Scheduling Fork-Join Computations on Distributed-Memory Multiprocessor Systems

Khalid H. Sheta, Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
email: sheta@cis.ohio-state.edu, singhal@cis.ohio-state.edu
Phone: 614-292-5839
Fax: 614-292-2991
and
Dr. Phillip E. Krueger
Sequent Computer Systems
15450 SW Koll Parkway
Beaverton, OR 97006

Abstract

In this paper, we develop two analytic models for scheduling fork-join computations on distributed-memory multiprocessor system. The first model allows for arriving tasks to multiplex processors with existing tasks, while the other model does not allow multiplexing. Both models assume that tasks block for I/O and that the overhead of rescheduling a task to another processor is prohibitive. We compare the performance of both models in terms of the overall job completion time. We show that multiplexing have a performance advantage over non-multiplexing over a wide range of conditions. We point out how the results can be used in the design of an adaptive multiprocessor scheduler.

Key words: distributed-memory multiprocessors, job scheduling, fork-join computations, performance analysis.

1 Introduction

Over the past several years, multiprocessor systems have become increasingly important. Many studies have dealt with scheduling concurrent computations on shared-memory multiprocessor systems [2, 7, 6] where a task execution can be transferred between processors with relatively little overhead. Due to their scalability, multiprocessors having distributed-memory architectures are particularly promising for large multiprocessors. In this paper, we address the problem of scheduling independent fork-join computations on distributed-memory multiprocessors, where the overhead of rescheduling a task to another processor is prohibitive. In particular we focus on fork-join jobs having tasks that block for I/O. Examples of such jobs include parallelized relational database operations, such as those proposed by Dewitt, et al.[1] and Qadah [3]. In the remainder of this paper we will use the word job to refer to a fork-join job of the type described above.

When a task blocks for I/O, it may become advantageous to multiprogram, or 'multiplex' the processor, by running a task from another job on that processor. The goal of this study is to determine whether a performance advantage can be gained by multiplexing processors and if so, to identify the conditions under which it occurs. We develop analytic models to predict the expected job completion time on systems that multiplex processors and on systems that do not.

We begin by presenting our models for multiplexing and non-multiplexing systems. We continue by comparing the results from these two models. Finally, we summarize our results and present conclusions.

2 The Model

In this section we present the two models we use to compare the expected completion time of parallel computations having a fork-join structure. A job is considered done when the last task belonging to that job finishes. First we present the model that examines the expected job completion time if tasks from an arriving job are allowed to share or 'multiplex' processors with tasks already scheduled and running. We denote this model as MUX. Next, we present a model in which each task executes alone on a single processor. We denote that model as NONMUX. We assume that once a task has been assigned to a processor, it remains at that processor until its completion.

A job is assumed to be composed of n tasks, with the service demand of each task assumed to be drawn independently from an exponential distribution with parameter μ. We represent task blocking by a factor B. We assume that blocking time is evenly spread across a task's execution time, so that if a task executes alone on a single processor, then for any given period of time, the portion of that time the task is blocked is $B/(B+1)$ and the portion that it executes is $1/(B+1)$. We assume that when two tasks are scheduled on the same processor, they share it using the Processor Sharing [4] discipline (PS).

Because our goal is to determine whether multiplexing can be advantageous under any circumstances, we consider the simplest case in which an arriving job finds a single job already executing. We assume that neither job has more tasks than there are processors in the system. Consequently, no processor is multiplexed between more than two tasks. If multiplexing processors can be advantageous, it will be advantageous in this case.

When a job arrives into the system, it finds a job composed of i tasks already executing that it must compete with. The blocking factor for the arriving job is B_a, while that of the job that is already executing is B_e. As a performance index to compare the two schemes, we use the overall expected job completion time which is the average of the expected completion time of the arriving job and that of the existing job.

2.1 MUX

First, we find the expected job completion time of the arriving job when some of its tasks multiplex processors with tasks from another job that is currently executing. Then, we find the expected completion time of the existing job which must multiplex processors with the arriving job.

We assume that the number of tasks in the arriving job is the same as the number of processors in the system, and denote both by n. This is done to filter out the effect of the tasks belonging to the arriving job multiplexing with other tasks from the same job (in case the number of tasks is greater than the number of processors) and the case where multiplexing would not be needed (if the number of tasks is smaller than the number of processors).

We begin by finding the expected completion time of the arriving job when it starts execution at a time when the system has i tasks already scheduled to i processors out of the total n processors. We denote that as $\mathrm{E}(Y_{(n)}, i)$. $Y_{(n)}$ is the n^{th} order statistics [8] of n distributions where the j^{th} distribution $j = 1, 2, ..., n$ represents the completion time of the j^{th} task of the arriving job. To find $\mathrm{E}(Y_{(n)}, i)$, we need to find the distribution of the completion time of the j^{th} task of the arriving job. We have two distributions for the task completion time depending on whether the arriving task executes alone or if shares the processor with another task already in the system. If the task executes alone the, distribution function is an exponential distribution function $F(y) = 1 - e^{-\mu/(B_a+1)\ y}$. If the task is processor sharing with another task already in the system, we show in the appendix that the distribution function $F_1(y)$ has the following form:

$$F_1(y) = \ C\ (1 - e^{-\mu_1 y})\ +\ A(-\mu_2 y e^{-\mu_2 y} + 1 - e^{-\mu_2 y})$$

where

$$A = (B_a + 1)/(B_a + B_e + 2)\ \ ,$$

$$C = (B_e + 1)/(B_a + B_e + 2)$$

$$\mu_1 = \mu/(B_a B_e/(B_a + B_e + 2 + 1))\ \ ,$$

$$\mu_2 = 2\mu \ / \ [\ ((B_a+1)(B_a+2B_e+3)+1) \ / \ (B_a+B_e+2) \ ]$$

Define $F_{(Y_{(n)},i)}$ as the distribution function of the n^th order statistic $Y_{(n)}$, when an arriving job containing n tasks is co-scheduled with an existing job containing i tasks, in a system having n processors. At most two tasks (one from the arriving job and one from the existing job) share a processor, Then:

$$F_{(Y_{(n)},i)} = \ F_1^i F^{n-i} \tag{1}$$

The density function $g_{(Y_{(n)},i)}$ can be obtained by:

$$g_{(Y_{(n)},i)} = \ d/dy \ F_{(Y_{(n)},i)} \tag{2}$$

and hence the expected value of the arriving job completion time when it is multiplexed with an existing job of i tasks is:

$$E(Y_{(n)},i) = \int_{-\infty}^{\infty} y g_{(Y_{(n)},i)} \ dy \tag{3}$$

The integration in (3) can be carried out numerically for given values of μ, i, B_a, and B_e.[1]

Now, we find the expected completion time of the existing job which contains i tasks. Each of its tasks must share its processor with a task from the arriving job. We denote that as $E(X_{(i)})$. $X_{(i)}$ is the i^{th} order statistic of i distributions where the j^{th} distribution $j = 1, 2, ..., i$ represents the completion time of the j^{th} task of the existing job. $E(X_{(i)})$ can be computed in the same way as $E(Y_{(n)}, i)$ if we exchange the identity of the arriving and the existing jobs. That is:

$$E(X_{(i)}) \ = \ E(Y_{(i)}, i)$$

but with the role of B_a and B_e reversed.

The overall job completion time when an arriving job of n tasks multiplexes processors with i tasks from another job is the average of $E(Y_{(i)}, i)$ and $E(X_{(i)})$.

2.2 NONMUX

In this section, we present the second scheduling scheme when a task executes alone on a single processor. First we find the expected job completion time of the arriving job, and then that of the existing job. Here a processor has at most one task running at any time. Our NONMUX is quite similar to the model described in [5], but we extend the model to

[1] If we assume that the probability of having i jobs initially running on i of n processors is binomial, $\Pr\{i$ jobs initially scheduled$\} = \binom{n}{i} / 2^n$, then (1) will have a closed form solution $F_{Y_{(n)}} = 1/2^n \ (F_1 + F)^n$ and (3) will be $E(Y_{(n)}) = \int_{-\infty}^{\infty} y g_{Y_{(n)}} \ dy$.

account for task blocking. All other assumptions are the same as those made for the MUX model.

For an arriving job of n tasks, define $T_{i,j}$ to be the expected completion time when j tasks of the arriving job still need to be executed, $j = 1, 2, ...$or n and i tasks of the existing job are yet to finish. Also define: $\mu_a = \mu/(B_a + 1)$ for the arriving job and $\mu_e = \mu/(B_e + 1)$ for the existing job. $T_{i,n}$ satisfies the following equations:

$$T_{0,j} \;=\; H_j/\mu_a \quad where \quad H_j = \sum_{k=1}^{j} 1/k$$

$$T_{i,j} \;=\; 1/[(j-i)\mu_a+(n-(j-i))\mu_e] \;+\; (j-i)T_{i-1,j-1}/n \;+\; (n-(j-i))T_{i-1,j}/n \quad , 0 < i <= j$$

Hence, the expected value of the arriving job completion time given, that i of its tasks multiplex processors with tasks from the existing job is $T_{i,n}$. This value can be obtained by first calculating $T_{0,j}$, $j = 1, 2, ..., n$, then calculating $T_(i, j)$, $j = 1, 2, ..., n$ for a given value of i.

Now, the expected job completion time for the existing job $T'_{(i)}$ is the same as the expected completion time of the last finishing task out of a total of i tasks. This is the i^{th} order statistic of i distributions having an exponential distribution function $F(y) = 1 - e^{-\mu_b y}$. This leads to :

$$T'_{(i)} \;=\; H_i/\mu_e \quad where \quad H_i = \sum_{k=1}^{i} 1/k$$

The overall expected job completion time is the average of $T_{i,n}$ and $T'_{(i)}$.

3 Results

In this section we present the results of our study comparing the behaviour of parallel computations under the MUX and NONMUX scheduling schemes. We examine the results from sets of experiments comparing the performance of the two scheduling schemes under varying system parameters. We've used the overall job completion time as the basic performance metric, and for comparative reasons we plot the overall completion time of both schemes against a different system parameter to show the sensitivity of the performance to that system parameter.

3.1 Main Comparison

We begin by comparing the performance of the two scheduling schemes as the number of tasks in the existing job (which the arriving job must compete with) is varied. For this comparison, we assume a system having 32 processors, arriving tasks having a blocking factor $B_a = 1$, and existing tasks having a blocking factor $B_e = 2$. Figure1 shows that the two schemes perform quite similarly when the number of existing tasks is small (less than 10), with NONMUX slightly better than MUX. This similarity is due to both schemes having the majority of arriving tasks running alone on processors. However, NONMUX has a small edge over MUX, since the likelihood of arriving tasks waiting for a an existing task to finish is relatively small. As the number of existing tasks increases, NONMUX performs poorly with respect to MUX.

This relatively poor performance occurs because the NONMUX scheme insists on allocating a processor exclusively for each task, and since we have a larger number of existing tasks, the expected time the arriving tasks must wait for processors to become available increases. Consequently, this longer waiting time increases the overall job completion time. MUX, on the other hand, schedules arriving tasks immediately to processors, therefore reducing the waiting time and achieving higher processor utilizations by sharing processors between arriving and existing tasks.

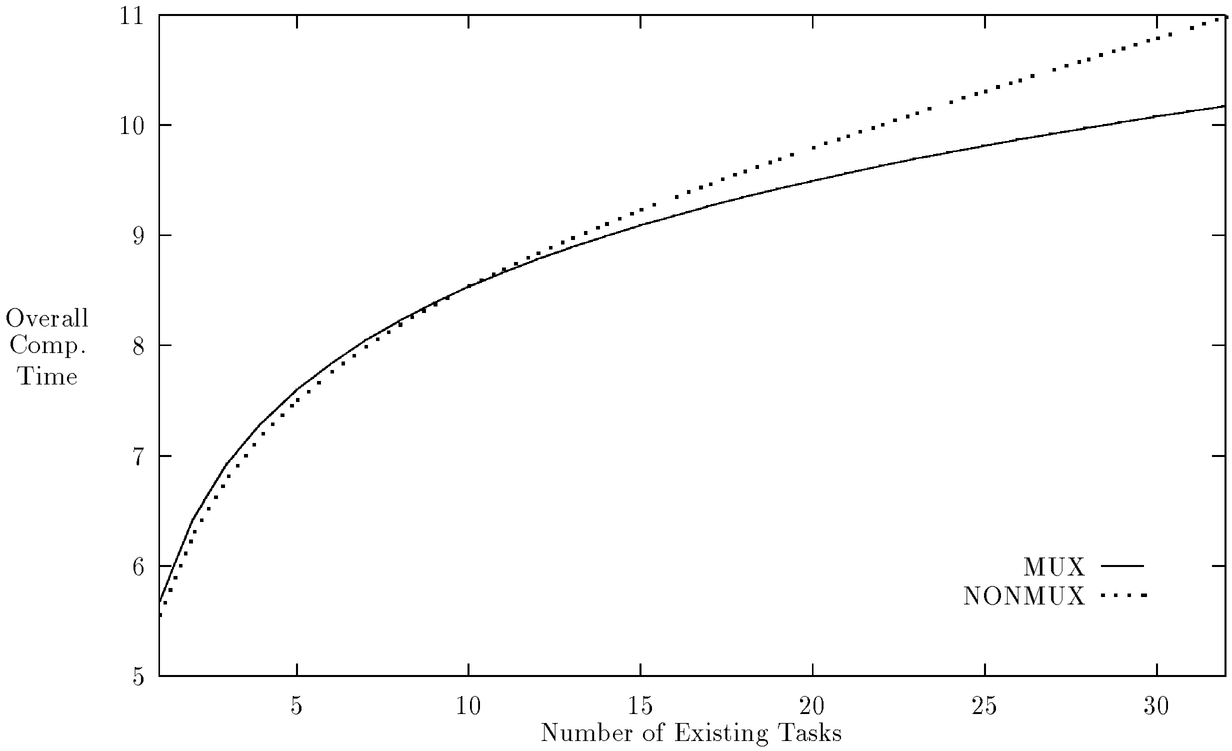

Figure 1: MUX, NONMUX for 32 processor system ($B_a = 1, B_e = 2$).

3.2 Sensitivity to n

We've conducted several experiments for various values of the number of processors in the system (n) to see how n affects the performance of the two schemes. Figure 2 shows the result for $n = 256$, which compares the two schemes in a system having a relatively large number of processors. It can be seen that the crossover point at which MUX starts performing better than NONMUX, is farther to the left where the advantage for NONMUX is even more slight. As system size increases, MUX becomes advantageous for almost all executing job sizes.

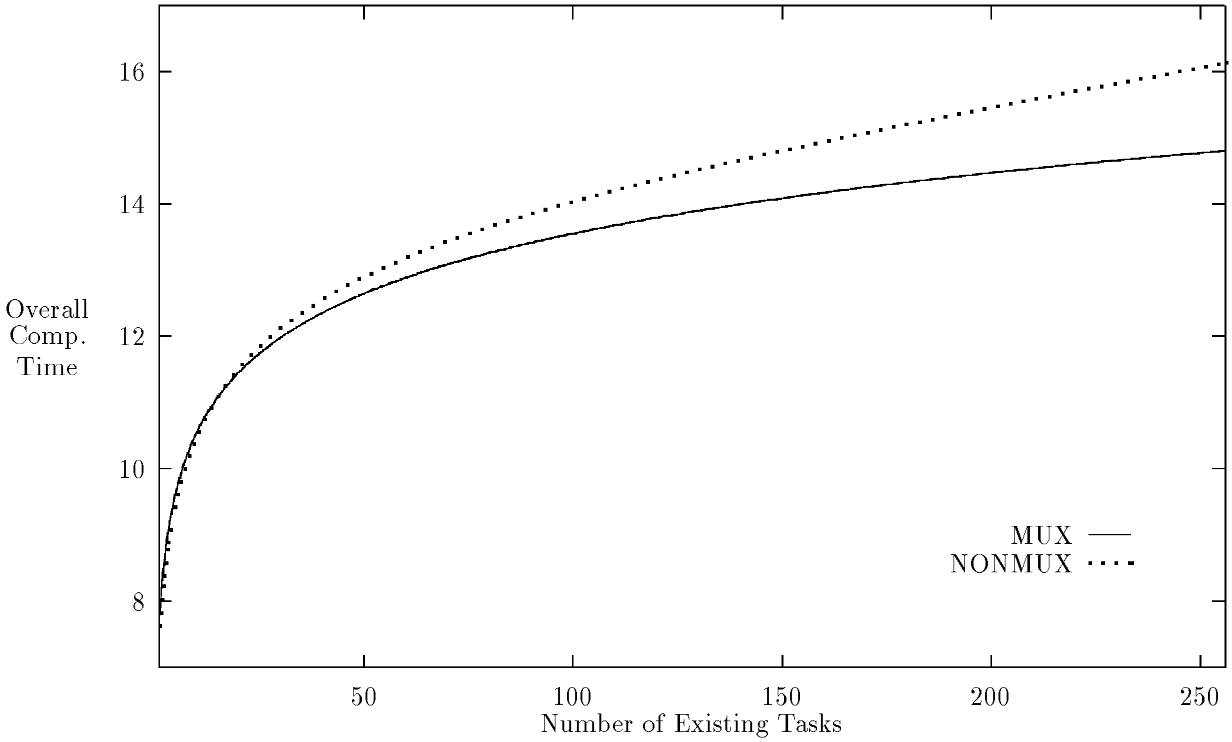

Figure 2: MUX, NONMUX for 256 processor system ($B_a = 1, B_e = 2$).

Figure 3 shows the crossover, at which MUX starts performing better than NONMUX, in terms of the number of existing tasks in the system for several values of n. Note that this point flattens out as n increases. In general MUX performs better than NONMUX as long as the number of existing tasks in the system is greater than 10 for $n >= 32$ and is greater than or equal to $n/2$ for $n <= 32$.

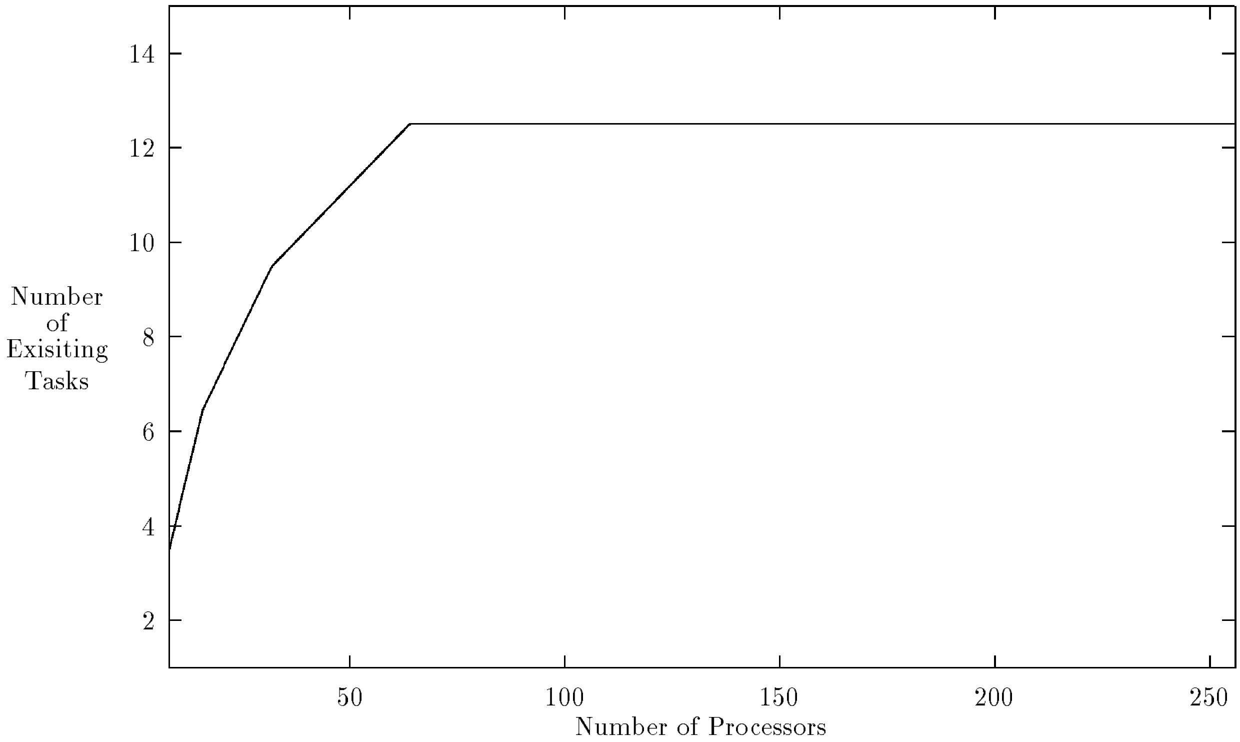

Figure 3: Cross Over of MUX, NONMUX ($B_a = 1, B_e = 2$).

3.3 Sensitivity to B_e

The main factor giving MUX an advantage over NONMUX is the use of processor idle time resulting from blocking by existing tasks. In this section, we show how the blocking factor B_e affects the overall response time. We do this comparison for a number of existing tasks of 16, 24, 32 tasks on our 32 processor system.

In Figures 4, 5, 6 we note that there is a crossover point between the two scheduling schemes, where NONMUX performs better than MUX in the B_e range to the left of the crossover point. This range shrinks as the number of existing tasks in the system increase. For example, at 24 existing tasks, equal blocking factors B_e and B_a result in an equal performance of both scheduling schemes. Again, this occurs because NONMUX has to wait longer for available processors as the number of existing tasks increase in the system. For the range of B_e to the right of the crossover point, MUX performs much better as the blocking factor B_e increases. A larger B_e allows more available processor time for the arriving job to make use of under MUX. In general, as both the number of existing tasks and B_e increase, the idle time of all processors will increase and MUX will utilize most of that time.

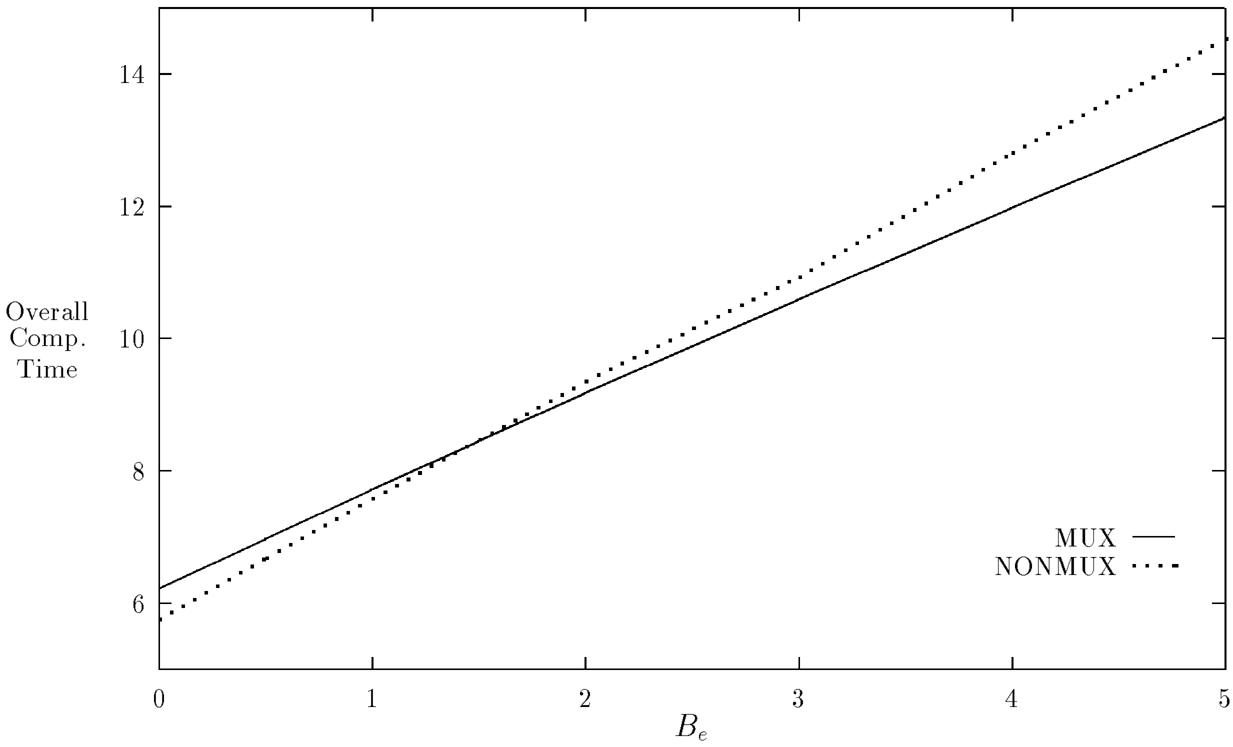

Figure 4: Effect of varying B_e ($n = 32, B_a = 1$, Number of Existing Tasks = 16).

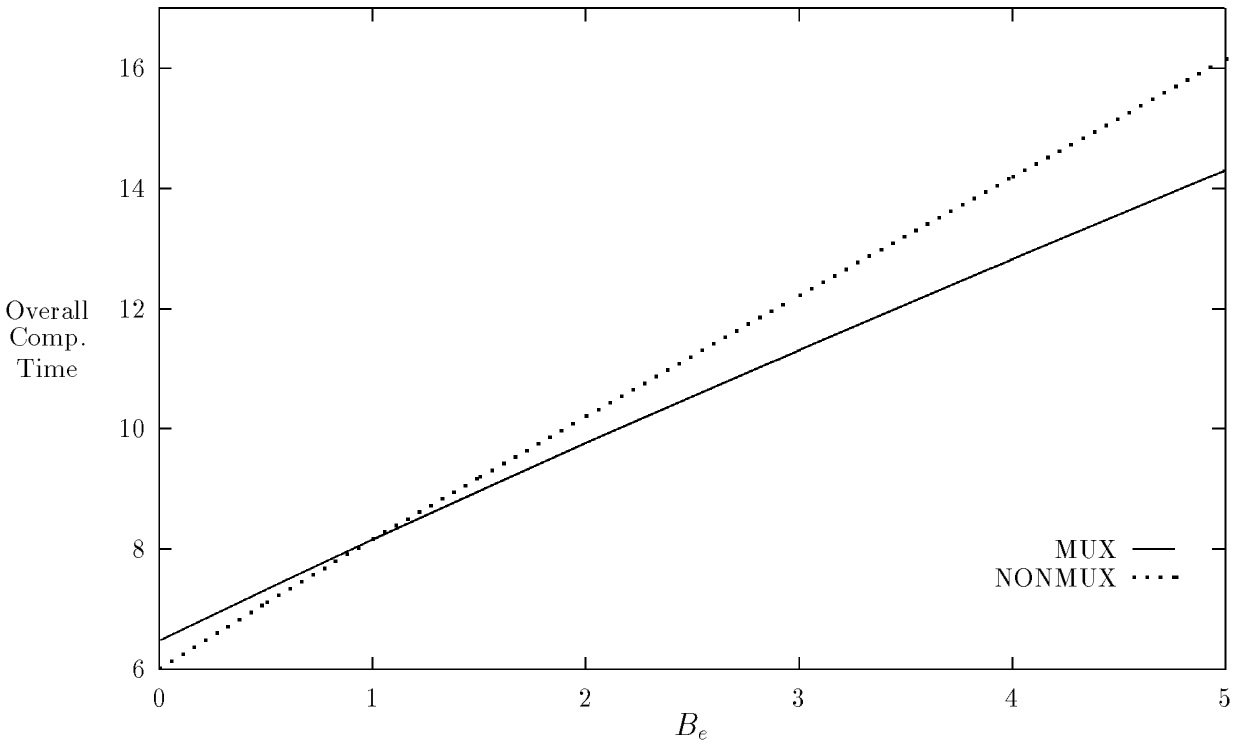

Figure 5: Effect of varying B_e ($n = 32, B_a = 1$, Number of Existing Tasks = 24).

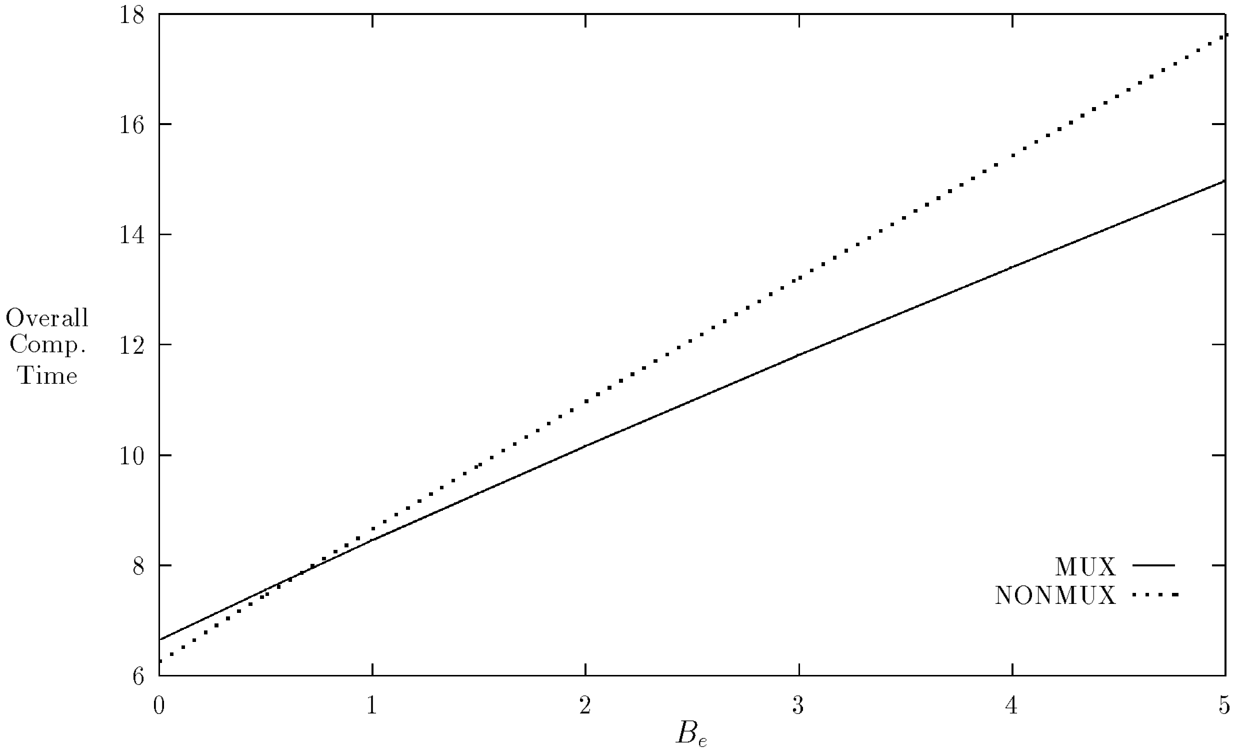

Figure 6: Effect of varying B_e ($n = 32, B_a = 1$, Number of Existing Tasks = 32).

3.4 Sensitivity to B_a

Figure 7 plots the completion time against B_a, assuming $n = 32$ and $B_e = 2$. We can see that performance is not sensitive to the value of B_a, with both MUX and NONMUX having a linear behaviour.

4 Conclusions

We have analyzed two models for scheduling fork-join computations on distributed-memory multiprocessor system. One model allows an arriving task to multiplex a processor with another existing task, while the second model does not allow multiplexing. We have derived an analytical expression for the overall job completion time for each model and used it to compare the two models. Based on this analysis, we have shown that multiplexing processors can provide a performance advantage, particularly when the number of tasks already executing is large and the task blocking is significant for the executing job. The advantage of multiplexing stems from its ability to make use of the idle processor time resulting from blocking of existing tasks.

These results can be used in the design of a multiprocessor scheduler which dynamically switches between multiplexing and nonmultiplexing schemes, depending on the current state of the system. For example, it may elect to not multiplex processors as long as task blocking is small or the number of executing tasks in the system is small relative to the number of processor. Then, it may choose to multiplex processors when the number of existing tasks and/or the amount of blocking time increases. The level of blocking by an executing task is easily measured; such measurements is provided by most operating systems.

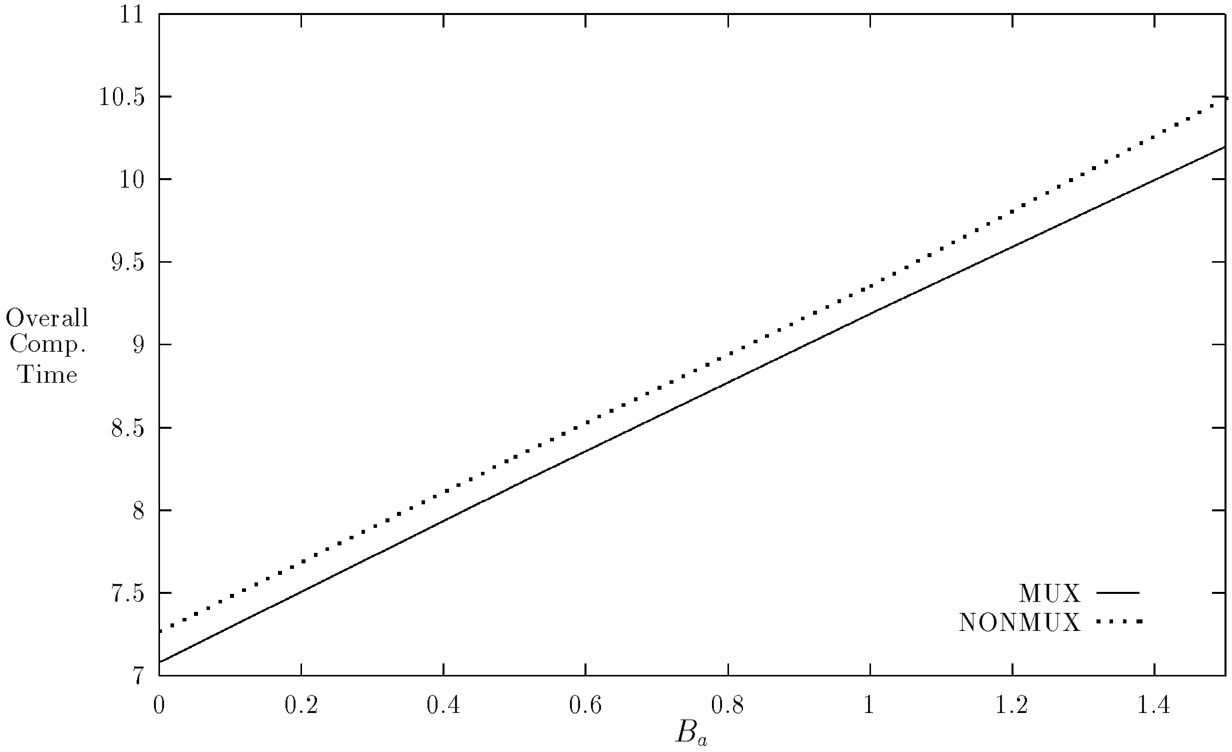

Figure 7: Effect of varying B_a ($n = 32, B_e = 2$, Number of Existing Tasks = 16).

Appendix

In this section we derive F_1, which is the distribution function for the completion time of a task in the arriving job when this task shares a processor with another task already executing in the system. We will refer to the task from the arriving job as the arriving task, and to the task it must share the processor as the existing task. We assume that blocking time is spread evenly across a task's execution time so that if a task executes alone on a single processor, then for any given period of time, a task blocks for $B/(B+1)$ and executes for $1/(B+1)$, where B is the blocking factor.

For two tasks that do not block, have an exponential service demand with density $f(x) = \mu e^{-\mu x}$, and share a single processor under the processor sharing discipline, then, using Laplace transform, the density function of the task completion time is: $F(s) = C\mu/(s+\mu)+A\mu^2/(s+\mu)^2$. The first term in this function corresponds to the case when the arriving task finishes first (with probability C and density function $\mu/(s+\mu)$ in the s domain). The second term corresponds to the case when the arriving task finishes last (with probability A and a density function $\mu_2/(s+\mu)^2$ in the s domain), representing the convolution in the time domain between the tasks service demands. Using the inverse transform, the density function of the arriving task completion time has the following form:

$$f(y) \;=\; C\mu e^{-\mu y} \;+\; A\mu^2 y e^{-\mu y}$$

where the first term produces the expected task completion time if the arriving task finishes first (case 1) and the second term produces the expected completion time if it finishes last (case 2).

When tasks block, the density function has the same form:

$$f(y) \;=\; C\mu_1 e^{-\mu_1 y} \;+\; A{\mu_2}^2 y e^{-\mu_2 y} \tag{1}$$

where the first and second terms have the same meaning as above. Our goal is to find the values for A, C, μ_1 and μ_2 for the blocking tasks case. We do this by finding C, A which are the probabilities of the arriving task finishing first (case 1), last (case 2) respectively. Then we find its expected completion time in case 1 and in case 2 to find μ_1 and μ_2.

Assume the service demand density function for an arriving task is $f(x_1) = \mu e^{-\mu x_1}$ and that of the existing task is $f(x_2) = \mu e^{-\mu x_2}$, with a blocking factor of B_a for the arriving task. Since the arriving task blocks for a total time of $B_a x_1$, hence the total time required by the arriving task is $x_1 + B_a x_1$. Similarly, the total time required by the existing task is $x_2 + B_e x_2$. We have two cases:

case 1: The arriving task finishes before the existing task, i.e. $(x_1 + B_a x_1) < (x_2 + B_e x_2)$.

case 2: The arriving task finishes after the existing task, i.e. $(x_1 + B_a x_1) > (x_2 + B_e x_2)$.

The probability of case 1 is:

$$\begin{aligned} Pr\{(x_1 + B_a x_1) < (x_2 + B_e x_2)\} &= \int_{x_2=0}^{\infty} \int_{x_1=0}^{\frac{(B_e+1)}{(B_a+1)}x_2} \mu e^{-\mu x_1}\; \mu e^{-\mu x_2}\; dx_1\, dx_2 \\ &= (B_e+1)/(B_a+B_e+2) \end{aligned} \tag{2}$$

and the probability of case 2 is:

$$\begin{aligned} Pr\{(x_1 + B_a x_1) > (x_2 + B_e x_2)\} &= 1 - Pr\{(x_1 + B_a x_1) < (x_2 + B_e x_2)\} \\ &= (B_a+1)/(B_a+B_e+2) \end{aligned} \tag{3}$$

For case 1, the completion time for the arriving task is: $x_1[1 \;+\; 1/(1+B_e)] \;+\; B_a x_1$ where the first term accounts for the service demand of the arriving task inflated by the execution of the existing task, and the second term accounts for the blocking time of the arriving task. Simplifying, the arriving task completion time is $[(1 + B_a) \; + \; 1/(1 + B_e)]x_1$ or simply bx_1 where $b = (1 + B_a) + 1/(1 + B_e)$. The expected completion time for an arriving task in case 1 is:

$$\begin{aligned} E(bX_1|case\ 1) &= \int_{x_2=0}^{\infty} \int_{x_1=0}^{\frac{(B_e+1)}{(B_a+1)}x_2} bx_1 \mu e^{-\mu x_1} \; \mu e^{-\mu x_2} \;\; dx_1 \; dx_2 \\ &= [(B_e + 1)/(B_a + B_e + 2)] \;\; [(b/\mu) \; (B_e + 1)/(B_a + B_e + 2)] \qquad (4) \end{aligned}$$

For case 2, the completion time for the arriving task is: $x_2(1 \; + \; 1/(1 + B_a)) \; + \; B_e x_2 \; + \; (x_1 + B_a x_1) - (x_2 + B_e x_2)$. The first two terms account for the period during which the two tasks compete (before the existing task finishes), with the first term accounting for the service demand of the existing task inflated by the execution of the arriving task and the second term is the blocking time of the existing task. The last two terms represent the execution of the arriving task after the existing task has completed. Simplifying, the completion time of the arriving task is: $(B_a + 1)x_1 + (1/(B_a + 1))x_1$ or $\alpha x_1 + \beta x_2$ where $\alpha = B + a$ and $\beta = 1/\alpha$. The expected value of the arriving task completion time in case 2 is:

$$\begin{aligned} E(\;(\alpha X_1 + \beta X_2)|case\ 2) &= \int_{x_1=0}^{\frac{B_a+1}{B_e+1}} \int_{x_2=0}^{\frac{B_a+1}{B_e+1}x_1} (\alpha x_1 + \beta x_2) \mu e^{-\mu x_1} \mu e^{-\mu x_2} \;\; dx_2 \; dx_1 \\ &= [(B_a + 1)/(B_a + B_e + 2)] \;\; (1/\mu) \;\; \times \\ & \quad [(\;(B_a + 1)(B_a + 2B_e + 3) + 1)/(B_a + B_e + 2)] \qquad (5) \end{aligned}$$

Hence, the expected completion time of the arriving task is equal to the sum of these conditional expected values in equations 4 and 5.

From equations 2, 3, 4 and 5, we can see that A, C, μ_1, mu_2 in equation 1 are:

$$A = (B_a + 1)/(B_a + B_e + 2) \;\; ,$$

$$C = (B_e + 1)/(B_a + B_e + 2)$$

$$\mu_1 = \mu/(B_a B_e/(B_a + B_e + 2 + 1)) \;\; ,$$

$$\mu_2 = 2\mu \; / \; [\; ((B_a + 1)(B_a + 2B_e + 3) + 1) \; / \; (B_a + B_2 + 2) \;]$$

Therefore, the distribution function of completion time of arriving tasks is:

$$\begin{aligned} F_1(y) &= \int_0^y f(y)dy \\ &= C \; (1 - e^{-\mu_1 y}) \;\; + \;\; A \; (-\mu_2 y e^{-\mu_2 y} + 1 - e^{-\mu_2 y}) \end{aligned}$$

$\square$

References

[1] D., Dewitt, et. al. "GAMMA - A high performance Dataflow Database Machine". In *Proc. 12th International Conf. on Very Large Databases*, 1986.

[2] D.Towsley, C.Rommel and J.Stankovic. "Analysis of fork-join program response times on multiprocessors". In *Trans. Parallel and Dist. Systems.*, volume Vol.1, No.3. IEEE, July 1990.

[3] G. Z. Qadah. "The Equi-Join Operation on a Multiprocessor Database Machine: Algorithms and the Evaluation of their Performance". In *Fourth International Workshop Grand bahamas Island*. Springer-Verlag, 1985.

[4] L. Kleinrock. *"Queueuing Systems, Volume I: Theory"*. Wiley, 1975.

[5] R.Nelson,D.Towsley, A.Tantawi. "Peformance Analysis of Parallel Processing Systems". In *Trans. Software Eng.*, volume Vol. 14, No.4. IEEE, April 1988.

[6] S. Majumdar, D.L.Eager and R.B.Bunt. "Scheduling in Multiprogrammed Parallel Systems". In *Proceedings of SIGMETRICS Conf. on Meas.& Modeling of Comp. Sys.* ACM, May 1988.

[7] S.T. Leutenegeger and M.K. Vernon. "The performance of Multiprogrammed Multiprocessor Scheduling Policies". In *Proceedings of SIGMETRICS Conf. on Meas. & Modeling of Comp. Sys.* ACM, May 1990.

[8] W.Mendenhall and R.Scheaffer. *"Mathematical Statistics with Applications"*. Duxbury Press, 1981.