The Relative Performance of Various Mapping Algorithms is Independent of Sizable Variances in Run-time Predictions *

Robert Armstrong
Debra Hensgen
Taylor Kidd

Computer Science Department
Naval Postgraduate School
Monterey, CA 93940

Abstract

In this paper we study the performance of four mapping algorithms. The four algorithms include two naive ones: Opportunistic Load Balancing (OLB), and Limited Best Assignment (LBA), and two intelligent greedy algorithms: an O(nm) greedy algorithm, and an $O(n^2m)$ greedy algorithm. All of these algorithms, except OLB, use expected run-times to assign jobs to machines. As expected run-times are rarely deterministic in modern networked and server based systems, we first use experimentation to determine some plausible run-time distributions. Using these distributions, we next execute simulations to determine how the mapping algorithms perform. Performance comparisons show that the greedy algorithms produce schedules that, when executed, perform better than naive algorithms, even though the exact run-times are not available to the schedulers. We conclude that the use of intelligent mapping algorithms is beneficial, even when the expected time for completion of a job is not deterministic.

1 Introduction

This paper describes the experiments and simulations that we executed to determine the relative performance of certain mapping algorithms in different heterogeneous environments. In this paper we assume that all jobs are independent of one another. That is, they do not communicate or synchronize with one another. This type of architecture is common in today's LAN-based distributed server environment.

Our goal was to determine whether using intelligent mapping algorithms would be beneficial, even if the jobs did not run for exactly the amount of time expected. Intelligent mapping algorithms utilize the expected run-times of each job on each different machine to attempt to minimize some scalar performance metric. For our experiments, this metric is the time at which the last job completes. In particular, we were concerned about whether it would still be beneficial to use intelligent mapping if one or several jobs run for a substantially different amount of time than expected, but are still accurately characterized statistically. Because determining a perfect mapping is an NP-complete problem, we examined the performance of several different (polynomial) heuristics. The algorithms we chose are listed below.

- A naive $O(n)$ algorithm known as Opportunistic Load Balancing (OLB). This algorithm simply places each job, in order of arrival, on the next available machine.
- A simple $O(nm)$ algorithm known as Limited Best Assignment (LBA). This algorithm uses the expected run-time of each job on each machine. It assigns each job to the machine on which it has the least expected run-time, ignoring any other loads on the machines, including that produced by the jobs that it has assigned.

  This algorithm, though easily implementable in a scheduling framework that automatically assigns jobs to machines, is very similar to the algorithm used by many users who remotely start their jobs by hand at supercomputer centers without examining queue lengths.
- Two greedy algorithms, one of order $O(nm)$ and the other of order $O(n^2m)$. Both of these algorithms make use of the expected run-time of each job on each machine as well as the expected loads on each machine. These algorithms will be more fully described in Section 2.

*This research was supported by DARPA under contract number E583. Additional support was provided by the Naval Postgraduate School and the Institute for Joint Warfare Analysis.

The primary reasons for our study are both that jobs rarely execute for exactly the expected run-time and often the expected run-times are not exactly known. In a system where each job has exclusive use of a machine, differences between actual and predicted run-times occur either because (1) all of the compute characteristics [10] are not known or enumerated by the designer of the program, or (2) because the time to access memory and disk is stochastic and not deterministic. Of course, in many environments, additional non-determinism is due to other jobs running on the machine or simultaneously using a shared network or a shared file server. This paper focuses on those cases where one or more of the jobs being scheduled have run-times that could differ substantially from the expected run-time. For those cases, we seek to determine whether there is still an advantage to using an algorithm that makes use of expected run-times or whether a computationally simpler algorithm that does not require estimating run-times, such as Opportunistic Load Balancing (OLB), might not yield equivalently good performance.

In the next section, we describe the two greedy algorithms that we used in our experiments and simulations. We then describe our experiments concerning the non-determinism of expected run-times and examine, using the derived distributions in simulations, the performance of the intelligent algorithms. That is, we collect run-times for various jobs on various machines, analyze their distributions, and extrapolate these distributions for use in our simulations. We conclude the paper with a short summary and comparison to related work.

2 The Greedy Algorithms

In addition to the simple OLB and LBA algorithms described in the previous section, our experiments used two greedy algorithms. We now describe those algorithms in detail.

The first algorithm is an $O(nm)$ algorithm, where n is the number of jobs and m is the number of machines, and the second algorithm is of order $O(n^2m)$. Each algorithm first estimates the expected run-time of each job on each machine, assuming that if a job cannot execute on a particular machine, the estimation will be set to some very large number. As we describe these algorithms we will consider these expected run-times as elements of a 2-dimensional, n by m matrix called A. That is, $A[i,j]$ is the expected run-time of job i on machine j.

The $O(nm)$ algorithm, which, like in the SmartNet documentation [6], we will call Fast Greedy, considers the jobs in the order requested[1]. It first determines the value $A_{1,j}$, such that $A_{1,j} \leq A_{1,k}\ \forall\ k \in \{1..m\}$. It then assigns job 1 to machine j. Following this, it adds $A_{1,j}$ to all $A_{i,j}\ \forall\ i \in \{2..n\}$. Then, for each remaining job, $p \in \{2..m\}$, it determines the value $A_{p,j}$, such that $A_{p,j} \leq A_{p,k}\ \forall\ k \in \{1..m\}$. It then assigns job p to machine j. Following this, it adds $A_{p,j}$ to all $A_{i,j}\ \forall\ i \in \{p+1..n\}$. At each step, then, it is assigning each job to its best machine, given the previous assignments. We note that the jobs are assigned in the order in which they were requested.

The $O(n^2m)$ algorithm, which again borrowing from SmartNet nomenclature we call simply Greedy, actually computes two mappings using two different sub-algorithms and then chooses the mapping that gives the smallest sum of the predicted run-times, minimized over all machines. The two sub-algorithms are similar to the first greedy algorithm above, differing only in the order in which they assign jobs to machines. We first enumerate the steps of the first sub-algorithm.

1. Initialize the set $\{RemainingJobs\}$ to contain all jobs.
2. $\forall\ i \in \{RemainingJobs\}$, find $A_{i,j} \leq A_{i,k}\ \forall\ k \in \{Machines\}$. Call such an $A_{i,j}$, A_{i,min_i}.
3. Determine p such that $A_{p,min_p} \leq A_{i,min_i}\ \forall\ i \in \{RemainingJobs\}$.
4. Remove p from $\{RemainingJobs\}$, scheduling job p on machine min_p.
5. Add A_{p,min_p} to $A_{i,min_p}\ \forall\ i \in \{RemainingJobs\}$.
6. If $\{RemainingJobs\}$ is not empty, return to step 2.

The idea behind this first sub-algorithm is that, at each step, we attempt to minimize the time at which the last job, which has been thus far scheduled, finishes.

The second sub-algorithm differs from the first sub-algorithm in that, at the third step, it finds p such that $A_{p,min_p} \geq A_{i,min_i}\ \forall\ i \in \{RemainingJobs\}$. This algorithm, then tries to minimize the worst case time at each step.

3 Effect of Non-Determinism on Algorithm Performance

We now examine the effect of non-determinism on the performance of the greedy and LBA algorithms that we described above. Our reason for studying this

[1] In describing these algorithms, we use the term *order requested* to mean the order in which the job requests have been placed prior to invocation of the algorithm. We also investigated the performance of these algorithms if jobs are first sorted before these algorithms are invoked.

is because both the LBA and the greedy algorithms use the expected run-time to produce their mappings. One of our major motivations for this work is to determine whether such intelligent algorithms are still useful if the actual run-time is non-deterministic, that is, essentially sampled from a distribution around the expected run-time. In order to determine what distributions we should sample our run-times from in our simulation, we first conducted some experiments with actual programs to try to determine what types of distributions characterize their run-times.

3.1 Job Run-time Distributions

We have already explained why job-machine run-times are typically not constant, but rather vary according to some distribution. To test the performance of our algorithms, it is essential to draw samples of the run-times of jobs from a particular distribution; but first we need to determine some realistic distributions that we can use in our simulations. Therefore, we repeatedly executed some parallel and sequential programs, gathered run-time statistics, and analyzed them.

We performed several experiments using the NAS Benchmarks [3]. These benchmarks were used to determine the types of run-time distributions that may be typical for at least some jobs on some machines. We needed to determine sample parameters for these run-time distributions so that they could be reproduced by our simulator. While performing our tests, we controlled the following environmental characteristics: server location, network and server load, number of processors, amount of memory, and processor speed. Table 1 summarizes the configurations of our machines `caesar` and `elvis` upon which we ran our experiments.

	`caesar`	`elvis`
Type SGI	Challenge L	Onyx
Proc Speed (MHz)	200	150
Proc Type (MIPS)	R4400	R4400
# of Processors	4	4
Memory (Mbytes)	64	192
Secondary Unified Cache	4 Mb	1 Mb

Table 1: Configuration of SGI machines `caesar` and `elvis`, both running IRIX64 v6.2.

The jobs that we used throughout these experiments were from two sources: NASA's reference implementation for some of the NAS Benchmarks, and our own implementations of other NAS Benchmarks that met the required criteria. Four of the experiments use some version of the NAS Integer Sort (IS) Benchmark, implemented either in parallel on four processors, or in single processor mode. Two other experiments used the NAS Embarrassingly Parallel (EP) Benchmark run on a single processor. We now explain our experiments and their results.

3.1.1 Integer Sort, Executed on Four Processors

This experiment examined the run-time distribution of a version of the NAS Integer Sort Benchmark executed on four processors. We implemented the integer sort using a counting sort [5, pages 175–178] algorithm. We used Silicon Graphic's light weight process (thread) support functions, including `mfork()`, to implement our version of this benchmark.

We ran this sort across a heavily loaded network, obtaining both the executable and the data from a file server that was also heavily loaded. When run on `caesar`, the run-time distribution, for 100 executions, appears Gaussian.[2] Figure 1 shows a histogram of this distribution. When run on `elvis`, the run-time distribution, again for 100 executions, appears exponential and is shown in Figure 2. We note that the origin of the exponential distribution shown in Figure 2 is translated to approximately 3.0. That means that the sort had to run for *at least* 3.0 seconds before stopping. The distribution that we see very closely matches an exponential distribution with a mean of around 0.20, translated 3.0 seconds to the right. We expect that many jobs would have a distribution similar to this, because all jobs must run at least some amount of time[3].

In these experiments, we also see that memory size, and so, the need to swap to local disk, can have a definite effect upon the run-time distribution of a job. The integer sort on `elvis` completes, on average, 30% sooner than the same job on `caesar`. We note that, in this case, the amount of memory has more influence

[2] The form of the distributions were determined by carefully selecting the bin size and then curve fitting. The authors are familar with both visual and analytical tests for normality, but analytical tests were not used given the strong visual similarity of the frequency plots to that of a Normal curve. (The fact that some sample point frequencies lie above and below the selected Normal distribution is due to the number of samples being finite. Such phenomena would have appeared even if 100 data points had been sampled from a known Normal run-time distribution.)

[3] An exponential distribution is defined to start at 0.0. If applied, without translation, in this case, that would mean there is a strong possibility of near-zero run-times.

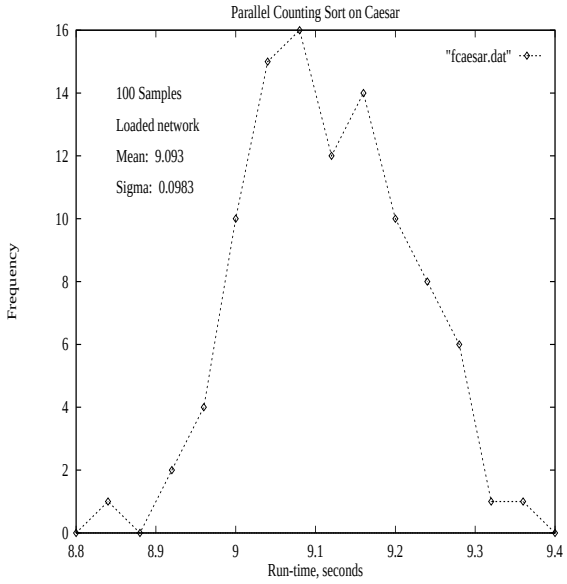

Figure 1: Forked counting sort, `caesar`.

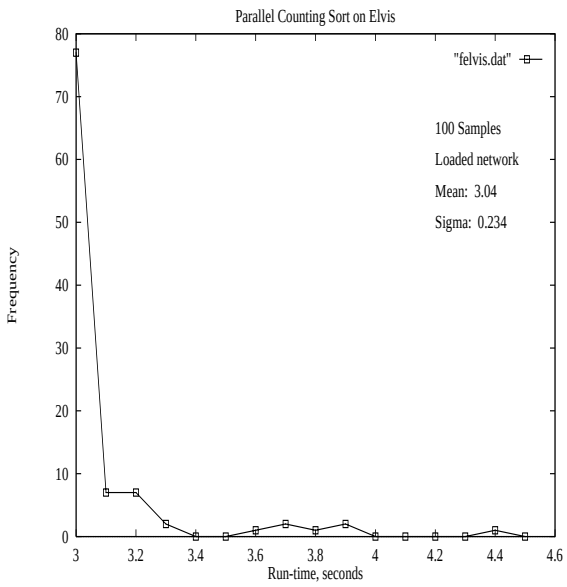

Figure 2: Forked counting sort, `elvis`.

on the run-time of the job than does the speed of the processor. Of primary importance, however, is the observation indicating that the same job, running on two different machines, not only has different mean run-times, but the distribution of run-times is different, yielding a Gaussian-like distribution on one machine and an exponential-like distribution on the other.

3.1.2 Integer Sort, Single Processor

This experiment is the same as that discussed in the last section, with the exception of being run on a single processor instead of being distributed across four processors. Although a slightly different C++ implementation was used, we again based our program on the counting sort.

When the integer sort was run on `caesar` and `elvis`, the run-time distribution was not easily characterized; however, it appears related to a Gaussian distribution. Histograms of the distributions, similar to that shown in Figure 4, are possibly multimodal, which indicates that multiple distributions may be present. While this experiment does not provide us with definitive results, it does point to the fact that run-time distributions can be quite complex. We suspect that these conditions are related to changes in the network and server loads.

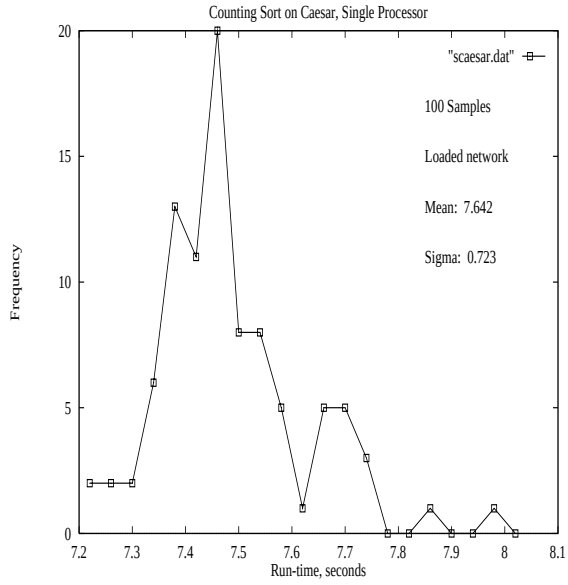

Figure 3: Counting sort, `caesar`, single processor.

Once again, this set of experiments showed us that additional memory can greatly enhance run-time performance. The tests on `elvis` ran 7 times faster than those run on `caesar`, which has the faster processors. The tests also show that run-time distributions can be very complex, and may be difficult to reproduce in a simulation. Although our simulations did not use such complex distributions, they should be modeled in future work.

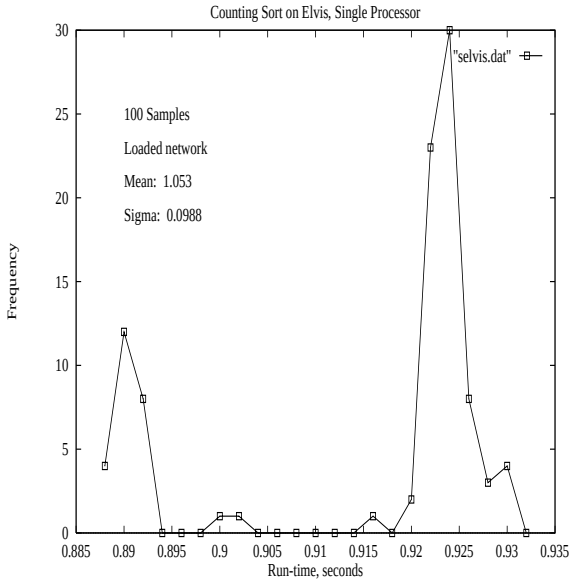

Figure 4: Counting sort, `elvis`, single processor.

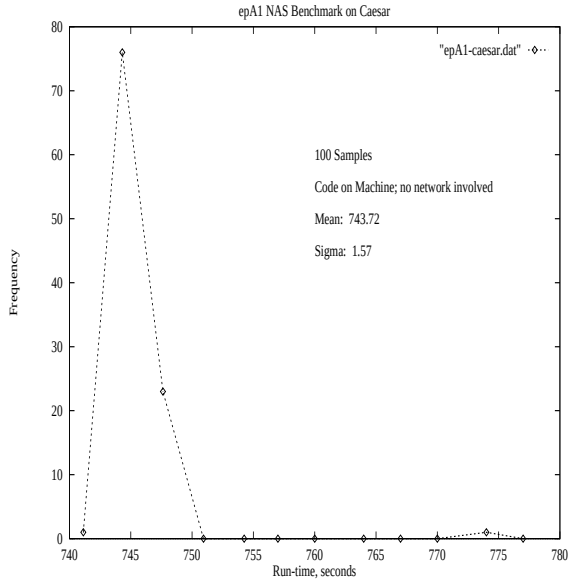

Figure 5: epA1 NAS Benchmark, with executable residing on local disk.

3.1.3 Embarrassingly Parallel NAS Benchmark

The next set of experiments that we describe compared the run-time distributions of compute intensive jobs run from local disk to those run across the network from a file server. The tests that we describe in this section were executed only on `caesar` because `elvis` did not have a sufficiently large local disk available. We used the reference implementation [3], from NASA, of the NAS Embarrassingly Parallel (EP) Benchmark. This implementation uses the portable Message Passing Interface (MPI) [12] to parallelize the code. The tests we ran, however, were compiled to be executed on a single processor[4]. The EP Benchmark was run 100 times for each test. See Figures 5 and 6.

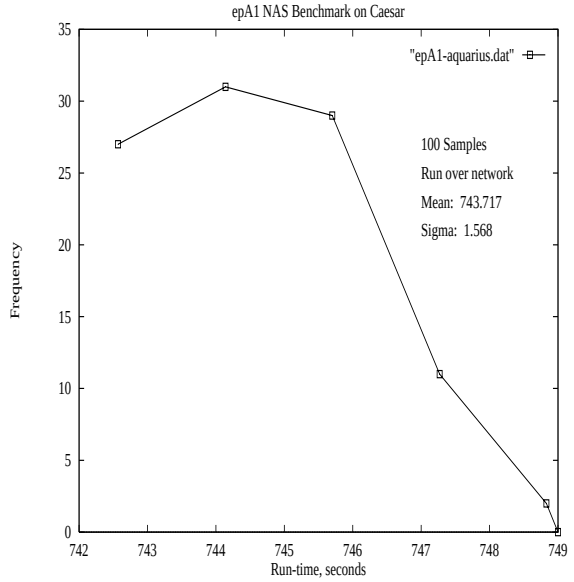

Figure 6: epA1 NAS Benchmark, files obtained over a lightly loaded network.

3.2 Simulation Experiments

We now describe our simulation experiments that are aimed at examining how well the mapping algorithms performed when the jobs scheduled did not execute for exactly the mean run-time. The matrices that we refer to in the description below have rows indexed by the job and columns indexed by the machine.

- Matrix Format. We used different matrices containing jobs and machines of varying characteristics. Each matrix contained mean run-times for each of five different jobs on each of ten different machines. The average means of the corresponding columns and rows were the same for all matrices and the jobs themselves were quite heterogeneous.

- Job Request Sets. In order to obtain different results for each matrix, we generated two random sequences of 125 job requests, which we will call `125-1` and `125-2`, where each individual request was chosen according to a uniform random distribution from among five different jobs. We also generated two more ran-

[4] The MPI mechanism is still utilized in the EP Benchmark when it is compiled for a single processor.

dom sets, this time of 500 job requests, calling them `500-3` and `500-4`. We did this to look at performance variations between job request orderings, as well as to examine any performance differences that might occur because fewer or more jobs were requested.

- Job Request Format. We generated each of the 5 jobs, for each request, at random. Thus, in these experiments, the jobs were requested in random order. This was done because the order of job request affects the schedule. The Fast Greedy Algorithm maps and schedules the jobs on machines in the order in which they are submitted. The Greedy Algorithm uses the order to break ties. We chose to execute these randomly ordered requests both because they more closely mimic a real environment where different jobs are submitted by different users and because we wished to examine whether these algorithms performed better or worse when unsorted, as opposed to sorted, requests were submitted.

- Run-time Generation for Simulations. We executed each simulation 15 times. In each run, a different value was used to seed the random number generator that was used to generate the simulated "actual" run-time duration. The total time required to execute each schedule was summed and the average was computed. Multiple seeds were used to ensure that our results were not skewed[5].

- Baseline Calculations. In addition to simulations where we generated simulated run-times from particular distributions, we performed some **baseline calculations**. These baseline calculations provided results that were, in effect, equivalent to running the simulation where the run-time of a job on a given machine was always exactly its expected run-time.

- Actual Run-time Distributions. When we generated run-times that were different from the mean predicted run-times, we ran experiments for both Gaussian and exponential distributions. Based upon our experiments with the NAS IS and EP Benchmarks above, we chose to implement a translated exponential distribution.

  Again, based upon our earlier experiments described in Section 3.1, we chose to use a truncated Gaussian distribution in our simulation experiments to mimic the Gamma distribution that best fit our data. We chose to truncate left of the mean at $\mu - \sigma$.

3.3 Results of Simulation Experiments where Jobs Ran for Times Different from the Predicted Run-times

This set of experiments examined the performance of intelligent mapping algorithms when job run-times differed from the expected run-times that were used to develop the mappings. Using the distributions identified in the previous experiments, we instantiated specific parameters in order to simulate some typical jobs. We simulated jobs with both exponential and truncated Gaussian run-time distributions. In this paper we summarize results; individual results from additional individual experiments, which are consistent with the conclusions that we make in this paper, can be found in Armstrong's thesis [2].

The graphs in this section compare the final completion times of the jobs under the various mappings. We use the label **Baseline** to mean that the value represented would be the completion time if all of the jobs ran for exactly their predicted mean run-times. In order to emphasize the differences between the values that we plot in the graph, we do not include the OLB run-times. The OLB run-times, for the exponential and Gaussian distribution simulations that we discuss below, averaged around 10,000 seconds in all cases shown, i.e., 500 requests.

3.3.1 Exponential Distribution Experiments

The results of these experiments compare the performance of the various mapping algorithms when all jobs have an exponential run-time distribution. We recall that the sample run-times from those experiments closely fit a shifted exponential distribution with mean of 3.0.

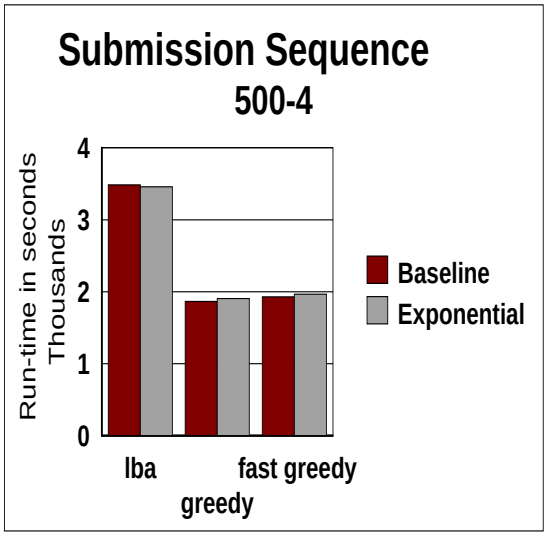

Figure 7: Exponential run-time distribution results, 500-4.

[5] This is a common method to reduce the influence of a single random number generation sequence that may be biased.

We now compare the time at which the last job finishes if executed according to each of the mappings, assuming that a job is not started on a machine until the last job completes. The figures in this section show both the expected completion time assuming deterministic run-times as well as under the assumption that the run-times are exponentially distributed, shifted to the right such that its mean matches the expected run-time.

Figure 7 shows these comparisons for some matrices that we used in our simulations. This figure shows that the schedules built by the intelligent mapping algorithms are still effective even though the actual run-time of a given job on a given machine can differ greatly from its expected run-time.

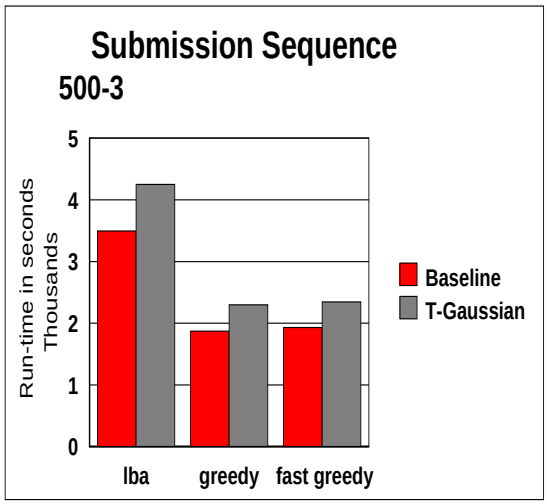

Figure 8: Truncated Gaussian run-time distribution results, 500-4.

3.3.2 Truncated Gaussian Experiments

We then performed additional simulations to examine the performance of the the intelligent mapping algorithms when all jobs had approximately Gamma run-time distributions. We determined from our experiments that we could approximate such a distribution by truncating a Gaussian distribution to the left of the mean at roughly $\mu - \sigma$. Throughout this experiment, the mean, μ, was the expected run-time for the individual job/machine pair, and σ^2 was set to 300% of μ. Therefore, these experiments are useful in determining whether, when the variance is very large for all jobs, the greedy algorithms still performed much better than both the LBA and OLB algorithms. No negative run-times were generated in our experiments because the truncation value was always positive.

The results in Figure 8 show that the schedules are finishing up to 25% later than in the previous experiments. This not unexpected, as truncation will shift the mean of the resulting distribution to the right. In the next section we provide a theoretical discussion as to why we would expect the times to be at least 20% later. The results also show that the greedy algorithms still perform better than the OLB and LBA algorithms when job run-time distributions are truncated Gaussian with very large variances. Our experiments, and the theoretical explanation below, imply that it may be worthwhile to update the mapping as the jobs are being executed, to minimize the effect of the large job variances.

3.3.3 Theoretical Explanation for Longer Run-times shown in Gaussian Experiments

To theoretically predict the new mean of the truncated distribution described in the last section, we can use simple Gaussian statistics [1]. Without loss of generality, our explanation uses a standard Gaussian distribution with a mean of 0 and a standard deviation of 1. If $A(z_1)$ is the area under the distribution from the mean, $z = 0$, to $z = z_1$, then it can be easily shown that the new mean, μ_{new}, for our truncated distribution is

$$\mu_{new} = A^{-1}\left[\frac{.5 - A(1)}{2}\right] \qquad (1)$$

Using this, we see that the new mean should be $\mu_{new} = .20\sigma$.

Unfortunately, the truncation of the Gaussian distribution only accounts for a 20% increase in the mean. Therefore, this explanation alone leaves some 5% unaccounted for. The remaining 5% is due to two factors. The first can be traced to the fact that we are using a truncated Gaussian instead of a Gamma distribution. The second is the fact that the expected value of the maximum of several Gaussian distributions is not the maximum of the expected values. The application of this well-known probability result to quality of service metrics is documented elsewhere [9].

3.3.4 Comparison of the Two Greedy Algorithms

We note that in our results, presented both here and in Armstrong's thesis, the Greedy and Fast Greedy algorithms appeared to perform similarly. Over all of our experiments we only saw the Greedy Algorithm performing up to 15% better than the Fast Greedy Algorithm. Other work has suggested that the improvement should be much higher. However, the other work, to our knowledge, was based upon presenting *sorted* requests to these mapping algorithms. The theoretical explanation for these results is beyond the scope of this paper and is discussed in another paper [7].

4 Related Work

To our knowledge, no one else has studied the performance of intelligent heterogeneous mapping algorithms when the run-times of jobs are non-deterministic, by using the distributions of run-times for actual programs determined under different resource loadings.

Ibarra and Kim [8] were the first to study the performance of the algorithms upon which we concentrated. Their analytical study centered around determining the worst-case performance of the algorithms. Weissman [15] used simulation to study interference-based policies; that is, policies that take into account the fact that as you increase the load on any shared resource, the rate of execution of other jobs decreases. Our policies, and simulations, assumed that the jobs were executed on a first-come, first-served basis. Although we did not study their performance here, genetic algorithms have been proposed as a good way to schedule tasks on heterogeneous resources, particularly when communication or synchronization is needed between tasks [13], [14]. Many systems have followed the lead of SmartNet [6] in implementing intelligent schedulers, such as those we describe here, in their resource management systems [11], [4], [16].

5 Summary

In this paper, we experimented with several applications on resources with differing loads and fitted their run-times to distributions. We then used these distributions to determine via simulation whether, when the run-times are non-deterministic, it is still beneficial to use intelligent algorithms that make use of the expected run-times to compute a mapping. We found that it continues to be beneficial even when the expected run-time distributions have large variances. As the distributions in our simulations were derived from the execution of actual programs, our distributions are realistic. However, there are additional distributions that are also realistic that we have not yet examined. We intend to pursue these in future work.

References

[1] Alder, H. L., and Roessler, E. B. *Introduction to Probability and Statistics*, third ed. Freeman, London, England, 1964.

[2] Armstrong, R. K. Investigation of Effect of Different Run-time Distributions on SmartNet Performance. Master's thesis, U.S. Naval Postgraduate School, September 1997.

[3] Bailey, D., et al. The NAS Parallel Benchmarks 2.0. Tech. Rep. NAS-95-020, NASA Ames Research Center, December 1995.

[4] Beguelin, A., et al. *HeNCE: A User' Guide.* Oak Ridge National Laboratory and University of Tennessee, December 1992. The document itself is available on the web at cs.utk.edu.

[5] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[6] Freund, R., Kidd, T., Hensgen, D., and Moore, L. Smartnet: A Scheduling Framework for Heterogeneous Computing. *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks* (1996).

[7] Hensgen, D., Kidd, T., and Armstrong, R. Comparison of greedy algorithms for scheduling jobs in a heterogeneous environments. In progress.

[8] Ibarra, and Kim. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the ACM* (1977).

[9] Kidd, T., and Hensgen, D. Why the mean is inadequate for accurate scheduling decisions. In progress.

[10] Kidd, T., Hensgen, D., Freund, R., Kussow, M., and Campbell, M. Compute Characteristics: A Useful Characterization of Job Runtimes. In preparation for submission (1998).

[11] Neuman, B. C., and Rao, S. The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems. *Concurrency: Practice and Experience* (1994).

[12] Pacheco, P. A User's Guide to MPI. Tech. rep., Department of Mathematics, University of San Francisco, March 1995.

[13] Singh, H., and Youssef, A. Mapping and Scheduling Heterogeneous Task Graphs using Genetic Algorithms. *Proceedings of the Heterogeneous Computing Workshop* (1996).

[14] Wang, L., Siegel, H. J., and Roychowdhury, V. P. A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing Environments. *Proceedings of the Heterogeneous Computing Workshop* (1996).

[15] Weissman, J. B. The Interference Paradigm for Network Job Scheduling. *Proceedings of the Heterogeneous Computing Workshop* (1996).

[16] Zhou, Zheng, Wang, and Delisle. Utopia: A load sharing facility for lage heterogeneous distributed computer systems. *Software: Practice and Experience* (1993).

Biographies

Major Robert K. Armstrong is currently in charge of the Modeling and Simulation Laboratory for the Marine Corps Air Ground Combat Center, Twentynine Palms, California. He received his BS in Engineering from the United States Naval Academy in 1985, is a graduate of the Amphibious Warfare School in Quantico, Virginia, and has earned an MS in Computer Science from the Naval Postgraduate School, Monterey, California in 1997. Major Armstrong has served in the capacity of Artillery Officer with the 1st Marine Division in Korea, Somalia, and Kuwait. His interests include computer architecture, distributed systems, and modeling and simulation for training.

Debra Hensgen received her Ph.D. in Computer Science, in the area of Distributed Operating Systems from the University of Kentucky in 1989. She is currently an Associate Professor of Computer Science at the Naval Postgraduate School in Monterey, California. She moved to Monterey from the University of Cincinnati three years ago where she was first appointed as an Assistant Professor and then a tenured Associate Professor of Electrical and Computer Engineering. Her research interests include resource management and allocation systems and tools for concurrent programming. She has authored numerous papers in these areas. She is currently a Subject Area Editor for the Journal of Parallel and Distributed Computing and is the chief architect and a co-Principal Investigator for the DARPA-funded MSHN project which is part of DARPA's larger QUORUM program.

Taylor Kidd is an Associate Professor of Computer Science at the Naval Postgraduate School (NPS) in Monterey, California. He received his Ph.D. in Electrical and Computer Engineering from the University of California at San Diego (UCSD) in 1991. He received his MS and BS, also in Electrical and Computer Engineering, from UCSD in 1986 and 1985 respectively. Prior to accepting a position at the NPS, he was a researcher at the Navy's NRaD laboratory in San Diego, California. His current interests include distributed computing and the application of stochastic filtering and estimation theory to distributed systems. He is a co-Principal Investigator, along with Debra Hensgen, for the DARPA-funded MSHN project which is part of DARPA's larger QUORUM program.