Computers with parallel computing capabilities may become generally available in the future. Some implications for the field of numerical analysis are indicated. An analogue of the bisection algorithm for root determination, employing redundancy in computation as a means of developing parallelism, is presented. An evaluation of the effect of redundant parallel computation on the speed and efficiency with which results are obtained is given for this algorithm.

Evaluation of redundancy in a parallel algorithm

by G. S. Shedler and M. M. Lehman

Computing systems capable of simultaneously executing a number of tasks may become more important in the future.[1] This paper considers an implication of this trend in the field of numerical analysis. We aim to give an informal description of one means whereby parallelism can be produced and exploited. A particular technique for the development of additional parallelism in conventional sequential algorithms (the introduction of redundancy) is illustrated by a discussion of the bisection technique for determining the roots of functions.

A preliminary discussion of serial and parallel algorithms is given in the following section. The consideration of an algorithm as a partially ordered set of tasks enables a distinction to be drawn between them. A graphic notation for presenting parallel algorithms is also introduced. The next section takes up the development of parallel numerical algorithms. In a later section, a redundant parallel algorithm that generalizes the bisection method for the determination of the value of an isolated root is given. Finally, the effect of parallel computation on the speed and efficiency of obtaining results by this algorithm is evaluated.

Serial and parallel algorithms

An algorithm (or a program to implement an algorithm) consists of a set T of tasks $T_1, T_2, \cdots, T_m$ and a specification of any necessary ordering of the tasks. The logic of the algorithm may impose an ordering among $T_1, T_2, \cdots, T_m$. That is, the result obtained by executing the set of tasks T may depend on the order or sequence in which the tasks are executed. In this event, a partial or complete ordering of the T_i will define a "correct" result. Specifically, we can define an *order relation* $<$ in T as follows. For $i \neq j$ and $i = 1, 2, \cdots, m$; and $j = 1, 2, \cdots, m$

$$T_i < T_j$$

if, in the algorithm, task T_i must be completed before task T_j can be initiated. In such a case, it is said that T_i precedes T_j or that T_j succeeds T_i.

As traditionally stated, the algorithms of numerical analysis are serial procedures; that is, the order of the tasks that comprise the algorithm is a linear ordering. There appears to be no intrinsic reason why such serial algorithms should dominate. In many cases, the reason seems to have been strictly historical in that throughout the development of the numerical calculus both human and machine calculators could execute only one task at a time. Thus there was, in general, little incentive to create or discuss non-serial computational sequences or to determine permissible departures from sequential execution. Moreover, such sequences are of the most general form in the sense that any parallel process can be expressed in sequential form.

Some algorithms, for example the bisection method for root determination, are completely ordered procedures, each task requiring the result of its predecessor. Such algorithms may be termed *properly serial.* They are, however, the exception rather than the rule. Within a given serial algorithm, one may often find tasks or sets of tasks that are logically independent and can be computed simultaneously if the appropriate computing facilities are available. Such independent tasks within an algorithm comprise an instance of what we refer to as *latent parallelism.*

If the task ordering specified for an algorithm is explicitly partial, it may be termed a *parallel algorithm.* For if there exist distinct indices i and j such that $T_i \nless T_j$ and $T_j \nless T_i$, then T_i and T_j can be performed concurrently.

A parallel algorithm can be described by a directed graph in which each node of the graph is associated with a task of the algorithm or with one of three control functions: *split, join,* or *terminate.* Each branch of the directed graph is associated either with unconditional or conditional data flow.

A split node (Figure 1) indicates permissible simultaneous initiation of all the tasks or functions that are the immediate successors of the split node. A join node (Figure 2) indicates that all tasks or functions associated with nodes that are direct predecessors of the join node must be completed before control passes beyond that node. A terminate node (Figure 3) indicates the immediate interruption of all tasks or functions associated with the nodes specified at the terminate node that are being executed when the terminate occurs. Subsequent initiation and execution of these tasks or functions is not inhibited by the terminate node. Dotted lines with

Figure 1 Split node

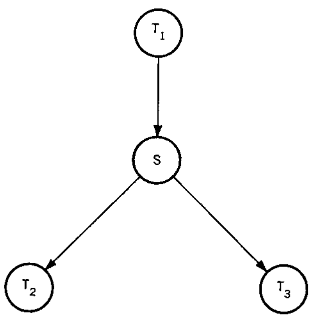


Figure 2 Join node

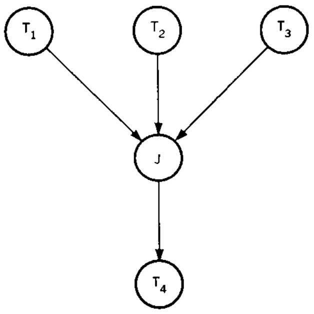


Figure 3 Terminate node

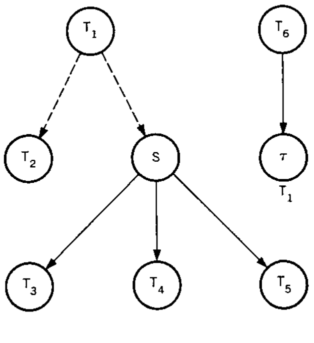
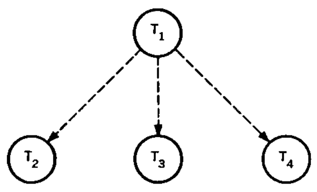


Figure 4 Conditional node



arrowheads are used to indicate branches associated with conditional data-flow (Figures 4), and solid lines with arrowheads are used to indicate branches associated with unconditional data-flow.

A node associated with a task T_i is denoted by $\textcircled{T_i}$. Split nodes are denoted by $\textcircled{S_i}$, and join nodes by $\textcircled{J_i}$, or by $\textcircled{S}$ and $\textcircled{J}$ when there is no ambiguity. A node associated with the termination of tasks T_i and T_j is denoted by $\textcircled{\tau}_{T_i,T_j}$. A stop task is used to indicate the end of an algorithmic procedure.

The configuration in Figure 1 means that upon completion of task T_1, tasks T_2 and T_3 are initiated simultaneously. Task T_1 is said to split to tasks T_2 and T_3. The interpretation of Figure 2 is that task T_4 is initiated as soon as tasks T_1, T_2, and T_3 have been completed. Tasks T_1, T_2, and T_3 are said to join at J. Figure 3 shows that if task T_1 is being executed at the completion of task T_6, then T_1 is interrupted immediately. Figure 4 requires that upon completion of task T_1, one of the tasks T_2, or T_3, or T_4 is initiated.

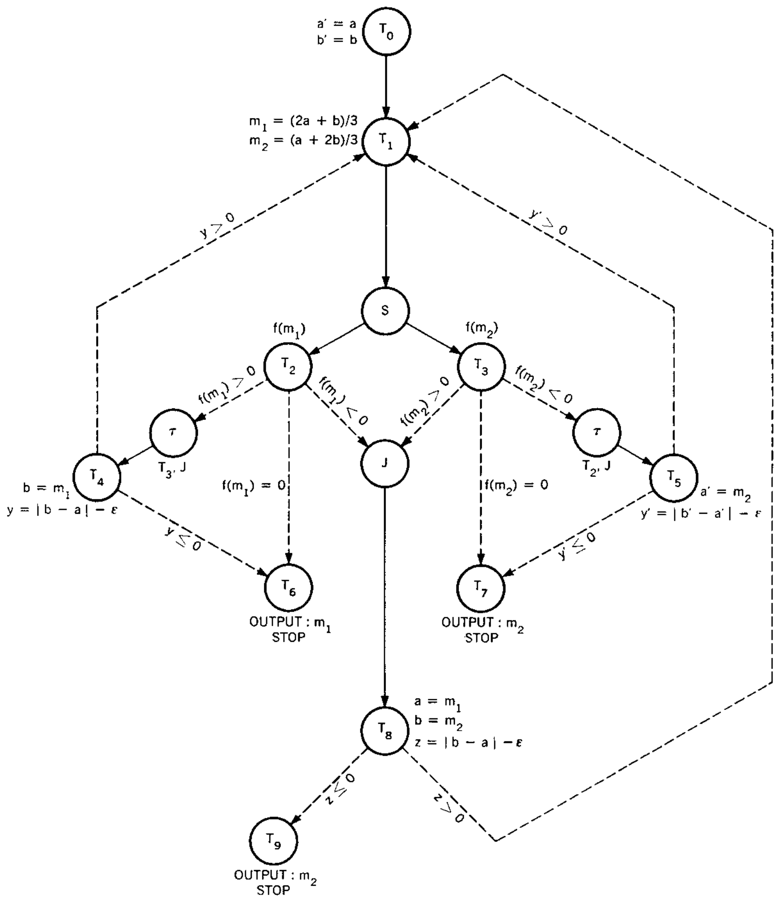
Parallel algorithms and numerical analysis

In an attempt to increase the speed of numerical computation, ever faster computers have been built through efforts in the areas of technological improvement and computer organization. At the same time, improved computational procedures were being developed. At the mathematical level, attempts to speed up numerical computation were almost exclusively in the direction of decreasing the amount of work to be done. Another mathematical approach involves the development of techniques that make efficient use of multiple computing facilities that a parallel processing system would provide. Algorithmic procedures composed of tasks, many of which can be performed in the absence of results from other tasks, are referred to as *highly parallel algorithms*. Each task comprises an ordered or partially ordered sequence of instructions or operations. This type of parallelism has been called macro-parallelism;[2] that is, meaningful strings of consecutive instructions define a task that can be performed simultaneously with similarly defined tasks. This is to be distinguished from micro-parallelism in which partial ordering is exploited among instructions by the simultaneous execution of independent instructions.

The first source of highly parallel algorithms is the set of standard algorithms of numerical analysis. Preliminary investigations[1] of such algorithms indicate that although many of these have generally and implicitly been accepted as serial, they do contain a great deal of latent parallelism.

One approach to the construction of parallel algorithms has been described by Nievergelt.[3] He suggests that reduced turnaround time can be obtained from a parallel-processing device by introducing redundant computation into a serial algorithm to obtain a highly parallel algorithm. Such a technique requires more work to obtain a particular result, but it takes advantage of a machine's parallel-processing capability to obtain the result more rapidly.

Figure 5 Bisection method for finding an isolated root



Moreover, it is expected that new, efficient, highly parallel algorithms can be developed. For example, Dorn[4] has been able to develop an efficient parallel algorithm for evaluating polynomials by generalizing Horner's rule. Karp and Miranker[5] have found parallel search policies for the maximum of a unimodal function. Also, Miranker and Liniger[6] have derived a class of numerical integration formulas of a parallel type for ordinary differential equations.

Redundancy and the bisection method

Redundancy was applied by Nievergelt[3] to the solution of ordinary differential equations. We present a redundant analogue of the bisection method for finding an isolated root of a function. A representation of this algorithm is given in Figure 5.

standard algorithm

Suppose that a continuous function f of a real variable x has a single root in the interval (a, b) and that $f(a)f(b) < 0$. Without loss of generality, we may assume that $f(a) < 0$ and $f(b) > 0$ and proceed as follows. Having calculated $m_1 = (2a + b)/3$ and

$m_2 = (a + 2b)/3$, split to two parallel tasks, T_2 and T_3, for the computation, respectively, of $f(m_1)$ and $f(m_2)$. If $f(m_1) > 0$, terminate T_3 and the join J of T_2 and T_3, set b equal to m_1, and repeat the procedure. If $f(m_1) = 0$, the output m_1 is the required root, and the computation stops. If $f(m_1) < 0$, join at J with task T_3. If $f(m_2) < 0$, terminate T_2 and the join J, set a' equal to m_2, and repeat the procedure. If $f(m_2) = 0$, the output m_2 is the required root, and the computation stops. If $f(m_2) > 0$, join with task T_2 at J. When the join J is satisfied, set a equal to m_1 and b equal to m_2, and repeat the procedure. The procedure is continued until the root is determined to the desired accuracy. Note that redundant computations are terminated when it is known that their outputs will not be needed.

generalized algorithm

The above procedure may be generalized in the following way. Let N be a positive integer greater than or equal to two. Having computed

$$m_i = \frac{[(N - i + 1)a + ib]}{N + 1} \qquad i = 1, 2, \cdots, N$$

split to N parallel tasks for the computation, respectively, of $f(m_1), \cdots, f(m_N)$. For any $i = 1, 2, \cdots, N$, if $f(m_i) = 0$, then m_i is the required root and the computation stops. Otherwise, terminate redundant tasks in accordance with the sign change of the function, and join with the remaining members of the set of N tasks. When the join of these tasks is satisfied, set a equal to m (where m is the greatest element of the set $a, m_1, m_2, \cdots, m_N, b$ such that $f(m) < 0$), and set b equal to m' (where m' is the least element of the set $a, m_1, m_2, \cdots, m_N, b$ such that $f(m') > 0$). Having set a and b, repeat the procedure, continuing in this way until the root is found to the desired accuracy. The choice of $N = 2$ yields the previously described algorithm.

It is perhaps appropriate to mention at this point that the standard Newton-Raphson algorithm is an inherently parallel algorithm; that is, a function value and its derivative can be computed simultaneously. Since the Newton-Raphson procedure converges quadradically, we expect it to be superior to the $N = 2$ redundant bisection algorithm described above. For larger values of N, however, the redundant algorithm could approach the performance of the (parallel) Newton-Raphson method, at the same time avoiding the difficulties associated with the choice of a first approximation. Limited computational experience with polynomials supports this assertion.[7]

Speed estimates

Under the assumption that the time required to initiate and terminate parallel tasks is small compared to the time required for a task, an iteration of one of the redundant algorithms takes approximately the same amount of time as an iteration of the standard algorithm. Therefore, it is possible to obtain a measure of the relative speeds of these procedures by calculating the expected value of the number of iterations required for a prescribed accuracy.

The validity of our assumption depends on the nature of the function involved. For example, in the case of polynomials of degree perhaps greater than fifty, we expect it to be valid.

For the purpose of this analysis, we consider an interval of length d and assume that the occurrence of a root of the function is uniformly distributed within that interval. Let an accuracy (error tolerance) of $\epsilon < 0$ be given.

For the standard algorithm, define n' to be the greatest positive integer such that

$$s_{n'} = 2^0(2\epsilon) + 2^1(2\epsilon) + \cdots + 2^{n'-1}(2\epsilon) \leq d$$

Then $E_1(\epsilon)$, the expected value of the number of iterations required to obtain accuracy ϵ, is computed as

$$E_1(\epsilon) = \sum_{k=1}^{n'} P_k(k) + \begin{cases} \left(1 - \sum_{j=1}^{n'} P_j\right)(n' + 1) & \text{if} \quad S_{n'} < d \\ 0 & \text{if} \quad S_{n'} = d \end{cases}$$

where

$$P_1 = \frac{2\epsilon}{d}$$

and

$$P_k = \left(1 - \sum_{j=1}^{k-1} P_j\right)\left[\frac{2^k\epsilon}{d - (2^k - 2)\epsilon}\right] \qquad k = 2, 3, \cdots, n'$$

For the Nth order redundant algorithm, define m' to be the greatest positive integer such that

$$S_{m'} = (N + 1)^0(2N\epsilon) + \cdots + (N + 1)^{m'-1}(2N\epsilon) \leq d$$

The expected value $E_N(\epsilon)$ of the number of iterations of the Nth order redundant algorithm required to obtain accuracy ϵ is computed as

$$E_N(\epsilon) = \sum_{k=1}^{m'} P_k(k) + \begin{cases} \left(1 - \sum_{j=1}^{m'} P_j\right)(m' + 1) & \text{if} \quad S_{m'} < d \\ 0 & \text{if} \quad S_{m'} = d \end{cases}$$

where

$$P_1 = \frac{2N\epsilon}{d}$$

and

$$P_k = \left(1 - \sum_{j=1}^{k-1} P_j\right)\left\{\frac{(N + 1)^{k-1}(2N\epsilon)}{d - 2[(N + 1)^{k-1} - 1]\epsilon}\right\}$$

for

$$k = 2, 3, \cdots, m'$$

Table 1 Expected value of number of iterations

Accuracy	*Number of processors*									
ϵ	1	2	3	4	5	6	7	8	9	10
10^{-4}	11.4	7.3	5.9	5.2	4.7	4.4	4.1	3.8	3.8	3.7
10^{-5}	14.7	9.4	7.6	6.6	5.9	5.6	5.3	4.9	4.8	4.7
10^{-6}	18.0	11.5	9.3	8.0	7.3	6.7	6.4	5.9	5.8	5.6
10^{-7}	21.3	13.6	10.8	9.5	8.6	7.8	7.5	6.9	6.8	6.6
10^{-8}	24.7	15.7	12.6	10.8	9.8	9.1	8.6	8.0	7.8	7.6

Table 2 Execution time compared to a single processor

Accuracy	*Number of processors*									
ϵ	1	2	3	4	5	6	7	8	9	10
10^{-4}	100.0	64.6	52.0	45.9	41.3	39.1	35.8	33.8	33.2	32.6
10^{-5}	100.0	64.1	51.5	45.0	40.0	38.2	35.7	33.0	32.5	31.8
10^{-6}	100.0	63.9	51.8	44.7	40.8	37.5	35.6	32.5	31.8	31.0
10^{-7}	100.0	63.6	51.0	44.6	40.3	36.6	35.3	32.5	31.8	31.0
10^{-8}	100.0	63.7	50.9	43.7	39.6	36.7	35.0	32.6	31.5	30.7

Table 3 Redundancy compared to a single processor

Accuracy	*Number of processors*									
ϵ	1	2	3	4	5	6	7	8	9	10
10^{-4}	100.0	129.2	156.0	183.7	206.4	234.4	250.4	270.1	299.3	326.3
10^{-5}	100.0	128.1	154.5	180.0	200.2	229.1	250.2	264.2	292.7	318.5
10^{-6}	100.0	127.8	155.4	178.8	204.1	224.8	249.6	261.5	289.7	314.5
10^{-7}	100.0	127.2	153.1	178.4	201.6	219.7	246.9	260.0	286.1	310.0
10^{-8}	100.0	127.4	152.7	174.9	197.9	220.4	244.6	260.6	283.9	307.1

The values of $E_1(\epsilon)$ and $E_N(\epsilon)$ have been computed for a range of ϵ from 10^{-4} to 10^{-8} and for N ranging from two to 10. The results of these computations are given in Table 1. The length d is equal to one.

For the same ranges of ϵ and N, execution times compared to the standard (single-processor) algorithm are tabulated in Table 2. These figures indicate the substantial increase in speed that can be obtained from parallel processing by means of the redundant algorithms.

A measure of the amount of redundant computation contained in these algorithms appears in Table 3. Again for the same ranges of ϵ and N, the amount of computation compared to the single-processor algorithm is tabulated. These figures represent worst-case estimates since savings due to termination are not considered.

Summary

The development of computing systems with highly parallel facilities suggests the study and development of techniques of numerical analysis designed to exploit the capabilities of such systems. Such investigations should aim to produce algorithms that result in faster computations than are now possible using conventional machines and standard algorithms. On the basis of the work described here and the referenced investigations, approaches that can be expected to succeed are the reformulation of classical procedures to expose latent parallelism, the introduction of computational redundancy into algorithms, and the development of new parallel algorithms.

CITED REFERENCES

1. M. M. Lehman, "A survey of problems and preliminary results concerning parallel processing and parallel processors," *Proceedings of the IEEE* **54**, No. 12, 1889–1901 (1966).
2. M. M. Lehman, "Serial-mode operation and high-speed parallel processing," *Information Processing 1965, Proceedings of IFIP Congress 1965*, North-Holland Publishing Co., Part 2, 631–632 (1966).
3. J. Nievergelt, "Parallel methods for integrating ordinary differential equations," *Communications of the Association for Computing Machinery* **7**, No. 12, 731–733 (1964).
4. W. S. Dorn, "Generalization of Horner's rule for polynomial evaluation," *IBM Journal of Research and Development* **6**, No. 2, 239–245 (April 1962).
5. R. M. Karp and W. L. Miranker, "Parallel minimax search for a maximum," to be published in *Journal of Combinatorial Theory.*
6. W. L. Miranker and W. M. Liniger, "Parallel methods for the numerical integration of ordinary differential equations," *Mathematics of Computation* **21**, No. 99, 304–320 (July 1967).
7. G. S. Shedler, "Parallel numerical methods for the solution of equations," *Communications of the Association for Computing Machinery* **10**, No. 5, 286–291 (1967).