Dynamic Resource Allocation of Computer Clusters with Probabilistic Workloads

Marwan Sleiman, Lester Lipsky, Robert Sheahan
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT 06269-2155
Email: {marwan, lester, roberts}@engr.uconn.edu

Abstract

Real-time resource scheduling is an important factor for improving the performance of cluster computing. In many distributed and parallel processing systems, particularly real-time systems, it is desirable and more efficient for jobs to finish as close to a target time as possible. This work models the execution time for such a stochastic environment and proposes a dynamic algorithm for optimizing the job completion times by dynamically allocating resources to jobs that are behind schedule and taking resources from jobs that are ahead of schedule. We validate our analytical model with simulations that represent the real computing environment. The results of our simulations show that our alternative is the best estimate to predict the time remaining by using earlier data. Emphasis is placed on where variance enters the system and how well it can be controlled. Also our dynamic algorithm involves modifying the architecture to help reduce the peak number of servers used to execute a job and thus optimize the computation cost.

I. Introduction

Being able to make accurate estimates of how long a job will take to finish in a distributed cluster computing environment is of primary interest in the performance community. In such an environment, multiple servers (also called processing elements or PEs) work on executing the *tasks* that make up a *job*. In many situations it is desirable to have a job complete at or near a specific time, called a *Target Time*. Target times are related to deadlines in that the average completion time should be at the target time but the average completion time should be a number of standard deviations before the deadline, the number of standard deviations determines the probability of missing the deadline. Deviation from the each side of the mean target time is undesirable - finishing too late is obviously undesirable because it misses the deadline, but finishing too early is also undesirable because the job wastes resources that could have helped another job finish within its deadline. Finishing far ahead of or behind a target time may also be undesirable if two jobs are linked and the first finisher considers the later one "failed" after too long a delay, even if the delay was large because the first finished so early. Jobs may have deadlines because they are part of a real-time application and results after the deadline are useless. Jobs may also have deadlines because they are part of a dependable system that will initiate failure recovery procedure or job reassignment if the deadline passes without results. In the later case, reducing the variance around the target time allows tighter timeout bounds and therefore faster response to failures.

When the task times of a job are known beforehand, an optimal schedule can be found, though finding it may be computationally complex. When the tasks times of a job are only known *probabilistically*, then there is no hope for an optimal solution, but one looks, instead, for schemes that will do well on average. There are many reasons why task times vary - the node processing a task may fail and restart, the node may slow down because it is sharing cycles with another process, the task may employ a randomized algorithm, or the task time may simply depend on the data. Regardless of why, task times vary in many situations and systems must be designed to perform well despite the variation. Building a dependable cost efficient computing environment will require achieving two constraints: optimizing the number of servers used to finish the job and making the job execution time as close as possible to the deadline by decreasing the variance. To achieve our goals, we developed an algorithm for *Dynamic Recourse Allocation* that we refer to as *DAA*.

The remaining of this paper is organized as follows: In section 2, we present a literature survey about previous efforts related to the topic and we explain our motivation. We build the analytical model and formulate the problem in section 3, then we present methodology and algorithm in section 4. In section 5, we talk about the simulation that represents the real world for our model. Then, in section 6, we discuss the simulation results and evaluation criteria of the algorithm. Finally, in section 7, we propose some topics for further investigation, and we conclude in section 8.

II. Background and Motivation

The multiprocessor load balancing problem was studied by several researchers but nobody gave a best estimate solution to the problem yet. For example, in [1], the authors discuss the possibility of load rebalancing by assuming that they know the

1-4244-0054-6/06/$20.00 ©2006 IEEE

size of existing jobs. But, because the computing environment is non-deterministic, this assumption is not very accurate because we can always have some jobs that are bigger than expected. Other researchers have used a genetic algorithm [2] & [3] for dynamic load-balancing [4]. But the problem with genetic algorithms is that they take a long time to converge so we do not recommend them for real time applications where we have jobs that consist of a small number of tasks. Also, the results of Monnier et al. [3] show that the GA algorithm shows slightly better results than the other clustering algorithms. The EDF algorithm was adopted as an optimal algorithm for meeting deadlines. However, this algorithm has its drawbacks in detecting deadline violations, as shown in [5]. Kwok and Ahmed [6] studied the static parallel scheduling by discussing the taxonomy of static parallel scheduling. Their focus on the static scheduling made their efforts partial because we will have a much better efficiency when we use the dynamic scheduling algorithm. This claim is supported by the work of Ramamritham et al. [10] who proposed different heuristics for solving the problem of dynamic scheduling and showed that the dynamic distributed scheduling improves the performance of real-time systems. Radulescu and Gemund [7] presented two low-cost approaches to compile-time list scheduling where the tasks' priorities are computed statically or dynamically for homogeneous systems. These two algorithms, FCP (Fast Critical Path) and FLB (Fast Load Balancing), have been shown to yield a performance equivalent to other algorithms with significantly higher costs, such as MCP and ETF (Earliest Task First). Amin shows in [8] modified versions that yield a good overall performance, which is generally comparable to algorithms specifically designed for heterogeneous systems, such as HEFT (Heterogeneous Earliest Finish Time) or ERT (these are versions of MCP and ETF, respectively, using the task's completion time as the task priority). Topcuoglu et al. [9] presented two scheduling algorithms; the Heterogeneous Earliest-Finish-Time (HEFT) algorithm and the Critical-Path-on-a-Processor (CPOP) algorithm, for a bounded number of heterogeneous processors with an objective to simultaneously meet high performance and fast scheduling time criteria. The HEFT algorithm selects the task with the highest upward rank value at each step and assigns the selected task to the processor, which minimizes its earliest finish time with an insertion-based approach. The CPOP algorithm uses the summation of upward and downward rank values for prioritizing tasks.

Most of the previous algorithms were not based on analytical models to represent the stochastic computing environment. Moreover, the CPU distribution times were shown to be Power-Tailed in [21] and [22] which implies a high variance in the execution time of each task. For a heavy tailed distribution, the average of the first k tasks is very likely to be less than the long-term mean even for sequential tasks. For this reason, we developed a Dynamic Resource Allocation Algorithm that uses schemes that relies on the past history rather than just the information from the current run.

III. PROBLEM DESCRIPTION AND FORMULATION

We consider a job made up of N independent tasks whose individual processing times are unknown but are taken from some distribution, F (x) = Pr($X \le x$), with mean time $\bar{x}$, and *Coefficient of variation*, $C_v^2 = \sigma^2 / \bar{x}^2$. Then, from any book on probability, if the tasks must be executed one at a time, the mean time for the job to be finished will be:

$$\bar{T} = N\bar{x},$$

with a variance of

$$\sigma_N^{\ 2} = N\sigma^2.$$

Since the individual task time is not known until the task is done, we can only say that about two-thirds of the time the job will finish within the range:

$$\frac{T}{N} \in \left[\bar{x} - \frac{\sigma}{\sqrt{N}}, \bar{x} + \frac{\sigma}{\sqrt{N}} \right]$$

This may not be a very useful estimate, but it's easy to calculate, and in any case it is the best we can do without more specific knowledge. Furthermore, it depends only on the mean and variance of the task-time distribution.

If the tasks can be executed in parallel, say P tasks at a time, then the problem complexity increases considerably. Let $T(N;P)$ be the random variable denoting the time it takes to process N tasks on P processors. Only when $P = N$ is there a general expression for the distribution of the job time, namely, from the theory of *Order Statistics* [24],

$$\Pr[T(N; N) \le x] = [F(x)]^N \qquad (1)$$

Obviously if all the tasks take exactly the same time (F (.) is the Deterministic Distribution), then the job time reduces to $\bar{x}$. For all other distributions the mean time until all are finished will be longer than that, usually much longer. With one other exception, a tedious integration must be performed. That exception is the exponential distribution. In this case, it is well known that

$$\mathrm{E}[T(N; N)] = \bar{x} H(N),$$

Where $H(N)$ is the harmonic sum,

$$H(N) = \sum_{l=1}^{N} \frac{1}{l} \Rightarrow \log(N) + \gamma;$$

Where γ is Euler's constant.

The variance of time is:

$$\sigma^2(N; N) = \bar{x} H_2(N) = \bar{x} \sum_{l=1}^{N} \frac{1}{l^2} < \bar{x}^2 \frac{\pi^2}{6}$$

Note that even though all N tasks start at the same time the last one to finish will take on the order of $\log(N)$ times longer than the mean. In other words, there will always be a straggler. It is also seen that most of the variance is contributed by the last few tasks.

If the number of tasks exceeds the number of processors, then the problem complexity gets much worse. Now, (1) is not even valid for the last P tasks because the last tasks all started at different times. In general the mean time to finish is very difficult to compute (see [13], and [21]). There are the usual two exceptions. If all tasks take exactly the same amount of time, then:

$$\mathrm{E}[T(N;P)] = \bar{x}\left\lceil \frac{N}{P} \right\rceil$$

While, if they are exponentially distributed it is:

$$\mathrm{E}[T(N;P)] = \bar{x}\,[\frac{N-P}{P} + H(P)] \qquad (2)$$

with a variance of:

$$\sigma^2(N,P) = \bar{x}^2\,[\frac{N-P}{P^2} + H_2(P)] \qquad (3)$$

The progress achieved by the system in completing the job can be marked by noting the time when the l^{th} task finishes. Let that be $T(l|N,P)$, then for exponentially distributed task times and for $l < N-P$ (number of remaining tasks is more than the number of servers P),

$$\mathrm{E}[T(l|N,P)] = \bar{x}\frac{l}{P} = \mathrm{E}[l-1|N,P] + \frac{\bar{x}}{P},$$

while, for $N-P \le l \le N$ (number of remaining tasks is less than the number of processors P),

$$\mathrm{E}[T(l|N,P)] = \mathrm{E}[l-1|N,P] + \frac{\bar{x}}{N+1-l} = \bar{x}\,[\frac{N-P}{P} + H(P) - H(N-l)]$$

where $\mathrm{E}[T(N|N,P)] = \mathrm{E}[T(N,P)]$. We see that the tasks finish at a steady rate until there are fewer than P tasks remaining, at which time the time between departures increases. The increase is only partly due to the fact that fewer tasks are available to use the resources. It is because longer tasks finish last.

These simple formulas are illustrative of what happens in all parallel and multitasking systems. For a period of time, tasks are completed at a steady rate, but then, as the job approaches its end, the time between task completions increases. More complicated systems, in addition, experience initial instability in which the first few tasks finish faster than the long-term average. Thus, the total job time can be broken up into three periods: the *start-up* or *transient*; *steady-state*, and the *draining* periods. For some useful model examples see [21]. Only if $N >> P$ will the simple estimate of $\mathrm{T}(N,P) \approx \bar{x}\,N/P$ approach the actual mean job time.

The accumulated variance is at least as complicated. Even if $\mathrm{E}[T(N,P)]$ can be calculated accurately, a particular job execution could be too far away from the target time for completion to be acceptable.

We finally get to the question posed in this paper: can the job-time variance (or some other object function) be reduced? It has often been proposed that the system hardware can be dynamically modified to slow down or speed up the execution of a job if it is running ahead or behind schedule.

But just what does “ahead of schedule" mean? Surely, observing that the actual time that a particular job took to finish its first l tasks was less than $\bar{x}\,l$/P is not at all a good estimate that it is ahead of schedule. The longer time for draining must be anticipated.

We see, then, that two things are necessary before a dynamic procedure can be useful. First, the hardware (or appropriate software) must be changeable quickly and efficiently; second, there must exist a model which accurately marks the time for the completion of each epoch (e.g., $\mathrm{E}[T(l|\mathrm{N,P})]$ and the l^{th} task completion in the simple case of exponential task times). Even if we can find these things, how much can we affect the performance? It was our goal to find out. Indeed, for our simple case, we found that the variance about the target time is smaller (by a factor of two or more compared with a static system), and does not get bigger with the number of tasks. But what if the model is *not* an exact predictor of the actual system? We explored this by assuming that the model was correct in form, but the task mean task time is not known. We then estimate the mean time using the time actually taken by the tasks that have finished, but based on the model's expectations. We found that there was no real difference with the case where the mean time is known beforehand. We leave for future studies the situation where the form of the model is not quite right (e.g., the task time distributions are inaccurate).

IV. Methodology and Algorithm

We examine here the simplest of systems, yet what we find is applicable to more complicated systems, and even to other object functions (other than variance). Consider a (large) cluster of processors which are dynamically available on demand. On this cluster, we run a *job* made up of N independent *tasks* whose demands for various resources are only known probabilistically. That is, the distribution of task demands (when averaged over many tasks) is known, but the demand by each task is not known until that task is finished. Our goal is to select a configuration of hardware resources [18] (e.g., CPU's, local discs, communication channels, centralized discs for shared data) so as to meet a target as closely as possible (not too early, and not too late) and reserve the minimum number of processors. Because the tasks can run in parallel (except, when sharing a resource), when a task finishes, the system configuration can be modified if the job is

ahead or behind schedule. Determining which particular task is ahead or behind schedule is itself a complex problem because the task time is probabilistic. What makes the problem more complicated is that the variance of the average time is not the same during all phases of the execution time of the job. Since the tasks are executed in parallel, the average of the first k tasks is very likely to be less than the long-term mean, their difference decreasing as 1/k [13]. The reason for this is that the first few tasks to finish are likely to be much less than the mean, and new tasks may actually finish before some of those that started before them. That is, one can only consider those tasks that have already finished. Our approach is a best case scenario for meeting the job deadline as close as possible while maintaining a minimal average number of used processors.

When one task finishes, it leaves the system and is replaced by another task in the queue, until all *N* tasks finish.

The first step in our Dynamic Allocation Algorithm (*DAA*) is to determine just how many resources will be needed (e.g., the number of processors that are needed if each processor is used exclusively by one task) for the whole job to finish just on time. This is done by giving $\bar{x}$ an initial (constant) value, x_0, in Equation 2. At the time a task finishes, called an *epoch*, we compare the actual time from the simulation with that calculated by Equation 2, and if the job is not on schedule we change *P* so that the tasks remaining can still finish on time. If we find that the job is ahead of schedule, we decrease *P* by one; if we find that it is late, we increase *P*.

Our goal is to see just how much control can be maintained in such an uncertain environment. In particular, what is the variation of finishing time from job to job?

As a means for getting some insight as to just what kind of and how much data we should look at to achieve our goals, we have done an extensive study of the simplest system within our framework. Here we have *P* processors and no peripherals, there is an infinite pool of processors that can be allocated and de-allocated instantaneously without cost, the number of allocated processors does not alter the time it takes any task to run [23], and all the service times are exponentially distributed. Clearly this system cannot exist [11], we use it only to determine the best case bounds. In the Future Work section, we discuss more complicated systems, but the rest of this paper uses the simple best case just described. For this system, the mean time for *N* tasks to complete can be written down directly, without any detailed calculations.

V. Description of Simulation

Now that we have a good analytical model representing our problem and an algorithm to resolve it, we are ready to write our simulations to verify that what we mentioned is accurate and is a representation of the real computing environment.

The *DAA* algorithm will be evaluated by comparing it with a Static Allocation Algorithm that uses a constant number of parallel processors for executing the tasks. The static algorithm is referred to as *SAA*.

The simulation we wrote is an event driven simulation where each event corresponds to an *epoch*; it consists of a program with several components. One component, *timeLeft*, takes as input the number of processors in use and the number of tasks remaining and returns an estimate of the time to finish. This component is used by the *procNeeded* component which takes the time remaining before target time, and current number of processors and returns the new number of processors.

The logic of *procNeeded* is roughly as follows (logic for limiting number of processors to a specified upper bound, and passing the distribution and system configuration information is not shown here for readability):

```
int ProcNeeded(oldProcCount, tasksLeft,
                 timeLeft)
newProc=oldProcCount-1

WHILE (targetTime <
          timeLeft(newProc,tasksLeft)
          && (newProc < tasksLeft +1))
    newProc++
ENDWHILE

RETURN newProc

Void oneRun(numProcs,numTasks)

//Pick target time for Task & avg Proc

targetTime=timeLeft(numProcs,numTasks)
newProc=oldProc=numProcs
timeNow=0

//Load the first processors with tasks

FOR (Task=0;Task < numProcs;Task++)

 targetTime=timeLeft(numProcs ,numTasks )
 push_heap(taskList(Task))

NEXT Task

//process until all tasks are finished

DO WHILE (heap_size() > 0)

  timeNow+=heap_pop
  newProc=procNeeded(oldProc, numTasks
     Task, targetTime-timeNow)
  FOR (i=oldProc; i <= newProc; i++)
    push_heap(timeNow+taskList(Task ))
    Task++
  NEXT
  oldProc=newProc

LOOP
```

The body of *timeLeft* is the heart of our analytic model. For the simple case we focus on here, it reduces to Equation (2). Another component is the random number generator, *taskList*. It produces individual task times based on the task time distribution, or can be overridden to produce a specific sequence (and/or log all values produced) for program validation. The logic of this function is also not central to understanding the simulation, but it is important to note that it

uses a 48 bit random number generator to prevent problems with the granularity of lower resolution random number generators. The simulator can be thought of as a heap structure that stores events such that the upcoming event most proximal in time is the top of the heap. A variable *timeNow* keeps track of where we are in the run. As with *procNeeded*, *oneRun* is simplified here to enhance readability. The logic for collecting measurements and passing additional information used for systems more complex than the one studied here has been removed.

VI. SIMULATION RESULTS

We have carried out an exhaustive set of simulation calculations over several parameters. Since the results are consistent with each other we present only a few here and describe their connection with the others.

We use early observations and order statistics to estimate the moments of the underlying distribution - we remarked earlier that order statistics make estimating the mean difficult when parallel processors are involved. To test our algorithm in this case, we run our simulations on both cases where the mean time is known and unknown. In the case where we assume that the mean time is unknown, we let the simulation determine the time by taking the average of the previous elapsed *epochs*. For the case where we assumed the time known, we took $\bar{x}=1$ in Equation 2, while for the second case,

$\bar{x} = \sum_i \frac{x_i}{n}$, where n is the number of elapsed *epochs.*

In both cases, we found that the plots of the response time and variance behaves in the same way and almost overlap. We also remarked that increasing the number of processors at each epoch by more than one has only a slight improvement on the remaining time regardless of the number of tasks, so we stick to increasing the number of processors by one. Clearly the limitation on reducing the number of processors by only one makes sense because you don't want to discard the work of a processor with a task in progress.

The accompanying figures are for simulations of jobs with $N = 64$ tasks and a system that always starts with $P = 4$. So the target time is $T_T = T(64|4;64) = 205/12 = 17.0833$ time units, where the initial average time per task is 1.0 time units. We then ran the simulation 200,000 times, keeping a running average of the mean time left to target time after each task finishes as well as the variance.

Obviously P can never exceed the number of tasks remaining, and therefore our algorithm (and any other algorithm that can only change the number of processors running in parallel) becomes ineffective at that point. This is what we previously called the *draining region*. There are several things to be noted about this zone.

The mean draining time is given by $H(P)$, and the variance is $H_2(P)$. Since $H_2(P) < \pi^2/6 = 1{:}6449$, we see that for the SAA algorithm, 30% of the variance of completion times is generated in this region. It would be less (more) if N were greater (smaller). For DAA, the influence of the draining zone is even more significant. We show this in the next three figures.

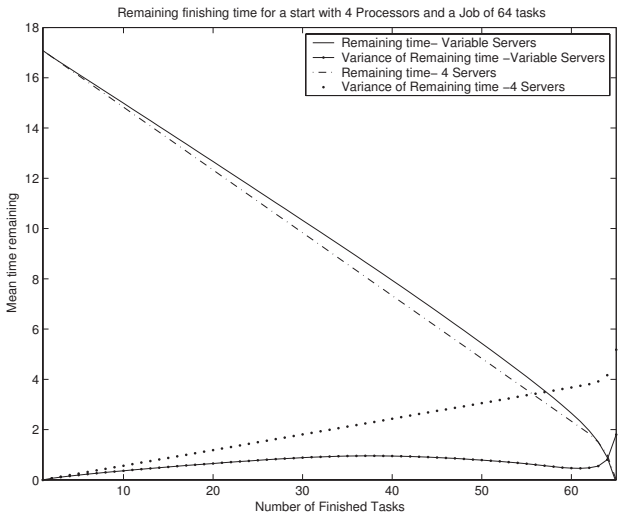

Figure 1. Time Remaining vs. Tasks Completed with variance, for both statically and dynamically allocated number of servers. The curves come averaging 200,000 simulation runs.

In Figure 1 we plot the mean time to the target time for both SAA and DAA, as well as their running variances. Let n be the number of tasks that have already completed, then for SAA, the mean time remaining, Tr(n|P;N), and variance, σ^2 (n|P;N), are known to be:
for $0 < n < N - P$,

$$Tr(n|P;N) = T(P;N)-n/P = \frac{N-n+P}{P} + H(P),$$

for $N - P < n < N$;

$$Tr(n/P;N) = H(P) - H(N - n),$$

while

$$\sigma^2(n|P;N) = n/P^2 \quad \text{for } n < N < P.$$

The SAA curves are exactly the plots of these equations. The time-remaining curve for DAA stays slightly ahead of schedule (slightly above the corresponding curve for SAA) until the draining region (n = 60), after which, the 2 curves coincide. The variance curves are quite different. Not only is DAA much smaller, but it actually decreases after n = 40, reaching its minimum at the beginning of the draining region $\sigma^2_{DAA}(60) = 0.4586$, while $\sigma^2_{SAA}(60) = 3.7433$), after which it rises the same amount as SAA, to $\sigma^2_{DAA} = 1.7983$ versus $\sigma^2_{SAA}(60) = 5.1795$, almost a factor of 3 difference.

To see if what we observed in Figure 2 carries through for other values of N, we ran many other 200,000 simulations for different values of N. In Figure 2, we plot the distribution of the completion time for the last task for both SAA and DAA, with $N = 64$ and $N = 128$. This is the time from the start of the job until that task runs, so it is the total time for the job. Although it's not clear, none of the four curves are symmetric

about their peaks, they are not quite Gaussian shaped. But rather, they all have an exponential tail on the negative side. This is due to the large impact of the draining region, in fact, of the last task.

The two SAA curves look very similar, differing by a factor of less than two because of the different variances as given by Equation (3). The striking result here is that the two DAA curves are virtually identical, even though they have distinctly different target times (17.0833 versus 33.0833). In fact, their variances differ by only 4% (1.7983 versus 1.8821). This is quite extraordinary, and implies that the variance of the completion time of a job made up of N tasks, executed under DAA, is independent of N as long as the target time is given by Equation (2) with fixed P. The simulations also show that the variance of the execution time for our algorithm decreases as the job approaches the completion time and this is independent of the number of tasks per job.

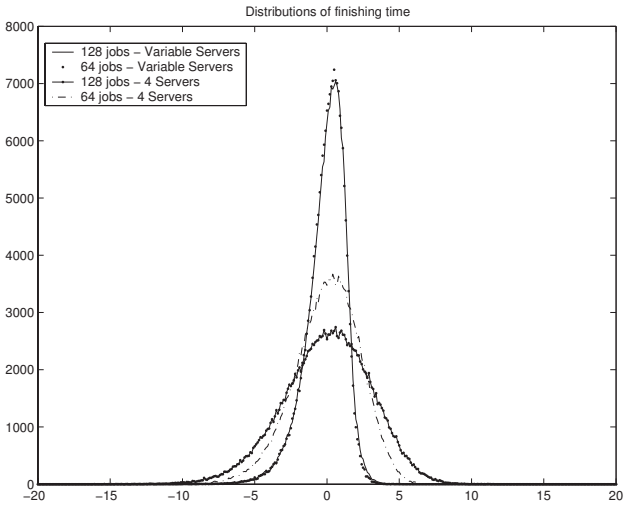

Figure 2. Distribution of completion time for the last task for both statically and dynamically allocated number of servers, with number of tasks = 64 and 128. For dynamically allocated servers, the plots align almost perfectly.

To show that the asymmetry of the curves in Figure 2 is due to the draining region, in Figure 3 we plot the distribution of the completion time for the fourth from the last task, i.e., just before the draining period begins, and therefore the last one we can speed up by adding processors. All four curves are indeed symmetric, with variances that are smaller than those for the total time by almost the same amount of $H_2(4)$. In other words, all four cases behave the same in the draining region. These results demonstrate that the analytical model represents the simulation.

VII. FUTURE WORK

This paper is part of a larger work examining parallel systems with both analytical and simulation techniques.

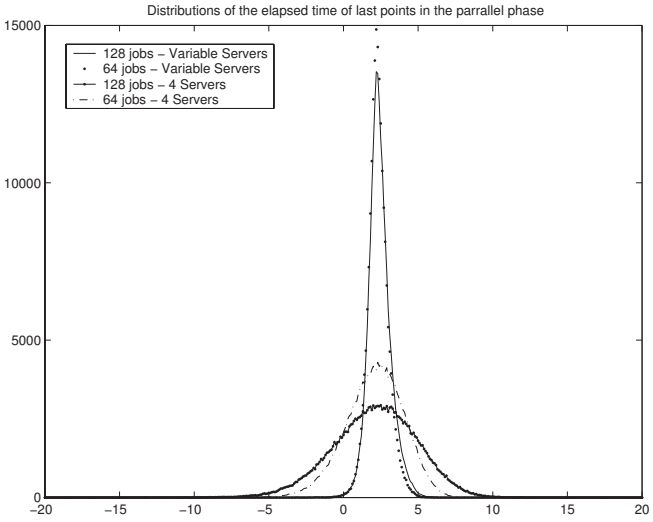

Figure 3. Distribution of completion time for the fourth from last task for both statically and dynamically allocated number of processors, with number of tasks = 64 and 128.

There are many topics we are either already investigating or hope to investigate soon. Some of the topics are given below: Incorporating an allocation/de-allocation cost function into the allocation decision module - as we mentioned in the introduction, using zero cost allows us to find a best case bound, but for real systems there will be some cost and we would like to know how it changes the behavior of the system. The cost could be in the form of a time delay or some other metric, possibly with a similar metric for deviation from target time so a minimal cost could be found.

Incorporating resource contention - this research started as part of a larger work involving processor allocation in parallel systems with resource contention and will continue to grow in that direction. This is an area with rich potential that has been known about for years but not fully investigated [17].

We want to experiment with a different average number of servers -our experiments have shown our technique controls the variance regardless of the average number of processors, but we don't yet know if the differences in variance are significant. We intend to run a large number of trials with different average number of processors and different task count/average processors ratios and see how consistent our results are across these conditions. Imposing a limit would allow a system using the DAA to schedule multiple jobs simultaneously using existing processor allocation algorithms as in [19] with only slight modification.

How much thrashing is there in processor allocation? Where in the job's life cycle does variation in the number of processors come? We would like to know how likely it is that a job that gets extra processors early in the run then has to release extras later or that a job that releases early has to acquire extras later. This behavior would be influenced by the allocation cost function described above, and might be a large concern in a real system. We could measure this by taking (at each task completion) the ratio of the number of changes to the peak number of processors and either the variance of the number of processors or the cumulative number of changes in

number of processors. A high number in any of these metrics might be cause to include a damping function of some type.

VIII. CONCLUSION

We have examined the effect of varying the number of servers executing a job that has a specific target time and is made up of *N* iid (independent identically distributed) tasks whose individual demands are not known. Our algorithm involves calculating the expected time remaining after each epoch (task completion), and increasing or decreasing the number of processors if speedup or slowdown is suggested. We have found that, for both exponentially and non-exponentially distributed task times where the target time can usually be met with relatively few processors (N/P >> 1), the variance of the completion time about the target is much smaller than for a system where the number of processors is kept constant. Furthermore, we found the surprising result that this variance does not depend on the length of the job (*N*), even though the variance grows linearly for systems with a constant number of processors. The variance starts very small during the transient state, then it increases, then it decreases if we apply our dynamic algorithm, and finally it increases dramatically during the draining period when we have fewer tasks than we have processors. These results hold for non-exponential distributions, for multi-processor systems with server contention, and for systems where the *expected remaining time* is itself an approximation. The techniques presented here can improve real-time and time-critical systems by reducing the chance of missing deadlines, dependable systems by both allowing tighter timeout bounds for detecting failures and compensating for load sharing based slowdowns, and shared processor pool systems by using no more resources than are actually needed.

REFERENCES

[1] G Aggarwal, R. Motwani, The "Load Rebalancing Problem", Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures. Pages: 258 - 265. Year of Publication: 2003

[2] The Genetic Algorithms Archive, http://www.aic.nrl.navy.mil/galist/

[3] Y. Monnier, J.-P. Beauvais, and A.-M. Deplanche, "A genetic algorithm for scheduling tasks in a real-time distributed system," presented at Proceedings 24th EUROMICRO Conference, Vasteras, Sweden, 1998.

[4] A. Page and T Naughton**, "**Observations on Using Genetic Algorithms for Dynamic Load-Balancing", IEEE Transactions on Parallel and Distributed Systems Pages: 899 – 911. Year of Publication: 2001

[5] F. Golatowski, J. Hildebrandt, J. Blumenthal, and D. Timmermann, "Framework for Validation, Test and Analysis of Real-Time Scheduling Algorithms and Scheduler Implementations"**,** 13th IEEE International Workshop on Rapid System Prototyping (RSP'02). July 01 - 03, 2002**.**

[6] Y.-K. Kwok and I. Ahmad, "Benchmarking the task graph scheduling algorithms," presented at Proceedings of the International Parallel Processing Symposium, IPPS, Orlando, FL, USA, 1998.

[7] A. Radulescu and A. J. C. van Gemund, "Fast and effective task scheduling in heterogeneous systems," presented at Proceedings 9th Heterogeneous Computing Workshop (HCW 2000), Cancun, Mexico, 2000.

[8] Alaa Amin, Ph.D. Dissertation. University of Connecticut, Storrs CT. 12/31/04

[9] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260-74, 2002.

[10] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Transactions on Computers*, vol. 38, pp. 1110-23, 1989.

[11] Bharadwaj, X Li, CC Ko, "On the influence of start-up costs in scheduling divisible loads on bus networks", *IEEE Transactions on Parallel and Distributed Systems*, 2000

[12] Mark Crovella, Lester Lipsky, Pierre Fiorini, "Consequence of Ignoring Self-Similar Data Traffic In Communications Modeling", Tenth International Conference on Parallel and Distributed Computing (PDCS-97), New Orleans, LA, (October 1997)

[13] L. Lipsky, T. Zhang, and S. Kang, "On the Performance of Parallel Computers: Order Statistics and Amdahls Law", 22nd International Conference for the Resource Management and Performance Evaluation of Computing Systems (CMG96), December 1996

[14] G Manimaran, CSR Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems", *IEEE Transactions on Parallel and Distributed Systems*, 1998

[15] C McCann, J Zahorjan, "Processor allocation policies for message-passing parallel computers", Proceedings of the ACM SIGMETRICS Conference, 1994

[16] Ahmed M. Mohamed, Lester Lipsky, Reda Ammar, "Modelling Parallel and Distributed Systems With Finite Workloads", Journal of Performance Evaluation, October 2004.

[17] VGJ Peris, MS Squillante, VK Naik, "Analysis of the impact of memory in distributed parallel processing systems", ACM SIGMETRICS Performance Evaluation Review Volume 22, Issue 1 (May 1994)

[18] E Rosti, G Serazzi, E Smirni, MS Squillante, "The impact of IO on program behavior and parallel scheduling", Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer, 56 - 65, 1998

[19] DD Sharma, DK Pradhan, "Job scheduling in mesh multicomputers", *IEEE Transactions on Parallel and Distributed Systems*, Volume 9 , Issue 1, 57 - 70 (January 1998)

[20] Gehan Weeransinghe, Lester Lipsky, Imad Antonios, "A Generalized Analytic Performance Model Of Distributed Systems That Perform N Tasks Using P Fault-Prone Processors", FTPDS-02, Fort Lauderdale, Fla, April 2002.

[21] Lester Lipsky, "Queueing Theory - A Linear Algebraic Approach, Maxwell Macmillan International publishing group, 1992.

[22] M. Greiner, M. Jobmann, and L. Lipsky, "The importance of power-tail distributions for modeling queueing systems." Operations Research, 47(2), 1999.

[23] JC Jacob, SY Lee, "Task spreading and shrinking on multiprocessor system and networks of workstations", *IEEE Transactions on Parallel and Distributed Systems*, 1999

[24] Herbert A. David and H. N. Nagaraja, "Order Statistics", Wiley-Interscience; 3 edition (July 25, 2003)