This paper discusses the development and current state of the art in parallel processing. The advantages and disadvantages of the various approaches are described from a pragmatic viewpoint. One that is described and discussed is a natural extension of the evolutionary approaches. This alternative covers the problems of locating, describing, controlling, and scheduling parallel processing. It is a general-purpose approach that is applicable to all types of data processing and computer configuration.

GENERAL APPROACH

The approach is based on the following assumptions.

1. The specification of potential parallel activity in processing will depend largely upon programmer specification.
2. Major benefits will accrue only if a high degree of parallelism exists; i.e., either many parallel paths exist or, if the parallel paths are few, they must be long.
3. Programmers will specify parallelism only if it is easy and straightforward to do so.
4. A simple control scheme is needed to provide straightforward scheduling.

Programmer Specification

It is obvious that an automatic scheme can be devised to recognize independent parallel paths when the paths are at a task or an instruction level. The latter scheme has been implemented in STRETCH* and other computers; the former is a simple matter of comparing the names of the input and output files of various tasks in a job.

At intermediate levels where subroutines, parameters, and many other forms of indirect addressing exist, the problem is at present unsolved from a general-purpose practical viewpoint.

Therefore, we must now look to the programmer to define independent processes.

Degree of Parallelism

Previous experience with simultaneous or overlapped I/O and multiprogramming suggest that full potential gains are never realized because of the
difficult scheduling problems. Therefore, we expect that scattered parallelism of two or three short paths may not lead to significant gains; multiple paths must be sought. If there are two or three long parallel paths, it may be more practical to specify them as separate tasks.

*Easy Specification*

Experience indicates that any new feature in parallel processing must be easy to specify; otherwise, the feature will not be much used or exploited. Both simultaneous I/O and multiprogramming gained most when they were made programmer independent. However, our first assumption implies that in this case we cannot rely on programmer independence, so we must make the specification of parallel processing easy and simple to encourage its use.

*Good Control Scheme*

There are two main requisites of a good control scheme. First, it must be simple so that the overheads of implementing it do not seriously erode the advantages gained. Second, it should be efficient with simple ad hoc scheduling algorithms. Experience has shown that ad hoc schemes are successful whereas complex schemes are not only difficult to implement but disappointing in general performance.9,13

**BACKGROUND**

This section discusses some of the existing literature on parallel processing and relates the findings to the assumptions previously made.

Parallel processing in various forms has produced interesting developments in the computer field. For those interested in pursuing the subject, a general bibliography attached to this paper indicates the continuing voluminous literature on this general subject and its pervasive influence on computer hardware and computer software architecture. It is possible to divide the development into six general classes:

- **Class 1** (1953) Simultaneous Input-Output
- **Class 2** (1957) Fork and Join Statements
- **Class 3** (1958) Multiprogramming Operating System
- **Class 4** (1959) Processor Arrays
- **Class 5** (1961) Planned Scheduling
- **Class 6** (1962) Re-entrant On-demand System

The dates are approximate origins of development, but there has been no attempt to assign single inventors because most ideas were developed in several places more or less concomitantly. These titles are useful labels for discussion in this paper but are not intended as formal definitions. The author regrets that he has not been able to incorporate in this paper formal definitions for the various labels used.

**Class 1, Simultaneous Input-Output** (achieved either by the use of hardware or even programmed time-sharing of the processor) made useful increases of two- or threefold in computer power by parallel use of input-output and computation processes.

**Class 2, Fork and Join Statements** provided a simple language mechanism for programs to take advantage of a multiprocessor computer system. The literature does not show great gains in this area. Examples given12,17 have not looked rewarding except one attributed to Poyen48 by Opler45 which, however, is somewhat clumsy; it is predicated on a specific number of processors and requires an exertion on the part of the programmer that will not in general be obtained. Opler, however, has recognized what we shall call “parallel-for” construction; his term is “do-together.”

**Class 3, Multiprogramming Operating System** provides a mechanism for a greater degree of simultaneous input-output by mixing programs and balancing the use of the parts of a computer configuration,14,26,27,56 which has resulted in useful gains of some two- to threefold.45 This subset of parallel processing maintains independent processes each of which is part of an independent job. These systems have enjoyed extensive development in both large and small computer systems, and in general their schedulers operate on a queue of tasks using ad hoc scheduling algorithms. Good results have been achieved with straightforward general-purpose approaches.

**Class 4, Processor Arrays** are computers with replicated arrays of processors which are powerful on a specific range of problems.44 These processors operate on an array of data obeying a common program. The SOLOMON54 computer and the structure proposed by Holland28 are the best known although they have not enjoyed popularity, partly because they depart radically both in architecture and solution techniques from the evolutionary mainstream (a tendency not likely to succeed as Brooks7 pre-
dictors) and partly, in the writer’s opinion, because they shun the qualifier “general-purpose,” which has been the touchstone of the success of the computer. Associative memories are another special case.

**Class 5, Planned Scheduling** is a title which covers schemes that propose to describe parallel-processing structures and to schedule them using a CPM or job shop scheduling approach. Typically, each part of a process is described separately and is associated with a set of conditions or inputs which cause it to be executed instead of being linked in a program sequence. Such schemes are the antithesis of Class 3 in that they pre-partition the problem into logical parts rather than apply the somewhat arbitrary paging of general operating systems. Some of those schemes attempt (or attempted) pre-scheduling, relying on expected time and space utilization estimates. Again this is a major departure from the mainstream of computer development and has not enjoyed much popularity.

**Class 6, Re-entrant On-demand Systems** are an interesting mutation of multiprogramming systems where the process is common to all users but the data sets may be independent. The classic example is an airlines reservation application. Since the queries are independent, a multiprogramming operating system approach can be used. The principle of re-entrant routines was developed to allow sharing of access to a common set of programs.

It is also relevant to point out a class of activity which has not existed, that is, automatic recognition of parallel processes. This is a subject which is still at an early stage of development. At present there are systems that seek out some overlap at the instruction level in large-scale processors, but no work has been done at the subroutine or program level. Even at the instruction level, as the processor reshuffles access and attempts parallelism, as in the look-ahead in STRETCH, the problem of interrupts and unwinding is formidable. The problems of finding implicit parallelisms in programs are also formidable, especially with the use of parameters, block structures, and late binding. Even if possible, the compiling overhead in analysis may be considerable.

The foregoing summary identifies the author as a mainstream evolutioner rather than a radical innovator. This does not mean that he designates classes 4 and 5 as useless but as yet unproved, and agrees that radical innovation must prove itself by large gains to offset its disruption whereas evolution can be justified by more modest gains.

**THE PARALLEL FOR**

The simple premise of this paper is that all FOR statements in an ALGOL program, and similar constructs in other languages, are good potential sources of parallel activity. FOR statements divide into two kinds: first, those which are iterative and second, those which are parallel. The second group is a large fraction of the total. Two examples may suffice, one scientific and one commercial.

A typical scientific example is matrix addition in which the pairwise additions of corresponding elements can be performed independently, and therefore in parallel. A typical commercial example is the extension of an invoice where the individual multiplications of price per unit times quantity for each item can be performed independently, and therefore in parallel.

It is easy to see that by using the simple device of describing each FOR statement as either iterative or parallel, a large group of parallel paths in programs can be identified. It is interesting to note that Opler and later Anderson use a FOR statement to illustrate their notations.

The FOR statement is a special case of a loop; it is, therefore, natural to dichotomize loops into those that are either iterative or parallel in nature. One simple example shows that parallel loops not only exist but exist in large numbers. The main loop in a payroll program (that is, the payslip loop) is independent for each man. It is, of course, a FOR loop where the meaning might be “for every record on the master file . . .”

It may appear unreasonable to distinguish between a PARALLEL FOR and a rewritten loop, but there are substantial pragmatic differences that would distinguish them at a programming language level.

**SUMMARY OF PARALLEL STRUCTURES**

We have identified three areas of parallel structure in programs:

1. FORK to several dissimilar paths and JOIN
2. PARALLEL FOR statements
3. LOOPS rewritten as PARALLEL FOR statements
All of these structures require some kind of JOIN as a prelude to subsequent processing. Techniques for Item 1 have been described elsewhere and are described later for Items 2 and 3. However, there also exist two other types that should be recorded for completeness:

4. FORKS without a JOIN
5. Completely independent processes

As an example of Item 4 consider a task that updates a file and then produces several reports. After the update there is a FORK to each report routine with no need to JOIN. PL/1 has a facility to do this.49

As an example of Item 5 consider an airlines reservation system and the individual queries and updates made upon it.

ADVANTAGES OF PARALLELISM

The identification of parallelism in programs is useful only if some advantage is gained thereby. For both uni- and multiprocessor systems there are advantages.

In general we can note that, whereas program structures in which FORK statements are used do not tend to show a high order of parallelism, PARALLEL FORS and PARALLEL LOOPS do naturally show a high degree of parallelism; moreover, because they tend to occur in nested sets, they increase the parallelism multiplicatively.

Multiprocessor Advantages

The multiprocessor situation contains four interesting cases: First, there is the simple case where the number of programs to be run is less than the number of processors. The availability of parallel paths enables the system to use the otherwise idle processors. This situation may not seem to occur frequently but it can occur often when a system begins to run out of peripheral devices, even if the load on them is not heavy.

Second, there is the case where a mixture of high- and low-priority jobs is being run. If the disparity in processing is large enough, several processors could work on the high-priority program and reduce its turnaround time.

The third and fourth are cases where batch processing response times for large jobs can be improved if the jobs are run serially rather than in parallel.

The third case is illustrated by the simple (and over-simplified) example of three jobs that each require one hour's capacity of the system. When run serially, they would be completed in one, two, and three hours, respectively, giving an average turnaround time of two hours. When run in parallel, they would have an average turnaround time of three hours. Of course, the choices are never as simple as this example supposes but, in general, serial operation is desirable.

The fourth case considers the savings in internal system overheads when parallel activity on one task replaces parallel activity on different tasks. In situations where frequent overlays are made for both data and program segments, the amount of shuffling can be reduced. In particular, if several processors keep approximately in step (not necessarily exactly in step as required in SOLOMON), they can share access to, and residence of, program segments and some data segments. Even if they do not share exact access, but are accessing a disc, cartridge, or other complex access device, then there is more opportunity to batch and optimize accesses for parts of one task than with parts of diverse tasks using different areas of auxiliary storage.

Single Processor Advantages

The single processor system contains two interesting cases. First, there is the case of a mix of multi-programmed tasks where one has an outstanding priority. Suppose it stalls due to some wait for I/O; then if parallel paths exist, it may be able to proceed on another path instead of passing control to a lower priority task. Such cases have been "hand-tailored" in the past relying on a complex look-ahead type of structure or the batching of accesses to auxiliary storage. If parallel paths were started automatically, the batching could adjust itself dynamically.

A typical example would be an operating system that has two kinds of scheduler, one that allocates processors and another that batches and queues access to auxiliary storage. Then parallel paths could utilize processors while more urgent processes are awaiting input.

Second, there is the case of reducing overlays. Particularly in systems where relatively small overlays are used, the problem of fitting internal loops within one overlay is difficult and sometimes impossible. On a loop that needs two segments, the overlays needed for x executions could be reduced from...
2x to 2x/n if n parallel paths were used. Both these cases also apply in the case of multiprocessor systems, and are illustrated in the example given in the section on Processor Scheduling.

TECHNIQUES

Basic Elements

This section is not a comprehensive treatment of all cases but a range of examples to show how simple techniques can be used to implement and control parallel activity. It will be seen that all the techniques can be handled by a simple organization of five simple functions (PREP, AND, ALSO, JOIN, and IDLE). This provides a uniform mechanism which can easily be incorporated in processor hardware if the economics justify it.

PREP is a function performed before a new set of parallel paths begins. It causes a variable called PPC (parallel path counter) to be established. If other PPC's exist for this process, they are pushed down, which allows sets of parallel paths to be nested. PREP sets the new PPC to the value one. AND (L) is a simple two-way fork; it requests the controller to start a parallel sequence at L and to add one to the current PPC for this process. The processor executing the AND continues to the next instruction. The controller queues the request for the AND sequence.

ALSO (L) is a simple two-way fork which is the same as AND except that no PPC converters are involved. It is equivalent to TASK in PL/1 and is used for divergent paths that do not rejoin. JOIN is a function used to terminate a set of parallel paths. When executed, it reduces the current PPC for the process by one. If PPC is then zero, it is popped up and processing continues. If PPC is not zero, meaning that more paths remain to be completed, it releases the processor executing the JOIN.

IDLE is a function that ends a parallel path and releases the processor executing the IDLE.

It should be noted that by associating PPC with a process rather than the JOIN statement, the property of re-entrant code is retained. JOIN is used as a device similar to CLOSE PARENCHESES.

Example 1. Classic FORK and JOIN

The popular form of the FORK statement is:

\[ \text{FORK (S1, S2, S3 ...)} \] (J)

At the execution of this statement the program divides into parallel paths that commence at statements labelled Si, and join at the statement labelled J.

Example 1 shows the implementation of FORK-JOIN for two paths. A1 establishes a new PPC counter and sets the counter to 1. A2 initiates A5 as a parallel path and increments the PPC counter to 2. A3 executes the second path S2. A4 ends S2 with a transfer to JOIN. A5 executes the first path S1. A6 is the join; when one path ends it reduces the PPC to 1 and frees the processor, when the other path ends, the PPC goes to zero, the next PPC is popped up and the processor continues to A7. The extension to three or more paths is obvious.

Example 2 shows the implementation of a PARALLEL FOR process. Boxes B1, B2, B3, and B4 are the translation of the FOR statement. B1 initializes the generation of the values of the parameter of the FOR statement and establishes the new PPC. B2 is the incrementing or stepping function. B3 is the end test and transfers to JOIN when all paths have been started. B4 starts a new parallel path. B5 is a parallel path. B6 is the JOIN.

In contrast to the examples given by Opler and Anderson, this scheme is independent of the number of processors available, and there is no need to state
Example 2. A PARALLEL FOR

Example 4 shows how the classic file maintenance, or payroll, application can be handled. The preliminary part is similar to the PARALLEL FOR but the JOIN structure is different because we take the case where the output is required to be in the same sequence as the input. To do this, each record processed has a link set to point to the following record. This is set when the following record is established in D2.

D1 opens files, sets initial links, and sets data areas. D2 gets the next record and makes links. D3 ends the loop that produces the parallel paths; D4 produces the parallel paths; and D5 is the parallel path. The JOIN occurs at D6 and uses a lockout mechanism to ensure that only one processor executes output at a time and in proper sequence. Where parallel processing is allowed, there are some functions that must be carried out by only one proc-

Example 3 shows how unjoined forks are implemented, which is the example discussed in Item 5 under Parallel Structures. Note that PREP is not used, no PPC counters are involved, and IDLE and ALSO are used instead of JOIN and AND.

explicitly the relationship between the FORK and the JOIN.

This process allows the number of paths to be data dependent, and even to be zero. Note that the PPC shows how many parallel paths are active. This concept has been described and defined in ALGOL by Wirth.58 He uses the term AND (which he attributes to van Wijngaarden, and which we borrow for consistency) but he does not show how the system controls the JOIN nor does he provide an explicit PARALLEL FOR. The programmer builds it himself. Wirth is correct in that a programming language does not specifically require the JOIN but the hardware does need a JOIN delimiter.

Example 3. Unjoined Forks
EXPLICIT PARALLEL PROCESSING DESCRIPTION AND CONTROL

Example 4. Serial File Processing

`INITIALIZE`

`ESTABLISH NEXT RECORD`

`END FLAG`

`AND (D2)`

`PROCESS AND MARK OUTPUT READY`

`LOCK`

`IS EARLIEST OUTPUT IN LIST READY`

`END FLAG`

`OUTPUT EARLIEST`

`UNLOCK`

`GO TO (D6)`

`CONTINUE`

ess at a time, for example, updating a record. Various hardware locks have been developed,9 and even some complex software locks.22,35 Wirth58 proposed a term SHARED in ALGOL to denote procedures that are not to be used in parallel.

Special Points to Note

First, we must note that these techniques presuppose that all programs use re-entrant code.

Second, we must provide some special protected functions such as “add to x” so that totals can be accumulated from each path, or a lock-out feature to temporarily restrict access to a section of data.2,22,35,43,59

Third, we must arrange the linkage in Example 4 so that the system can tolerate such situations as the case of an empty file, and one record being completed before the next is initiated.

This is necessary because the link from one record to its successor, used to control output, may not be established if a delay occurs such that a record is output before its successor is input. One alternative is to delay output; another is to provide an interim dummy linkage.

Fourth, we must make box D2 include choosing an insertion to the file. The full logic for this box determines whether the next record to be processed is

1. one record from the master file and a matching record from the detail file.
2. one record from the master file and no matching detail record.
3. one record from the detail file and no matching master record.

PROCESSOR SCHEDULING

Apart from the wish to run certain problems faster in a multiprocessor, the most interesting case in scheduling is the possibility of shared access to program segments by processors and the subsequent reduction in overlay overheads. It may seem at first that this would lead to more complex dynamic storage allocation than is desirable. However, it turns out that shared access arises naturally. There are three basic advantages that this scheme enjoys:

- Processes generate parallel paths dynamically one at a time on an “as needed” basis, which contrasts with the schemes proposed by Opler46 and Anderson.9 It limits scheduling overheads in that the process generates only records of parallel processes as they are encountered. For example, the number existing for an N-way PARALLEL-FOR will be only one plus the number initiated, however large N may be.
- Distribution of parallel paths among varying numbers of processors is an easy ad hoc process.
- The same basic data can be used by the scheduler to exploit any of the alternative advantages of parallel processing.

The following examples illustrate the last two points. Consider a system with three processors: I, II, and III. Consider that there are six tasks in the system. Each task has one or more processes live at a time. Each process is either active in a processor or is in one of three queues. A process in queue ‘a’ has the segments it needs in core, but no processor.
A process in queue 'c' does not have all its segments in core. A process in queue 'b' is awaiting termination of some peripheral I/O, in particular, perhaps the arrival of its needed segment in core. It is obviously desirable to have more overlay slots than processors to build up work for processors. The current segment needed by a process is shown in line 2. The status of each process is shown as either the identity of the processor executing it or the queue 'a', 'b', or 'c' where it is located.

In the case of equal priority processes we assert that the scheduler (i) prefers to initiate processes in queue 'a', (ii) prefers to get a segment wanted by most members of 'b', and (iii) gives chronological preference.

Table I shows how the system reacts with a set of equal priority processes and how a set of parallel paths builds up demand for its segments. Table II shows how the system reacts for priority classes and uses low priorities to fill the gaps.

### ASSETS OF THIS SCHEME

There are nine significant assets of this way of considering parallel processing:

1. It indicates profitable sources of parallelism in conventional program structures.
2. It enables some parallelism in FOR statements to be shown in existing programs with trivial changes.
3. It provides an easy, and hopefully popular, way to express parallelisms in all conventional programming languages.
4. It provides a simple mechanism to control parallel activities which could also be incorporated in hardware.
5. It requires only a small expansion to the bookkeeping required in operating systems.

### Table I

An example without priorities.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>STATUS</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Five processes exist, A1 through E1</td>
<td>Process</td>
<td>Al BI CI DI EI SI TI UI VI WI</td>
</tr>
<tr>
<td>Four have segments in core, Al through D1</td>
<td>Segment</td>
<td>A1 BI CI DI EI SI TI UI VI WI</td>
</tr>
<tr>
<td>There are active, A1 through C1</td>
<td>State</td>
<td>I II III I a c</td>
</tr>
</tbody>
</table>

A1 creates A2 which uses SI, and A2 goes into queue 'b'.

A2 needs next segment S2, A1 goes into queue 'c', D1 obtains processor I

D1 needs next segment T3, D1 goes into queue 'c', A2 obtains processor II

Now a new segment can be called to overlay T2. W1 has waited longest and E1 goes into 'c'.

A2 needs next segment S2, A2 goes into queue 'c'. E1 obtains processor II

W1 arrives and E1 goes into queue 'a'.

C1 needs next segment UI. C1 goes into queue 'c'. E1 obtains III.

Now E1 can be overlaid. E1 is chosen because two processes in queue 'c' need it. A1 and A2 go into queue 'b'.

### Table II

An example with priorities.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>STATUS</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Five processes exist, A1 through E1</td>
<td>Process</td>
<td>Al BI CI DI EI SI TI UI VI WI</td>
</tr>
<tr>
<td>Four have segments in core, Al through D1</td>
<td>Segment</td>
<td>A1 BI CI DI EI SI TI UI VI WI</td>
</tr>
<tr>
<td>Three are active, A1 through C1</td>
<td>State</td>
<td>I II III I a c</td>
</tr>
</tbody>
</table>

A1 creates A2 which uses T1 and goes into queue 'a'.

A2 preempts C1 and gets II, C1 goes into queue 'b'.

A1 needs next segment S2 and goes into queue 'c'. C1 gets I

Priority calls for III to be requested to overrun V1. Al goes into queue 'b', D1 goes into queue 'c'.

A2 creates A3 which uses SI and goes into queue 'a'.

A3 preempts C1 and gets I, C1 goes into queue 'a'.

VII overwrites V1, A1 goes into queue 'a'.

A1 preempts B1 and gets II, B1 goes into queue 'a'.

---

From the collection of the Computer History Museum (www.computerhistory.org)
6. It is applicable to all types of computer structures and applications.
7. It permits the dynamic changing of the number of processors available.
8. It permits the exploiting of a re-entrant code by individual programs.
9. It offers opportunities to reduce heavy page turning overheads.

REFERENCES—BIBLIOGRAPHY