Parallel Execution of a Sequence of Tasks on a Asynchronous Multiprocessor

G.M. Baudet,[+] R.P. Brent,[†] and H.T. Kung[*]

Given a sequence of tasks to be performed serially, a parallel algorithm is proposed to accelerate the execution of the tasks on an asynchronous multiprocessor by taking advantage of fluctuations in the execution times of individual tasks.

A parallel program requiring no critical section is given to implement the algorithm and its correctness is proved. A spacewise more efficient implementation which requires the use of critical sections is also given.

An analysis is presented (for both implementations) to estimate the speed-up achievable with the parallel algorithm. When the execution times are exponentially distributed, and no critical section is used, the algorithm with k processes yields a speed-up of order $k^{1/2}$.

Keywords and Phrases: Asynchronous machines, critical sections, multiprocessors, parallelism, queuing theory, redundancy, speed-up, synchronization, zero-finding.

CR Categories: 4.32, 5.25, 6.20

1. INTRODUCTION

We are interested in the design and analysis of parallel algorithms for asynchronous multiprocessors such as C.mmp (Wulf and Bell, 1972) and Cm* (Fuller et al, 1977). For any given task, the task execution time on such a system is dependent upon the properties of the operating system, effects of other users, processor-memory interference, and many other factors. As a result, it is often necessary to assume that task execution times are random variables rather than constants. The fluctuations in task execution times may be significant (Baudet, 1978). In this paper we propose a novel way of using asynchronous multiprocessors, to achieve a speed-up by taking advantage of fluctuations in task execution times. (The usual way of achieving a speed-up on multiprocessors is through the exploitation of inherent parallelism in tasks. The two approaches can, of course, be combined.)

To illustrate with an every-day example: suppose that a husband and wife go shopping together and find queues at both checkouts in a supermarket. They can minimise their expected waiting time by joining separate queues (and exchanging items when one of them reaches the head of a queue).

We shall present our result as a solution to the problem of execution of a sequence of n tasks $w_1, \ldots, w_n$ under the following conditions:

C1. For $i = 2, \ldots, n$, task w_i cannot be started before the completion of w_{i-1} (i.e., the tasks are linearly ordered).

C2. For $i = 1, \ldots, n$, no parallelism can be utilized in the execution of w_i (i.e., we are not allowed to decompose a task).

C3. The execution time of a task is a random variable rather than a constant. (This condition formalises the asynchronous nature of the multiprocessor.)

We view a parallel algorithm for asynchronous multiprocessors as a collection of asynchronous processes which communicate through the use of global variables. Such an algorithm is defined by giving the procedure each of its processes executes when assigned to a processor. *While analyzing the algorithm, we always assume that a processor is available for any of the runnable processes of the algorithm.* (See Kung [1976] for a general discussion of asynchronous parallel algorithms, and Kung and Song [1977], Robinson [1979] for examples of such algorithms.) In Section 2 we give an algorithm which uses $k \geqslant 1$ asynchronous processes to solve the problem. The algorithm is interesting because at most one process is doing useful work at any given time. Nevertheless, by taking advantage of condition C3, the mean execution time is less for $k > 1$ than for $k = 1$, i.e., a speed-up is achieved.

As an example, consider the computation of $x_1, \ldots, x_n$ defined by

$$x_{i+1} = \psi(x_i, \ldots, x_{i-d}),$$

where $x_0, x_{-1}, \ldots, x_{-d}$ are given and ψ is some iteration function. Let w_{i+1} be the task of computing $\psi(x_i, \ldots, x_{i-d})$. Our algorithm could be used to execute tasks $w_1, \ldots, w_n$, which is equivalent to evaluating $x_1, \ldots, x_n$. The application to root-finding and minimisation algorithms is obvious.

The speed-up ratio $S_k(n)$ of a parallel algorithm using k processes is defined in Section 3, and some preliminary results are proved there. In Section 4 we give

*+IRIA, 78150 Rocquencourt, Le Chesnay, France. †Department of Computer Science, The Australian National University, Canberra, A.C.T. 2600. *Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 15213, U.S.A.* *This research was supported in part by the National Science Foundation under Grant MCS 75-222-55, the Office of Naval Research under Contract N00014-76-C-0370, and a research grant from the Institut de Recherche d'Informatique et d'Automatique (IRIA), France. Most of the results were obtained while H.T. Kung was visiting the Australian National University under the ANU-CMU Computer Science exchange scheme. Manuscript received 8 June 1980.*

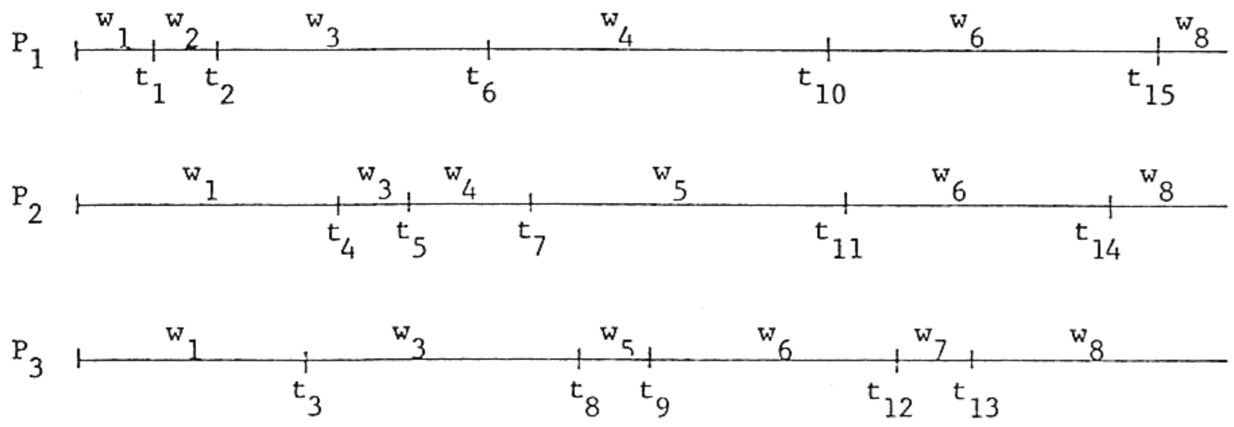

Figure 1. A possible task scheduling with three processes.

programs to implement our algorithm both with and without using critical sections and prove their correctness. In Section 5 we consider the implementation without critical sections, and obtain an analytic expression for the speed-up under certain assumptions (A1 and A2 of Section 5). For large n and k, our result is $S_k(n) \sim (2k/\pi)^{1/2}$.

In Section 6 we consider the implementation which uses critical sections. Here the analysis is more difficult, and we can obtain analytic results only for $k \leqslant 2$. Some conclusions and open problems are stated in Section 7.

Some caution is necessary when interpreting the results of this paper. Under conditions C1-C3 above we can obtain a real speedup in the expected execution time of a sequence of given tasks. We do not claim to increase the system throughput. In fact, since redundant computations are often performed, overall system throughput may be degraded. It might be necessary to prohibit our approach if it became too popular amongst the users of a system of asynchronous multiprocessors! For further discussion see Section 7.

2. THE ALGORITHM

For each positive integer k, we define an algorithm with k processes for executing tasks $w_1, \ldots, w_n$ under conditions C1 and C2 stated in the preceding section. The algorithm is specified as follows:

Whenever a process, P, is ready to execute a task,

(i) if no task has yet been completed by any process, process P starts executing task w_1,

(ii) otherwise, if the last task w_n has not yet been completed by any process, process P starts executing a task which is unfinished and ready for execution.

For simplicity, we shall assume that no two tasks are completed at the same time. Then due to the linear ordering of the tasks, (ii) defines without ambiguity a unique task to be executed by process P.

Let $t_1, t_2, t_3, \ldots$ with $t_i < t_{i+1}$ be the time instants of completions of tasks by the processes. The diagram in Figure 1 illustrates a possible scheduling of the tasks when they are executed by the algorithm with three processes.

Note that when process P_3 finishes task w_3 at time t_8, process P_2 has already completed w_4. Thus, after P_3 completes w_3, it starts executing w_5 rather than w_4. Task w_4 is skipped by P_3. Similarly, tasks w_5 and w_7 are skipped by P_1, and tasks w_2 and w_7 by P_2. After any one of the three processes has executed six tasks, tasks w_1 through w_8 rather than tasks w_1 through w_6 are completed. A speed-up has been achieved!

Observe that at any given time at most one process is doing work useful for later computation. With respect to the scheduling given by Figure 1, the time intervals on which processes are doing useful computations are indicated in Figure 2. Thus *the speed up is not achieved by sharing work among processes but is achieved by taking advantage of fluctuations in the execution times.*

3. A SPEED-UP MEASURE

Consider the algorithm with k processes as specified in the preceding section. The algorithm is said to be the sequential algorithm if k = 1 and to be a parallel algorithm if $k > 1$. Let

$T_k(n)$ = the time to execute tasks $w_1, w_2, \ldots, w_n$ by the algorithm with k processes.

Let $\bar{T}_k(n)$ be the mean of the random variable $T_k(n)$. We define the *speed-up ratio* of the parallel algorithm with k processes to be

$$S_k(n) = \frac{\bar{T}_1(n)}{\bar{T}_k(n)}$$

For each k and for each execution of the algorithm with k processes, we define $s_{k,i}$ to be the time instant of the first completion of task w_i, and define $s_{k,0} = 0$. For example, with respect to the scheduling of Figure 1, with k = 3, we have

$s_{3,1} = t_1,\ s_{3,2} = t_2,\ s_{3,3} = t_5,\ s_{3,4} = t_7,\ s_{3,5} = t_9,\ s_{3,6} = t_{12},\ s_{3,7} = t_{13}, \ldots$

The following theorem describes the relation between

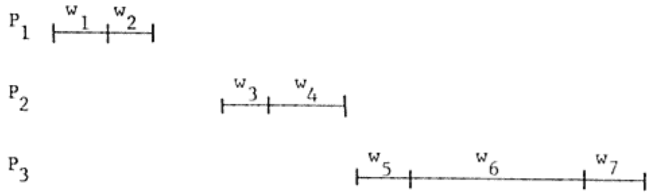

Figure 2. Time intervals on which processes are doing useful work.

$\{s_{k,i}\}$ and $\{t_l\}$ in terms of the scheduling of the tasks. This theorem is important in Sections 5 and 6 for computing speed-up ratios.

Theorem 3.1

Suppose that $s_{k,i} = t_l$ with $1 \leqslant i \leqslant n\text{-}1$. Then $s_{k,i+1} = t_{l+j}$ for some $1 \leqslant j \leqslant k$ if and only if

(a) the j processes completing tasks at times $t_l, t_{l+1}, \ldots, t_{l+j-1}$ are all distinct, and

(b) the process completing task w_{i+1} at time t_{l+j} is one of the j processes mentioned in (a).

Proof

(=>) Suppose that some process P completes two tasks at times t_{l+h} and t_{l+m} for $0 \leqslant h < m \leqslant j-1$. Then, since at time t_{l+h} task w_i has already been completed, the task completed at time t_{l+m} by process P must be w_{i+1}. This contradicts the fact that w_{i+1} is completed for the first time at time t_{l+j}, since $t_{l+m} < t_{l+j}$. This proves (a).

Let P be the process completing task w_{i+1}, for the first time, at time t_{l+j}. Suppose that P does not complete any task in the time interval $[t_l, t_{l+j-1}]$. Then the task completed by P at time t_{l+j} must be started before time t_l. But at any time before t_l, task w_i is not completed yet. Hence any task started before time t_l cannot be w_{i+1}. In particular, the task completed by P at time t_{l+j} cannot be w_{i+1}. This contradiction proves (b).

(<=) The proof is omitted, since it is similar to the one above.

For i = 1,2, . . . ,n, let $\tau_k(i)$ be the random variable representing the quantity $s_{k,i} - s_{k,i-1}$. Then since $T_k(n) = s_{k,n}$, we have

(3.1) $T_k(n) = \tau_k(1) + \tau_k(2) + \ldots + \tau_k(n)$.

(3.1) will be used later to compute $\overline{T}_k(n)$, which is needed for evaluating the speed-up ratio $S_k(n)$.

4. PARALLEL PROGRAMS FOR THE ALGORITHM AND THEIR CORRECTNESS

We give two programs to implement the algorithm with k processes: one without critical sections and one with critical sections.

4.1 A Program without Critical Sections

Program A:

```
global (integer or real) array U[1:n];
global Boolean array M[1:n+1];
Initialization: begin
                for m ← 1 to n+1 do M[m] ← false;
                start processes P_1, . . . ,P_k
                end
Process P_j:    begin integer m_j;
                m_j ← 1;
(4.1)           while M[m_j] do m_j ← m_j+1;
(4.2)           while m_j ⩽ n do
                     begin
(4.3)                ( perform task w_{m_j};                      )
(4.4)                ( write the output of task w_{m_j} on U[m_j]; )
(4.5)                M[m_j] ← true;
(4.6)                while M[m_j] do m_j ← m_j+1
                     end
                end
```

Assume that the tasks are not allowed to alter the array M and integers m_j. We shall prove that Program A is correct in the following sense:

P1. For m = 2, . . . ,n, task w_m is executed only if task w_{m-1} has been finished and its output has been written on U[m−1].

P2. For j = 1, . . . ,k, process P_j can execute the loops at (4.1), (4.2) and (4.6) at most n times.

P3. All the tasks $w_1, \ldots, w_m$ will have been completed at the time when any one of the processes $P_1, \ldots, P_k$ terminates its execution.

Property P2 guarantees that the program will terminate. (Note that there is no possibility of deadlocks in the program.) Property P1 ensures that the linear ordering requirement of the executions of the tasks is maintained, and property P3 implies that when the program terminates all the tasks are completed.

Lemma 4.1

(i) For m = 1, . . . ,n, if M[m] is set to **true**, it remains **true** afterwards.

(ii) After being initialized to **false**, M[n+1] is never modified.

Proof

After initialization, M can only be modified through statement (4.5) executed by some process P_j. But when entering the main while-loop (starting at (4.2)), m_j satisfies the condition $m_j \leqslant n$ and is not modified before the execution of (4.5). Therefore M[n+1] can never be modified.

Lemma 4.2

For j = 1, . . . ,k, if m_j has the value $m \geqslant 2$, then M[m−1] is **true**.

Proof

Suppose that $m_j = m$ with $m \geqslant 2$ at time t. If m_j was incremented by one to the value m inside the while statement (4.1) or (4.6), then the test of $M[m_j]$ being **true** with $m_j = m-1$ must have been satisfied. Hence M[m−1] was **true** at some time before t. Thus, by Lemma 4.1 M[m−1] is **true** at time t.

Lemma 4.3

For m = 2, . . . ,n, if M[m] is **true**, then M[m−1] is **true**.

Proof

Suppose that M[m] is **true**. Then M[m] must have been assigned to **true** through instruction (4.5) by some process P_j with m_j having the value m. Therefore, by Lemma 4.2, M[m−1] is **true.**

Lemma 4.4

For m = 1, . . . ,n, if M[m] is **true**, then task w_m is completed and its output is on U[m].

Proof

Suppose that M[m] is **true**. Then M[m] must have been assigned to **true** through instruction (4.5) by some process P_j with m_j having the value m. Since P_j executes instruction (4.5) only after the completion of task w_{m_j} and since m_j is not modified in between, we conclude that task w_m is completed.

We are now able to prove the following theorem.

Theorem 4.1

Program A satisfies properties P1, P2 and P3.

Proof

1. Suppose that process P_j is executing w_m with $m = m_j \geqslant 2$. Then by Lemma 4.2 $M[m-1]$ is **true**, and hence by Lemma 4.4 w_{m-1} is completed and its output is on $U[m-1]$. We conclude that program A satisfies P1.
2. Property P2 follows from (ii) of Lemma 4.1, since m_j is incremented by one in each execution of a loop.
3. Suppose that a process, say process P_j, terminates. This happens only when $m_j = m+1$. Thus by Lemma 4.2 $M[n]$ is **true** and hence by Lemma 4.3 $M[m]$ is **true** for all $m = 1, \ldots, n$. Therefore by Lemma 4.4 all tasks are completed. We have shown that Program A satisfies property P3.

Program A is very reliable in the following sense. Property P3 implies that, even if some processes fail (for reasons external to the algorithm: e.g., "crash" of the processors executing the processes) the program may still continue executing tasks and eventually complete all the tasks provided that there remains at least one active process. We shall not pursue this reliability issue any further in this paper, though we believe it is important.

4.2 A Program with Critical Sections

For problems where we are only interested in the output of the last task w_n, the use of the global arrays $U[1{:}n]$ and $M[1{:}n+1]$ in Program A can be avoided at the expense of using critical sections.

We shall illustrate the idea with the following example. Consider the problem of generating the nth iterate x_n by $x_i \leftarrow \varphi(x_{i-1})$ given the initial iterate x_0. Suppose that we use Program A. Then corresponding to the global array $U[1{:}n]$ we have the global array $x[0{:}n]$ where $x[i]$ keeps the value of the ith iterate, and (4.3) and (4.4) become

$$x[m_j] \leftarrow \varphi(x[m_j-1]).$$

Note that we only need $x[n]$. The use of the array $x[0{:}n]$ is wasteful in space, and might even be impractical (e.g., when n is large and when the elements $x[0], x[1], \ldots, x[n]$ are themselves vectors or complicated structures). The following program solves the problem:

Program B:

```
global integer m; global real x;
Initialization: begin
                $m \leftarrow 1; x \leftarrow x_0$;
                start processes $P_1, \ldots, P_k$
                end
Process $P_j$:  begin integer $m_j$; real $y_j$;
(4.7)           { $m_j \leftarrow m; y_j \leftarrow x$ }
                while $m_j \leqslant n$ do
                    begin
                    $y_j \leftarrow \varphi(y_j)$
(4.8)               { if $m_j = m$ then $(m \leftarrow m_j+1; x \leftarrow y_j)$ };
(4.9)               { $m_j \leftarrow m; y_j \leftarrow x$ }
                    end
                end
```

It is crucial to assume that the statements enclosed within a pair of curly brackets (lines (4.7), (4.8) and (4.9)) are programmed as critical sections. (As a matter of fact, the two lines (4.8) and (4.9) can be programmed as one critical section.) With this assumption it is possible to prove the correctness of the above program. The proof is based on the observation that the global variable m is a non-decreasing function of time which takes on all integer values between 1 and n+1. The proof is relatively easy and hence is omitted here.

Note that, as we already mentioned, x and y_j may represent a large amount of data. Hence the execution of $x \leftarrow y_j$ or $y_j \leftarrow x$ inside a critical section may take a significant amount of time. After presenting, in Section 5, an analysis for programs which do not have critical sections, we will give, in Section 6, an analysis for programs which do have critical sections.

5. SPEED-UP RATIOS — IMPLEMENTATIONS WITHOUT CRITICAL SECTIONS

Let $t_{i,j}$ be the random variable representing the time to execute task w_i by process P_j. *In this and the next section, we assume that the t_{ij}, for $i = 1, \ldots, n$ and $j = 1, \ldots, k$, are independent and identically distributed.* The assumption is reasonable when all tasks are of the same complexity. We shall use T to denote any of the random variables $t_{i,j}$, and use τ to denote the mean of T. We assume that T is independent of k, the number of processes in the algorithm. This is a reasonable assumption when there are more than k processors, memory is sufficiently interleaved and the time lost due to memory interference can be ignored.

It is easy to obtain $T_1(n)$. By (3.1) with k = 1, we have

$$T_1(n) = \tau_1(1) + \tau_1(2) + \ldots + \tau_1(n).$$

Since, in this case, the $\tau_1(i)$ are independent and identically distributed with mean τ, we deduce

(5.1) $\bar{T}_1(n) = n\tau$.

In the rest of the paper, in order to evaluate $\bar{T}_k(n)$, we impose the following further assumptions:

A1. *All processes start at the same time $t = 0$.* (I.e., at t_0 all the k processes start with the execution of task w_1.)

A2. *The random variable T is exponentially distributed with mean τ.*

We observe that by the independence of the $t_{i,j}$ and by assumption A2 the quantities $\tau_k(i)$, $i = 1, \ldots, n$, are independent random variables. It follows, from equation (3.1) and assumption A2, that

(5.2) $\bar{T}_k(n) = \bar{\tau}_k(1) + \ldots + \bar{\tau}_k(n)$.

In addition, by assumption A1, $\tau_k(1)$ is given by the minimum of k random variables distributed as T. Since T is exponentially distributed, the minimum has the mean:

(5.3) $\bar{\tau}_k(1) = \frac{\tau}{k}$.

We now consider $\tau_k(i+1)$ for $i = 1, \ldots, n-1$. Define the distribution probability $p_{k,j}$, $j = 1, 2, \ldots$, as follows. (We use here the same notation as in Section 3.) Let $p_{k,j}$

be the probability that $s_{k,i+1} = t_{l+j}$, given that $s_{k,i} = t_l$ for some l. Hence for $1 \leqslant j \leqslant k$, $p_{k,j}$ is the probability that conditions (a) and (b) of Theorem 3.1 hold. Using the same argument as used in the proof of Theorem 3.1, it is easy to show that $p_{k,j} = 0$ if $j > k$. In addition, assumption A2 implies that, from the memory-less property of the exponential distribution, $p_{k,j}$ is independent of i and l. We have

$$(5.4)\quad \tau_k\,(i+1) = \begin{cases} t_{l+1} - t_l \text{ with probability } p_{k,1}, \\ (t_{l+1} - t_l) + (t_{l+2} - t_{l+1}) \\ \quad\cdot \quad \text{with probability } p_{k;2}, \\ \quad\cdot \\ \quad\cdot \\ (t_{l+1} - t_l) + \ldots + (t_{l+k} - t_{l+k-1}) \\ \quad\quad \text{with probability } p_{k,k}. \end{cases}$$

Since by assumption A2 the random variables $t_{l+1} - t_l$, $l = 1,2, \ldots,$ are independent (and identically distributed) random variables with mean $\frac{1}{k}\tau$, we derive from (5.4) that, for $i = 1, \ldots, n-1$, the mean of $\tau_k\,(i+1)$ is given by:

$$(5.5)\quad \bar{\tau}_k\,(i+1) = \sum_{j=1}^{k} p_{k,j}\,(j\tfrac{\tau}{k}) = \frac{\tau}{k}\sum_{j=1}^{k} jp_{k,j}.$$

By (5.2), (5.3) and (5.5), we obtain that

$$(5.6)\quad \bar{\bar{T}}_k\,(n) = \frac{1}{k}\tau[1 + (n-1)\sum_{j=1}^{k} jp_{k,j}\,].$$

To evaluate $\bar{\bar{T}}_k\,(n)$, we need to know the following quantity:

$$N_k = \sum_{j=1}^{k} jp_{k,j}.$$

Lemma 5.1

$$(5.7)\quad p_{k,j} = \frac{jk!}{k^{j+1}(k-j)!}, \quad \text{for } j = 1, \ldots, k.$$

Proof

We first observe that, by assumption A2, for $l = 1,2, \ldots,$ any one of the k processes is equally likely to complete a task at time t_l. Suppose that $s_{k,i} = t_l$ and $s_{k,i+1} = t_{l+j}$. Then by condition (a) of Theorem 3.1, the j processes completing tasks at time $t_l, t_{l+1}, \ldots, t_{l+j-1}$ are different. This occurs with probability

$$(5.8)\quad \frac{k}{k} \cdot \frac{(k-1)}{k} \cdot \ldots \cdot \frac{(k-j+1)}{k} = \frac{k!}{k^j(k-j)!}\ .$$

Moreover, by condition (b) of Theorem 3.1, the process completing a task at time t_{l+j} must be one of the j processes mentioned above. This occurs with probability $\frac{j}{k}$. Hence the probability that $s_{k,i} = t_l$ and $s_{k,i+1} = t_{l+j}$ is $\frac{j}{k} \cdot \frac{k!}{k^j(k-j)!}$.

The problem of computing the leading terms in the asymptotic series for N_k is rather difficult. Fortunately, some known results can be used here. Define

$$Q_k = \sum_{j=1}^{k} \frac{k!}{k^j\,(k-j)!}.$$

Lemma 5.2

$N_k = Q_k$.

Proof

We have

$$\begin{aligned} N_k &= \sum_{j=1}^{k} jp_{k,j} = \sum_{j=1}^{k} (k-(k-j))\,p_{k,j} \\ &= k\sum_{j=1}^{k} p_{k,j} - \sum_{j=1}^{k} (k-j)\,p_{k,j} \\ &= \sum_{j=1}^{k} \frac{jk!}{k^j\,(k-j)!} - \sum_{j=1}^{k-1} \frac{jk!}{k^{j+1}\,(k-j-1)!} \\ &= \sum_{j=1}^{k} \frac{jk!}{k^j\,(k-j)!} - \sum_{j=1}^{k} \frac{(j-1)k!}{k^j\,(k-j)!} \\ &= \sum_{j=1}^{k} \frac{k!}{k^j\,(k-j)!}. \end{aligned}$$

The leading terms in the asymptotic series for Q_k are known (Knuth, 1973, Eq. (25) in §1.1.11.3):

$$Q_k = \sqrt{\frac{\pi}{2}k} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2k}} - \frac{4}{135k} + \frac{1}{288}\sqrt{\frac{\pi}{2k^3}} + 0\,(k^{-2}).$$

Hence by (5.1), (5.6) and Lemma 5.2, we have the following theorem:

Theorem 5.1

Using k processes, the speed-up ratio is given by

$$S_k\,(n) = \frac{nk}{1 + (n-1)N_k},$$

where

$$N_k = \sqrt{\frac{\pi}{2}k} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2k}} - \frac{4}{135k} + \frac{1}{288}\sqrt{\frac{\pi}{2k^3}} + 0\,(k^{-2}).$$

Asymptotically, when both n and k are large we obtain

$$S_k\,(n) \sim \sqrt{\frac{2}{\pi}} \cdot \sqrt{k} \simeq 0.798\sqrt{k}.$$

6. SPEED-UP RATIOS — IMPLEMENTATIONS WITH CRITICAL SECTIONS

In this section, we analyze speed-up ratios achievable by the algorithms when they are implemented with critical sections.

The diagram of Figure 3 illustrates a portion of a possible scheduling of the tasks by the parallel algorithm with two processes. In the diagram, the crosses and circles indicate the sequences of time instants u_i and v_i, $i=1,2, \ldots,$ when a process completes a task and when the same process completes the subsequent critical section. Since, at any time, only one process can execute the critical section, a process may have to wait before entering the critical section. The periods of waiting times are indicated by the shaded lines. The time instants t_i when processes actually enter the critical section are indicated by the triangles.

As in the preceding section, we assume that the time a process takes to execute a task is a random variable independent of the process and of the task. Let F be its distribution function, and f its density function. Similarly, we assume that the time a process takes to execute the critical section is a random variable independent of the process. Let

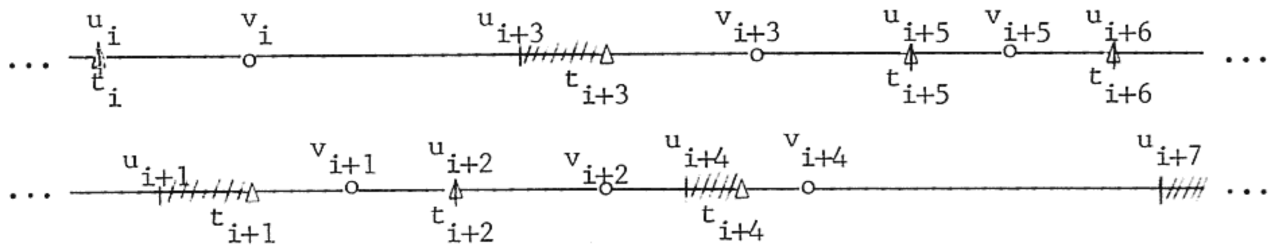

Figure 3. A possible task scheduling with two processes.

B be its distribution function and b its density function. Furthermore, let τ and β denote the average execution times for a task and for the critical section, respectively.

In the following, we derive a general formula for evaluating the speed-up ratio achievable by the parallel algorithm with two processes for the case when *F is an exponential distribution function and B is a general distribution function.*

Observe that at time t_i when a process enters the critical section, the second process is necessarily performing some task (possibly just starting a task). Since the distribution function F is exponential, at time t_i the remaining execution time for the task performed by the second process is distributed according to the same distribution function F. Therefore the evolution of the processes, from time t_i on, is independent of the past for any distribution B. In particular, the random variables $t_{i+1} - t_i$, for $i = 1,2,\ldots$, are independent and identically distributed, and the same holds for the random variables $\tau_k\ (i+1)$, for $i = 1,2,\ldots$, defined in Section 3.

In this section, let $T_1\ (n)$ and $T_2\ (n)$ denote the time to complete task w_n *and the subsequent critical section* by the sequential algorithm and the parallel algorithm with two processes, respectively. Let $\overline{T}_1\ (n)$ and $\overline{T}_2\ (n)$ denote their means. It follows from the above discussion that, for k = 1 and 2, we have:

(6.1) $\overline{T}_k\ (n) = \overline{\tau}_k\ (1) + \overline{\tau}_k\ (2) + \ldots + \overline{\tau}_k\ (n) + \beta$

where the last term, β, accounts for the time to execute the last critical section (after the completion of task w_n).

Consider first the sequential algorithm. In this case, we simply have $\overline{\tau}_1\ (1) = \tau$, and, for $i = 2,\ldots,n$, $\overline{\tau}_1\ (i) = \beta + \tau$. Therefore, by equation (6.1): (6.2) $\overline{T}_1\ (n) = n(\tau + \beta)$.
(Here we ignore the fact that in the sequential algorithm the code corresponding to critical sections in the parallel algorithm can be shortened, since there is no need to include synchronization primitives.)

Consider now the parallel algorithm. As (5.3), we have

(6.3) $\overline{\tau}_2\ (1) = \frac{1}{2}\tau$.

For j = 1 and 2, let p_j be the probability that $s_{2,i+1} = t_{l+j}$, given that $s_{2,i} = t_l$ for some l. As in Section 5, by Theorem 3.1, we obtain, for $i = 1,2,\ldots,n-1$,

(6.4) $\tau_2\ (i+1) = \begin{cases} t_{l+1} - t_l & \text{with probability } p_1, \\ (t_{l+1} - t_l) + (t_{l+2} - t_{l+1}) & \text{with probability } p_2. \end{cases}$

We have already mentioned that the random variables $t_{l+1} - t_l$, $l = 1,2,\ldots$, are independent and identically distributed. Let μ denote their mean. It follows from (6.4) that the mean of $\tau_2\ (i+1)$ is given by

(6.5) $\overline{\tau}_2\ (i+1) = p_1 \cdot \mu + p_2 \cdot 2\mu = (2 - p_1)\mu$,

since $p_1 + p_2 = 1$.

The following lemma establishes the values of μ and p_1.

Lemma 6.1

(6.6) $\mu = \beta + \frac{\tau}{2} B^*(1/\tau)$,

(6.7) $p_1 = \frac{1}{2} B^*(1/\tau)$,

where B^* is the Laplace transform of the distribution function B.

Proof

We consider transitions for passing from time t_i to time t_{i+1}. Up to a permutation of the processes, there are three possible transitions as defined by the diagrams in Figure 4,
where the notation of Figure 3 is assumed.

Let $H_j\ (t)$, j = 1,2 and 3, be the probability that transition A_j takes place and that $t_{i+1} - t_i \leqslant t$. We have:

$$H_1\ (t) = \int_0^t\ [1 - F(x)]\ \int_0^x\ b(y)f(x\text{-}y)dy\ dx,$$

$$H_2\ (t) = \int_0^t\ f(x) \int_0^x\ b(y)[1 - F(x\text{-}y)\ dy\ dx,$$

$$H_3\ (t) = \int_0^t\ b(x)F(x)dx.$$

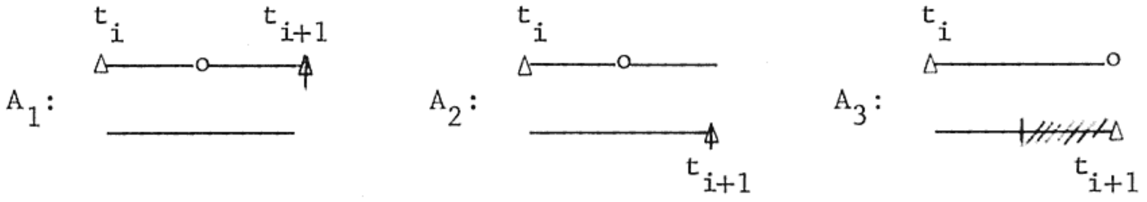

Figure 4. Three possible transitions.

But we observe that $H(t) = H_1(t) + H_2(t) + H_3(t)$ is the distribution function for $t_{i+1} - t_i$ and that the same process enters the critical section at both times t_i and t_{i+1} only with transition A_1. Hence:

$$\mu = \int_0^\infty t\,dH(t) = \int_0^\infty [1 - H(t)]\,dt,$$

$$p_1 = \int_0^\infty dH_1(t) = \int_0^\infty [1 - F(x)] \int_0^x b(y)f(x-y)\,dy\,dx,$$

from which equations (6.6) and (6.7) follow easily.

By collecting the preceding results, we obtain the following theorem:

Theorem 6.1:

$$S_2(n) = \frac{n(\tau+\beta)}{(n\text{-}1)\,[2 - \frac{1}{2}B^*(1/\tau)]\,[\beta + \frac{1}{2}\tau B^*(1/\tau)] + \frac{1}{2}\tau + \beta}$$

$$= \frac{1}{2 - \frac{1}{2}B^*(1/\tau)} \quad \frac{\tau+\beta}{\beta + \frac{1}{2}B^*(1/\tau)} + 0(\tfrac{1}{n})$$

We give below $B^*(1/\tau)$ for some distribution functions B.

(i) B is exponential (with parameter $1/\beta$):

$$B^*(1/\tau) = \frac{\tau}{\tau+\beta}.$$

(ii) B is the Dirac function at the point β:

$$B^*(1/\tau) = e^{-\beta/\tau}.$$

(iii) B is uniform over $[a,b]$:

$$B^*(1/\tau) = \frac{e^{-a/\tau} - e^{-b/\tau}}{(b\text{-}a)/\tau}.$$

In Figure 5, we have plotted the asymptotic speed-up ratio S_2 as a function of the ratio $\alpha = \tau/(\tau+\beta)$ for the three distributions mentioned above (in the third case, a and b have been chosen as $\beta/2$ and $3\beta/2$, respectively).

When α tends to 0 (or β tends to infinity), the algorithm approaches its worst performance, since the evaluations of the two processes tend to be exactly interleaved. When $\alpha = 1$ (or $\beta = 0$), the critical section is non-existent and we have the results of Section 5.

We observe from Figure 5 that the best speed-up ratio is always obtained when B is an exponential distribution (the first case). We also note that the results obtained for the two other cases are very close to each other and close to the results obtained with the exponential distribution. This suggests that the results obtained with the exponential distribution could be used as approximations to results obtained with other distributions.

We can observe from Figure 5 that, unlike the implementation without critical section, better speed-up is not necessarily achieved by using more processes, though we assume that a processor is always available to each process! More precisely, the figure indicates that (when B is an exponential distribution) in order to achieve the best speed-up when two processors are available, one should create two processes when $\alpha > 0.586$, but only one process when $\alpha \leqslant 0.586$. Similar results are useful in practice, since they can be used to determine the optimal number of processes to create in order to minimize the overall execution time.

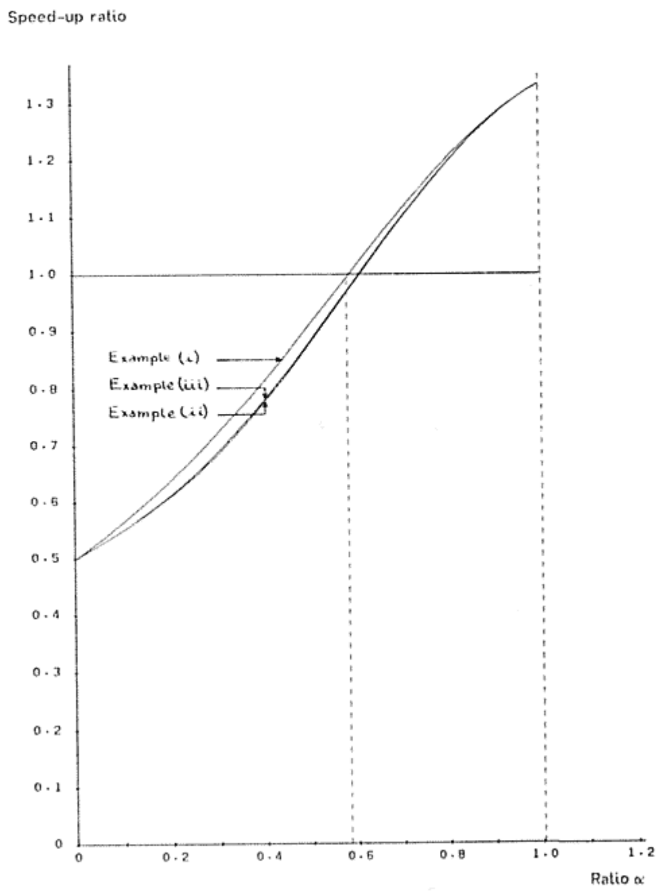

Figure 5. Speed-up ratio with two processes for various distributions B.

7. CONCLUSION AND OPEN PROBLEMS

In recent years, research in parallel algorithms has dealt mostly with synchronized array or vector processors such as the ILLIAC IV or the CDC STAR, and there are very few results on the design and analysis of algorithms for asynchronous multiprocessors. In this paper, we have proposed a novel method of using asynchronous multiprocessors which takes advantage of their asynchronous behaviour. We have also presented analytic techniques to evaluate the performance of an asynchronous algorithm using the method. The algorithm is expected to achieve a large speed-up when the fluctuations in the task execution times are relatively large. If inherent parallelism in tasks can also be utilized, a larger speed-up may be obtained.

As noted in Section 2, at any given time at most one process in the algorithm presented is doing work useful for later computation. In this sense, the algorithm achieves its speed-up through redundant computation performed concurrently by a number of processes. It has long been recognized that redundancy brings reliability. Indeed, as observed in Section 4, the algorithm enjoys a nice reliability property. Thus, this paper shows a technique by which both speed and reliability can be achieved through redundant computation. We expect that similar ideas can be used to construct fast and reliable algorithms for a variety of problems.

For the implementation with critical sections we obtained analytic results for two processes. The results show that the parallel algorithm using two processes is not necessarily faster than the sequential algorithm, because of the critical section overheads associated with the parallel algorithm. This confirms practical experience that the speed-up ratio does not necessarily increase as the number of processes increases. It would be interesting to extend our analytic results for more than two processes. We have chosen to deal with a simple problem by imposing the condition that the tasks are linearly ordered. An interesting extension would be to consider a set of tasks (possibly generated dynamically) which are ordered by a directed graph (i.e., partially rather than linearly ordered). Another interesting extension would be to design algorithms where the execution of a task by a process may be interrupted by another process. We expect that this approach could result in more efficient algorithms, since processes which were not doing useful work could be interrupted. A careful performance analysis including the additional overheads introduced by the interruption mechanism would be needed.

Following circulation of a draft of this paper, Barak and Downey (1980) generalised our results by considering the execution of a chain of tasks with interrupts, and suggested several other directions in which our results might be extended.

The results of this paper are not only applicable to multiprocessor systems. The ideas can be used to solve any problem in operation research which satisfies conditions similar to C1, C2 and C3.

REFERENCES

BARAK, A.B. and DOWNEY, P.J. (1980): "Asynchronous parallel execution of a chain of tasks with interrupts", to appear. (Available as Tech. Report No. 227, Comp. Sci. Dept., Pennsylvania State Univ., Dec. 1977).

BARLOW, R.H. and EVANS, D.J. (1979): "A parallel organisation of the bisection algorithm", *Comp. J.* 22, 267-269.

BAUDET, G.M. (1978): "Asynchronous Iterative Methods for Multiprocessors", *J. ACM* 25, 226-244.

FULLER, S.H., JONES, A.K. and DURHAM, I. (Eds.) (1977): "The Cm* Review Report", Technical Report, Carnegie-Mellon University, Department of Computer Science, June 1977.

KNUTH, D.E. (1973): *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA, second edition.

KUNG, H.T. (1976): "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", *Algorithms and Complexity: New Directions and Recent Results*, edited by J.F. Traub, Academic Press, 153-200.

KUNG, H.T. and SONG, S.W. (1977): A Parallel Garbage Collection System and Its Correctness Proof, *Proc. 18th Annual IEEE Symposium on Foundations of Computer Science*, October 1977, 120-130.

ROBINSON, J.T. (1979): "Analysis of Asynchronous Multiprocessor Algorithms with Application to Sorting", *IEEE Transactions on Software Engineering*, SE-5, 24-31.

WULF, W.A. and BELL, C.G. (1972): "C.mmp – A Multi-Mini-Processor", *Proc. of the AFIPS 1972 Fall Joint Computer Conference*, Vol. 41, 1972, 765-777.

BIOGRAPHICAL NOTES

G.M. Baudet obtained his Ph.D. in Computer Science at Carnegie-Mellon University in 1978. His thesis was on "The design and analysis of algorithms for asynchronous multiprocessors". He is currently employed as a research scientist at IRIA-LABORIA, France.

R.P. Brent obtained his Ph.D. in Computer Science at Stanford University in 1971. He is currently Professor of Computer Science and Head of the Department of Computer Science at the Australian National University. His interests include computational complexity, numerical software, parallel processing and VLSI design.

H.T. Kung obtained his Ph.D. in Mathematics at Carnegie-Mellon University in 1973. He is currently Associate Professor of Computer Science at Carnegie-Mellon University. His interests include parallel processing, computational complexity and VLSI design.