

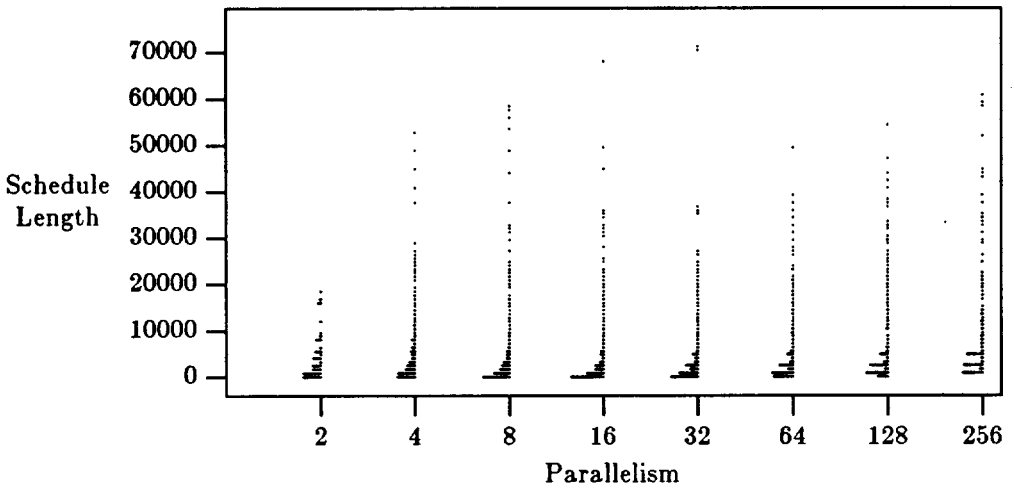
B.2.2. Figure B.14. — Scheduler 2

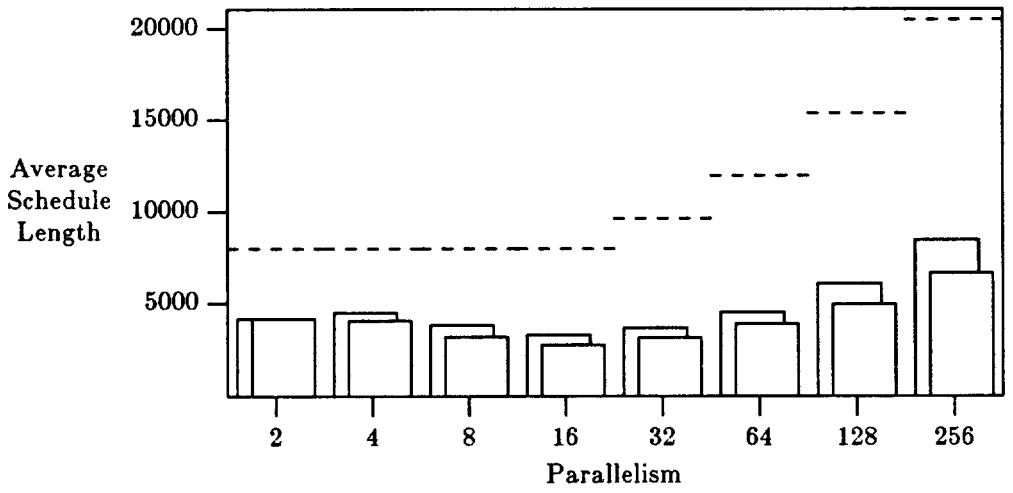




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	100.00	95.46	89.88	88.97	89.09	88.34	88.48	87.65
S/P	1.89	1.93	2.21	2.48	2.70	2.70	2.62	2.42
S/C	1.89	1.96	2.46	2.93	3.20	3.22	3.10	2.86
P/C	1.00	1.02	1.12	1.18	1.18	1.19	1.18	1.18
P Eff	0.34	0.34	0.46	0.56	0.64	0.68	0.69	0.68
C Eff	0.34	0.34	0.46	0.57	0.65	0.69	0.70	0.69
CPU Sec	183.66	217.03	269.72	303.77	395.41	514.76	697.28	1022.49

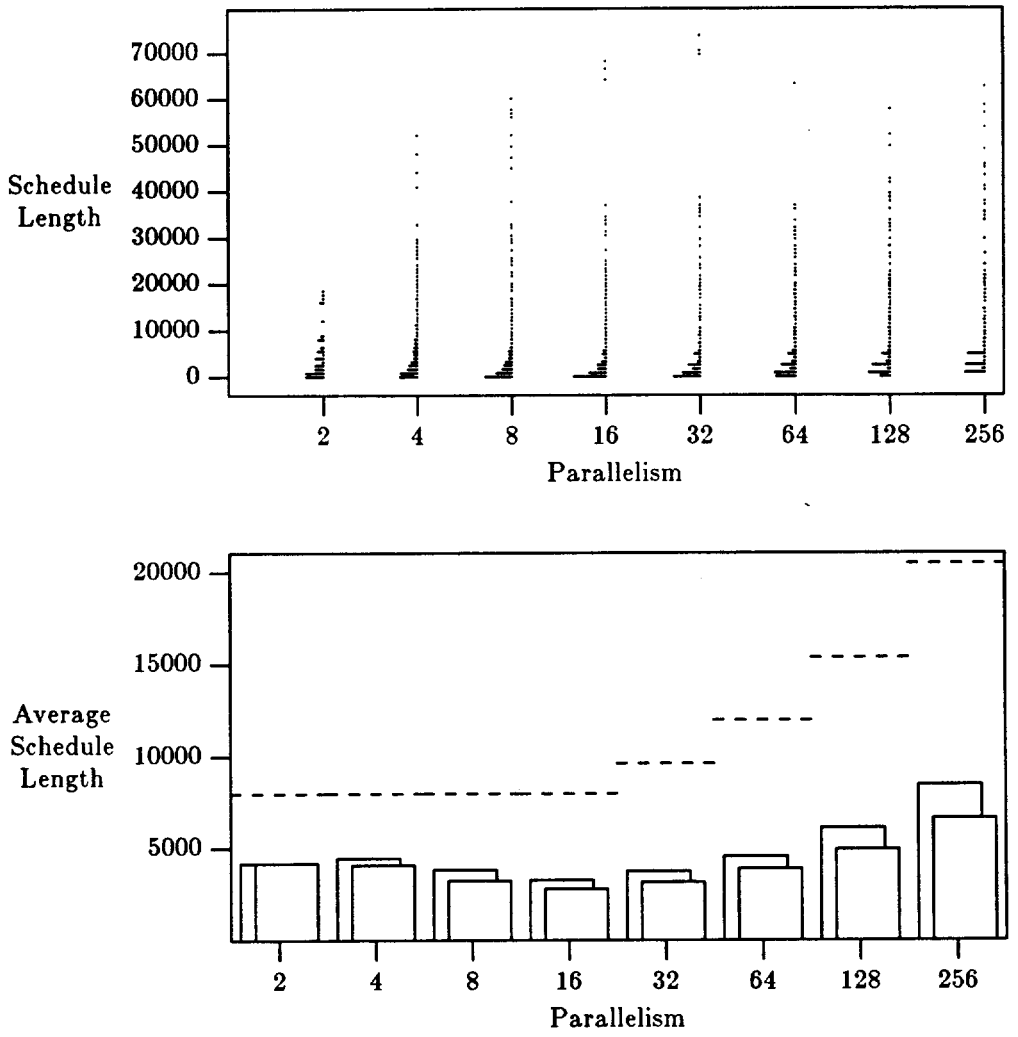
B.2.3. Figure B.15. — Scheduler 3





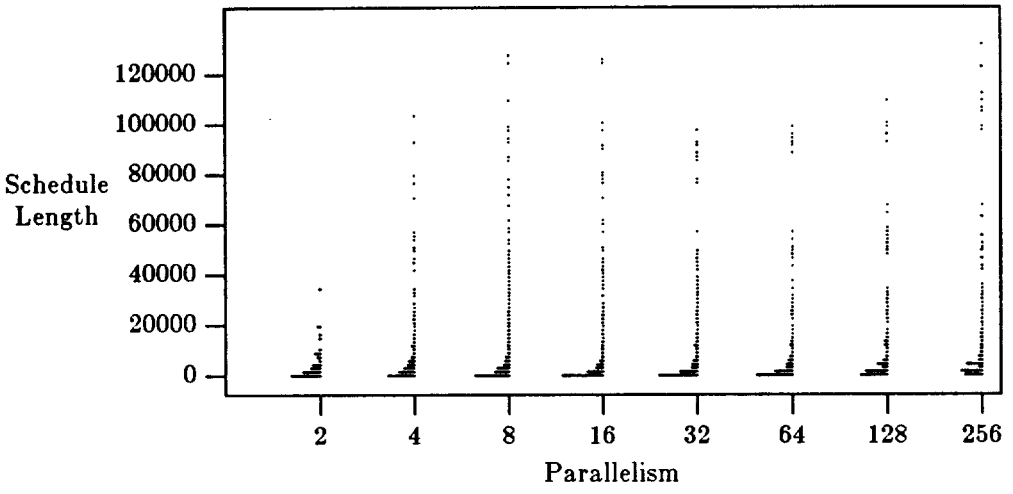
	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	100.00	88.43	87.65	88.89	89.40	88.89	88.68	87.24
S/P	1.91	1.78	2.09	2.41	2.61	2.64	2.54	2.43
S/C	1.91	1.96	2.49	2.89	3.04	3.09	3.11	3.09
P/C	1.00	1.10	1.19	1.20	1.16	1.17	1.23	1.27
P Eff	0.34	0.33	0.46	0.56	0.64	0.68	0.70	0.71
C Eff	0.34	0.34	0.47	0.57	0.65	0.69	0.71	0.72
CPU Sec	198.23	234.00	297.69	338.98	475.82	680.99	985.83	1493.03

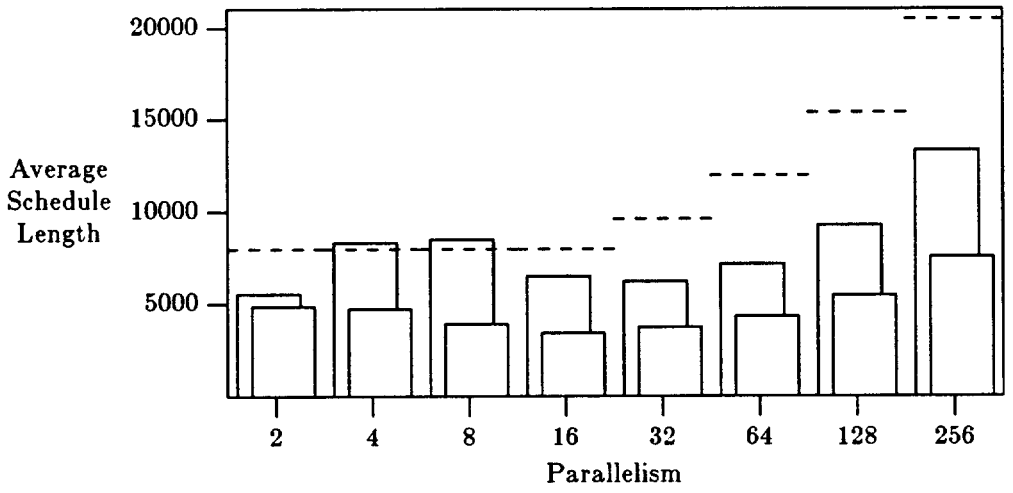
B.2.4. Figure B.16. — Scheduler 4



	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	100.00	88.52	87.41	89.38	90.33	89.30	88.68	88.07
S/P	1.91	1.78	2.08	2.43	2.56	2.63	2.53	2.42
S/C	1.91	1.96	2.48	2.89	3.04	3.09	3.12	3.10
P/C	1.00	1.10	1.20	1.19	1.19	1.17	1.23	1.28
P Eff	0.34	0.33	0.46	0.57	0.64	0.68	0.70	0.71
C Eff	0.34	0.34	0.47	0.57	0.65	0.69	0.71	0.72
CPU Sec	388.22	463.48	567.71	639.18	870.06	1205.80	1750.92	2770.58

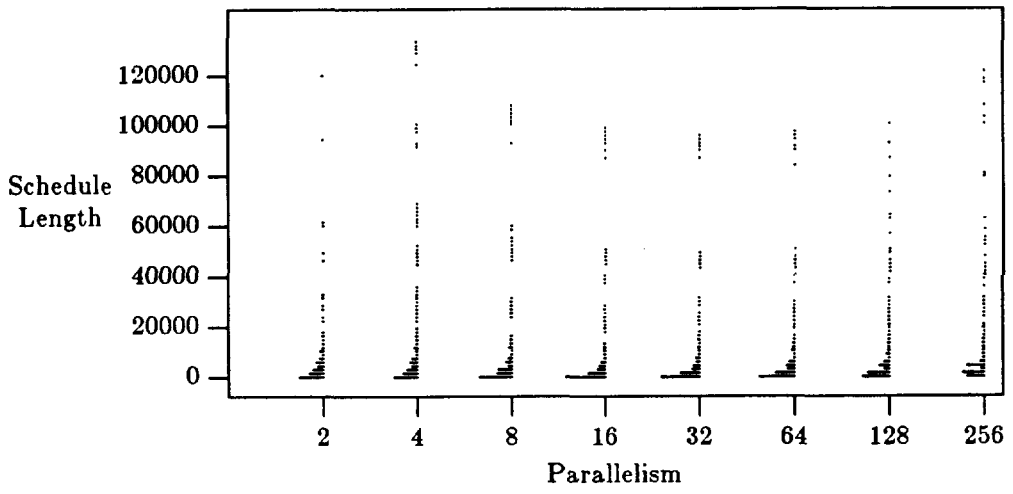
B.2.5. Figure B.17. — Scheduler 5

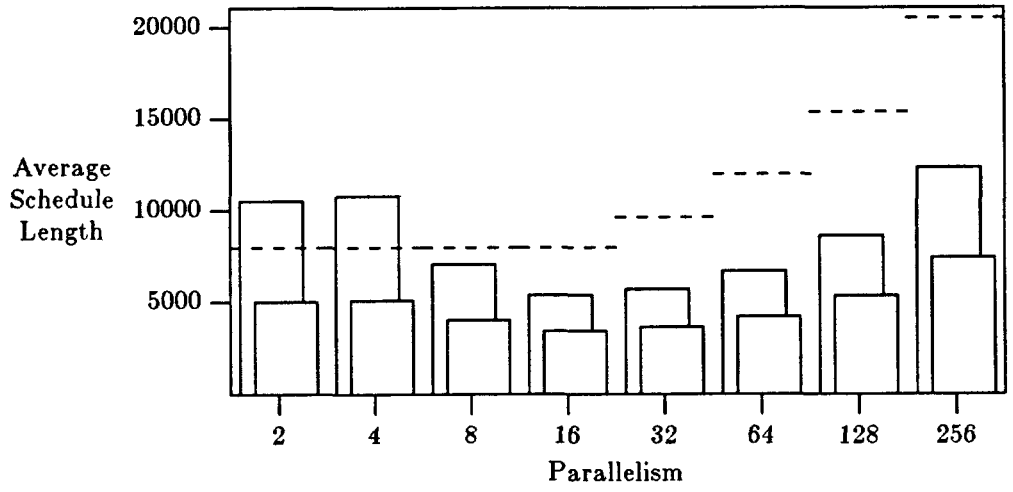




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	77.78	69.26	72.76	76.46	80.97	82.72	84.16	83.54
S/P	1.44	0.96	0.94	1.23	1.55	1.68	1.66	1.54
S/C	1.63	1.68	2.03	2.32	2.58	2.75	2.81	2.72
P/C	1.13	1.76	2.17	1.89	1.66	1.64	1.70	1.76
P Eff	0.31	0.29	0.40	0.50	0.58	0.62	0.65	0.64
C Eff	0.31	0.31	0.42	0.52	0.60	0.64	0.66	0.66
CPU Sec	1.54	1.72	1.98	2.20	3.08	4.84	8.92	18.79

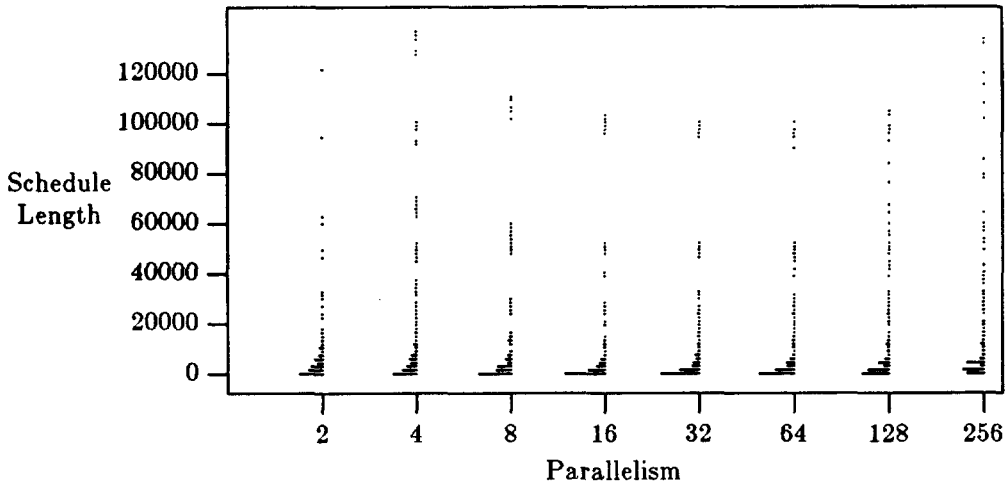
B.2.6. Figure B.18. — Scheduler 6

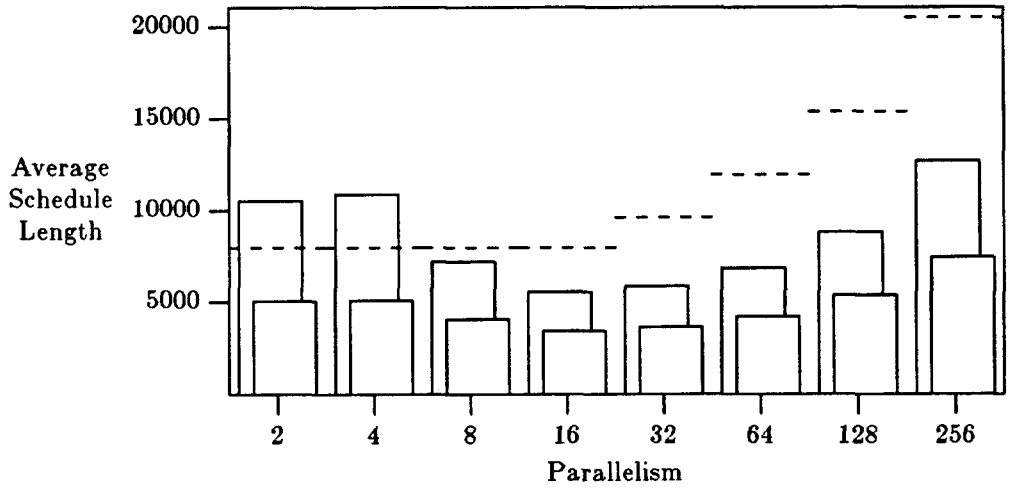




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	66.67	66.02	74.07	78.52	83.85	84.22	84.57	83.95
S/P	0.76	0.74	1.13	1.48	1.70	1.80	1.79	1.66
S/C	1.58	1.56	1.97	2.33	2.65	2.85	2.89	2.77
P/C	2.09	2.12	1.74	1.58	1.56	1.59	1.61	1.67
P Eff	0.27	0.25	0.38	0.49	0.57	0.63	0.65	0.65
C Eff	0.30	0.28	0.40	0.51	0.59	0.64	0.67	0.67
CPU Sec	30.87	34.53	39.91	43.70	57.17	76.72	111.04	183.65

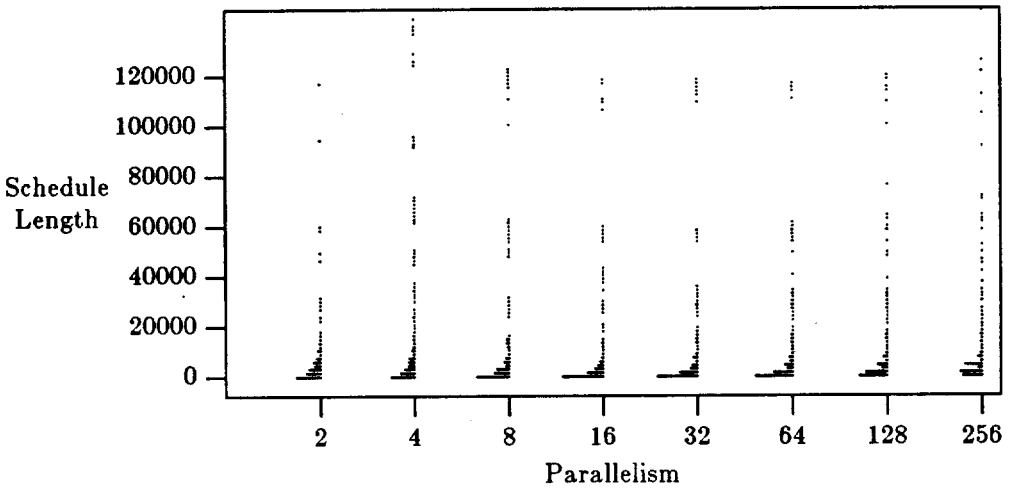
B.2.7. Figure B.19. — Scheduler 7

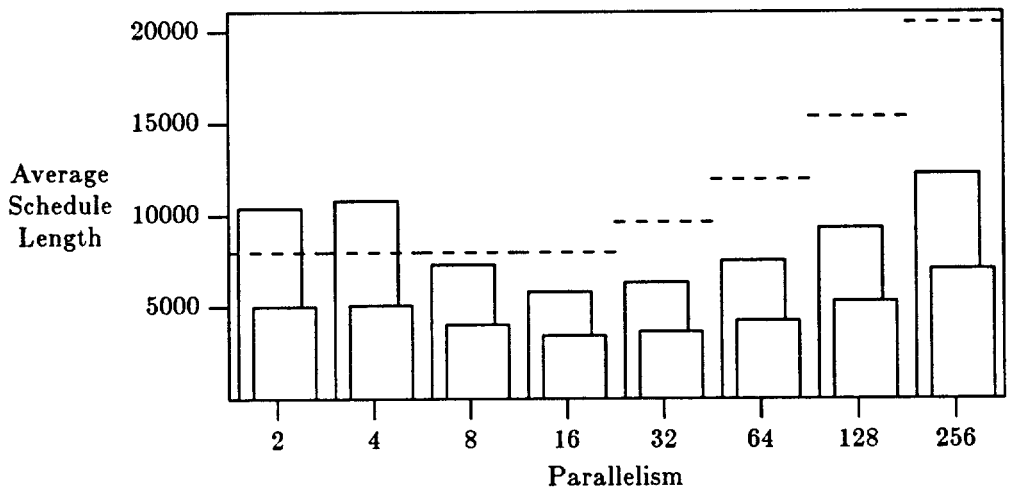




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	66.67	65.19	74.07	78.11	83.85	84.09	83.95	84.36
S/P	0.76	0.73	1.11	1.44	1.65	1.75	1.75	1.61
S/C	1.58	1.56	1.96	2.32	2.64	2.84	2.88	2.76
P/C	2.09	2.13	1.77	1.62	1.60	1.62	1.65	1.71
P Eff	0.27	0.25	0.38	0.49	0.57	0.62	0.65	0.65
C Eff	0.30	0.28	0.40	0.51	0.59	0.64	0.67	0.67
CPU Sec	16.44	17.23	19.43	20.86	26.86	35.26	48.60	73.04

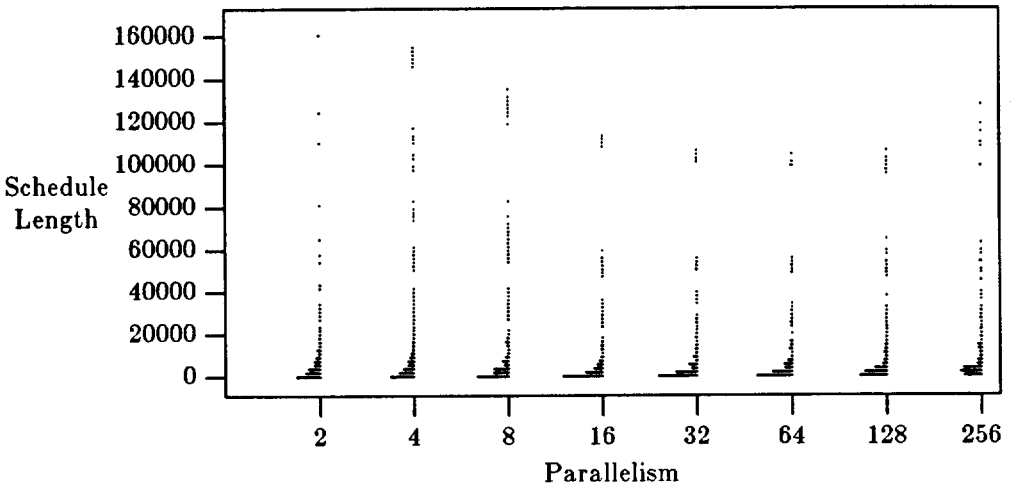
B.2.8. Figure B.20. — Scheduler 8

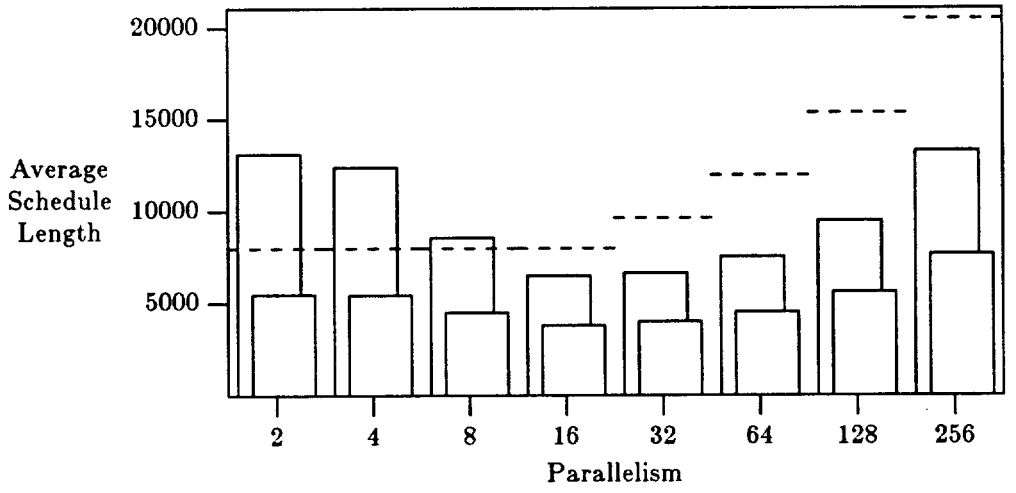




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	66.67	65.00	74.07	79.26	84.98	85.05	85.39	85.60
S/P	0.77	0.73	1.09	1.37	1.52	1.60	1.65	1.67
S/C	1.58	1.56	1.97	2.33	2.64	2.82	2.90	2.91
P/C	2.07	2.13	1.80	1.71	1.74	1.77	1.75	1.74
P Eff	0.27	0.25	0.38	0.50	0.58	0.63	0.66	0.67
C Eff	0.30	0.28	0.40	0.52	0.60	0.65	0.68	0.69
CPU Sec	25.44	28.98	36.46	41.35	55.60	78.74	119.16	195.49

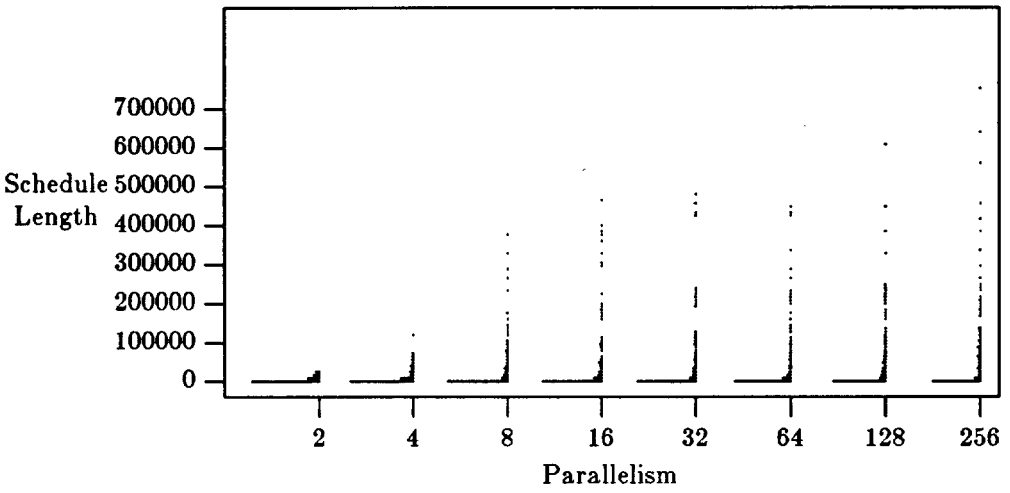
B.2.9. Figure B.21. — Scheduler 9

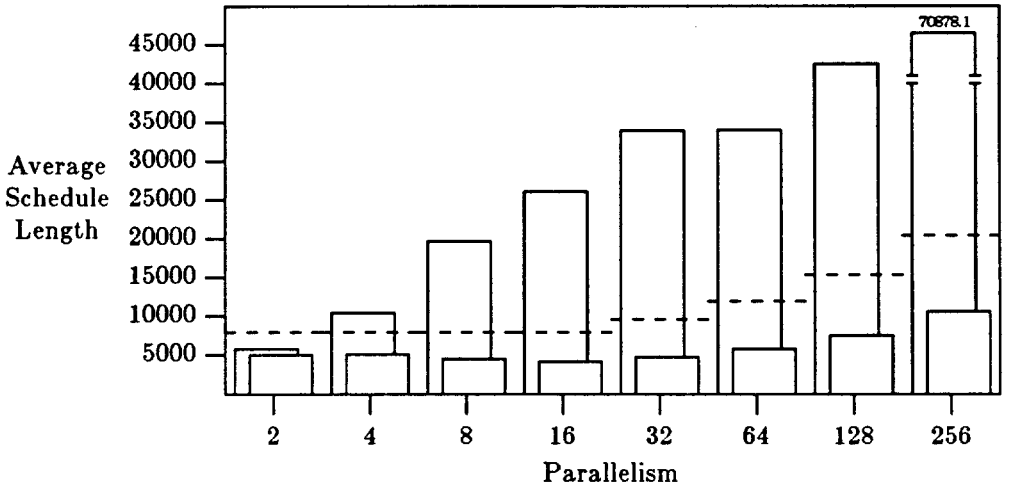




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	61.48	63.15	72.76	77.45	80.56	83.54	84.57	83.95
S/P	0.61	0.64	0.93	1.22	1.45	1.59	1.63	1.54
S/C	1.45	1.46	1.78	2.11	2.40	2.65	2.74	2.68
P/C	2.39	2.28	1.91	1.72	1.65	1.66	1.69	1.74
P Eff	0.21	0.21	0.31	0.40	0.48	0.55	0.60	0.61
C Eff	0.25	0.25	0.33	0.42	0.50	0.57	0.61	0.63
CPU Sec	0.79	0.87	0.95	1.00	1.25	1.58	2.06	2.82

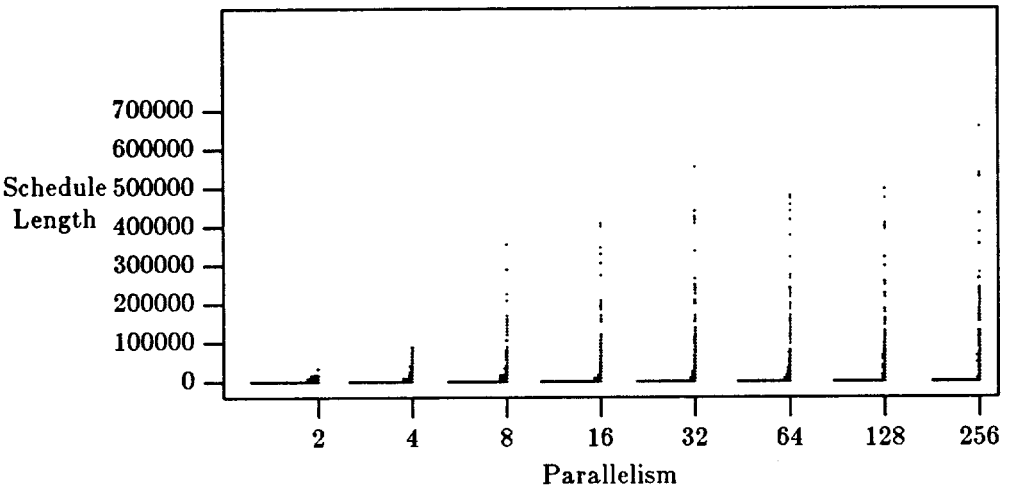
B.2.10. Figure B.22. — Scheduler 10

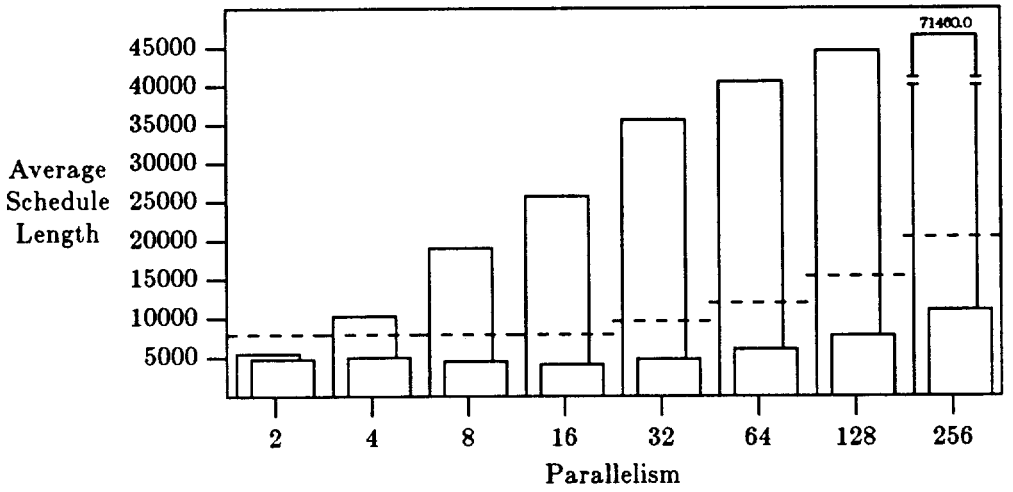




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	73.33	58.70	59.01	61.48	63.99	65.02	63.79	59.67
S/P	1.37	0.76	0.40	0.30	0.28	0.35	0.36	0.29
S/C	1.57	1.55	1.73	1.89	2.01	2.06	2.03	1.92
P/C	1.15	2.04	4.30	6.22	7.11	5.86	5.62	6.66
P Eff	0.29	0.26	0.35	0.44	0.50	0.54	0.55	0.54
C Eff	0.30	0.29	0.39	0.48	0.54	0.58	0.59	0.58
CPU Sec	5.85	7.94	11.72	15.25	22.59	34.76	55.88	105.92

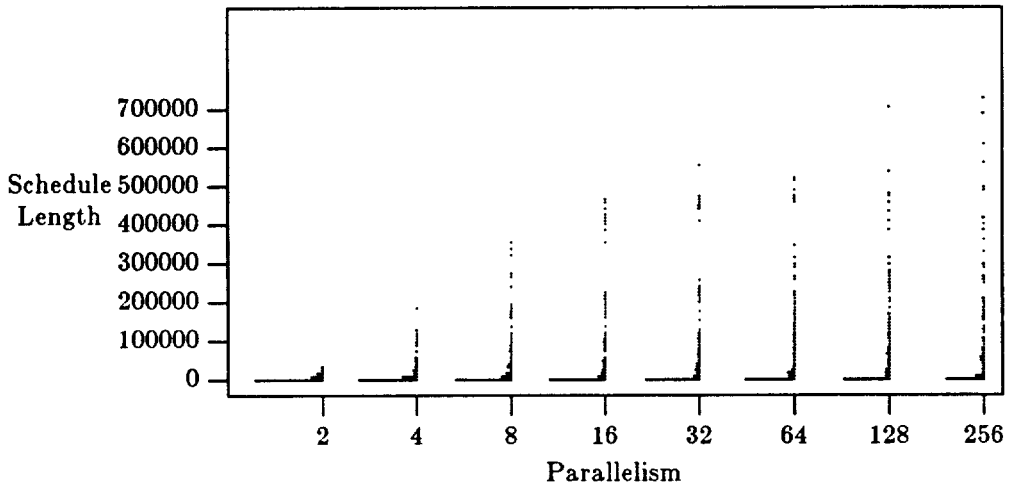
B.2.11. Figure B.23. — Scheduler 11

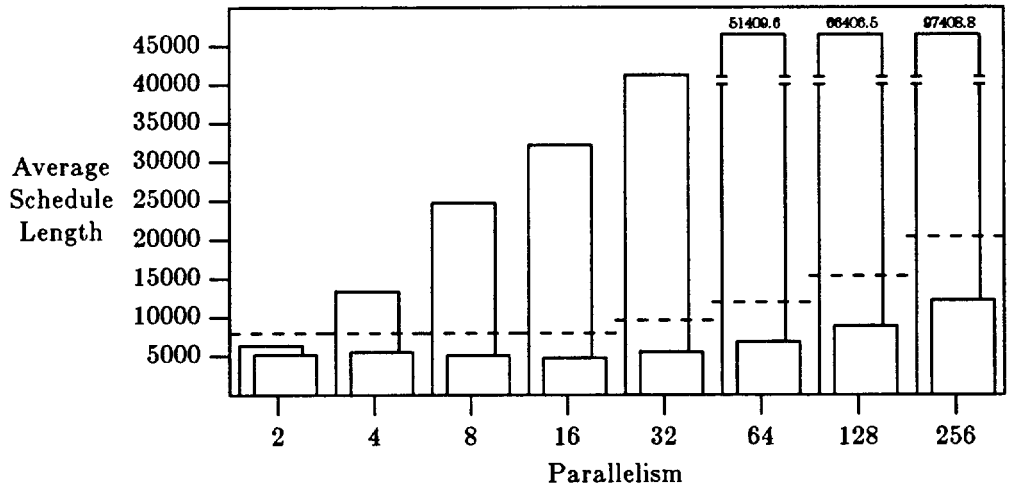




	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	77.78	60.28	59.18	62.22	61.73	61.18	59.88	56.79
S/P	1.43	0.77	0.42	0.31	0.27	0.29	0.34	0.29
S/C	1.64	1.58	1.76	1.92	1.98	1.96	1.97	1.86
P/C	1.14	2.06	4.24	6.21	7.36	6.65	5.72	6.50
P Eff	0.30	0.27	0.36	0.44	0.50	0.53	0.54	0.53
C Eff	0.31	0.30	0.40	0.48	0.54	0.57	0.59	0.58
CPU Sec	19.89	24.87	31.71	37.43	51.88	73.10	111.80	200.41

B.2.12. Figure B.24. — Scheduler 12

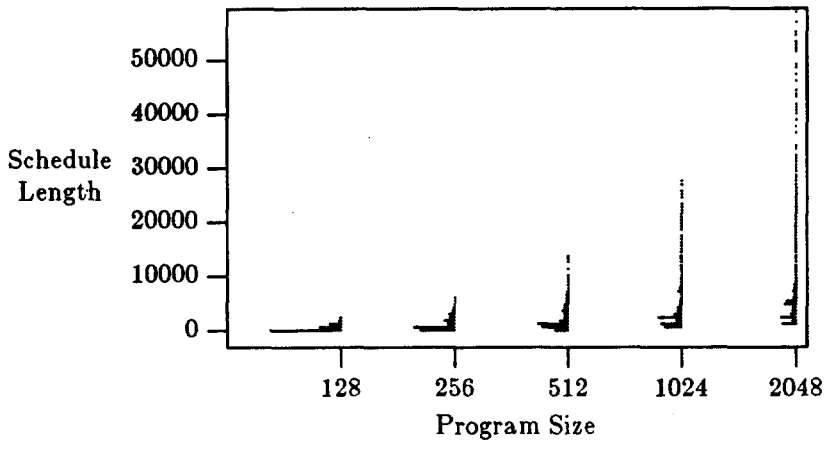


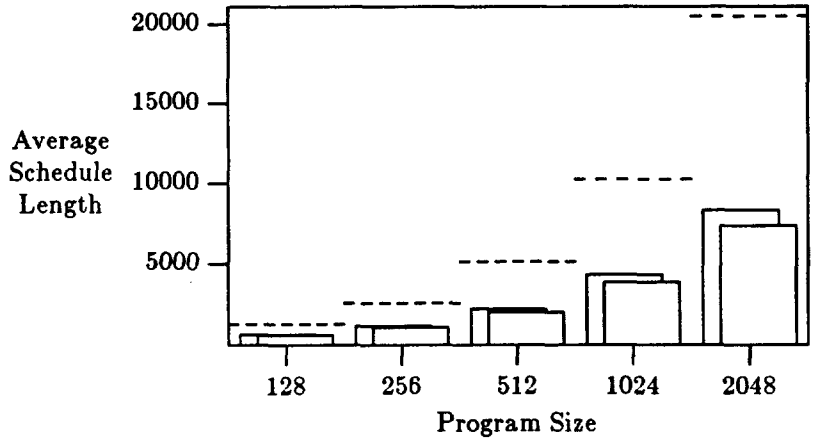


	Average Parallelism							
	2	4	8	16	32	64	128	256
%P≤S	73.33	58.06	55.72	55.64	56.58	56.79	54.73	53.91
S/P	1.24	0.59	0.32	0.25	0.23	0.23	0.23	0.21
S/C	1.53	1.44	1.55	1.65	1.72	1.73	1.73	1.68
P/C	1.23	2.43	4.84	6.68	7.38	7.44	7.48	7.97
P Eff	0.28	0.23	0.28	0.36	0.41	0.44	0.46	0.44
C Eff	0.29	0.27	0.33	0.41	0.46	0.49	0.51	0.50
CPU Sec	5.49	6.96	9.55	11.85	16.62	23.24	34.02	53.51

B.3. Scheduler Performance By Program Size

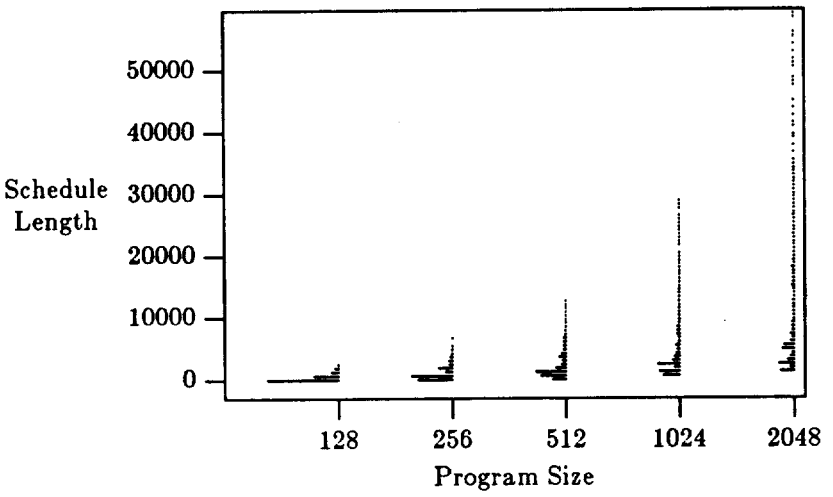
B.3.1. Figure B.25. — Scheduler 1

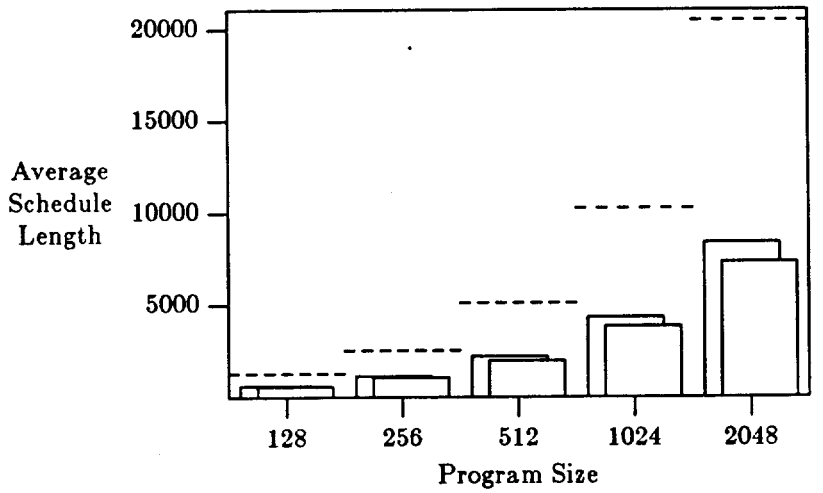




	Program Size				
	128	256	512	1024	2048
%P≤S	87.65	88.89	88.97	89.44	89.54
S/P	2.07	2.22	2.33	2.38	2.47
S/C	2.24	2.42	2.57	2.68	2.80
P/C	1.08	1.09	1.10	1.13	1.14
P Eff	0.44	0.49	0.54	0.57	0.60
C Eff	0.44	0.50	0.54	0.57	0.60
CPU Sec	36.53	79.69	170.33	362.19	751.52

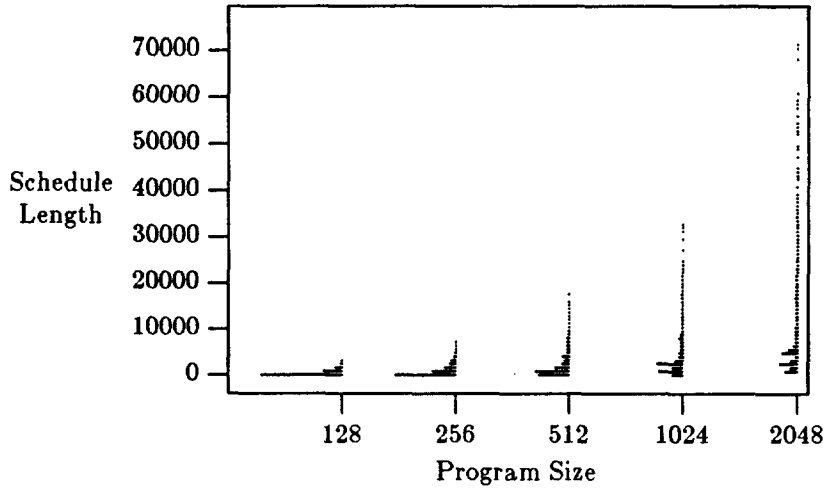
B.3.2. Figure B.26. — Scheduler 2

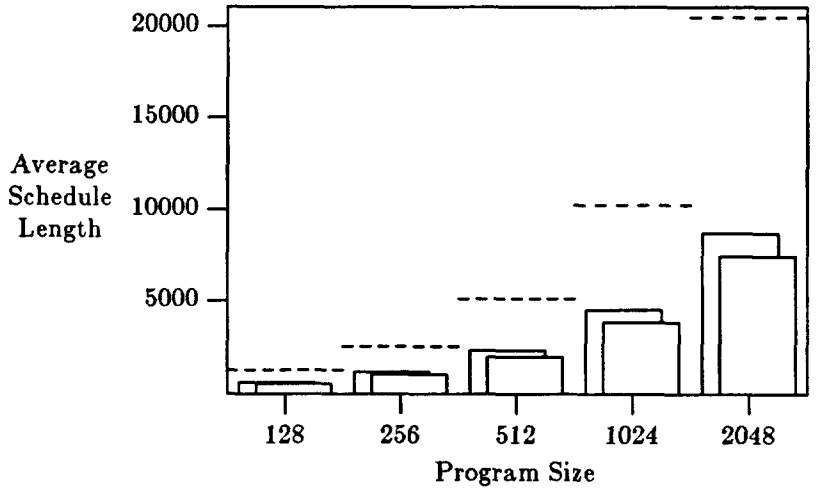




	Program Size				
	128	256	512	1024	2048
%P≤S	91.36	90.64	89.96	89.71	90.77
S/P	2.09	2.21	2.30	2.36	2.44
S/C	2.25	2.41	2.56	2.67	2.80
P/C	1.07	1.09	1.11	1.13	1.14
P Eff	0.44	0.49	0.53	0.56	0.59
C Eff	0.44	0.49	0.54	0.57	0.60
CPU Sec	36.64	80.76	173.16	382.43	840.53

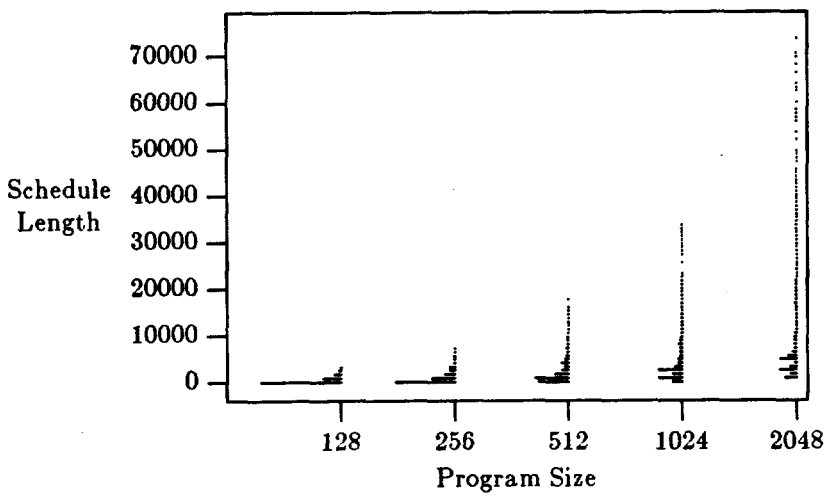
B.3.3. Figure B.27. — Scheduler 3

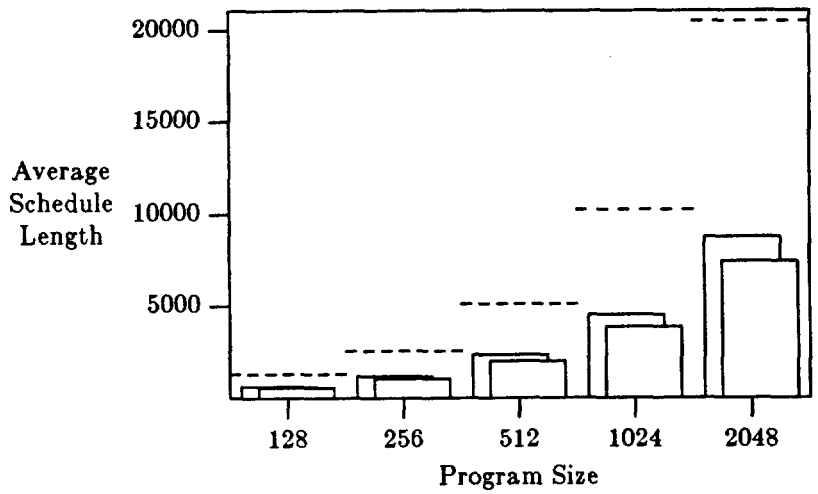




	Program Size				
	128	256	512	1024	2048
%P≤S	88.07	88.58	88.64	88.75	89.42
S/P	2.02	2.17	2.18	2.27	2.36
S/C	2.28	2.44	2.58	2.67	2.76
P/C	1.13	1.13	1.19	1.18	1.17
P Eff	0.44	0.49	0.54	0.56	0.59
C Eff	0.45	0.50	0.54	0.57	0.60
CPU Sec	38.91	86.89	197.48	458.79	1077.15

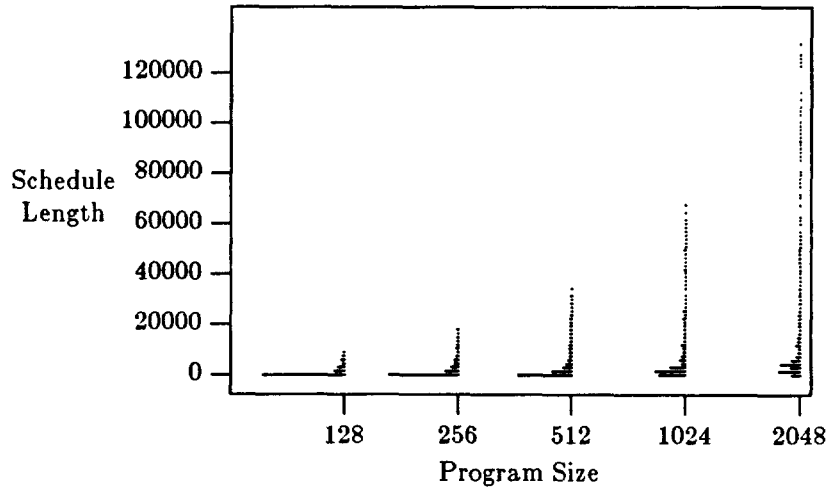
B.3.4. Figure B.28. — Scheduler 4

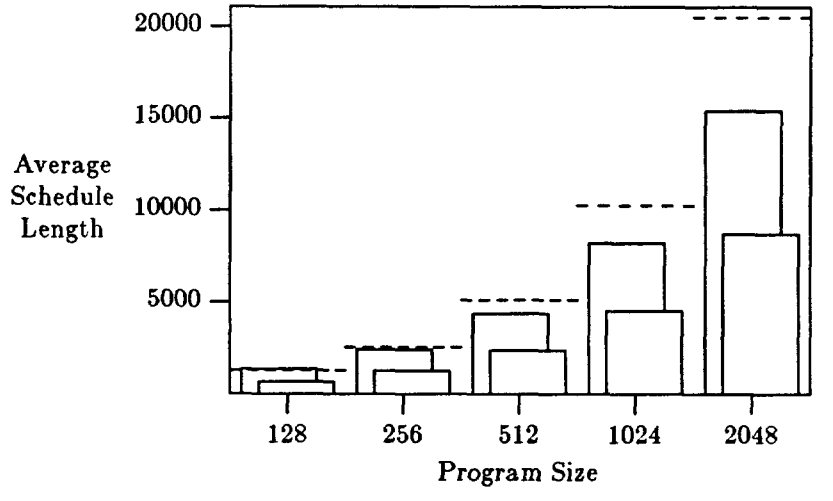




	Program Size				
	128	256	512	1024	2048
%P≤S	88.20	88.58	88.97	89.09	89.89
S/P	2.02	2.14	2.19	2.26	2.34
S/C	2.28	2.45	2.58	2.67	2.76
P/C	1.13	1.14	1.18	1.18	1.18
P Eff	0.44	0.49	0.54	0.56	0.59
C Eff	0.45	0.50	0.54	0.57	0.60
CPU Sec	41.81	99.53	255.76	714.34	2227.39

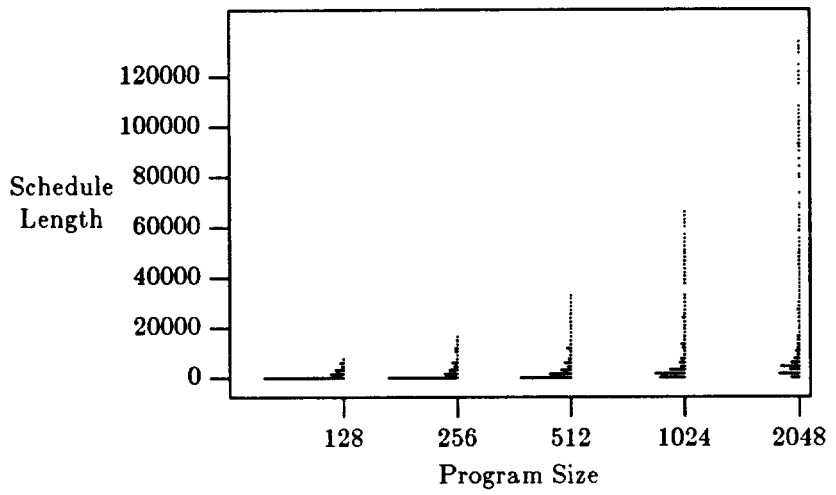
B.3.5. Figure B.29. — Scheduler 5

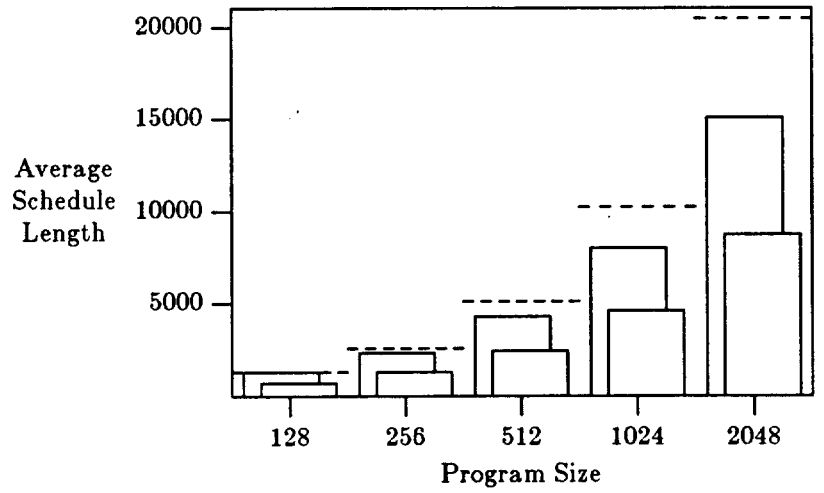




	Program Size				
	128	256	512	1024	2048
%P≤S	73.25	74.38	76.63	78.05	78.89
S/P	0.95	1.06	1.17	1.25	1.33
S/C	1.94	2.06	2.17	2.26	2.36
P/C	2.05	1.94	1.86	1.81	1.77
P Eff	0.39	0.44	0.48	0.51	0.54
C Eff	0.41	0.46	0.50	0.53	0.55
CPU Sec	0.26	0.62	1.43	3.45	8.82

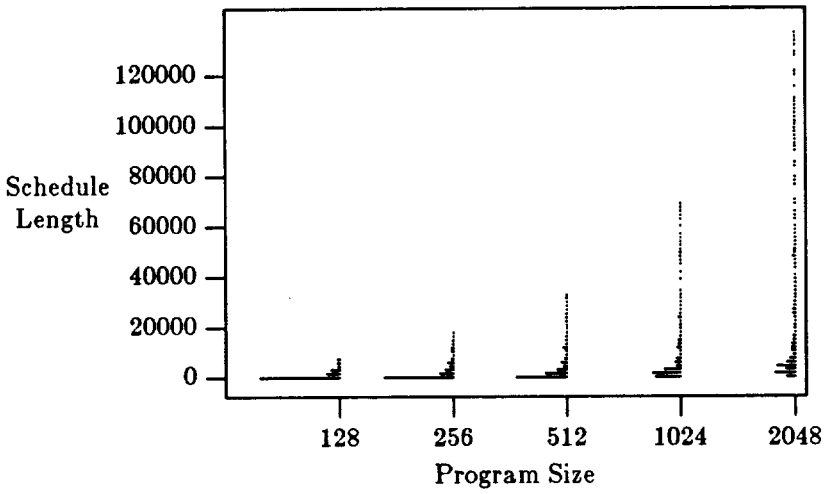
B.3.6. Figure B.30. — Scheduler 6

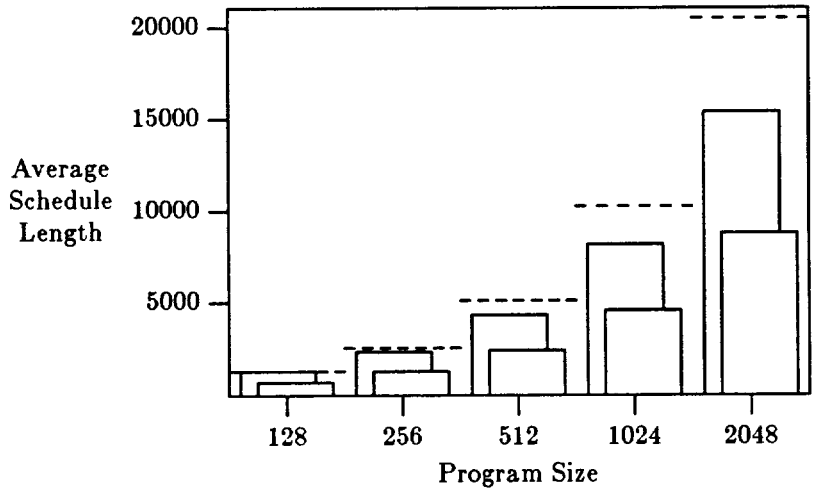




	Program Size				
	128	256	512	1024	2048
%P≤S	72.70	74.69	77.20	79.01	79.66
S/P	1.01	1.12	1.20	1.28	1.36
S/C	1.88	2.02	2.13	2.24	2.34
P/C	1.87	1.80	1.77	1.75	1.73
P Eff	0.37	0.42	0.46	0.50	0.53
C Eff	0.39	0.44	0.48	0.52	0.55
CPU Sec	3.20	7.28	17.77	47.95	148.28

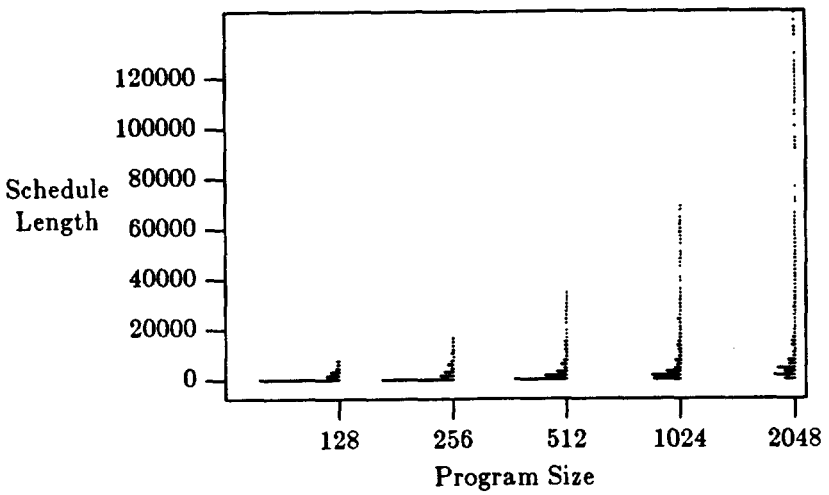
B.3.7. Figure B.31. — Scheduler 7

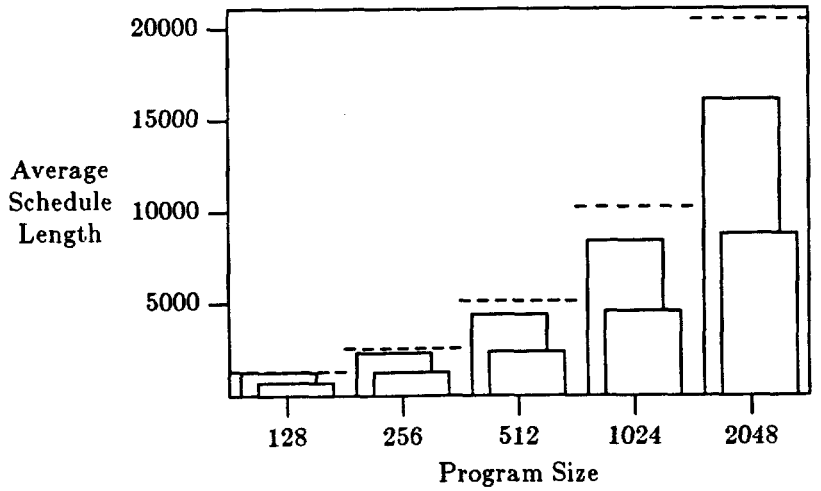




	Program Size				
	128	256	512	1024	2048
%P≤S	72.70	74.38	76.87	78.46	79.54
S/P	0.99	1.10	1.18	1.26	1.33
S/C	1.88	2.01	2.13	2.23	2.34
P/C	1.90	1.84	1.80	1.78	1.76
P Eff	0.37	0.42	0.46	0.49	0.53
C Eff	0.39	0.44	0.48	0.51	0.54
CPU Sec	2.97	6.31	13.16	27.75	57.75

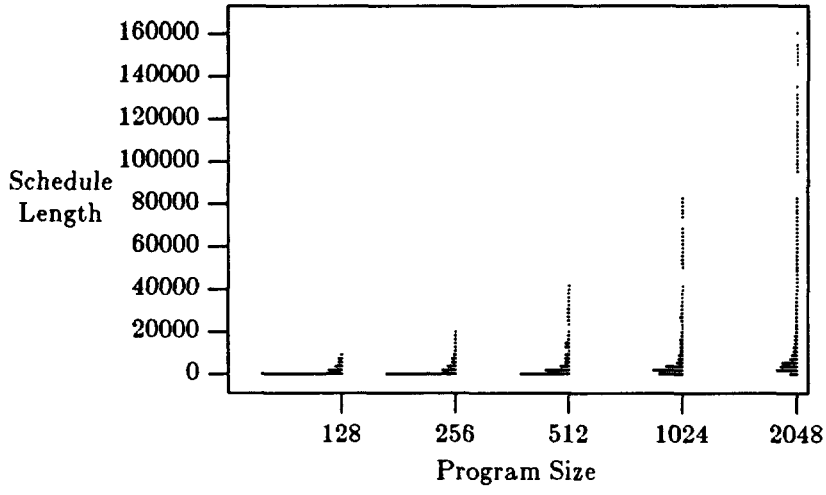
B.3.8. Figure B.32. — Scheduler 8

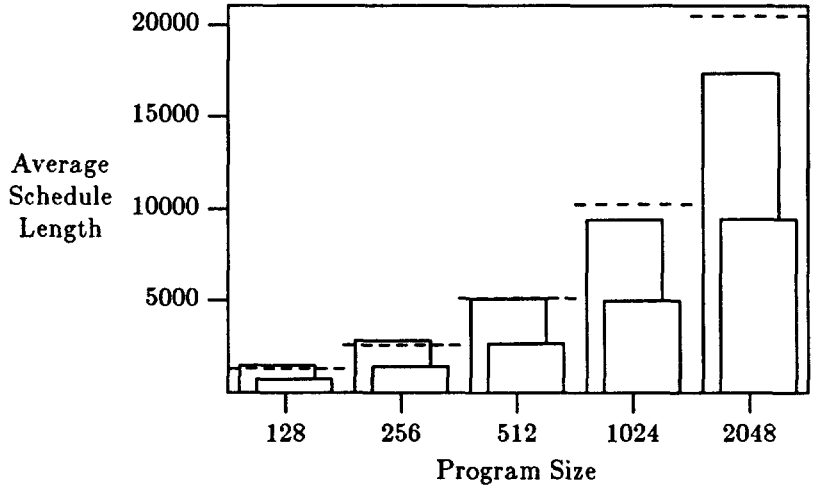




	Program Size				
	128	256	512	1024	2048
%P≤S	73.11	75.82	77.70	78.94	79.89
S/P	1.02	1.11	1.17	1.22	1.27
S/C	1.91	2.05	2.16	2.25	2.33
P/C	1.87	1.84	1.84	1.84	1.84
P Eff	0.38	0.43	0.47	0.50	0.53
C Eff	0.40	0.45	0.49	0.52	0.55
CPU Sec	4.64	10.97	23.79	55.76	130.44

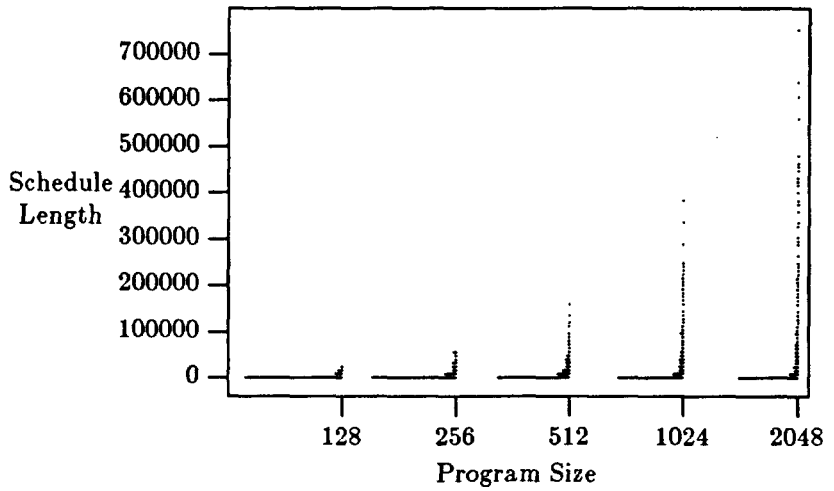
B.3.9. Figure B.33. — Scheduler 9

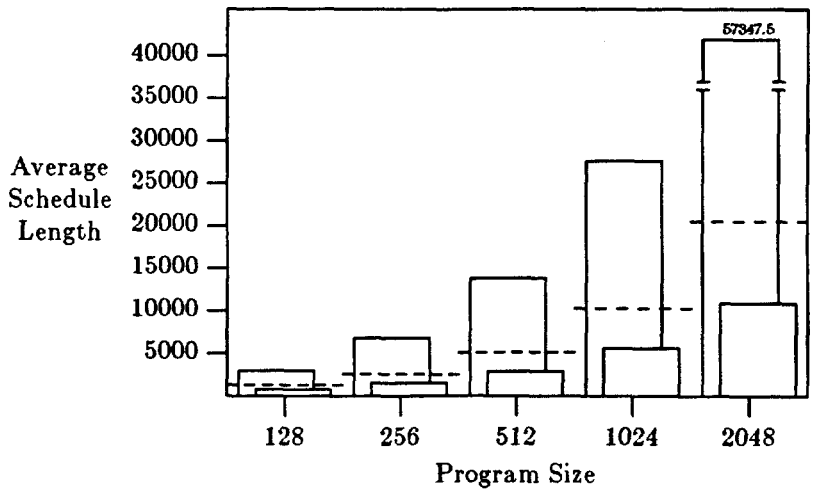




	Program Size				
	128	256	512	1024	2048
%P≤S	70.92	72.84	74.73	77.09	78.78
S/P	0.88	0.92	1.01	1.09	1.18
S/C	1.76	1.82	1.95	2.06	2.17
P/C	2.00	1.99	1.92	1.89	1.84
P Eff	0.32	0.33	0.38	0.42	0.46
C Eff	0.34	0.36	0.40	0.44	0.48
CPU Sec	0.11	0.27	0.59	1.27	2.68

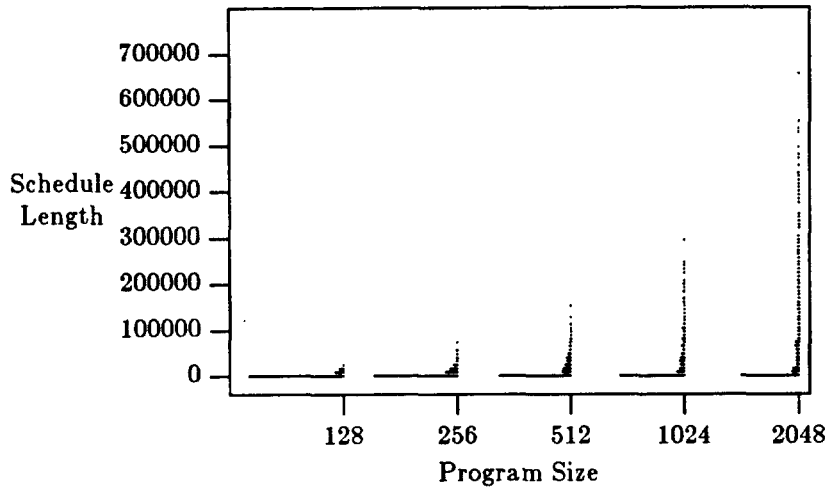
B.3.10. Figure B.34. — Scheduler 10

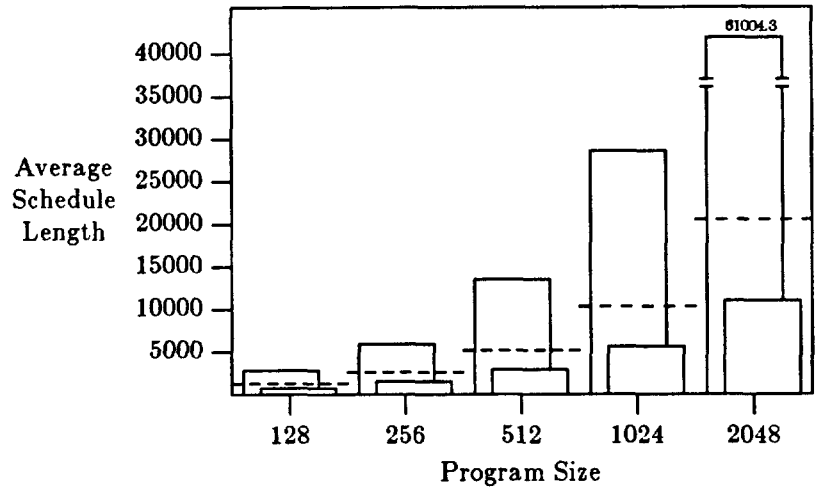




	Program Size				
	128	256	512	1024	2048
%P≤S	60.77	60.49	60.82	62.41	62.79
S/P	0.44	0.38	0.37	0.37	0.36
S/C	1.72	1.76	1.80	1.83	1.89
P/C	3.90	4.66	4.87	4.95	5.28
P Eff	0.35	0.39	0.42	0.44	0.46
C Eff	0.38	0.43	0.46	0.48	0.50
CPU Sec	1.86	4.38	9.91	22.77	53.78

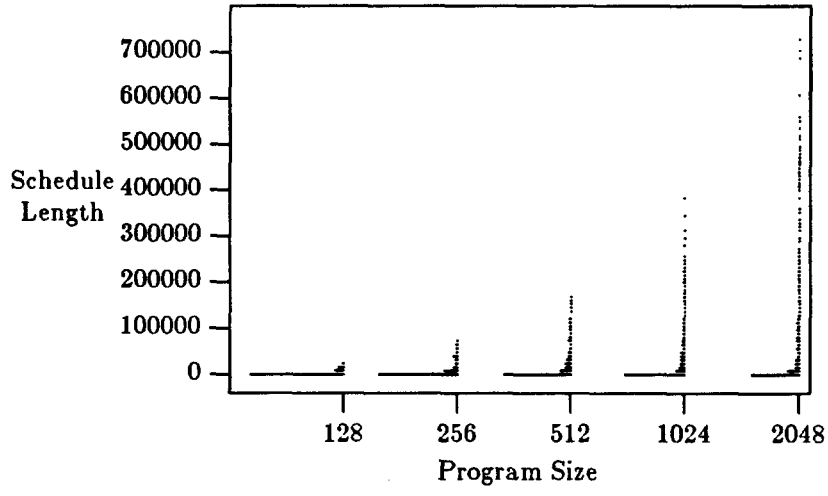
B.3.11. Figure B.35. — Scheduler 11

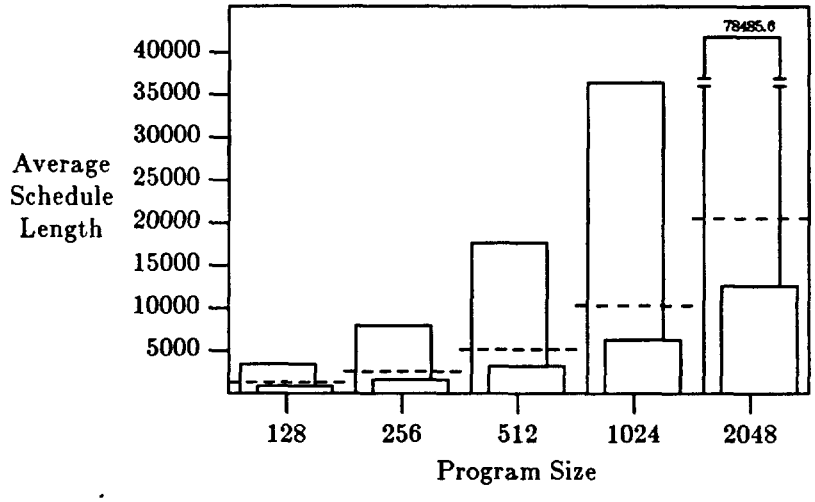




	Program Size				
	128	256	512	1024	2048
%P≤S	61.04	60.80	60.16	61.39	61.38
S/P	0.46	0.44	0.38	0.36	0.34
S/C	1.73	1.77	1.82	1.84	1.86
P/C	3.75	4.04	4.77	5.13	5.55
P Eff	0.35	0.39	0.42	0.44	0.46
C Eff	0.39	0.42	0.46	0.48	0.50
CPU Sec	2.01	5.09	13.82	42.07	138.59

B.3.12. Figure B.36. — Scheduler 12

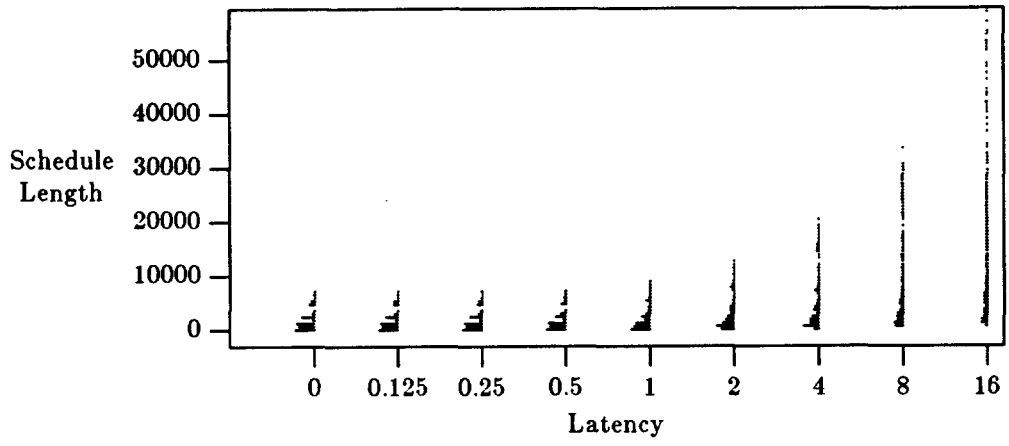


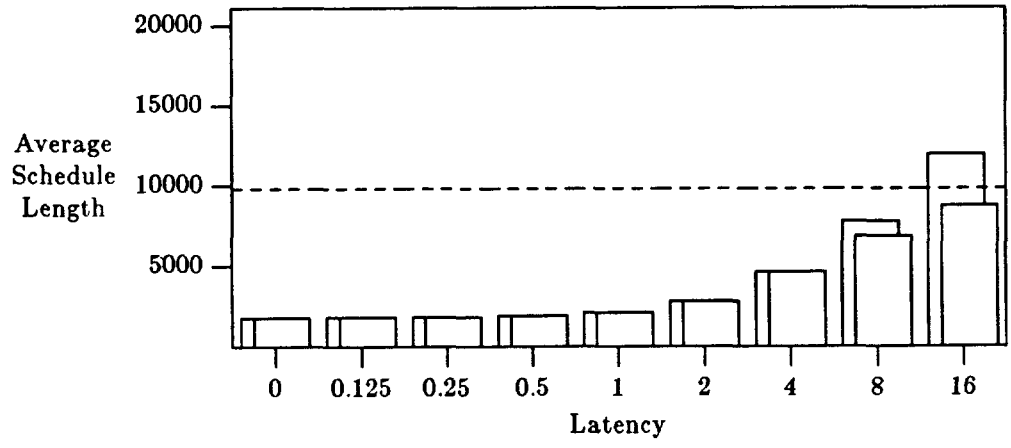


	Program Size				
	128	256	512	1024	2048
%P≤S	57.48	56.48	56.79	56.72	56.14
S/P	0.38	0.32	0.29	0.28	0.26
S/C	1.58	1.60	1.62	1.63	1.63
P/C	4.13	4.95	5.57	5.81	6.25
P Eff	0.30	0.33	0.35	0.36	0.37
C Eff	0.35	0.38	0.40	0.40	0.41
CPU Sec	1.58	3.52	7.58	16.28	34.89

B.4. Scheduler Performance By Communication Latency

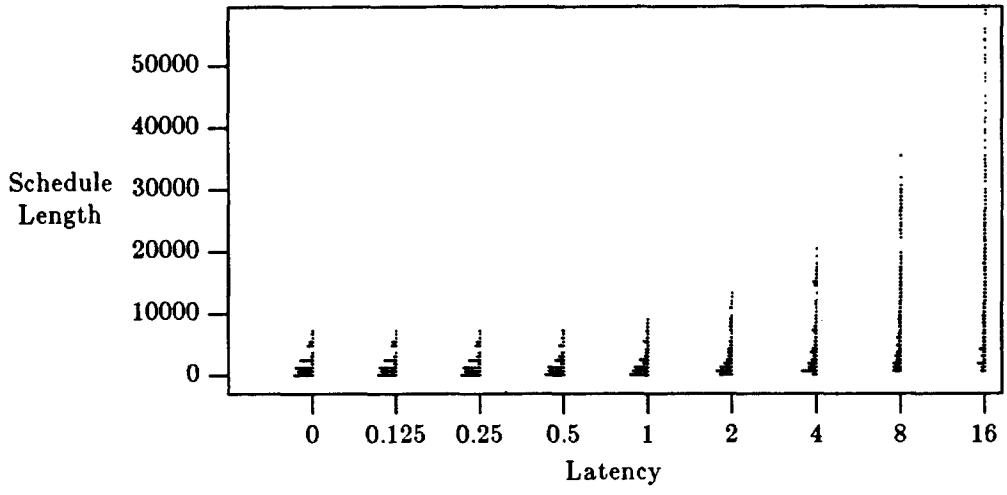
B.4.1. Figure B.37. — Scheduler 1

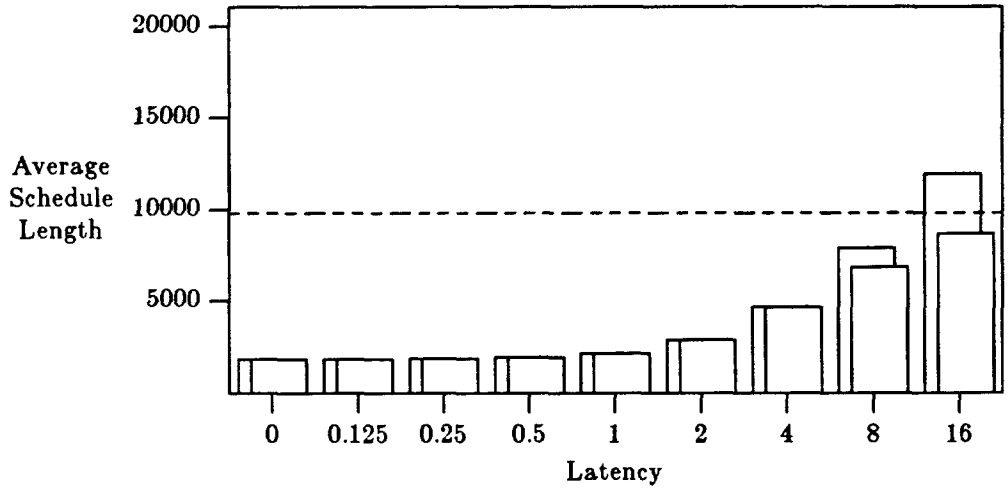




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	99.41	65.63	36.59
S/P	5.41	5.39	5.33	5.15	4.63	3.49	2.13	1.26	0.82
S/C	5.41	5.39	5.33	5.15	4.63	3.49	2.13	1.43	1.12
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.14	1.37
P Eff	0.77	0.77	0.76	0.74	0.68	0.53	0.33	0.18	0.12
C Eff	0.77	0.77	0.76	0.74	0.68	0.53	0.33	0.20	0.16
CPU Sec	236.01	367.55	377.65	377.22	373.21	364.08	357.92	351.85	331.47

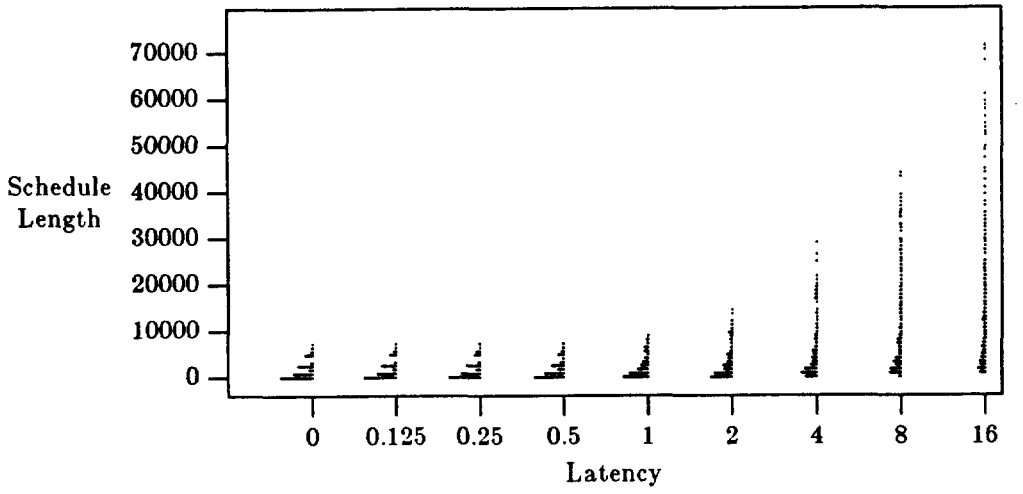
B.4.2. Figure B.38. — Scheduler 2

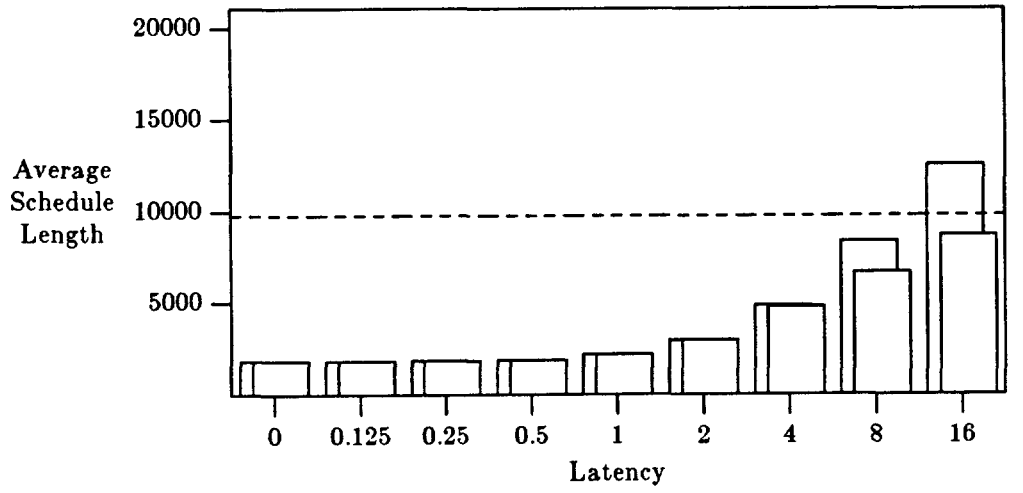




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	98.52	68.15	46.96
S/P	5.40	5.36	5.29	5.11	4.58	3.42	2.11	1.24	0.82
S/C	5.40	5.36	5.29	5.11	4.58	3.42	2.11	1.43	1.13
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.16	1.38
P Eff	0.77	0.76	0.76	0.74	0.67	0.52	0.32	0.18	0.12
C Eff	0.77	0.76	0.76	0.74	0.67	0.52	0.32	0.20	0.16
CPU Sec	269.11	398.91	408.24	408.83	405.98	397.70	388.44	379.80	354.74

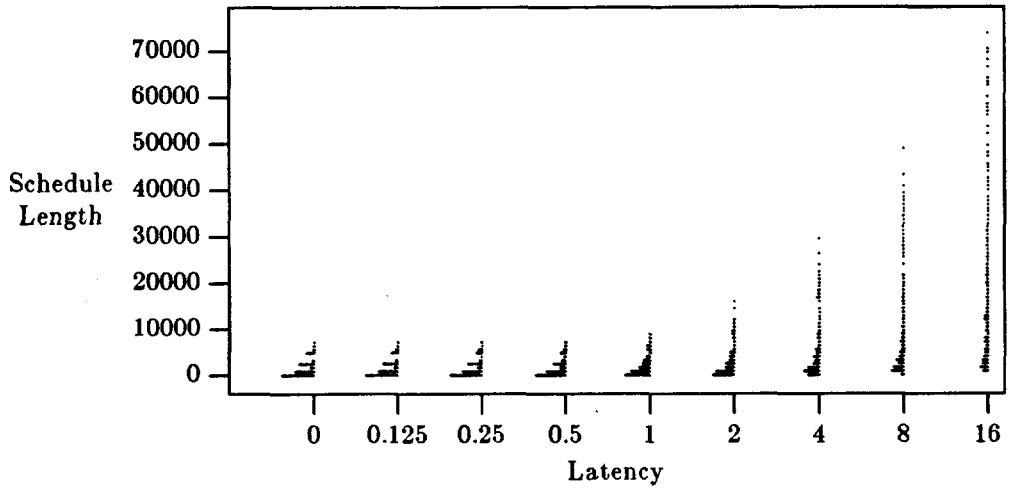
B.4.3. Figure B.39. — Scheduler 3

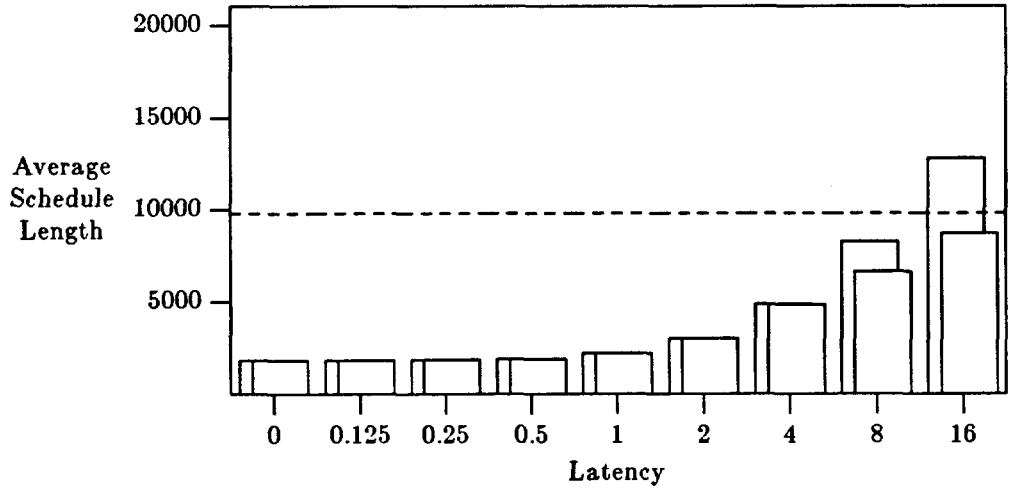




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	94.67	67.70	36.89
S/P	5.40	5.39	5.32	5.15	4.48	3.28	2.01	1.17	0.78
S/C	5.40	5.39	5.32	5.15	4.48	3.28	2.04	1.47	1.13
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.25	1.45
P Eff	0.77	0.77	0.76	0.74	0.66	0.51	0.34	0.19	0.12
C Eff	0.77	0.77	0.76	0.74	0.66	0.51	0.34	0.22	0.16
CPU Sec	289.44	783.10	688.41	513.20	404.12	379.77	380.50	389.73	399.73

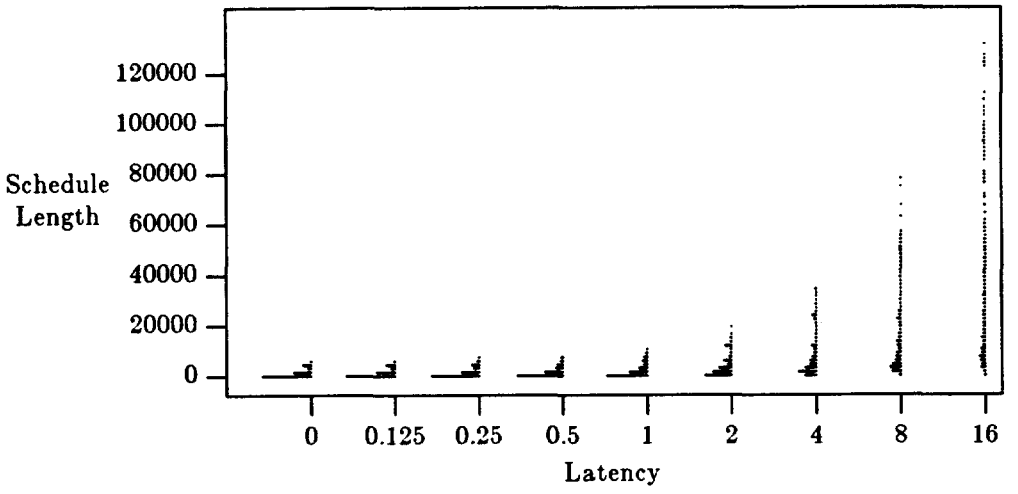
B.4.4. Figure B.40. — Scheduler 4

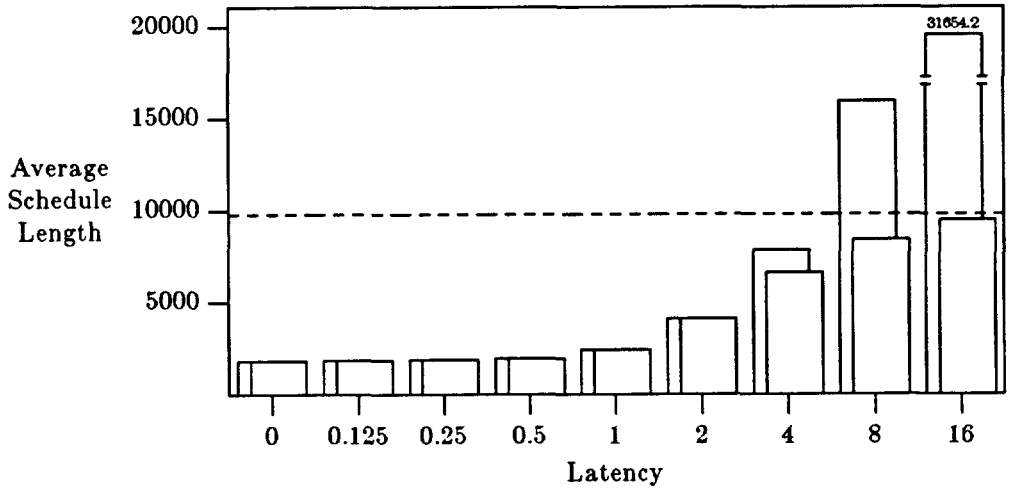




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	94.96	69.33	37.63
S/P	5.40	5.39	5.32	5.15	4.48	3.28	2.01	1.19	0.76
S/C	5.40	5.39	5.32	5.15	4.48	3.28	2.03	1.47	1.13
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.24	1.47
P Eff	0.77	0.77	0.76	0.74	0.66	0.51	0.34	0.19	0.12
C Eff	0.77	0.77	0.76	0.74	0.66	0.51	0.34	0.22	0.16
CPU Sec	693.65	1147.67	1065.19	908.85	807.54	786.65	789.18	799.60	806.52

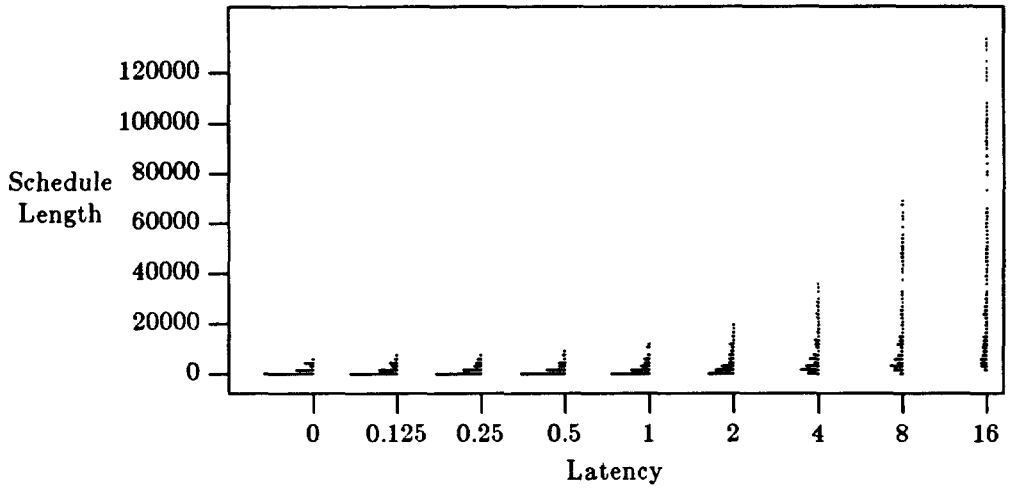
B.4.5. Figure B.41. — Scheduler 5

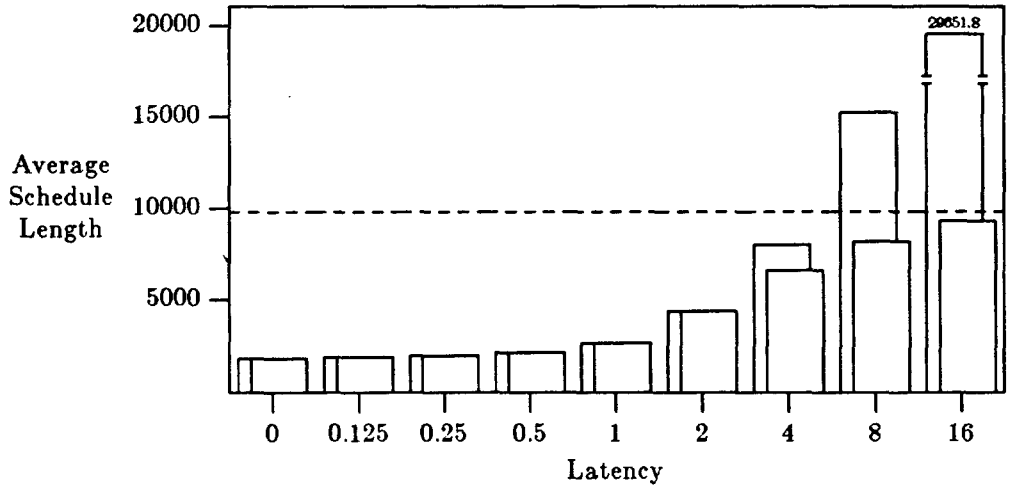




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	98.67	51.56	29.78	11.56
S/P	5.38	5.34	5.25	5.01	4.04	2.40	1.25	0.61	0.31
S/C	5.38	5.34	5.25	5.01	4.04	2.40	1.48	1.16	1.04
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.18	1.89	3.35
P Eff	0.77	0.76	0.75	0.73	0.61	0.38	0.20	0.10	0.05
C Eff	0.77	0.76	0.75	0.73	0.61	0.38	0.22	0.16	0.15
CPU Sec	3.80	3.81	3.80	3.78	3.76	3.72	3.67	3.60	3.50

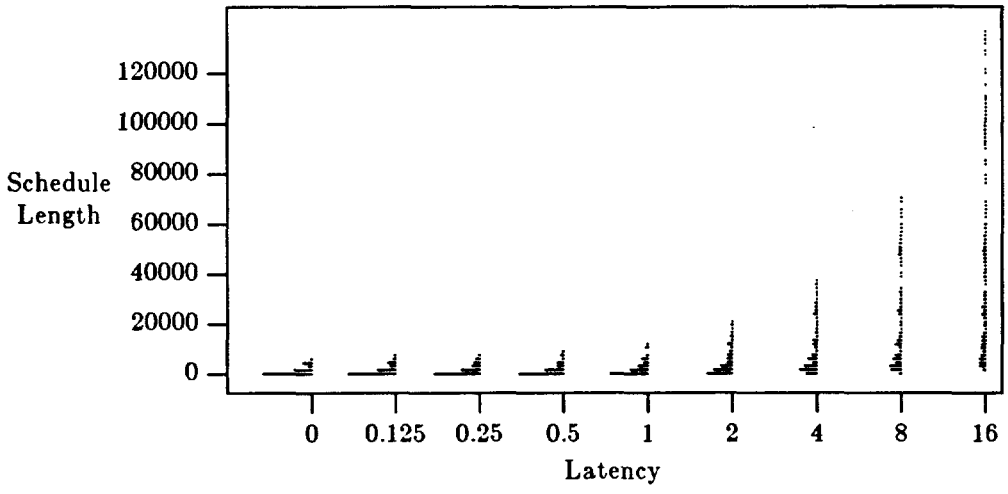
B.4.6. Figure B.42. — Scheduler 6

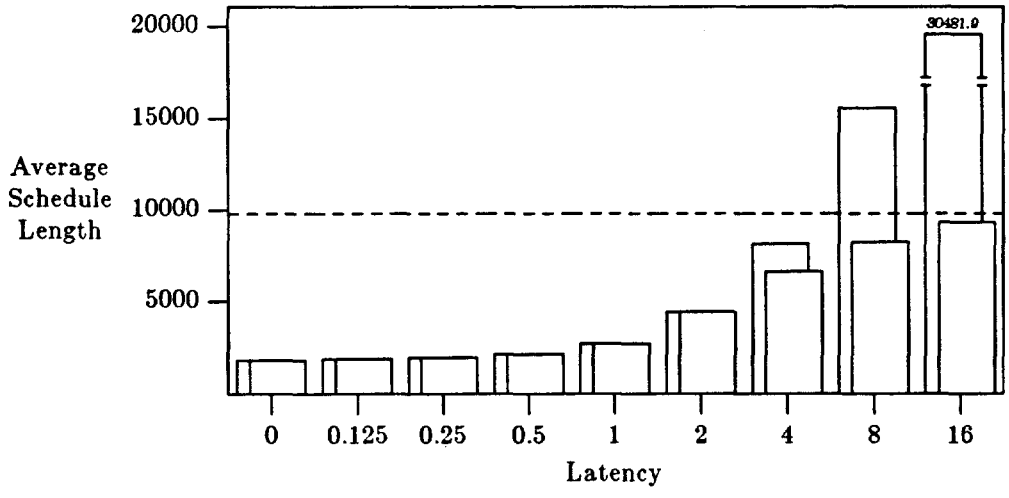




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	98.96	53.48	32.59	11.41
S/P	5.40	5.24	5.04	4.63	3.68	2.25	1.23	0.64	0.33
S/C	5.40	5.24	5.04	4.63	3.68	2.25	1.49	1.20	1.05
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.21	1.86	3.20
P Eff	0.77	0.75	0.73	0.69	0.57	0.37	0.20	0.10	0.05
C Eff	0.77	0.75	0.73	0.69	0.57	0.37	0.22	0.17	0.15
CPU Sec	48.06	58.07	58.86	58.97	59.53	59.83	60.06	60.02	59.78

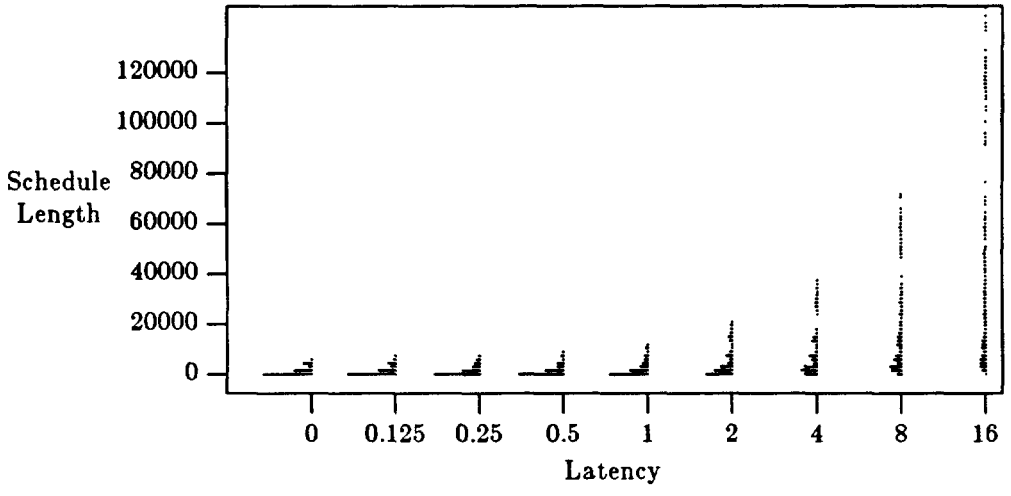
B.4.7. Figure B.43. — Scheduler 7

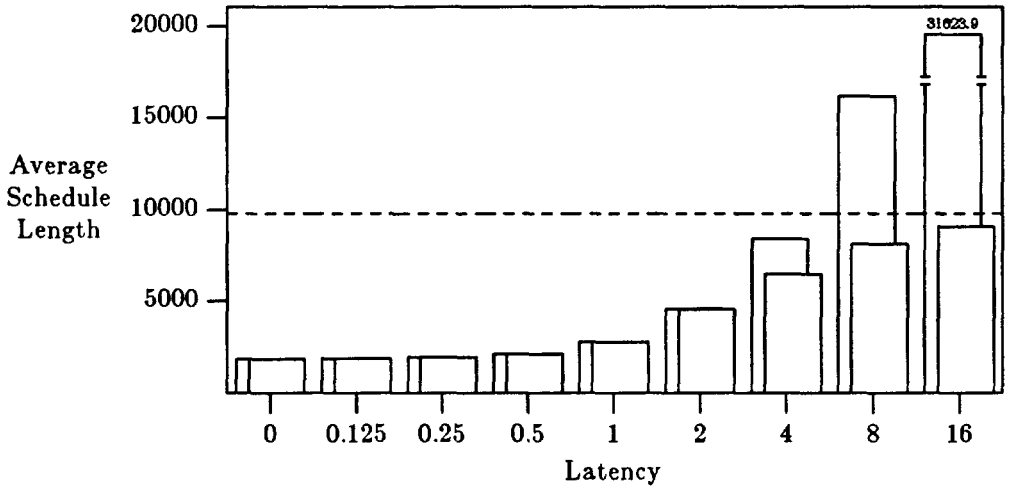




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	97.63	53.33	31.70	11.26
S/P	5.40	5.24	5.04	4.62	3.66	2.22	1.21	0.63	0.32
S/C	5.40	5.24	5.04	4.62	3.66	2.22	1.48	1.19	1.05
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.23	1.90	3.28
P Eff	0.77	0.75	0.73	0.69	0.57	0.36	0.20	0.10	0.05
C Eff	0.77	0.75	0.73	0.69	0.57	0.37	0.22	0.17	0.15
CPU Sec	17.19	26.89	27.52	27.56	27.91	28.35	28.65	28.67	28.69

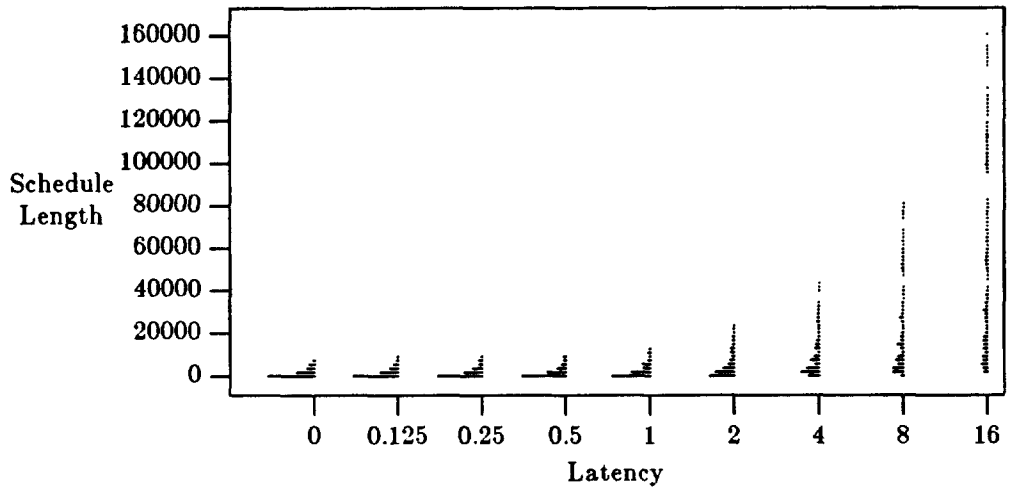
B.4.8. Figure B.44. — Scheduler 8

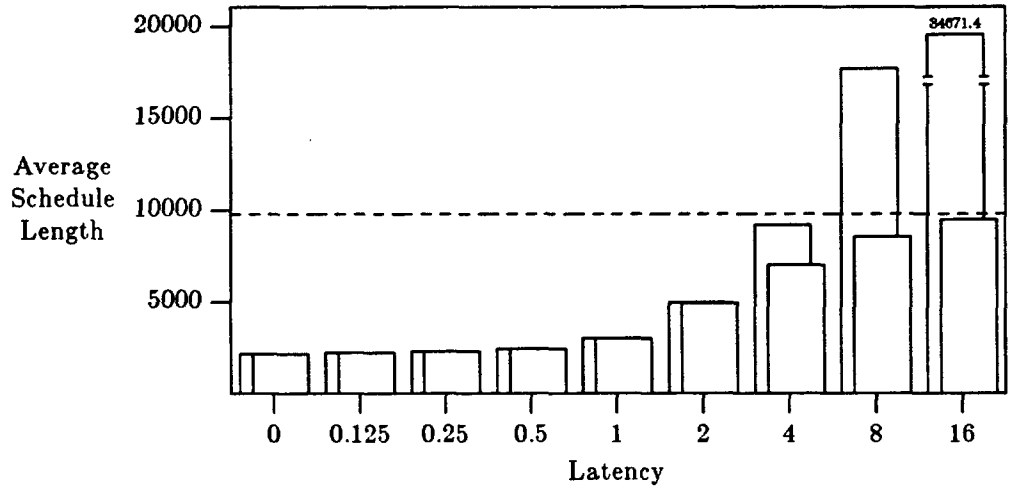




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	97.33	54.81	34.07	13.63
S/P	5.39	5.29	5.08	4.64	3.55	2.14	1.16	0.61	0.31
S/C	5.39	5.29	5.08	4.64	3.55	2.14	1.51	1.21	1.08
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.30	1.99	3.48
P Eff	0.77	0.77	0.74	0.69	0.56	0.37	0.21	0.11	0.05
C Eff	0.77	0.77	0.74	0.69	0.56	0.37	0.24	0.17	0.15
CPU Sec	33.51	93.66	83.21	61.73	50.06	47.28	47.46	47.84	48.01

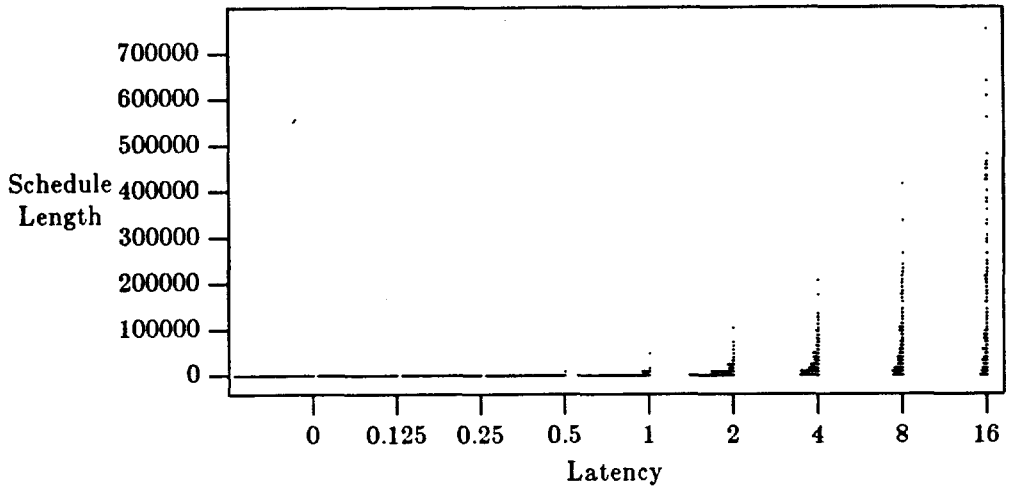
B.4.9. Figure B.45. — Scheduler 9

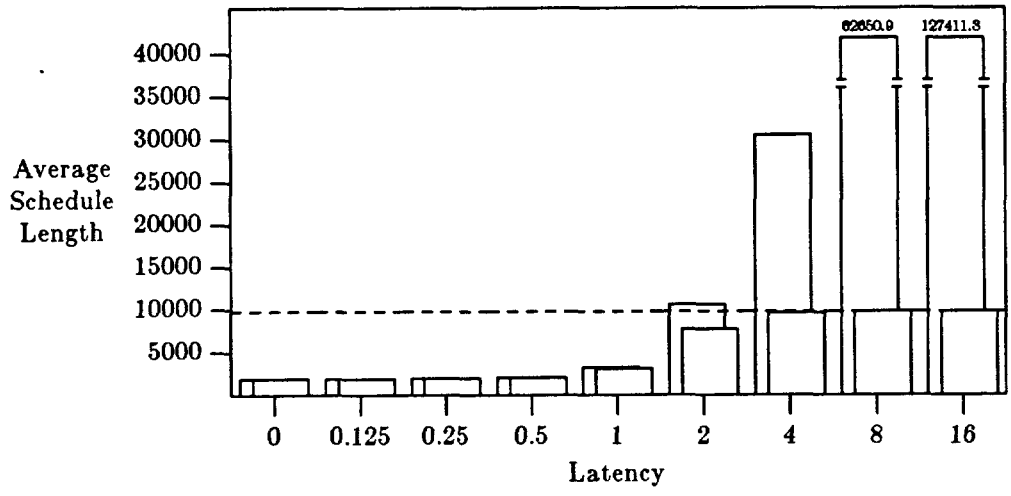




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	100.00	93.33	50.96	28.74	8.00
S/P	4.51	4.41	4.28	4.00	3.25	1.97	1.06	0.55	0.28
S/C	4.51	4.41	4.28	4.00	3.25	1.99	1.40	1.14	1.04
P/C	1.00	1.00	1.00	1.00	1.00	1.01	1.32	2.07	3.67
P Eff	0.64	0.63	0.61	0.59	0.50	0.31	0.17	0.09	0.04
C Eff	0.64	0.63	0.61	0.59	0.50	0.31	0.20	0.16	0.15
CPU Sec	1.23	1.23	1.23	1.23	1.23	1.23	1.23	1.23	1.23

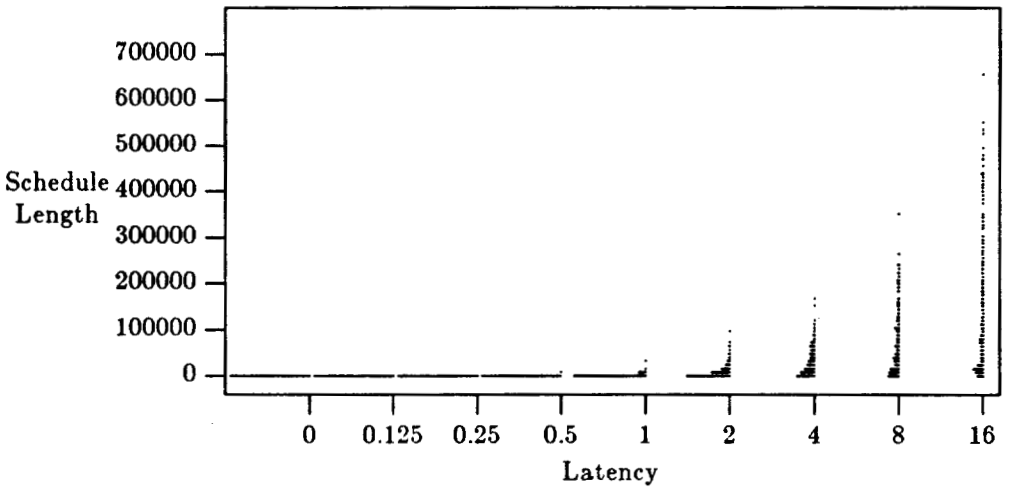
B.4.10. Figure B.46. — Scheduler 10

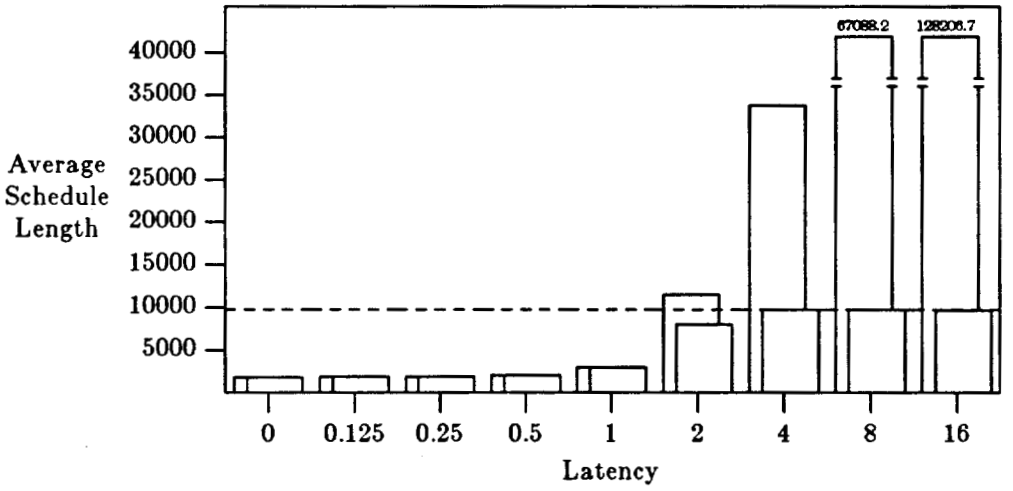




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	98.52	43.70	5.48	1.78	5.78
S/P	5.34	5.28	5.15	4.82	3.00	0.93	0.32	0.16	0.08
S/C	5.34	5.28	5.15	4.82	3.18	1.28	1.02	1.00	1.01
P/C	1.00	1.00	1.00	1.00	1.06	1.37	3.18	6.42	13.11
P Eff	0.77	0.76	0.75	0.71	0.52	0.16	0.06	0.03	0.03
C Eff	0.77	0.76	0.75	0.71	0.52	0.19	0.15	0.15	0.15
CPU Sec	19.46	27.48	27.95	27.80	27.87	25.88	19.92	17.75	16.75

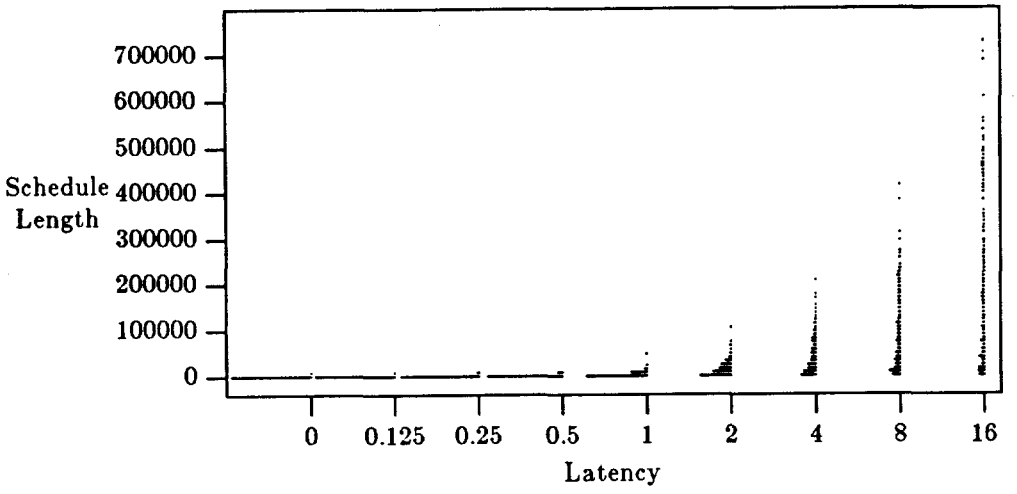
B.4.11. Figure B.47. — Scheduler 11

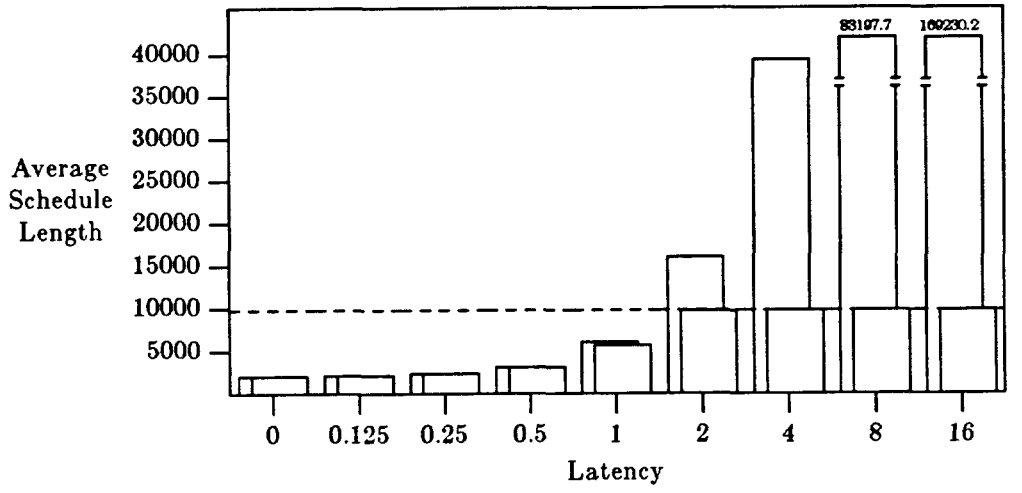




	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	99.41	38.96	1.33	2.22	7.11
S/P	5.36	5.26	5.15	4.89	3.32	0.85	0.29	0.15	0.08
S/C	5.36	5.26	5.15	4.89	3.37	1.22	1.00	1.00	1.01
P/C	1.00	1.00	1.00	1.00	1.01	1.43	3.47	6.88	13.23
P Eff	0.77	0.76	0.74	0.72	0.54	0.16	0.05	0.03	0.03
C Eff	0.77	0.76	0.74	0.72	0.54	0.19	0.15	0.15	0.15
CPU Sec	49.89	57.91	58.45	58.33	58.15	54.14	48.40	45.42	43.83

B.4.12. Figure B.48. — Scheduler 12

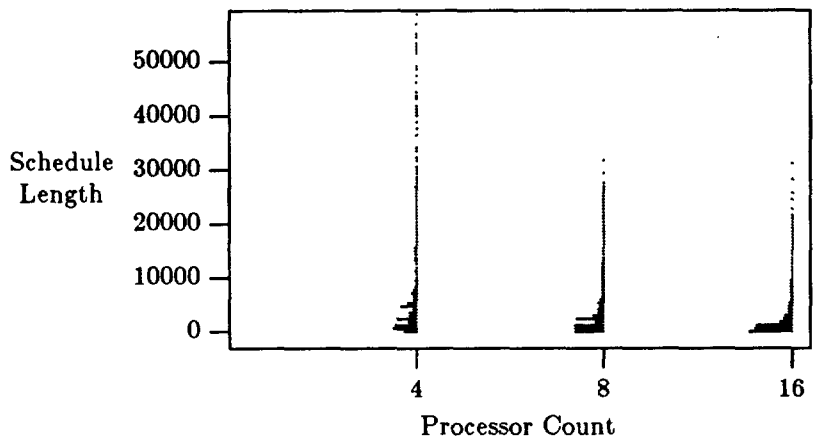


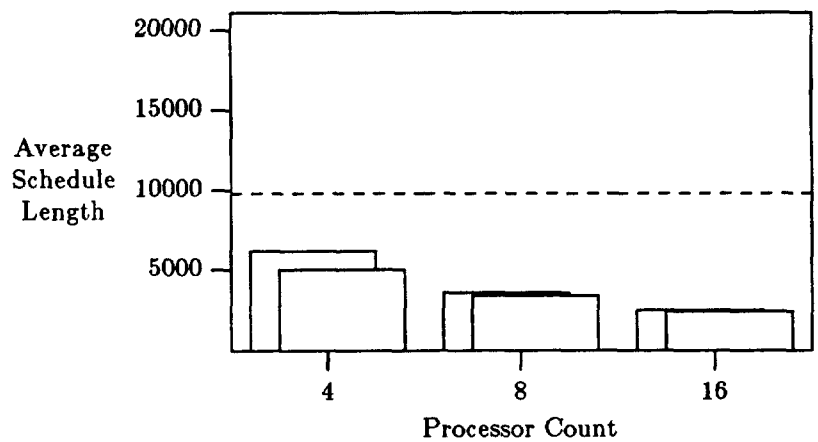


	Latency								
	0	0.125	0.25	0.5	1	2	4	8	16
%P≤S	100.00	100.00	100.00	100.00	95.56	6.96	0.00	1.33	5.78
S/P	4.90	4.66	4.28	3.20	1.62	0.61	0.25	0.12	0.06
S/C	4.90	4.66	4.28	3.20	1.73	1.01	1.00	1.00	1.01
P/C	1.00	1.00	1.00	1.00	1.07	1.65	4.02	8.52	17.41
P Eff	0.75	0.72	0.68	0.54	0.26	0.09	0.04	0.02	0.02
C Eff	0.75	0.72	0.68	0.54	0.26	0.15	0.15	0.15	0.15
CPU Sec	19.63	24.19	23.56	20.50	16.88	12.14	9.66	8.77	8.19

B.5. Scheduler Performance By Processor Count

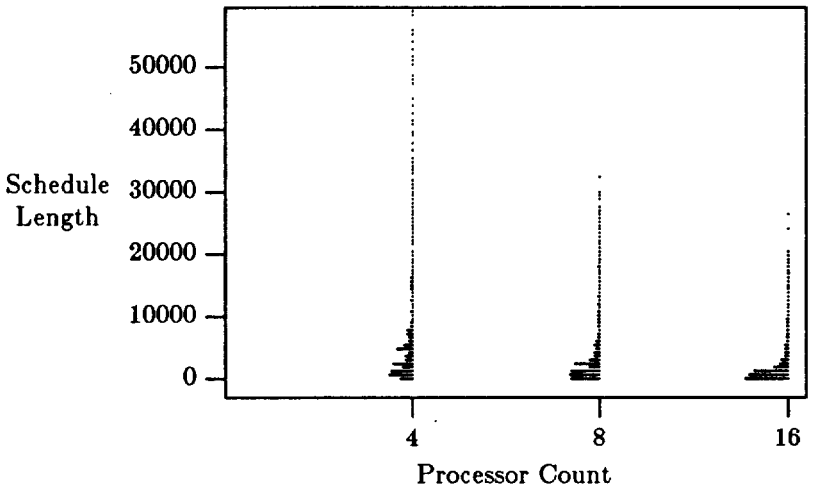
B.5.1. Figure B.49. — Scheduler 1

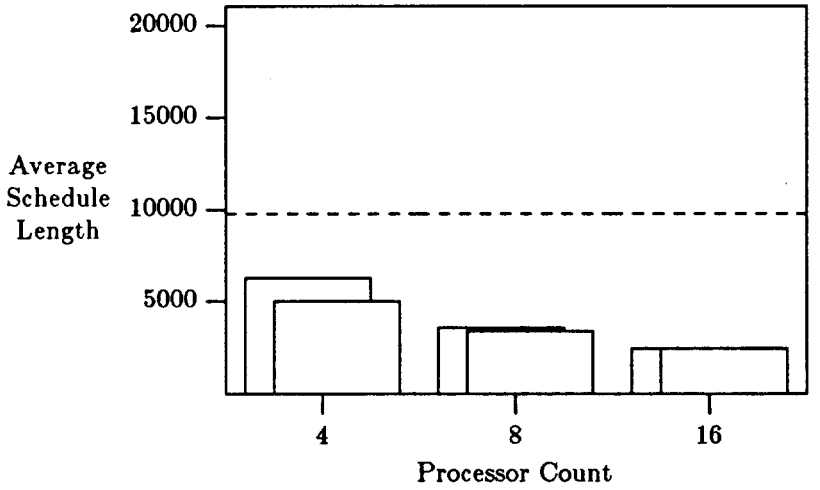




	Processor Count		
	4	8	16
%P≤S	79.90	90.47	96.84
S/P	1.59	2.75	3.98
S/C	1.95	2.90	4.05
P/C	1.23	1.05	1.02
P Eff	0.63	0.55	0.44
C Eff	0.65	0.56	0.44
CPU Sec	22.33	110.17	913.15

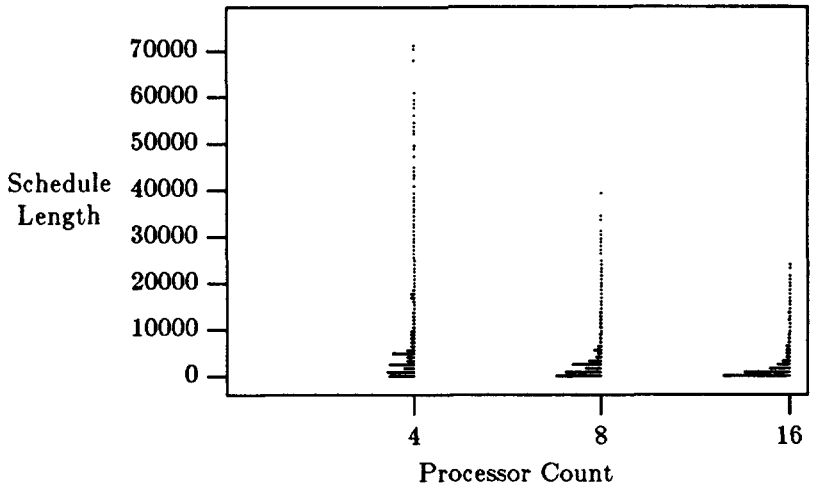
B.5.2. Figure B.50. — Scheduler 2

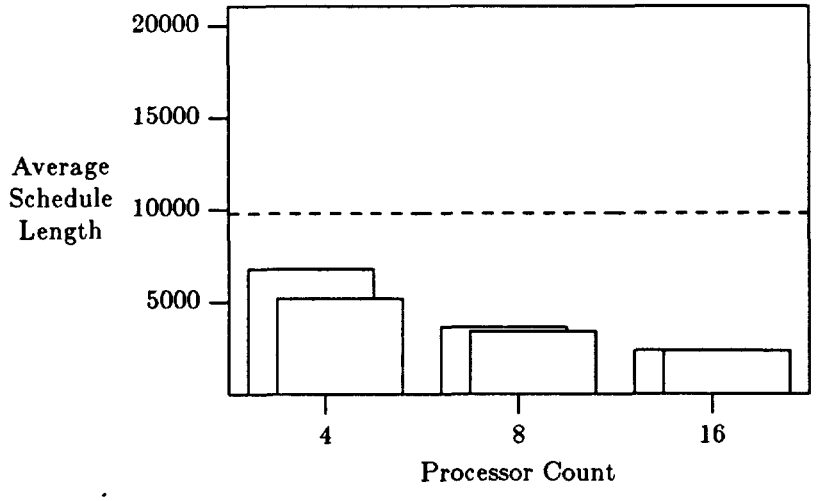




	Processor Count		
	4	8	16
%P≤S	81.09	91.75	98.37
S/P	1.56	2.75	3.98
S/C	1.95	2.89	4.01
P/C	1.25	1.05	1.01
P Eff	0.63	0.55	0.44
C Eff	0.64	0.55	0.44
CPU Sec	54.02	142.68	940.55

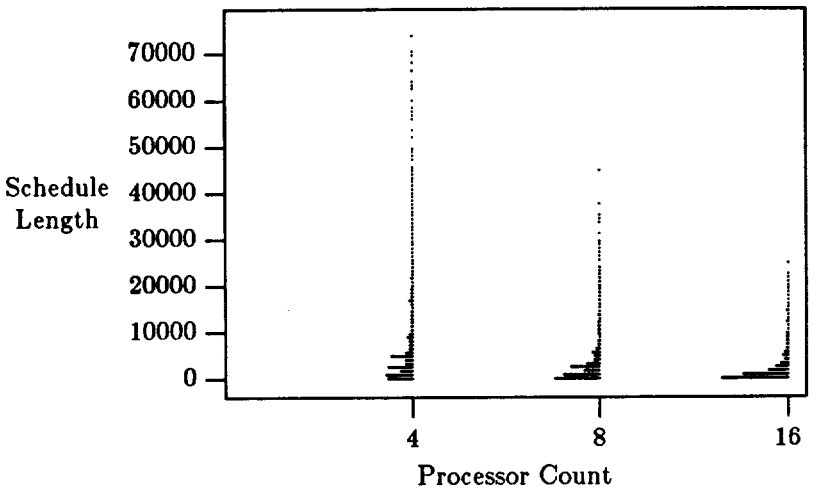
B.5.3. Figure B.51. — Scheduler 3

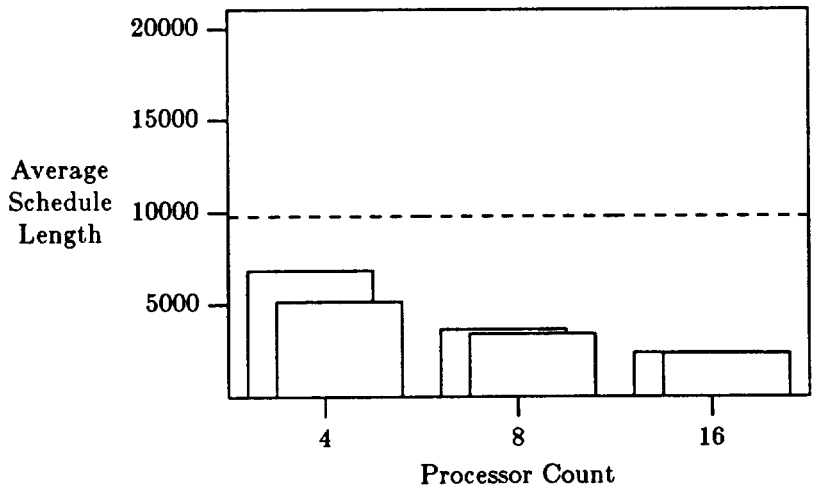




	Processor Count		
	4	8	16
%P≤S	78.62	90.37	97.43
S/P	1.45	2.69	4.13
S/C	1.89	2.90	4.17
P/C	1.31	1.08	1.01
P Eff	0.61	0.55	0.45
C Eff	0.63	0.56	0.45
CPU Sec	26.53	146.93	1235.87

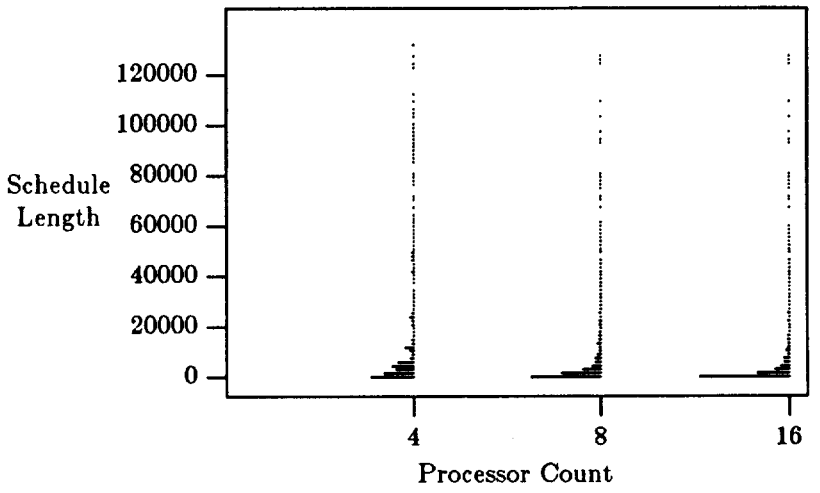
B.5.4. Figure B.52. — Scheduler 4

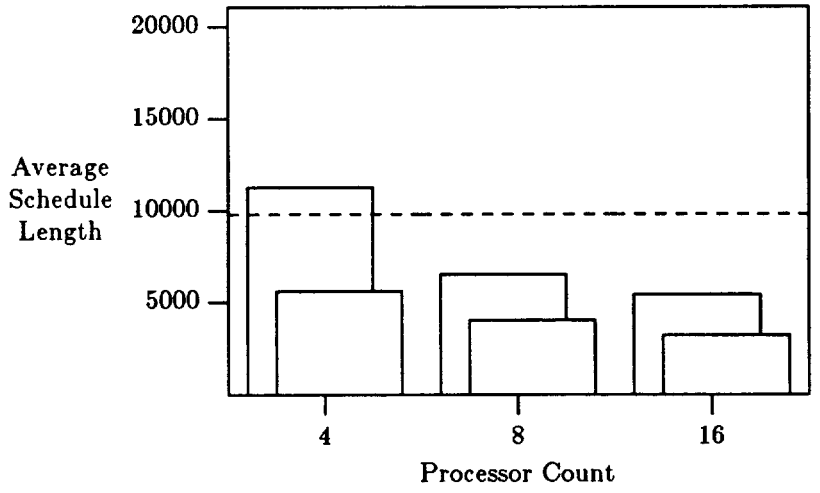




	Processor Count		
	4	8	16
%P≤S	79.36	90.67	97.28
S/P	1.43	2.70	4.12
S/C	1.90	2.90	4.16
P/C	1.32	1.07	1.01
P Eff	0.61	0.55	0.45
C Eff	0.63	0.56	0.45
CPU Sec	401.09	520.76	1679.76

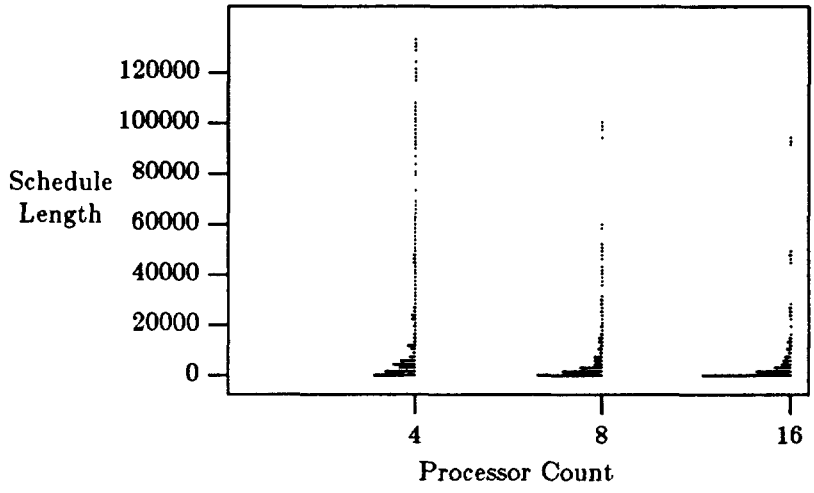
B.5.5. Figure B.53. — Scheduler 5

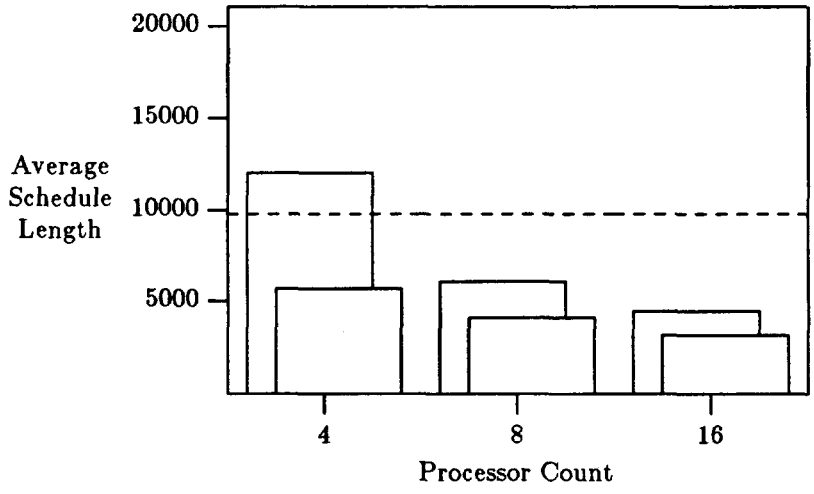




	Processor Count		
	4	8	16
%P≤S	67.31	80.00	83.21
S/P	0.87	1.51	1.82
S/C	1.76	2.43	3.04
P/C	2.02	1.61	1.67
P Eff	0.56	0.50	0.39
C Eff	0.60	0.51	0.40
CPU Sec	3.09	3.55	4.50

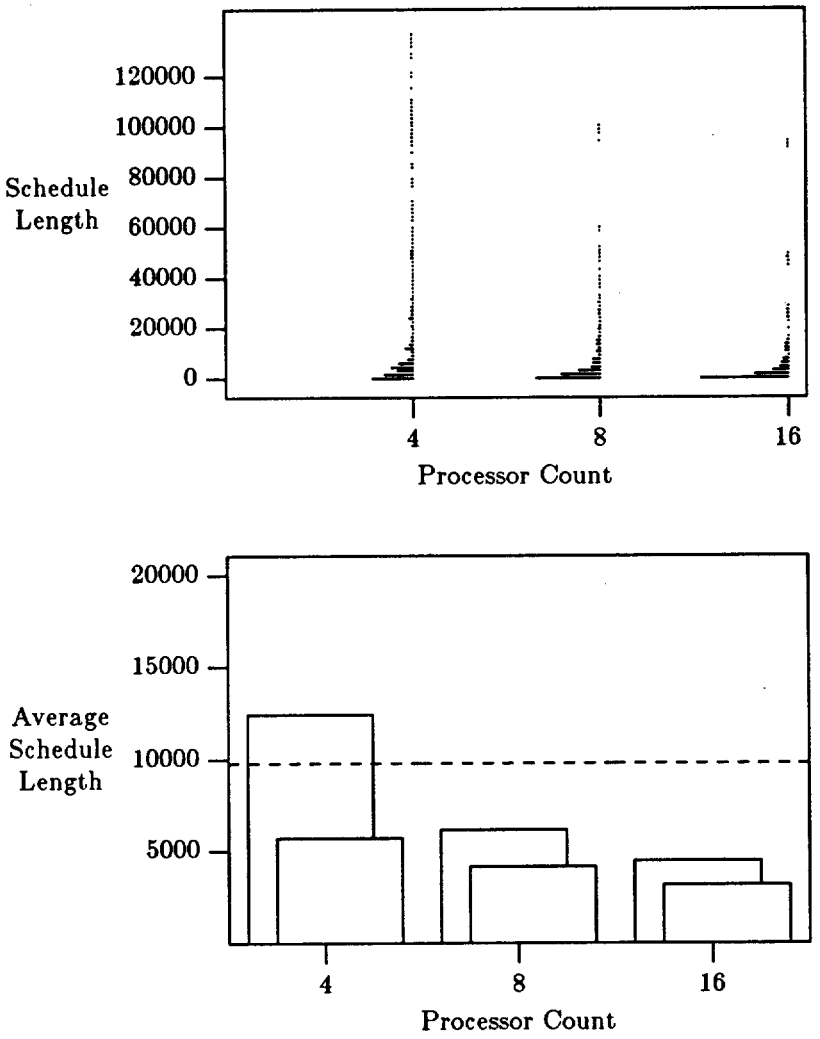
B.5.6. Figure B.54. — Scheduler 6





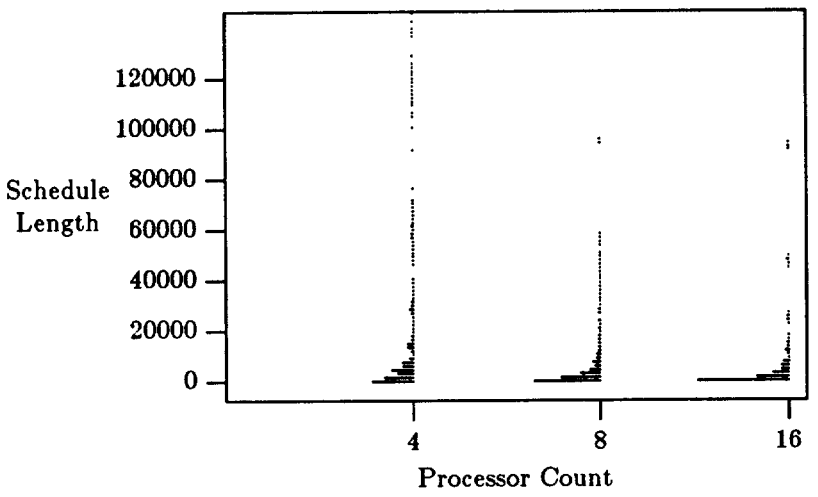
	Processor Count		
	4	8	16
%P≤S	66.37	79.75	86.02
S/P	0.81	1.62	2.19
S/C	1.73	2.38	3.10
P/C	2.13	1.47	1.41
P Eff	0.54	0.48	0.39
C Eff	0.59	0.50	0.39
CPU Sec	37.83	46.06	90.50

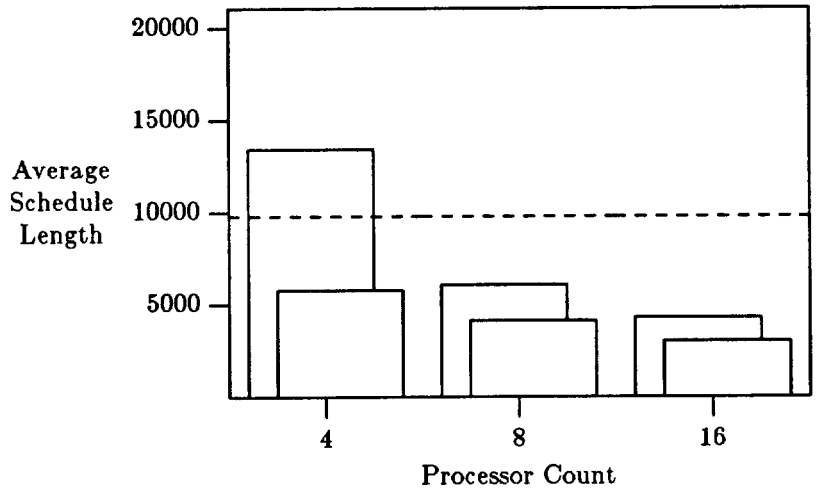
B.5.7. Figure B.55. — Scheduler 7



	Processor Count		
	4	8	16
%P≤S	65.88	79.46	85.98
S/P	0.79	1.60	2.20
S/C	1.72	2.37	3.10
P/C	2.19	1.48	1.41
P Eff	0.54	0.48	0.39
C Eff	0.59	0.49	0.39
CPU Sec	6.58	14.85	59.04

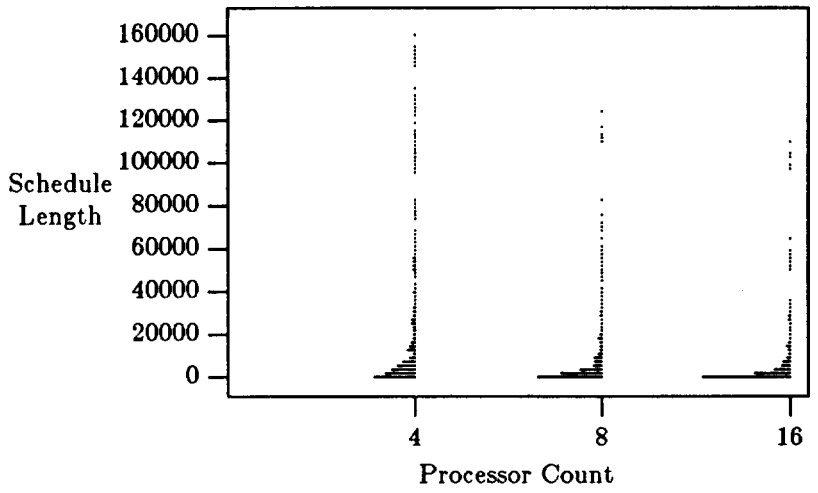
B.5.8. Figure B.56. — Scheduler 8

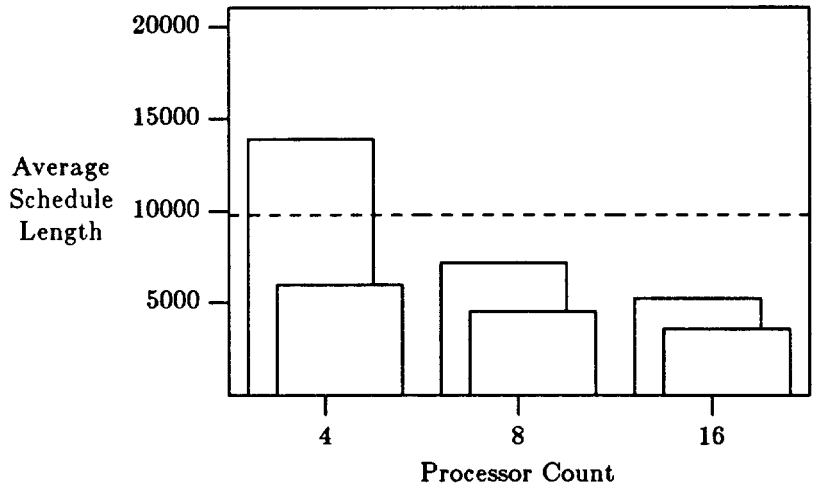




	Processor Count		
	4	8	16
%P≤S	66.27	80.44	86.57
S/P	0.73	1.62	2.29
S/C	1.70	2.38	3.24
P/C	2.33	1.47	1.42
P Eff	0.53	0.49	0.40
C Eff	0.58	0.50	0.41
CPU Sec	12.64	30.62	127.67

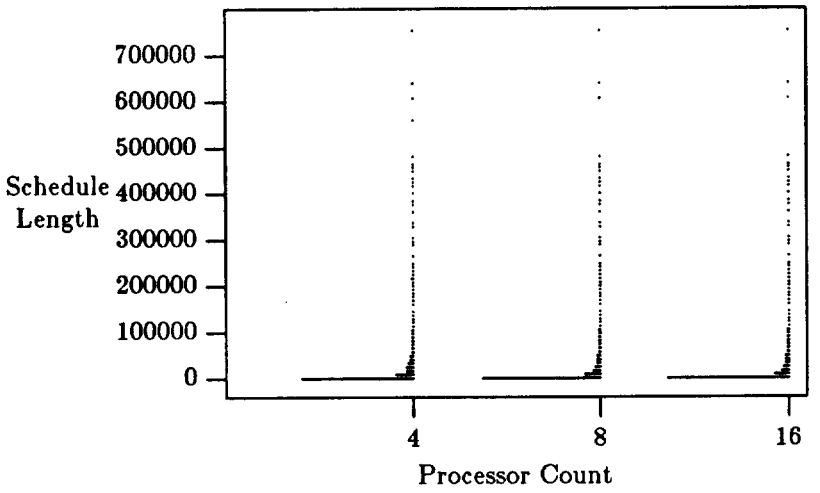
B.5.9. Figure B.57. — Scheduler 9

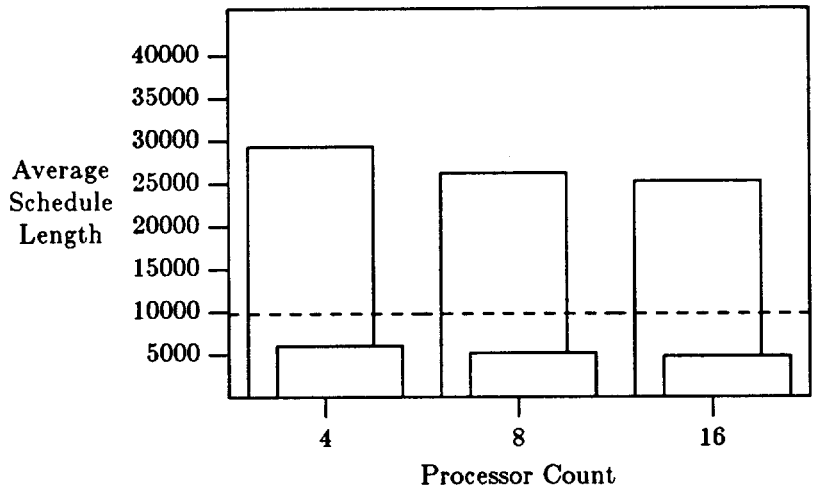




	Processor Count		
	4	8	16
%P≤S	64.54	77.78	84.69
S/P	0.70	1.37	1.88
S/C	1.64	2.17	2.75
P/C	2.33	1.59	1.46
P Eff	0.48	0.41	0.31
C Eff	0.53	0.42	0.31
CPU Sec	1.23	1.23	1.23

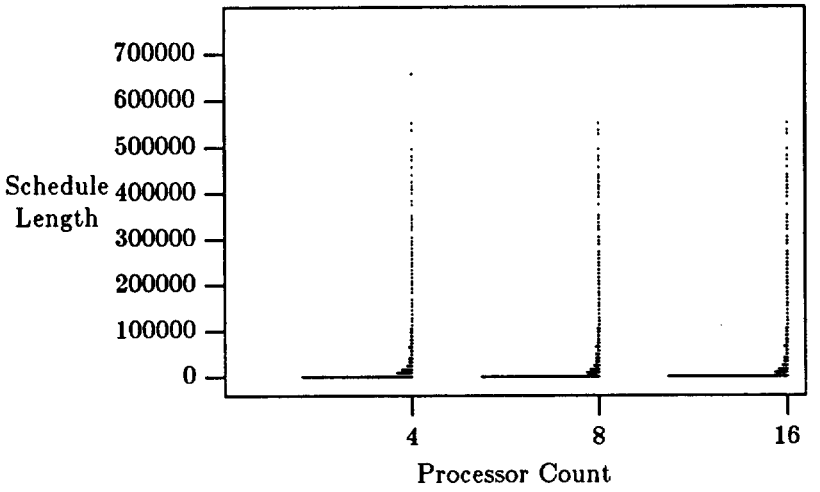
B.5.10. Figure B.58. — Scheduler 10

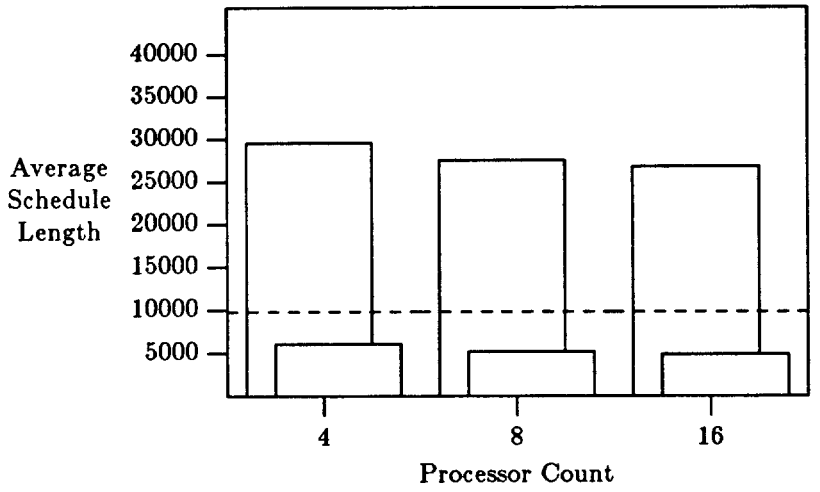




	Processor Count		
	4	8	16
%P≤S	58.96	62.37	63.75
S/P	0.33	0.37	0.39
S/C	1.63	1.92	2.06
P/C	4.90	5.15	5.31
P Eff	0.50	0.43	0.33
C Eff	0.57	0.46	0.34
CPU Sec	10.21	16.09	43.99

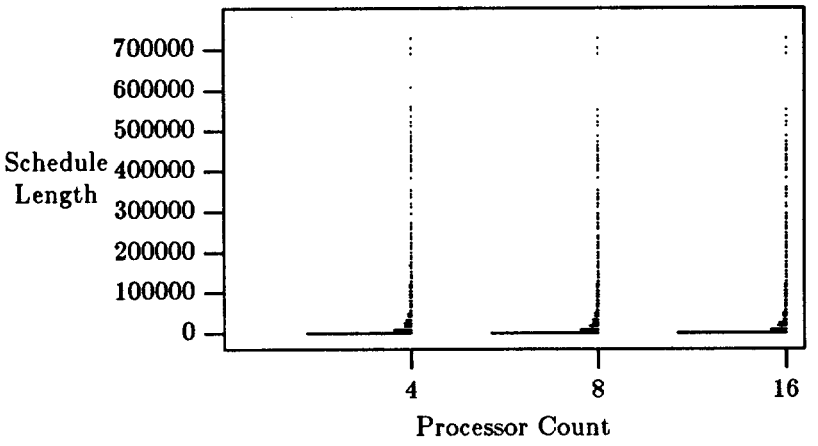
B.5.11. Figure B.59. — Scheduler 11

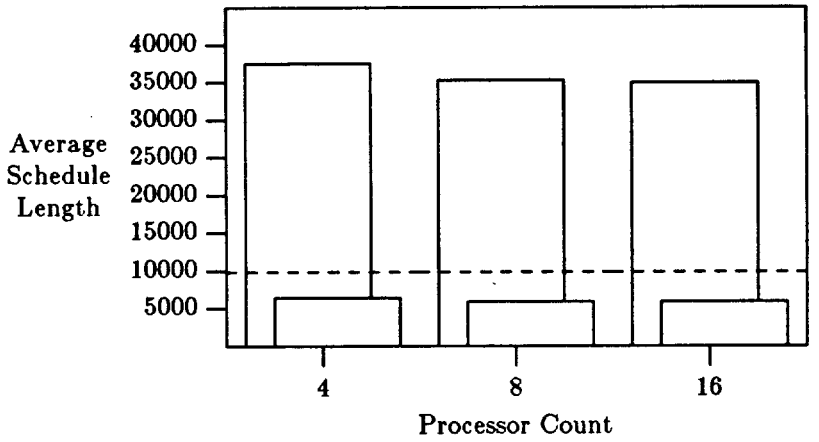




	Processor Count		
	4	8	16
%P≤S	58.77	61.83	62.42
S/P	0.33	0.36	0.37
S/C	1.63	1.92	2.04
P/C	4.91	5.39	5.58
P Eff	0.50	0.43	0.33
C Eff	0.57	0.47	0.34
CPU Sec	40.00	45.74	72.43

B.5.12. Figure B.60. — Scheduler 12





	Processor Count		
	4	8	16
%P≤S	55.70	56.64	57.53
S/P	0.26	0.28	0.28
S/C	1.54	1.67	1.68
P/C	5.91	6.02	6.01
P Eff	0.46	0.35	0.24
C Eff	0.54	0.38	0.26
CPU Sec	6.32	11.51	30.01

APPENDIX C

Comparison of Schedulers By Problem Characteristic

In this appendix the performance of different schedulers is compared for each variable in the problem space. The variables are: task distribution, average parallelism, program size, communication latency, and processor count. It shows how different schedulers respond to a given characteristic, such as program size. Each scheduler/characteristic pair uses a histogram chart, a bar chart, and a table. The histogram chart shows histograms side-by-side, to compare the distributions of schedule lengths for the different schedulers (1 through 12). The bar chart shows the average sequential schedule length (dashed line), the average length of the parallel schedules, and the average length of min(*sequential schedule, parallel schedule*), which is referred to as the *corrected schedule length.* This third item recognizes that the parallel schedules generated by the different schedulers are not always shorter than a sequential schedule, and shows the effect of selecting the shorter of the two.

The table gives specific values of interest in a numerical form. The values are:

%P≤S Percentage of parallel schedules that were shorter than a sequential schedule.

S/P The speedup gained by the parallel schedule, or $\frac{T_s}{T_p}$, where T_s is the length of a sequential schedule and T_p is the length of the parallel schedule.

S/C The speedup gained by the corrected schedule, or $\frac{T_s}{T_c}$, where T_c is the length of the corrected schedule.

P/C The speedup gained by correcting the parallel schedule for schedules which are longer than a sequential schedule, or $\frac{T_p}{T_c}$.

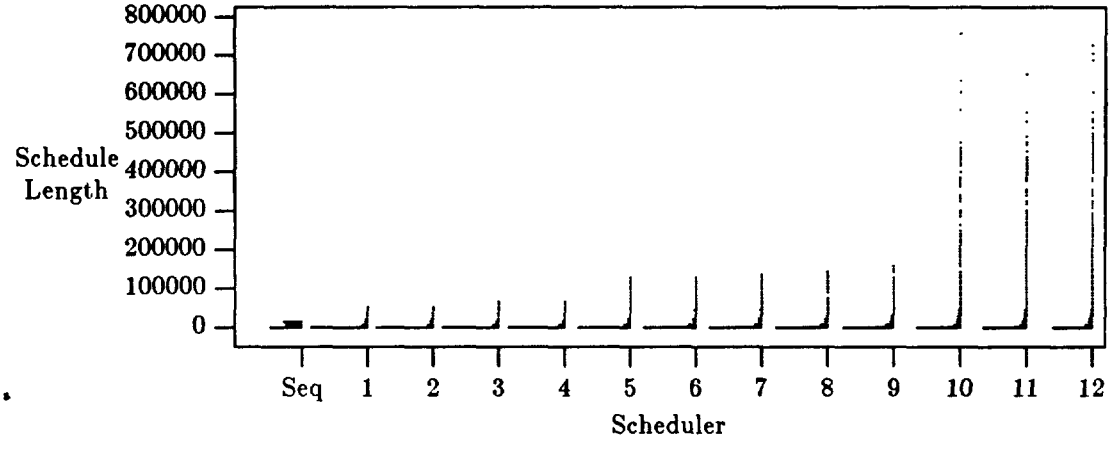
P Eff This entry gives the *parallel efficiency* of a schedule, defined as $\frac{T_s}{n \times T_p}$, where n is the number of available processors.

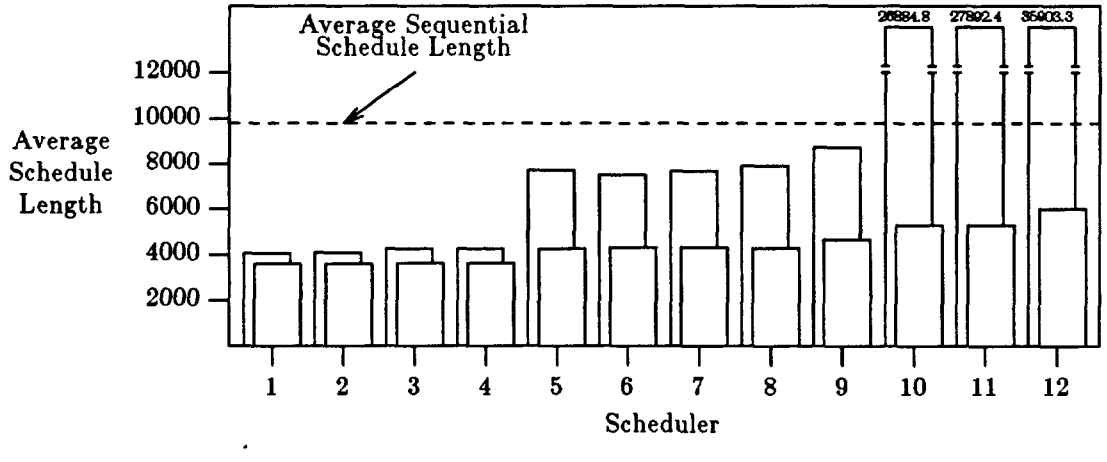
C Eff This entry gives the *corrected parallel efficiency* of a schedule, defined as $\frac{T_s}{n \times T_c}$, where n is the number of available processors.

CPU Sec This field gives the average number of CPU seconds on a Sequent Symmetry™ used to schedule the programs.

C.1. All Test Cases

C.1.1. Figure C.1. — All Tests (6075 Cases)

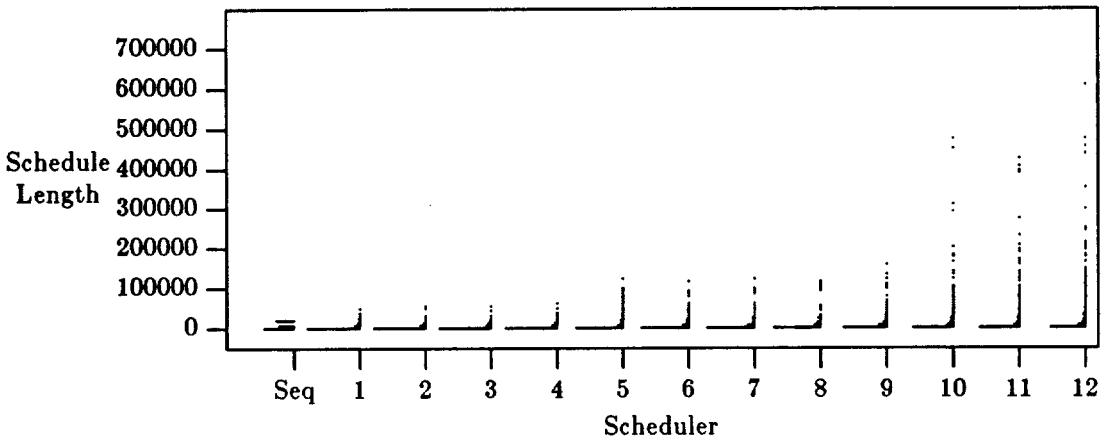


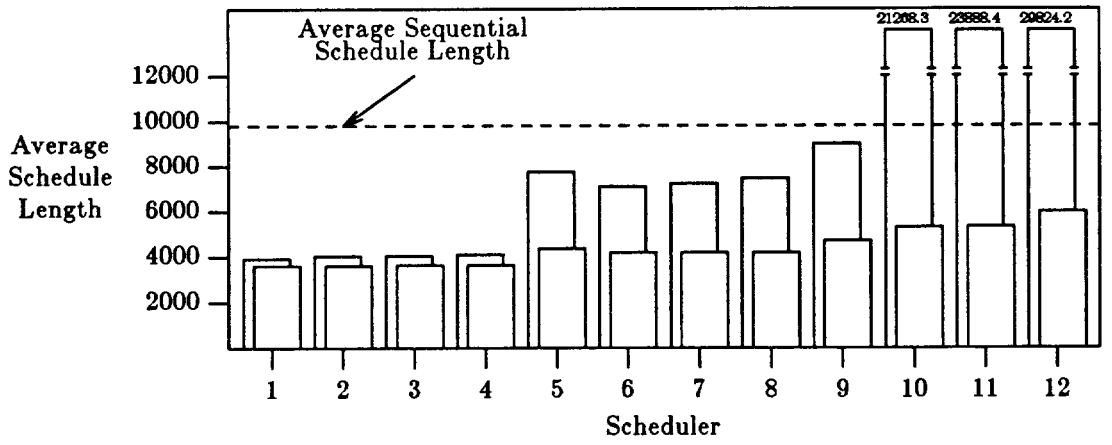


	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	89.07	90.40	88.81	89.10	76.84	77.38	77.10	77.76	75.67	61.70	61.00	56.63
S/P	2.41	2.39	2.30	2.29	1.27	1.30	1.28	1.24	1.12	0.36	0.35	0.27
S/C	2.72	2.71	2.69	2.70	2.29	2.27	2.26	2.27	2.09	1.86	1.85	1.63
P/C	1.13	1.13	1.17	1.18	1.81	1.74	1.77	1.84	1.87	5.10	5.27	5.98
P Eff	0.54	0.54	0.54	0.54	0.48	0.47	0.47	0.47	0.40	0.42	0.42	0.35
C Eff	0.55	0.54	0.55	0.55	0.50	0.49	0.49	0.50	0.42	0.46	0.46	0.39
CPU Sec	348.55	379.08	469.78	867.21	3.72	58.13	26.83	56.98	1.23	23.43	52.72	15.95

C.2. Comparison By Task Distribution

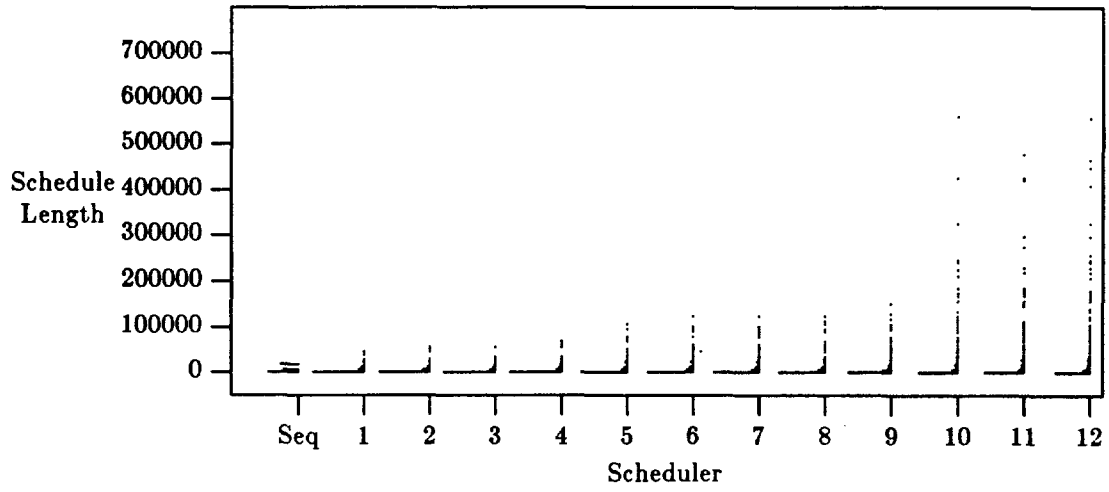
C.2.1. Figure C.2. — Distribution = 0 (675 Cases)

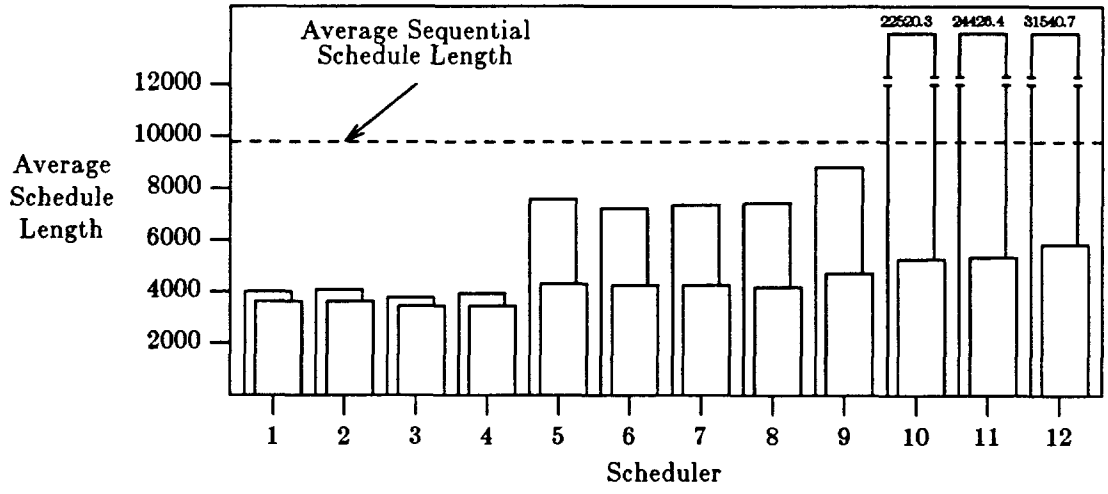




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	91.26	93.04	91.11	91.41	76.59	77.93	78.07	78.52	75.11	64.00	63.11	59.26
S/P	2.49	2.43	2.42	2.40	1.26	1.38	1.35	1.31	1.09	0.46	0.41	0.33
S/C	2.71	2.71	2.70	2.70	2.26	2.35	2.34	2.35	2.08	1.85	1.84	1.64
P/C	1.09	1.12	1.12	1.13	1.78	1.70	1.73	1.79	1.91	4.03	4.49	5.01
P Eff	0.57	0.57	0.57	0.57	0.51	0.51	0.51	0.51	0.40	0.44	0.45	0.37
C Eff	0.58	0.57	0.58	0.58	0.52	0.53	0.53	0.53	0.42	0.48	0.48	0.41
CPU Sec	324.47	352.66	419.27	798.10	2.98	54.50	25.37	50.05	1.19	19.86	46.83	14.30

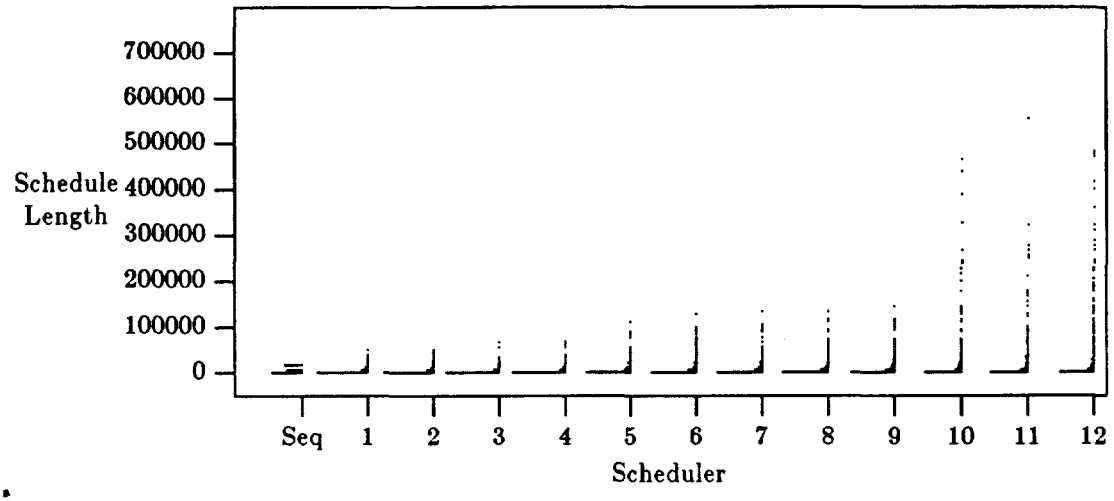
C.2.2. Figure C.3. — Distribution = 1 (675 Cases)

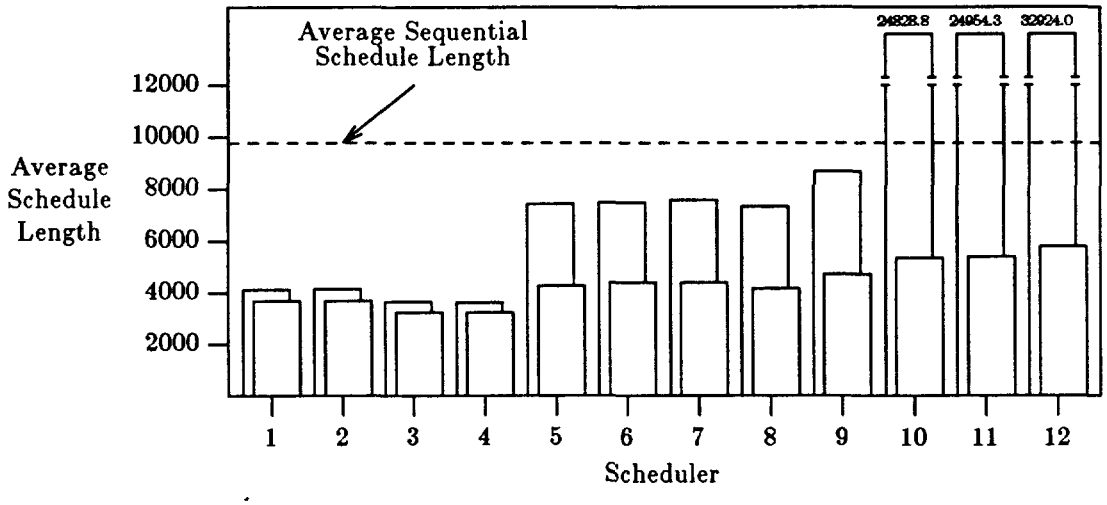




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	89.04	90.67	93.19	93.33	78.07	78.07	77.78	79.26	75.26	62.96	62.96	58.07
S/P	2.46	2.40	2.61	2.51	1.29	1.36	1.33	1.32	1.11	0.43	0.40	0.31
S/C	2.72	2.71	2.86	2.85	2.28	2.30	2.30	2.35	2.09	1.87	1.84	1.68
P/C	1.11	1.13	1.09	1.14	1.77	1.70	1.73	1.78	1.88	4.30	4.60	5.41
P Eff	0.55	0.55	0.57	0.57	0.49	0.48	0.48	0.50	0.40	0.43	0.43	0.38
C Eff	0.56	0.55	0.57	0.57	0.51	0.50	0.50	0.52	0.42	0.47	0.47	0.42
CPU Sec	329.20	358.93	425.95	815.98	3.38	55.39	25.42	52.41	1.21	21.73	49.04	15.60

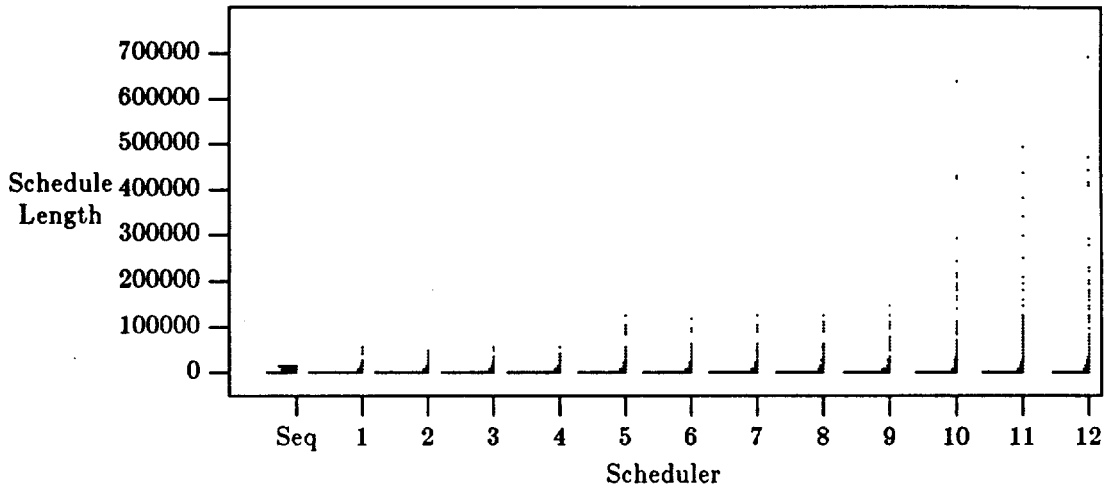
C.2.3. Figure C.4. — Distribution = 2 (675 Cases)

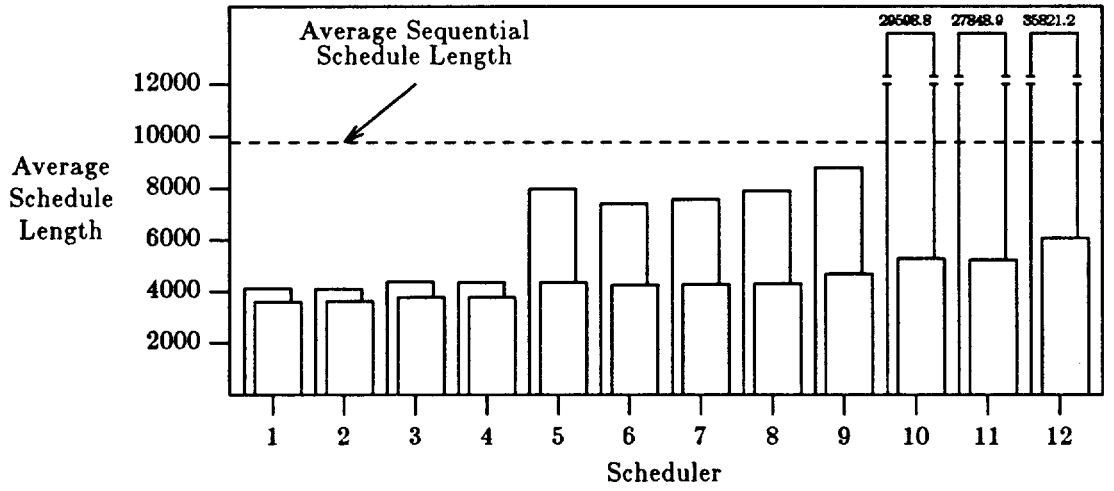




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.59	89.93	93.19	93.19	77.33	76.74	76.30	78.81	75.11	60.00	58.81	56.89
S/P	2.38	2.37	2.69	2.70	1.32	1.31	1.29	1.33	1.13	0.39	0.39	0.30
S/C	2.68	2.67	3.01	3.03	2.29	2.23	2.22	2.35	2.07	1.84	1.82	1.69
P/C	1.13	1.13	1.12	1.12	1.74	1.70	1.72	1.76	1.84	4.66	4.64	5.69
P Eff	0.53	0.52	0.57	0.57	0.47	0.45	0.45	0.49	0.39	0.41	0.41	0.37
C Eff	0.53	0.53	0.57	0.57	0.49	0.47	0.47	0.50	0.42	0.45	0.45	0.42
CPU Sec	336.16	368.04	433.74	824.09	4.18	57.03	25.95	55.03	1.24	24.74	52.50	17.22

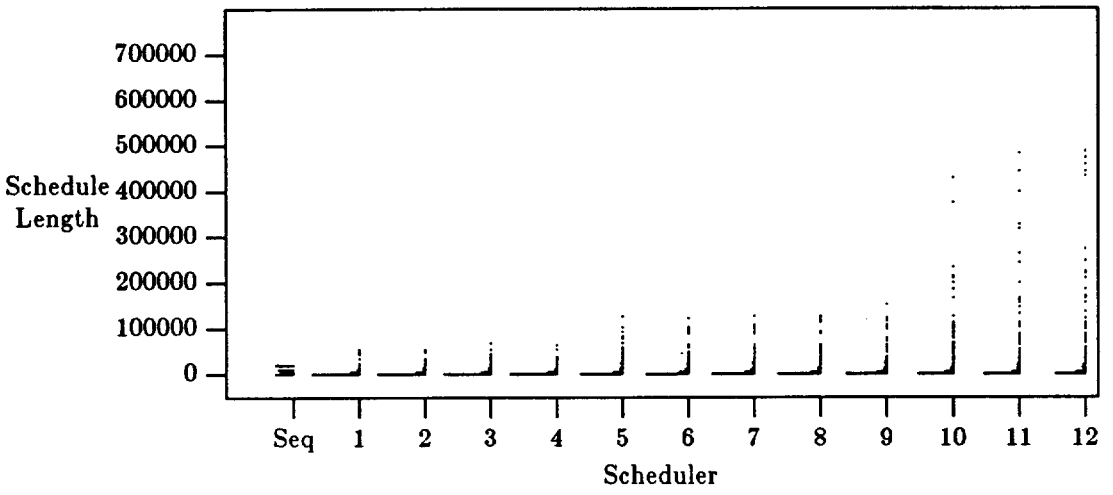
C.2.4. Figure C.5. — Distribution = 3 (675 Cases)

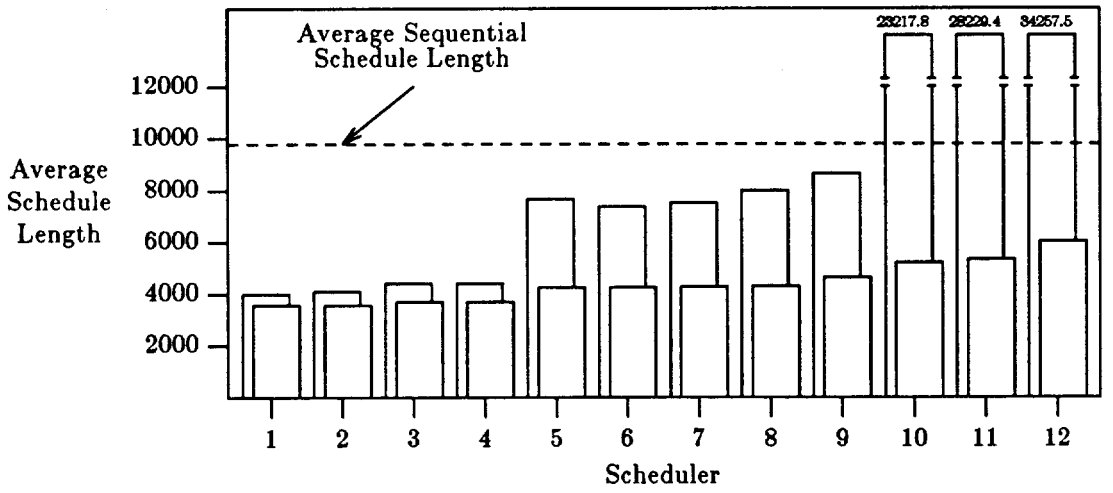




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	90.22	91.56	88.44	88.74	76.74	77.93	77.63	78.37	75.41	61.48	62.07	58.67
S/P	2.38	2.39	2.24	2.25	1.22	1.32	1.29	1.24	1.11	0.33	0.35	0.27
S/C	2.73	2.72	2.61	2.60	2.26	2.30	2.29	2.28	2.10	1.86	1.87	1.61
P/C	1.15	1.14	1.16	1.16	1.84	1.74	1.77	1.84	1.88	5.62	5.33	5.90
P Eff	0.55	0.55	0.54	0.54	0.49	0.48	0.48	0.48	0.40	0.43	0.43	0.34
C Eff	0.56	0.55	0.55	0.55	0.51	0.50	0.50	0.50	0.42	0.47	0.47	0.38
CPU Sec	345.11	373.73	460.77	853.34	3.32	57.21	26.46	54.65	1.22	20.81	50.58	14.91

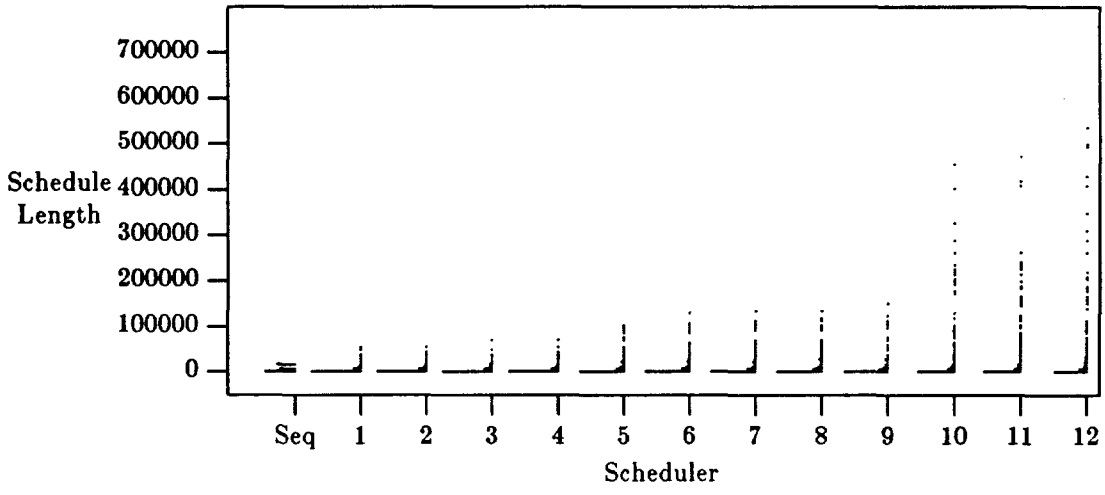
C.2.5. Figure C.6. — Distribution = 4 (675 Cases)

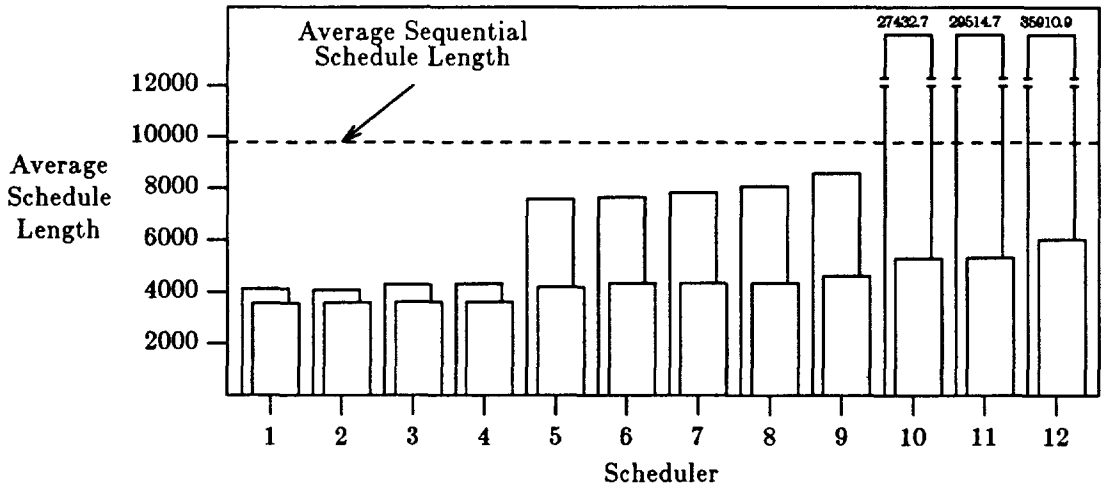




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	89.19	89.93	88.00	89.04	76.15	78.22	77.93	77.93	76.15	62.22	61.33	55.85
S/P	2.46	2.39	2.22	2.23	1.27	1.33	1.30	1.22	1.13	0.42	0.35	0.29
S/C	2.75	2.74	2.66	2.66	2.31	2.30	2.30	2.27	2.12	1.88	1.84	1.63
P/C	1.12	1.15	1.19	1.19	1.81	1.74	1.77	1.86	1.87	4.47	5.31	5.70
P Eff	0.55	0.55	0.54	0.54	0.49	0.48	0.48	0.47	0.40	0.43	0.43	0.35
C Eff	0.56	0.55	0.55	0.55	0.51	0.50	0.50	0.50	0.43	0.47	0.47	0.40
CPU Sec	349.93	380.61	477.16	875.43	3.56	57.89	26.89	57.34	1.23	22.38	52.23	15.82

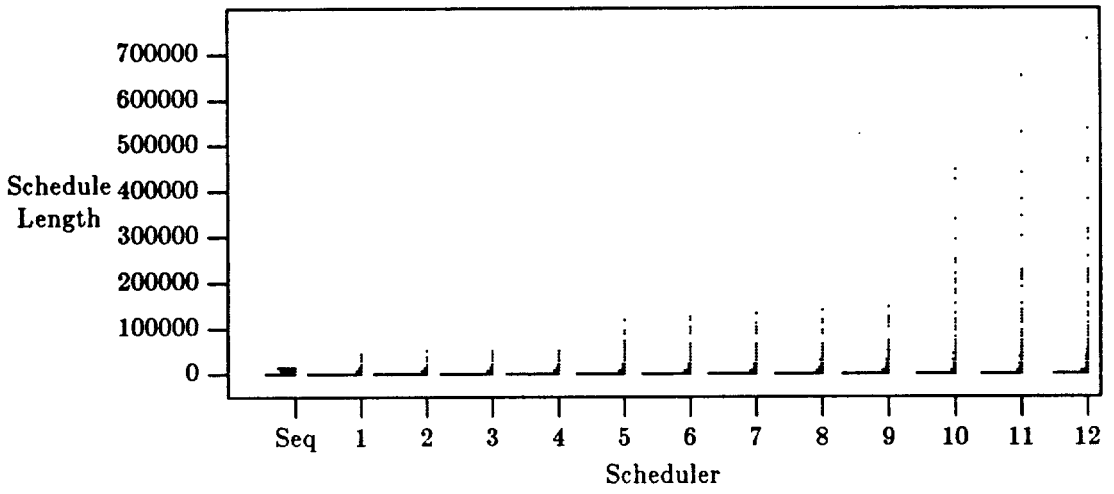
C.2.6. Figure C.7. — Distribution = 5 (675 Cases)

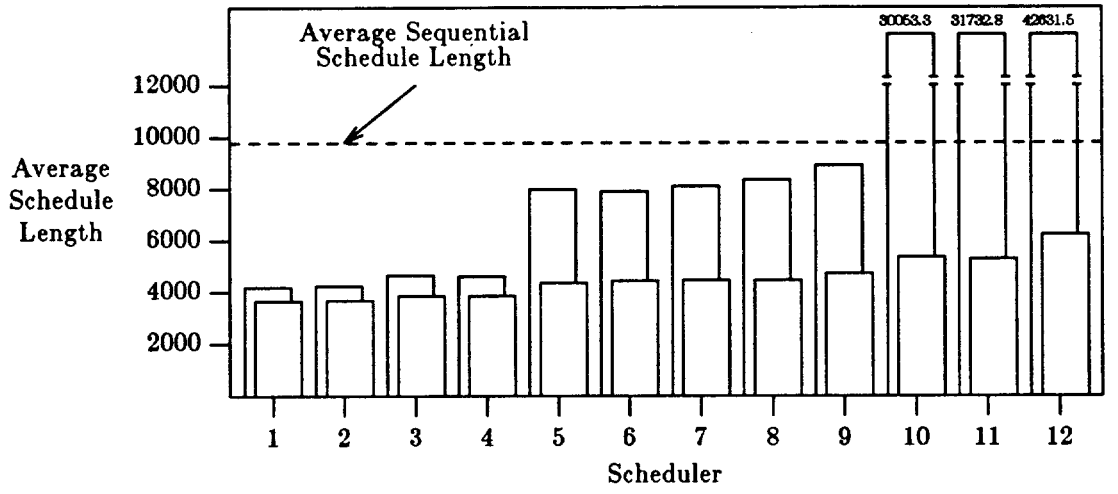




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.30	90.07	87.85	88.44	77.33	77.04	76.89	77.48	76.44	61.63	59.41	56.59
S/P	2.38	2.42	2.28	2.27	1.29	1.28	1.25	1.21	1.14	0.36	0.33	0.27
S/C	2.74	2.73	2.70	2.70	2.34	2.26	2.25	2.26	2.12	1.85	1.84	1.62
P/C	1.15	1.13	1.19	1.19	1.81	1.77	1.80	1.86	1.86	5.19	5.56	5.97
P Eff	0.54	0.53	0.53	0.53	0.48	0.46	0.46	0.47	0.40	0.41	0.41	0.35
C Eff	0.54	0.54	0.54	0.54	0.50	0.48	0.48	0.49	0.43	0.45	0.45	0.40
CPU Sec	357.54	388.22	486.70	888.79	3.92	59.25	27.42	59.87	1.24	24.88	54.42	16.73

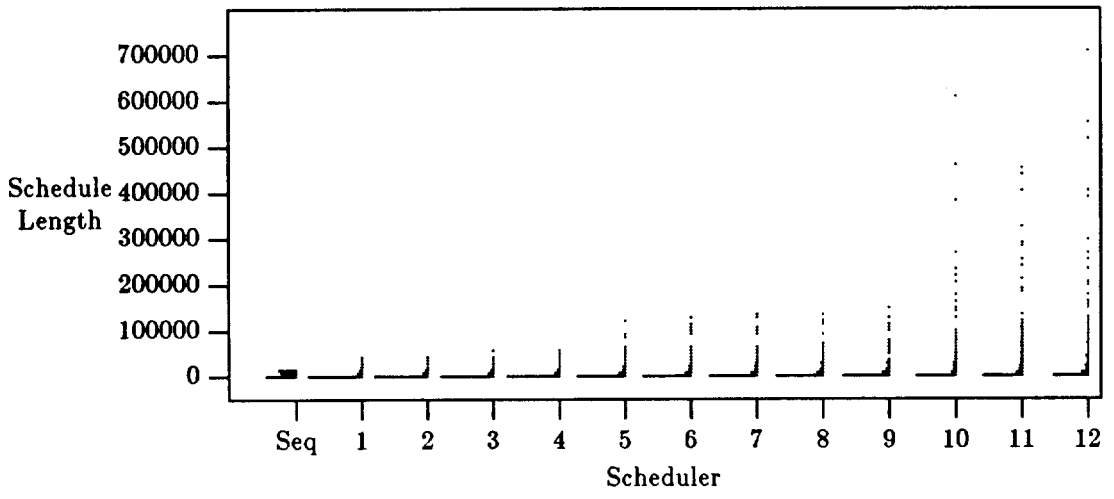
C.2.7. Figure C.8. — Distribution = 6 (675 Cases)

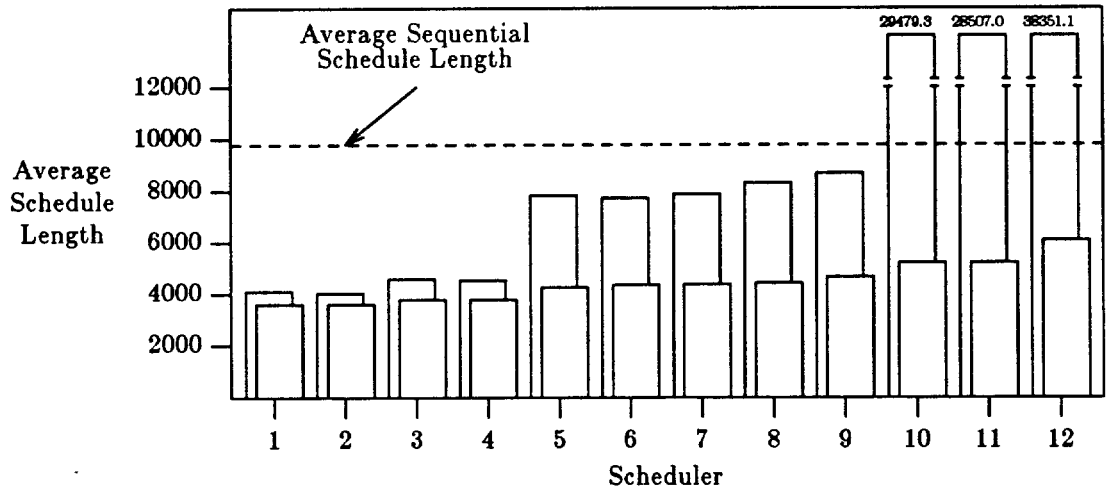




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	87.85	88.44	85.04	84.59	75.85	76.59	76.30	76.30	75.41	60.15	60.44	53.33
S/P	2.35	2.32	2.11	2.13	1.23	1.24	1.21	1.17	1.10	0.33	0.31	0.23
S/C	2.69	2.68	2.54	2.54	2.26	2.22	2.20	2.20	2.07	1.84	1.85	1.57
P/C	1.15	1.15	1.21	1.19	1.84	1.79	1.82	1.88	1.88	5.64	6.02	6.85
P Eff	0.52	0.52	0.51	0.51	0.47	0.45	0.45	0.45	0.39	0.41	0.41	0.31
C Eff	0.53	0.53	0.52	0.52	0.49	0.48	0.47	0.47	0.41	0.45	0.45	0.36
CPU Sec	363.99	395.58	507.85	914.43	3.93	60.57	27.83	60.13	1.24	24.10	55.65	15.63

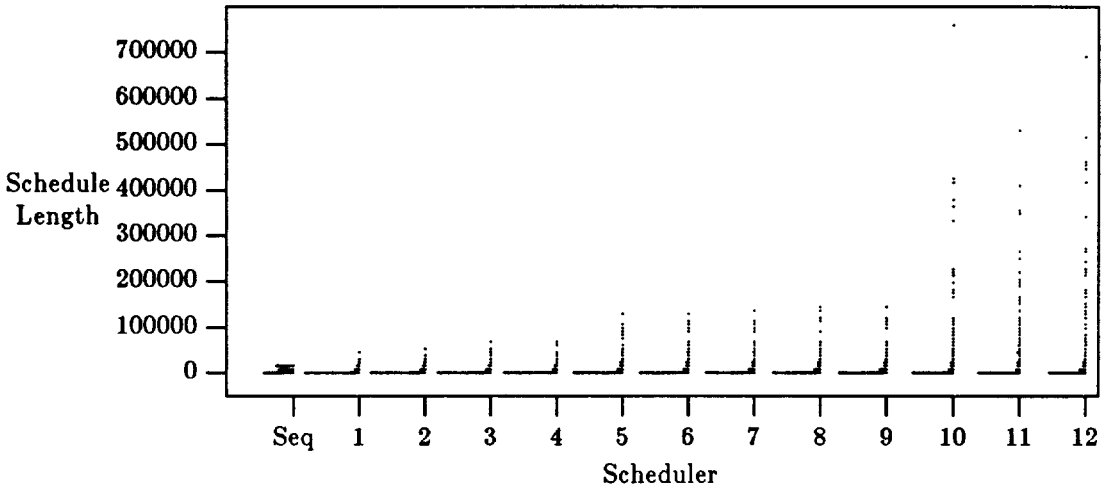
C.2.8. Figure C.9. — Distribution = 7 (675 Cases)

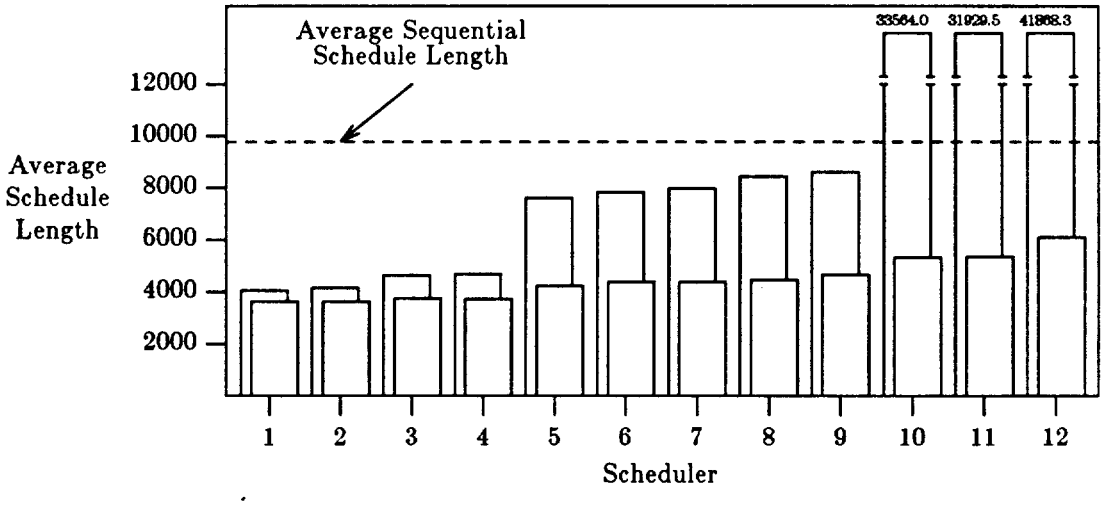




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.59	90.81	85.78	85.93	76.30	77.48	76.89	76.74	76.00	62.81	60.59	55.11
S/P	2.39	2.43	2.15	2.16	1.25	1.27	1.24	1.18	1.13	0.33	0.34	0.25
S/C	2.72	2.72	2.61	2.61	2.31	2.25	2.24	2.22	2.10	1.88	1.88	1.61
P/C	1.14	1.12	1.21	1.21	1.84	1.77	1.80	1.89	1.87	5.67	5.48	6.32
P Eff	0.53	0.53	0.52	0.52	0.48	0.46	0.46	0.46	0.40	0.42	0.41	0.33
C Eff	0.54	0.54	0.53	0.53	0.50	0.48	0.48	0.48	0.42	0.46	0.46	0.38
CPU Sec	362.10	393.50	503.09	909.83	3.96	60.19	27.71	60.70	1.24	24.16	55.02	16.41

C.2.9. Figure C.10. — Distribution = 8 (675 Cases)

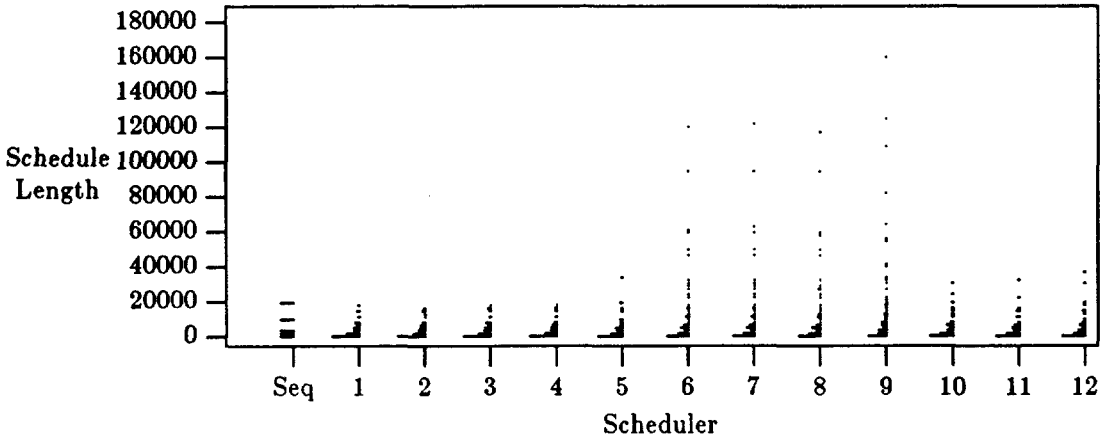


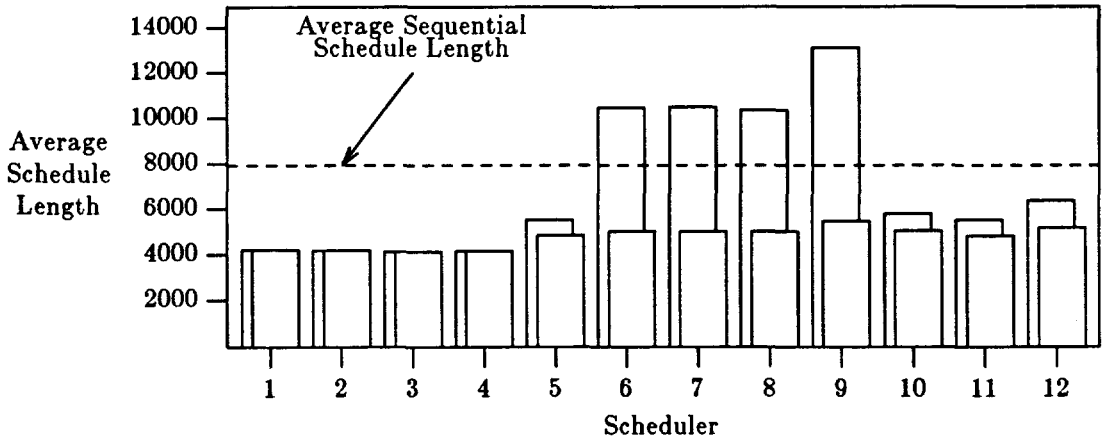


	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.59	89.19	86.67	87.26	77.19	76.44	76.15	76.44	76.15	60.00	60.30	55.85
S/P	2.42	2.37	2.12	2.10	1.29	1.25	1.23	1.16	1.14	0.29	0.31	0.23
S/C	2.71	2.71	2.63	2.63	2.33	2.24	2.23	2.20	2.11	1.84	1.83	1.61
P/C	1.12	1.14	1.24	1.25	1.81	1.79	1.82	1.90	1.85	6.33	5.98	6.88
P Eff	0.53	0.53	0.52	0.52	0.48	0.45	0.45	0.45	0.40	0.41	0.41	0.34
C Eff	0.54	0.53	0.53	0.53	0.50	0.47	0.47	0.47	0.42	0.45	0.45	0.39
CPU Sec	368.47	400.46	513.46	924.85	4.19	61.12	28.38	62.60	1.26	28.19	58.25	16.89

C.3. Comparison By Parallelism

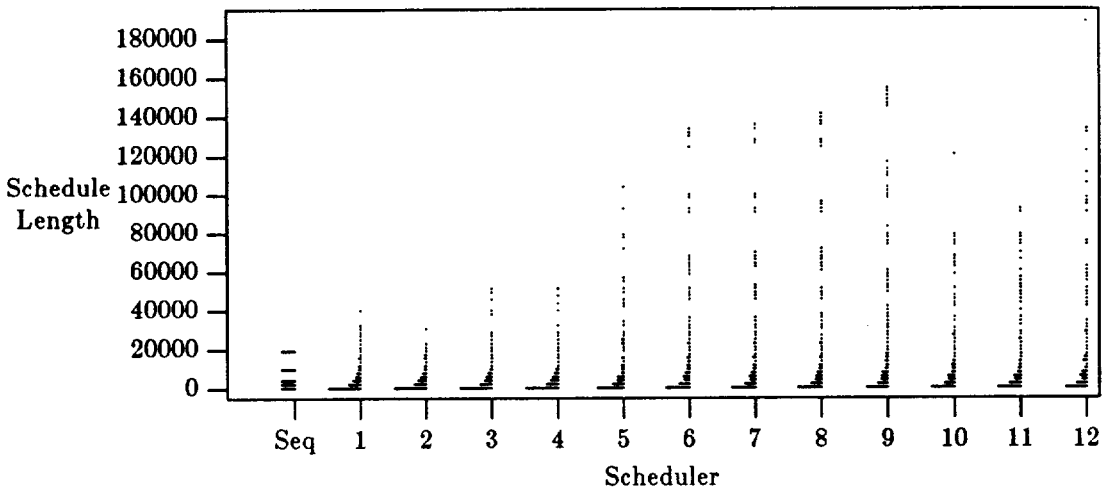
C.3.1. Figure C.11. — 1.5 < Parallelism < 3 (135 Cases)

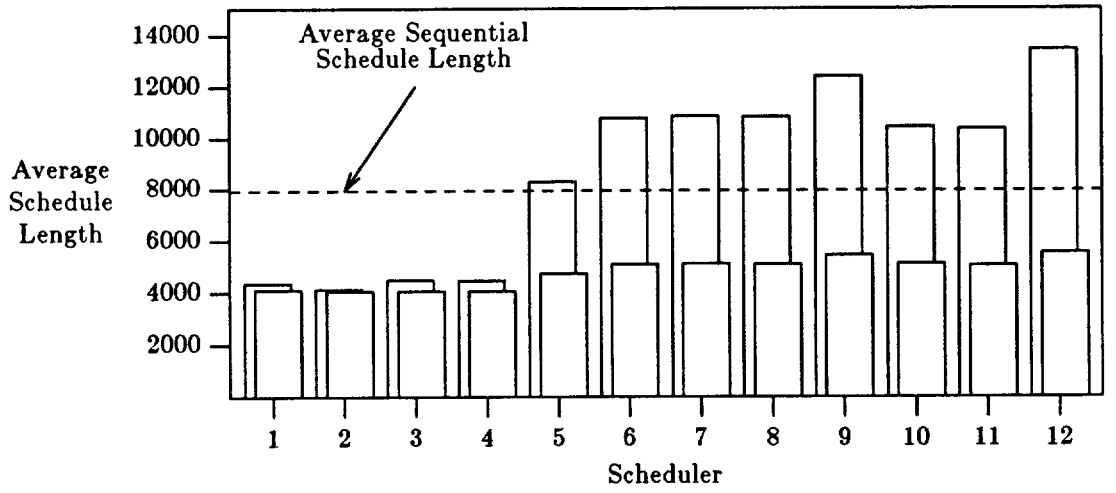




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	100.00	100.00	100.00	100.00	77.78	66.67	66.67	66.67	61.48	73.33	77.78	73.33
S/P	1.87	1.89	1.91	1.91	1.44	0.76	0.76	0.77	0.61	1.37	1.43	1.24
S/C	1.87	1.89	1.91	1.91	1.63	1.58	1.58	1.58	1.45	1.57	1.64	1.53
P/C	1.00	1.00	1.00	1.00	1.13	2.09	2.09	2.07	2.39	1.15	1.14	1.23
P Eff	0.34	0.34	0.34	0.34	0.31	0.27	0.27	0.27	0.21	0.29	0.30	0.28
C Eff	0.34	0.34	0.34	0.34	0.31	0.30	0.30	0.30	0.25	0.30	0.31	0.29
CPU Sec	171.53	183.66	198.23	388.22	1.54	30.87	16.44	25.44	0.79	5.85	19.89	5.49

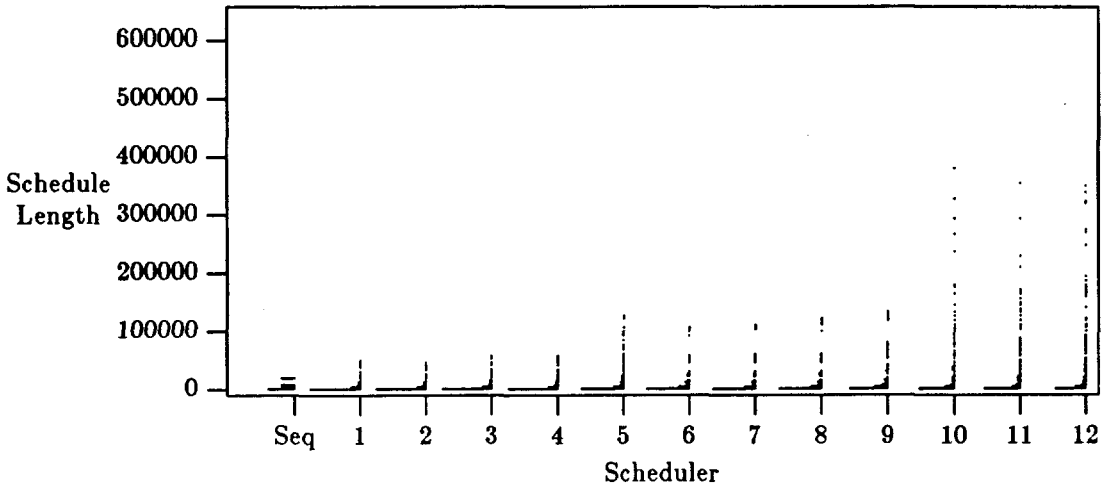
C.3.2. Figure C.12. — 3 < Parallelism < 6 (1080 Cases)

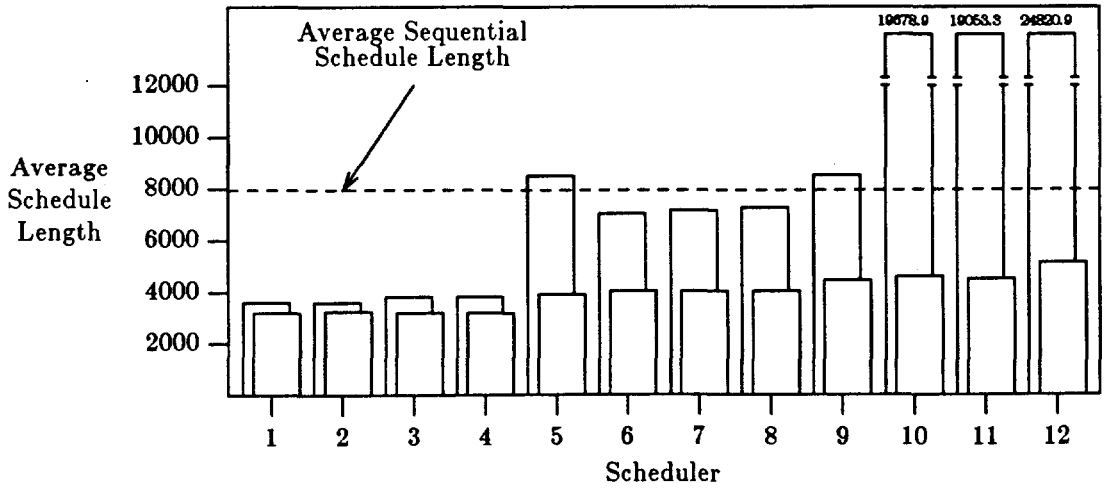




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.52	95.46	88.43	88.52	69.26	66.02	65.19	65.00	63.15	58.70	60.28	58.06
S/P	1.82	1.93	1.78	1.78	0.96	0.74	0.73	0.73	0.64	0.76	0.77	0.59
S/C	1.94	1.96	1.96	1.96	1.68	1.56	1.56	1.56	1.46	1.55	1.58	1.44
P/C	1.07	1.02	1.10	1.10	1.76	2.12	2.13	2.13	2.28	2.04	2.06	2.43
P Eff	0.33	0.34	0.33	0.33	0.29	0.25	0.25	0.25	0.21	0.26	0.27	0.23
C Eff	0.34	0.34	0.34	0.34	0.31	0.28	0.28	0.28	0.25	0.29	0.30	0.27
CPU Sec	203.84	217.03	234.00	463.48	1.72	34.53	17.23	28.98	0.87	7.94	24.87	6.96

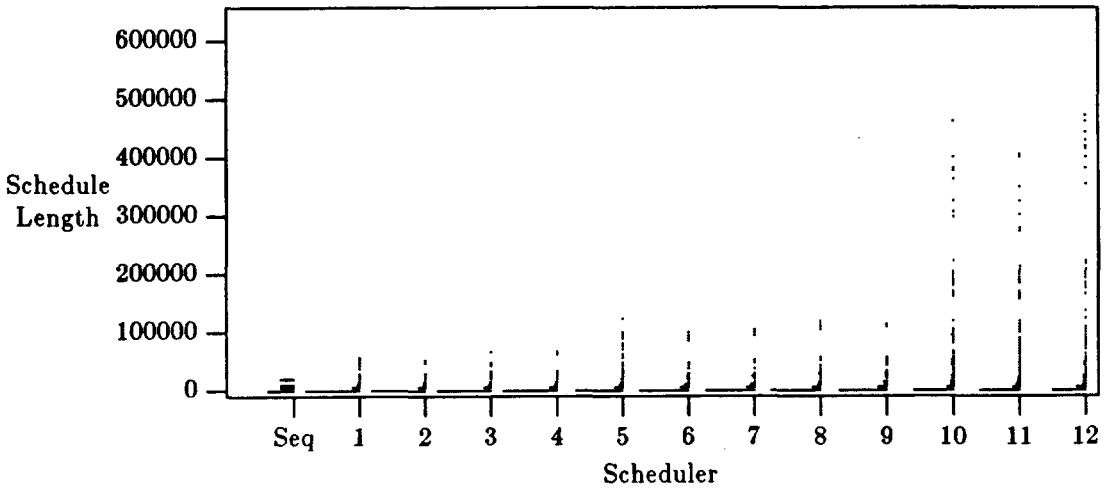
C.3.3. Figure C.13. — 6 < Parallelism < 12 (1215 Cases)

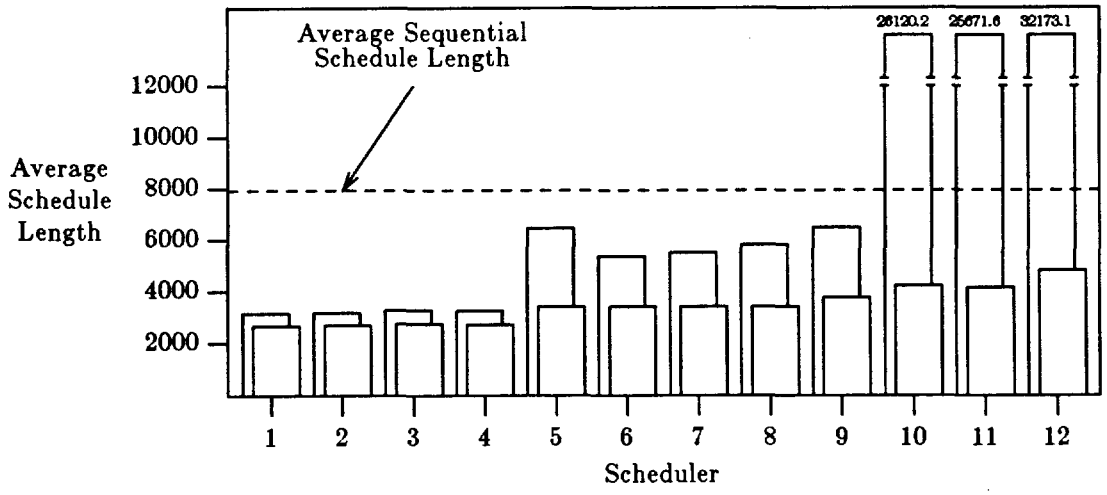




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.40	89.88	87.65	87.41	72.76	74.07	74.07	74.07	72.76	59.01	59.18	55.72
S/P	2.21	2.21	2.09	2.08	0.94	1.13	1.11	1.09	0.93	0.40	0.42	0.32
S/C	2.49	2.46	2.49	2.48	2.03	1.97	1.96	1.97	1.78	1.73	1.76	1.55
P/C	1.13	1.12	1.19	1.20	2.17	1.74	1.77	1.80	1.91	4.30	4.24	4.84
P Eff	0.46	0.46	0.46	0.46	0.40	0.38	0.38	0.38	0.31	0.35	0.36	0.28
C Eff	0.47	0.46	0.47	0.47	0.42	0.40	0.40	0.40	0.33	0.39	0.40	0.33
CPU Sec	250.10	269.72	297.69	567.71	1.98	39.91	19.43	36.46	0.95	11.72	31.71	9.55

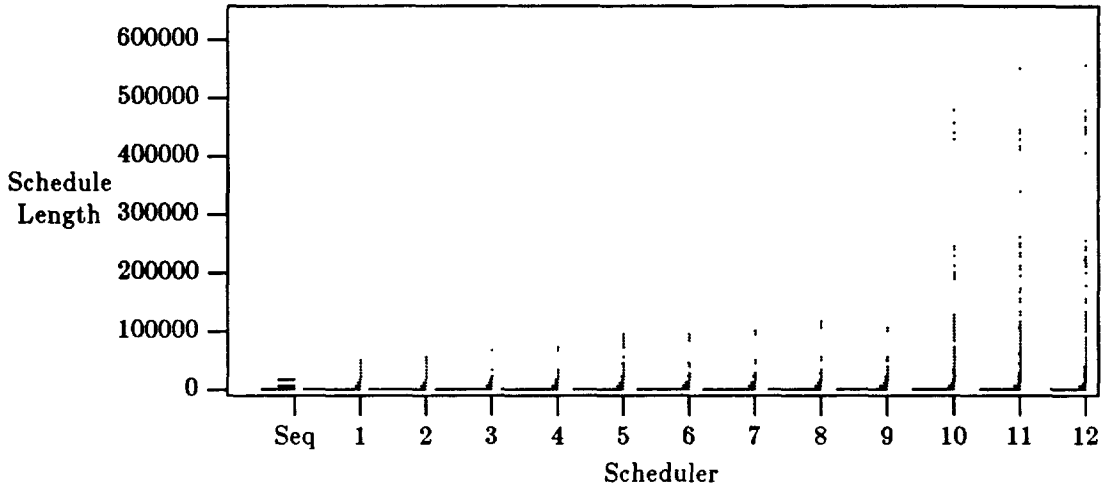
C.3.4. Figure C.14. — 12 < Parallelism < 24 (1215 Cases)

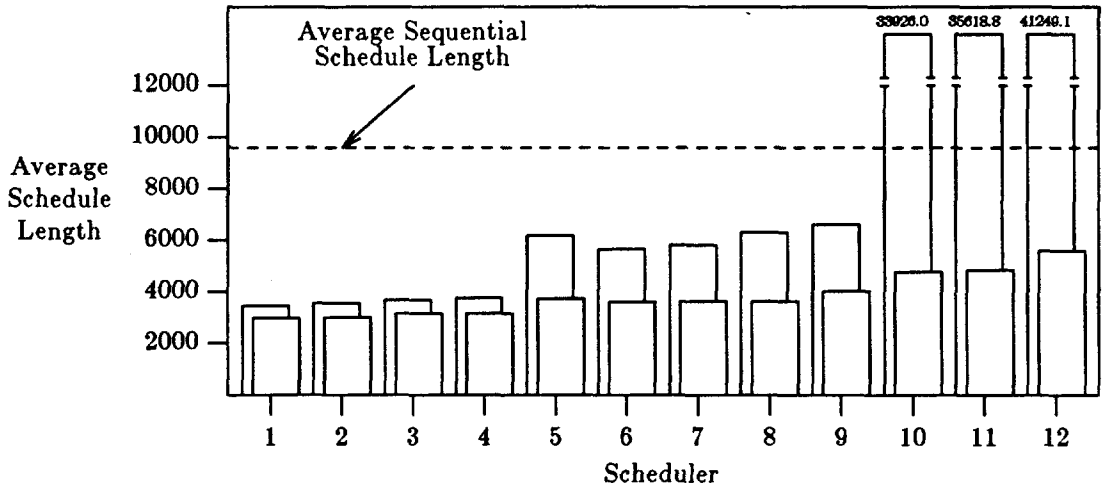




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.89	88.97	88.89	89.38	76.46	78.52	78.11	79.26	77.45	61.48	62.22	55.64
S/P	2.52	2.48	2.41	2.43	1.23	1.48	1.44	1.37	1.22	0.30	0.31	0.25
S/C	2.97	2.93	2.89	2.89	2.32	2.33	2.32	2.33	2.11	1.89	1.92	1.65
P/C	1.18	1.18	1.20	1.19	1.89	1.58	1.62	1.71	1.72	6.22	6.21	6.68
P Eff	0.57	0.56	0.56	0.57	0.50	0.49	0.49	0.50	0.40	0.44	0.44	0.36
C Eff	0.58	0.57	0.57	0.57	0.52	0.51	0.51	0.52	0.42	0.48	0.48	0.41
CPU Sec	280.26	303.77	338.98	639.18	2.20	43.70	20.86	41.35	1.00	15.25	37.43	11.85

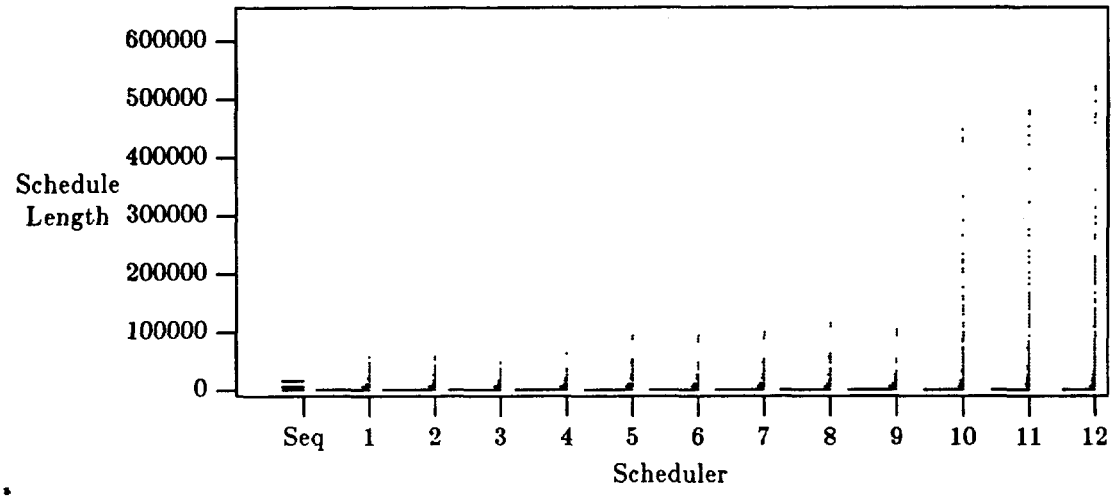
C.3.5. Figure C.15. — 24 < Parallelism < 48 (972 Cases)

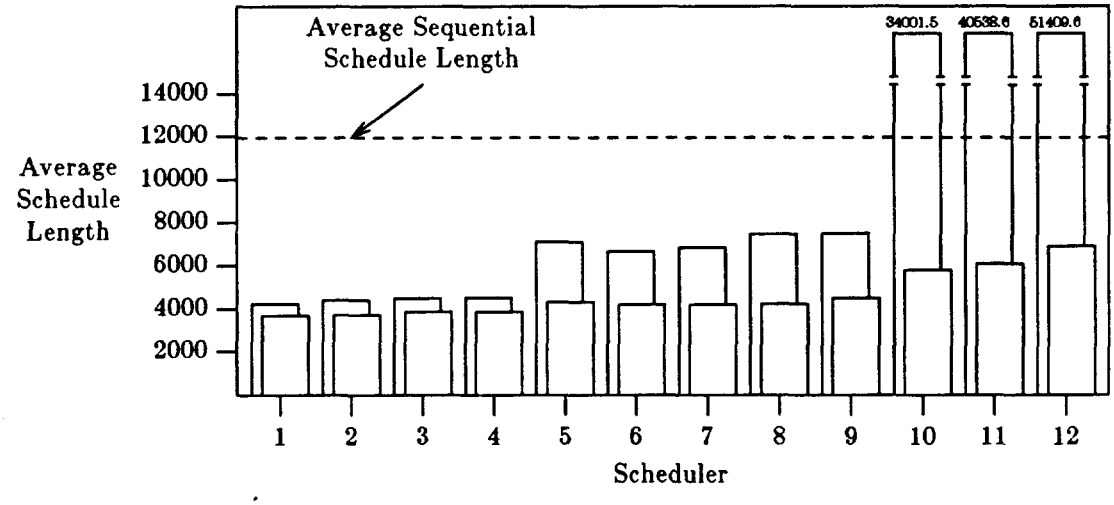




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	89.51	89.09	89.40	90.33	80.97	83.85	83.85	84.98	80.56	63.99	61.73	56.58
S/P	2.76	2.70	2.61	2.56	1.55	1.70	1.65	1.52	1.45	0.28	0.27	0.23
S/C	3.21	3.20	3.04	3.04	2.58	2.65	2.64	2.64	2.40	2.01	1.98	1.72
P/C	1.16	1.18	1.16	1.19	1.66	1.56	1.60	1.74	1.65	7.11	7.36	7.38
P Eff	0.65	0.64	0.64	0.64	0.58	0.57	0.57	0.58	0.48	0.50	0.50	0.41
C Eff	0.65	0.65	0.65	0.65	0.60	0.59	0.59	0.60	0.50	0.54	0.54	0.46
CPU Sec	363.91	395.41	475.82	870.06	3.08	57.17	26.86	55.60	1.25	22.59	51.88	16.62

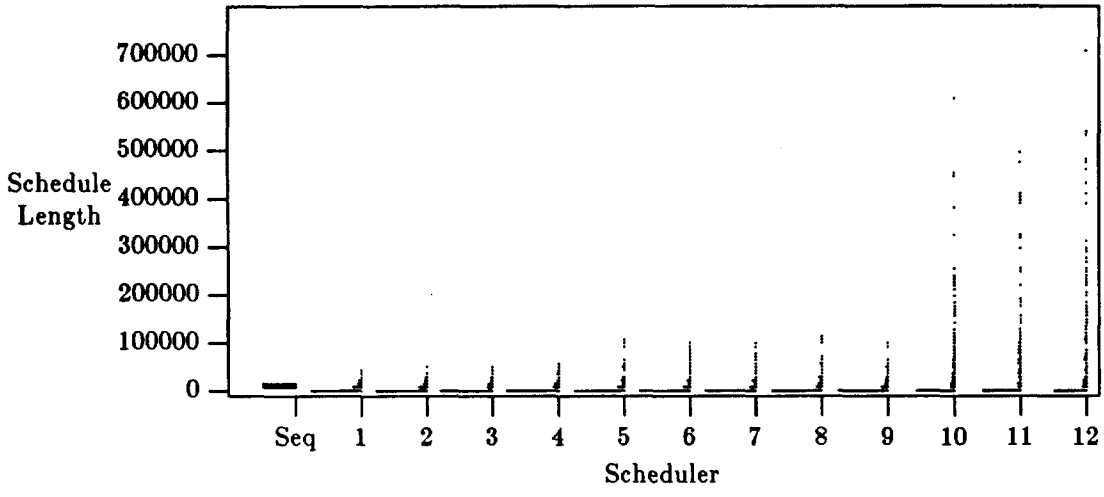
C.3.6. Figure C.16. — 48 < Parallelism < 96 (729 Cases)

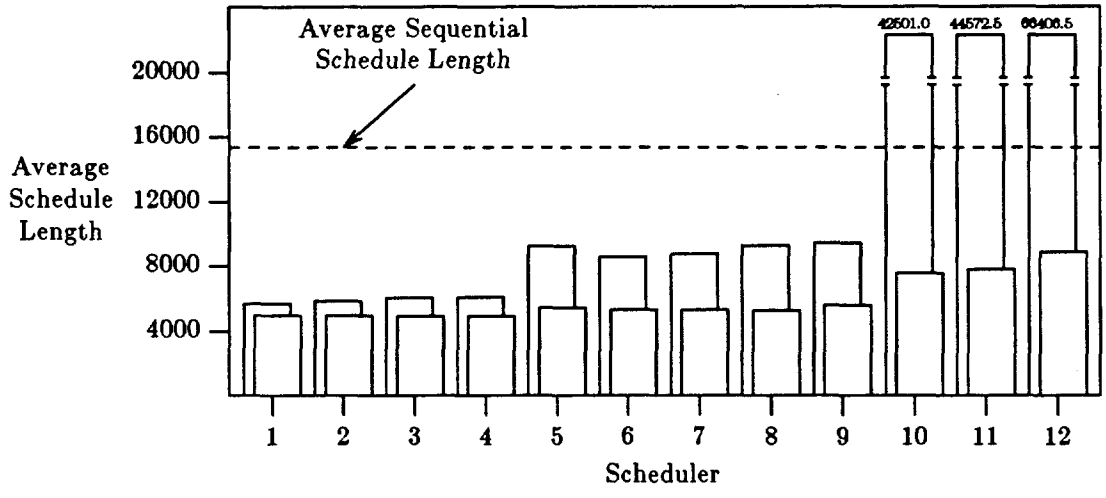




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	89.16	88.34	88.89	89.30	82.72	84.22	84.09	85.05	83.54	65.02	61.18	56.79
S/P	2.82	2.70	2.64	2.63	1.68	1.80	1.75	1.60	1.59	0.35	0.29	0.23
S/C	3.22	3.22	3.09	3.09	2.75	2.85	2.84	2.82	2.65	2.06	1.96	1.73
P/C	1.14	1.19	1.17	1.17	1.64	1.59	1.62	1.77	1.66	5.86	6.65	7.44
P Eff	0.68	0.68	0.68	0.68	0.62	0.63	0.62	0.63	0.55	0.54	0.53	0.44
C Eff	0.69	0.69	0.69	0.69	0.64	0.64	0.64	0.65	0.57	0.58	0.57	0.49
CPU Sec	473.72	514.76	680.99	1205.80	4.84	76.72	35.26	78.74	1.58	34.76	73.10	23.24

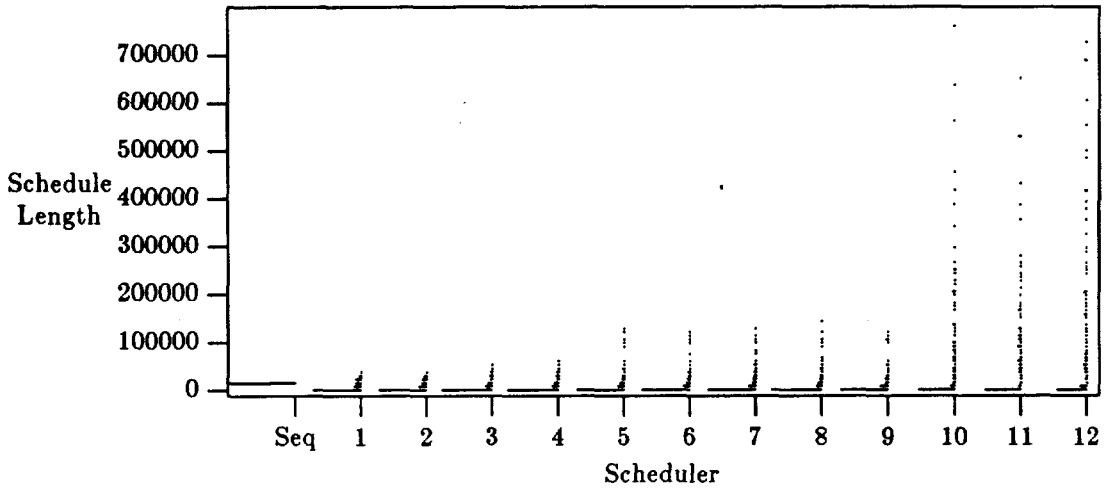
C.3.7. Figure C.17. — 96 < Parallelism < 192 (486 Cases)

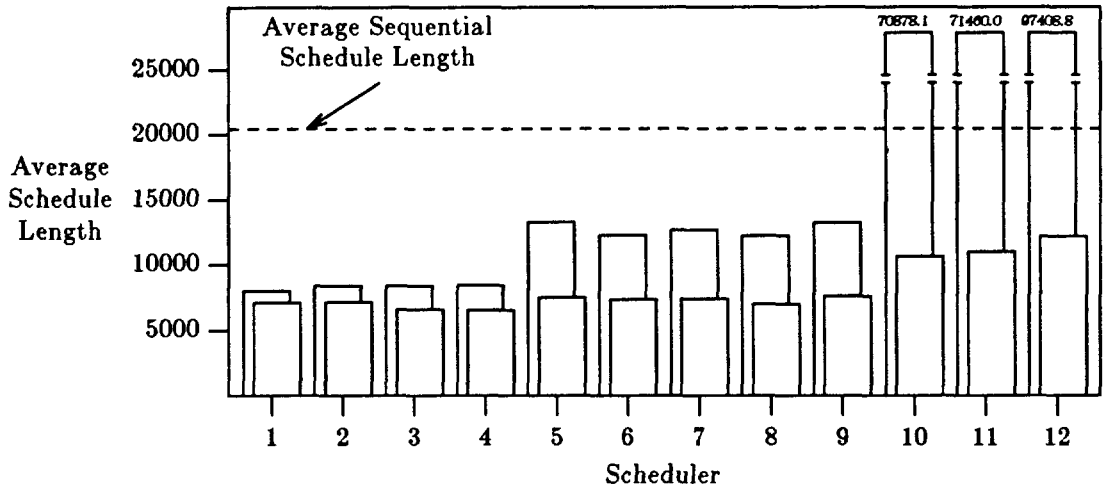




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.68	88.48	88.68	88.68	84.16	84.57	83.95	85.39	84.57	63.79	59.88	54.73
S/P	2.70	2.62	2.54	2.53	1.66	1.79	1.75	1.65	1.63	0.36	0.34	0.23
S/C	3.10	3.10	3.11	3.12	2.81	2.89	2.88	2.90	2.74	2.03	1.97	1.73
P/C	1.14	1.18	1.23	1.23	1.70	1.61	1.65	1.75	1.69	5.62	5.72	7.48
P Eff	0.70	0.69	0.70	0.70	0.65	0.65	0.65	0.66	0.60	0.55	0.54	0.46
C Eff	0.70	0.70	0.71	0.71	0.66	0.67	0.67	0.68	0.61	0.59	0.59	0.51
CPU Sec	637.22	697.28	985.83	1750.92	8.92	111.04	48.60	119.16	2.06	55.88	111.80	34.02

C.3.8. Figure C.18. — 192 < Parallelism < 384 (243 Cases)

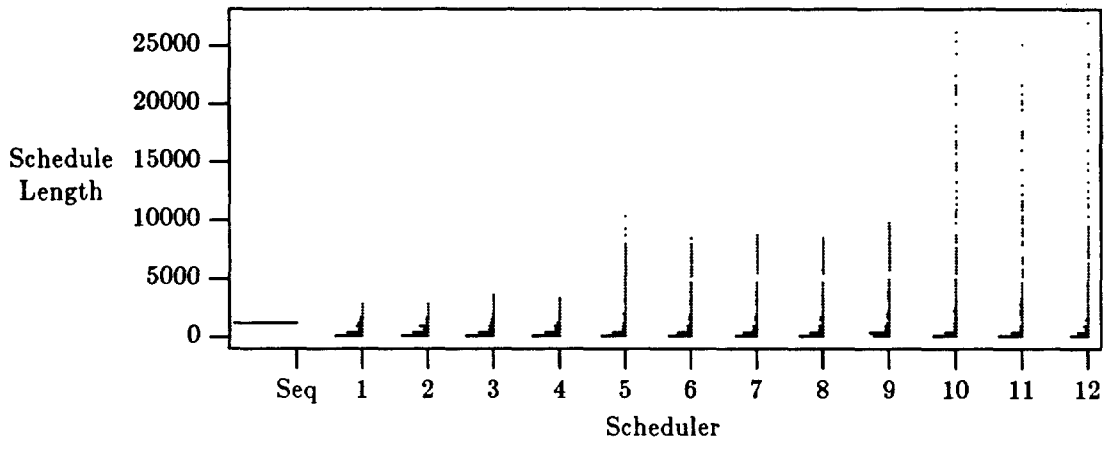


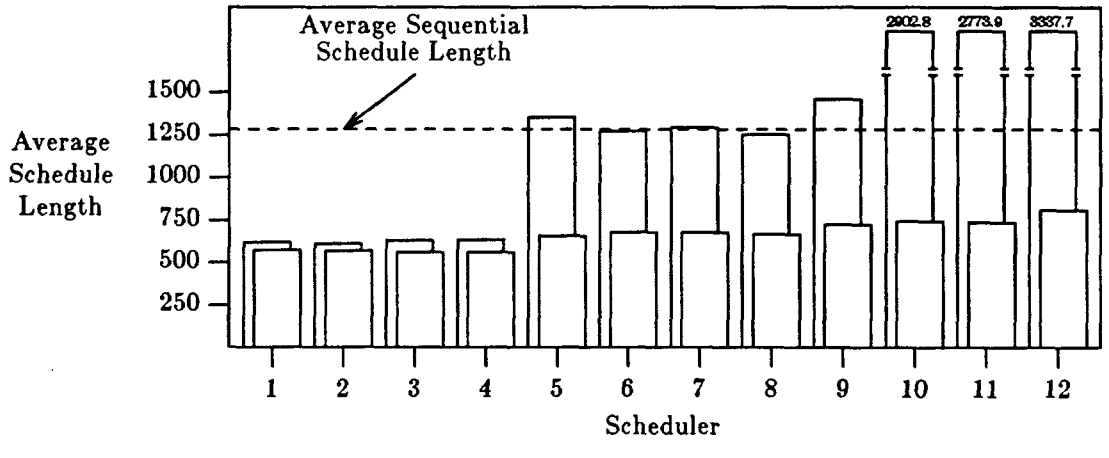


	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.48	87.65	87.24	88.07	83.54	83.95	84.36	85.60	83.95	59.67	56.79	53.91
S/P	2.55	2.42	2.43	2.42	1.54	1.66	1.61	1.67	1.54	0.29	0.29	0.21
S/C	2.86	2.86	3.09	3.10	2.72	2.77	2.76	2.91	2.68	1.92	1.86	1.68
P/C	1.12	1.18	1.27	1.28	1.76	1.67	1.71	1.74	1.74	6.66	6.50	7.97
P Eff	0.68	0.68	0.71	0.71	0.64	0.65	0.65	0.67	0.61	0.54	0.53	0.44
C Eff	0.69	0.69	0.72	0.72	0.66	0.67	0.67	0.69	0.63	0.58	0.58	0.50
CPU Sec	909.51	1022.49	1493.03	2770.58	18.79	183.65	73.04	195.49	2.82	105.92	200.41	53.51

C.4. Comparison By Program Size

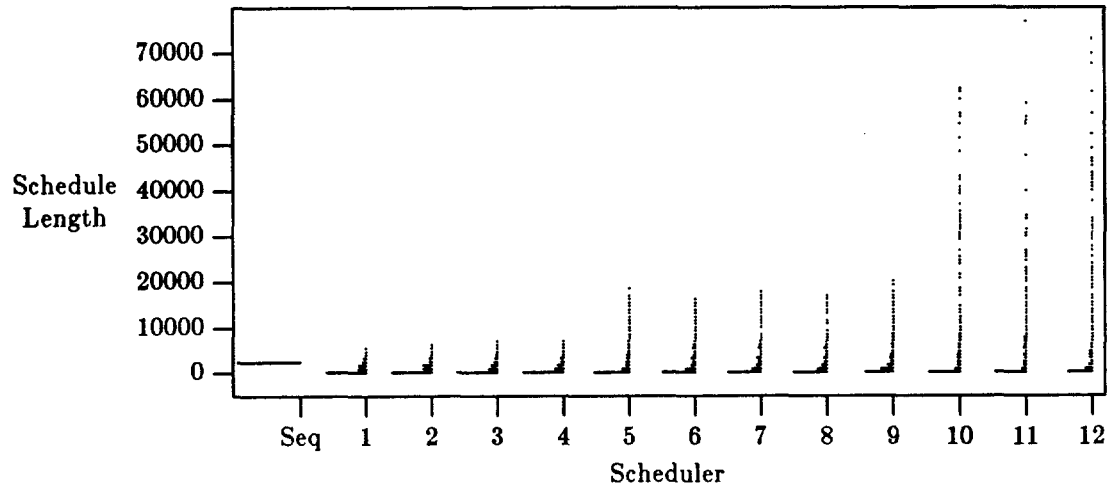
C.4.1. Figure C.19. — Program size = 128 (729 Cases)

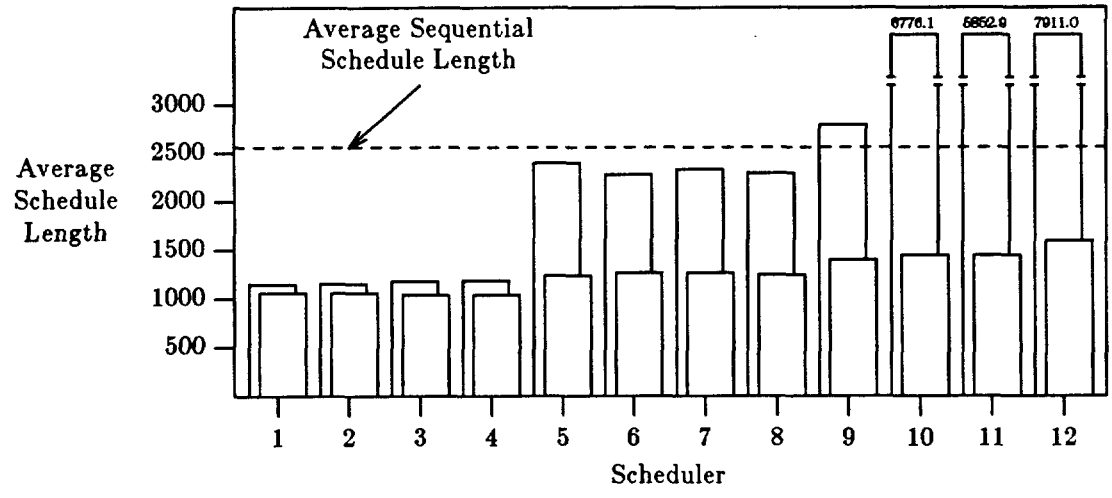




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	87.65	91.36	88.07	88.20	73.25	72.70	72.70	73.11	70.92	60.77	61.04	57.48
S/P	2.07	2.09	2.02	2.02	0.95	1.01	0.99	1.02	0.88	0.44	0.46	0.38
S/C	2.24	2.25	2.28	2.28	1.94	1.88	1.88	1.91	1.76	1.72	1.73	1.58
P/C	1.08	1.07	1.13	1.13	2.05	1.87	1.90	1.87	2.00	3.90	3.75	4.13
P Eff	0.44	0.44	0.44	0.44	0.39	0.37	0.37	0.38	0.32	0.35	0.35	0.30
C Eff	0.44	0.44	0.45	0.45	0.41	0.39	0.39	0.40	0.34	0.38	0.39	0.35
CPU Sec	36.53	36.64	38.91	41.81	0.26	3.20	2.97	4.64	0.11	1.86	2.01	1.58

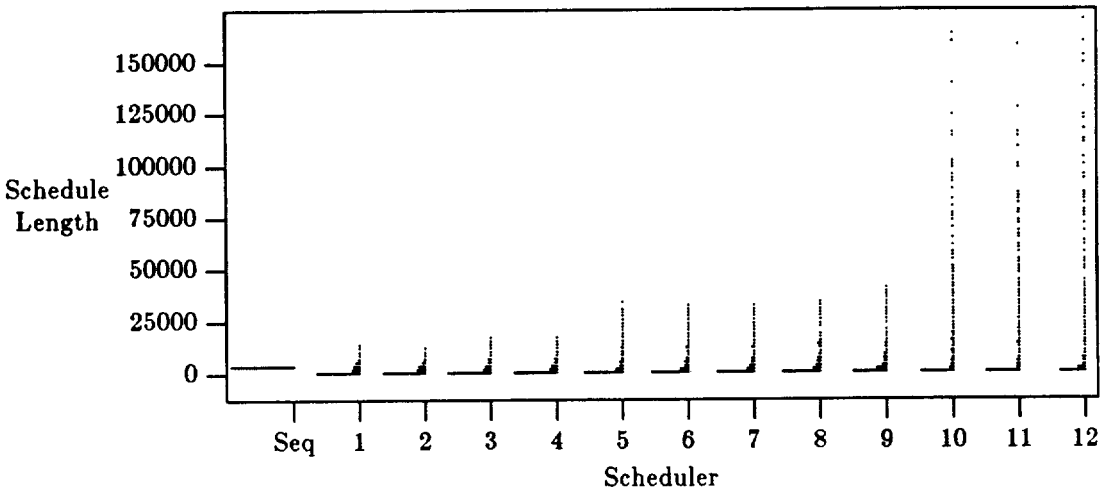
C.4.2. Figure C.20. — Program size = 256 (972 Cases)

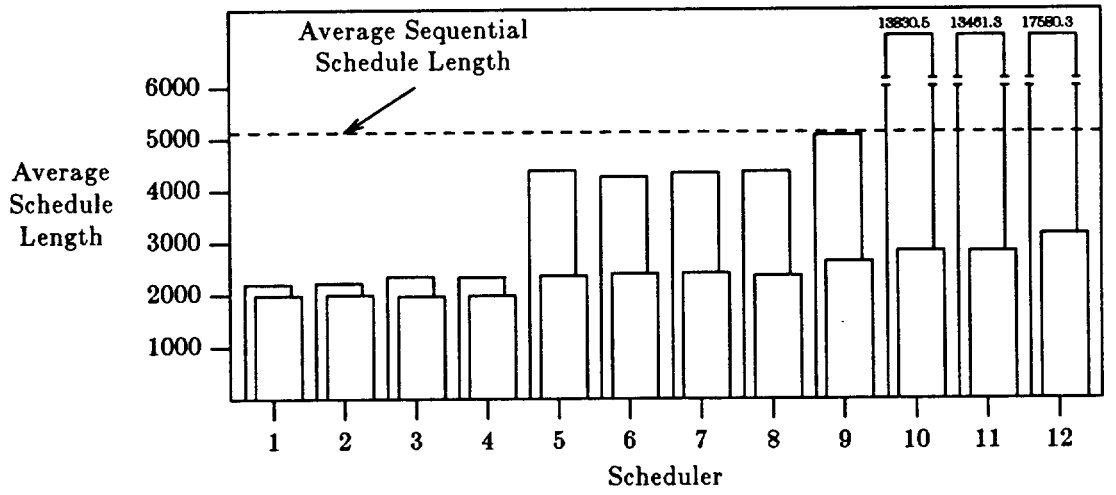




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.89	90.64	88.58	88.58	74.38	74.69	74.38	75.82	72.84	60.49	60.80	56.48
S/P	2.22	2.21	2.17	2.14	1.06	1.12	1.10	1.11	0.92	0.38	0.44	0.32
S/C	2.42	2.41	2.44	2.45	2.06	2.02	2.01	2.05	1.82	1.76	1.77	1.60
P/C	1.09	1.09	1.13	1.14	1.94	1.80	1.84	1.84	1.99	4.66	4.04	4.95
P Eff	0.49	0.49	0.49	0.49	0.44	0.42	0.42	0.43	0.33	0.39	0.39	0.33
C Eff	0.50	0.49	0.50	0.50	0.46	0.44	0.44	0.45	0.36	0.43	0.42	0.38
CPU Sec	79.69	80.76	86.89	99.53	0.62	7.28	6.31	10.97	0.27	4.38	5.09	3.52

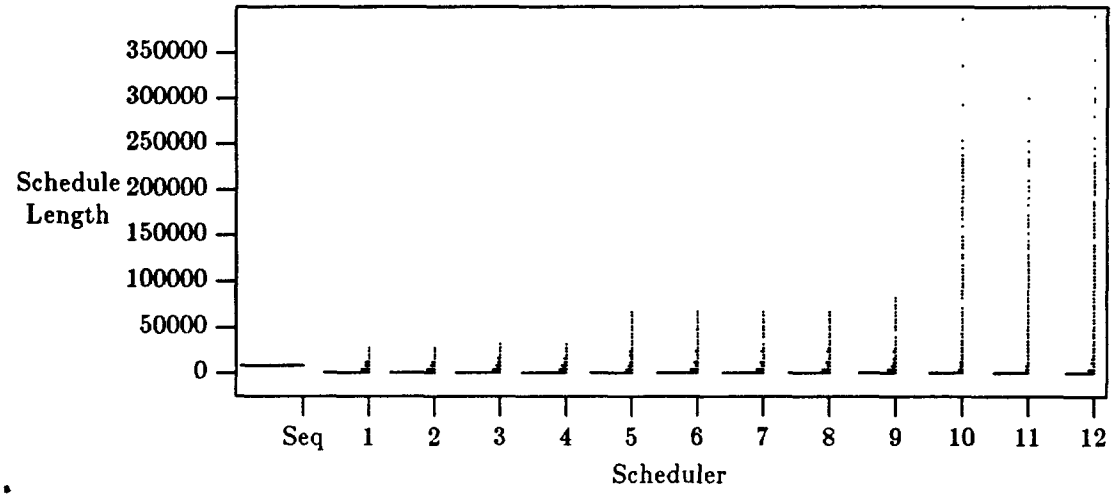
C.4.3. Figure C.21. — Program size = 512 (1215 Cases)

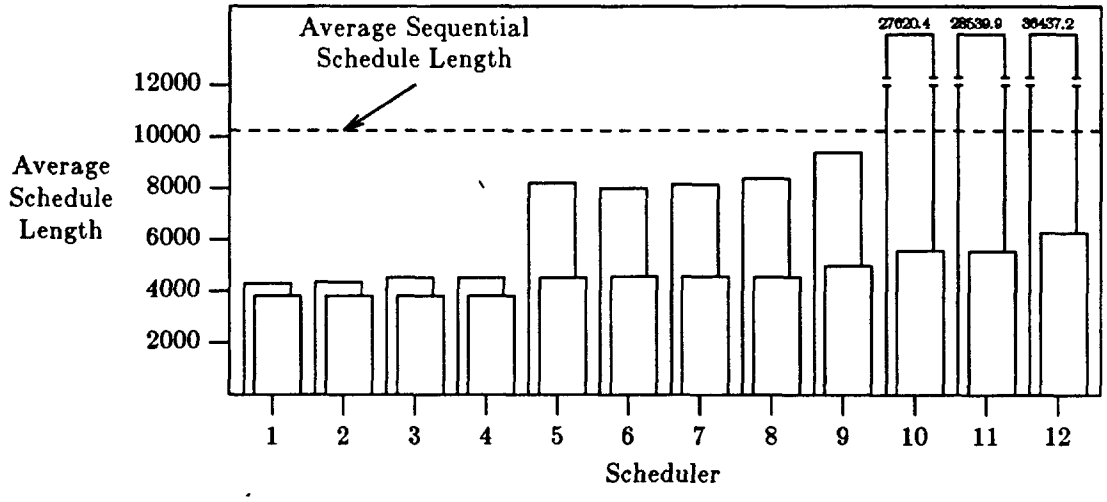




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	88.97	89.96	88.64	88.97	76.63	77.20	76.87	77.70	74.73	60.82	60.16	56.79
S/P	2.33	2.30	2.18	2.19	1.17	1.20	1.18	1.17	1.01	0.37	0.38	0.29
S/C	2.57	2.56	2.58	2.58	2.17	2.13	2.13	2.16	1.95	1.80	1.82	1.62
P/C	1.10	1.11	1.19	1.18	1.86	1.77	1.80	1.84	1.92	4.87	4.77	5.57
P Eff	0.54	0.53	0.54	0.54	0.48	0.46	0.46	0.47	0.38	0.42	0.42	0.35
C Eff	0.54	0.54	0.54	0.54	0.50	0.48	0.48	0.49	0.40	0.46	0.46	0.40
CPU Sec	170.33	173.16	197.48	255.76	1.43	17.77	13.16	23.79	0.59	9.91	13.82	7.58

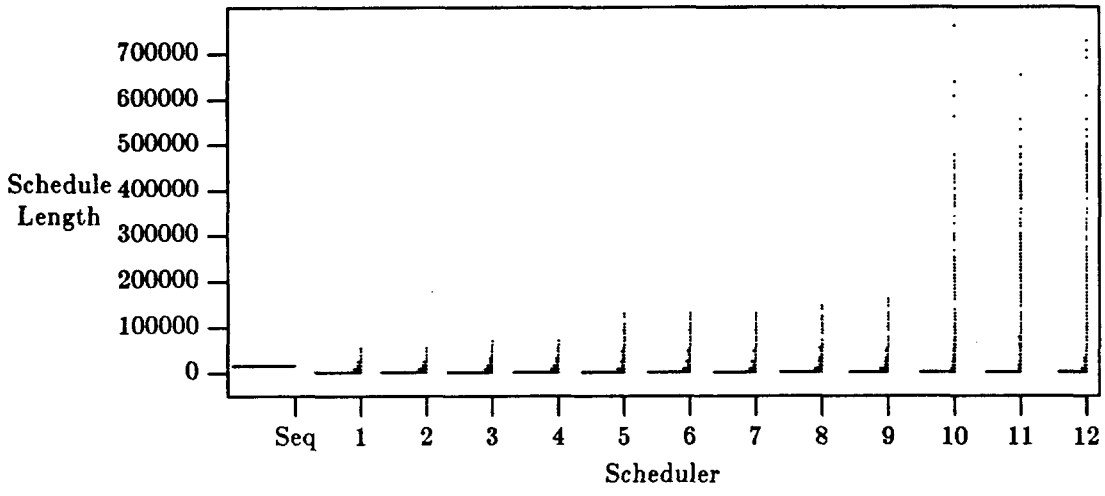
C.4.4. Figure C.22. — Program size = 1024 (1458 Cases)

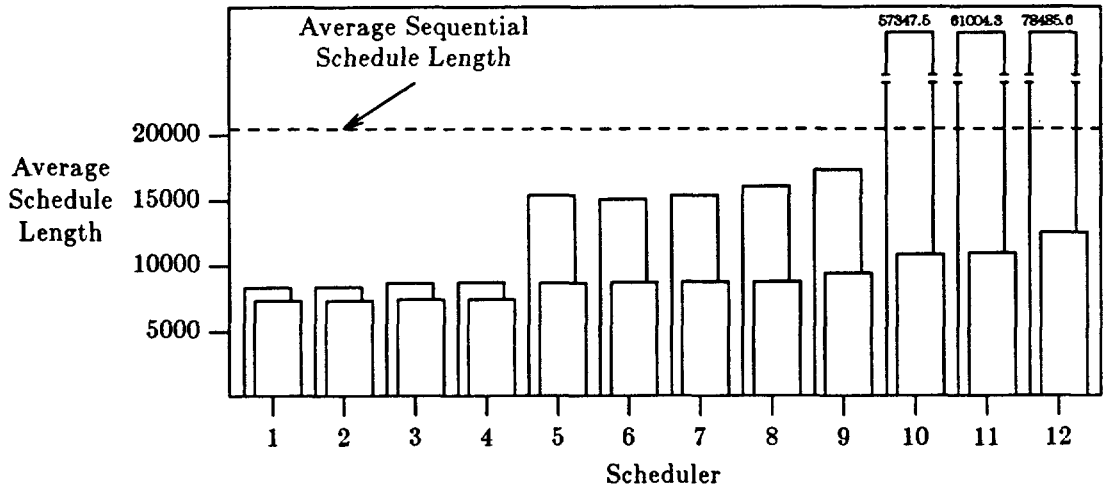




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	89.44	89.71	88.75	89.09	78.05	79.01	78.46	78.94	77.09	62.41	61.39	56.72
S/P	2.38	2.36	2.27	2.26	1.25	1.28	1.26	1.22	1.09	0.37	0.36	0.28
S/C	2.68	2.67	2.67	2.67	2.26	2.24	2.23	2.25	2.06	1.83	1.84	1.63
P/C	1.13	1.13	1.18	1.18	1.81	1.75	1.78	1.84	1.89	4.95	5.13	5.81
P Eff	0.57	0.56	0.56	0.56	0.51	0.50	0.49	0.50	0.42	0.44	0.44	0.36
C Eff	0.57	0.57	0.57	0.57	0.53	0.52	0.51	0.52	0.44	0.48	0.48	0.40
CPU Sec	362.19	382.43	458.79	714.34	3.45	47.95	27.75	55.76	1.27	22.77	42.07	16.28

C.4.5. Figure C.23. — Program size = 2048 (1701 Cases)

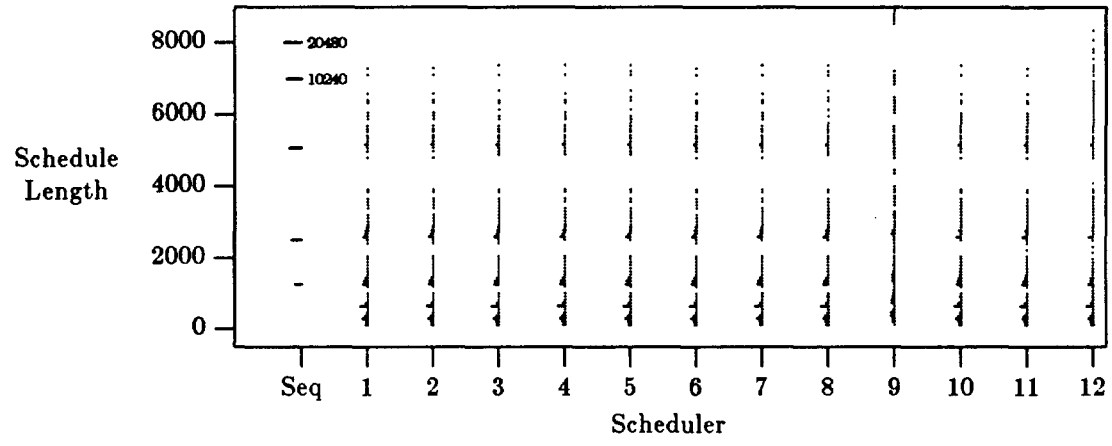


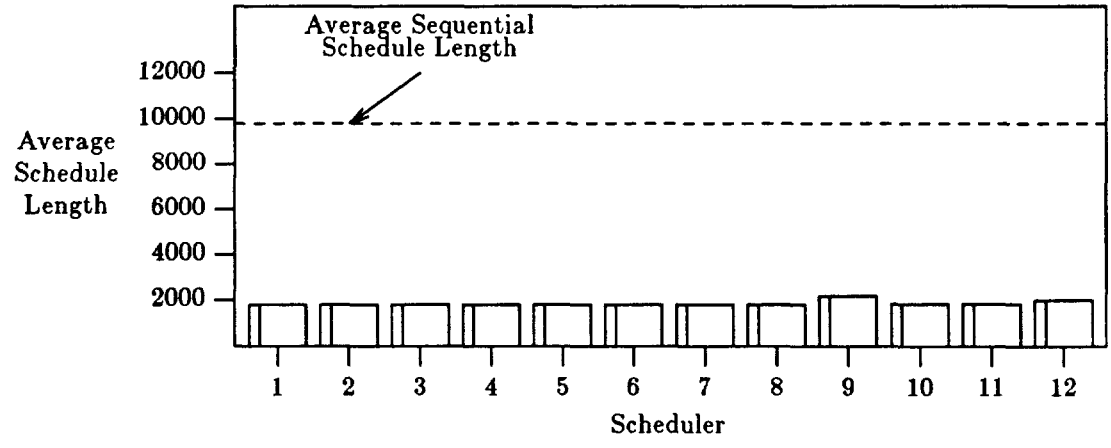


	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	89.54	90.77	89.42	89.89	78.89	79.66	79.54	79.89	78.78	62.79	61.38	56.14
S/P	2.47	2.44	2.36	2.34	1.33	1.36	1.33	1.27	1.18	0.36	0.34	0.26
S/C	2.80	2.80	2.76	2.76	2.36	2.34	2.34	2.33	2.17	1.89	1.86	1.63
P/C	1.14	1.14	1.17	1.18	1.77	1.73	1.76	1.84	1.84	5.28	5.55	6.25
P Eff	0.60	0.59	0.59	0.59	0.54	0.53	0.53	0.53	0.46	0.46	0.46	0.37
C Eff	0.60	0.60	0.60	0.60	0.55	0.55	0.54	0.55	0.48	0.50	0.50	0.41
CPU Sec	751.52	840.53	1077.15	2227.39	8.82	148.28	57.75	130.44	2.68	53.78	138.59	34.89

C.5. Comparison By Communication Latency

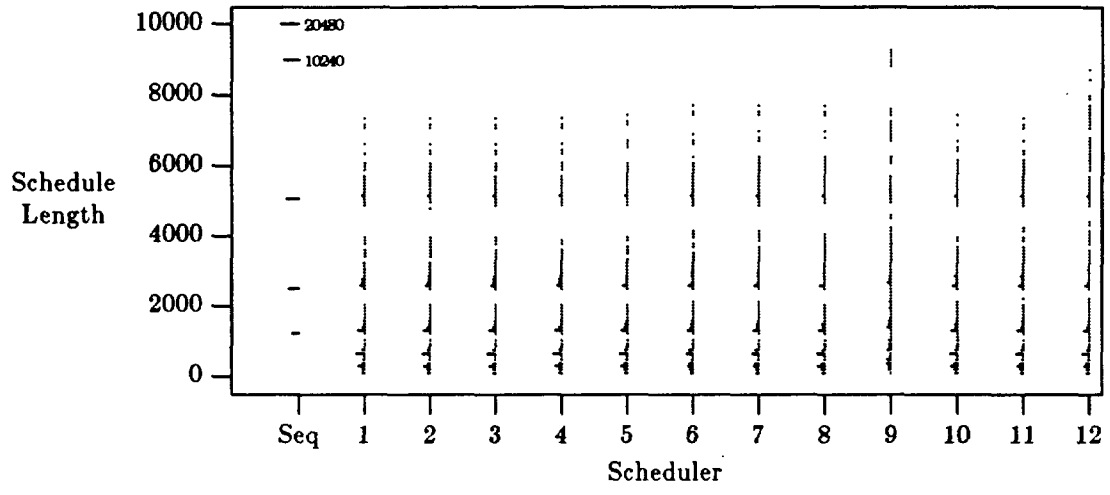
C.5.1. Figure C.24. — Latency = 0 (675 Cases)

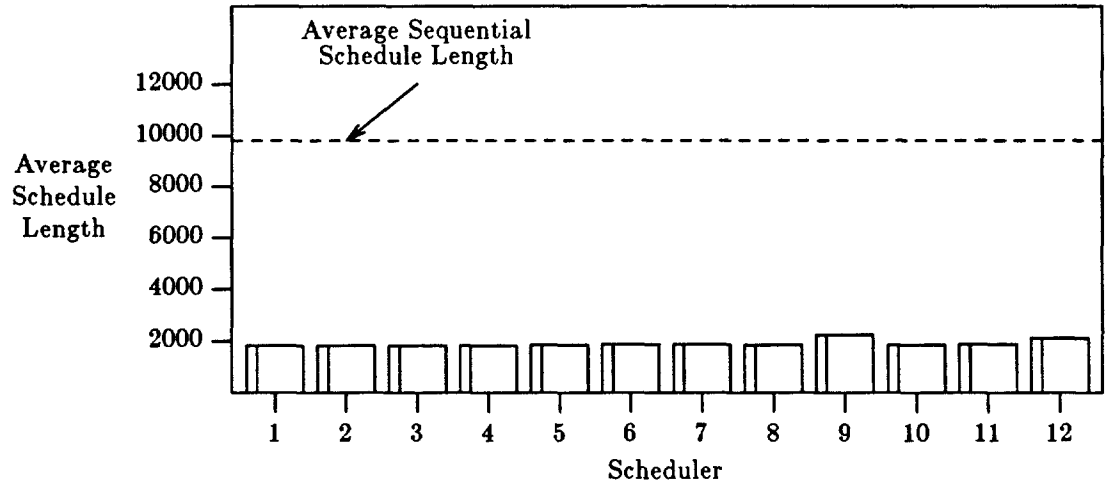




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
S/P	5.41	5.40	5.40	5.40	5.38	5.40	5.40	5.39	4.51	5.34	5.36	4.90
S/C	5.41	5.40	5.40	5.40	5.38	5.40	5.40	5.39	4.51	5.34	5.36	4.90
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
P Eff	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.64	0.77	0.77	0.75
C Eff	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.64	0.77	0.77	0.75
CPU Sec	236.01	269.11	289.44	693.65	3.80	48.06	17.19	33.51	1.23	19.46	49.89	19.63

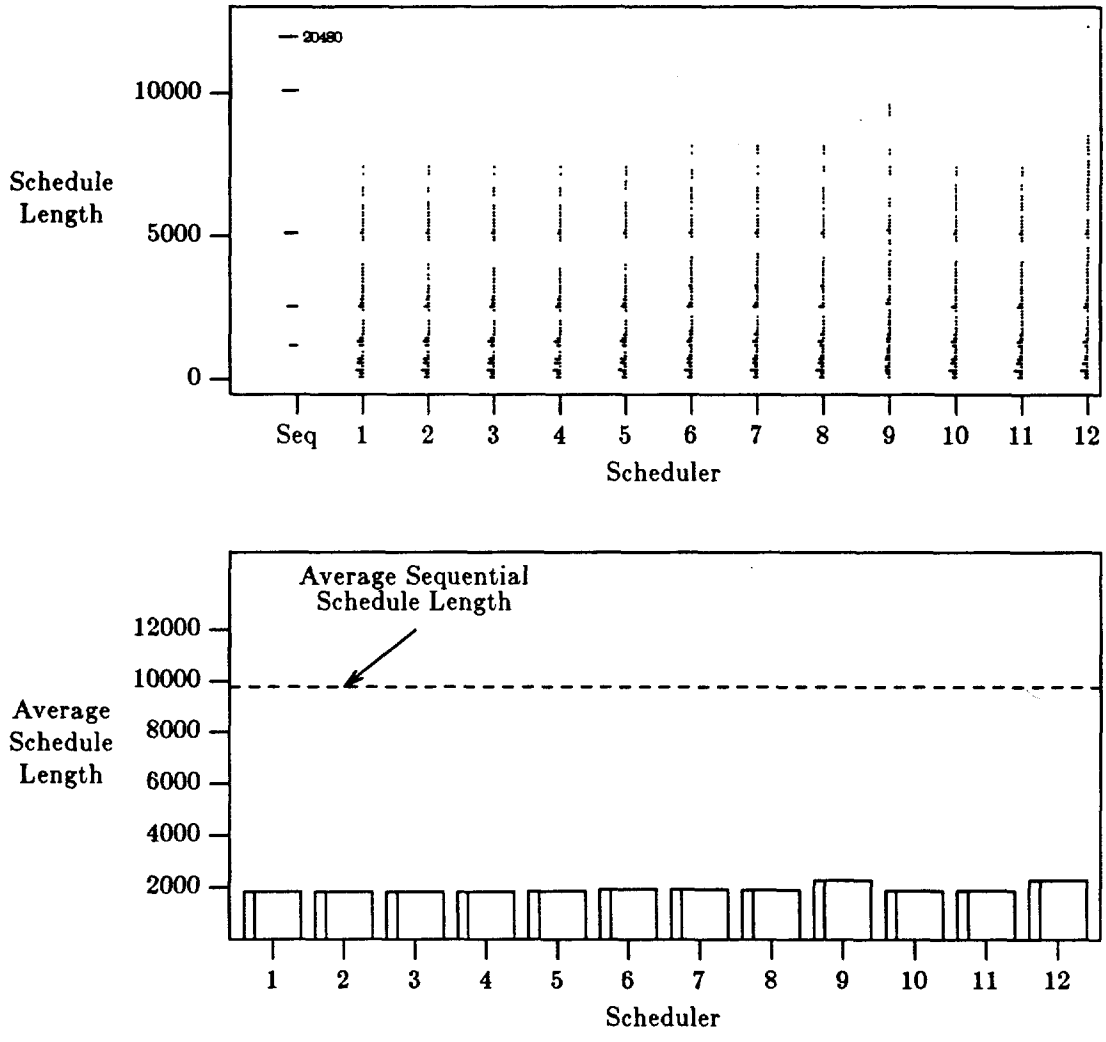
C.5.2. Figure C.25. — Latency = 0.125 (675 Cases)





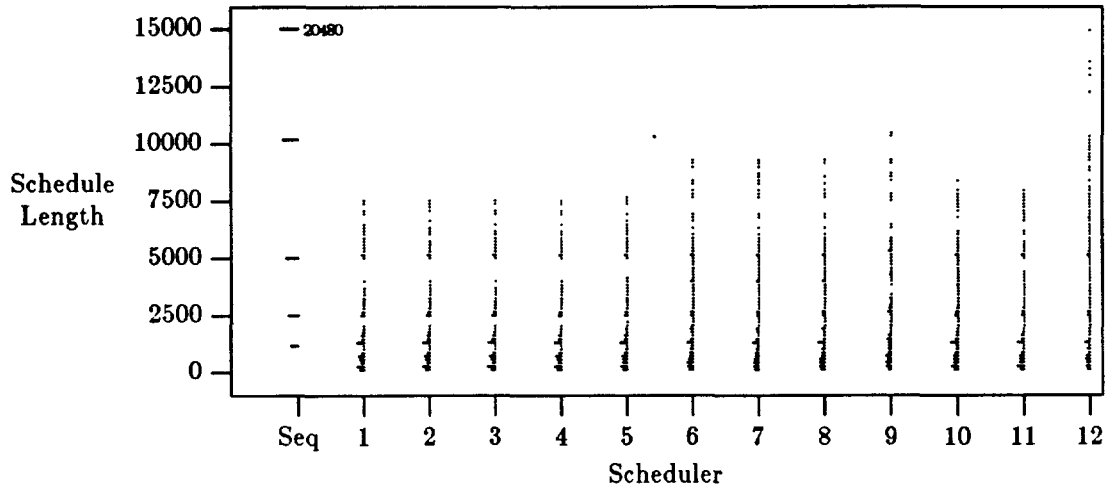
	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
S/P	5.39	5.36	5.39	5.39	5.34	5.24	5.24	5.29	4.41	5.28	5.26	4.66
S/C	5.39	5.36	5.39	5.39	5.34	5.24	5.24	5.29	4.41	5.28	5.26	4.66
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
P Eff	0.77	0.76	0.77	0.77	0.76	0.75	0.75	0.77	0.63	0.76	0.76	0.72
C Eff	0.77	0.76	0.77	0.77	0.76	0.75	0.75	0.77	0.63	0.76	0.76	0.72
CPU Sec	367.55	398.91	783.10	1147.67	3.81	58.07	26.89	93.66	1.23	27.48	57.91	24.19

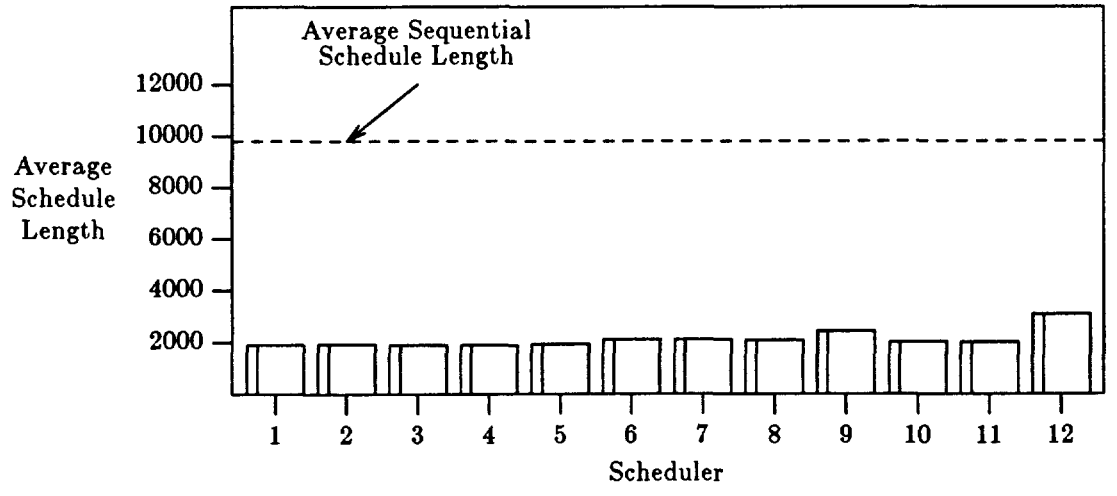
C.5.3. Figure C.26. — Latency = 0.25 (675 Cases)



	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
S/P	5.33	5.29	5.32	5.32	5.25	5.04	5.04	5.08	4.28	5.15	5.15	4.28
S/C	5.33	5.29	5.32	5.32	5.25	5.04	5.04	5.08	4.28	5.15	5.15	4.28
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
P Eff	0.76	0.76	0.76	0.76	0.75	0.73	0.73	0.74	0.61	0.75	0.74	0.68
C Eff	0.76	0.76	0.76	0.76	0.75	0.73	0.73	0.74	0.61	0.75	0.74	0.68
CPU Sec	377.65	408.24	688.41	1065.19	3.80	58.86	27.52	83.21	1.23	27.95	58.45	23.56

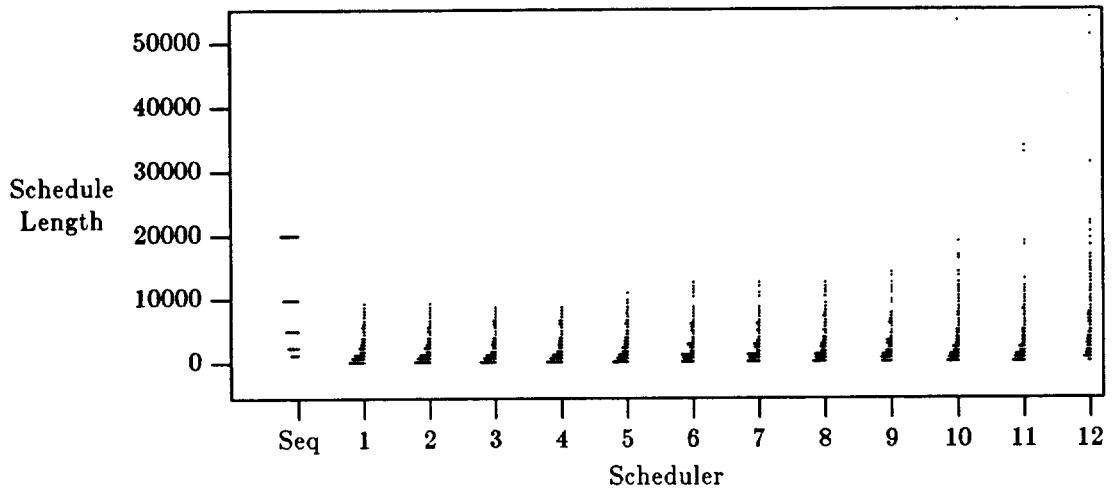
C.5.4. Figure C.27. — Latency = 0.5 (675 Cases)

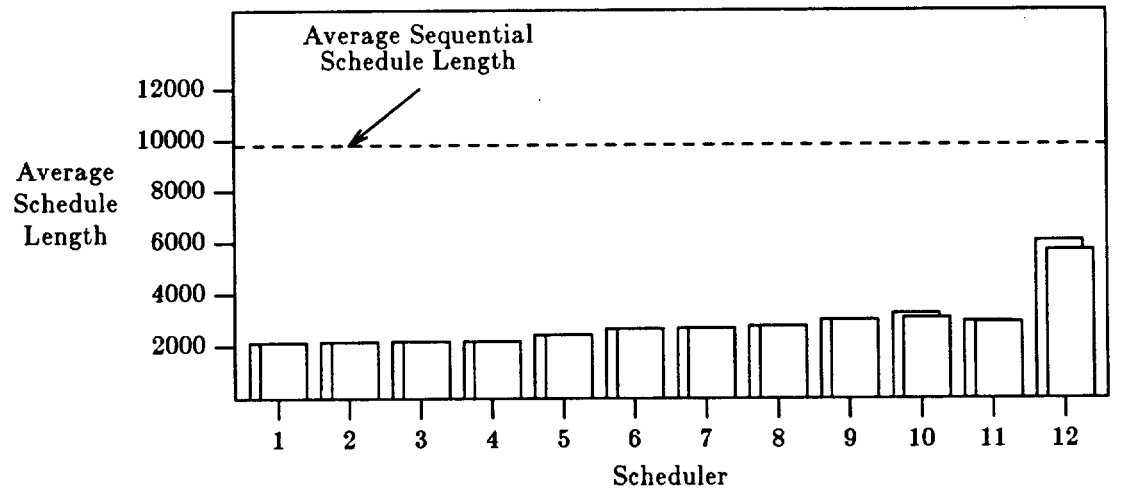




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
S/P	5.15	5.11	5.15	5.15	5.01	4.63	4.62	4.64	4.00	4.82	4.89	3.20
S/C	5.15	5.11	5.15	5.15	5.01	4.63	4.62	4.64	4.00	4.82	4.89	3.20
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
P Eff	0.74	0.74	0.74	0.74	0.73	0.69	0.69	0.69	0.59	0.71	0.72	0.54
C Eff	0.74	0.74	0.74	0.74	0.73	0.69	0.69	0.69	0.59	0.71	0.72	0.54
CPU Sec	377.22	408.83	513.20	908.85	3.78	58.97	27.56	61.73	1.23	27.80	58.33	20.50

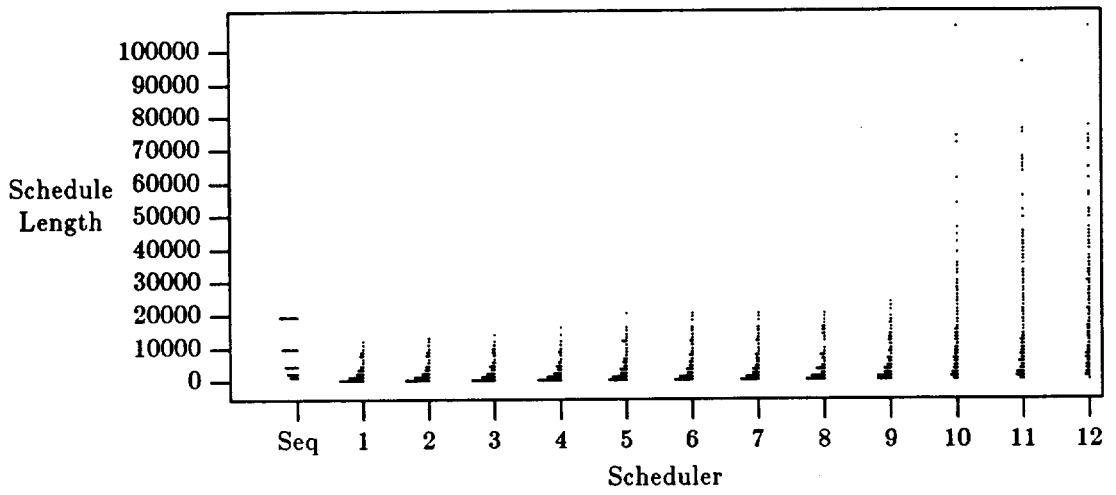
C.5.5. Figure C.28. — Latency = 1 (675 Cases)

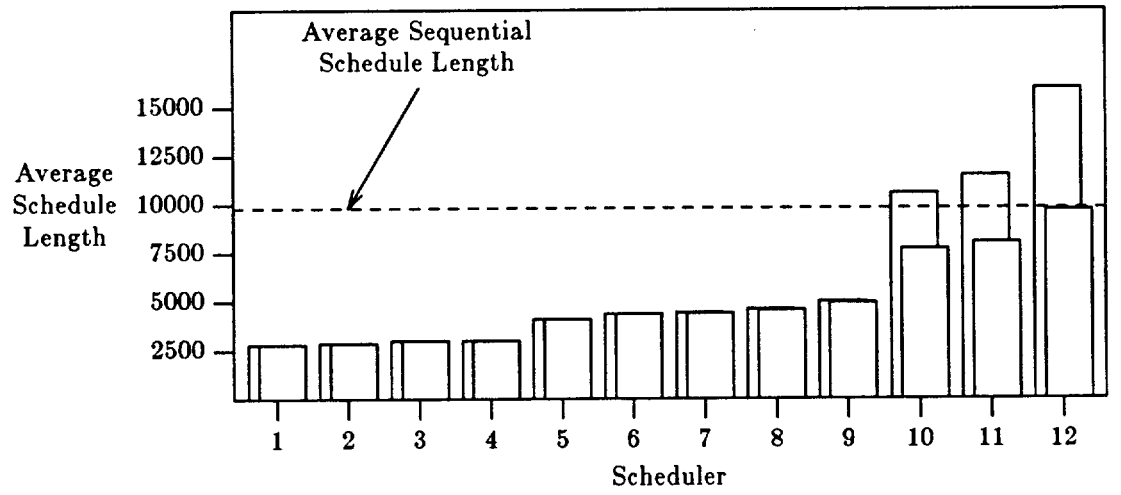




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	98.52	99.41	95.56
S/P	4.63	4.58	4.48	4.48	4.04	3.68	3.66	3.55	3.25	3.00	3.32	1.62
S/C	4.63	4.58	4.48	4.48	4.04	3.68	3.66	3.55	3.25	3.18	3.37	1.73
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.06	1.01	1.07
P Eff	0.68	0.67	0.66	0.66	0.61	0.57	0.57	0.56	0.50	0.52	0.54	0.26
C Eff	0.68	0.67	0.66	0.66	0.61	0.57	0.57	0.56	0.50	0.52	0.54	0.26
CPU Sec	373.21	405.98	404.12	807.54	3.76	59.53	27.91	50.06	1.23	27.87	58.15	16.88

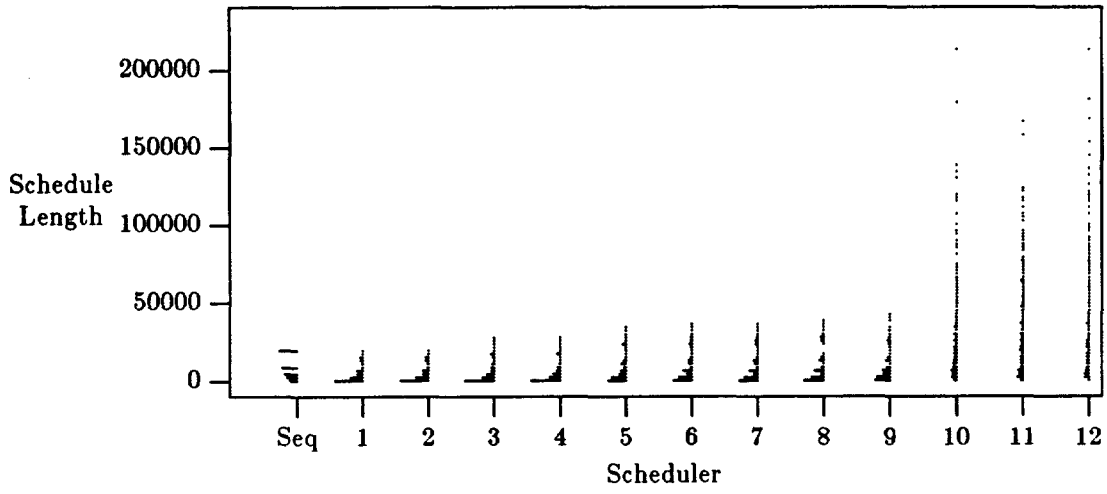
C.5.6. Figure C.29. — Latency = 2 (675 Cases)

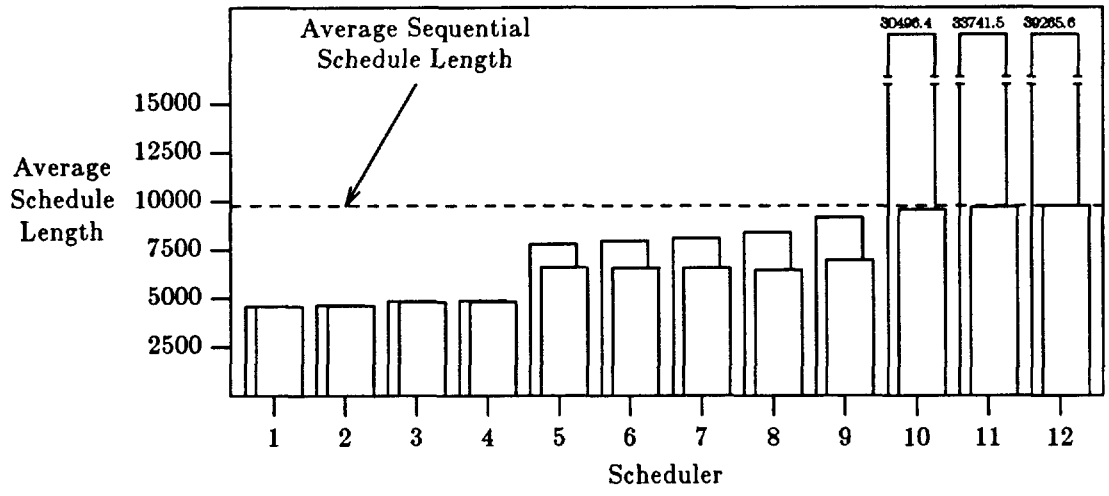




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	100.00	100.00	100.00	100.00	98.67	98.96	97.63	97.33	93.33	43.70	38.96	6.96
S/P	3.49	3.42	3.28	3.28	2.40	2.25	2.22	2.14	1.97	0.93	0.85	0.61
S/C	3.49	3.42	3.28	3.28	2.40	2.25	2.22	2.14	1.99	1.28	1.22	1.01
P/C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.37	1.43	1.65
P Eff	0.53	0.52	0.51	0.51	0.38	0.37	0.36	0.37	0.31	0.16	0.16	0.09
C Eff	0.53	0.52	0.51	0.51	0.38	0.37	0.37	0.37	0.31	0.19	0.19	0.15
CPU Sec	364.08	397.70	379.77	786.65	3.72	59.83	28.35	47.28	1.23	25.88	54.14	12.14

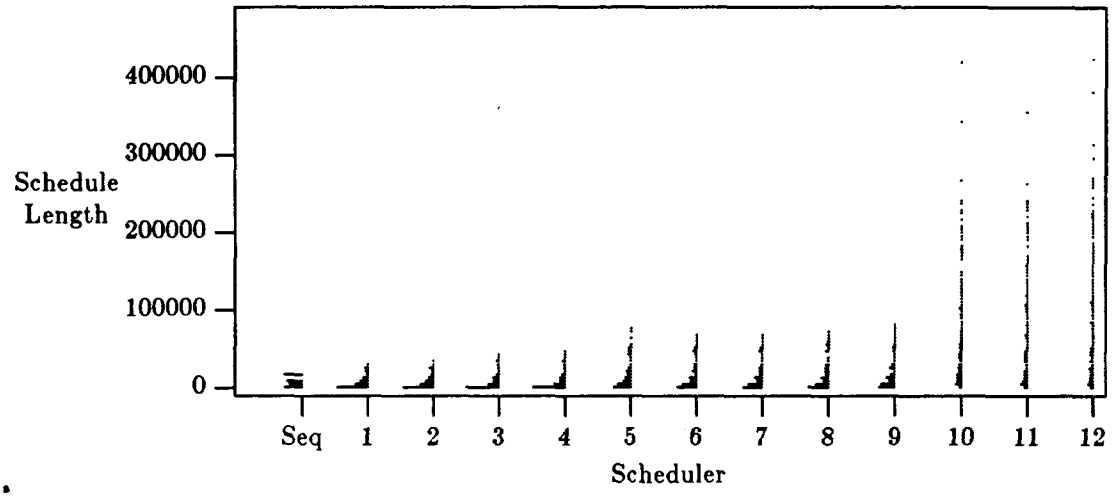
C.5.7. Figure C.30. — Latency = 4 (675 Cases)

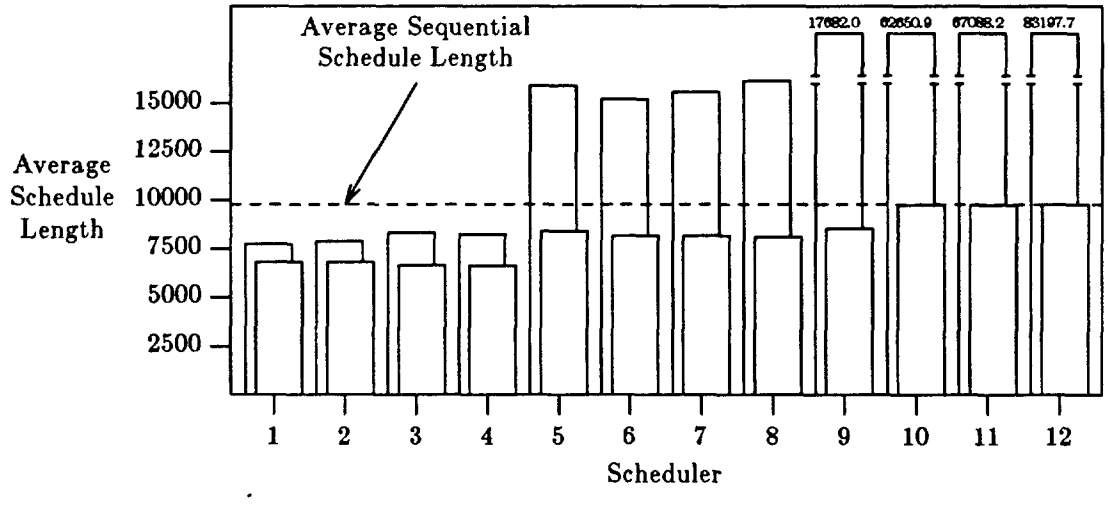




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	99.41	98.52	94.67	94.96	51.56	53.48	53.33	54.81	50.96	5.48	1.33	0.00
S/P	2.13	2.11	2.01	2.01	1.25	1.23	1.21	1.16	1.06	0.32	0.29	0.25
S/C	2.13	2.11	2.04	2.03	1.48	1.49	1.48	1.51	1.40	1.02	1.00	1.00
P/C	1.00	1.00	1.01	1.01	1.18	1.21	1.23	1.30	1.32	3.18	3.47	4.02
P Eff	0.33	0.32	0.34	0.34	0.20	0.20	0.20	0.21	0.17	0.06	0.05	0.04
C Eff	0.33	0.32	0.34	0.34	0.22	0.22	0.22	0.24	0.20	0.15	0.15	0.15
CPU Sec	357.92	388.44	380.50	789.18	3.67	60.06	28.65	47.46	1.23	19.92	48.40	9.66

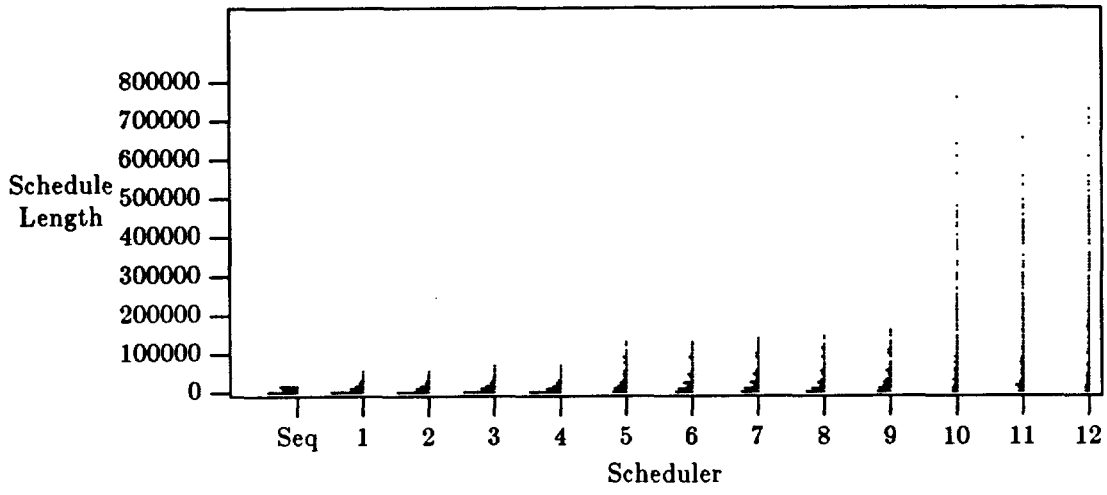
C.5.8. Figure C.31. — Latency = 8 (675 Cases)

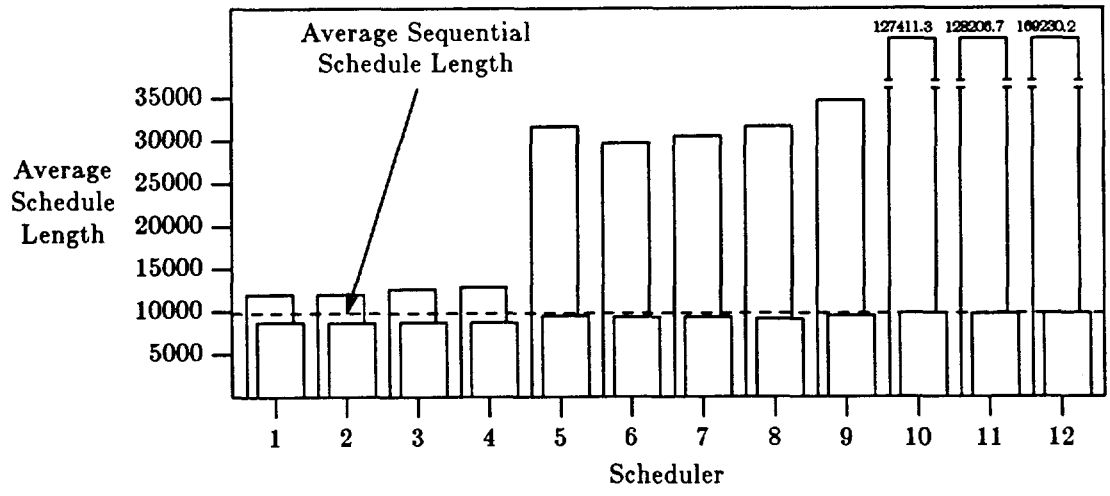




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	65.63	68.15	67.70	69.33	29.78	32.59	31.70	34.07	28.74	1.78	2.22	1.33
S/P	1.26	1.24	1.17	1.19	0.61	0.64	0.63	0.61	0.55	0.16	0.15	0.12
S/C	1.43	1.43	1.47	1.47	1.16	1.20	1.19	1.21	1.14	1.00	1.00	1.00
P/C	1.14	1.16	1.25	1.24	1.89	1.86	1.90	1.99	2.07	6.42	6.88	8.52
P Eff	0.18	0.18	0.19	0.19	0.10	0.10	0.10	0.11	0.09	0.03	0.03	0.02
C Eff	0.20	0.20	0.22	0.22	0.16	0.17	0.17	0.17	0.16	0.15	0.15	0.15
CPU Sec	351.85	379.80	389.73	799.60	3.60	60.02	28.67	47.84	1.23	17.75	45.42	8.77

C.5.9. Figure C.32. — Latency = 16 (675 Cases)

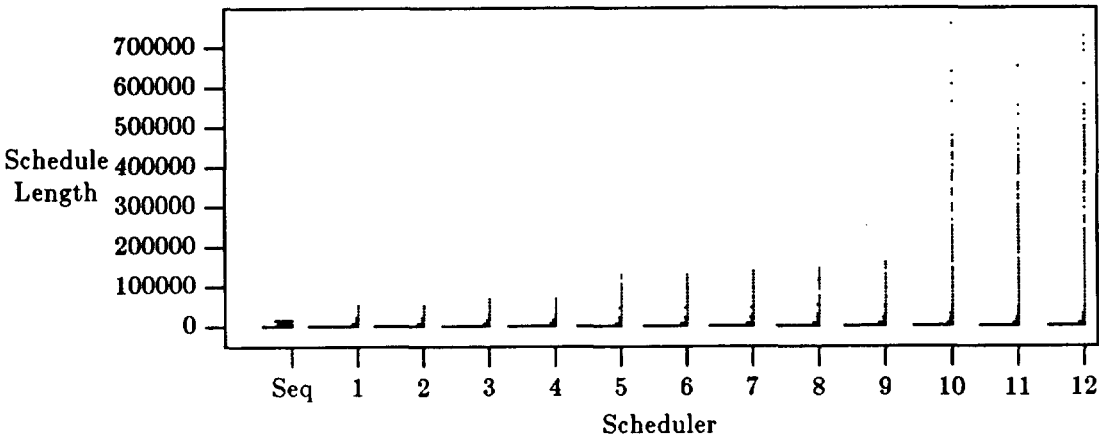


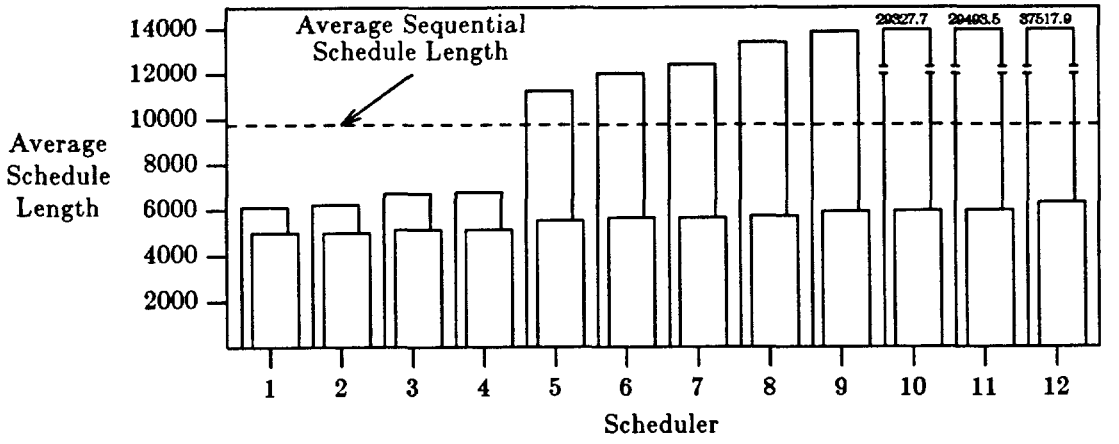


	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	36.59	46.96	36.89	37.63	11.56	11.41	11.26	13.63	8.00	5.78	7.11	5.78
S/P	0.82	0.82	0.78	0.76	0.31	0.33	0.32	0.31	0.28	0.08	0.08	0.06
S/C	1.12	1.13	1.13	1.13	1.04	1.05	1.05	1.08	1.04	1.01	1.01	1.01
P/C	1.37	1.38	1.45	1.47	3.35	3.20	3.28	3.48	3.67	13.11	13.23	17.41
P Eff	0.12	0.12	0.12	0.12	0.05	0.05	0.05	0.05	0.04	0.03	0.03	0.02
C Eff	0.16	0.16	0.16	0.16	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15
CPU Sec	331.47	354.74	399.73	806.52	3.50	59.78	28.69	48.01	1.23	16.75	43.83	8.19

C.6. Comparison By Processor Count

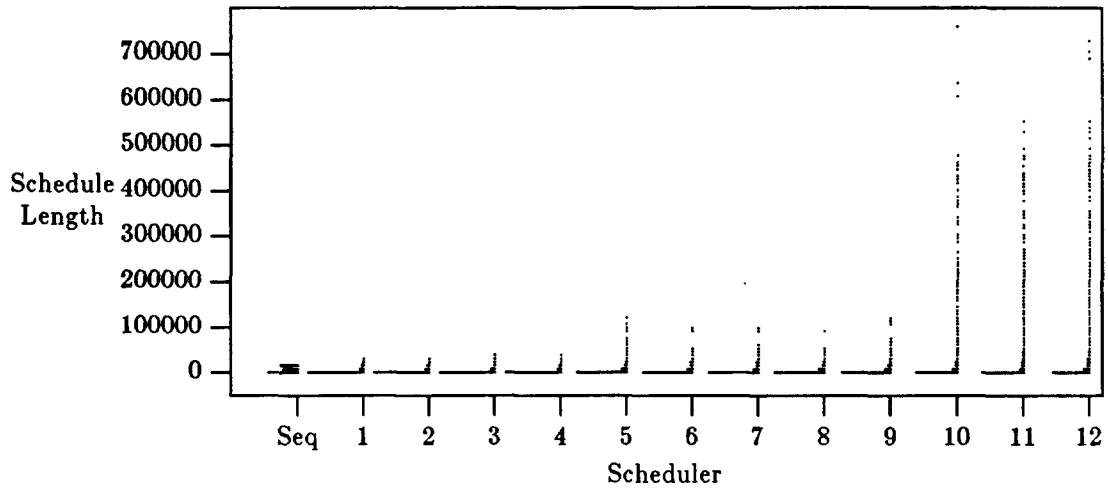
C.6.1. Figure C.33. — Processor count = 4 (2025 Cases)

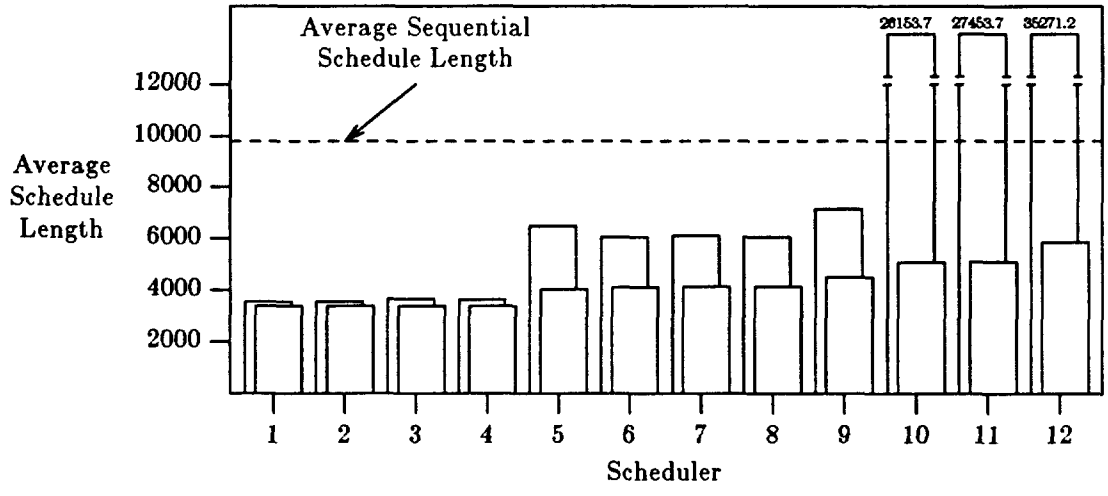




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	79.90	81.09	78.62	79.36	67.31	66.37	65.88	66.27	64.54	58.96	58.77	55.70
S/P	1.59	1.56	1.45	1.43	0.87	0.81	0.79	0.73	0.70	0.33	0.33	0.26
S/C	1.95	1.95	1.89	1.90	1.76	1.73	1.72	1.70	1.64	1.63	1.63	1.54
P/C	1.23	1.25	1.31	1.32	2.02	2.13	2.19	2.33	2.33	4.90	4.91	5.91
P Eff	0.63	0.63	0.61	0.61	0.56	0.54	0.54	0.53	0.48	0.50	0.50	0.46
C Eff	0.65	0.64	0.63	0.63	0.60	0.59	0.59	0.58	0.53	0.57	0.57	0.54
CPU Sec	22.33	54.02	26.53	401.09	3.09	37.83	6.58	12.64	1.23	10.21	40.00	6.32

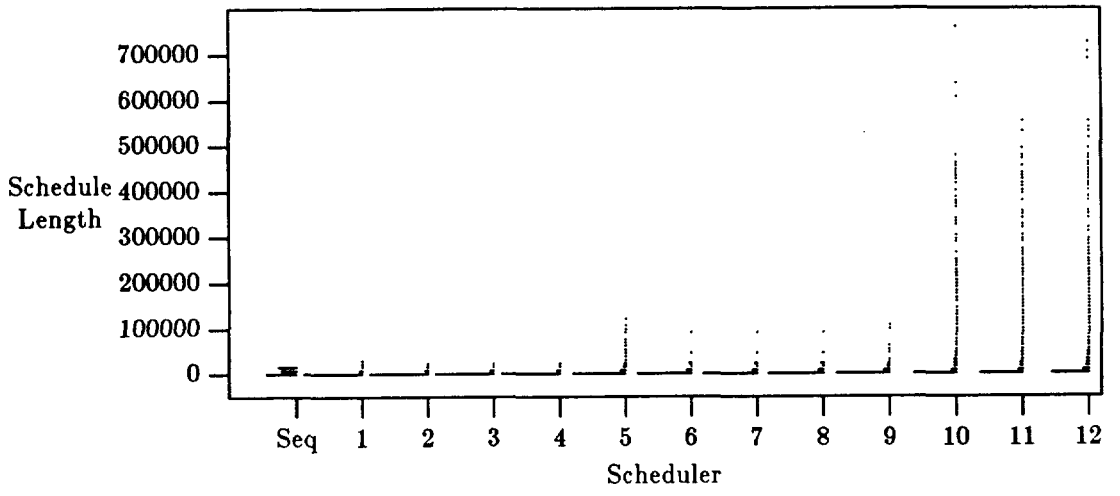
C.6.2. Figure C.34. — Processor count = 8 (2025 Cases)

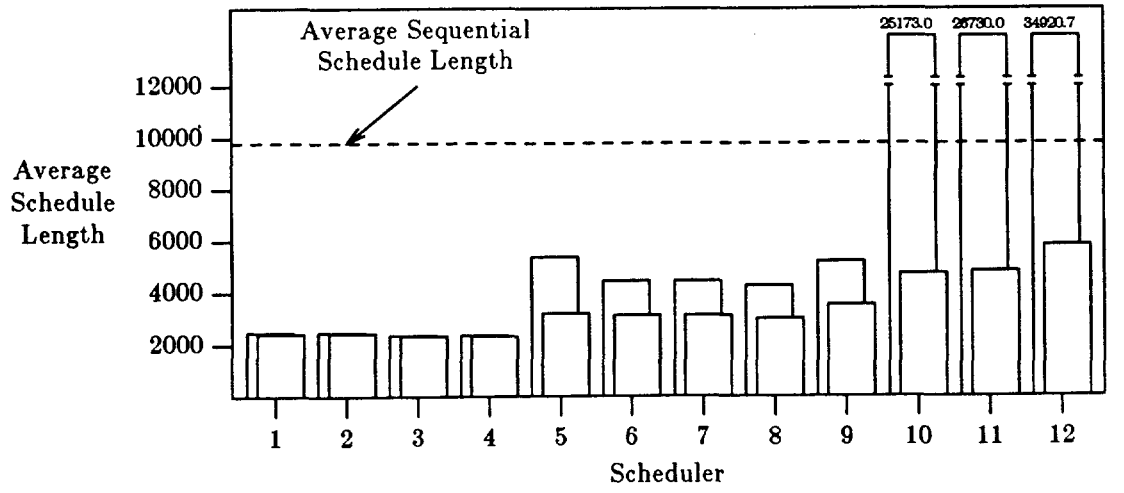




	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	90.47	91.75	90.37	90.67	80.00	79.75	79.46	80.44	77.78	62.37	61.83	56.64
S/P	2.75	2.75	2.69	2.70	1.51	1.62	1.60	1.62	1.37	0.37	0.36	0.28
S/C	2.90	2.89	2.90	2.90	2.43	2.38	2.37	2.38	2.17	1.92	1.92	1.67
P/C	1.05	1.05	1.08	1.07	1.61	1.47	1.48	1.47	1.59	5.15	5.39	6.02
P Eff	0.55	0.55	0.55	0.55	0.50	0.48	0.48	0.49	0.41	0.43	0.43	0.35
C Eff	0.56	0.55	0.56	0.56	0.51	0.50	0.49	0.50	0.42	0.46	0.47	0.38
CPU Sec	110.17	142.68	146.93	520.76	3.55	46.06	14.85	30.62	1.23	16.09	45.74	11.51

C.6.3. Figure C.35. — Processor count = 16 (2025 Cases)





	Scheduler											
	1	2	3	4	5	6	7	8	9	10	11	12
%P≤S	96.84	98.37	97.43	97.28	83.21	86.02	85.98	86.57	84.69	63.75	62.42	57.53
S/P	3.98	3.98	4.13	4.12	1.82	2.19	2.20	2.29	1.88	0.39	0.37	0.28
S/C	4.05	4.01	4.17	4.16	3.04	3.10	3.10	3.24	2.75	2.06	2.04	1.68
P/C	1.02	1.01	1.01	1.01	1.67	1.41	1.41	1.42	1.46	5.31	5.58	6.01
P Eff	0.44	0.44	0.45	0.45	0.39	0.39	0.39	0.40	0.31	0.33	0.33	0.24
C Eff	0.44	0.44	0.45	0.45	0.40	0.39	0.39	0.41	0.31	0.34	0.34	0.26
CPU Sec	913.15	940.55	1235.87	1679.76	4.50	90.50	59.04	127.67	1.23	43.99	72.43	30.01

APPENDIX D

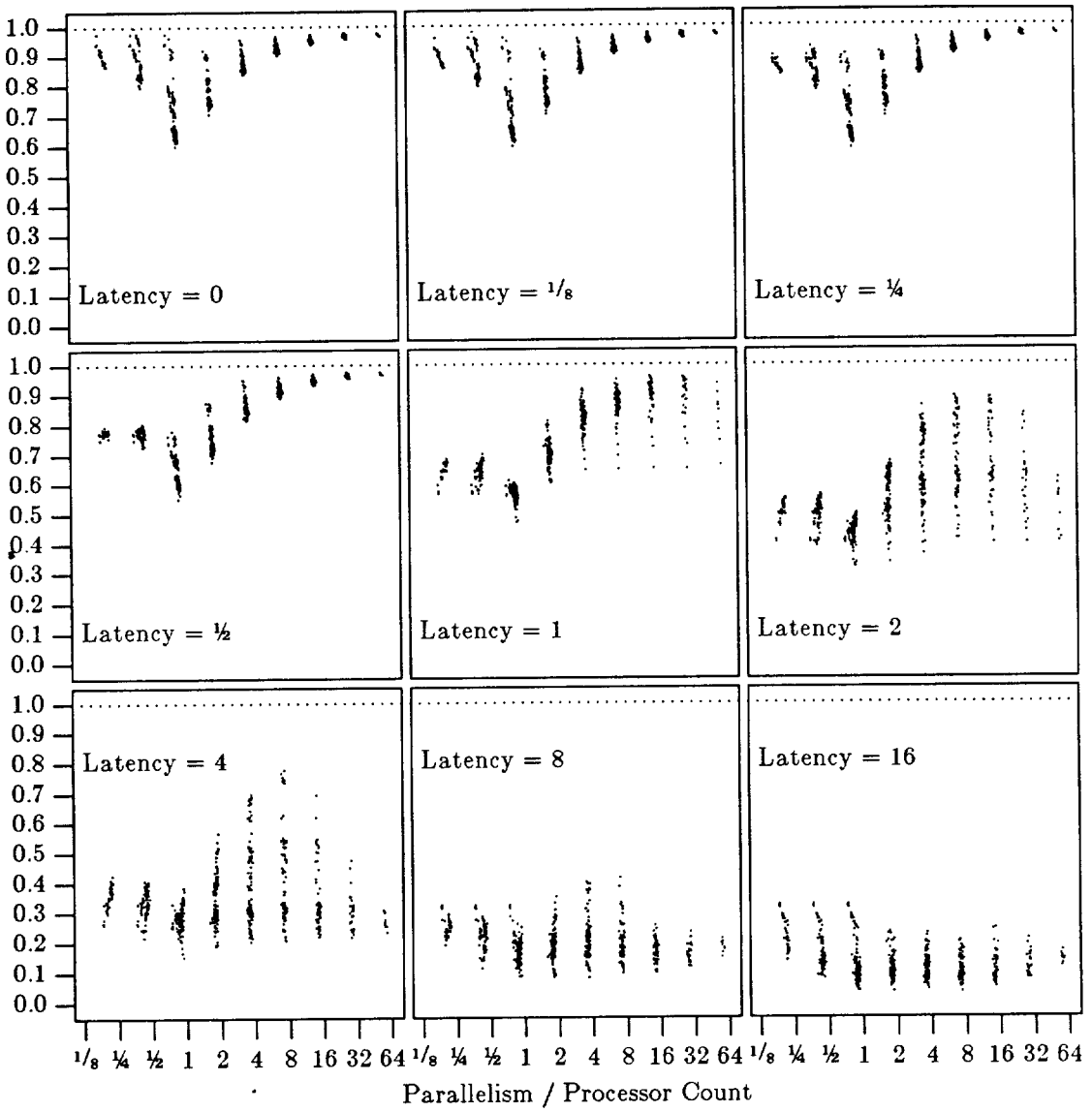
Relative Efficiencies of Schedulers

This appendix gives plots of the relative efficiencies of different schedules. The relative efficiency of a schedule is defined as $\frac{T_s}{\min(p,n)\times T_p}$. T_s is the length of a sequential schedule for the program, and T_p is the length of the parallel schedule. The values p and n are the average parallelism in the program and the number of processors in the machine, respectively.

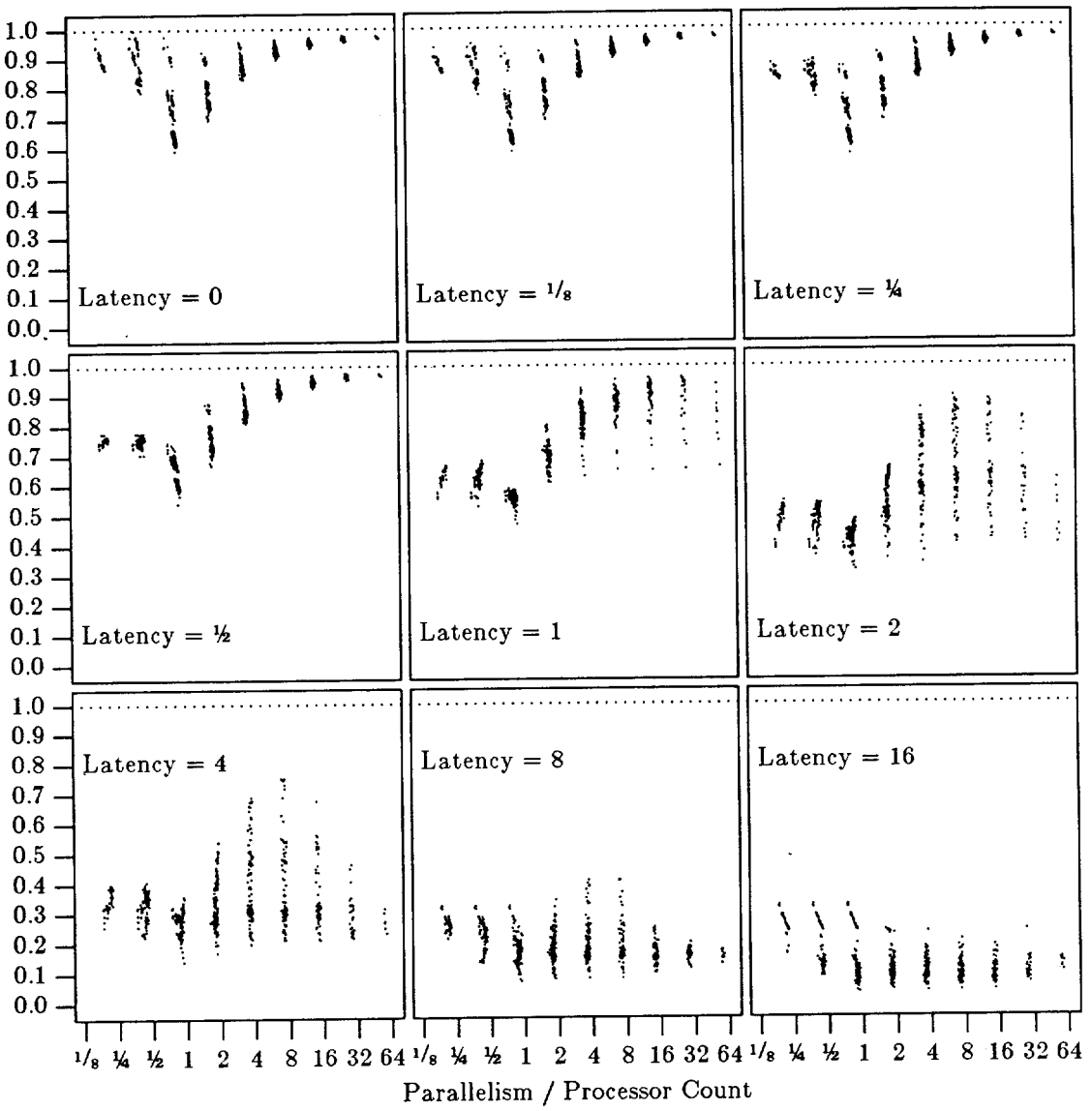
Relative efficiency has the advantage over $\frac{T_s}{n\times T_p}$ in that the relative efficiency does not penalize a schedule for having more processors than the problem can actually keep busy. For example, if a program has an average parallelism of 2, then no scheduler will ever have a parallel speedup that exceeds 2, no matter how many processors are available.

Conversely, the fact that a particular program has an average parallelism of 2 does not imply that there exists a two-processor schedule which gives that parallel parallel speedup. There could easily be precedence constraints which make all the parallelism available at the same instant in time. This would mean that half of the graph has no parallelism, and half has lots of parallelism. In order to get the speedup of 2, many processors would have to be available for the short time when the parallelism is available.

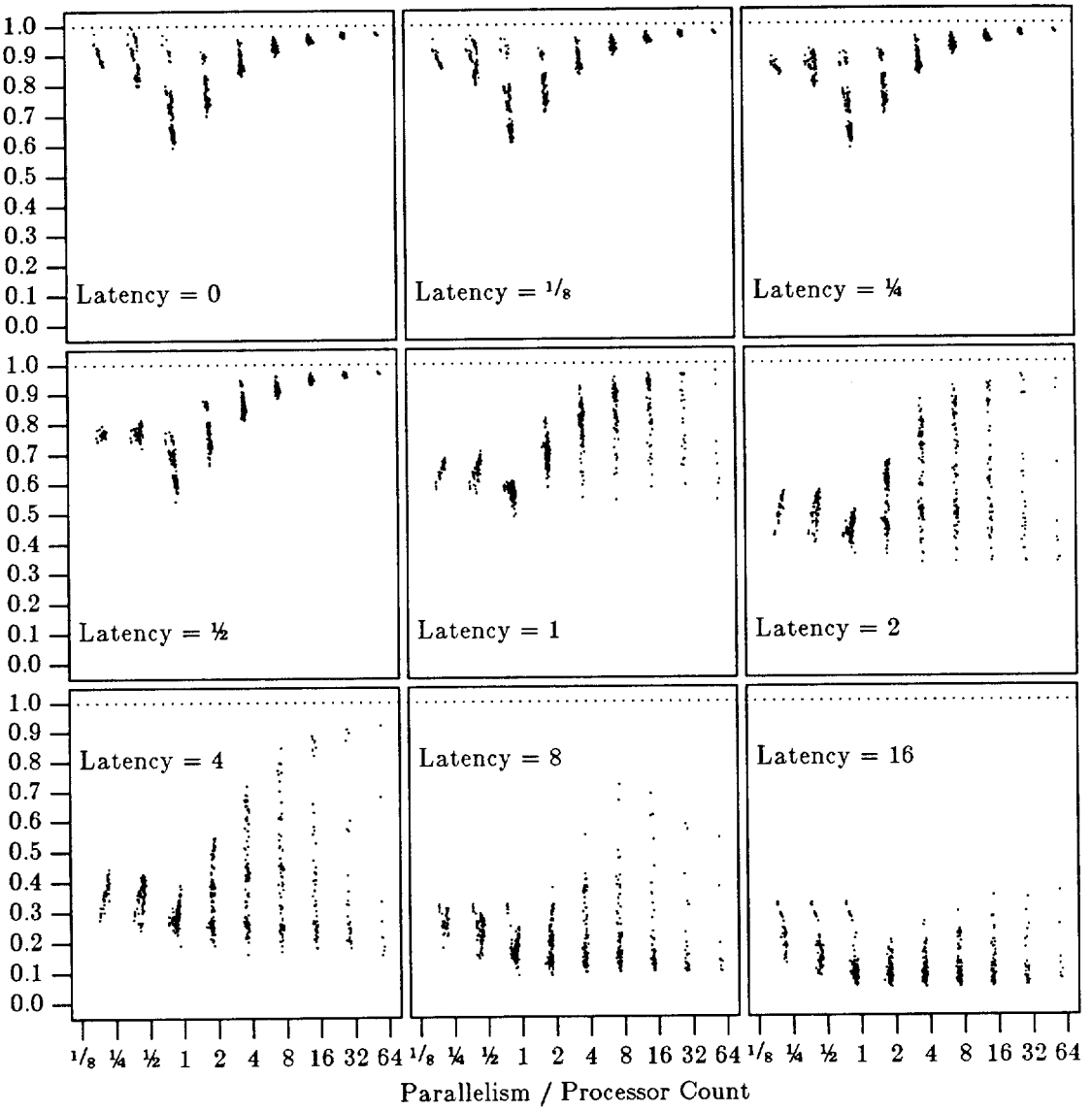
Each page in this appendix contains nine plots, representing the relative efficiencies of programs as the communication latency varies from 0 to 16. The x-axis represents the average parallelism relative to the number of processors in the system. The y-axis gives the relative efficiency.



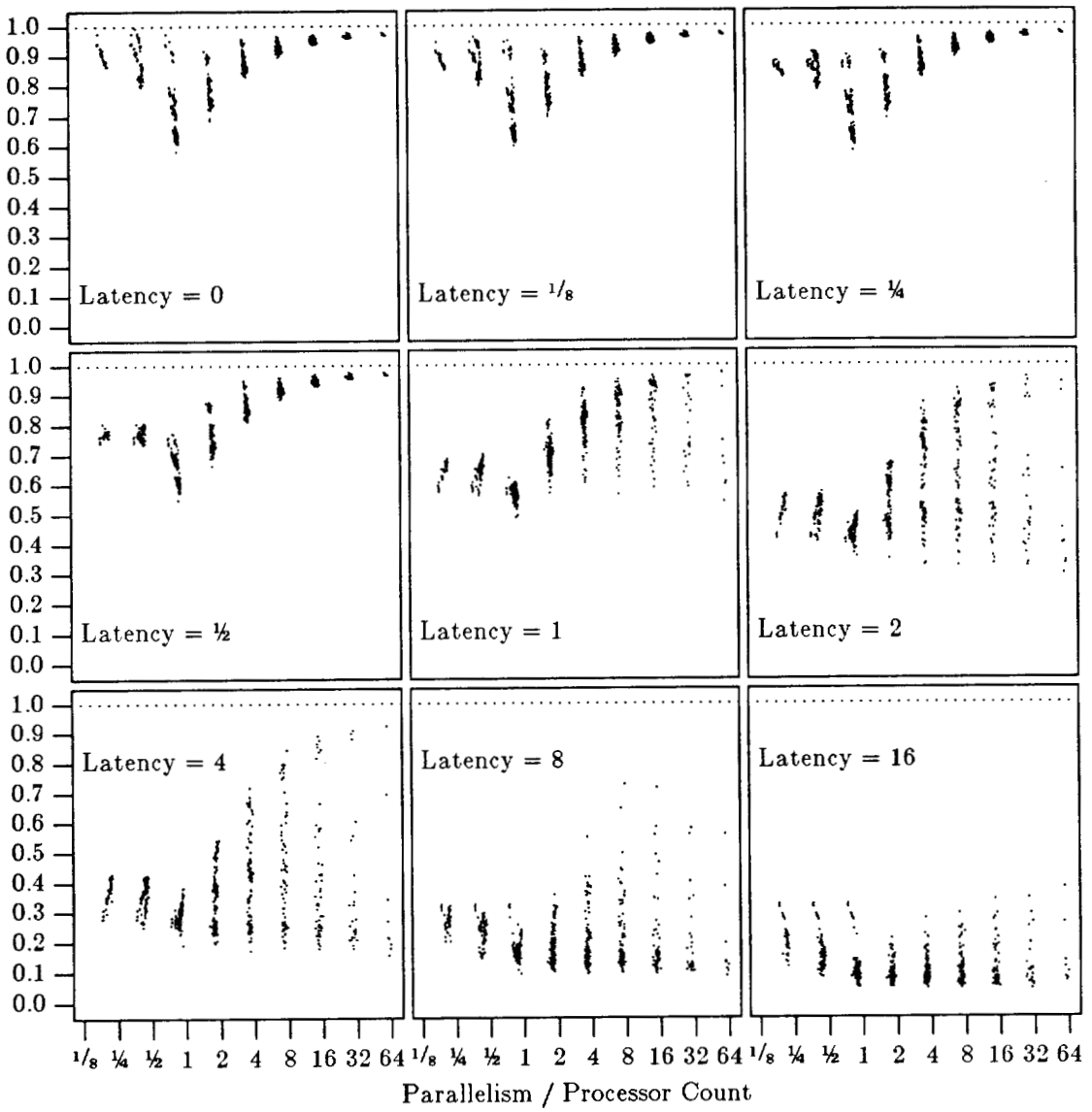
Scheduler 1 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



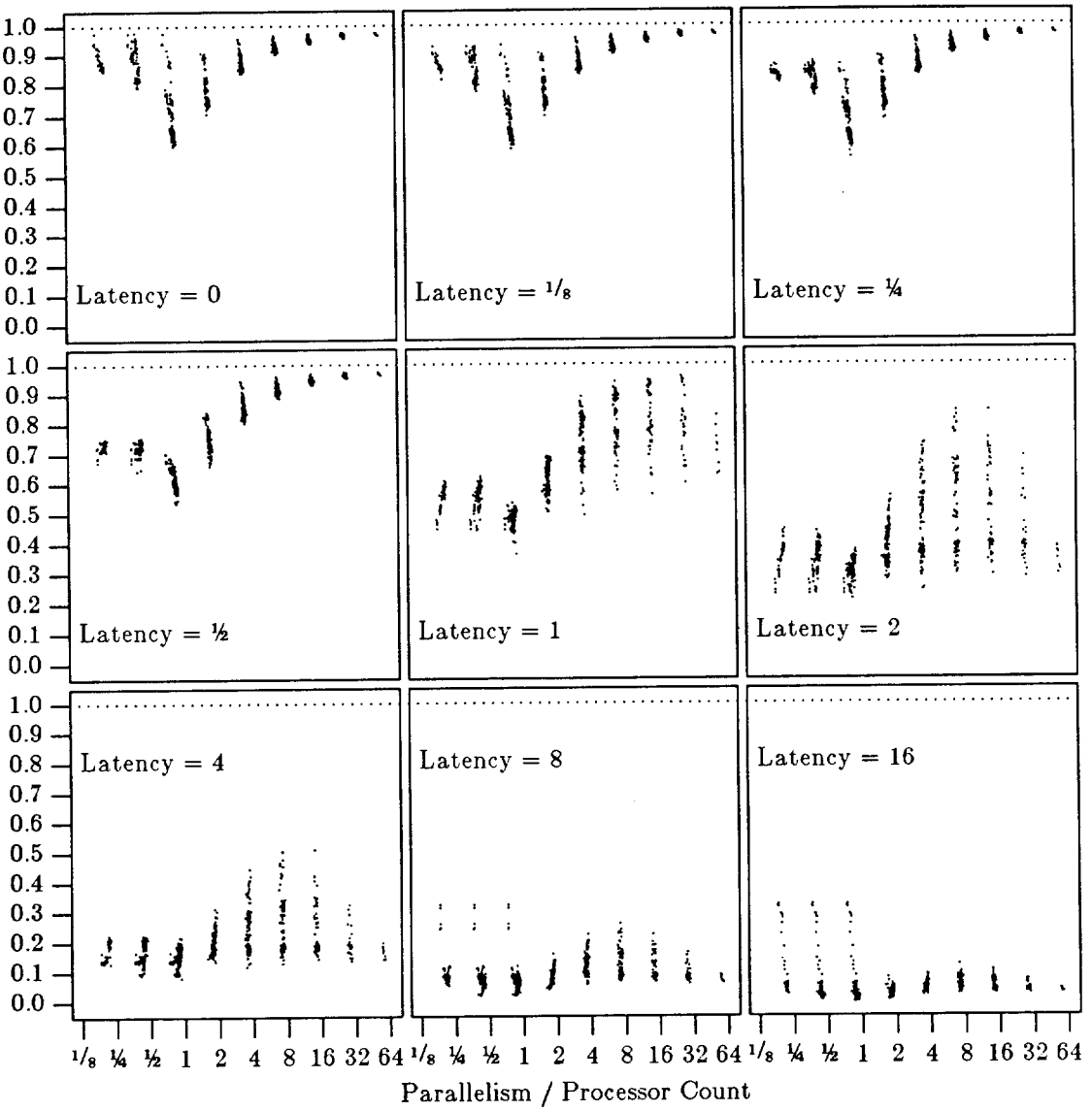
Scheduler 2 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



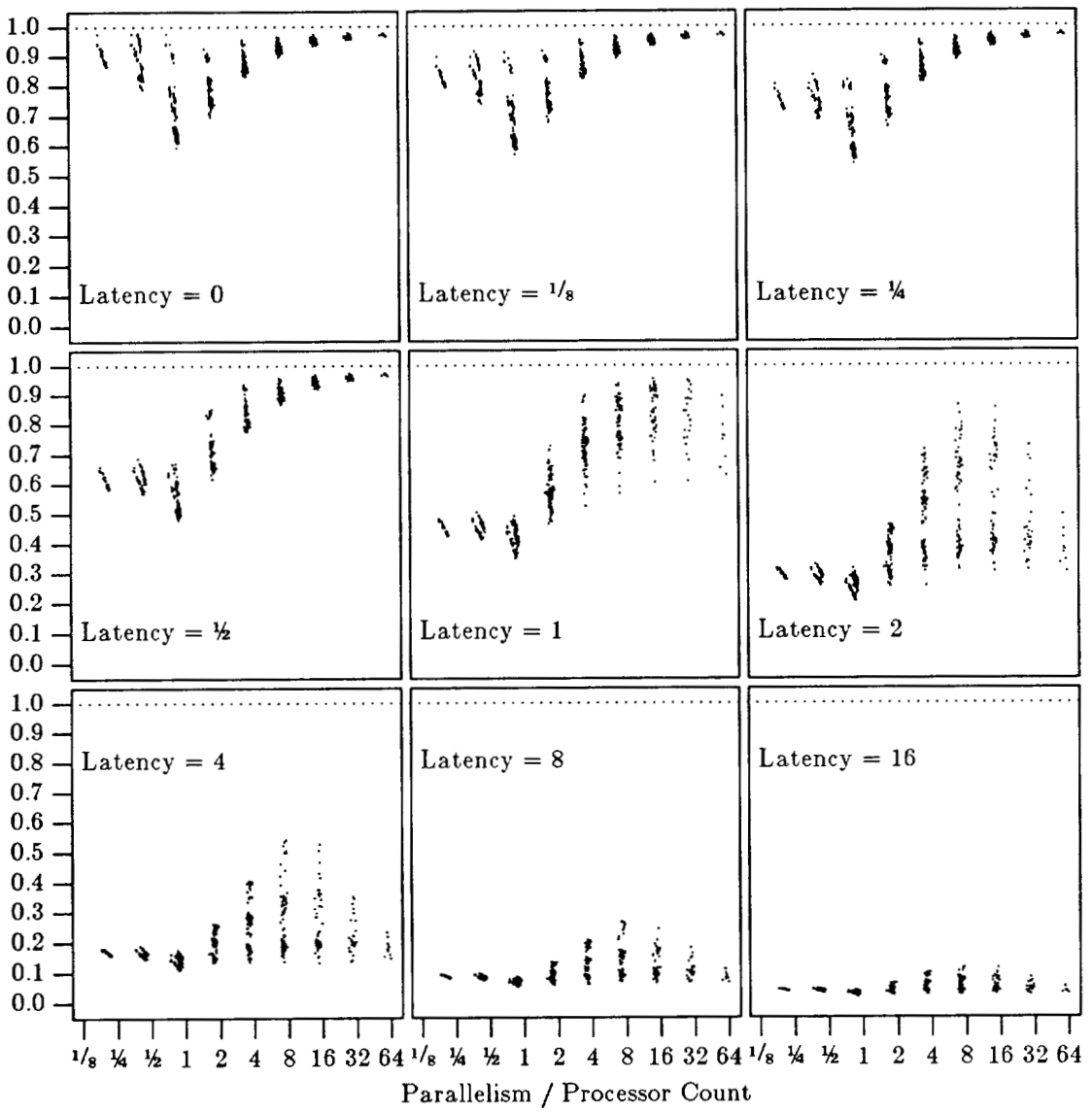
Scheduler 3 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



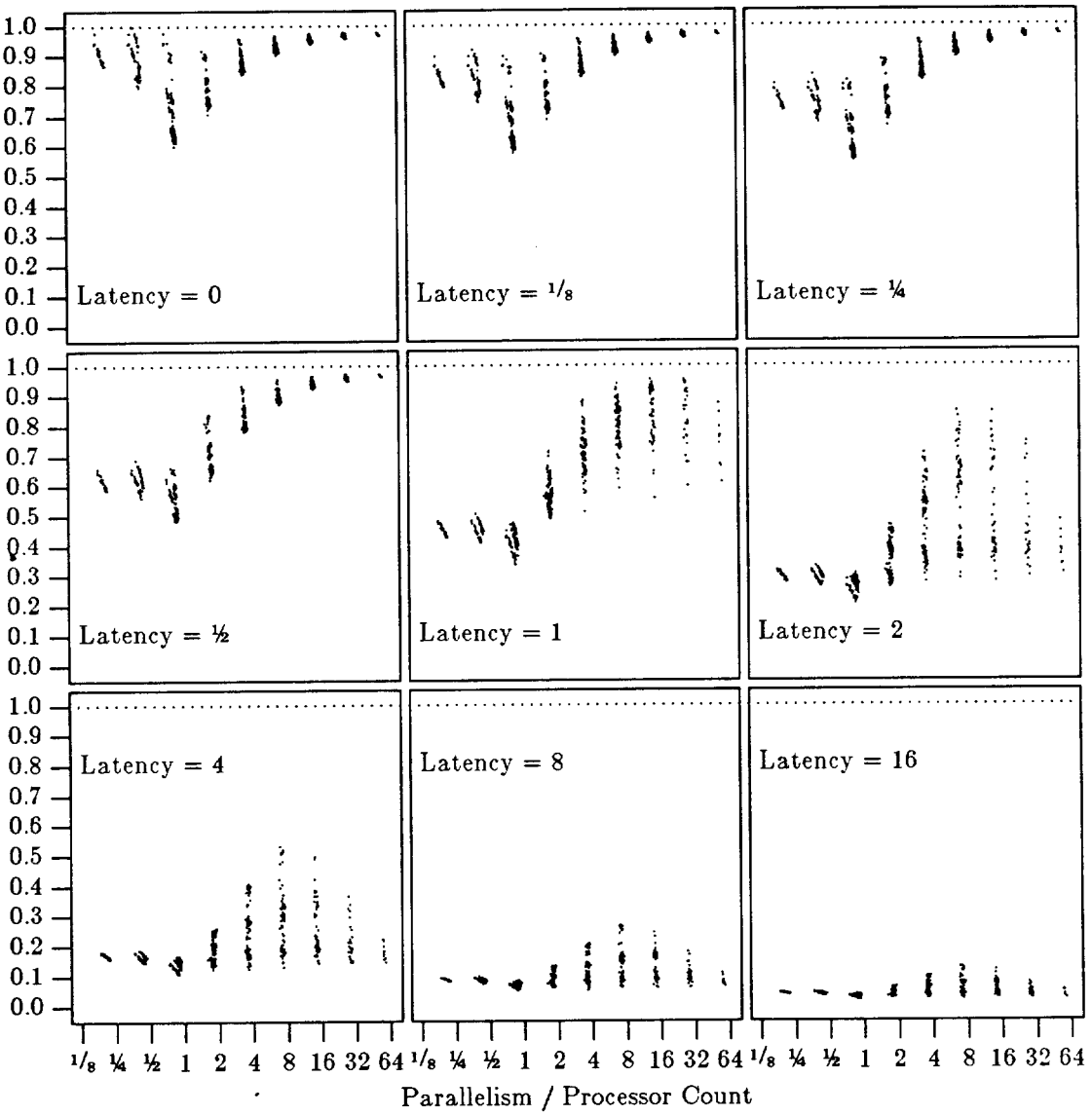
Scheduler 4 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



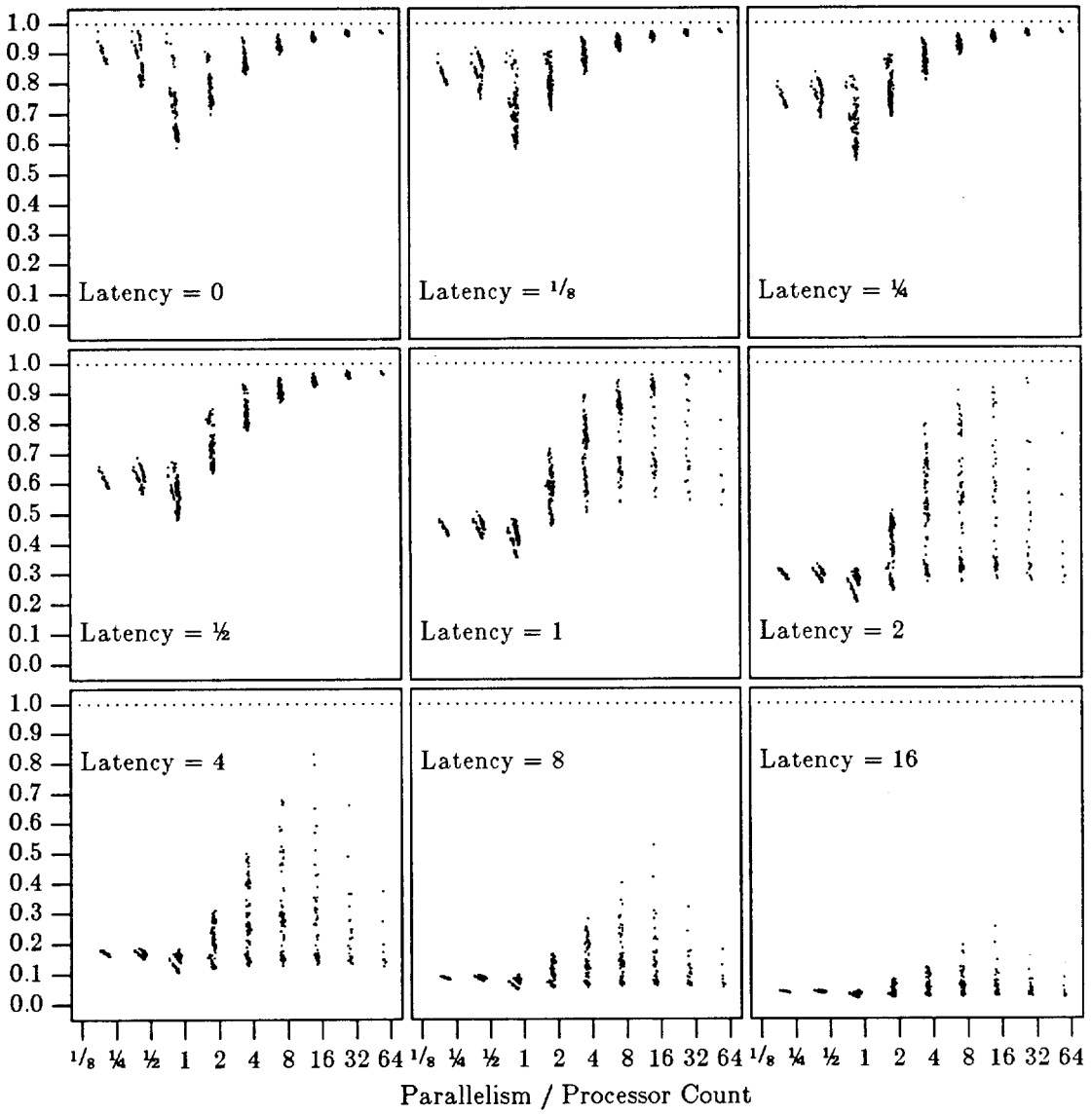
Scheduler 5 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



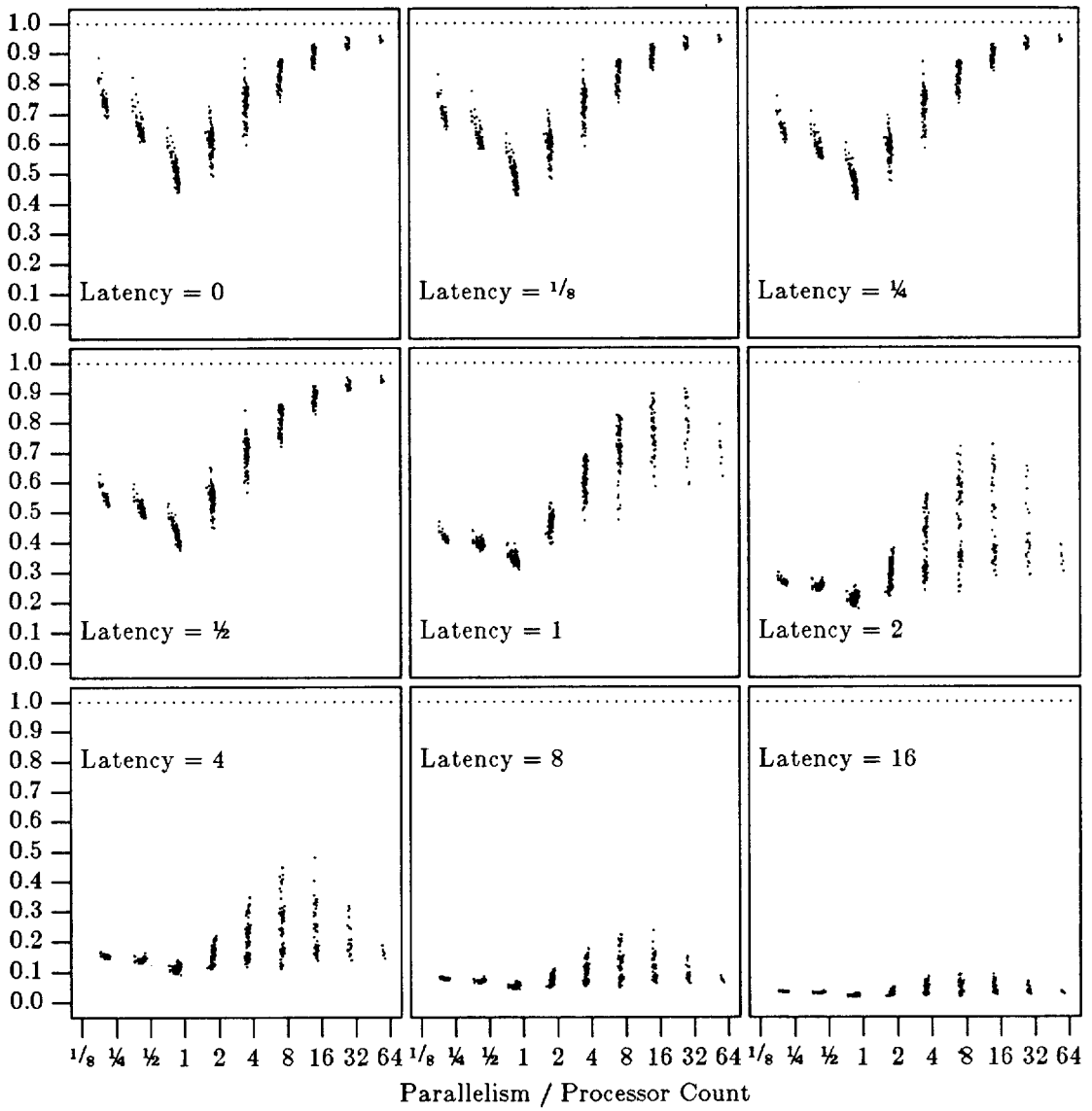
Scheduler 6 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



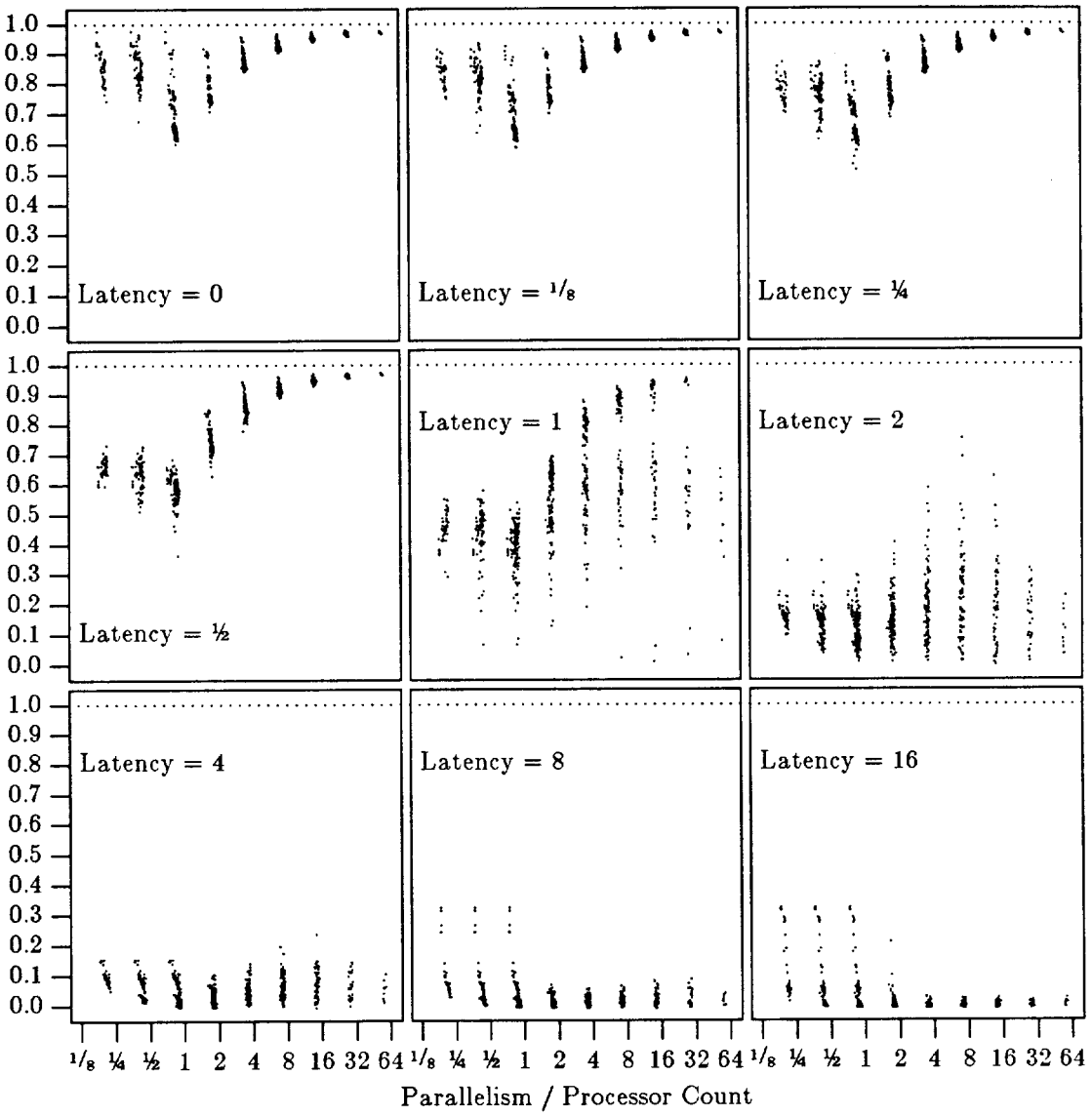
Scheduler 7 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



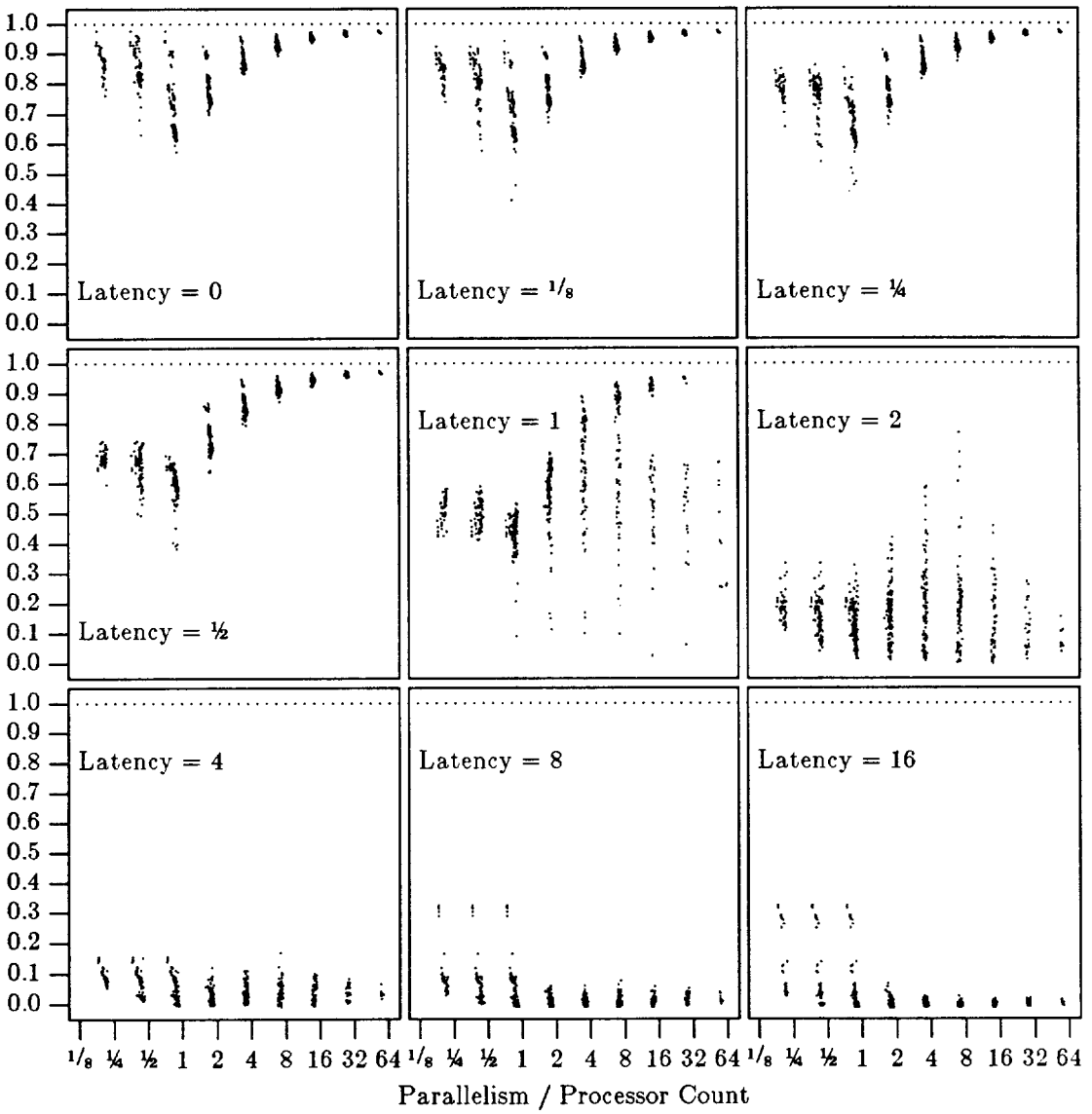
Scheduler 8 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



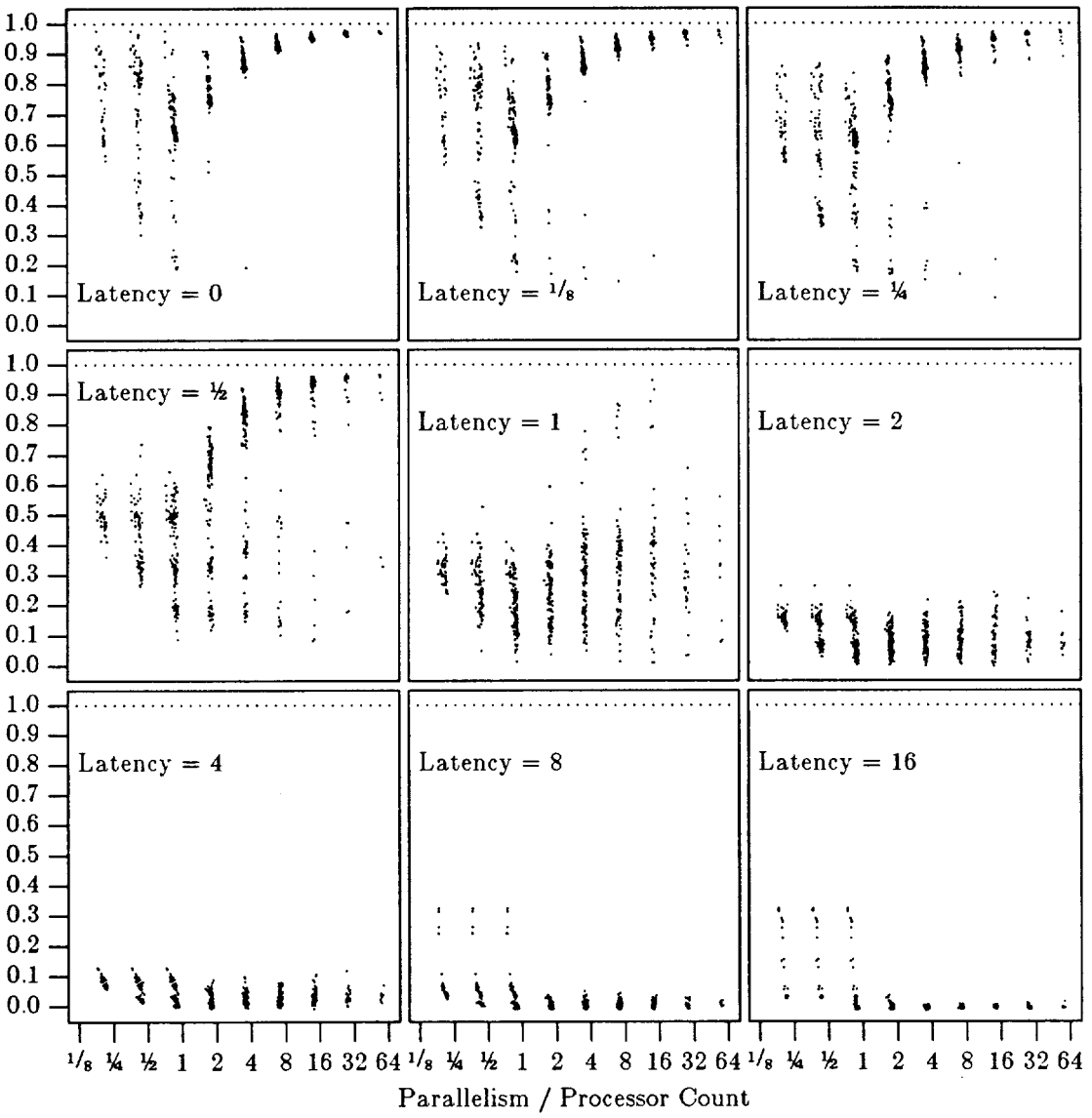
Scheduler 9 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



Scheduler 10 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



Scheduler 11 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$



Scheduler 12 — $\frac{p}{n}$ vs. $\frac{T_s}{\min(p,n)\times T_p}$

APPENDIX E

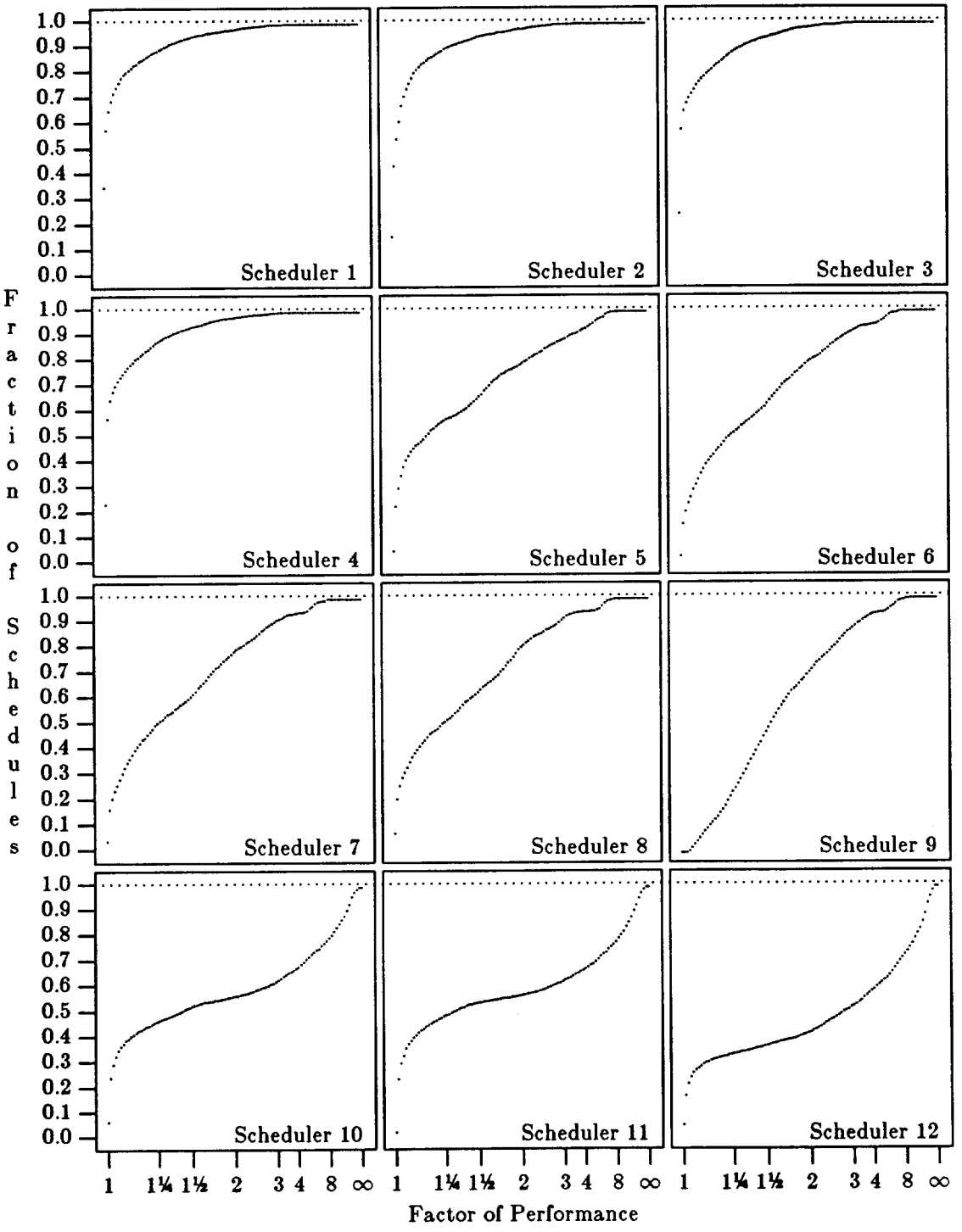
Cumulative Histograms of Relative Performance

An important method of displaying scheduler behavior is through the use of cumulative histograms. A cumulative histogram is different from other histograms in that each column is the sum of all sample values that occur to its left. It is, in effect, the integration of the curve described by a common histogram. The main advantage over a common histogram is that the shape of a cumulative histogram is insensitive to the width of its columns.

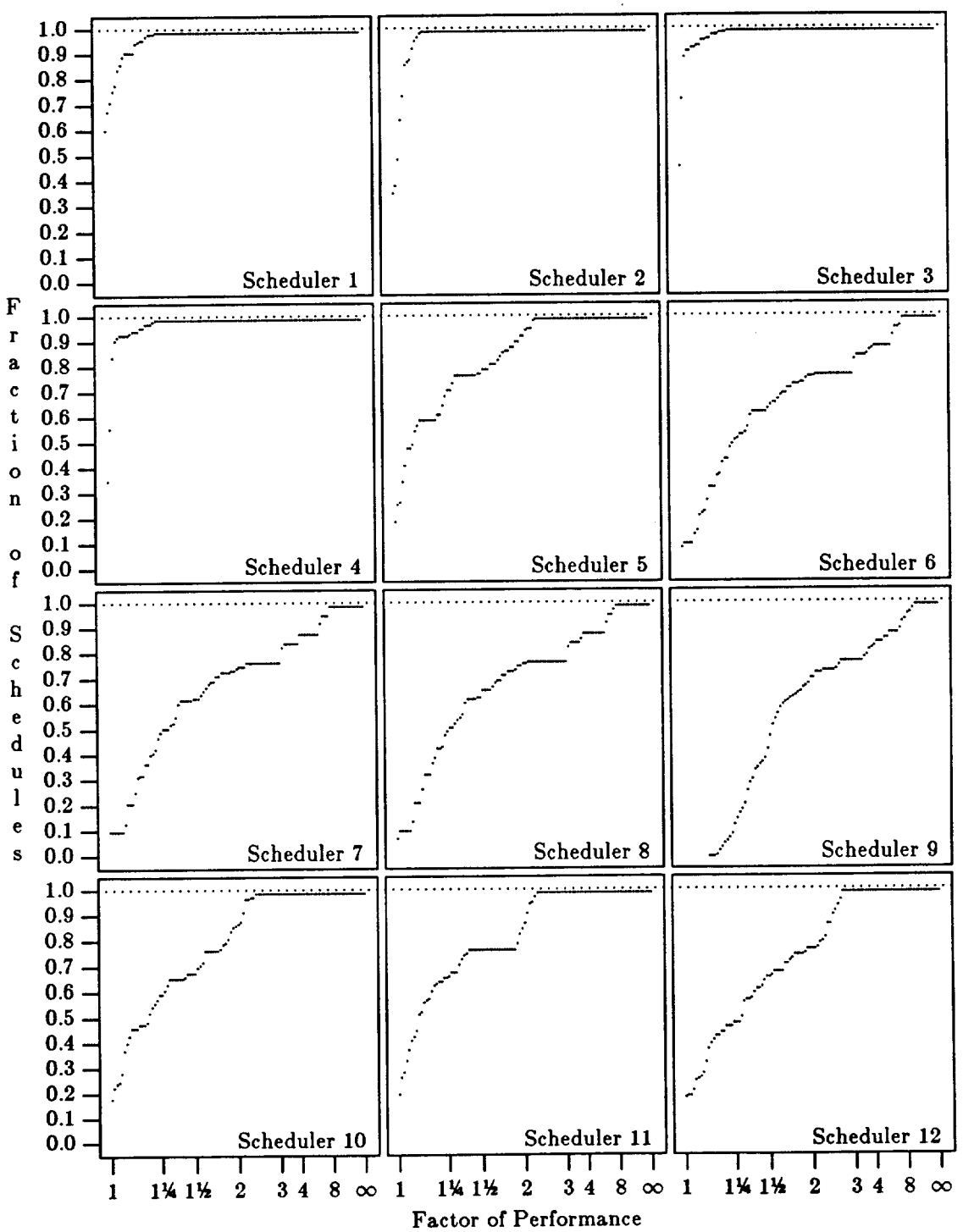
This appendix contains cumulative histograms of schedule lengths relative to the shortest available schedule. Each program/architecture pair was was used by the 12 schedulers to generate a parallel schedule. The 12 parallel schedules and a sequential schedule were compared for length, and the shortest was selected. This schedule was used as a reference for later comparisons as the shortest available schedule for that program/architecture pair. These histograms were created by dividing each parallel schedule length by the shortest available length, and histograming the result. Thus the x-axis represents the ratio of a schedule to the best schedule, and the y-axis represents the number of parallel schedules that did *at least* that well.

For a concrete example, consider the histogram for scheduler #1 in section E.1. One point along the curve occurs at (1.25, 0.88). This means that overall, 88% of the schedules generated by scheduler #1 were no longer than 1.25 times the length of the best known schedules for the corresponding program/architecture pairs.

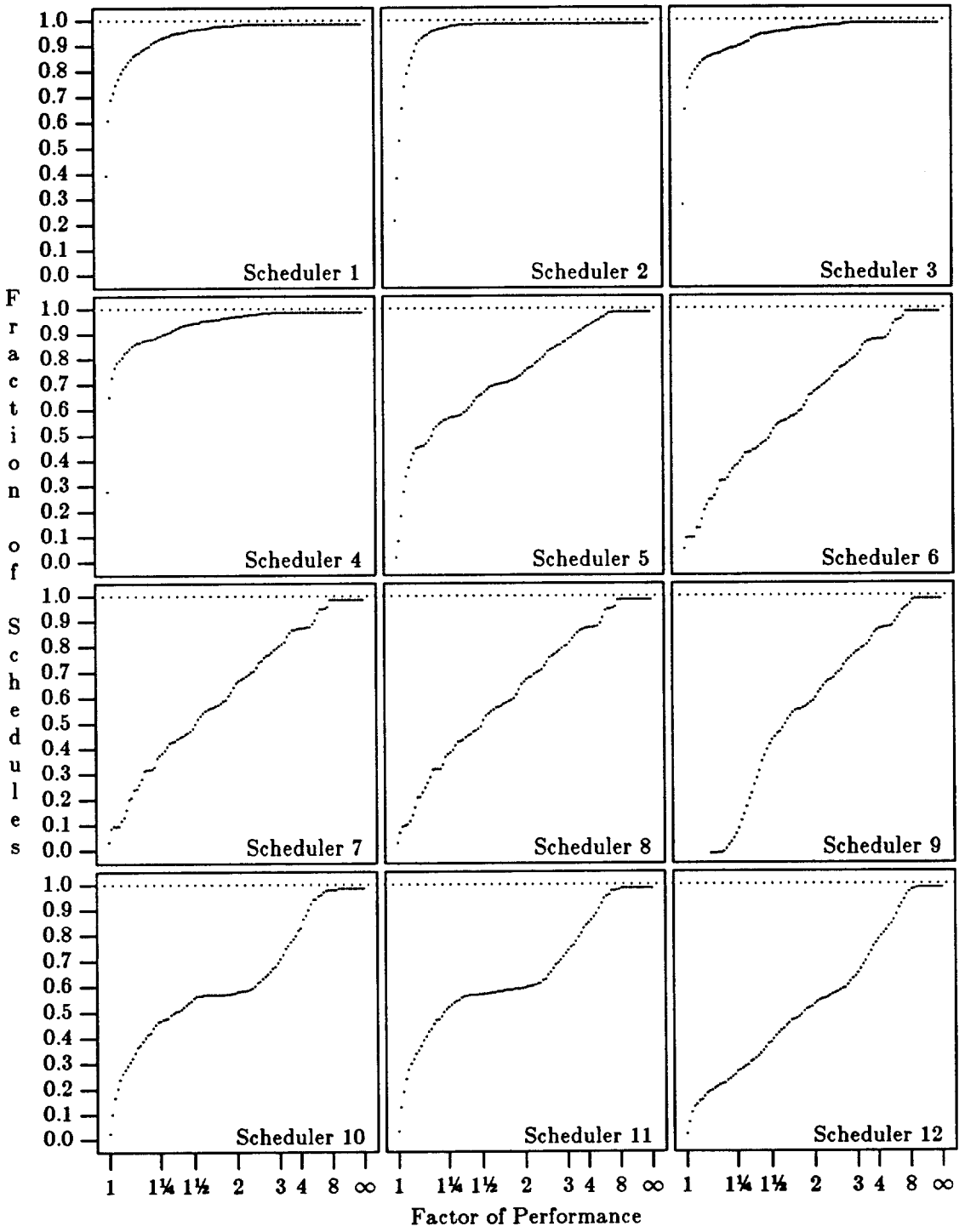
E.1. All Test Cases



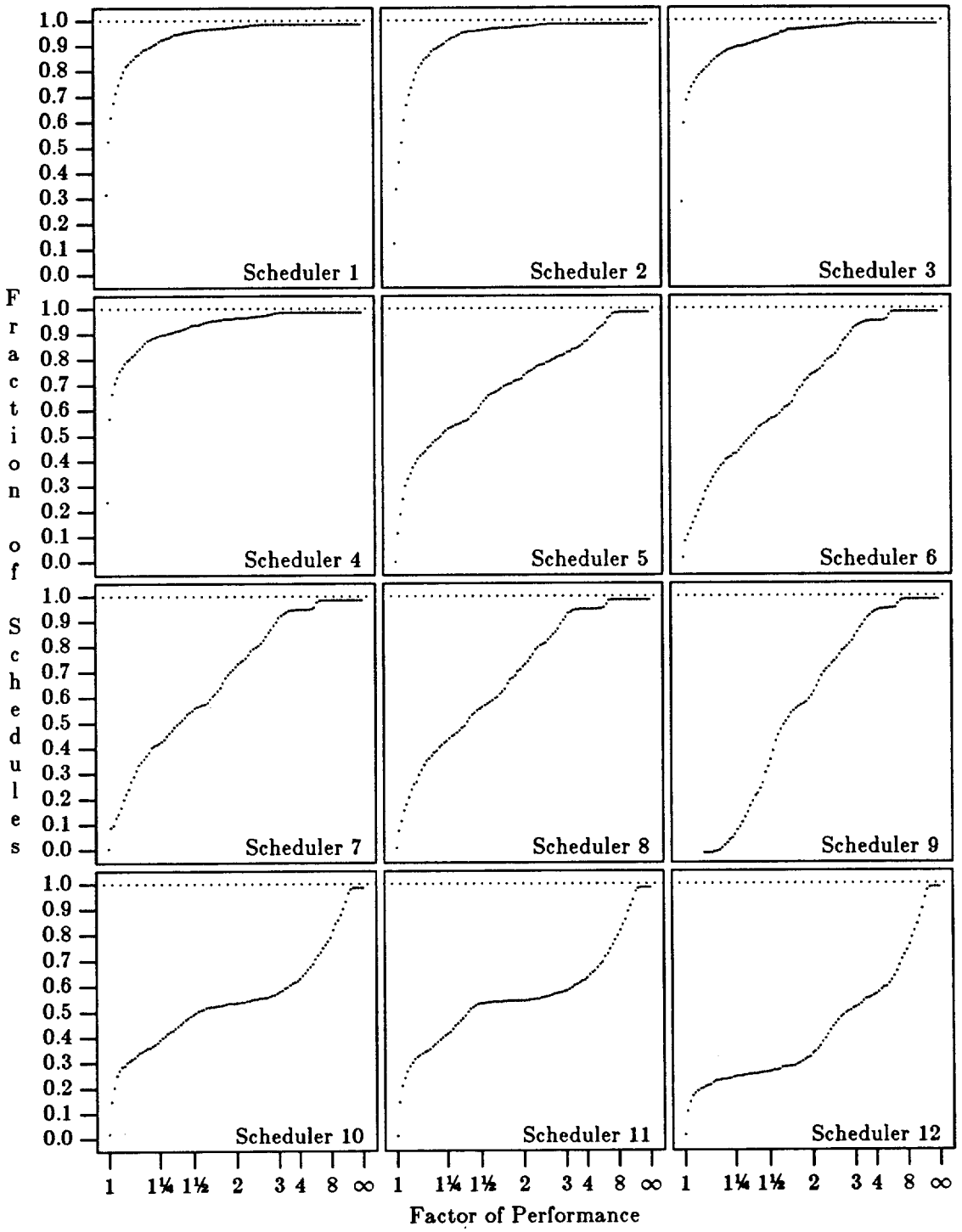
E.2. Average Parallelism



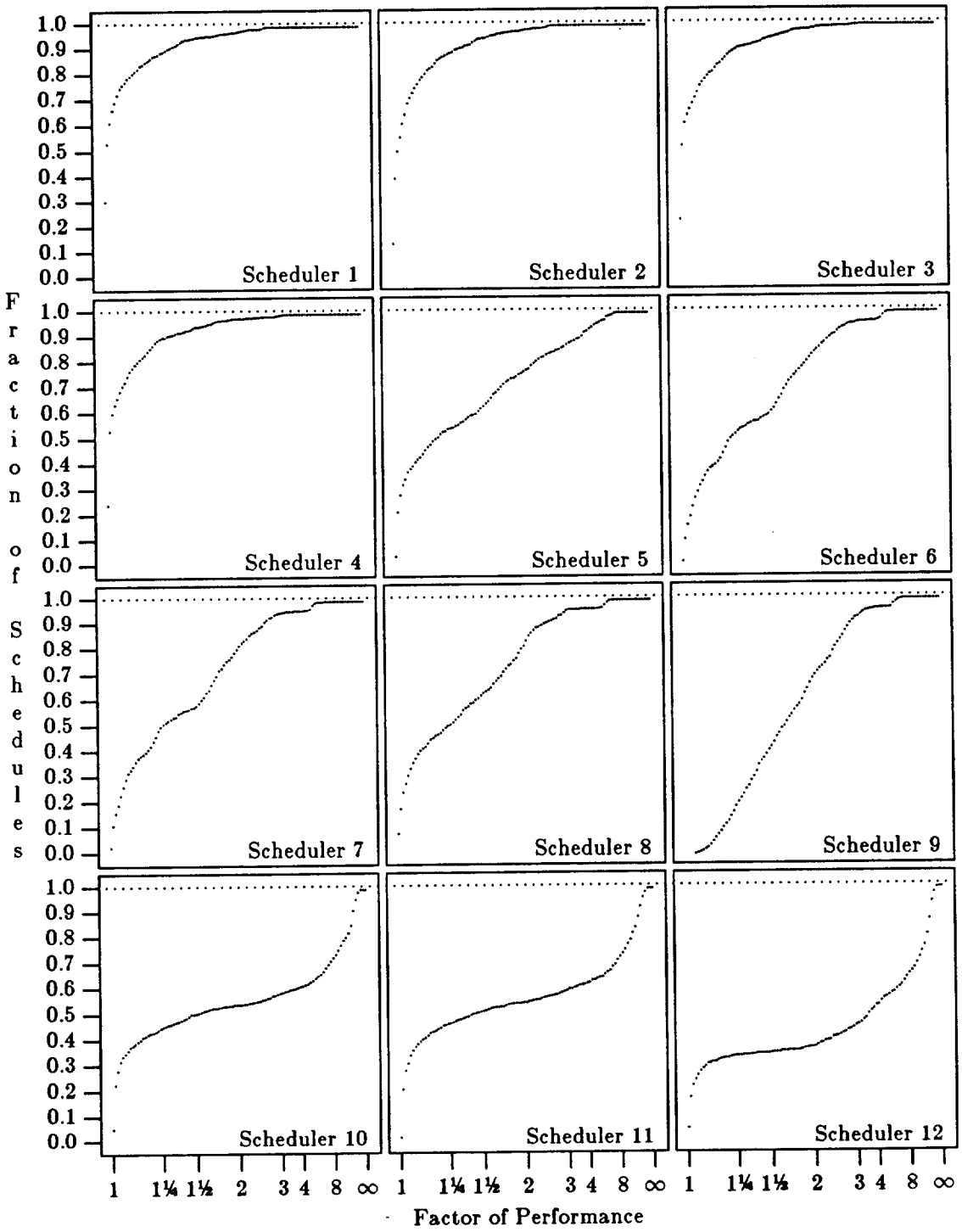
Parallelism 2



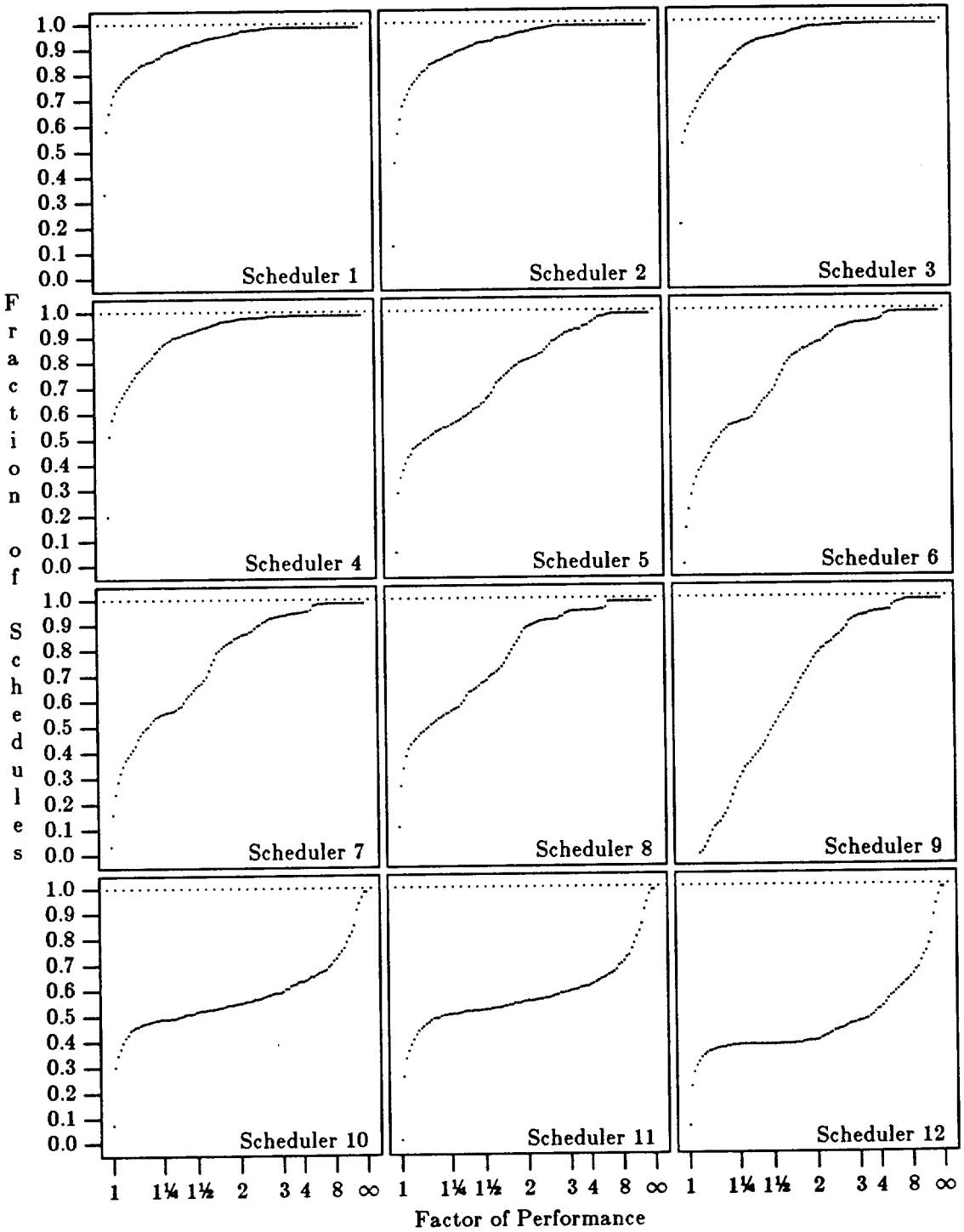
Parallelism 4



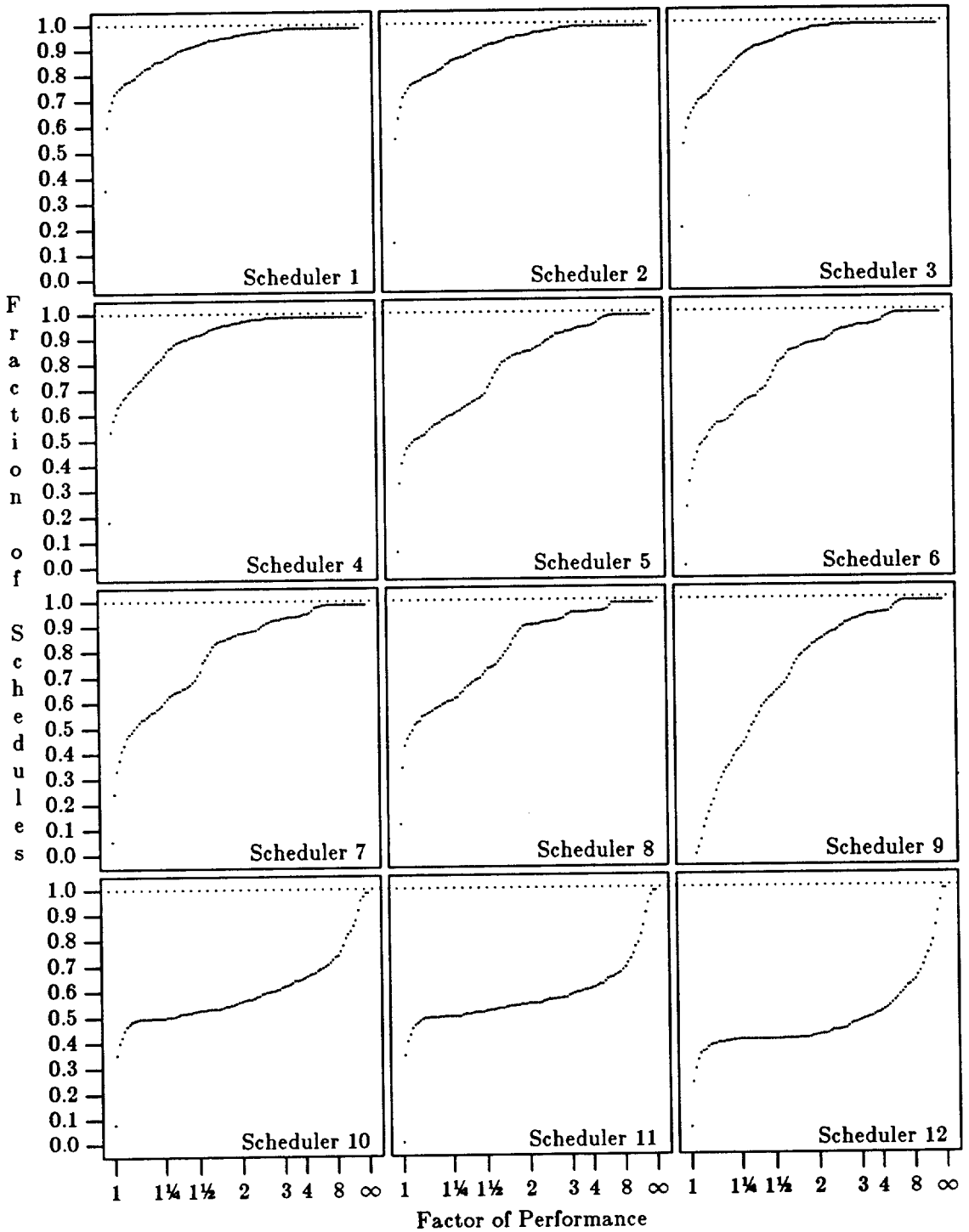
Parallelism 8



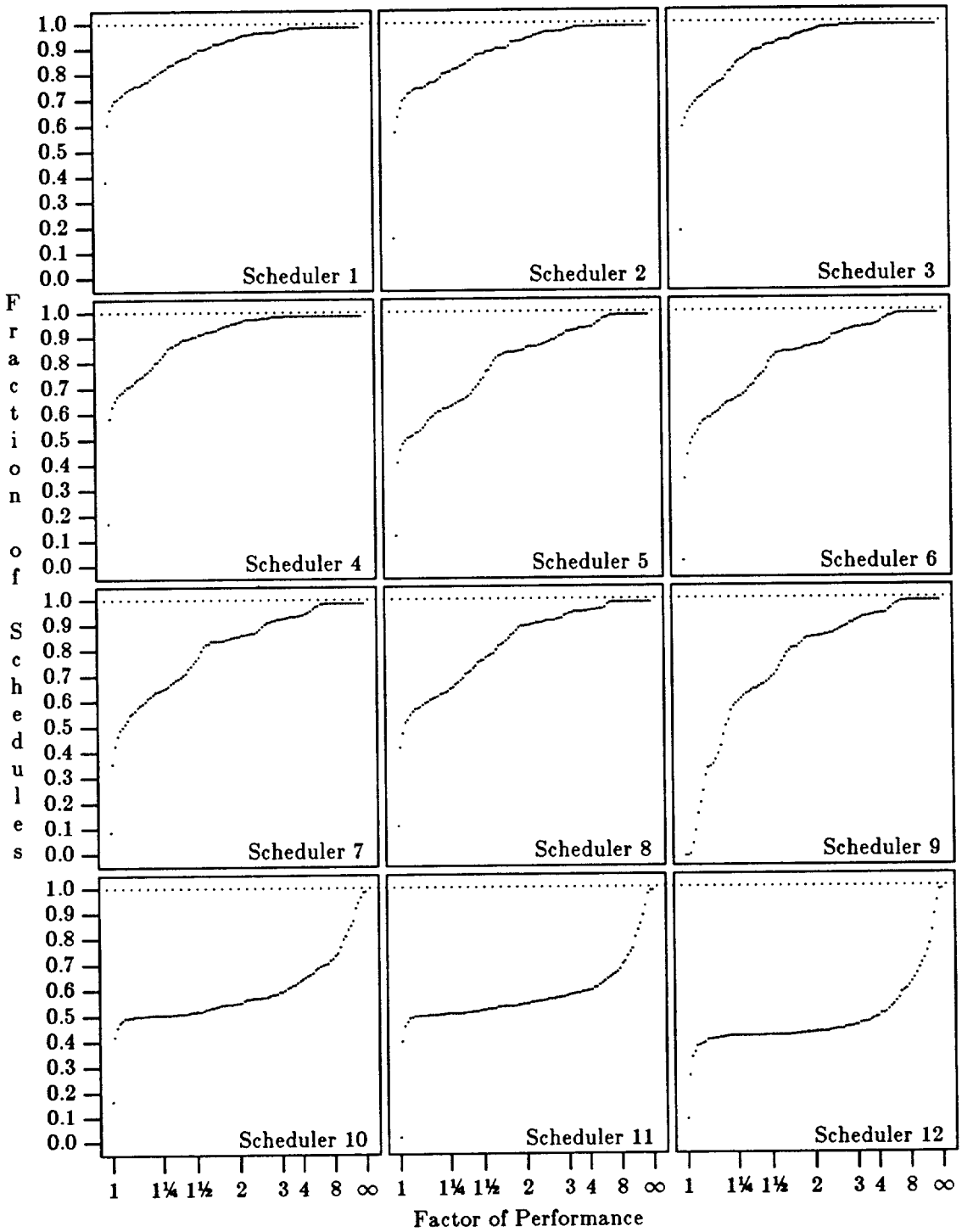
Parallelism 16



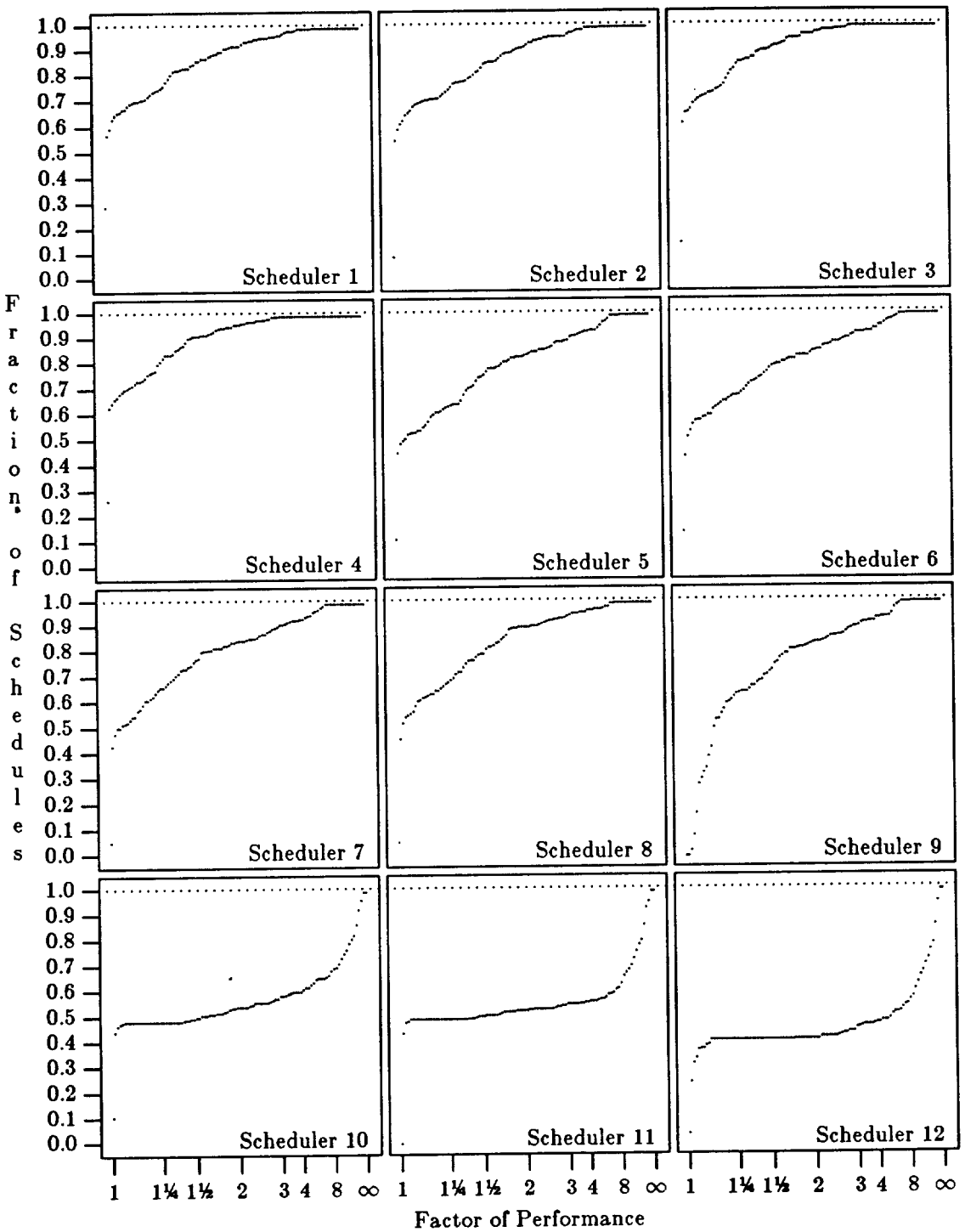
Parallelism 32



Parallelism 64

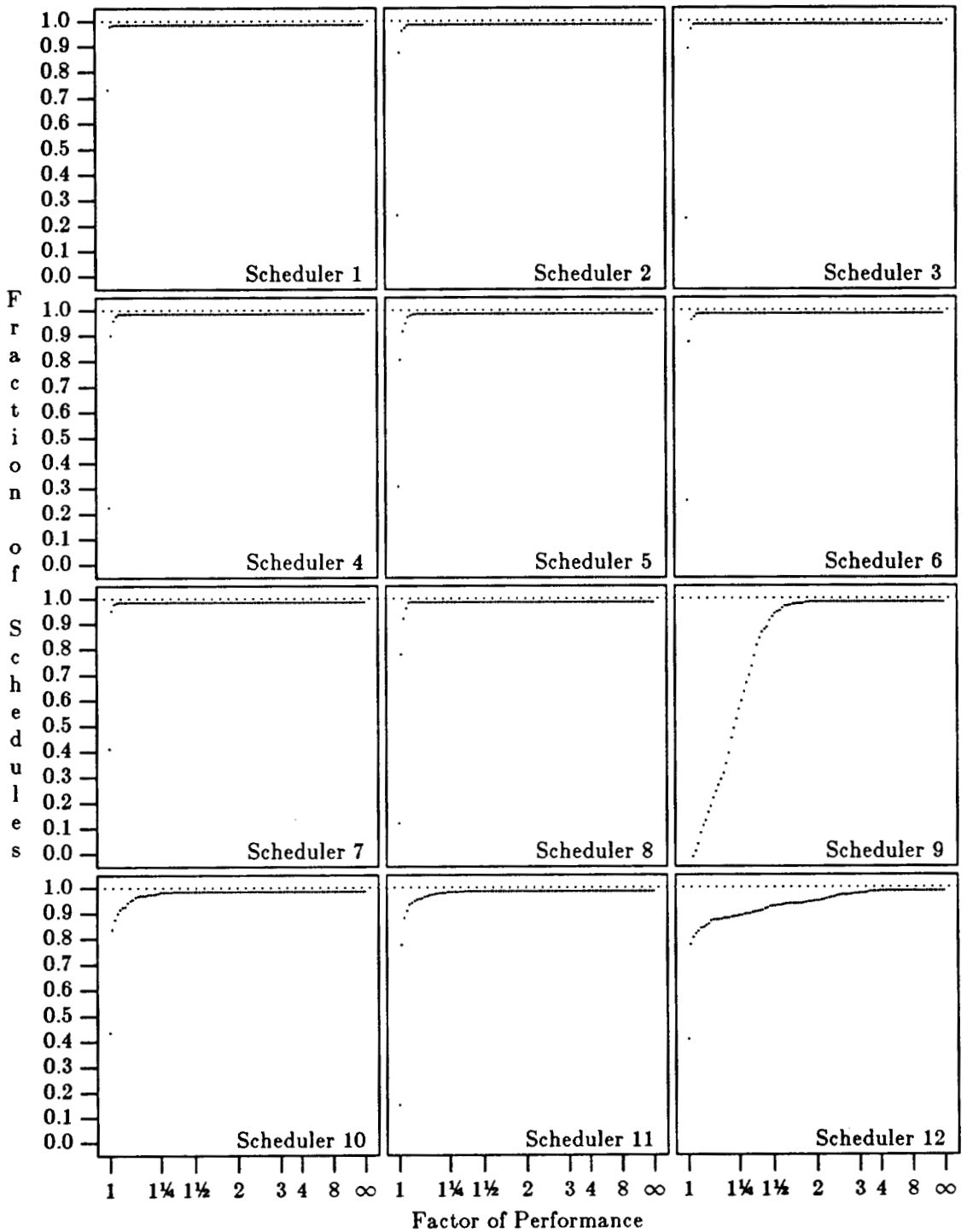


Parallelism 128

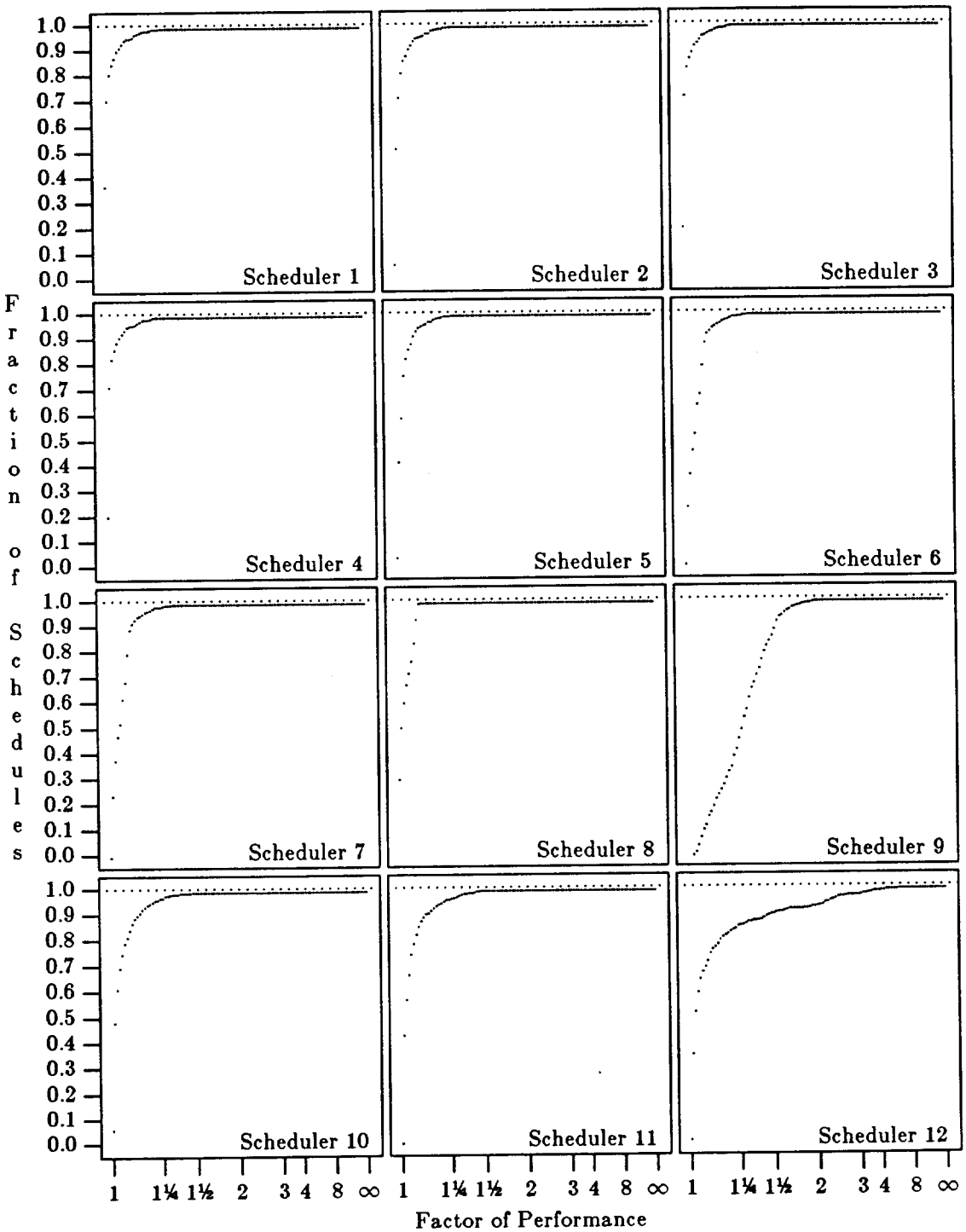


Parallelism 256

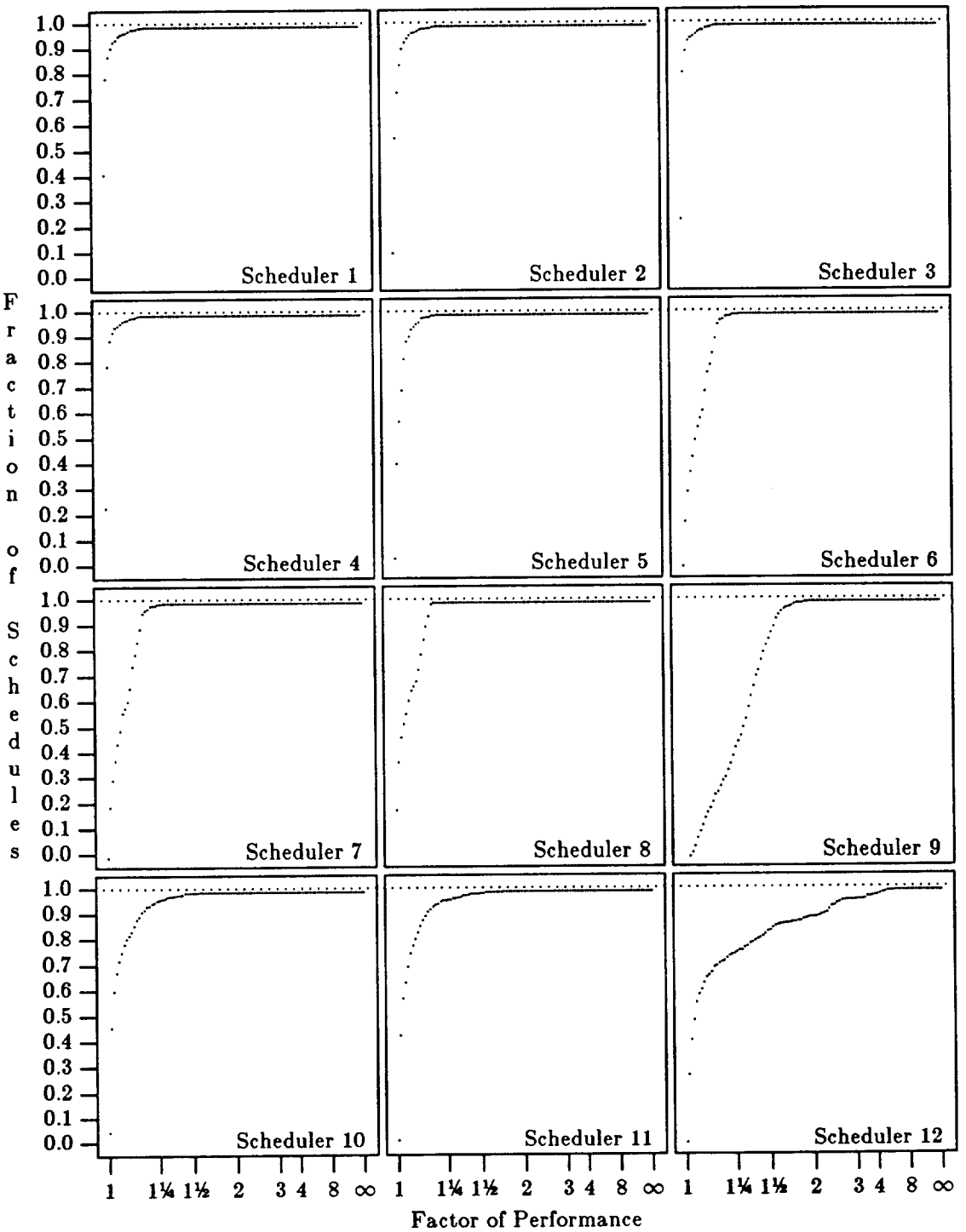
E.3. Latency

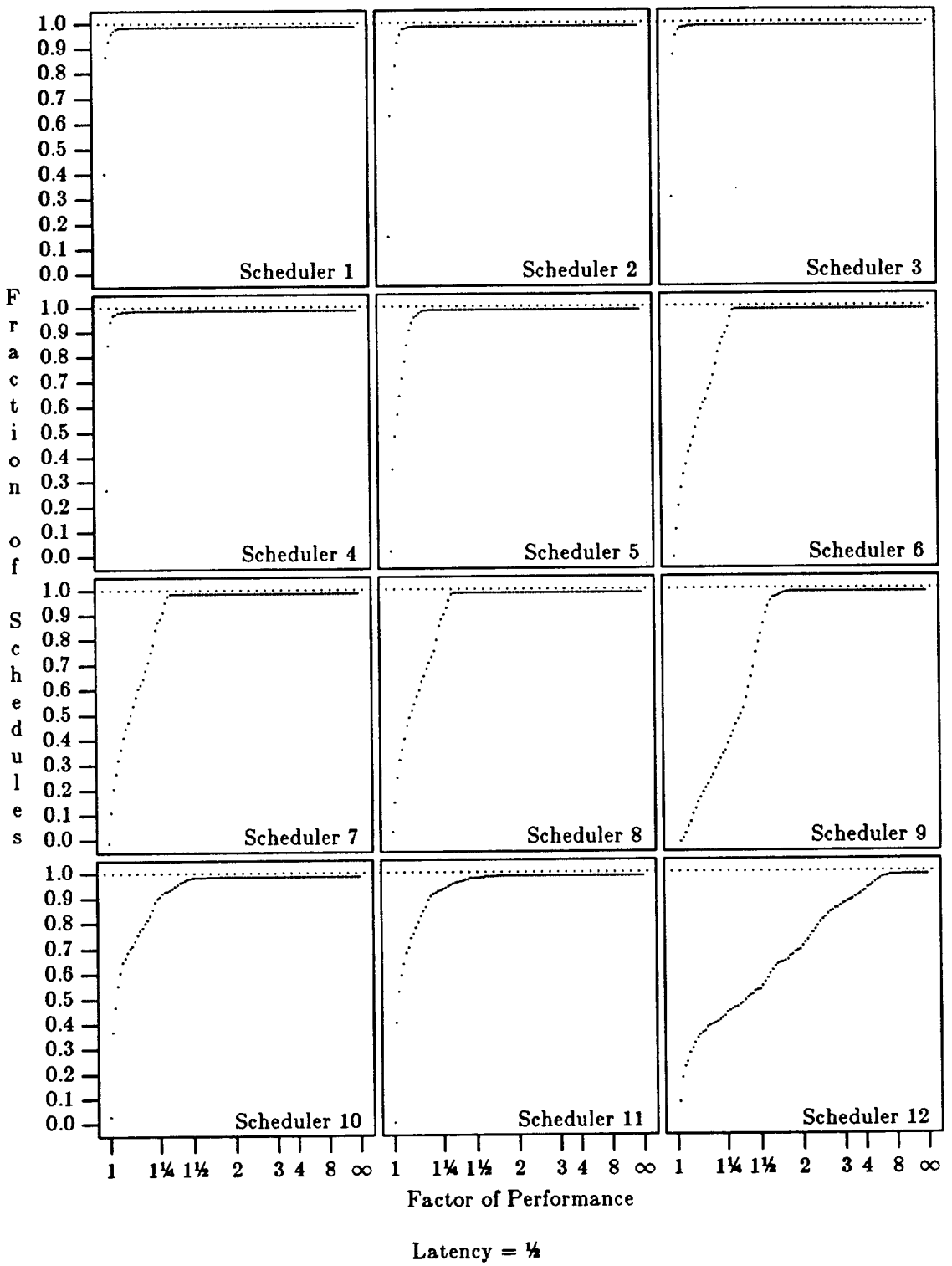


Latency = 0

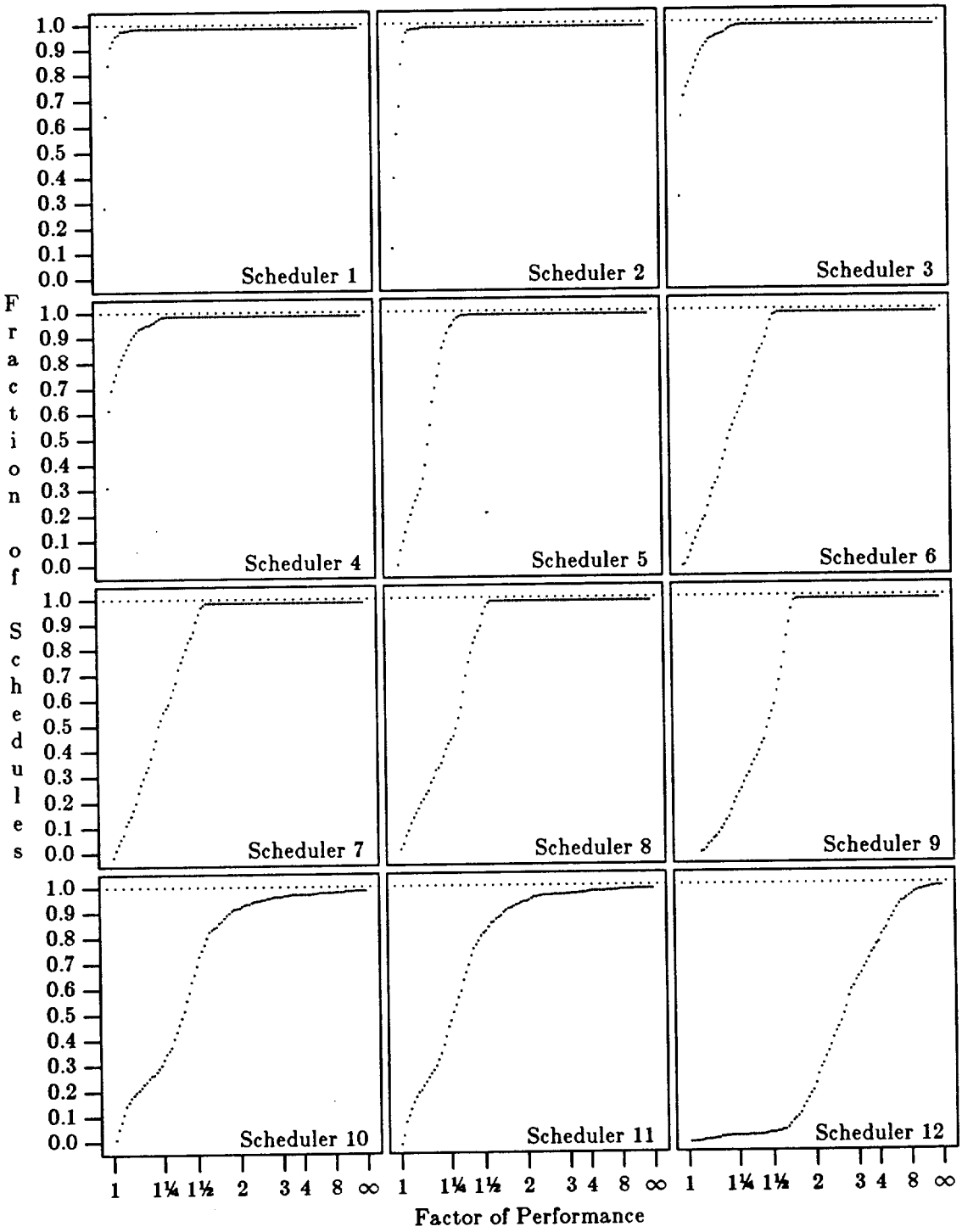

Scheduler 1
Scheduler 2
Scheduler 3
Scheduler 4
Scheduler 5
Scheduler 6
Scheduler 7
Scheduler 8
Scheduler 9
Scheduler 10
Scheduler 11
Scheduler 12
Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1 1¼ 1½ 2 3 4 8 ∞
Factor of Performance

Latency = 1/8

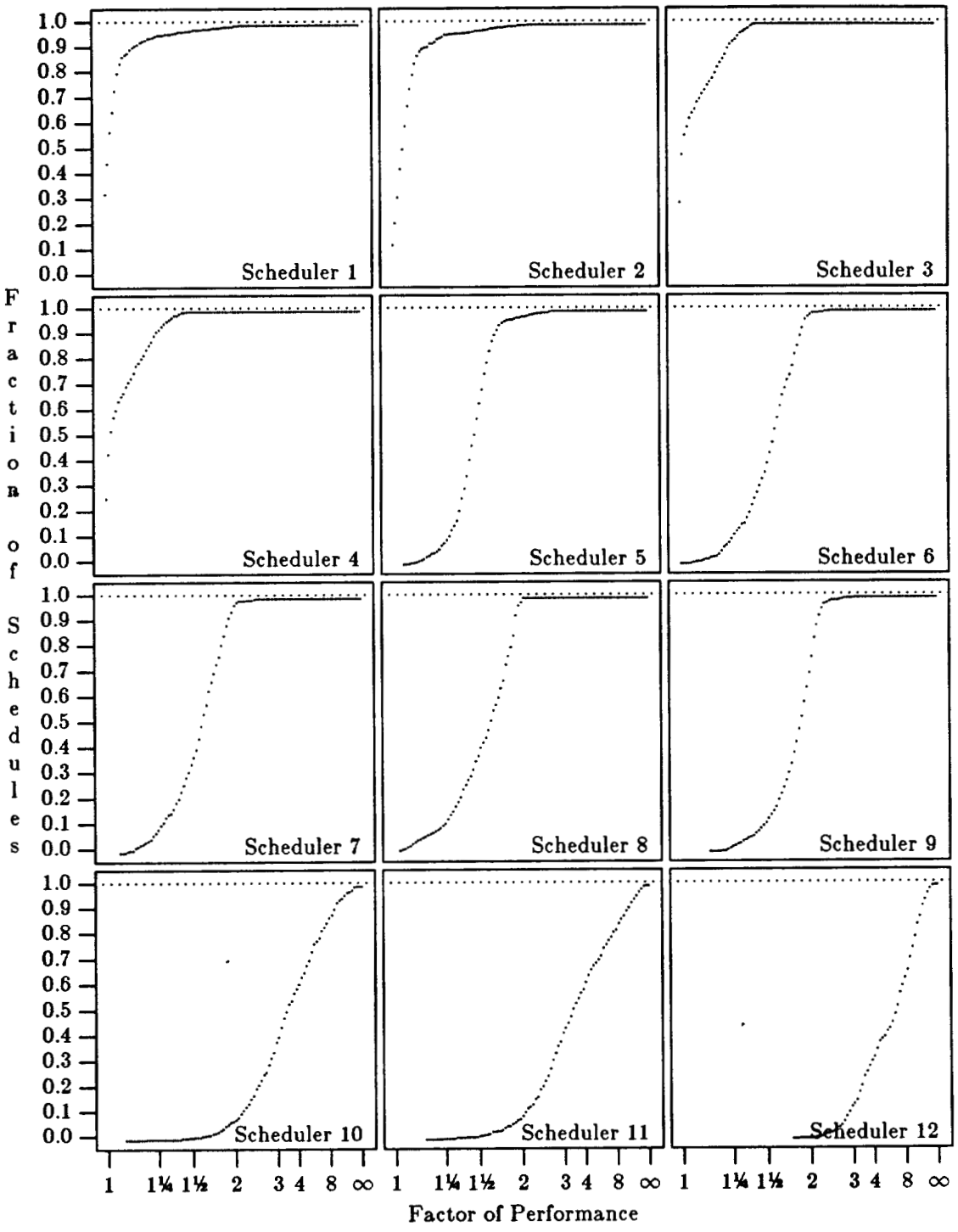

Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Scheduler 1
Scheduler 2
Scheduler 3
Scheduler 4
Scheduler 5
Scheduler 6
Scheduler 7
Scheduler 8
Scheduler 9
Scheduler 10
Scheduler 11
Scheduler 12
1 1¼ 1½ 2 3 4 8 ∞
Factor of Performance
Latency = ¼


Scheduler 1
Scheduler 2
Scheduler 3
Scheduler 4
Scheduler 5
Scheduler 6
Scheduler 7
Scheduler 8
Scheduler 9
Scheduler 10
Scheduler 11
Scheduler 12
Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1 1¼ 1½ 2 3 4 8 ∞
Factor of Performance

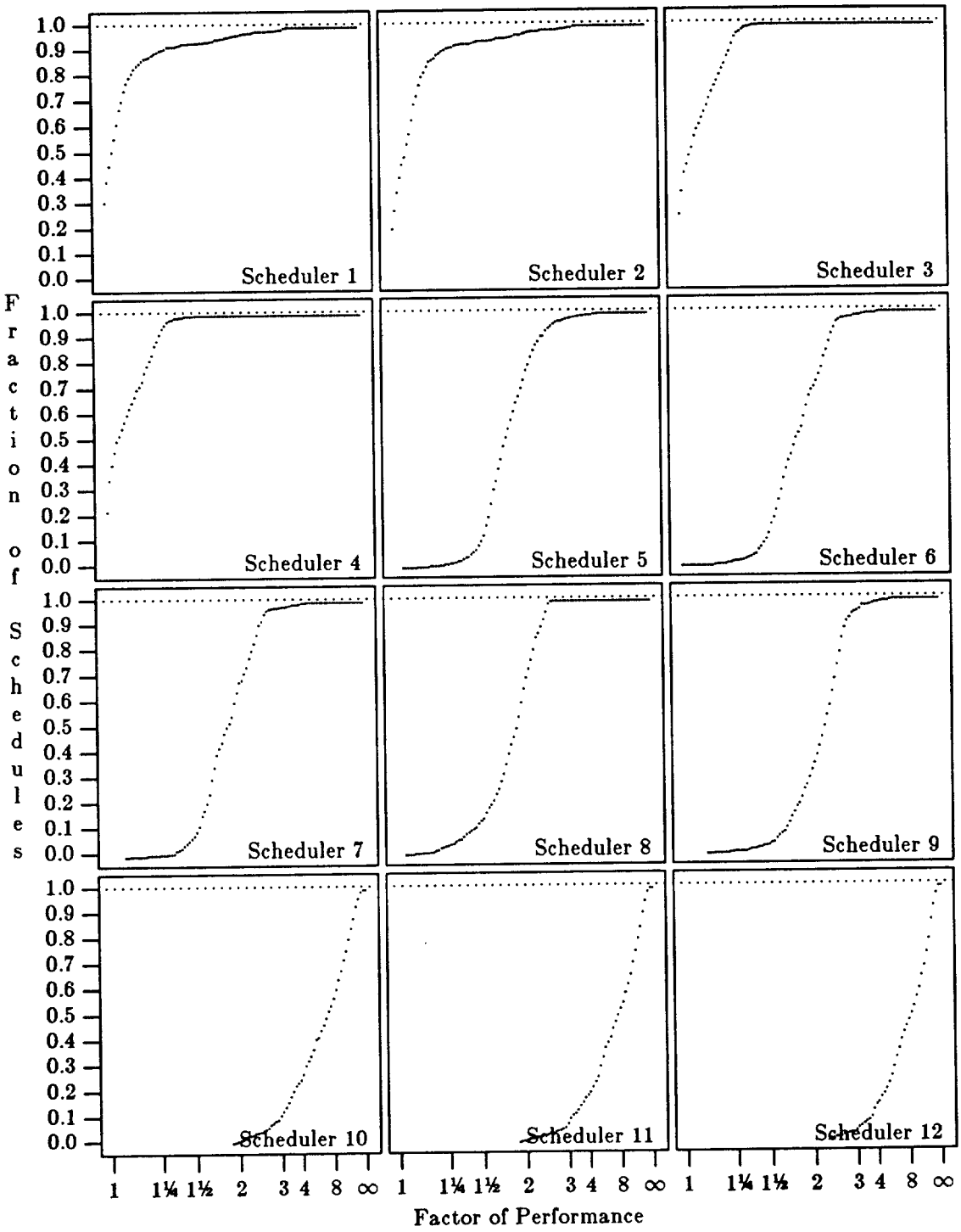
Latency = ½


Scheduler 1
Scheduler 2
Scheduler 3
Scheduler 4
Scheduler 5
Scheduler 6
Scheduler 7
Scheduler 8
Scheduler 9
Scheduler 10
Scheduler 11
Scheduler 12
Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1 1¼ 1½ 2 3 4 8 ∞
Factor of Performance

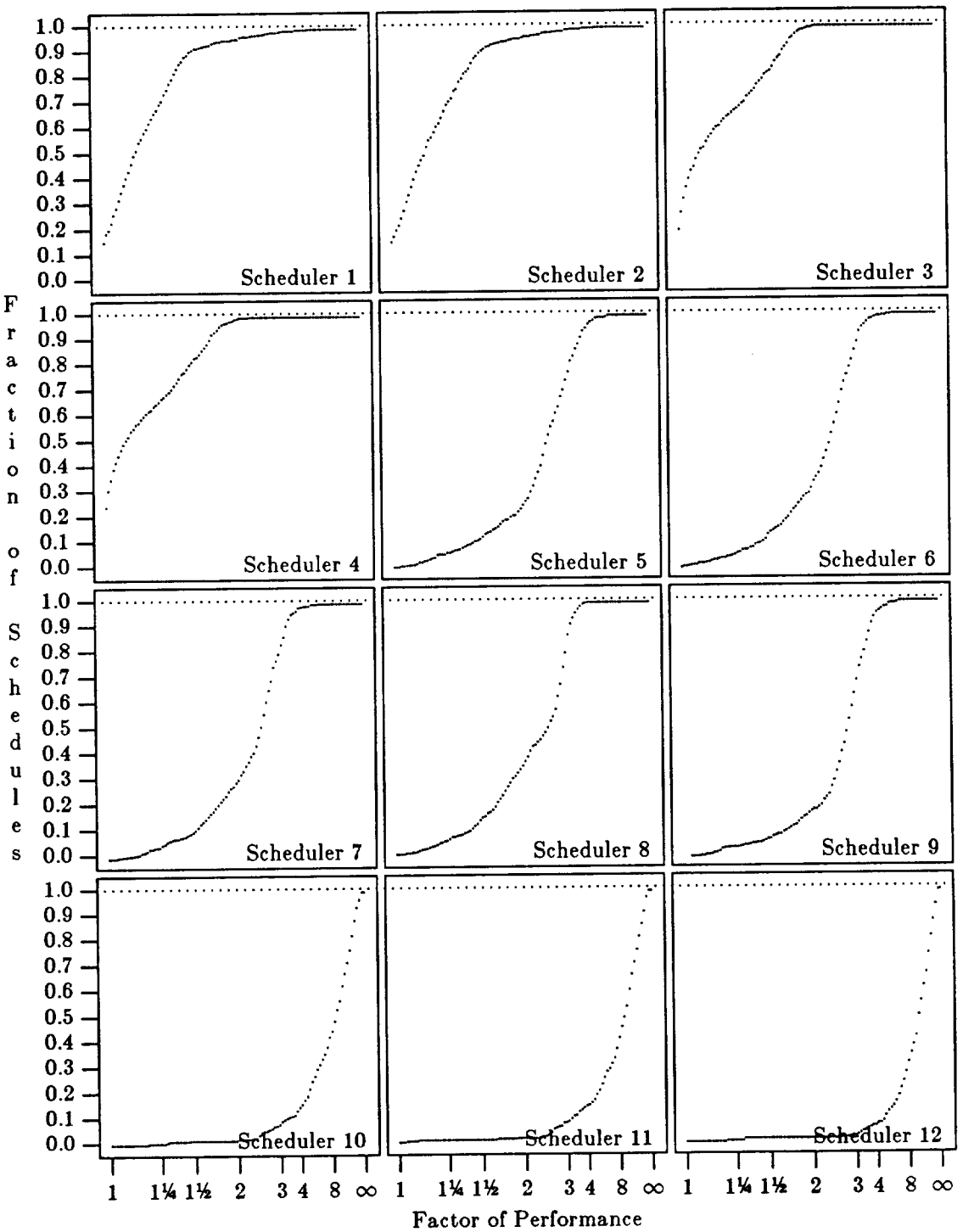
Latency = 1


Scheduler 1
Scheduler 2
Scheduler 3
Scheduler 4
Scheduler 5
Scheduler 6
Scheduler 7
Scheduler 8
Scheduler 9
Scheduler 10
Scheduler 11
Scheduler 12
Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1 1¼ 1½ 2 3 4 8 ∞
Factor of Performance

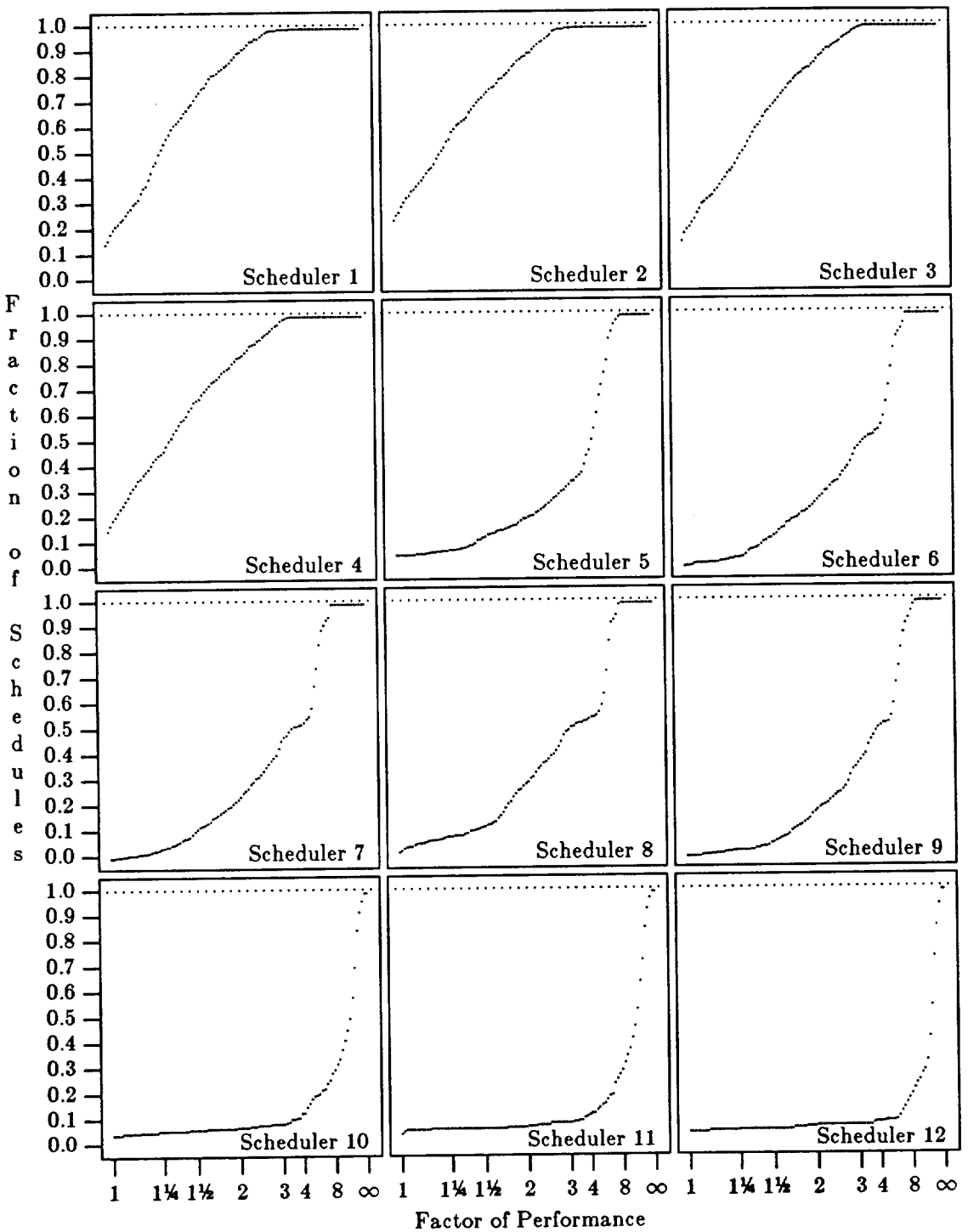
Latency = 2


Scheduler 1
Scheduler 2
Scheduler 3
Scheduler 4
Scheduler 5
Scheduler 6
Scheduler 7
Scheduler 8
Scheduler 9
Scheduler 10
Scheduler 11
Scheduler 12
Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1 1¼ 1½ 2 3 4 8 ∞
Factor of Performance

Latency = 4

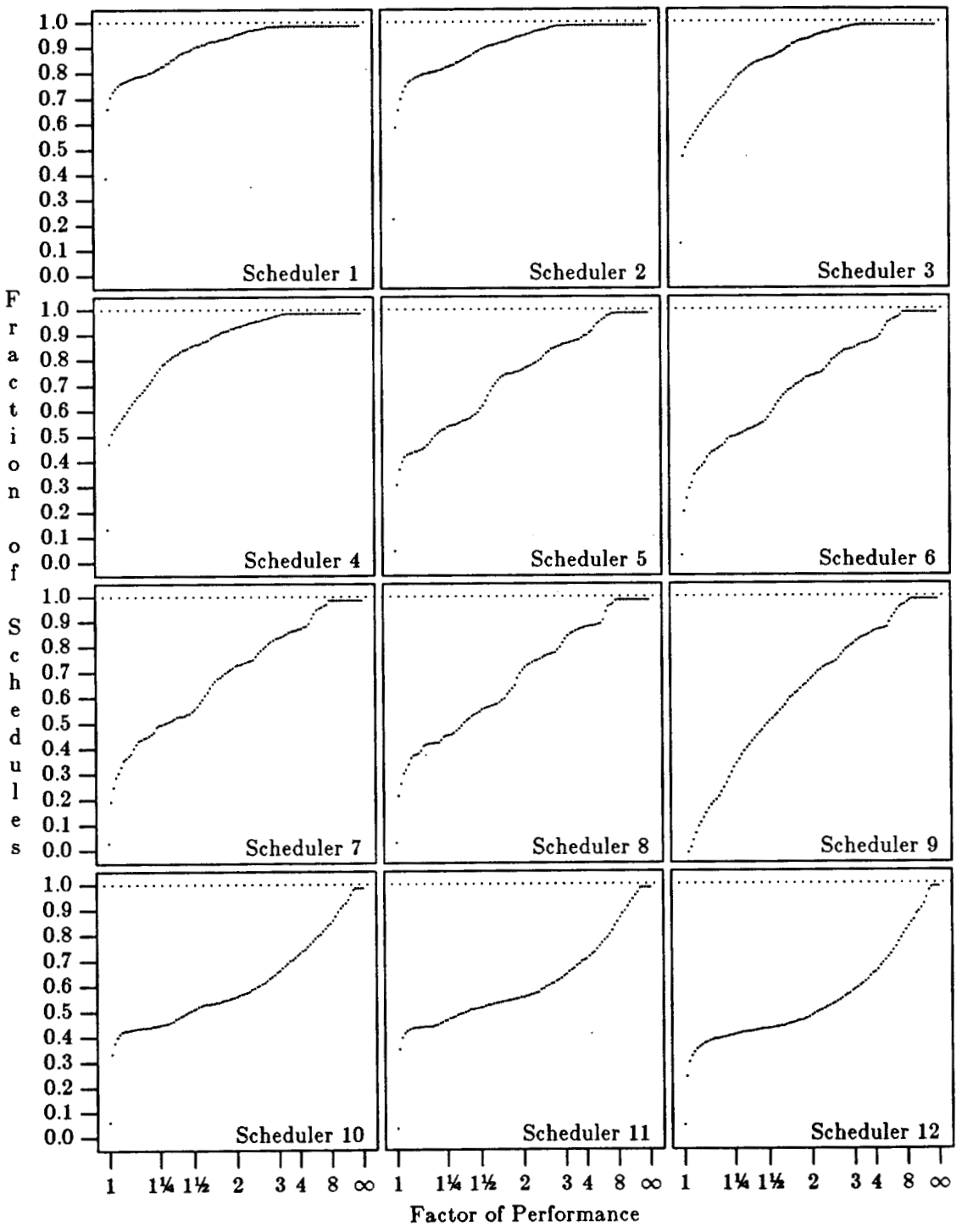

Scheduler 1
Scheduler 2
Scheduler 3
Scheduler 4
Scheduler 5
Scheduler 6
Scheduler 7
Scheduler 8
Scheduler 9
Scheduler 10
Scheduler 11
Scheduler 12
Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1 1¼ 1½ 2 3 4 8 ∞
Factor of Performance

Latency = 8

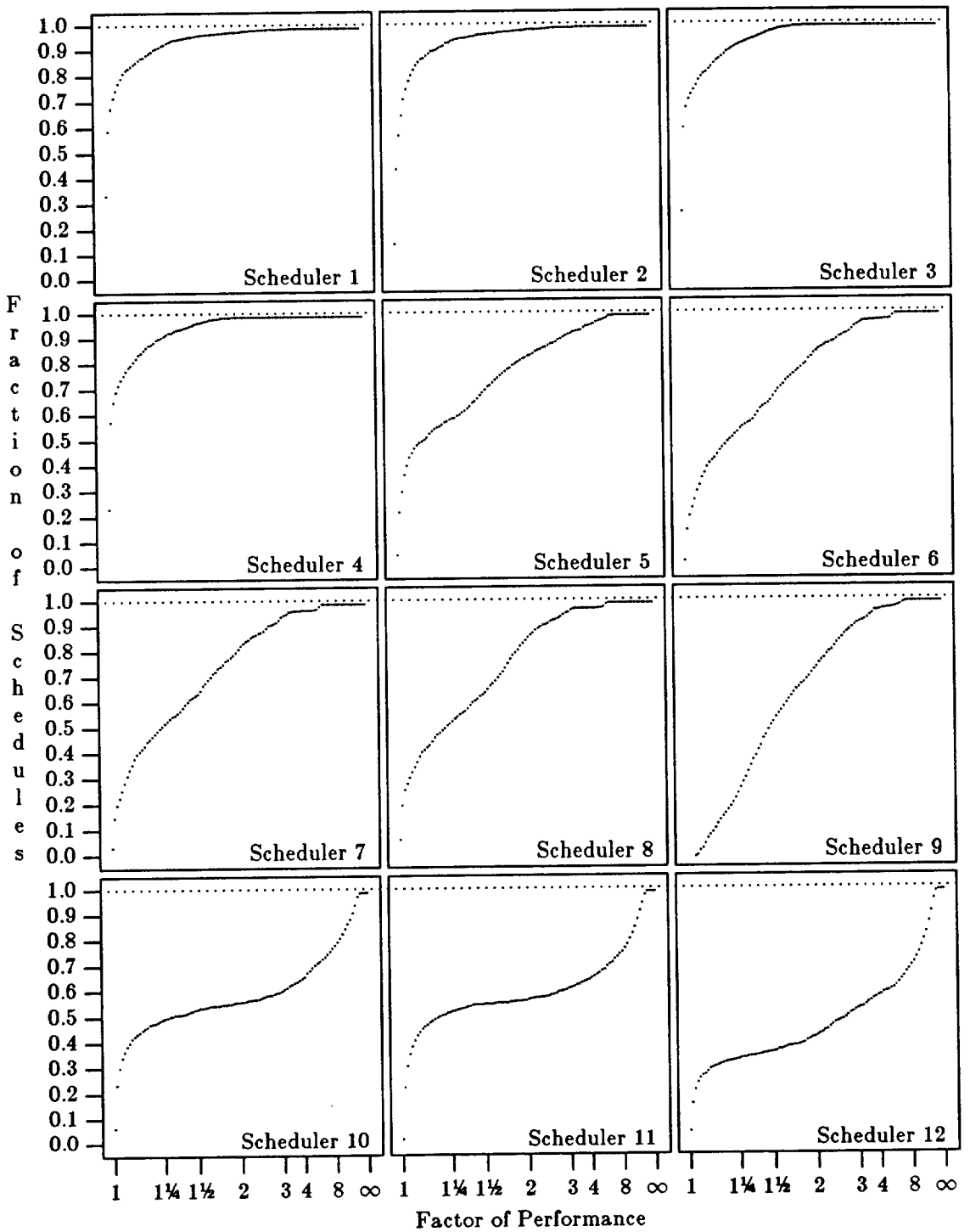

Scheduler 1
Scheduler 2
Scheduler 3
Scheduler 4
Scheduler 5
Scheduler 6
Scheduler 7
Scheduler 8
Scheduler 9
Scheduler 10
Scheduler 11
Scheduler 12
Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1 1¼ 1½ 2 3 4 8 ∞
Factor of Performance

Latency = 16

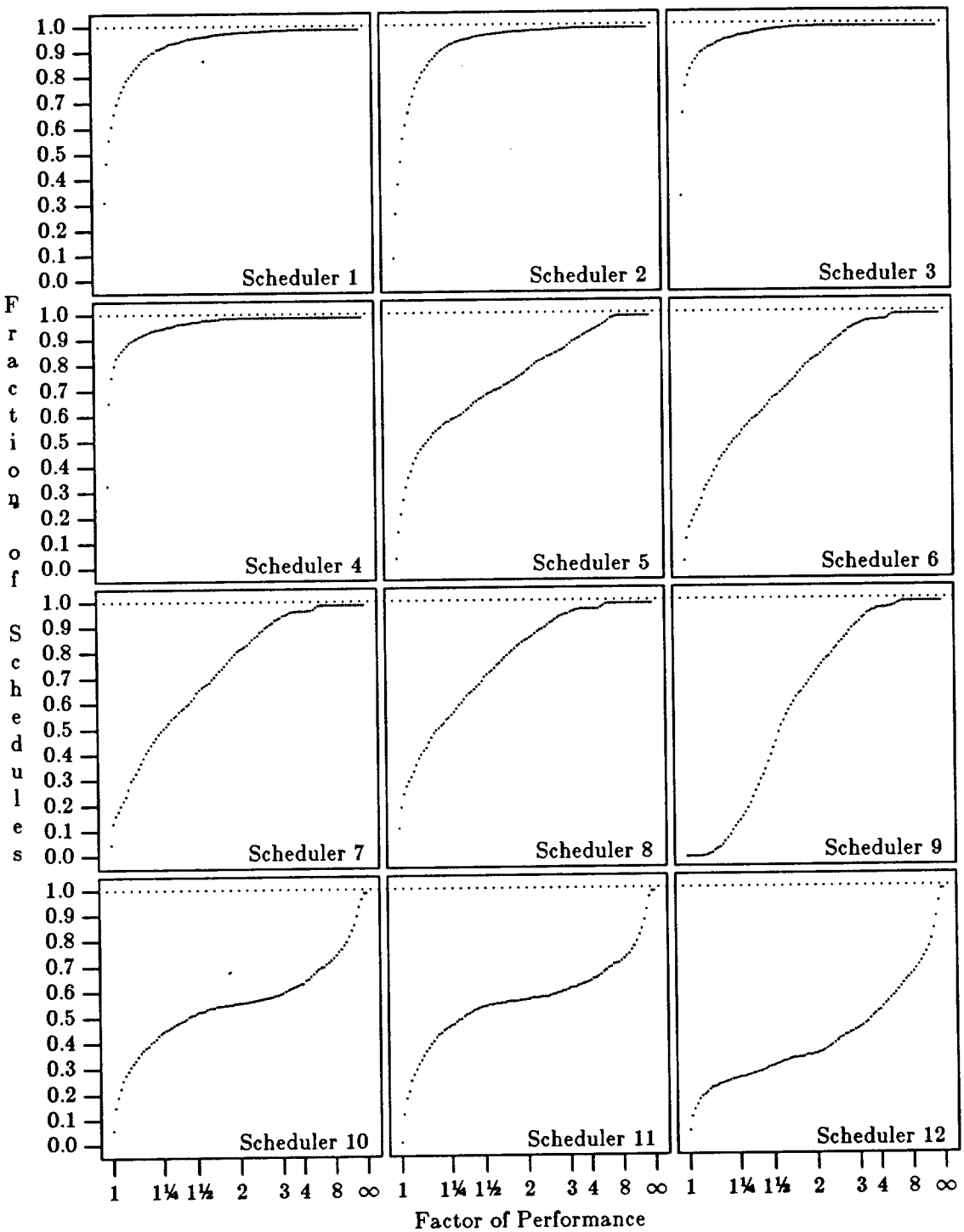
E.4. Processor Count



Processor Count 4

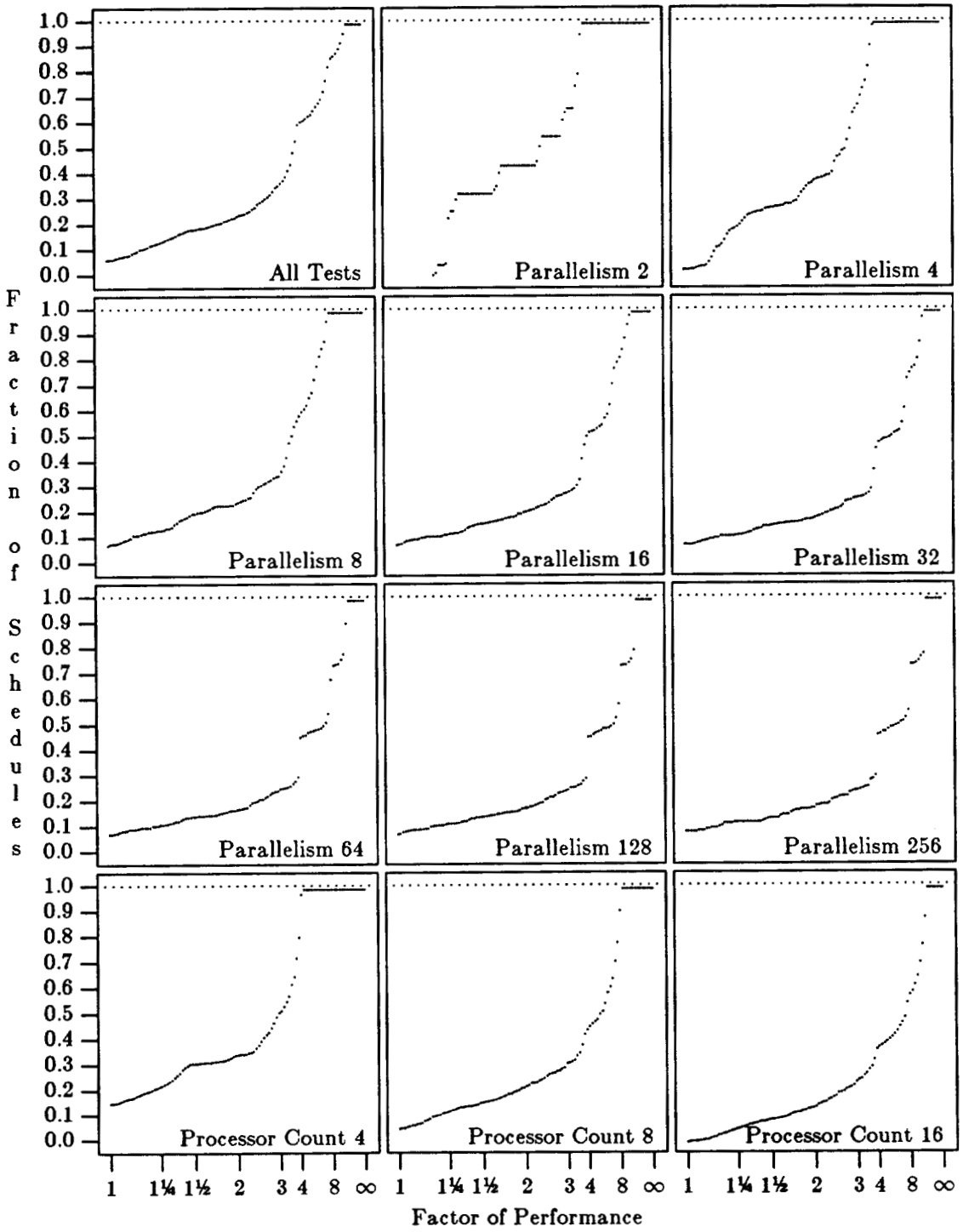


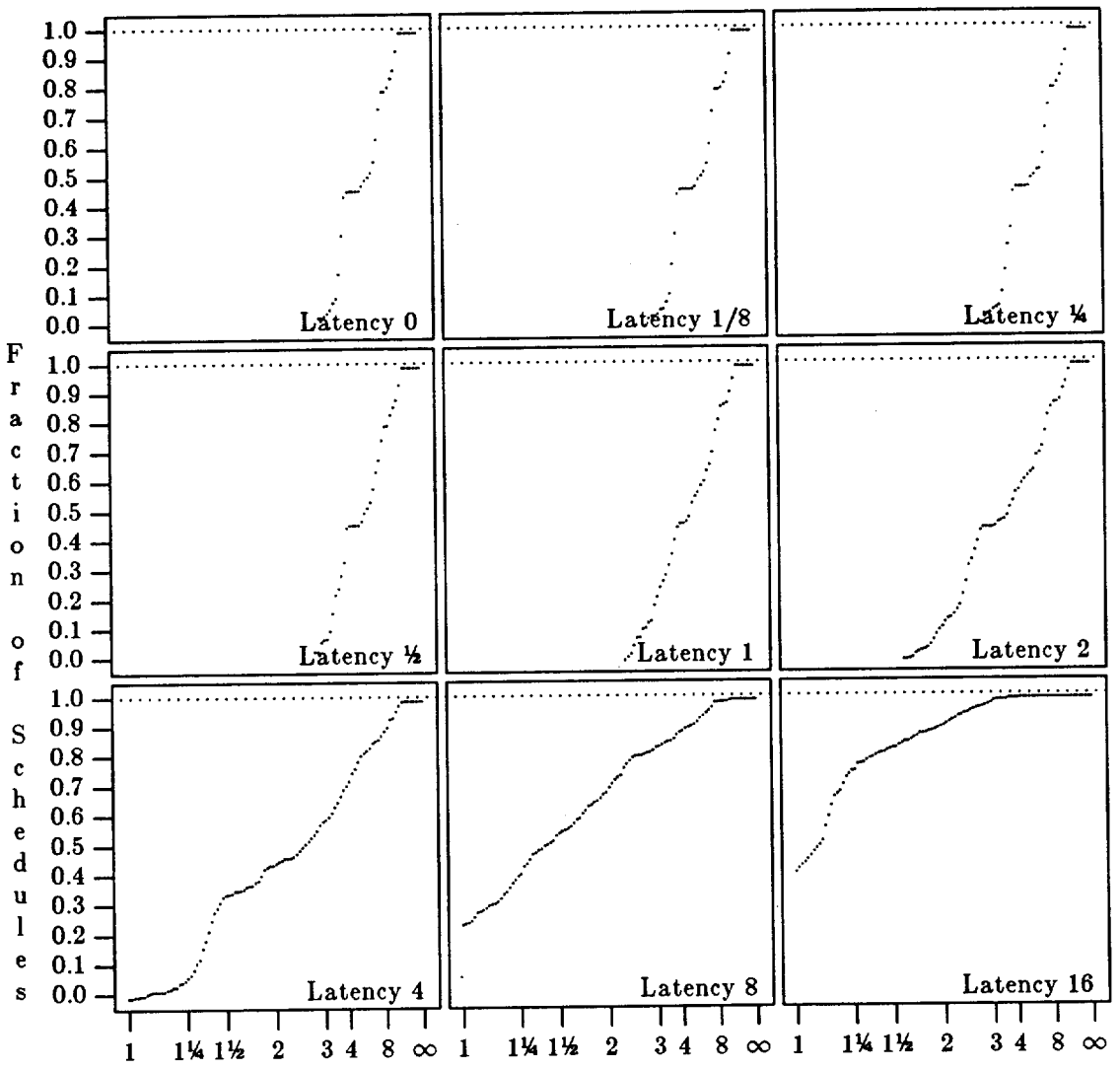
Processor Count 8



Processor Count 16

E.5. Sequential Scheduler




Fraction of Schedules
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Latency 0
Latency 1/8
Latency ¼
Latency ½
Latency 1
Latency 2
Latency 4
Latency 8
Latency 16
1 1¼ 1½ 2 3 4 8 ∞

References

[Ack82] W. B. Ackerman, "Data Flow Languages," *IEEE Computer*, vol. 15, 2 (February 1982), pp. 15-25.

[ACD74] T. L. Adam, K. M. Chandy and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM*, vol. 17, 12 (December 1974), pp. 685-690.

[AkK84] S. B. Akers and B. Krishnamutrhy, "Group Graphs as Interconnection Networks," *14th International Conference on Fault Tolerant Computing*, Kissimmee, Florida, June 1984, pp. 422-427.

[AkK87] S. B. Akers and B. Krishnamurthy, "On Group Graphs and Their Fault Tolerance," *IEEE Transactions on Computers*, vol. C-36, 7 (July 1987), pp. 885-888.

[AlC72] F. E. Allen and J. Cocke, "A Catalogue of Optimizing Transformations," in *Design and Optimization of Compilers*, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Bab87] R. G. Babb II, ed., *Programming Parallel Processors*, Addison-Wesley, 1987.

[BSV83] A. F. Bashir, V. Susarla and K. Vairavan, "A Statistical Study of the Performance of a Task Scheduling Algorithm," *IEEE Transactions on Computers*, vol. C-32, 8 (August 1983), pp. 774-777.

[BeS87] F. Berman and L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures," *Journal of Parallel and Distributed Computing*, vol. 4, 5 (October 1987), pp. 439-458.

[BWD84] J. Blazewicz, J. Weglarz and M. Drabowski, "Scheduling independent 2-processor tasks to minimize schedule length," *Information Processing Letters*, vol. 18, 5 (June 1984), pp. 267-274.

[Bok81a] S. H. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. C-30, 3 (March 1981), pp. 207-214.

[Bok81b] S. H. Bokhari, "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System," *IEEE Transactions on Software Engineering*, vol. SE-7, 6 (November 1981), pp. 583-589.

[CHA88] C. C. Carroll, A. Homaifar and K. G. Ananthram, "An Intelligent Allocation Algorithm For Parallel Processing," BER Report 416-17, The University of Alabama, January 1988.

[Cas87] T. L. Casavant, "Analysis of Three Dynamic Distributed Load-Balancing Strategies with Varying Global Information Requirements," *The 7th International Conference on Distributed Computing Systems*, September 1987, pp. 185-182.

[CDJ84] F. B. Chambers, D. A. Duce and G. P. Jones, eds., *Distributed Computing*, Academic Press, New York, 1984.

[CoG72] E. G. Coffman and R. L. Graham, "Optimal Scheduling for Two-Processor Systems," *Acta Informatica*, vol. 1(1972), pp. 200-213.

[Cof76] E. G. Coffman, Jr., ed., *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, 1976.

[DaS87] W. J. Dally and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, vol. C-36, 5 (May 1987), pp. 547-553.

[DaK82] A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *IEEE Computer*, vol. 15, 2 (February 1982), pp. 26-41.

[Den80] J. B. Dennis, "Data Flow Supercomputers," *Computer*, vol. 13, 11 (November 1980), pp. 48-56.

[Dot84] K. W. Doty, "New Designs for Dense Processor Interconnection Networks," *IEEE Transactions on Computers*, vol. C-33, 5 (May 1984), pp. 447-450.

[ELZ86] D. L. Eager, E. D. Lazowska and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, 5 (May 1986), pp. 662-675.

[Fen81] T. Feng, "A Survey of Interconnection Networks," *IEEE Computer*, vol. 14, 12 (December 1981), pp. 12-27.

[FeB73] E. B. Fernandez and B. Bussell, "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules," *IEEE Transactions on Computers*, vol. C-22, 8 (August 1973), pp. 745-751.

[GaJ77] M. R. Garey and D. S. Johnson, "Two-Processor Scheduling With Start-Times and Deadlines," *SIAM Journal on Computing*, vol. 6, 3 (September 1977), pp. 416-426.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability — A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[Gon77] M. J. Gonzalez Jr., "Deterministic Processor Scheduling," *ACM Computing Surveys*, vol. 9, 3 (September 1977), pp. 173-204.

[Gra69] R. L. Graham, "Bounds On Multiprocessing Timing Anomalies," *SIAM Journal of Applied Mathematics*, vol. 17, 2 (March 1969), pp. 416-428.

[GKS87] M. Granski, I. Koren and G. M. Silberman, "The Effect of Operation Scheduling on the Performance of a Data Flow Computer," *IEEE Transactions on Computers*, vol. C-36, 9 (September 1987), pp. 1019-1029.

[Gur84] J. R. Gurd, "Fundamentals of Dataflow," in *Distributed Computing*, F. B. Chambers, D. A. Duce and G. P. Jones (eds.), Academic Press, New York, 1984.

[Ham80] D. Hammerstrom, "Dynamic, Decentralized Load Leveling," *Euromicro 80*, London, England, October 1980.

[HRW85] J. G. Harp, J. B. G. Roberts and J. S. Ward, "Signal Processing With Transputer Arrays (TRAPS)," *Computer Physics Communications*, vol. 37(1985), pp. 77-86.

[HiL74] F. S. Hillier and G. J. Lieberman, *Operations Research*, Holden-Day, San Francicso, 1974.

[HoT77] R. V. Hogg and E. A. Tanis, *Probability and Statistical Inference*, Macmillan, 1977.

[Hu61] H. C. Hu, "Parallel Sequencing And Assembly Line Problems," *Operations Research*, vol. 9, 6 (November 1961), pp. 841-848.

[HwB84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

[Jor87] H. F. Jordan, "Interpreting Parallel Processor Performance Measurements," *SIAM Journal on Scientific and Statistical Computing*, vol. 8, 2 (March 1987), pp. s220-s226.

[KaN84] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Transactions on Computers*, vol. C-33, 11 (November 1984), pp. 1023-1029.

[Kau74] M. T. Kaufman, "An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem," *IEEE Transactions on Computers*, vol. C-23, 11 (November 1974), pp. 1169-1174.

[Koh75] W. H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Transactions on Computers*, vol. C-24, 12 (December 1975), pp. 1235-1238.

[Kru87] B. Kruatrachue, "Static Task Scheduling and Grain Packing in Parallel Processing Systems," Ph.D. Thesis, Oregon State University, Corvallis, 1987.

[KrL87] B. Kruatrachue and T. Lewis, "Duplication Scheduling Heuristic (DSH), A New Precedence Task Scheduler for Parallel Systems," Technical Report 87-60-3, Oregon State University, Corvallis, OR, 1987.

[KrL88a] B. Kruatrachue and T. Lewis, "Grain-Size Determination for Parallel Processing," *IEEE Software*, vol. 5, 1 (January 1988), pp. 23-33.

[KrL88b] B. Kruatrachue and T. Lewis, "Grain Determination for Parallel Processing Systems," *Proceedings of the 21st Hawaii International Conference on System Sciences*, vol. 2(January 1988), pp. 119-128.

[Kun81] M. Kunde, "Nonpreemptive LP-Scheduling on Homogeneous Multiprocessor Systems," *SIAM Journal on Computing*, vol. 10, 1 (February 1981), pp. 151-173.

[LaS77] S. Lam and R. Sethi, "Worst Case Analysis of Two Scheduling Algorithms," *SIAM Journal on Computing*, vol. 6, 3 (September 1977), pp. 518-536.

[LaL78] E. L. Lawler and J. Labetoulle, "On Preemptive Scheduling of Unrelated Parallel Processors by Linear Programming," *Journal of the Association for Computing Machinery*, vol. 25, 4 (October 1978), pp. 612-619.

[LiK87] F. C. H. Lin and R. M. Keller, "The Gradient Model Load Balancing Method," *IEEE Transactions on Software Engineering*, vol. SE-13, 1 (January 1987), pp. 32-38.

[Llo82] E. L. Lloyd, "Critical Path Scheduling With Resource and Processor Constraints," *Journal of the Association for Computing Machinery*, vol. 29, 3 (July 1982), pp. 781-811.

[MTH78] H. Miyahara, Y. Teshigawara and T. Hasegawa, "Delay and Throughput Evaluation of Switching Methods in Computer Communication Networks," *IEEE Transactions on Communications*, vol. COM-26, 3 (March 1978), pp. 337-344.

[Pas87] D. M. Pase, "Load Balancing Heuristics and Network Topologies for Distributed Evaluation of Prolog," Technical Report CS/E 87-005, Oregon Graduate Center, Beaverton, OR, 1987.

[Pas88] D. M. Pase, "Contention and The Star Graph as a Network Topology," Technical Report CS/E 88-023, Oregon Graduate Center, Beaverton, OR, 1988.

[PrV81] F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Communications of the ACM*, vol. 24, 5 (May 1981), pp. 300-309.

[RCG72] C. V. Ramamoorthy, K. M. Chandy and M. J. Gonzalez, Jr., "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers*, vol. C-21, 2 (February 1972), pp. 137-146.

[SaH86] V. Sarkar and J. Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," *1986 ACM Lisp Conference (?)*, 1986, pp. 202-211.

[Sar87] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," Ph.D. Thesis, CSL-Tech. Rep.-87-328, Stanford University,

[LiK87] F. C. H. Lin and R. M. Keller, "The Gradient Model Load Balancing Method," *IEEE Transactions on Software Engineering*, vol. SE-13, 1 (January 1987), pp. 32-38.

[Llo82] E. L. Lloyd, "Critical Path Scheduling With Resource and Processor Constraints," *Journal of the Association for Computing Machinery*, vol. 29, 3 (July 1982), pp. 781-811.

[MTH78] H. Miyahara, Y. Teshigawara and T. Hasegawa, "Delay and Throughput Evaluation of Switching Methods in Computer Communication Networks," *IEEE Transactions on Communications*, vol. COM-26, 3 (March 1978), pp. 337-344.

[Pas87] D. M. Pase, "Load Balancing Heuristics and Network Topologies for Distributed Evaluation of Prolog," Technical Report CS/E 87-005, Oregon Graduate Center, Beaverton, OR, 1987.

[Pas88] D. M. Pase, "Contention and The Star Graph as a Network Topology," Technical Report CS/E 88-023, Oregon Graduate Center, Beaverton, OR, 1988.

[PrV81] F. P. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Communications of the ACM*, vol. 24, 5 (May 1981), pp. 300-309.

[RCG72] C. V. Ramamoorthy, K. M. Chandy and M. J. Gonzalez, Jr., "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers*, vol. C-21, 2 (February 1972), pp. 137-146.

[SaH86] V. Sarkar and J. Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," *1986 ACM Lisp Conference (?)*, 1986, pp. 202-211.

[Sar87] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," Ph.D. Thesis, CSL-Tech. Rep.-87-328, Stanford University,

Stanford, 1987.

[Sed83] R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, Massachusetts, 1983.

[Set76] R. Sethi, "Scheduling Graphs on Two Processors," *SIAM Journal on Computing*, vol. 5, 1 (March 1976), pp. 73-82.

[Sta84] J. A. Stankovic, "Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms," *Computer Networks*, vol. 8, 3 (June 1984), pp. 199-217, North-Holland.

[Ull75] J. D. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and System Sciences*, vol. 10, 3 (June 1975), pp. 384-393.

[Von83] C. Von Conta, "Torus and Other Networks as Communication Networks with up to Some Hundred Points," *IEEE Transactions on Computers*, vol. C-32, 7 (July 1983), pp. 657-666.

Stanford, 1987.

[Sed83] R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, Massachusetts, 1983.

[Set76] R. Sethi, "Scheduling Graphs on Two Processors," *SIAM Journal on Computing*, vol. 5, 1 (March 1976), pp. 73-82.

[Sta84] J. A. Stankovic, "Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms," *Computer Networks*, vol. 8, 3 (June 1984), pp. 199-217, North-Holland.

[Ull75] J. D. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and System Sciences*, vol. 10, 3 (June 1975), pp. 384-393.

[Von83] C. Von Conta, "Torus and Other Networks as Communication Networks with up to Some Hundred Points," *IEEE Transactions on Computers*, vol. C-32, 7 (July 1983), pp. 657-666.

Vita

The author was born, which, all things considered, was a very good start indeed. He spent the bulk of his youth studying the biological and geological sciences in exotic locations such as Eagar Arizona. Eventually tiring of worms and dirt, the author's ever curious mind turned to the black art of Mathematics. This led him to that great Citadel of Intellectual Prowess, Northern Arizona University, where he became exceedingly proficient at holding meaningful conversations with inanimate objects. This unusual talent led him to obtain a Bachelor's Degree in Mathematics, with a dual major in Computer Science.

Somehow during his stay at NAU he managed to meet up with the wonderfully desirable Anne Cecile Heil. Through what can only be called an astounding display of fabrication, exaggeration, and outlandish promises he convinced her to marry him, which was certainly the best thing *he* ever did.

Through the natural course of events, one beautiful daughter came along, then another. Each of these wonderful girls delighted the eye and enchanted the soul of all who met them. In the words of one astute observer, "Either those kids aren't his, or Nature's playing tricks on us again."