SYNCHRONIZED AND ASYNCHRONOUS PARALLEL ALGORITHMS FOR MULTIPROCESSORS

H. T. Kung

June 1976

To appear in New Directions and Recent Results in Algorithms and Complexity, Edited by J. F. Traub, Academic Press, 1976.

SYNCHRONIZED AND ASYNCHRONOUS PARALLEL ALGORITHMS FOR MULTIPROCESSORS

H. T. Kung
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

Abstract

Parallel algorithms for multiprocessors are classified into synchronized and asynchronous algorithms. Important characteristics with respect to the design and analysis of the two types of algorithms are identified and discussed. Several examples of the two types of algorithms are considered in depth.

Table of Contents

This research was supported in part by the National Science Foundation under Grant MCS75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422.

1. Introduction

The multiprocessor user is currently confronted with a large and increasing number of processors. For efficient system utilization and fast response to the user, it is necessary to use parallel algorithms for solving a single problem. This paper studies parallel algorithms for multiprocessors.

Following Flynn's [66] classification scheme, parallel computers are classified into SIMD (single-instruction stream-multiple-data stream) machines and MIMD (multiple-instruction stream-multiple-data stream) machines. With SIMD machines, one stream of instructions controls a number of synchronized processors, each operating upon its own memory. An example of SIMD machines is the array processor such as ILLIAC IV (Barnes, et al. [68]). With MIMD machines, the processors have independent instruction counters, and operate in a speed independent manner on shared memories. In this paper, by multiprocessors we mean MIMD machines. An example of multiprocessors is C.mmp (Wulf and Bell [72]) at Carnegie-Mellon University. Considerations in designing algorithms for SIMD machines and those for multiprocessors are quite different. Algorithms for SIMD machines are not dealt with in this paper. The reader is referred to the recent survey by Heller [76] for numerical SIMD algorithms, and the paper by Thompson and Kung [76] for an example of nonnumerical SIMD algorithms. For a survey of parallel computation in a broad sense, the reader is referred to Kuck [75]. Although quite a lot of research has been done on SIMD algorithms, there are few results available concerning the design and analysis of multiprocessor algorithms. As multiprocessors with increasing numbers of processors are becoming available, research on multiprocessor algorithms seems to be of utmost importance at this time. This is the motivation behind this paper.

This paper intends to identify some of the important and unique issues concerning multiprocessor algorithms. They are illustrated by three specific examples, which are given in Sections 3, 4, and 5. In Section 2, multiprocessor algorithms are classified into synchronized and asynchronous algorithms, and basic concepts are introduced. Section 6 considers asynchronous algorithms where processes can be interrupted, and Section 7 contains considerations on the optimal number of processes one should create for a multiprocessor algorithm. Summary and conclusions are given in the last section.

Parts of this paper are summaries of results from other papers, in particular, results of Section 3, belong to Hyafil and Kung [76], and results of Section 4.2 belong to Baudet, Brent and Kung [76].

2. Basic Concepts and Definitions

2.1 Parallel Algorithm as a Collection of Concurrent Processes

We define a parallel algorithm for multiprocessors as a collection of concurrent processes that may operate simultaneously for solving a given problem. We do not attempt to define formally the term "process" here. The reader could read, for example, Habermann[76] for a good discussion on the concept of processes. For our purpose we view a process as the execution of a procedure in a multiprocessor operating system. Thus, a process is controlled by a program and at most one processor, which is assigned by the operating system, carries out this program at any given time. During the lifetime of a process, different processors may be assigned to it on various time intervals. It turns out that it is convenient and more useful to think that a program for solving a given problem is carried out by processes rather than processors. For instance, with this concept one can often regard a piece of the program to be carried out by <u>one</u> process, although it may actually be done by many processors. This is part of the motivation for our definition of a parallel algorithm. As the section proceeds, the reader will find that it is useful to have such a definition for describing many other concepts. If a parallel algorithm is a collection of k processes, we shall often say it is a <u>parallel algorithm with k processes</u>. If k = 1, it is called a <u>sequential algorithm.</u>

To ensure that a parallel algorithm works correctly and uses parallelism effectively for solving a given problem, it is usually necessary to have interactions among the processes. Hence in the program which controls a process there may be some points where the process can communicate with other processes. We call such points <u>interaction points</u>. The interaction points divide a process into <u>stages</u>. Thus, at the end of each stage a process may communicate with other processes before starting the next stage.

2.2 Fluctuations in Process Speed

The time taken by a fixed stage of a process is usually not a constant. The fluctuations may be due to both the multiprocessor system and the input to the stage.

<u>System</u>

(i) The multiprocessor may consist of processors with different speeds. For example, the current configuration of C.mmp includes both PDP-11/20 and PDP-11/40 processors. The latter processors are considerably faster than the former ones. A process may be run in a fast or slow manner, depending upon which processors are assigned to it during the stage.

(ii) The individual processors may be asynchronous.

(iii) A process may be delayed due to memory conflicts.

(iv) From time to time the operating system on the multiprocessor may assign certain processors to perform I/O, allocate processors to processes, switch a processor from one process to another, and so on. Hence during the period when a processor is carrying out a stage of a process, it could be interrupted by the operating system and start doing something else. In this case, a time consuming context swap is performed, and the stage is either taken over by another processor or suspended for an indefinite amount of time.

(v) In a multiple user environment, the amount of resources allocated to a particular process at a given time is a variable, depending upon the number of processes the users have created and their priorities. Thus, the speed of a process may be influenced by the whole user community.

Input

The work taken by an algorithm may depend on the instances of its input. For example, the number of comparisons needed to sort n elements by Quicksort ranges from O(n log n) to $O(n^2)$, depending upon the ordering of the input elements. As another example, consider the problem of evaluating a function. Suppose that we want to evaluate the normal distribution function

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{1}{2}t^2} dt$$

at a point. To achieve good accuracy, the evaluation at a point in the "central area" is done by a Taylor series approximation, while that at a point in one of the "tail areas" is done by a continued fraction approximation. Hence the work needed to evaluate a function at a point may depend upon the position of the point. Since in general the property of the input to a stage of a process is unpredictable (or regarded as unpredictable), so is the work performed by the stage. This implies that the time taken by a stage can very in an unpredictable way.

Motivated by the preceding discussion, we assume in this paper that in running a parallel algorithm the time taken by a stage of any of its processes is a random variable satisfying some distribution function. In a fixed computing environment the distribution function can (hopefully) be estimated.

2.3 Synchronized Parallel Algorithms

A synchronized parallel algorithm is a parallel algorithm consisting of processes with the following property: There exists a process such that some stage of the process is not activated until another process has finished a certain portion of its program. The needed timing can be achieved by using various synchronization primitives. For example, suppose that we want to compute $(A\times B) + (C\times D\times E)$ by two processes. We may construct a parallel algorithm by creating process P_1 consisting of only one stage, $X \leftarrow A\times B$, and process P_2 consisting of two stages, $Y \leftarrow C\times D\times E$ and $S \leftarrow X+Y$. Clearly, the activation of the second stage of process P_2 is subject to the condition that process P_1 is complete. Thus, this is a synchronized parallel algorithm. More examples will be presented later in the paper. If we use notation in Habermann [76], then a parallel algorithm consisting of "cooperating processes" is a synchronized parallel algorithm. For convenience, we shall often say synchronized algorithms instead of synchronized parallel algorithms.

Since the time taken by a stage of a process is a random variable, synchronized algorithms have the drawback that some processes may be blocked at a given time, so the performance of the algorithm is degraded. To be more precise, consider a synchronized algorithm with k processes. Assume that this algorithm is run on a multiprocessor system which consists of k identical processors and the algorithm takes time

U. During the time U let u_i denote the total time that i processes are active, i.e., k-i processes are blocked. Note that $U = u_0 + u_1 + \ldots + u_k$ and that the algorithm can be run on a single processor of the system in time at most $\sum_{i=1}^{k} iu_i$. Thus, by using k processors, the algorithm is sped up at most by a factor of

$$\bar{S}_k = \sum_{i=1}^{k} iu_i / \sum_{i=1}^{k} u_i ,$$

which is $\le k$ as one might expect. $\bar{S}_k$ may be computed if the u_i are known. For example, if we know that at most k/2 processes are active 50 percent of the time,

i.e., $\sum_{i=1}^{k/2} u_i \ge \sum_{i=(k/2)+1}^{k} u_i$, then

$$\bar{S}_k \le \frac{k}{2}\left[1 + \left(\sum_{i=(k/2)+1}^{k} u_i / \sum_{i=1}^{k} u_i\right)\right] \le \frac{3k}{4} .$$

Hence in this case the speed up is at most 3/4 of what one would hope.

The degradation of performance can be made even clearer by considering the class of synchronized parallel algorithms, where only identical stages of processes are synchronized. In general, the synchronized parallel algorithms adapted from algorithms for SIMD machines are of this type. Suppose that we want to synchronize k identical stages and that the time taken by the ith stage is a random variable t_i. Since the stages are all identical, $t_1, \ldots, t_k$ are identically distributed random variables with mean, say, $\bar{t}$. Synchronizing the stages means that until all stages are complete a new stage of any process can not be activated. Therefore the expected time taken by the synchronized stage of any process is the mean $\bar{T}$ of the random variable $T = \max(t_1,\ldots,t_k)$ rather than $\bar{t}$. In general, $\bar{T}$ is larger than $\bar{t}$. We define the ratio $\lambda_k = \bar{T}/\bar{t}$ to be the penalty factor of synchronizing the k identical stages. Clearly if the penalty factor is large, then the performance of the synchronized algorithm is largely degraded. G. Baudet has made an interesting observation that if the t_i are identical and independent exponentially distributed random variables, then the penalty factor λ_k is the kth harmonic number H_k. Note that H_k grows like ln k as k increases.

Both the speed up bound $\bar{S}_k$ and the penalty factor discussed above give us some indications of the average performance of synchronized algorithms. But in the worst cases, synchronized algorithms may take an unacceptably long time, since it is possible that a process is blocked while waiting for a signal which is supposed to be issued by some "dead" process. Finally, we note that the execution time of the needed synchronization primitives in synchronized algorithms is often non-negligible in practice (see Section 7).

2.4 Asynchronous Parallel Algorithms

An asynchronous parallel algorithm is a parallel algorithm with the following properties:

(i) There is a set of global variables accessible to all processes.

(ii) When a stage of a process is complete, the process first reads some global variables. Then based on the values of the variables together with the results just obtained from the last stage, the process modifies some global variables, and then activates the next stage or terminates itself. In many cases, to ensure logic correctness, the operations on global variables are programmed as critical sections (cf. Dijkstra [68]).

Thus in an asynchronous parallel algorithm, the communications between processes are achieved through the global variables, or shared data. There is no explicit dependency between processes, as found in synchronized parallel algorithms. The main characteristic of an asynchronous parallel algorithm is that its processes never wait for inputs at any time but continue or terminate according to whatever information is currently contained in the global variables. It is called an "asynchronous" parallel algorithm because synchronizations are not needed for ensuring that specific inputs are available for processes at various times. However, one should note that processes may be blocked from entering critical sections, which are needed in many algorithms. We shall often say asynchronous algorithms for short, instead of asynchronous parallel algorithms.

2.5 The Time Taken by a Parallel Algorithm

The time taken by a parallel algorithm is defined to be the elapsed time of the process in the algorithm which finishes last. The elapsed time of a process is the sum of the following three quantities:

(i) Basic Processing Time

Recall that a process consists of consecutive stages and that the time taken by a stage is a random variable. The basic processing time of a process is the sum of the times taken by its stages. In this paper, it is always assumed that the random variable associated with each stage is known.

(ii) Blocked Time

A process may be blocked at the end of a stage because it waits for inputs in a synchronized algorithm, or for the entering of a critical section in an asynchronous algorithm. The blocking time of a process is the total time that the process is blocked.

(iii) Execution Time of Synchronization Primitives

Synchronization primitives are needed for synchronizing processes and implementing critical sections. The execution time of these primitives is often non-negligible in practice.

We assume that the random variable associated with a stage of a process is invariant under the addition of another process to the multiprocessor system, as long as the total number of processes having been created is no more than the number of processors in the system. In other words, we assume that the basic processing time of a process is not affected by the presence of other processes in the system when the system is not "over-saturated". This assumption seems to hold for most plausible multiprocessor systems. Throughout the paper, when we compare the times taken by parallel algorithms consisting of different numbers of processes, it is always assumed that the system is not over-saturated as any of the algorithms is running. Thus, in the analysis, the time taken by a stage of a process in a parallel algorithm is a random variable, which is defined independently of how many processes the algorithm consists of. However, when we compare two parallel algorithms with the <u>same</u> number of processes, we can allow the situation where the system is over-saturated. In this case, we just imagine that each process is run by a virtual processor which has only a fraction of the processing power of a real processor.

3. <u>The First Example: Search for Zeros</u>

In this section we consider the classical problem of locating a zero of a function, which is defined as follows: Given a continuous function f having opposite signs at the endpoints of an interval of length L, locate a zero of f within a unit interval. One should note that the algorithms presented in this section can be easily modified to deal with discrete f, so they can, for example, be used for searching an ordered list for a desired item in the list. Furthermore, our asynchronous zero-searching parallel algorithms can also be modified to locate the maximum of a unimodal function (see the end of Section 3.4).

In the following definitions we assume that a single process is used. Let the time needed to evaluate f at a point in the interval be a random variable t with mean $\bar{t}$, and, following each function evaluation, the time needed to calculate the position of the next evaluation point and to check the stopping criteria be another random variable c with mean $\bar{c}$. In this section we assume that $\bar{t}$ is much larger than $\bar{c}$ so that c can be ignored in the analysis. We also assume that execution time of synchronization primitives can be ignored. These assumptions will be dropped in Section 7.

Binary search is probably the best known search method. It takes at most $\lceil \log_2 L \rceil$ function evaluations and is optimal in the minimax sense. The expected running time is $\lceil \log_2 L \rceil \cdot \bar{t}$. The method is inherently sequential, since only one function evaluation can be done at any given time. In the following we consider some parallel algorithms.

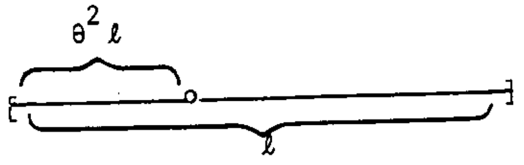
3.1 Synchronized Zero-Searching Algorithms

The obvious parallel zero-searching algorithm consisting of k processes is defined as follows. At each "iteration" each process evaluates f at one of the k points which divide the current interval of uncertainty into k+1 subintervals of equal length. The evaluation is considered as a stage of the process. The k identical stages are synchronized in the sense that when all of them are complete, one of the processes computes a new interval of uncertainty. Clearly, every iteration reduces the length of the interval of uncertainty by a factor of k+1. Hence the algorithm uses $\lceil \log_{k+1} L \rceil$ iterations. As far as the number of iterations is concerned, the algorithm is clearly optimal in the minimax sense, among all synchronized parallel algorithms with k processes. However, the expected time for each iteration is $\lambda_k \bar{t}$ rather than $\bar{t}$, where λ_k is the penalty factor of synchronizing k function evaluations. Thus the expected running time of the algorithm is $\lceil \log_{k+1} L \rceil \cdot \lambda_k \bar{t}$. The synchronized parallel algorithm can be inefficient when λ_k is large, which usually happens when k is large.

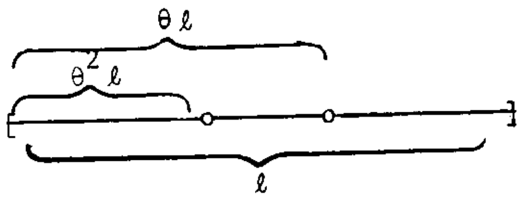
3.2 An Asynchronous Zero-Searching Algorithm with Two Processes - Algorithm AZ_2

We shall introduce a natural asynchronous zero-searching algorithm with two processes, which is based on a Fibonacci law. The algorithm, called Algorithm AZ_2, will be defined by its transitions between various states. There are two types of states, $A_1(\ell)$ and $A_2(\ell)$, which are defined by the following graphs:

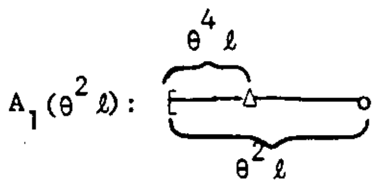
$A_1(\ell)$:



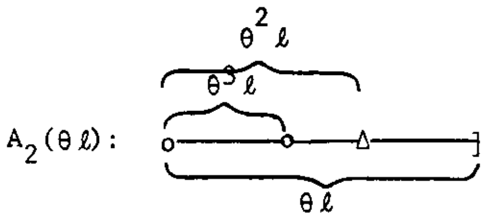
$A_2(\ell)$:



where $\theta^2+\theta = 1$, i.e., $\theta = .618 \ldots$ is the reciprocal of the golden ratio ϕ. The first graph indicates that state $A_1(\ell)$ is the state for which the interval of uncertainty is of length ℓ and f is evaluated simultaneously at the point "o" inside the interval and another point outside the interval, which is nct shown in the graph. Similarly, the second graph indicates that state $A_2(\ell)$ is the state for which the interval of uncertainty is of length ℓ and f is evaluated simultaneously at two points, denoted by "o", both inside the interval. Suppose that we are at state $A_2(\ell)$ and that without loss of generality the evaluation at the left point will finish first. Then after the evaluation at the left point is complete, the new interval of uncertainty is either [——o or o——o————], depending upon the sign of the outcome. (Here we assume that the outcome is nonzero, for otherwise a zero is found and we are done.) If the first case occurs, then the process which just finished the evaluation at the left point activates a new evaluation at the point "Δ", which is defined by the following graph:



Hence state $A_1(\theta^2 \ell)$ is obtained. Similarly, state $A_2(\theta \ell)$ can arise from the second case, as depicted by the graph:



Hence state $A_2(\ell)$ is transited to either state $A_1(\theta^2 \ell)$ or $A_2(\theta \ell)$. This transition is denoted by

(3.1) $A_2(\ell) \rightarrow A_1(\theta^2 \ell) \vee A_2(\theta \ell)$.

It is not difficult to see that the corresponding rule for state $A_1(\ell)$ is:

(3.2) $A_1(\ell) \rightarrow A_1(\theta^2 \ell) \vee A_1(\theta \ell) \vee A_2(\ell)$.

In fact, transition rules (3.1) and (3.2) completely define Algorithm AZ_2. Suppose that the algorithm starts from state $A_1(L)$. Then it associates with the following transition tree (assume that f does not vanish at any of the evaluation points):

$$\begin{tikzcd}
&&&& A_1(L) \arrow[dlll] \arrow[d] \arrow[drrr] &&& \\
& A_1(\theta^2L) \arrow[dl] \arrow[d] \arrow[dr] &&& A_1(\theta L) \arrow[dl] \arrow[d] \arrow[dr] &&& A_2(L) \arrow[dl] \arrow[d] \\
A_1(\theta^4L) & A_1(\theta^3L) & A_2(\theta^2L) & A_1(\theta^3L) & A_1(\theta^2L) & A_2(\theta L) & A_1(\theta^2L) & A_2(\theta L) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{tikzcd}$$

Figure 3.1

The algorithm passes through all the states on one of the paths in the tree. The particular path taken by the algorithm depends upon the input function f and the relative speeds of the two processes.

Formally we can define Algorithm AZ_2 as an asynchronous algorithm consisting of two identical concurrent processes P_i, $i = 1,2$, which are controlled by the following program:

```
process P_i
begin
    while the length of the interval of uncertainty > 1 do
        begin

(3.3)   compute the position of the next evaluation point "Δ";

(3.4)   evaluate f at the point "Δ";

(3.5)   read and update the global variables
        end;
end
```

The global variables in the program consist of the type of the current state and the positions of the endpoints of the current interval of uncertainty. By examining the global variables, the position of the next evaluation point "Δ" can be computed at step (3.3). After the function evaluation at step (3.4) is complete, the global variables are updated at step (3.5). To guarantee that transition rules (3.1) and (3.2) are satisfied it is necessary that steps (3.3) and (3.5) be programmed within a critical section.

An important property of Algorithm AZ_2 is that it associates with a very simple transition tree (Figure 3.1), so it can be analyzed. Let N be the number of function evaluations completed by the algorithm. Since the evaluations are done by two concurrent processes, the expected time taken by the algorithm is $\sim N\bar{t}/2$ as $N \to \infty$. (A rigorous proof of this, in fact, will be given later in Section 5 in a rather different context.) Thus, the speed-up ratio between the expected time taken by binary search and that by Algorithm AZ_2 is

$$S_2 \sim \frac{(\log_2 L)\bar{t}}{\frac{N}{2}\bar{t}} = \frac{2 \log_2 L}{N}, \qquad \text{as } N \to \infty.$$

Therefore we are interested in determining the value of N. Note that the value of N in the worst case is given by the length of a longest path in the transition tree, in the best case by the length of a shortest path, and in the average case by the average path length.

Let p be the probability that two consecutive evaluations are executed by the same process. Some of the results in Hyafil and Kung [76] are summarized in the following:

(i) In the worst case:

$$N \sim \log_\phi L, \quad S_2 \sim 1.388.$$

Algorithm AZ_2 is optimal in the minimax sense, as far as the number of required function evaluations is concerned. Algorithm AZ_2 beats the synchronized algorithm with two processes, when the penalty factor $\lambda_2 > 1.142$.

(ii) In the best case:

$$N \sim (\log_\phi L)/2 \ , \ S_2 \sim 2.777.$$

(iii) In the average case:

$$N \sim a(p) \cdot \log_\phi L, \ S_2 \sim 1.388 \frac{1}{a(p)},$$

Algorithm AZ_2 beats the synchronized algorithm with two processes, when $\lambda_2 > 1.142 \cdot a(p)$, where $a(p)$ is a function of p defined as follows:

<u>Case 1</u>: If the zero is uniformly distributed in the original interval of uncertainty, then

$$a(p) = \frac{2.236-1.618p}{2.236-1.382p} .$$

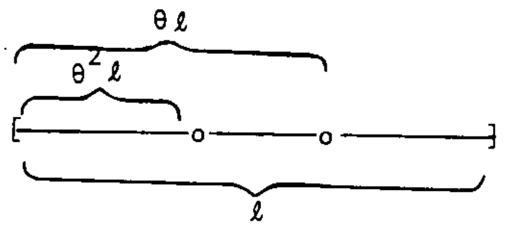
<u>Case 2</u>: If the sign of the function value at any point inside the original interval of uncertainty is equally likely to be positive or negative, then

$$a(p) = \frac{6-4p}{6-3p} .$$

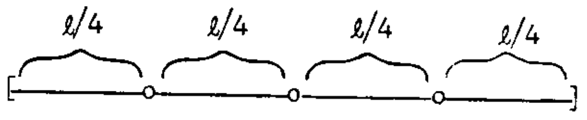
Note that in both cases, $a(p)$ decreases as p increases. This means that the algorithm is better when the variances in the evaluation time are large. The value of p can be derived from a formula in Baudet, Brent and Kung [76], as long as, for example, the probability density function of the random variable t is known.

3.3 Asynchronous Zero-Searching Algorithms with Three or More Processes

The basic pattern for defining the states in Algorithm AZ_2 is



which is state $A_2(\ell)$ as it stands and becomes state $A_1(\ell)$ if the right middle point is deleted. An asynchronous algorithm with three processes can be similarly defined by using the following two patterns:



and

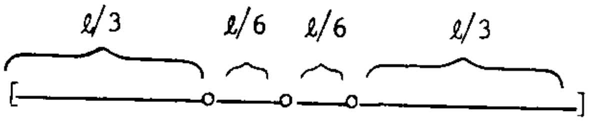


Figure 3.2

In general, $\lfloor k/2 \rfloor+1$ patterns are sufficient for defining an asynchronous algorithm with k processes. If k is odd, the algorithm is optimal in the minimax sense. No similar result is known if k is even. In particular, we do not know whether or not an asynchronous algorithm with four processes can be constructed by using two patterns. This is an interesting open problem.

An asynchronous zero-searching algorithm with k processes corresponds, in a natural way, to an asynchronous algorithm with k-1 processes for locating the maximum of a unimodal function. For example, if $k = 2$, the pattern used for

Algorithm AZ_2 is exactly that used in the well-known Fibonacci search for the maximum (Kiefer [53]). The patterns in Figure 3.2 give us an asynchronous algorithm with two processes for locating the maximum of a unimodal function, which turns out to be always faster than the optimal synchronized algorithm with two processes (Avriel and Wilde [66] and Karp and Miranker [68]) as long as the penalty factor is greater than one. The details of the results mentioned above can be found in Hyafil and Kung [76].

4. The Second Example: Iterative Algorithms

Many problems in practice are solved by iterative methods. For example, zeros of functions f may be computed by the Newton iteration:

(4.1) $x_{i+1} = x_i - f'(x_i)^{-1} f(x_i)$,

and solutions of linear systems by iterations of the form

(4.2) $\bar{x}_{i+1} = A\bar{x}_i + \bar{b}$,

where the $\bar{x}_i$, $\bar{b}$ are n-vectors and A is an nxn matrix. Assume that we are given a general iterative method,

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

and are interested in designing algorithms for which multiprocessors can be employed to speed up the computation of the iterative process (4.3). Several types of algorithms will be presented in this section. All of them are based on the following two strategies or a combination:

(i) The first strategy is to exploit parallelism within the iteration function φ. For example, one may observe that in the iteration (4.1) the evaluations of f and f' at x_i can be done in parallel, and in the matrix iteration (4.2) all the components of the vector $\bar{x}_{i+1}$ can be computed simultaneously.

(ii) The second strategy is to exploit the fluctuations in process speed (cf. Section 2.2). The idea is to use more than one process to compute the same function in parallel, and expect that the process which obtains the result first takes less than the average time.

Iterations (4.1) and (4.2) will often be used for illustrating the algorithms.

4.1 Synchronized Iterative Algorithms

In a synchronized iterative algorithm, the iteration function is decomposed so that each iteration step is done by more than one process, and the processes are synchronized at the end of each iteration. Essentially, the algorithm generates the iterates by (4.3) just as the sequential algorithm does, except that within each iteration parallel computation is used. Thus, the algorithm differs from the sequential algorithm in the time taken by each iteration.

There is a natural synchronized iterative algorithm with two processes for performing the Newton iteration (4.1). At each iteration of the algorithm, $f(x_i)$ and $f'(x_i)$ are computed in parallel, and only after both evaluations are complete the computation for x_{i+1} is allowed to start (this is the place where synchronization is needed). Since f and f' are not the same function in general, the times needed for evaluating f and f' are probably different. In fact, when f is a vector function consisting of n components, a good approximation to $f'(x_i)$ will need n+1 evaluations of f. Hence, if n is large, then the process which evaluates f probably wastes much of its time in waiting at each iteration for the other process to finish the evaluation of f'. This certainly degrades the performance of the algorithm. This example illustrates the fact that synchronized iterative algorithms are not suitable for those iteration functions which cannot be decomposed into mutually independent tasks of the same complexity.

Synchronized iterative methods which do not suffer from the drawback mentioned in the preceding remark can be easily constructed for the matrix iteration (4.2). For simplicity, let us assume that we are interested in constructing a parallel algorithm consisting of two concurrent processes. Perhaps the most natural approach (especially for SIMD machines) is to decompose each vector $\bar{x}_i$ into two segments $\bar{x}_i^{(1)}$ and $\bar{x}_i^{(2)}$ each of size n/2, and update them by two parallel processes as follows:

$$(4.4)\qquad \begin{bmatrix} \bar{x}_{i+1}^{(1)} \\ \bar{x}_{i+1}^{(2)} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \bar{x}_i^{(1)} \\ \bar{x}_i^{(2)} \end{bmatrix} + \begin{bmatrix} \bar{b}^{(1)} \\ \bar{b}^{(2)} \end{bmatrix},$$

where $\bar{x}_{i+1}^{(1)} = A_{11}\bar{x}_i^{(1)} + A_{12}\bar{x}_i^{(2)} + \bar{b}^{(1)}$ and $\bar{x}_{i+1}^{(2)} = A_{21}\bar{x}_i^{(1)} + A_{22}\bar{x}_i^{(2)} + \bar{b}^{(2)}$. That is, at an iteration step, each process updates half of the components and starts the next iteration only after both processes have finished the updating. Since the computations for $\bar{x}_{i+1}^{(1)}$ and $\bar{x}_{i+1}^{(2)}$ involve the same amount of work (here we do not assume any sparsity structure on the matrix A), one might be tempted to conclude that this is the best scheme using two processors. This is not necessarily true! Note that, though the amounts of work for computing $\bar{x}_{i+1}^{(1)}$ and for $\bar{x}_{i+1}^{(2)}$ are the same, the actual times for computing them could still differ significantly due to various reasons as discussed in Section 2.2. Thus, it is possible that the penalty factor of synchronizing two processes at the end of each iteration is very large, which is certainly undesirable. One general solution to this kind of problem will be introduced in Section 5.

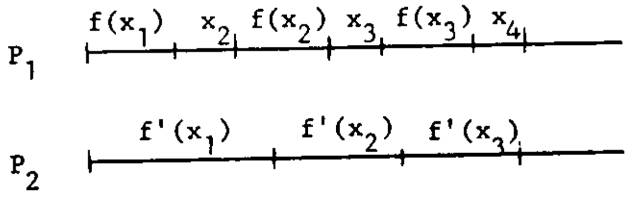
4.2 Asynchronous Iterative Algorithms

Asynchronous iterative algorithms are parallel iterative algorithms consisting of parallel processes which are not synchronized at any time. In particular, by removing all the synchronization restrictions from a synchronized iterative algorithm an asynchronous iterative algorithm will be obtained. For illustration, we first show an asynchronous iterative algorithm corresponding to the Newton iteration (4.1). It is convenient to suppose that each iteration step updates the three variables $f(x)$, $f'(x)$, x, rather than x alone. For example, after the iteration step (4.1), $f(x_{i-1})$, $f'(x_{i-1})$, x_i are updated as $f(x_i)$, $f'(x_i)$, x_{i+1}. Suppose that the evaluation of f' is more expensive than that of f. Then a reasonable asynchronous iterative algorithm consisting of two processes P_1 and P_2 can be defined as follows. Let v_1, v_2, v_3 be global variables which are accessible to both processes and contain the current values of $f(x)$, $f'(x)$, x, respectively. In the algorithm, v_1, v_3 are updated by P_1 and v_2 by P_2 in parallel. More precisely, processes P_1 and P_2 are controlled by the following programs:

```
process P_1
begin
    while condition S is not satisfied do
        begin
              $v_1 \leftarrow f(v_3)$;
(4.5)         $v_3 \leftarrow v_3 - v_2^{-1} v_1$
        end;
end

process P_2
begin
    while condition S is not satisfied do
        $v_2 \leftarrow f'(v_3)$
end
```

(In this paper, Condition S always stands for some global criterion for stopping a process.) Thus, as soon as a process finishes updating a global variable, it starts the next updating by using the current values of the relevant variables, without any delay. Suppose that the iterates are labeled in the order they are computed by step (4.5) of process P_1. Then in general they do not satisfy the recurrence (4.1). For example, suppose that $v_1 = f(x_0)$, $v_2 = f'(x_0)$, $v_3 = x_1$ are given initially and that the time lines of the processes are as follows:



where the subdivisions on each line give the sequence of tasks executed by the corresponding process. Then

$$x_2 = x_1 - f'(x_0)^{-1}f(x_1),$$
$$x_3 = x_2 - f'(x_1)^{-1}f(x_2),$$
$$x_4 = x_3 - f'(x_2)^{-1}f(x_3).$$

In general, we have

$$(4.6) \quad x_{i+1} = x_i - f'(x_j)^{-1}f(x_i) \quad \text{where } j \leq i.$$

Hence the iterates generated by the asynchronous iterative algorithm are different from those generated by the sequential algorithm or synchronized iterative algorithms. It seems difficult to derive any general theory of the properties of the sequence $\{x_i\}$.

To design an asynchronous iterative algorithm for a general iterative process (4.3), we first identify some variables $v_1,\ldots,v_m$ such that each iterative step can be regarded as computing the new values of the v_i's from their old values. Generally speaking, it is desirable to choose the v_i's such that the updating of each v_i constitutes a significant portion of the work involved in one iteration. For the Newton iteration (4.1), $\{v_1,v_2,v_3\} = \{f(x),f'(x),x\}$ seems to be a good choice for the v_i's. For the matrix iteration (4.2), v_i's may be chosen as segments of equal size of the components in a vector iterate. After the v_i's have been chosen, concurrent processes which update the v_i's asynchronously can be defined as follows. Note that a process can be specified by a permutation on some subset of $\{v_1,\ldots,v_m\}$ in the following sense: the process updates the v_i's in the subset sequentially according to the sequence which defines the permutation. Suppose that a permutation on a set of size s is represented by an s-tuple. Then, for example, the previous asynchronous iterative algorithm corresponding to the Newton iteration is defined by the processes specified by the permutations (v_1, v_3) and (v_2). Hence in general we define a process as an s-tuple over the set $\{v_1, \ldots, v_m\}$ for some s, and an asynchronous iterative algorithm is a collection of such processes which work asynchronously and have the property that each v_i appears in at least one of the tuples associated with the processes. The latter restriction guarantees that every v_i is taken care of by at least one process.

One can easily see that an unlimited number of asynchronous iterative algorithms can be constructed, even based on a simple iteration such as the Newton iteration. The problem is how to choose an algorithm. Since the iterates generated by an asynchronous iterative algorithm in general do not satisfy any recurrence such as (4.3), it is difficult to obtain a general theory concerning conditions for convergence or the speed of convergence. Perhaps a more fruitful approach here is to run experiments on multiprocessors, or on simulated multiprocessors as done in Rosenfeld and Driscoll [68]. Gerard Baudet of Carnegie-Mellon University has done experiments on C.mmp for solving the Dirichlet problem. It is found that the formulas developed from the observed results can help us to predict with reasonable accuracy the performance of certain asynchronous algorithms. The experiments also show the superiority in speed of some asynchronous iterative algorithms over the synchronized counterparts. The advantage of asynchronous iterative algorithms is that processes are never blocked and the overheads due to the execution of synchronization primitives are avoided. It seems that in practice those asynchronous iterative algorithms which are carefully chosen can be very competitive to the best synchronized iterative algorithms. Research on the performance of asynchronous iterative algorithms is of most interest.

4.3 Simple Asynchronous Iterative Algorithms

In this section we give parallel algorithms for speeding up the iterative process (4.3), which do not use any parallelism inside the iteration function φ. The algorithms are derived purely from the second strategy mentioned in the beginning of the section. We shall examine how much we can gain in speed by making use of fluctuations in the evaluation time.

Consider the asynchronous parallel algorithm which consists of k identical processes $P_1,\ldots,P_k$ each of which evaluates the iteration function φ by using the most recent iterates available at the time when the evaluation starts. To be more precise, P_ℓ is controlled by the following program, where the i and x_i are global variables and the variable j is local to the process.

```
process P_ℓ
begin
     while condition S is not satisfied do
          begin
               j ← i+1;
               x_j ← φ(x_{j-1}, x_{j-2}, ..., x_{j-d});
(4.7)          if i < j then i ← j
          end;
end
```

It is our intention that at any time the value of variable i will be the index of the iterate which was most recently computed. To achieve this, statement (4.7) is assumed to be programmed as a critical section. The remarkable thing about the algorithm just described is that it always generates the same sequence of iterates as the sequential algorithm does, no matter what φ is. Such an algorithm is called a simple asynchronous iterative algorithm. In the following, we illustrate some properties of the algorithm. For simplicity, we shall only take into account the time needed for evaluating the iteration function φ and assume that the algorithm consists of only two processes. The time lines of the processes in Figure 4.1 illustrate a possible outcome by executing the algorithm for the iteration, say, $x_{i+1} = \varphi(x_i, x_{i-1})$ starting from x_0, x_{-1}:

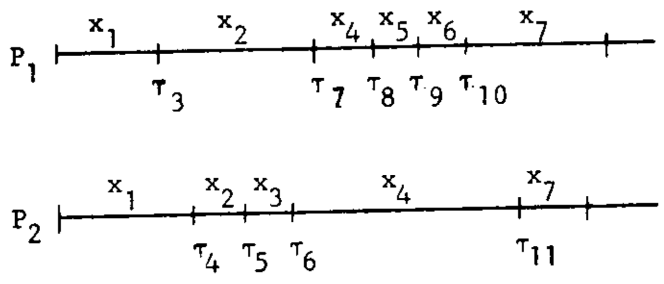


Figure 4.1

In the figure, τ_i denotes the time when the ith evaluation starts, and the iterate computed by a process at any time inside a time interval is shown above the interval. One should note that by the time P_1 completes its computation for x_2, x_3 is already computed by P_2. Thus, when P_1 completes x_2, it starts to compute x_4 by using x_2 and x_3. This means that the computation for x_3 is skipped by P_1. Similarly, the computations for x_5 and x_6 are skipped by P_2. After both processes have each completed five evaluations, iterate x_7 rather than x_5 is computed. A speed up has been achieved! Note that at any time at most one process is doing useful computation. Thus the speed up is not achieved by sharing work in two processes, but is achieved by taking advantage of the fluctuations in the evaluation time.

Let u_i be the index of the iterate computed by the ith evaluation. Then the iterate computed by the evaluation starting at time τ_i is x_{u_i}. For example, in Figure 4.1, we have $u_3 = 2$, $u_4 = 2$, $u_5 = 3$, $u_6 = 4$, $u_7 = 4$, $u_8 = 5$, $u_9 = 6$, $u_{10} = 7$ and $u_{11} = 7$. We observe that

$$u_i = \begin{cases} u_{i-1} + 1 & \text{for } i = 5, 6, 8, 9, 10, \\ u_{i-2} + 1 & \text{for } i = 7, 11. \end{cases}$$

It turns out that $u_i = u_{i-1} + 1$ should be used whenever the evaluations starting at times τ_{i-1} and τ_i are executed by the same process; otherwise, $u_i = u_{i-2} + 1$ should be used. Let p be the probability that two consecutive evaluations are executed by one process. (Note that the same p was used in Section 3.2.) Then we have

$$(4.8) \qquad u_i = \begin{cases} u_{i-1} + 1 & \text{with probability } p, \\ u_{i-2} + 1 & \text{with probability } 1-p. \end{cases}$$

It follows that the expected value of u_i is

$$\bar{u}_i = \frac{i}{2-p} + O(1).$$

For computing x_n, we expect to evaluate [illegible] j times such that $\bar{u}_j = n$. This implies that $j \sim (2-p)n$. For large n, $(2-p)n/2$ evaluations are expected to be executed by each process. Hence the speed-up ratio between the expected time taken by the sequential algorithm and that by the simple asynchronous iterative algorithm with two processes is

$$S_2 = \frac{n}{\frac{(2-p)n}{2}} = \frac{2}{2-p},$$

as $n \to \infty$ (here the blocked time and the execution time of synchronization primitives due to critical sections are ignored). Note that S_2 increases as p increases. When $p = 0$, i.e., when the τ_1's on the time lines of the two processes are interleaved, the algorithm has its worst performance. Suppose that we are given the probability density function of the random variable t, which is the time needed for one evaluation of the iteration φ by one process. In Baudet, Brent and Kung [76] a closed form for computing p is derived. Some of the results reported there concerning the speed-up factor are summarized in the following:

(i) If t may be approximated by a random variable normally distributed with mean $\bar{t}$ and standard deviation σ, then $S_2 = \frac{2}{2-(1/\sqrt{\pi})(\sigma/\bar{t})}$.

(ii) If t is exponentially distributed, then $S_2 = 4/3$,

(iii) If t is Erlang-2 distributed, then $S_2 = 16/13$.

(iv) If t is uniformly distributed in the interval (a,b), then

$$S_2 = \frac{6(b+a)}{5b+7a}$$

which approaches its maximum 1.2 as $\frac{b-a}{a} \to \infty$.

(v) If a simple asynchronous iterative algorithm with k processes is used and if t is exponentially distributed, then the speed-up factor is $\sqrt{2/\pi} \cdot \sqrt{k} \doteq .798\sqrt{k}$.

The main advantage of simple asynchronous iterative algorithms is their general applicability. The algorithms are not restricted to numerical iterative processes only. They can be employed to speed up any sequence of tasks. The algorithms become particularly attractive when the decomposition of the tasks is difficult. There are, however, some disadvantages. First, we note that critical sections are needed in the algorithms. Second, it seems that unless fluctuations in computation time due to the system are large and $\sigma/\bar{t}$ is large, the speed up of the algorithms is quite limited. See Baudet, Brent and Kung [76] for further results on this.

4.4 Semi-Synchronized (or Semi-Asynchronous) Iterative Algorithms

Synchronized iterative algorithms (cf. Section 4.1) have the drawback that processes may be blocked, and general asynchronous iterative algorithms (cf. Section 4.2) have the drawback that the analysis of the algorithms seems to be extremely difficult. Here we are interested in iterative algorithms which are compromises between the two types of algorithms. In general such algorithms can be constructed by making use of the special features of individual iterations. In the following, we illustrate an idea along this line by considering iterations (4.1) and (4.2).

Consider first the asynchronous iterative algorithm corresponding to the Newton iteration (4.1). We may impose a condition that $i-j < b$ on (4.6) for all i,j. This condition implies that no update uses a value of f' at an iterate which was produced by an update more than b steps previously. Using this fact and the standard techniques of iteration theory, it is possible to deduce properties of the sequence of the iterates such as its order of convergence. However, to enforce the condition $i-j < b$, it is necessary to synchronize processes P_1 and P_2, which are defined in Section 4.2. Note that the "strictly" synchronized iterative algorithm considered in Section 4.1 corresponds to this scheme with $b = 1$. Thus, if $b > 1$, then the chance that some process is blocked in this scheme is not as big as if $b = 1$. Furthermore, it is well-known that in a Newton-like iterative process, there is no need to update the value of f' very often. Therefore, the scheme produces iterates which are guaranteed to have favorable rates of convergence without paying the excessive synchronization penalty which might be found in the strictly synchronized iterative algorithm. The optimal choice of b depends on the relative speeds of P_1 and P_2. Its analysis will be reported in a future paper. Clearly, this idea is also applicable to the matrix iteration (4.2). The resulting scheme is called "chaotic iterative scheme" by Chazan and Miranker [60]. In their paper, conditions guaranteeing convergence of the scheme are given.

In the following, we consider another semi-synchronized iterative algorithm, based on a different idea. In practice, band linear systems $Bx = b$ are often solved by Gauss-Seidel's method. Unlike Jacobi's method, Gauss-Seidel's method seems inherently sequential. The components of its iterates cannot be computed in parallel, since they are logically dependent upon each other. Note, however, that if d is the band width of the matrix B, then components whose indices differ by more than d-1 are in fact independent. Thus, for solving $Bx = b$ by Gauss-Seidel's method we may use a parallel algorithm, in which each process performs the sequential Gauss-Seidel iteration, but the processes are synchronized so that components whose indices differ by less than d are not allowed to be updated simultaneously. In other words, the synchronization ensures that one process will not follow "closely" on the heels of another. In case the size of the linear system is much larger than d and than the number of processes, we may expect that the processes will not be blocked very often. Note that the iterates generated by this scheme are exactly the ones generated by the sequential Gauss-Seidel's method.

The main characteristics of a semi-synchronized iterative algorithm can be summarized as follows.

(i) The algorithm is "loosely" synchronized so that processes are not expected to be blocked very often.

(ii) The synchronization, however, guarantees that the iterates generated by the algorithm satisfy some desirable properties.

A semi-synchronized iterative algorithm is attractive because it reduces the drawback of synchronized iterative algorithms by its first characteristic and also the drawback of asynchronous iterative algorithms by its second characteristic.

5. The Third Example: Adaptive Asynchronous Algorithms

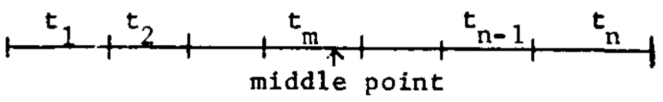
We consider the problem of executing n independent tasks $J_1,\ldots,J_n$ by parallel algorithms. Let the execution time of the task J_i by one process be a random variable t_i. We assume that all the tasks are of the same complexity, i.e., $t_1,\ldots,t_n$ are identically distributed random variables, say, with mean $\bar{t}$ and standard deviation σ. For example, J_i may be the task of updating the ith component of an iterate in the synchronized iterative algorithm for performing the matrix iteration (4.2) (cf. Section 4.1). For simplicity, we mainly consider parallel algorithms with two processes.

The first algorithm is the obvious one which performs the first n/2 tasks by one process and the second n/2 by the other process. Thus the time taken by the algorithm is the random variable $T_1 = \max(t_1 + \dots + t_{n/2}, t_{(n/2)+1} + \dots + t_n)$. We wish to compute the mean $\bar{T}_1$ of T_1. By the central limit theorem, as $n \to \infty$ the distribution of $t_1 + \dots + t_{n/2}$ or $t_{(n/2)+1} + \dots + t_n$ approaches to the normal distribution with mean $(n/2)\bar{t}$ and standard deviation $(\sqrt{n/2})\sigma$. Using a result from order statistics for normally distributed random variables (see, e.g., Gibbons [71, p.34]), we obtain that

$$(5.1) \quad \bar{T}_1 \sim \frac{\bar{t}}{2} \cdot n + \frac{\sigma}{\sqrt{2\pi}} \cdot \sqrt{n}$$

as $n \to \infty$.

In the second algorithm, the list of tasks is made into a global deque ("double-ended queue", Knuth [69]), which is accessible to both processes of the algorithm. One process is allowed to remove tasks from only one end of the deque. In the algorithm, each process repeats the following until the deque becomes empty: remove a task from one end of the deque and execute it. Observe that the finishing times of the two processes can differ at most by the execution time of the task which finishes last. Thus, the algorithm is expected to be efficient, since the time that only one process is active is small. Indeed, the following analysis supports this argument. Consider the time line of the n tasks:



Note that the execution time of the task which finishes last is the length t_m of subinterval containing the middle point, since the algorithm works from both ends toward the middle.

By a result from renewal theory (see, for example, Kleinrock [75]), we know that the expected value of t_m is $\bar{t} + \frac{\sigma^2}{\bar{t}}$ as $n \to \infty$. Hence if the time taken by the algorithm is a random variable T_2, then its mean $\bar{T}_2$ satisfies

$$(5.2) \quad \bar{T}_2 \le \frac{\bar{t}}{2} \cdot n + \frac{\bar{t}}{2} + \frac{\sigma^2}{2\bar{t}} .$$

Comparing (5.1) and (5.2) we conclude that the mean time of the second algorithm is less than that of the first one when n is large.

The implementation of the second algorithm, which uses a deque, is of interest. For example, we can use the following programs to control the two processes P_1 and P_2, where i,j are global variables and initially $i = 1$ and $j = n$.

```
process P_1
begin
     while i < j do
          begin
               execute task J_i;
               i ← i+1
          end;
     execute task J_i
end

process P_2
begin
     while i < j do
          begin
               execute task J_j;
               j ← j-1
          end;
end
```

It is not difficult to check that all the tasks $J_1,\ldots,J_n$, except the one which finishes last, will be executed exactly once. (There is a chance that the task which finishes last is executed by both processes. We do not regard this as a serious drawback.) The point we want to make here is that the second algorithm can be implemented without using critical sections. The reason that we can achieve this is mainly due to the fact that only one process is allowed to operate at each end of the deque. (A critical section is needed if more than one process operates at one end of a list.) Hence the second algorithm essentially does not involve more overheads than the first one. If tasks were removed from only one end of the list as in the case where a queue or a stack is used, then the extra overheads due to critical sections would be involved.

The use of a deque could be advantageous even in parallel algorithms with more than two processes. We can let, say, half of the processes obtain their tasks from one end of the deque, and the other half from the other end. This is better than the scheme where all the processes obtain their tasks from only one end of the list. The reason is that the less processes operating at an end of a list, the less chance there is that processes are blocked from entering critical sections. However, if the tasks are of various complexities, it is often desirable to perform the tasks in the order of decreasing complexities, in order to reduce the difference in the finishing times of the processes. Then in this case a priority queue is more appropriate than a deque. Moreover, we note that the number of times that critical sections are executed can be reduced by letting processes take more than one task from the list at a time. However, in this case, the difference in the finishing times of processes will increase.

Careful analysis on various techniques mentioned above will be reported elsewhere.

The tasks performed by a particular process in the second algorithm are not specified a priori but depend upon the relative speeds of the two processes. Thus, it is an adaptive algorithm. The first algorithm is not adaptive because it assigns tasks to processes statically. The efficiency of an adaptive algorithm is obtained from the fact that the processes are able to adjust themselves during the computation so that they can all finish in about the same time. The concept of adaptive algorithms seems to be fundamental to the design of many efficient asynchronous algorithms. For example, two ordered files can be merged by two asynchronous processes in the following way: One process merges from left to right and the other one from right to left, until one of the files is exhausted. The two subfiles merged by a particular process are unpredictable; they depend on the relative speeds of the processes and the orderings in the original files. Note that in this example, two deques are needed; one for each file.

6. Asynchronous Algorithms Where Processes can be Interrupted

The speed of an asynchronous algorithm may be improved if those processes which are not doing useful computations can be interrupted promptly and if the extra cost due to interruption is not excessive. Note that whether or not a process is doing useful computation at a given time may be determined by examining the current contents of some of the global variables. Hence interruptions may be implemented by letting the process check the new status of those variables as soon as it realizes that the value of some variable has been modified by some process. The ability of a process to be interrupted certainly causes overheads in the time taken by the process. However, for a given asynchronous algorithm and a given interruption scheme, the overheads usually can be estimated. Hence in this case it is possible to decide whether we should allow processes to be interrupted in the algorithm. In the following we briefly study two examples.

First consider Algorithm AZ_2 in Section 3.2. It is clear that at state $A_1(\ell)$ the process which is evaluating the point outside the interval of uncertainty will not lead to any useful information. Thus, the process could be interrupted and start a new evaluation at an appropriate point in the interval, resulting in state $A_2(\ell)$. Suppose that we do so. Then, only states of type $A_2(\ell)$ will ever occur. They satisfy the transition rule:

$$A_2(\ell) \rightarrow A_2(\theta\ell) \vee A_2(\theta^2\ell).$$

At any state, two evaluations are performed simultaneously. The time that the algorithm is at the state is bounded above by the time taken by the evaluation which finishes first. Because the interruption facility introduces extra overheads, the evaluation time by one process is a random variable t', which is greater than t. Hence the expected time that the algorithm is at a state is bounded above by the mean $\bar{T}'$ of the random variable $T' = \min(t_1', t_2')$, where t_1' and t_2' are independent and identically distributed random variables satisfying the same distribution function as t'. Let $M(\ell)$ be the number of state transitions the algorithm encounters, if it starts from state $A_2(\ell)$. Then the expected time of the algorithm is at most $M(L) \cdot \bar{T}'$. Since, if t' is given $\bar{T}'$ can always be computed at least numerically, we assume $\bar{T}'$ is known.

It remains to compute $M(L)$. Assume that $L \to \infty$. It is easily seen that $M(L) \sim \log_\phi L$ in the worst case and $M(L) \sim (\log_\phi L)/2$ in the best case. On the average, we have the following (notation in Section 3.2 is assumed).

Case 1:

$$M(L) = \theta M(\theta L) + \theta^2 M(\theta^2 L) + 1. \text{ Thus}$$

$$M(L) \sim \frac{1}{1+\theta^2} \log_\phi L \doteq .276 \log_\phi L.$$

Interruptions should be used when $.276(\log_\phi L)\bar{T}' < \frac{a(p)}{2} \cdot (\log_\phi L)\bar{t}$, i.e., $\bar{T}' < 1.81\, a(p) \cdot \bar{t}$.

Case 2:

$$M(L) = \frac{1}{2} M(\theta L) + \frac{1}{2} M(\theta^2 L) + 1. \text{ Thus}$$

$$M(L) \sim \frac{2}{3} \log_\phi L.$$

Interruptions should be used when $\frac{2}{3}(\log_\phi L)\bar{T}' < \frac{a(p)}{2}(\log_\phi L)\bar{t}$, i.e., $\bar{T}' < \frac{3}{4}\, a(p)\bar{t}$.

As our second example, we consider a simple asynchronous iterative algorithm with two processes which is defined in Section 4.3. We observed that at any time at most one process is doing useful computation. Thus, it is natural to consider interruptions here. Assume that interruptions are allowed. Let the evaluation time of the iteration function by one process be a random variable t', which is presumably larger than t. Then the expected time of computing a new iteration by the algorithm is the mean $\bar{T}'$ of the random variable $T' = \min(t_1', t_2')$, where the t_i' are independent and identically distributed random variables satisfying the same distribution function as t. For computing x_n, the expected time is $n\bar{T}'$. Hence, interruption should be used when $\bar{T}' < \frac{2-p}{2}\bar{t}$.

We observe that the minimum of a number of random variables is concerned in the case where processes can be intertupted. This is contrary to the case for synchronized algorithms where the maximum of a number of random variables should be considered. Hence large fluctuations in process speed in fact will often reduce the time taken by an asynchronous algorithm if processes can be interrupted.

7. On the Optimal Number of Processes One Should Create

To perform a given task on a multiprocessor, one has to decide how many processes should be created. Some considerations on choosing the optimal number of processes are given in this section.

Consider synchronized parallel algorithms first. Note that the execution of synchronization primitives is usually time consuming and that the penalty factor tends to increase as the number of synchronized processes increases. Hence those synchronized parallel algorithms which are based on the maximal decomposition of a given problem may not be desirable. For example, suppose that we have three tasks J_1, J_2, J_3, where J_1, J_2 have to be completed before J_3 is allowed to start. Assume that the time needed for J_1 or J_2 is approximated by a normally distributed random variable with mean $\bar{t}$ and standard deviation σ and that for J_3 is another random variable with mean $\bar{t}$. Suppose that the time of executing the synchronization primitives needed for synchronizing two tasks is s. Consider the following two methods of performing the three tasks.

The first method is based on the maximal decomposition principle. J_1 and J_2 are done in parallel, and when both have finished, J_3 starts. Hence, the expected time needed by the method is

$$\bar{t} + \frac{\sigma}{\sqrt{\pi}} + s + \bar{t}'.$$

(See Section 5.)

The second method is the obvious method which performs J_1, J_2, J_3 sequentially. The expected time needed by the method is $2\bar{t} + \bar{t}'$. Hence when $s > \bar{t} - \frac{\sigma}{\sqrt{\pi}}$ the sequential algorithm should be used even if more than one processor is available. This example shows that the maximal decomposition of a problem may not necessarily lead to the optimal number of processes that should be created for solving the problem.

Now consider asynchronous parallel algorithms. As noted in the preceding sections, critical sections are needed in many asynchronous algorithms. In these algorithms, processes may be blocked from entering critical sections, so the performance of the algorithms is degraded. The amount of degradation can be estimated as follows. Consider an asynchronous algorithm with k processes, $P_1,\ldots,P_k$. Let T_i be the total time taken by process P_i and C_i the time spent in the critical sections of the process, under the assumption that the process is never blocked. Define $\alpha_i = C_i/T_i$ and let α be a lower bound on the $\alpha_1,\ldots,\alpha_k$. In general, an estimate on α can be obtained by examining the programs of the processes. α may or may not be a function of k. For example, in the asynchronous zero-searching algorithms considered in Section 3, each process has to update the global variables within critical sections. As the number k of concurrent processes increases, so does the number of possible states. This implies that the complexity of updating the global variables grows as k increases. Thus, α is an increasing function of

k. As another example, we consider simple asynchronous iterative algorithms defined in Section 4.3. In this case, we may take $\alpha = s/\bar{t}$ where s is the execution time of the synchronization primitives needed in implementing critical sections and $\bar{t}$ is the evaluation time of the iteration function. Here α is independent of k. At any rate, we shall write $\alpha(k)$ for α. Note that the executions of critical sections in the parallel algorithm cannot be overlapped. Hence the speed-up factor of the algorithm is at most

$$\frac{T_1+\ldots+T_k}{C_1+\ldots+C_k} \le \max\left(\frac{1}{\alpha_1},\ldots,\frac{1}{\alpha_k}\right) = \frac{1}{\alpha(k)}$$

It is trivial that the speed-up factor is also bounded by k. Therefore, an optimal choice of k exists, which is, in fact, bounded above by the smallest positive solution of the equation $k = 1/\alpha(k)$. The above arguments indicate that a large number of processes in an asynchronous algorithm cannot help unless α can be kept small. In practice, it is important to design algorithms which use small critical sections and to select the synchronization tool which takes as little time as possible.

How to find out the optimal number of processes in a synchronized or asynchronous parallel algorithm for performing a given task is a real problem in multiprocessor programming. The problem, however, does not have easy solutions, as we have seen in this section. The good choice generally would require a rather involved analysis.

8. Summary and Conclusions

A parallel algorithm is viewed as a collection of concurrent processes. To ensure that the algorithm works correctly and uses the parallelism effectively, processes must communicate with each other. However, due to various reasons as stated in Section 2.2, the speed of a process is unpredictable. Thus, one can never be sure that an input needed by one process will be produced in time by another process. There are two approaches for solving the problem. The first one is to synchronize processes so that they wait for inputs whenever necessary. This results in a synchronized parallel algorithm. The second approach is to let processes continue or terminate according to the information currently contained in some global variables, so processes never wait for inputs. This results in an asynchronous parallel algorithm. Several examples of the two types of parallel algorithm are considered in the paper. It is hoped that through these examples important features of each type of parallel algorithms can be identified. Some of them are summarized and discussed in the following.

In a synchronized algorithm, a task is decomposed into subtasks, which, hopefully, are of the same size, so that each subtask is solved by one process of the algorithm. Processes are synchronized at interaction points. At those points processes may be blocked while waiting for inputs. The loss due to waiting may be captured by the penalty factor defined in Section 2.3. The penalty factor increases as the number of synchronized processes increases. Hence synchronized algorithms should be used when the fluctuations in process speed are small and when there are only few processes to be synchronized. Furthermore, the execution time of the needed synchronization primitives is usually non-negligible. Thus, it is not always advantageous to create as many processes as possible according to the maximal decomposition of a task (cf. Section 7). In general, the analysis of a synchronized algorithm is not too much different from that of its sequential counterpart, except that techniques of order statistics may be needed in analyzing the time taken by the synchronized algorithm.

Asynchronous parallel algorithms arise naturally in the use of multiprocessors, where the processors are not synchronized and communication between cooperating processors is by means of shared data. When the fluctuations in computation time are large, asynchronous algorithms are in general more efficient than synchronized ones for the following three reasons. First, the processes never waste time in waiting for inputs. Second, the algorithms can take advantage of processes which are run fast. Results produced by those processes can be immediately used. In particular, by making use of these results those "slow" processes which are doing useless computations may be discovered and aborted at early times (cf. Section 7). Third, the algorithms are "adaptive", so the processes can finish at about the same time (cf. Section 6). This guarantees that the maximal parallelism is used during most computation times. Furthermore, we note that in general, asynchronous algorithms are more reliable than synchronized algorithms in the following sense. Even if some processes are blocked forever, an asynchronous algorithm may still continue computing the solution of its problem, as long as no blocking occurs in critical sections, which are presumably small, and there remains at least one active process. (One may easily verify that, for example, Algorithm AZ_2 and simple asynchronous iterative algorithms indeed have this nice reliability property.) For solving a given problem, it is almost always possible to construct a large number of asynchronous algorithms (cf. Section 4.2.). However, the analysis of an asynchronous algorithm seems to be always non-trivial. But if an asynchronous algorithm is defined by few simple state transition rules such as (3.1), (3.2) and (4.8), then it can be analyzed. How to construct asynchronous algorithms which involve simple state transitions is an interesting and challenging task for many problems.

A promising direction is to design semi-synchronized (or semi-asynchronous) algorithms which are compromises between synchronized and asynchronous algorithms, and which may take advantage of the special features of individual problems (cf. Section 4.4).

One of the motivations for analyzing multiprocessor algorithms is to determine how many processes should be created for solving a problem. In the analysis it is crucial to include overheads due to the execution of synchronization primitives and critical sections (cf. Section 7.3). In practice, programming techniques, such as the use of deques as described in Section 5 and the selection of synchronization tools which take as little time as possible, are often important to the performance of algorithms. However, if the existence of some indivisible operations such as "add to store" $\{x \leftarrow x + 1\}$ and "swap (x,local)" is assumed, then many problems due to critical sections can be eliminated (see Dijkstra [72]).

In view of the parallel algorithms considered in this paper, it is found that there are tradeoffs among the basic processing time, blocking time and synchronization time of a process (cf. Section 2.5) in the following sense. In order to reduce one quantity, it is often necessary to increase one or two other quantities. For example, processes in a synchronized algorithm generally have smaller basic processing times but larger blocked times than those in its asynchronous counterpart. It is of interest to build abstract models for studying these tradeoffs.

Acknowledgments

The author wishes to thank G. Baudet for many discussions. Helpful comments on a draft of paper were made by G. Baudet, J. Robinson, J. Traub, B. Weide and G. Yuval at Carnegie-Mellon University, H. Chang at the Institute for Advanced Computation, and R. Brent at the Australian National University.

References

Avriel and Wilde [66] Avriel, M. and D. J. Wilde, "Optimal Search for a Maximum with Sequences of Simultaneous Function Evaluations," Management Sci., 12, 1966, 722-731.

Baudet, Brent and Kung [76] Baudet, G., R. P. Brent and H. T. Kung, "Simple Asynchronous Iterative Algorithms for Multiprocessors," to appear.

Barnes, et al. [68] Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnik, R. A. Stoker, "The ILLIAC IV Computer," IEEE Trans. on Comp. 17, 1968, 746-757.

Chazan and Miranker [69] Chazan, D. and W. L. Miranker, "Chaotic Relaxation," Linear Algebra and Appl, 2, 1969, 207-217.

Dijkstra [68] Dijkstra, E. W., "Cooperating Sequential Processes," in Programming Languages (F. Genuys, ed.), Academic Press, New York, 1968, 43-112.

Dijkstra [72] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes," in Operating Systems Techniques (C. A. R. Hoare and R. H. Perrott, ed.), Academic Press, London, 1972, 72-93.

Flynn [66] Flynn, M. J., "Very High-Speed Computing Systems," Proc. IEEE 54, 1966, 1901-1909.

Gibbons [71] Gibbons, J. D., Nonparametric Statistical Inference, McGraw-Hill Book Co., New York, 1971.

Habermann [76] Habermann, A. N., Introduction to Operating System Design, Science Research Associates, Inc., Chicago, 1976.

Heller [76] Heller, D., A Survey of Parallel Algorithms in Numerical Linear Algebra, Carnegie-Mellon Department of Computer Science Report, 1976.

Hyafil and Kung [76] Hyafil, L. and H. T. Kung, "Search for Zeros and Maxima by Asynchronous Multiprocessors," to appear.

Karp and Miranker [68] Karp, R. M. and W. L. Miranker, "Parallel Minimax Search for a Maximum," J. Comb. Theory 4, 1968, 19-35.

Kiefer [53] Kiefer, J., "Sequential Minimax Search for a Maximum," Proc. Amer. Math. Soc. 4, 1953, 502-506.

Kleinrock [75] Kleinrock, L., Queueing Systems, Vol. 1: Theory, John Wiley and Sons. New Ycrk, 1975.

Knuth [69] Knuth, D. E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Massachusetts, 1969.

Kuck, David J., "Parallel Processor Architecture--A Survey," 1975 Sagamore Computer Conference on Parallel Processing, 1975, 15-39.

Rosenfeld and Driscoll [69] Rosenfeld, J. L. and G. C. Driscoll, "Solution of the Dirichlet Problem on a Simulated Parallel Processing System," Information Processing 68, North-Holland Publishing Co., Amsterdam, 1969, 499-507.

Thompson and Kung [76] Thompson, C. D. and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer," Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, 58-64. Also to appear in Communications of the ACM.

Wulf and Bell [72] Wulf, W. A. and C. G. Bell, "C.mmp - A Multi-Mini-Processor," Proc. AFIPS 1972 FJCC, Vol. 41, Part II, AFIPS Press, Montvale, N. J., 1972, 765-777