ISSN 0105-8517

CONCURRENT ALGORITHMS FOR ROOT SEARCHING

by

Ole Eriksen
and
Jørgen Staunstrup

DAIMI PB-147
July 1982

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

CONCURRENT ALGORITHMS FOR ROOT SEARCHING

June 1982

Ole Eriksen and Jørgen Staunstrup
Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Aarhus C

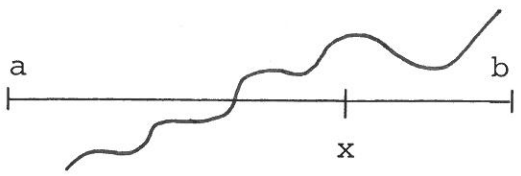
Abstract

Concurrent algorithms for finding the root of a real continous function are analyzed. A lower bound on the running time is given, this lower bound is obtained by a synchronous algorithm. A new asynchronous algorithm is discussed in detail and its running time is analyzed. Finally, the results from running the asynchronous algorithm on a multiprocessor are shown.

INTRODUCTION

Let H be a real continuous function defined on the closed interval [a,b] and assume that $H(a)\cdot H(b) \leq 0$, i.e. that H has at least one root in [a,b]. In this paper we describe algorithms for finding a root of H under the additional assumption that evaluating H is computationally very slow.

We consider the class of partitioning algorithms, which are iterative algorithms where each iteration reduces the current interval containing the root. Let $x \in [a, b]$; based on the function value, H(x), it can be decided which of the intervals [a, x] or [x, b] that contains the root, this interval becomes the the new current interval.

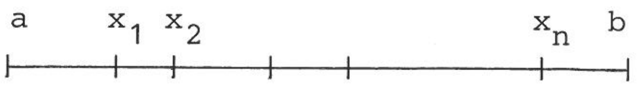


The well known bisection algorithm, where x is the middle of [a,b], is an example of a partitioning algorithm. We restrict ourselves to partitioning algorithms because they are simple and guarantee convergence without other assumptions on the function than continuity.

Let T_H be the time it takes to evaluate H. When T_H dominates other quantities the running time, B_T, for the bisection algorithm is:

$$B_T \approx T_H \cdot \log_2 \frac{b-a}{eps}$$

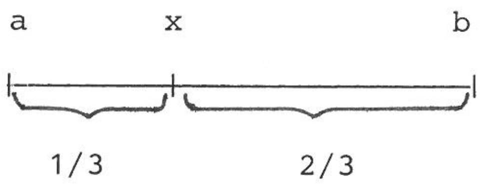
where eps is the absolute accuracy with which the root is obtained. This running time can be reduced by letting n processes evaluate H at different points $x_1, x_2, \ldots, x_n$ concurrently:



When a process, p_i, finishes its evaluation of $H(x_i)$, it can be decided which of the intervals $[a,x_i]$ or $[x_i,b]$ contains a root. This information must be communicated to the other processes. Based on its own results and those received from others, a process now selects a new point, x, in the interval and computes H(x). The running time of the algorithm depends on how the points $x_1,x_2,\ldots,x_n$ are placed in the interval. Intuitively, the n points should be spread out as evenly as possible, to reduce the worst case running time. Although this is very easy to achieve initially by dividing the interval into n+1 intervals of length $\frac{b-a}{n+1}$, it is difficult to maintain such an even spreading after a few iterations. In fact the remaining part of this paper is concerned with how to keep the points evenly spread.

2. A LOWER BOUND ON THE RUNNING TIME

In this section we assume that the time it takes to evaluate H, T_H is constant. Under this assumption we give a lower bound on the time it takes to locate a root using n or fewer processes. First, note that the number of function evaluations needed to locate the root with any partitioning algorithm may vary with the location of the root. Consider an algorithm which always partitions the current interval into two, in proportion 1:2.



Locating roots close to "a" is of course much faster than locating roots close to "b". When comparing algorithms we therefore compare the maximum time needed to locate the root when this varies between a and b. The best algorithm is the one with the minimal maximum, the minmax criteria.

Consider the following algorithm:

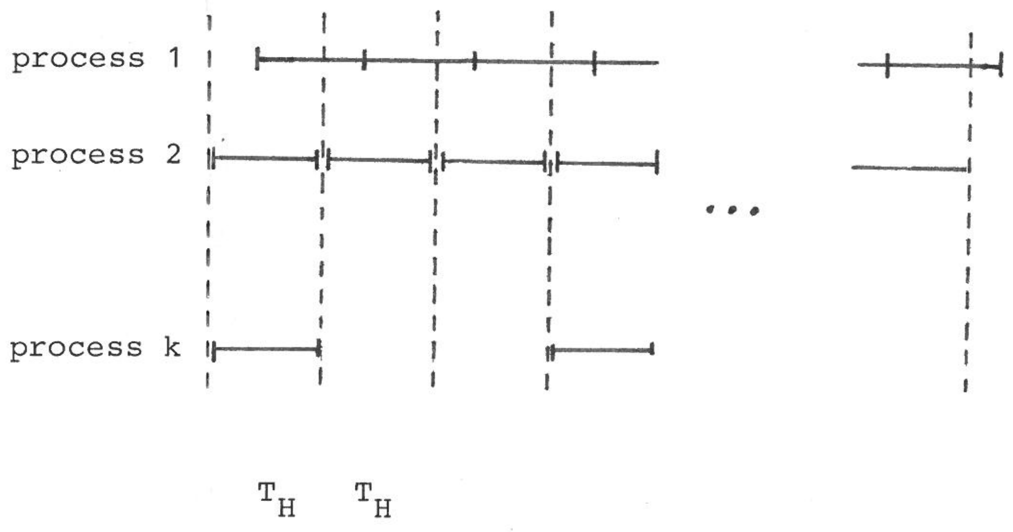
```
ha:= H(a); hb:= H(b);
repeat   " ha*hb <= 0 "
  cobegin
    $x_1$:= a + 1/(n+1)*(b-a); $y_1$:= H($x_1$);//
    $x_2$:= a + 2/(n+1)*(b-a); $y_2$:= H($x_2$);//
     ⋮
    $x_n$:= a + n/(n+1)*(b-a); $y_n$:= H($x_n$);
  coend;
  if
    ha*$y_1$ $\leq$ 0 → b := $x_1$; hb:= $y_1$;   /
    $y_1$*$y_2$ $\leq$ 0 → a := $x_1$; b := $x_2$; ha:= $y_1$; hb:= $y_2$;   /
     ⋮
    $y_i$*$y_{i+1}$ $\leq$ 0 → a := $x_i$; b := $x_{i+1}$; ha:= $y_i$; hb:= $y_{i+1}$; /
     ⋮
    $y_n$*hb $\leq$ 0 →  a:= $x_n$; ha:= $y_n$;
  end;
until (b-a) $\leq$ eps;
```

The cobegin $S_1 // S_2 // \ldots // S_n$ coend means concurrent execution of the statements $S_1, S_2, \ldots, S_n$. Hence in the above algorithm the function is evaluated concurrently on all n points $x_1, x_2, \ldots, x_n$.

When calculating the number of function evaluations made by an algorithm, the concurrently executed function evaluations are counted as one evaluation of H. It is not worthwhile to elaborate all details of the abstract machine model underlying this assumption. But we envisage a model with a number of processors each with a local store which can be accessed without disturbing the computation of other processors. Furthermore the processors can somehow communicate e.g. through a common store. In such a model, it is obvious that the running time of the above algorithm is:

$$T_n \approx \log_{n+1}(b-a/eps) \cdot T_H$$

Furthermore, when the evaluation time for H is constant $\log_{n+1}(b-a/eps) \cdot T_H$ is a lower bound (in the minmax sense) on the running time for any partitioning algorithm using n or fewer processes. To see this, first observe that the above shown algorithm is optimal among all synchronous algorithms. A synchronous algorithm is one where all n processes complete one iteration before any of them go on to the next iteration as it is the case with the above algorithm. Another synchronous algorithm could differ from the above by not letting the processes work on equidistantly placed points or by preventing some processes from using all the information obtained by the other processes. Both of these alternatives would clearly give a longer worst case running time. If the above shown algorithm is not optimal, there must be a faster algorithm where the n processes are not completely synchronized in each step. The execution of one such algorithm is illustrated below:

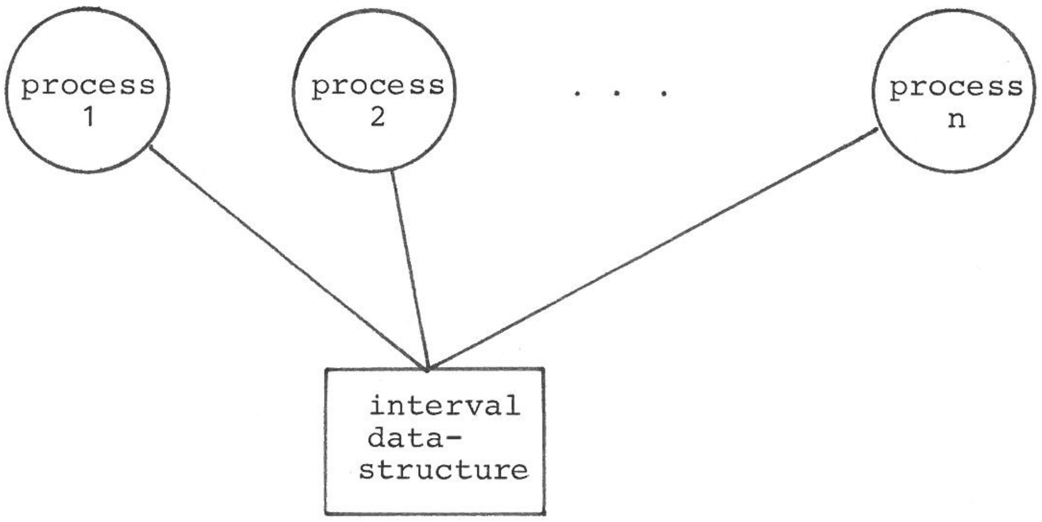


A partial evaluation of H does not yield any information which can be utilized by other processes. Therefore delaying an iteration, as illustrated by process 1 above, cannot lead to a better worst case running time. Hence given an algorithm without synchronization as the one above it can always be transformed into a synchronous algorithm having the same or a better worst case running time. This justifies the claim made above that $\log_{n+1}(b-a/eps) \cdot T_H$ is a lower bound on the running time for any partitioning algorithm.

When T_H is not constant, neither the above algorithm nor the lower bound are useful, since the complete synchronization of all processes is inefficient. Processes which finish their evaluation quickly are forced to wait for the remaining. In the next section we will therefore describe a class of <u>asynchronous</u> partitioning algorithms. The distinction between synchronous and asynchronous algorithms were first made by Kung [1976].

3. ASYNCHRONOUS PARTITIONING ALGORITHMS

In this section we consider asynchronous partitioning algorithms. Similarly to the synchronous algorithms all processes evaluate H on different points of the interval concurrently. But in contrast to the synchronous algorithm, when a process has finished its evaluation it does not wait for all other processes to finish their evaluation. Based on its evaluation and those evaluations from other processes which have been completed it makes a partitioning of the interval and proceeds with a new evaluation. The main difficulty is coordinating the partitionings caused by different processes in such a way that they all contribute to the common goal of decreasing the size of the interval. This coordination can be done through a datastructure representing the current interval which is shared by all processes.



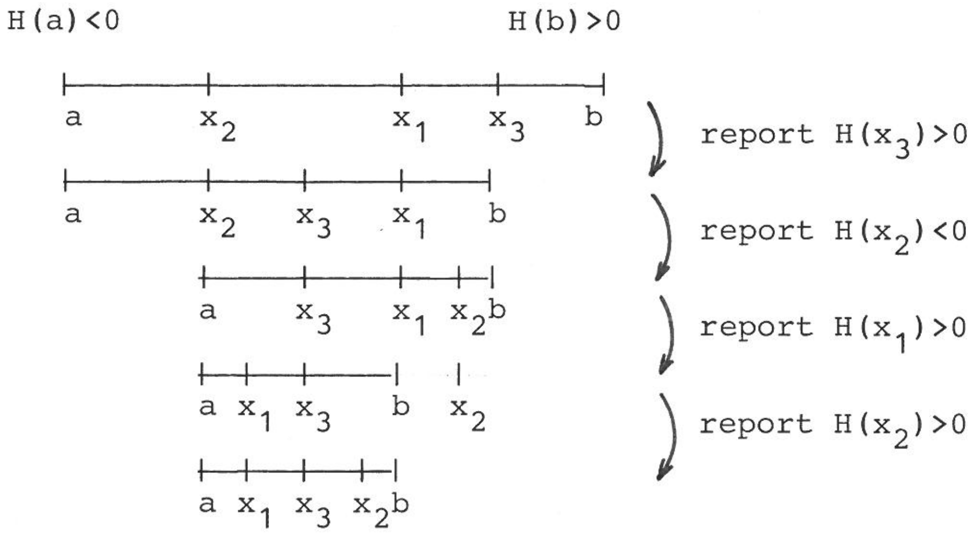
The datastructure is updated by indivisible operations to ensure its consistency. This is achieved by programming the interval as a monitor [Brinch Hansen 75]:

```
TYPE INTERVAL  =  MONITOR;

   TYPE  POINT = REAL;
         INTER = RECORD
                      A, B: POINT;
                        HA: REAL
                 END;

   VAR CURRENT: INTER;

  FUNCTION  LENGTH( T: INTER ): POINT;
  BEGIN LENGTH:= T.B - T.A   END;

  PROCEDURE ENTRY REPORT(VAR X: REAL; HX: REAL);
  VAR TEMP: INTER;

  BEGIN
     TEMP:= CURRENT;
     WITH  TEMP  DO
       IF HX*HA <= 0.0
          THEN B:= X
          ELSE BEGIN
                 A:= X; HA:= HX
               END;
     IF  LENGTH( TEMP ) < LENGTH( CURRENT )
       THEN CURRENT:=  TEMP;
  END;


  PROCEDURE  ENTRY  RECEIVE( VAR X: POINT );
  BEGIN
    X:=  ...  "SOME POINT IN [A, B]"
  END;

BEGIN
  CURRENT:= ....;
END;
```

The body of the evaluation processes are as follows:

```
cycle
   receive(x);
   y := H(x);
   report(y,x);
end
```

Below is shown a small fragment of the history of a computation with three processes, x_i is the point in which process "i" evaluates H:



Note, that for short periods one or more processes are left "working" outside the current interval. This only lasts until they have finished their current evaluation and call "receive".

In the interval monitor it is not specified which point in the interval a process gets by calling "receive". This allocation of interval points to processes has a strong influence on the running time of the algorithm and it is discussed in detail in the next section.

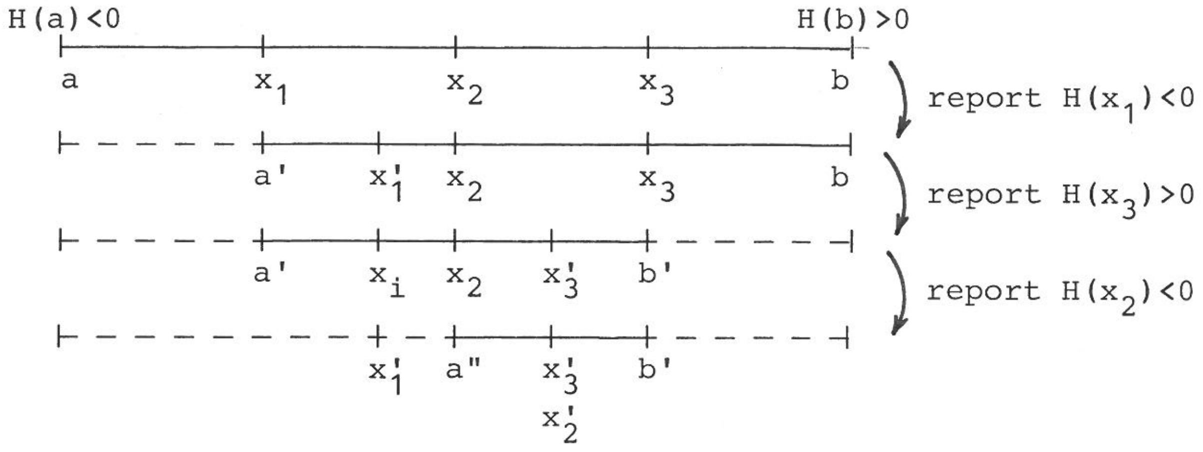
3.1 Equidistant Algorithm

In section 2 we showed that in the optimal synchronous algorithm, the processes always worked on equidistantly placed points. It seems obvious to choose the same allocation in the asynchronous algorithm. Then the procedure "receive" becomes:

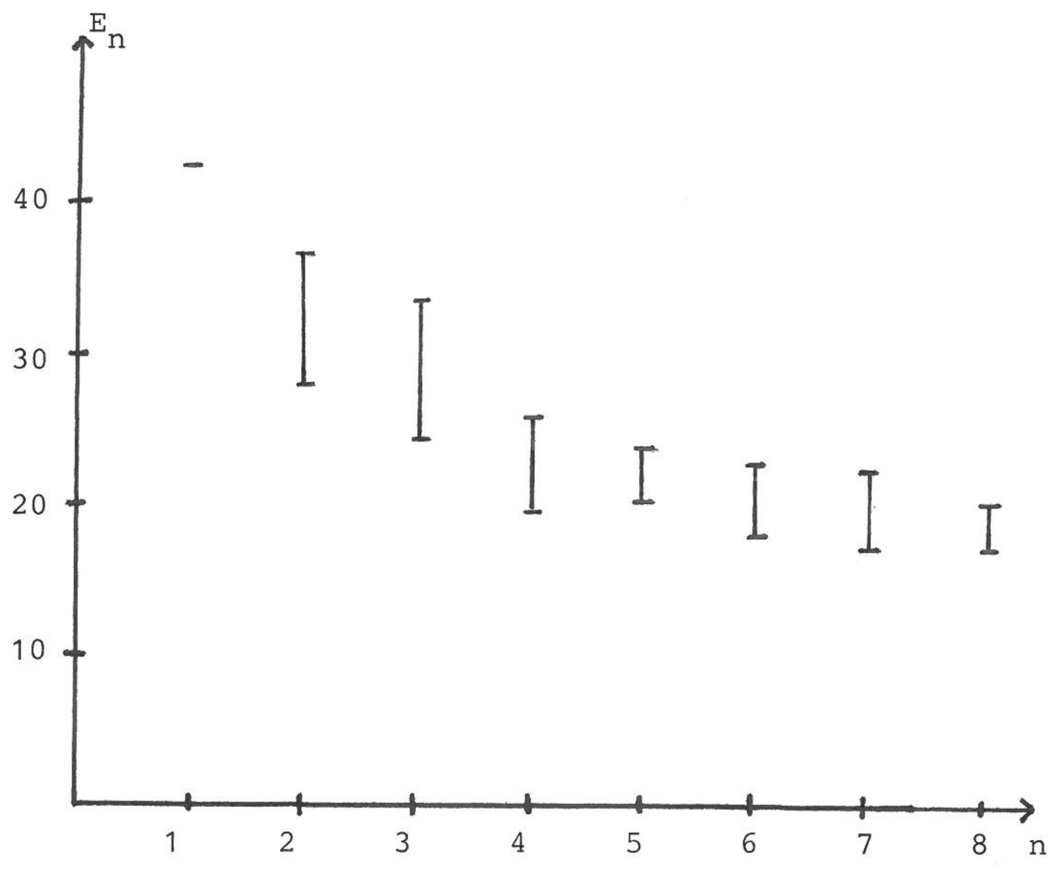
```
procedure entry receive (var x: point);
begin
  x := a + (me/(n+1))*(b-a);
end;
```

where me is a function returning the number of the calling process, and n is the number of processes.

Unfortunately, this very simple strategy gives a very poor performance. The reason is that the processes tend to cluster. Consider the following history where n=3.



The example shows how two processes may end up working in the same point. Since this happens frequently the equidistant allocation leads to many redundant function evaluations. The following diagram shows E_n the running time of the equidistant algorithm for various n.



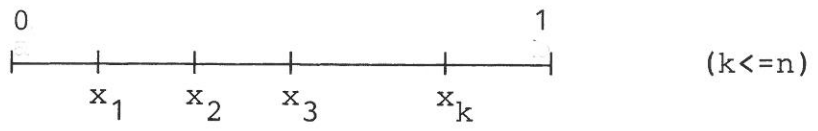
For a fixed n, E_n varies with the location of the root, the bars in the above diagram show the minimum and maximum of E_n when the root varies in [a,b].

In the next section we show another way of allocating the points in the interval which leads to a better worst case running time and to a smaller variation. Both of these improvements are essential in practical applications.

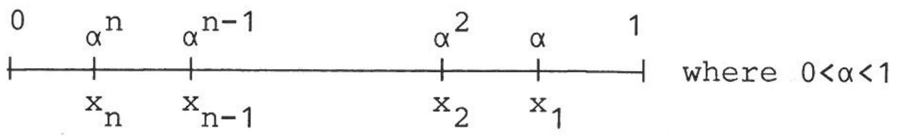
3.2 Invariant ratio algorithm

In this section we present a new asynchronous partitioning algorithm which avoids clustering of processes. The objective of this is to reduce the worst case running time and to make the variation in the running time as small as possible. To simplify the presentation we assume that the interval [a,b] is [0,1] throughout this section.

Consider a snapshot of the interval:



To reduce the worst case running time the next call of "receive" should return a point within the largest of the sub-intervals $[a,x_1]$, $[x_1,x_2]$,...,$[x_k,b]$, but searching through the intervals to find the largest is time consuming. By allocating the n processes as follows:

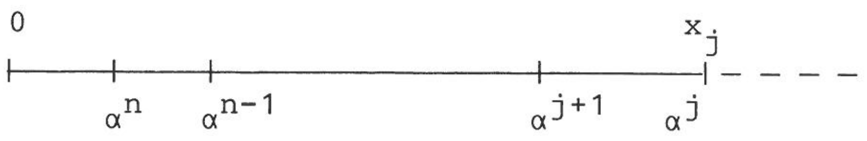


we obtain

1) that it is simple to locate the largest of the subintervals, since it is either $[0,\alpha^n]$ or $[\alpha,1]$,

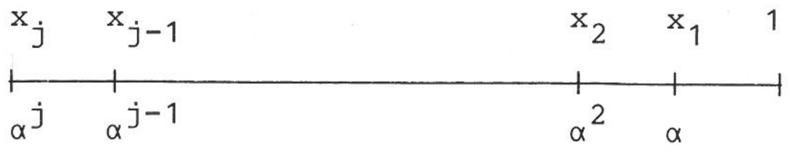
2) that the processes do not cluster.

Consider a situation where the process evaluating H at x_j reports that there is a root in $[0,x_j]$. The interval is then reduced to $[0,x_j]$.



Hence the length of the interval is reduced to α^j. This means that in the new interval there are processes working at the points αl, $\alpha^2 l$,...,$\alpha^{n-j} l$, where $l=\alpha^j$ is the length of the new interval. Hence these processes are automatically allocated correctly, with an invariant ratio, namely as $\alpha, \alpha^2, \ldots, \alpha^{n-j}$. The other j processes are of course allocated to the points $\alpha^{n-j+1}, \ldots, \alpha^{n-1}, \alpha^n$. Exactly in the same ratio as initially.

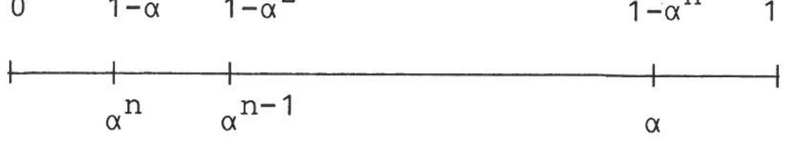
The situation is more complicated when the root is reported to be in the interval $[x_j, 1]$.



If we could find an α such that:

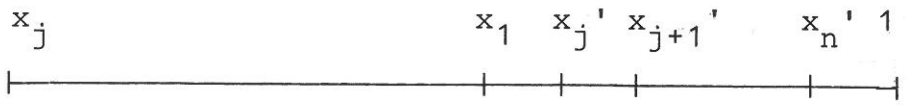
$$\text{I:}\quad \begin{aligned} \alpha^n &= 1-\alpha \\ \alpha^{n-1} &= 1-\alpha^2 \\ &\vdots \end{aligned}$$

this situation would be equally simple. The only change would be to reverse the direction of the interval



Unfortunately the set of equations I: does not have a solution for n>2. For n=2 the above sketched algorithm is the same as the golden section algorithm proposed by Kung [Kung 1976].

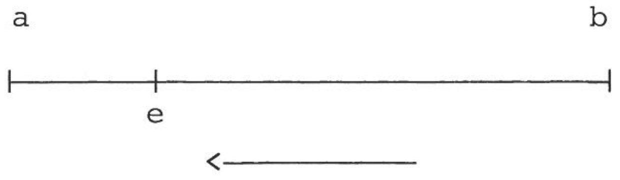
When reducing the interval to $[x_j, 1]$ we propose the following allocation of the remaining processes to the points $x_j', x_{j+1}', \ldots, x_n'$;



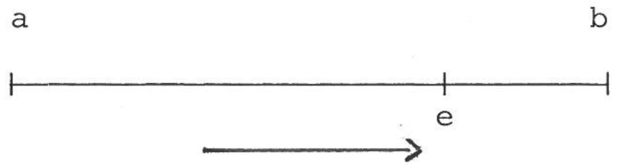
where

$$\begin{aligned}
x_j' &= 1-\alpha(1-x_1)\\
x_{j+1}' &= 1-\alpha^2(1-x_1)\\
&\vdots\\
x_n' &= 1-\alpha^{n-j+1}(1-x_1)
\end{aligned}$$

This may of course be slightly different from the perfect allocation at $\alpha,\alpha^2,\ldots,\alpha^n$ but our experiments show that it is sufficiently close to give a very stable algorithm. Furthermore, this algorithm is very simple to program. The interval is represented by the endpoints "a" and "b", an orientation and the value of the last point "e" allocated in the direction given by the orientation:



The next process calling "receive" will be allocated to the point $a+\alpha(e-a)$. A reduction of the interval to $[a,x_k]$ will neither change e nor the orientation. But a reduction of the interval to $[x_k,b]$ requires a reversal of the orientation and setting e equal to the rightmost point currently allocated:

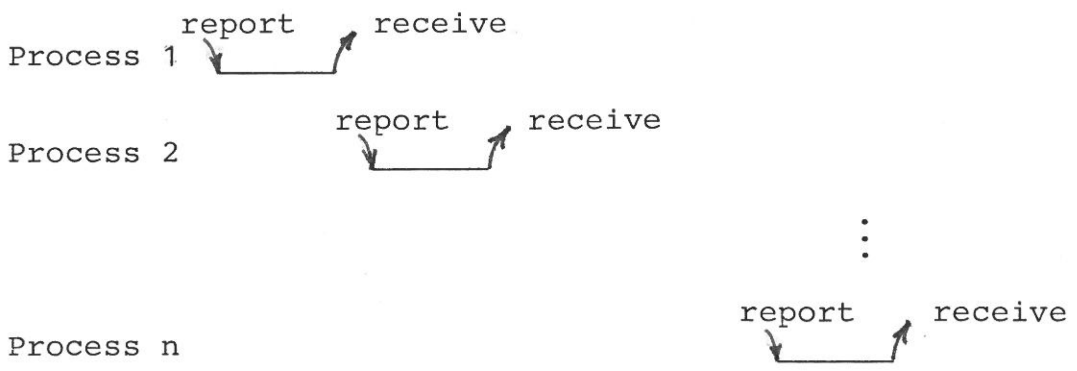


This algorithm aims at distributing the points in $[a, b]$ as α, α^2, ... α^n are distributed in $[0, 1]$. We claim that the minimal worst case running time is obtained when α is chosen such that $\alpha = 1-\alpha^n$, i.e. the length of the two end intervals is the same. This claim is justified by the experimental results presented in the next section. All further details are given by the program shown below.

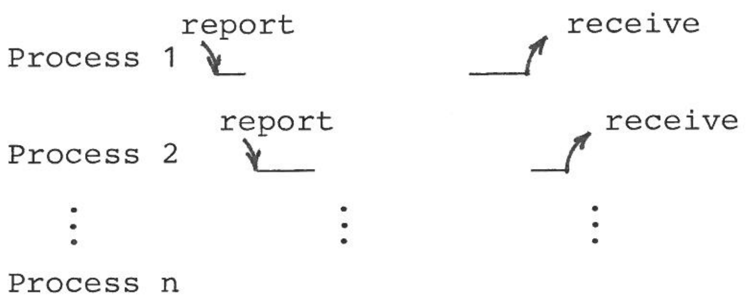
```
TYPE INTERVAL  =  MONITOR;

   TYPE  POINT = REAL;
         DIRECTION = (LEFT, RIGHT );
         INTER = RECORD
                      A, B:POINT;
                        HA:REAL;
                       DIR: DIRECTION;
                         E:POINT
                 END;

   VAR CURRENT: INTER;
          ALFA: REAL;


   PROCEDURE UPDATELEFT( VAR T:INTER );
   BEGIN
     WITH T DO
      BEGIN
          IF DIR = RIGHT
            THEN E:= B - ALFA*(B-A);
          DIR:= LEFT;
      END
   END;

   PROCEDURE UPDATERIGHT( VAR T:INTER );
   BEGIN
     WITH T DO
      BEGIN
          IF DIR = LEFT
            THEN E:= A+ALFA*(B-A);
          DIR:= RIGHT;
      END
   END;

  FUNCTION  LENGTH( T:INTER ):POINT;
  BEGIN LENGTH:= T.B - T.A   END;

  PROCEDURE ENTRY REPORT(VAR X:REAL; HX:REAL; VAR FINISH:BOOLEAN);
  VAR TEMP:INTER;

  BEGIN
     TEMP:= CURRENT;
     WITH  TEMP  DO
       IF HX*HA <= 0.0
             THEN BEGIN
                      UPDATELEFT( TEMP );
                      B:= X
                  END
             ELSE BEGIN
                     UPDATERIGHT( TEMP );
                      A:= X; HA:= HX
                  END;
     IF  LENGTH( TEMP ) < LENGTH( CURRENT )
       THEN CURRENT:=  TEMP;
     FINISH:= ( LENGTH( CURRENT ) <= EPS )
  END;
```

```
PROCEDURE  ENTRY  RECEIVE( VAR X:POINT );
BEGIN
    WITH  CURRENT  DO
    BEGIN
        IF DIR=LEFT
          THEN E:= A+ALFA*(E-A)
          ELSE E:= B-ALFA*(B-E);
        X:= E;
    END
END;
```

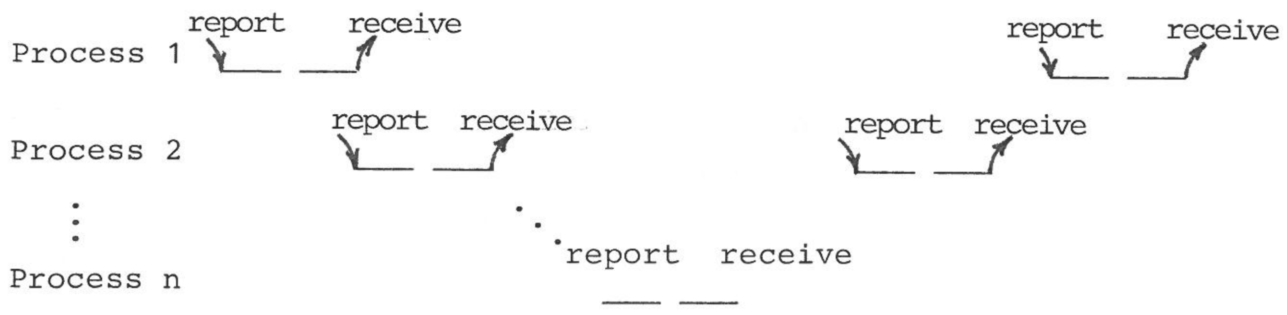
In an earlier version of the algorithm the two procedures "report" and "receive" were combined into one. This means that reporting a result and receiving a new point to work at is done indivisibly.



Such a coupling of report and receive may happen in an asynchronous algorithm e.g. when the evaluation of H is very long or when it fluctuates. There is no logical reason to enforce that the two procedures are coupled, that is why they were separated in the program shown above. By separating them we may get



When the time it takes to evaluate H is constant the execution approximates that of the synchronous algorithm. This happens when all processes report before any of them receive a new point to work at. On the other hand there are also executions where report and receive from one process always follow each other immediately:



This represents a worst case of the algorithm where the choice of α has a strong influence on the running time.

In this section we have presented a partitioning algorithm called the invariant ratio algorithm, the major assets of this algorithm are

- it is very simple to program
- choosing the invariant ratio so that $\alpha^n = 1-\alpha$ minimizes the worst case running time.

In the next section we present the running times of the algorithm obtained on a multiprocessor. These support the claims made above, furthermore they show that the running time is very close to the lower bound given in section 2.

3.3 Experimental results

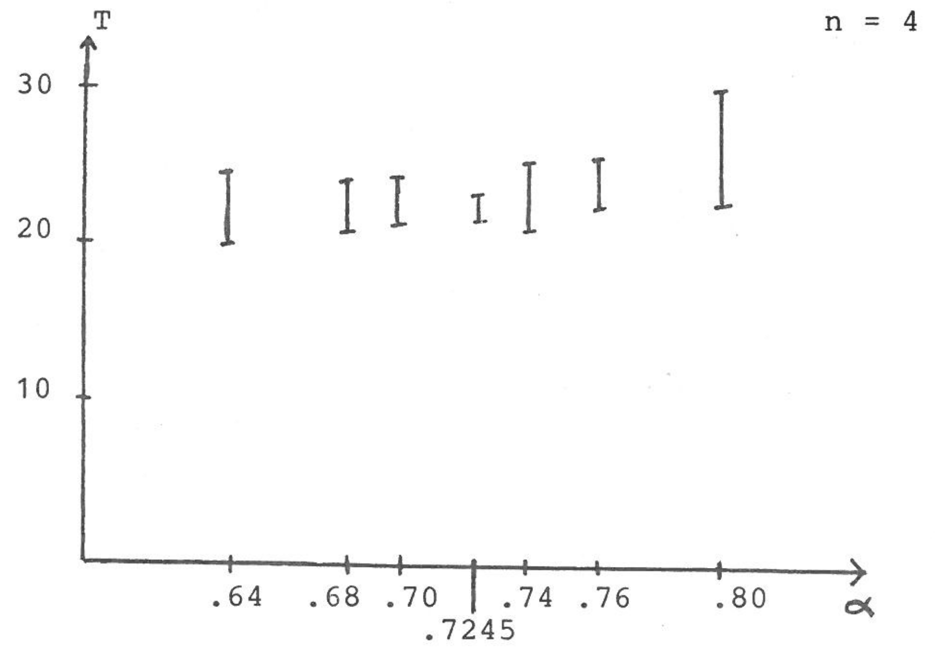
The invariant ratio algorithm has been analyzed on an experimental multiprocessor, Multi-Maren [Møller-Nielsen and Staunstrup 82]. Below the results of these experiments are summarized. The experiments support the claims made in the previous section:

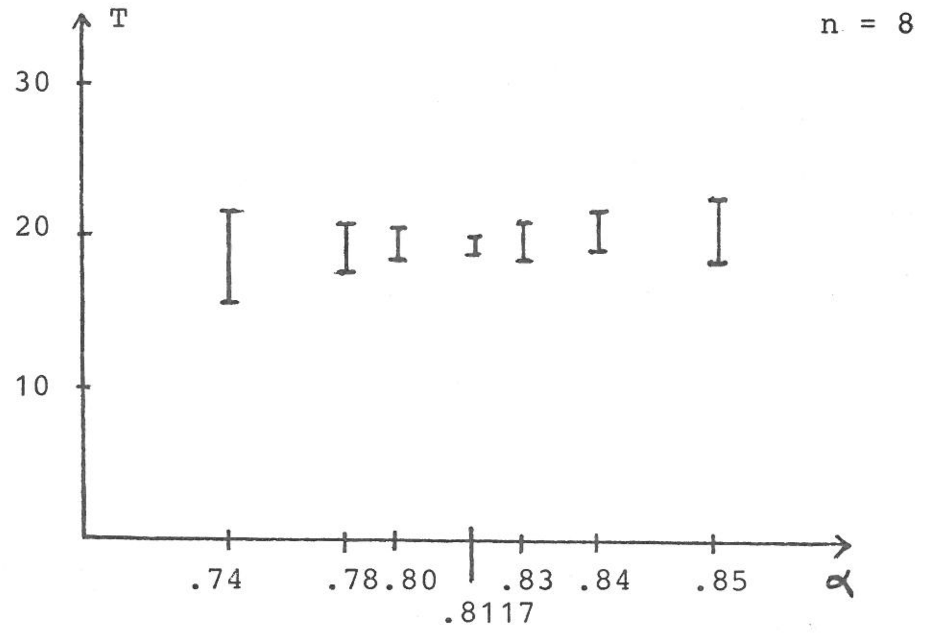
1) the minimal worst case running time using n-processors is obtained by choosing α so that $\alpha^n = 1-\alpha$

2) when using this α, the invariant ratio algorithm performs significantly better than the asynchronous equidistant algorithm.

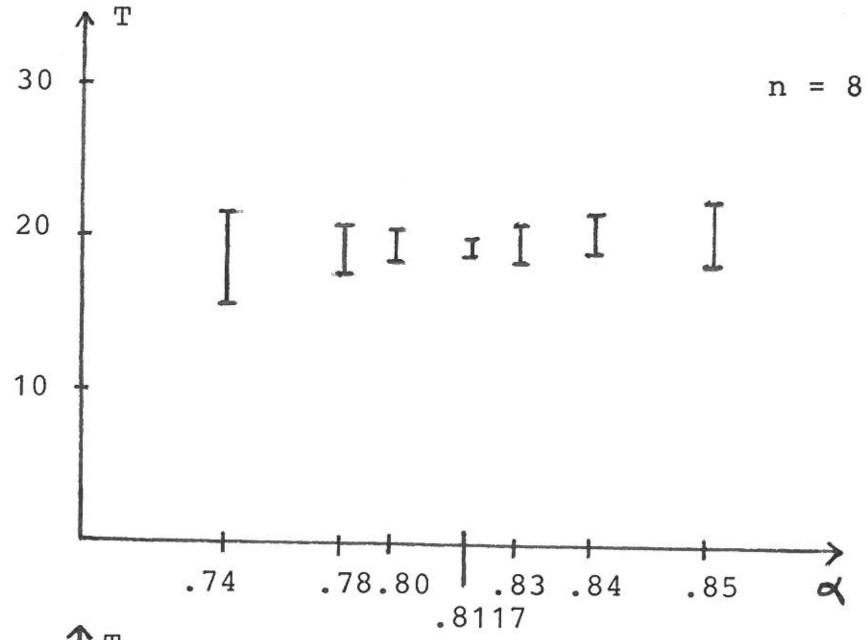
3) when using this α the running time of the invariant ratio algorithm is very close to the lower bound derived in section 2.

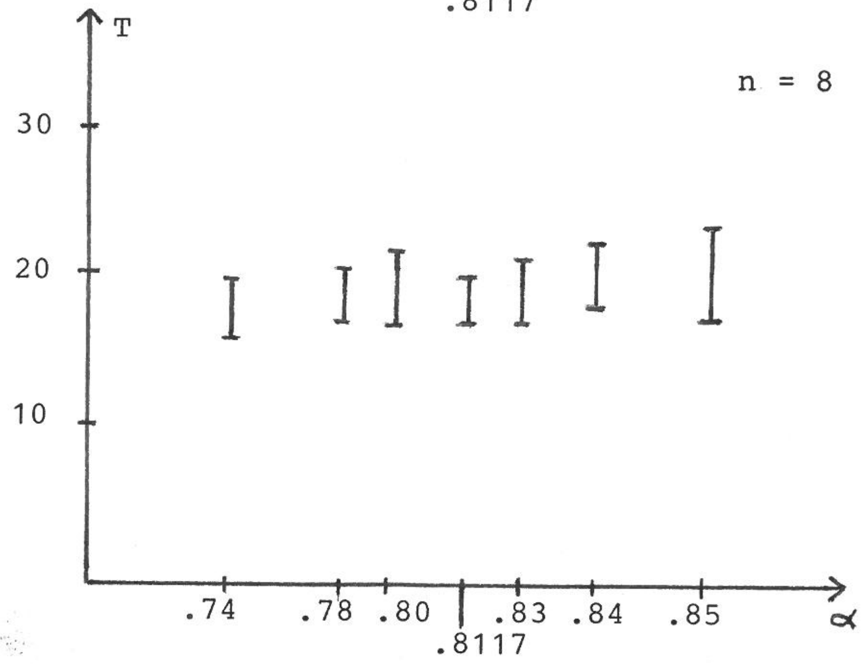
To verify claim 1) the running time needed by the invariant ratio algorithm was measured for different values of α. The following two diagrams show the variation in the running time for various α's, the variation stems from placing the root in different points of the interval. The bars show the minimal and maximal running time.



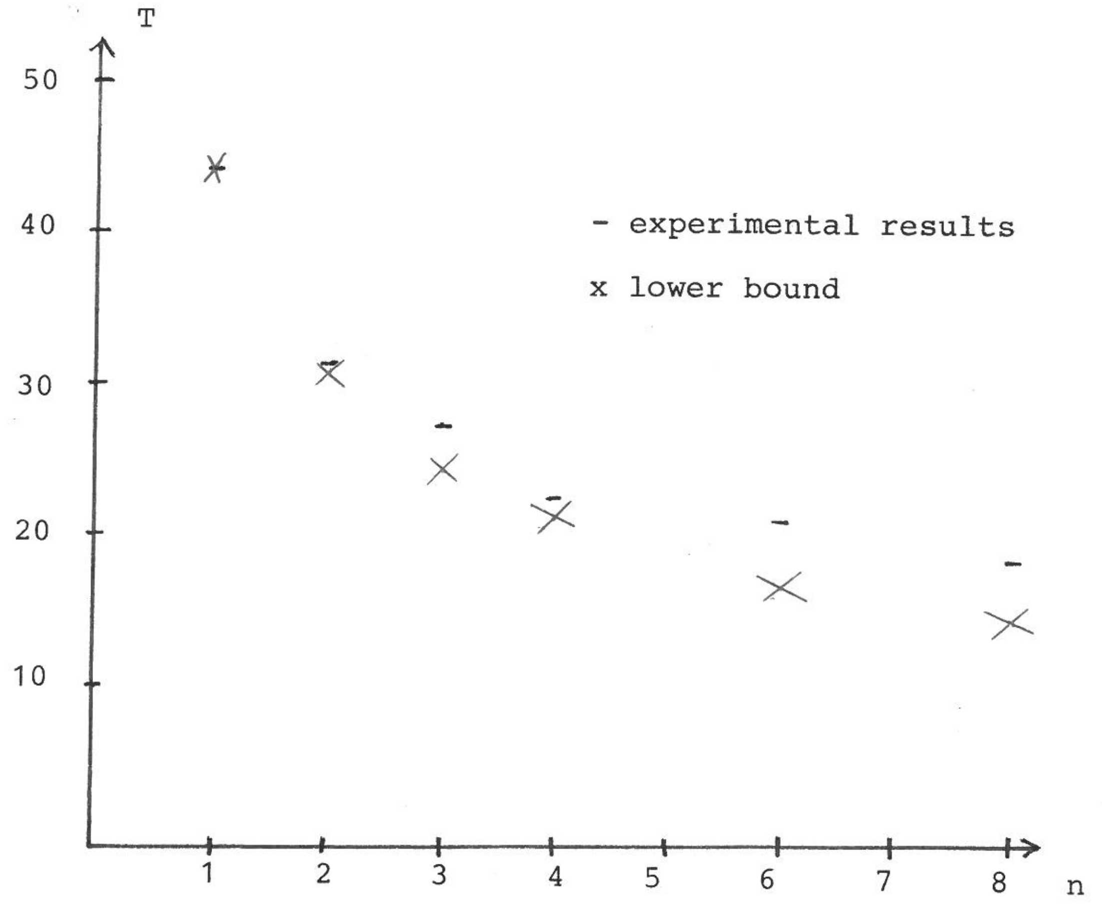


Both diagrams show that the worst case running time, average running time and variation in running time is minimal around $\alpha=0.72$, $\alpha=0.81$ respectively. The results shown above were obtained with a version of the algorithm where respond and receive are combined tofocus on the worst case as explained in section 2. When the two procedures are separated, the average is improved considerably but the worst case is not, as shown in the next diagram:





When the two procedures receive and report are separated, the average number of function evaluations needed by the invariant ratio algorithm approximates the lower bound derived in section 2. The experimental results support this claim; the diagram below shows the average number of function evaluations needed by the invariant ratio algorithm together with the lower bound curve from section 2.



By separating the procedures "report" and "receive" so that they are not executed indivisibly the average number of function evaluations is lowered, but the worst case is of course not since there are still execution histories where they appear pairwise. So the net effect of separating them is to give a faster execution in most cases without increasing the worst case.

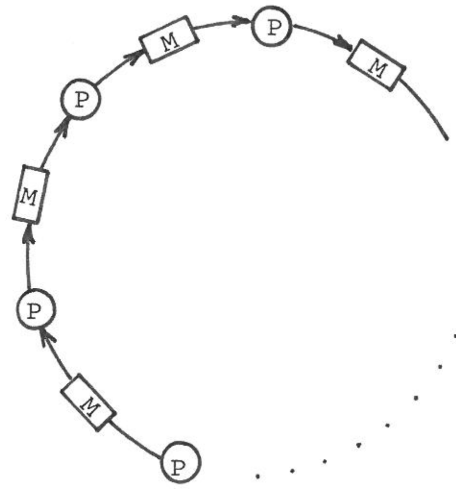
Decentralized algorithm

In the previous section we considered a class of algorithms where the datastructure representing the interval was shared by all processes. Such a program structure has an inherent bottleneck, since the processes will delay each other when they use the shared datastructure.

Let the time to evaluate the function H be T_H and the time one process reserves the monitor in each iteration be T_I, then it is obvious that the maximum number of processes which can be utilized is

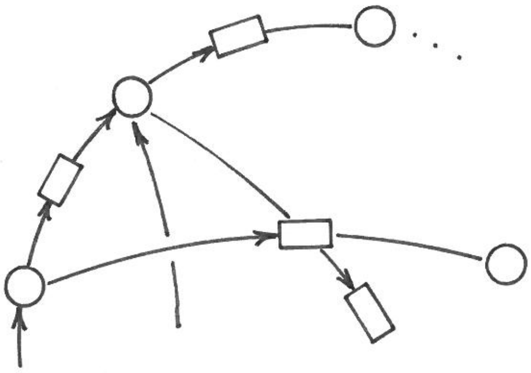
$$N^* = T_H / T_I$$

and there will probably be an overhead even for smaller values of n. For large values of T_H this is not going to be a significant problem as the numbers in the previous section indicate, but for moderate values of T_H where T_H and $n*T_I$ are of the same magnitude, it would be nice to find a decentralized algorithm i.e. one without a global shared datastructure. One may, for example, consider an algorithm where the processes are pairwise connected.



If all processes receive from their left neighbour and report to their right neighbour, the processes and monitors are the same as in the centralized algorithm, but in the decentralized algorithm there are n instances of the monitor interval administrator. It is obvious that this algorithm will find the root, a process which narrows the interval, sends it to its right neighbour which uses it and probably narrows it even further before sending it to the right etc.

This is, however, not a very efficient algorithm, since a "good" partitioning will take n iterations to propagate to all other processes, hence the whole speed-up is lost in propagation delays. By extending the fan-in and fan-out of all processes to two, the propagation delay can be reduced to $\log_2 n$ iterations.

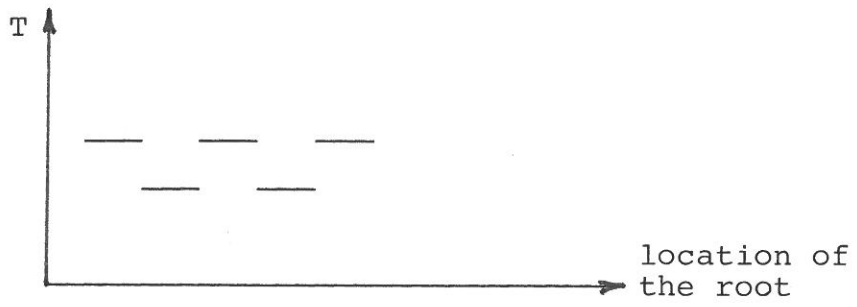


But with a propagation delay of $\log_2 n$ the speed-up is lost. If the n processes were working independently they would in $\log_2 n$ steps reduce the interval by $(1/2)^{\log_2 n}=1/n$. So in $\log_2 n$ steps the processes can themselves reduce the interval by 1/n. To eliminate the propagation delays there must be processes which reduce the interval by more than 1/n in each iteration. The lower bound given in section 2 shows that this is not possible. The only hope would be to find an interconnection pattern which established a path of constant length (independent of n) between any two processes, but this is impossible if we only allow limited fan-in and -out of all processes.

We therefore conclude that for the rootsearching problem, there is no efficient decentralized algorithm.

Conclusion

One may see the algorithm presented in this paper as a representative of a class of algorithms with one common characteristic: worst case reduction. Consider a sequential algorithm with a fluctuating running time:



This is precisely the case with a sequential partitioning algorithm where $\alpha \neq \frac{1}{2}$. By choosing different α's the peaks in the running time change. This behaviour can be utilized in a concurrent algorithm where different α's can be tried simultaneously by different processes. By a careful choice of α's such as the one described in this paper all peaks can be covered, thus effectively reducing the worst case.

Although we have not yet tried this strategy on other problems, it appears to be one of the useful heuristics for devising new concurrent algorithms.

Appendix: Details about the experiments

In this appendix, we present some of the details of the program used to obtain the experimental results presented in section 3.3.

Choice of H

Rather than using some complicated and time-consuming function, we chose to use the identity function, H(x)=x, and then introduce an artificial delay in each call of H

```
function H(x: real): real
 var
    i: integer;
 begin
    for i:=1 to H_delay do;     "artificial delay"
    H:=x
 end;
```

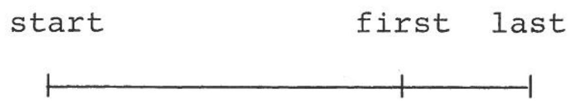
This has the advantage that it is very easy to adjust the length of the delay. Two kinds of delays were used: fixed delay where H_delay was the same for all points in the interval and varying delay where H_delay is varied with x.

Variation of root

All experiments involved measuring the number of function evaluations for various placements of the root in the interval [a,b]. This variation has been obtained by keeping the root fixed in 0 and then vary a from -10^{15} to 0 and b from 0 to 10^{15}. This strategy has the advantage that the root can always be obtained with the same absolute accuracy regardless of its relative placement in the interval.

Measuring the running time

When the root has been found not all processes may detect this simultaneously, those that are in the middle of a function evaluation will not find out until the next time they report to the interval monitor. Hence there may be a span between the time when the first process detects that the root has been found and the time when the last process detects this:



Both of these are reasonable to use as the running-time as long as the same is used in all experiments. We have measured both in all our experiments, but all the results reported in section 3.3 are based on the last process reporting.

Acknowledgement

We want to thank Peter Møller-Nielsen for many stimulating discussions on the algorithms presented here and for his comments on earlier versions of this report. He also suggested that worst case reduction might be a generally applicable heuristic for constructing concurrent algorithms.

References

[Brinch Hansen 75] : The programming language Concurrent Pascal, _IEEE Transactions on Software Engineering 1_, 2 (June 1975), 199-207.

[Kung 76] : Synchronized and asynchronous parallel algorithms for multiprocessors in _"Algorithms and Complexity: New Directions and Recent Results"_ (J.F. Traub, ed.), Academic Press, New York, 1976.

[Møller-Nielsen and Staunstrup 82] : Early experience from a multiprocessor project, DAIMI PB-142, Computer Science Department, Aarhus University, January 1982.