Performance Effects of Scheduling Strategies for Master/Slave Distributed Applications

Gary Shao*and Rich Wolski†and Fran Berman‡
University of California, San Diego
UCSD CSE Dept. Technical Report #CS98-598, September 1998

Abstract

The achievement of parallel application performance on non-dedicated workstation clusters requires careful attention to the scheduling of tasks and communication on the underlying platform. In the literature, application scheduling policies are usually chosen by matching the resource requirements of an application with the performance characteristics of the target platform. However, when clusters of workstations are shared with other users, platform performance is non-uniform and varies over time. As a result, the performance of distinct scheduling policies may also vary depending on dynamic system state and particular characteristics of the job being run.

Our experimental work focuses on a master/slave parallel ray-tracing application executing on a set of workstation clusters at UCSD and the San Diego Supercomputer Center. The experiments show that two different scheduling strategies, one static and one dynamic, exhibit very different performance sensitivities to variabilities in resource capabilities and workload distribution. We demonstrate for our example application that neither scheduling strategy by itself consistently induces the best application performance (minimal execution time) when running on the same resources under normally experienced production operating conditions. These results support the idea that dynamic selection of appropriate scheduling strategies to match run-time conditions provides a promising approach to achieving application performance for master/slave applications on heterogeneous time-shared workstation clusters.

1 Introduction

Harnessing the power of widely available clusters of time-shared workstations for running simply parallelized versions of ordinary sequential codes is an attractive and highly cost-effective way to deliver increased application performance. Achieving good performance from applications running in these heterogeneous and dynamic environments requires that applications be able to schedule their activities to match time-varying resource conditions and capabilities with application requirements. We are interested in investigating the essential characteristics of scheduling strategies that can be successful in making this match and delivering good application performance over a wide range of conditions.

*Research supported by NASA GSRP Fellowship #NGTS-50131.

†Research supported in part by NSF grant #ASC-9701333, Darpa grant #N66001-97-C-8531, DoD Modernization Program.

‡Research supported in part by NSF grant #ASC-9701333, Darpa grant #N66001-97-C-8531, DoD Modernization Program.

This paper examines how schedulers interact with applications and environments to strongly influence parallel performance. In particular, we focus on the instantiation of scheduling parameters which promote application performance in dynamic environments. Such scheduling parameters can generally be divided into two types: *observables* and *controllables*. Observables are those parameters which effect the performance of an application, and/or have a state which can be monitored, but which are not under direct control of a scheduler. Controllables are those parameters which affect performance and have a state that can be directly manipulated by a scheduler. Using the example of a master/slave ray-tracing application which breaks a large image into a number of smaller image blocks for parallel processing, observable parameters a scheduler might choose to monitor while making scheduling decisions include: processor speed, processor availability, and work load distribution of image blocks. Controllable parameters for the ray-tracing application might include: the time and frequency at which blocks are distributed to slaves, the number of blocks to assign each slave processor, and the size and shape of each block.

Observable parameters and controllable parameters form the mechanism by which schedulers perform their functions. Scheduling *policies* are the rules which specify how and when mechanisms are invoked to produce scheduler actions. We define *scheduling strategies* as specific combinations of observable parameters, controllable parameters, and scheduling policies. Scheduling strategies may differ when they use different sets of observable data to make their scheduling decisions, employ different sets of controlling mechanisms to perform scheduling actions, or are governed by distinctly different sets of policies.

In this paper, we demonstrate that scheduling strategy has a great impact on application performance in dynamic environments. In particular, experimental data will be used to show how scheduler performance is heavily influenced by the degree of uncertainty attached to scheduler handling of observable and controllable parameters. The results suggest that success in handling parameter variabilities can be a powerful determinant in deciding which scheduling strategies will have the most success for a given set of conditions. While all the experimental work presented in this paper was done with a single ray-tracing application, many of the scheduling concerns expressed here are believed to have relevance to scheduling other master/slave implementations running on time-shared systems as well.

Note that our findings showing that static scheduling approaches in ray-tracing suffer from the high variability in workload distribution within the image being ray-traced are consistent with numerous other studies of parallel ray-tracing performance on distributed-memory machines [1][2][3][4][5]. We differ from these previous studies by considering as a factor the heterogeneous and time-varying behavior of resources operating in an open non-dedicated environment. We are also studying strictly master/slave configurations, so we consider only cases where work assignments must all originate from the master process.

2 Scheduling for Master/Slave Performance

Consider a simple model for master/slave application performance where all the slaves i begin processing at the same time and run for time T_i. The execution time T_{finish} for an application will be the time it takes for the last slave to finish its work.

$$T_{finish} = \max(T_i). \tag{1}$$

Each slave's execution time will be the sum of the time $T_{comp,i}$ required for computation and the time $T_{comm,i}$ required for performing communication with the master. In our general model, we assume the worst case, i.e. that communication and computation are not overlapped. But in many master/slave implementations, effective techniques can be employed to overlap communication time with computation time, leaving the computation time as the dominant factor to consider.

$$T_i = T_{comp,i} + T_{comm,i}. \tag{2}$$

Computation time can be modeled as the amount of work w_i sent to each processor i divided by the rate each processor is able to process the work. In a time-shared system with multiple competing processes, the rate $P_{speed,i}$ at which a processor can perform work will actually be a time-varying function of the conditions present at time t_i.

$$T_{comp,i} = w_i / P_{speed,i}(t_i). \tag{3}$$

For a given amount of work, the shortest execution time occurs when all processors can be kept busy doing useful work and they all finish at roughly the same time. The goal of attempting to minimize execution time can thus be cast as an effort to distribute the total work $W = \sum w_i$ in a way which equalizes the finishing times of all the slaves involved in a calculation, a process which is strictly *time balancing*, but is commonly referred to as *load balancing*.

Load balancing strategies can be divided into two categories: *static load balancing* and *dynamic load balancing*[6]. Static load balancing attempts to calculate the proper distribution for each w_i once at application startup, keeping the same work distribution throughout the life of the computation. The weakness of static strategies is their susceptibility to load imbalances caused by time varying changes in the work rate of individual processors and the uncertainties of accurately predicting data-dependent workloads. Dynamic load balancing strategies allow work to be redistributed while an application is running, in an effort to compensate for varying and unpredictable load imbalances. A potential drawback of dynamic load balancing approaches is that they can add additional computational overhead, and may introduce additional communication time which cannot be effectively overlapped with computations[7].

In order to help us evaluate the performance of different load balancing strategies, we define the metric of **resource efficiency** RE as follows:

$$RE = \sum_{i=1}^{P} \left(\frac{T_{comp,i}}{T_{finish}} * \frac{P_{speed,i}(t_i)}{\sum_{j=1}^{P} P_{speed,j}(t_j)} \right). \tag{4}$$

The first multiplicative term reflects the percentage of total execution time during which a slave processor is computing useful results. In a system with heterogeneous resource characteristics, the second multiplicative term in Equation 4 is included to fairly weight the contributions of each processor by the percentage of computational power they are actually delivering to the computation. A maximum resource use efficiency of 1.0 is achieved when all processors have computed for the same amount of time. For problems which benefit from distribution, higher efficiency correlates with smaller execution time[8].

3 Ray-tracing Application

For this study, we selected a master/slave adaptation of a sequential program to investigate the issues of load-balancing performance in a heterogeneous time-shared computing environment. We chose the PVMPOV [9] ray-tracer, a master/slave parallel adaptation of the popular POV [10] ray-trace program that has been modified to run under the PVM [11][12] system, as our test

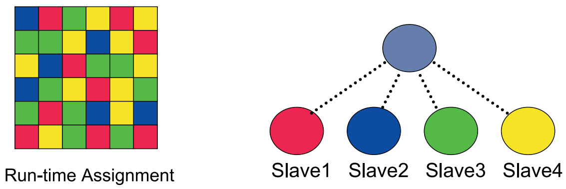

Figure 1: **Fixed Distribution.** All blocks are assigned to slaves at the beginning of the computation. Distribution of the blocks is made in proportion to the estimated performance capabilities of the slaves.

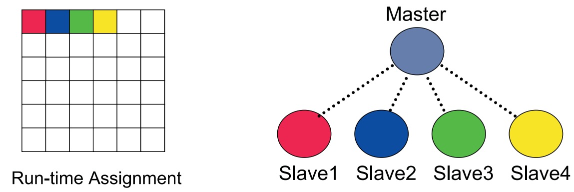

Figure 2: **Work Queue.** Slaves are initially assigned a single block to work on. Other blocks are assigned one-at-a-time by the master process to slaves that finish processing a block and then request more work.

application. The basic function of ray-tracing programs is to calculate the colors of display pixels in a 2-dimensional image to realistically represent 3-dimensional scenes that have been modeled in a geometric description language. Because the act of calculating pixel colors does not involve any dependencies between individual pixels, ray-tracing is a natural candidate for applying parallel processing to reduce overall application execution time.

The approach used by PVMPOV in parallelizing a ray-tracing application is to divide up the display image into equal-sized rectangular blocks of pixels, and then to distribute the work of calculating pixels in each block to a number of different slave processors. Distributing the blocks to the slaves is done by a single master process, running on the machine where the job is first started. As slave processes finish the job of rendering the pixels in a block, they send the pixel data back to the master process to be assembled into the final image. Once finished, the image is written to a file by the master process in an image format specified by the user. Note that to warrant parallel processing, the work needed to calculate an image must be large enough to offset any overhead time required for communication with tangible benefits due to concurrent computation.

Our interest in this application focuses on the impact of the scheduling strategy chosen to assign blocks to processors to achieve the the shortest time for computing an entire image. We consider two load balancing strategies, one static and one dynamic: (1) a static fixed allocation of all blocks at the beginning of the computation, and (2) dynamic run-time allocation of blocks one-at-a-time as needed by slaves free to do work. We will refer to the first strategy from now on as *fixed distribution* scheduling, and the second strategy as *work queue* scheduling. Figures 1 and Figure 2 illustrate how blocks are allocated at execution time for each of the two strategies. Work queue scheduling was the only strategy implemented for PVMPOV before our experiments began, so we created an implementation of fixed distribution scheduling for our ray-tracing application and modified PVMPOV to accept command-line arguments for choosing which strategy to use.

Load balancing scheduling policies for the ray-tracing application in a distributed time-sharing environment must address two issues: (1) compensating for the variability in workload distribution between the different blocks of a typical scene being traced (which cannot easily be determined beforehand), and (2) accounting for the heterogeneous and time-varying performance characteristics of resources in the system. These factors are important because they make the job of determining the performance efficient distribution of work to slaves a difficult task to achieve.

3.1 Fixed distribution scheduling

The general principle behind fixed distribution scheduling is to assign blocks to slave processors in a way that results in slaves being allocated work that is directly proportional to the capacity of the processors to do the work. By making the allocation decisions only once, at the beginning of the computation, unnecessary overhead costs and delays caused by master-slave communication in support of dynamic block allocation are avoided.

Our implementation estimates each processor i's capacity to do work with a speed metric calculated as:

$$P_{speed,i}(t_i) = BenchMark_i * AvailCPU_i(t_i). \tag{5}$$

$BenchMark_i$ is a measured benchmark which relatively rates the speed with which an unloaded processor performed the ray-tracing function on a fixed-size image of a specific test scene. The slowest processor in our experimental setup is given a rating of 1000, and all other processors are rated relative to this reference machine. We are only concerned with relative speeds, so the choice of 1000 as a reference value is not important as long as higher numbers always correspond to proportionally faster machines, i.e. a machine with a rating of 2000 was measured to be twice as fast as a machine with a rating of 1000. $AvailCPU_i$ is a dynamic run-time estimate of CPU availability for processor i. This value is computed by combining prediction information from the Network Weather Service (NWS [13], a resource monitoring and prediction system) with past application history data if it is available. Availability is expressed as a fraction between 0.0 and 1.0, where 1.0 indicates all CPU cycles are available for use by a new process.

The number of blocks B_i to allocate to each processor i is given by:

$$w_i = B_i = B_{total} * \frac{P_{speed,i}(t_i)}{\sum_{j=1}^{P} P_{speed,j}(t_j)}. \tag{6}$$

The B_i blocks allocated to processor i are selected from a randomized list of the B_{total} blocks in the image.

3.2 Work queue scheduling

Work queue scheduling attempts to handle both the variability in resource performance and the variability in workload distribution by deferring allocation decisions for as long as possible. Blocks are not allocated to slave processors until the slaves indicate to the master that they are free to do more work. In this way, slower processors will tend to be assigned fewer blocks over time, and no more than one block which takes a long time to render will be assigned to any one processor at a time.

This method of allocating a single block at a time does require additional communication between the slave processes and the master. In order to prevent slaves from waiting for a response from the master after they have sent notification of a need for more work, each slave in the PVMPOV application sends a request for more work as soon as it receives a new block allocation from the master. In this way, the master will have time to receive the new request and send the next block back to a slave while the slave is performing calculations on the previous block. In most cases, it is expected that few significant delays should be incurred by a slave having to sit idle while waiting for a work request to be answered by the master.

4 Experimental Environment

In order to evaluate and quantify the influence of observable and controllable scheduling parameters on application performance, we conducted a number of experiments while running our ray-tracing

Processor Type	Clock (MHz)	Memory (MB)
Sun SPARC-4	85	32
Sun SPARC-5	85	32
Sun SPARC-5	110	32
Sun SPARC-10	50	32
Sun SPARC-10(2)	50	64
Sun UltraSPARC	167	128
Sun UltraSPARC	200	128
Intel Pentium Pro	200	128
Intel Pentium II	266	128
DEC 3000/400	75	256

Table 1: Hardware platforms used for ray-tracing experiments.

application. These experiments were conducted on both heterogeneous and homogeneous clusters of computers. Table 1 shows the different types of workstations and high-performance personal computers (PC's) which were available for our use. The machines were networked together in a campus-wide area network configuration consisting of interconnected local-area networks (LANs). The Sun and Intel-based machines were connected together by a combination of 10 and 100 Mbit/sec ethernet LAN interfaces at the University of California, San Diego (UCSD), while the DEC workstations were linked together with a high-speed switched interconnect network at the San Diego Supercomputer Center (SDSC).

Each of the machines was operated in a non-dedicated, production mode. Outside users were free to start competing processes at any time. It should be noted that the machines at UCSD experienced distinctly different loading patterns than the machines at SDSC. The UCSD machines were primarily used as interactive terminals by students in the daytime, so computational loads were generally light and of short duration. The SDSC machines, on the other hand, were almost always utilized as parallel computational platforms for long-running jobs, resulting in multiple competing processes being present for most of the time our experiments were being run.

5 Experimental Results

We present in this section experimental data collected while running the ray-tracing application on different clusters of time-shared workstations and PC's. Because we are interested in illustrating some time-varying aspects of application behavior in this environment, some of the graphs take the form of traces which track the progress of characteristics we are interested in observing over a period of time. Unless otherwise indicated, all trials were conducted using an image size of 512 by 512 pixels, and a block size of 32 by 32 pixels. Trials were conducted using multiple sets of processor configurations, but in the interest of space only a few representative samples are presented here.

5.1 Scheduling performance

In the first set of experiments, we compared the performance of our two load balancing strategies when applied to ray-tracing two different scenes. One test scene, referred to as the "large workload"

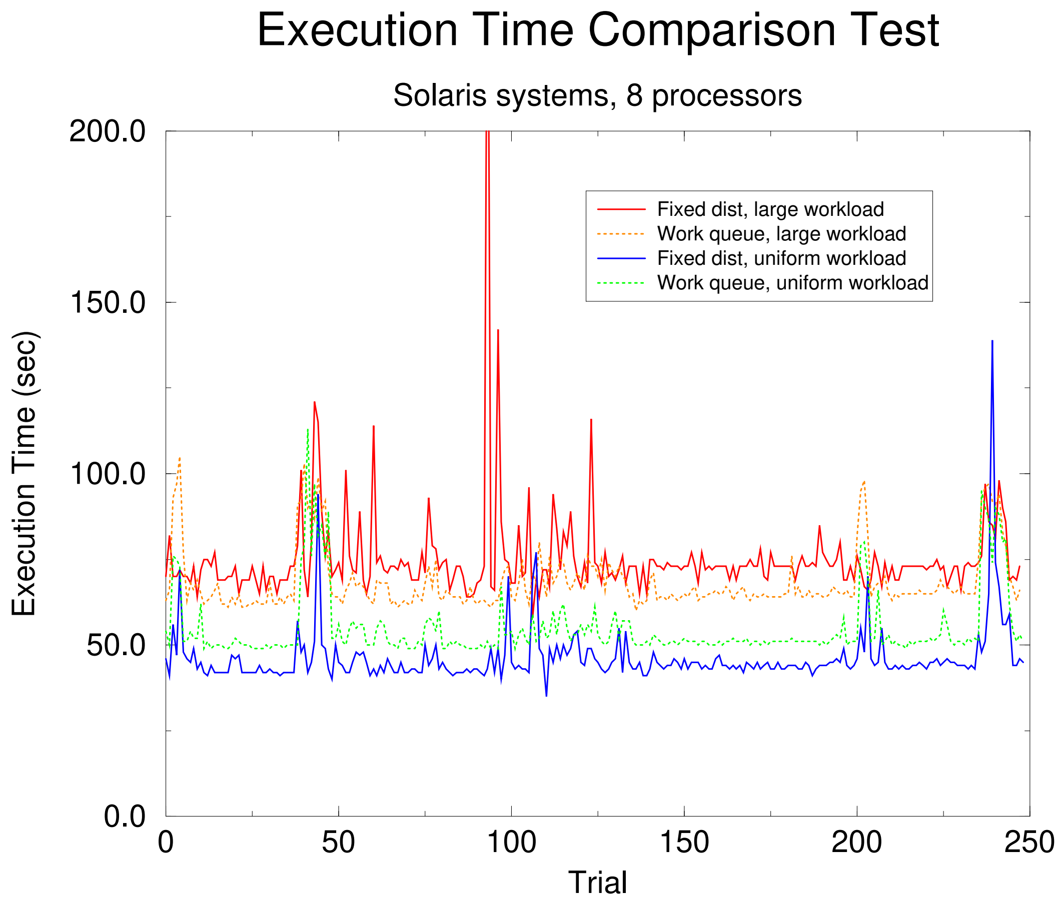

Figure 3: Scheduling strategies exhibit different performance results for different test images.

case, is a scene commonly used as a benchmark for evaluating the performance of platforms to which the POV ray-tracer has been ported (see Figure 4). The second scene, referred to as the "uniform workload" case, was designed to have an almost uniform work distribution across the entire image (see Figure 5). It was created for the purpose of allowing observation of ray-tracing scheduling performance while free of the effects of workload imbalance in the input scene. Trials were run back-to-back on a cluster of eight Sun workstations, alternating the combinations of scheduling strategy and test scene with each trial to expose each combination to approximately the same conditions over the course of the experiment. Figure 3 shows execution time traces for a representative set of trials.

The experiments indicated that work queue scheduling is consistently better for the large workload case (9.6% better on average), while fixed distribution scheduling is consistently better for the uniform workload case (18.6% better on average). Resource efficiency traces for the same trials as shown in Figure 3 are shown in Figure 6 and Figure 7. In particular, **no one scheduling strategy was best for all combinations of load conditions and data sets**. In order to explain this phenomenon, a careful analysis was performed on data from the trials to identify the major factors contributing to the performance results shown here.

Figure 4: Large workload example image.

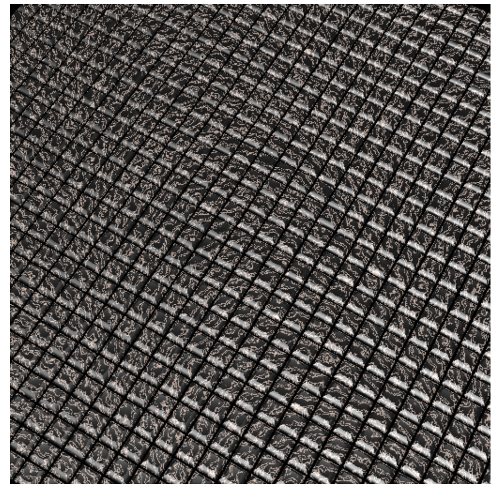

Figure 5: Uniform workload example image.

5.2 Resource efficiency

In Figure 6 we plot resource efficiency for the trials with our two scheduling strategies applied to the large workload case. The graph of resource efficiency appears to be consistent with the execution time graph in Figure 3, as higher RE values correspond to lower execution times. In addition, the resource efficiency graph gives us a qualitative measure for how well the different strategies are performing relative to the best expected results. We observe that work queue load balancing has an average RE of nearly 91%, while fixed distribution scheduling has an average of under 83%. The RE metric suggests that, for the conditions of this experiment, there remains a small amount of potential performance that is not being utilized by work queue scheduling.

If we look at the graph of resource efficiency for trials applying our two scheduling strategies to the uniform workload case, we observe a condition which is almost the inverse of the large workload case. Figure 7 shows that fixed distribution scheduling now has a decided advantage over work queue scheduling in terms of our RE metric. Fixed distribution scheduling has a resource efficiency average of 93%, while work queue scheduling averages only 81%.

5.3 Work allocation accuracy

To help explain why fixed distribution scheduling might exhibit poor efficiency under some conditions, we looked at several factors. We first considered the manner in which blocks are allocated to the different processors, a controllable parameter under the fixed distribution scheduling strategy. Equation 6 shows that the allocation is made by taking fractional parts of the total number of blocks in the image, with the assumption that selecting from a randomized list of blocks will even out non-uniformities in block calculation times. For many ray-traced scenes, the work distribution among blocks is quite irregular, and may span a fairly wide range. The result is that the actual proportion of work being assigned to a processor will not be the same as the proportion of blocks being allocated. Figure 8 shows a trace of the worst differences between the proportion of work and the proportion of blocks assigned for fixed distribution scheduling during each trial of the experiment.

Observed inaccuracies in the allocation of blocks illustrate that variance in the accuracy of controllable actions can have serious effects on overall scheduling performance. The difference of

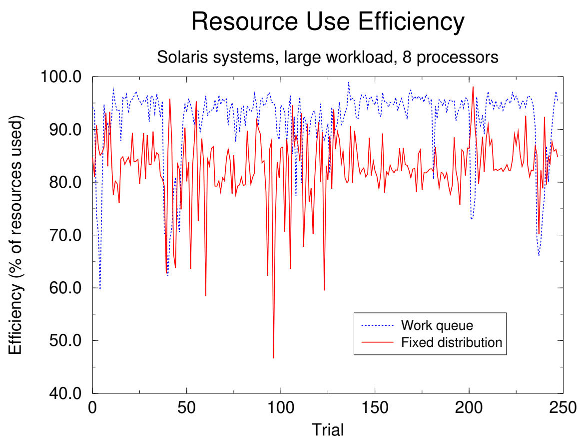

Figure 6: Resource efficiency is consistently higher for work queue scheduling of large workload example when compared to fixed distribution scheduling.

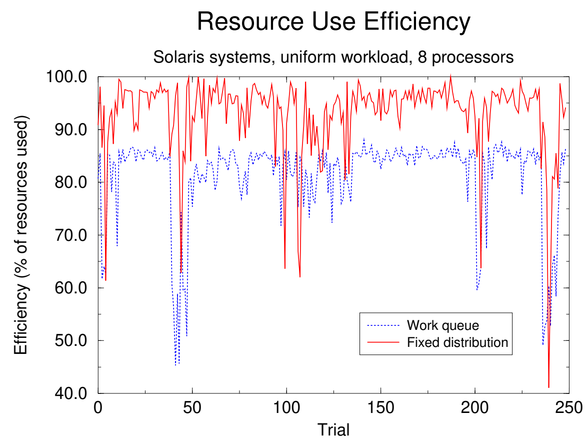

Figure 7: Resource efficiency is consistently higher for fixed distribution scheduling of uniform workload example when compared to work queue scheduling.

approximately 20% shown for the large workload case is significant, since it means that at least one processor in every run was allocated a nearly 20% greater amount of work than what was calculated to promote a good balance. This error in work allocation by itself is a prominent reason why fixed distribution in the large workload case does relatively poorly, since execution time is determined by the finishing time of the worst performing slave in each trial. As the work allocation error appears as a consistently large factor in every trial, it helps explain why fixed distribution scheduling for the large workload case is consistently handicapped when compared to work queue scheduling.

5.4 Processor prediction errors

The second factor we investigated was the effect of inaccuracies in the prediction of processor availability ($AvailCPU_i$) which is used in Equation 5. $AvailCPU_i$ is an example of an observable parameter for the fixed distribution scheduling strategy. Figure 9 shows a trace of the maximum difference between the predicted availability for a processor and the processor availability actually delivered during execution for each trial using fixed distribution scheduling. Observing the behavior of the trace for the large workload trials and comparing it with the efficiency graph of Figure 6, we can observe a high correlation between many of the large downward spikes in efficiency with the large upward spikes in processor prediction errors. A similar observation can be made about the correlation between upward spikes in the execution time graph of Figure 3 for fixed distribution scheduling of the large workload case with instances of large prediction errors. Specific examples can be noted by observing that four occurrences of maximum prediction error exceeding 75% correspond to trials when execution time exceeded average execution time by 50% for fixed distribution scheduling. Observed inaccuracies in the prediction of CPU availability illustrate that variance in the accuracy of observing and predicting observable conditions can also have serious effects on overall scheduling performance.

It should be noted that the traces in Figure 9 show that there were significant periods of time when even the worst estimate of machine availability proved to be relatively accurate, achieving an accuracy within 10% of the actual recorded value. There were also a few instances where prediction inaccuracy exceeded 50%. This behavior demonstrates one example of how monitoring dynamic

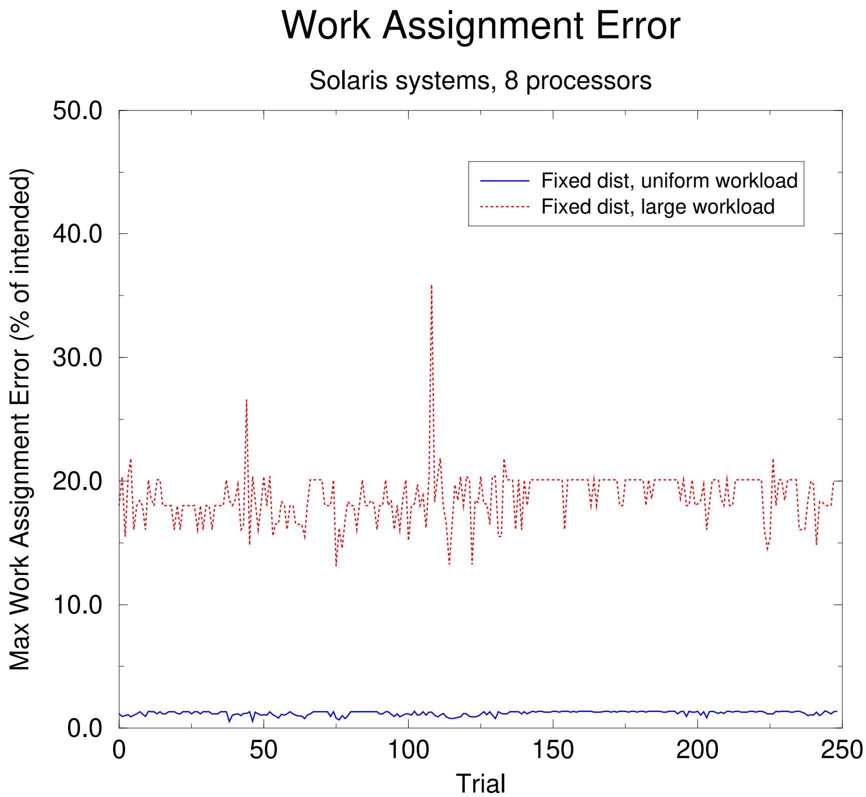

Figure 8: Errors in actual work allocation versus intended work allocation are consistently high for at least one processor, contributing strongly to poor performance of fixed distribution scheduling in the large workload case.

system characteristics often leads to highly time-varying accuracy in the observations made.

5.5 Effects of controllable parameters

Applications often include the ability to change certain program parameters for tuning performance in response to conditions experienced at run-time. These are **controllable parameters**. Schedulers should be aware of these controllable parameters, and include the ability to effectively modify those which may lead to better performance. In an additional set of experiments, we examined the effects of varying two controllable block characteristics: *size* and *shape*. We had seen in earlier results that work allocation error was a major contributor to poor fixed distribution scheduling performance. We were interested in searching for ways in which the inter-block work variance could be improved without requiring advance knowledge of actual workload distributions in an image.

One potential way to reduce the detrimental effects of highly variant inter-block work distribution is to reduce the size of each block, effectively increasing the total number of blocks to be processed. Figure 10 shows the effects on average execution time for a series run with fixed distribution scheduling of the large workload case. This graph shows the most desirable region of operation to be at the 16x16 block size, but the difference between the best and worst average execution times was not large, amounting to 9.3% of the best time shown. Other trials not shown here, with

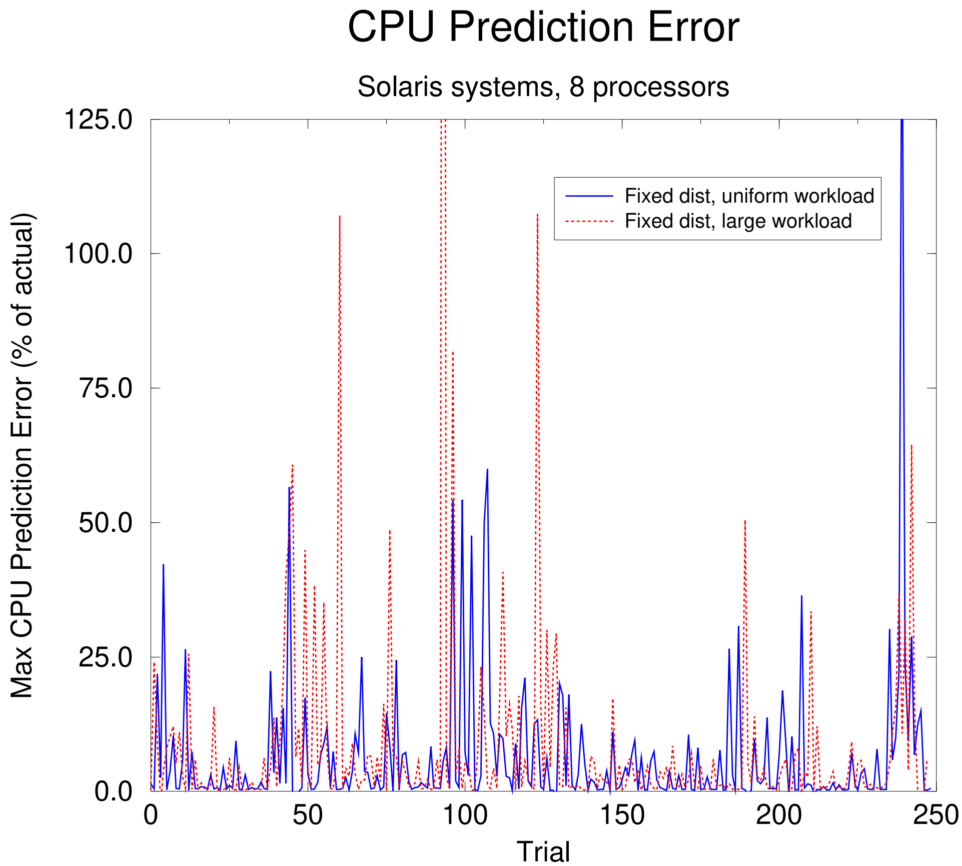

Figure 9: Errors in predicting processor availability are a function of time-varying resource characteristics, and at times contribute heavily to reduced performance of fixed distribution scheduling in both workload cases.

different combinations of scheduling policies and test images, showed similarly sized effects due to increasing or decreasing block size.

A second potential way to reduce work distribution variance between blocks is to consider reducing the spatial locality of pixels within each block, making blocks either taller and thinner or shorter and wider. We wanted to know whether radically changing the block shape might improve the ability to accurately perform work allocations. Using fixed distribution scheduling for the large workload image, trials were run on a heterogeneous cluster containing three Intel-based high-performance PC's and one Sun UltraSPARC workstation for slaves. Consecutive trials were run, varying the block shape between ones that were either square, tall and thin, or short and wide. For comparison purposes, trials using work queue scheduling for the large workload image were included in the experiment. Figure 11 shows the average execution times of the different cases observed in the experiment.

The results show a moderate improvement in average execution time by moving away from the default square block shape. Fixed distribution scheduling with the original 32x32 block shape performed 20% worse on average than work queue scheduling, while a tall and thin 8x128 block shape which contains the same number of pixels performs only 7% worse than work queue scheduling

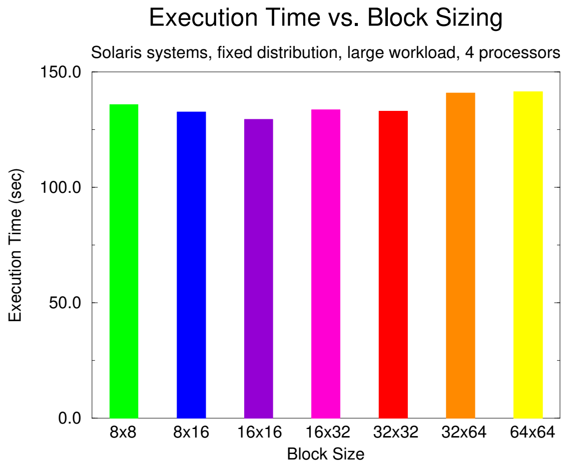

Figure 10: In the large workload case, changing block size for fixed distribution scheduling has only a small effect on application performance.

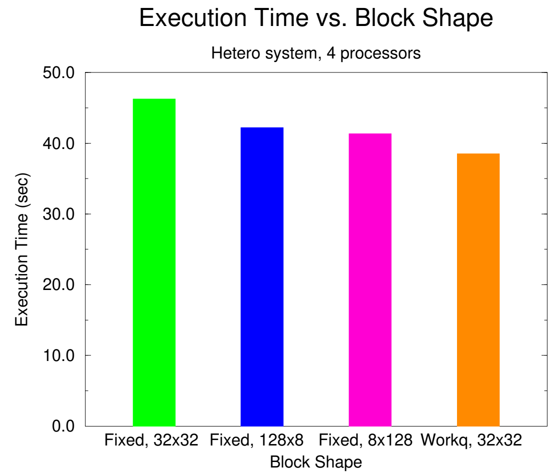

Figure 11: In the large workload case, better random distribution of workload was observed to occur in fixed distribution scheduling by using non-square block shapes.

on average. Since all of the blocks in the trials contain the same number of pixels, and the formula for calculating work allocation for all trials was the same, the difference in performance between the cases can be explained by a change in the pattern of work distribution between the blocks.

6 Conclusion

We have shown in this paper, for an example ray-tracing application, that a number of observable and controllable scheduling parameters have a direct influence on overall application performance. This makes it difficult for any single scheduling strategy to provide the best execution performance for all expected combinations of data sets and environmental conditions. Traditionally, applications are scheduled according to a single scheduling policy, considered the most appropriate for that class of application. This single scheduling strategy operates by essentially deriving values for a set of controllable parameters based on current knowledge of observable states and hard-coded decisions made by the scheduler developer. We would like to make a case that this standard approach to scheduling is not sufficiently flexible to accommodate the complex performance interactions of parallel applications running on distributed heterogeneous time-shared resources. In the set of experiments described in the previous sections, we have attempted to show how specific characteristics of an application and the environment it runs in can impact the performance of different scheduling strategies.

Oftentimes characteristics which adversely effect one scheduling strategy can be found to have fewer negative effects, and possibly even some positive effects, on an alternative strategy. To the extent these complementary strategies can be identified and properly evaluated during the scheduling process, adaptive selection of the best scheduling strategy should result in better application performance under all conditions. Our challenge is to be able to effectively evaluate the relative merits of multiple scheduling strategies, based on the application characteristics and existing and predicted environmental conditions; and from this evaluation select the one strategy which delivers the best performance under current conditions. Because we believe in the positive performance benefits that come from the added flexibility which selection between multiple strategies promises, we are preparing further investigations to test the benefits of extending this *adaptive scheduling* philosophy beyond the master/slave model presented in this paper to more general classes of applications.

References

[1] A. Heirich and J. Arvo, "A competitive analysis of load balancing strategies for parallel ray tracing," *Journal of Supercomputing*, vol. 12, pp. 57–68, 1998.

[2] T.-Y. Lee, C. S. Raghavendra, and J. B. Nicholas, "Parallel implementation of a ray tracing algorithm for distributed memory parallel computers," *Concurrency: Practice and Experience*, vol. 9, pp. 947–965, 1997.

[3] E. Reinhard and F. W. Jansen, "Rendering large scenes using parallel ray tracing," *Parallel Computing*, vol. 23, pp. 873–885, 1997.

[4] A. Reisman, C. Gotsman, and A. Schuster, "Parallel progressive rendering of animation sequences at interactive rates on distributed-memory machines," in *Proceedings of the IEEE Symposium on Parallel Rendering, PRS '97*, pp. 39–47, October 1997.

[5] C. N. Sekharan, V. Goel, and R. Sridhar, "Load balancing methods for ray tracing and binary tree computing using pvm," *Parallel Computing*, vol. 21, pp. 1963–1978, 1995.

[6] O. Kremien and J. Kramer, "Methodical analysis of adaptive load sharing algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 747–760, 1992.

[7] M. J. Zaki, W. Li, and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations," in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, HPDC '96*, August 1996.

[8] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Transactions on Computers*, vol. 38, pp. 408–423, 1989.

[9] A. Dilger and H. Deischinger. PVMPOV patch to POV source code. `http://www-mddsp.enel.ucalgary.ca/People/adilger/povray/pvmpov.html`.

[10] Persistence of Vision Development Team, Persistence of Vision (POV) Ray-tracer. `http://www.povray.org`.

[11] G. A. Geist and V. S. Sunderam, "Network based concurrent computing on the pvm system," *Journal of Concurrency: Practive and Experience*, vol. 4, pp. 293–311, 1992.

[12] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[13] R. Wolski, "Forecasting network performance to support dynamic scheduling using the network weather service," in *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing, HPDC '97*, August 1997.