A MULTIPROCESSOR SYSTEM DESIGN

Melvin E. Conway
Directorate of Computers, USAF
L. G. Hanscom Field
Bedford, Mass.

INTRODUCTION

Parallel processing is not so mysterious a concept as the dearth of algorithms which explicitly use it might suggest. As a rule of thumb, if N processes are performed and the outcome is independent of the order in which their steps are executed, provided that within each process the order of steps is preserved, then any or all of the processes can be performed simultaneously, if conflicts arising from multiple access to common storage can be resolved. All the elements of a matrix sum may be evaluated in parallel. The *ith* summand of all elements of a matrix product may be computed simultaneously. In an internal merge sort all strings in any pass may be created at the same time. All the coroutines of a separable program[1] may be run concurrently.

The problem is not so much finding procedures employing parallelism as it is finding computer systems which could handle the procedures without undue preplanning. A desirable system which flexibly accommodates a collection of identical, concurrently operating sequential processors should exhibit the following properties.

1. At every point in time the number of active processors should be the minimum of the number of processors in the system and the number of parallel paths in the program at that time.
2. If insufficient processors are available, program paths specified to be parallel should be executed serially.
3. Although the coder must specify all parallelism, he should have little concern about the number of processors in the system at execution time.
4. The means of specifying parallelism should be simply coded and rapidly handled by the system so that for highly parallel programs the processing time is inversely proportional to the number of processors, subject to the boundary condition that a one-processor system would run only slightly faster if the specifications of parallelism were removed from the program.

In short, a system which accommodates programs with parallel paths by means of a plurality of sequential computing elements should be dynamically self-scheduling. This paper suggests a design for such a system.

SPECIFYING PARALLELISM

Because a procedure can be thought of as originating at one point in its flowchart, all parallelism is the result of forks in flowchart paths. Figure 1 provides a convention for specifying such forks. Hereinafter, the word *fork* will have the meaning suggested by Figure 1. Parallel paths may rejoin at a *join.* A convention for drawing joins is also given in Figure 1.

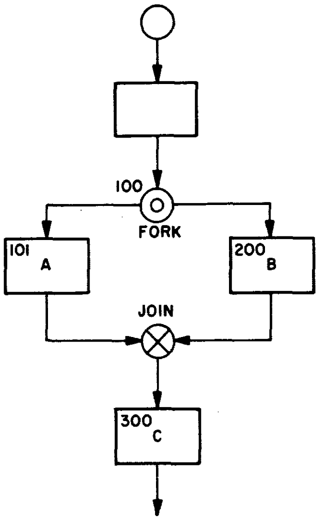

Figure 1. Conventions for drawing fork and join.

The fork and join in flowcharts have their counterparts in the FORK and JOIN instructions which are added to the instruction set of the system.* FORK is simply an instruction with two successors. It is written and acts like a branch instruction. However, if location 100 contains a FORK 200 instruction, then instructions at 200 *and* at 101 will be subsequently executed. The execution of a FORK instruction calls another processor into activity, if it is available. Notice that FORK has an associativity property; N parallel paths may be specified equally well by many possible arrangements of N—1 forks.

The JOIN, which is, in effect, the reverse of the FORK, has a vital additional job: it waits. In Figure 1, box C must not be begun until boxes A and B are completed. Assume that the coding for box A runs from location 101 through 105, that the coding for box B runs from 200 through 219, and that the coding for box C begins at 300. After the FORK at 100 is executed two processors are called to participate; one executes five instructions from 101 to 105, the other executes twenty instructions from 200 to 219. The processor finishing first, say at 105, should be released; the one finishing last then simply branches to 300.

In the case of an N—ary fork the only processor with a distinguished role is the one which finishes last, for it is the one which must branch. All others are released. If the N processors are operating independently, how does each know whether it is last to finish? Returning to the example, the information required to notify the last processor is available in the form of a counter at location 299. The FORK at 100 sets the counter to 2. Each processor, when it comes to the end of its parallel path, decrements the counter by one. The processor which produces zero as a result knows that it is last to finish. There are two JOIN instructions, at 106 and 220. Each one reads: JOIN 299. This means, "Decrement the counter at 299 by one. If the result is zero branch to 299 + 1. Otherwise release this processor." The FORK at 100 reads: FORK 200, 299, 2. This means, "Set the contents of 299 to 2. Then fork to 101 and 200."

If location 500 began a four-way fork to 503, 520, 540, and 560, all to recombine at a counter (called the *junction*) at 600, the coding would be thus:

```
500: FORK 520, 600, 4
501: FORK 540
502: FORK 560
```

This illustrates that a second kind of FORK is necessary. It forks but does not affect the junction.

So far we have described three new instructions peculiar to the system being developed. Here are two more. The first is a variation of JOIN which, instead of releasing its processor if it is not on the last path to finish, keeps it to do busy work. It reads: JOIN J, B and means, "Branch to B if something which ends up at J is still going on. Otherwise, JOIN J." It acts as follows. The counter at J is removed from storage and 1 is subtracted. If the result is nonzero, the *original value* is returned to J and a branch to B is executed. If the result is zero, J is set to zero and a branch to J + 1 is executed. We shall see later that this instruction is a generalization of the class of "Branch

* The fork-join notion has been around for a while. The equivalent of FORK is elsewhere given these names: in CL-II[2] and the Burroughs D825 AOSP[3], BRANCH; in the GAMMA 60[4], SIMU; in a conceptual machine discussed by Richards[5], BRT (Branch Transfer).

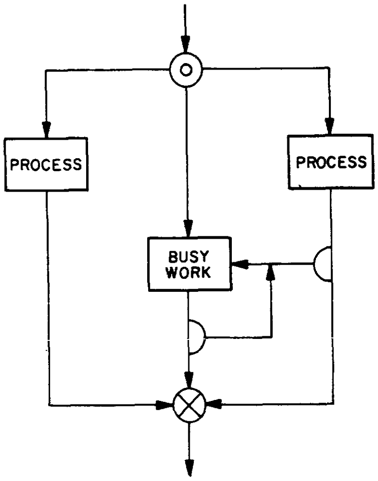

Figure 2. A convention for the "Branch to Busy Work" operation.

on busy I/O device" instructions. Figure 2 suggests a flowchart notation.

The other new instruction adjusts the value of the junction in the event that the possibility of execution of a FORK is conditional. It reads: FORK A, J and means, "Increment the value at J by one, then FORK A."

In summary, the five following instructions permit an adequate specification of parallelism:

```
FORK A, J, N
FORK A, J
FORK A
JOIN  J, B
JOIN  J
```

THE STATE WORD

One question which occurs to people thinking about multiprocessor systems may be stated thus: How much of the main memory should be private to each processor and how much should be "community" storage? The design to be presented here makes it clearly uneconomical to reserve any of the main memory for each processor. Indeed, if private storage is required, it belongs not to each processor but to each parallel flowchart path. The distinction between processors and paths is a crucial one; confusion over this matter seems to be muddying up much contemporary thinking about parallel processing. The four criteria stated in the introduction to this paper demand that the distinction be made. Processors have no identity of their own. During a computation they can be swapped, added, or removed without altering the results of the computation. What does have an identity of its own is a set of bits in each processor determining the state of the processor between instruction executions. This set of bits is called the *state word.* The notion of the state word will now be elucidated.

When the executive routine of a multiprogrammed single-processor computing system takes control from program A and gives it to program B it establishes an appropriate environment for the new program by storing all the processor registers used by program A in an area reserved for that program, and loading these registers from a similar area for program B. The content of such a reserved area preserves the state of a program at the time it is taken off the processor so that the same program can be later returned to the processor without any disturbance to the computation as a result of the interruption. We may call the content of a program's reserved area the state word for that program. Normally, a state word consists of at least an address (the sequence counter), and generally includes several arithmetic and index registers and a few indicator bits. We may call the aggregation of all the reserved areas for holding state words of inactive programs the *control memory.*

When the state word is loaded into the processor from control memory it occupies a set of storage positions the aggregation of which we might call the processor's *state register.* Thus, switching control from program A to program B may be conceptualized as a sequence of two processor operations: store state register into program A area in control memory; load state register from program B area in control memory.

When n programs share a common memory and a single processor the scheduling function consists of choosing the time intervals during which each of the n state words will occupy the single state register. This description of the scheduling function as a resource allocation can be generalized to the case wherein n programs share a common memory which is equally accessible to k identical processors: the scheduler attempts to optimize, according to some value scheme, the time-allocation of n state words to k state registers.

Consider a system with k processors, where $k \geq 2$. To take advantage of the potential overall speed increase without paying for state word transfer time we must build k flow paths, one for each processor, each capable of simultaneous operation between its processor and control memory. Ignoring timing restrictions in the control and main memories, we can see that for $1 \leq k \leq n$, where n is the number of state words (presumed fixed), the system speed (number of standard instructions executed per unit time) is proportional to k. For $n \leq k$ the system speed is constant.

In real life, of course, k is fixed and n varies with the amount of parallelism in the total system at any given time. What varies n? The FORK and JOIN instructions. FORK makes two state words from one, and JOIN (except the last one executed, for which the junction becomes zero) makes state words disappear. This suggests an easily implemented processor allocation algorithm: a processor executing a FORK sends one state word to control memory; one executing a JOIN halts until it receives a new state word from control memory. Richards[6] has shown that when the control memory is a single queue and when halted processors are given state words as soon as they are available, the allocation is not optimum in the sense of minimizing total execution time or maximizing processor duty cycle.

Figure 3 shows the system configuration as derived so far. Notice that there are two information subsystems. The control subsystem circulates state words, and the program subsystem shuttles instructions and operands back and forth. Note also that the only time that a channel between a processor and the control memory need be busy is after the processor executes a FORK or JOIN.

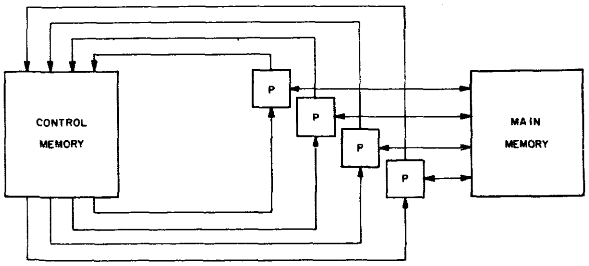

Figure 3. A tentative system configuration.

This design is so far unsatisfactory because serious flow bottlenecks can be expected at the two memories unless precautions are taken to avoid them.

Before we address this difficulty some observations are in order. First, notice that the FORK-JOIN approach provides no justification for distinguishing between parallelism within a program and parallelism between programs. The difference between simultaneous multiprogramming and a parallel algorithm is simply the position of the FORK instruction. This observation raises the hope that executive programs for a system of this sort will not be complicated by the parallel structure and may even be simplified by it.

Second, we might contemplate the role of the interrupt in this system. To rephrase the words of Buchholz[7] and others, interrupts have two distinct functions. The internal interrupt is triggered by the execution of a particular instruction and demands the insertion of a portion of code immediately following completion of this instruction. Overflow, divide check, and invalid address alarms exemplify the internal interrupt. The external interrupt is triggered by an external event not closely timed to the instruction currently being executed and which demands execution of a portion of code not necessarily related to the code being executed at interrupt time. The I/O operation complete and time clock interrupts are examples of this type.

External interrupts came into popular use when concurrent, program-controlled I/O was introduced. They attempted to control sequencing of parallel operations in a basically serial system. Because the code activated by an external interrupt could be executed in parallel with the code which is active at the time of interrupt (see the rule of thumb in the introduction to this paper), one might expect the handling of external interrupts to be different in the proposed system. In fact, external interrupts are unnecessary. Consider that an I/O instruction is simply a very lengthly one which may be executed in parallel with other code, such as computation and editing. Then simply precede it with a FORK and follow it with a JOIN, and all the functions of external interrupts are accounted for in a much more elegant manner. It is now seen how the "Branch to Busy Work" variation of the JOIN can be used as an I/O activity test.

Here is the first concrete illustration that the present structure simplifies executive programming. The elimination of the external interrupt provides simpler handling of interrupts for two reasons.

1. There are fewer interrupts to process and they all permit similar handling.
2. Internal interrupts are simpler to process because they are known not to occur at random. In particular, the routines processing internal interrupts can control further occurrences of interrupts during their durations.

While we are on the subject of simplification of the executive function, we might note that the extensive use of hardware in the processor allocation function can greatly simplify the executive program in comparison to a conventional multiprocessor system. Also, processor allocation in hardware helps make feasible goal number 4 stated in the Introduction.

The third observation we can make is that at any point in time during a computation there is no particular distribution of programs on the set of processors. Furthermore, observing the time history of any single processor reveals no particular sequence of programs or paths being serviced by that processor. This makes the problem of logging system usage by each program a nontrivial one for the system being discussed. This apparent anarchy also suggests that storage protection among programs might be hopeless; it will be seen that this is not the case.

Finally, we observe that with appropriate relief of certain bottlenecks in the system and with a certain class of highly parallel computations it may actually make sense to talk about speeding up the system by adding processors. The next two sections will attempt to show that the principal bottlenecks of the system are not essential. That is, they can be designed wide enough to match any prior choice of memory size and number of processors.

THE STORAGE SUBSYSTEM

In a system of the type being considered, particularly if its application involves servicing many independent programs, three problems related to the high-speed storage arise.

1. Complete storage protection must be incorporated in order to isolate the several programs.
2. Scavenging and allocation of storage to newly entering programs should not be an expensive process.
3. The effective service rate of the memory should not seriously depreciate the speed increase gained as a result of the addition of processors.

The first two problems can be solved fairly straightforwardly. The third problem is the kind which could keep a system like this from leaving the drawing board. However, there seems to be a sizeable class of applications for which the memory design presented here constitutes a solution to the third problem.

Assume for the sake of discussion that no individual program would require more than 2^{14} (16,384) words of storage and that the high-speed memory will never contain more than 2^5 (32) programs (including the executive) at any one time. All programs will be coded using a contiguous block of storage beginning at address zero.

When a program enters the high-speed memory it is assigned a five-bit program number by the executive. (Let zero be reserved for the executive program itself.) This program number is a constant of all state words in that program. Clearly then, the storage protection problem can be solved in principle by providing 2^{19} (524,288) words of storage and addressing the memory with a 19-bit "system address" which is a concatenation of the 14-bit address obtained from the program and the 5-bit program number. Such a worst-case design is of course not economical; there would frequently be large blocks of unused storage in the memory.

Now assume that the memory is divided up into small independent modules of, say, 2^8 (256) words each, such that the high-order 11 bits of a system address specify a module and the low-order 8 bits specify a word within a module. Now, instead of being forced to buy 2048 modules, let us elect to buy only 100 modules in a system. If no problem mix requires more than 25,600 words of storage (assuming also that no two programs share a module) then the left hand 11 bits of every meaningful system address form at most 100 possible module-selecting combinations. (See Figure 4.) We can then use a 100-word associative memory with 11-bit words to map the upper 11 bits of the system address into a module specification. See Figure 5.

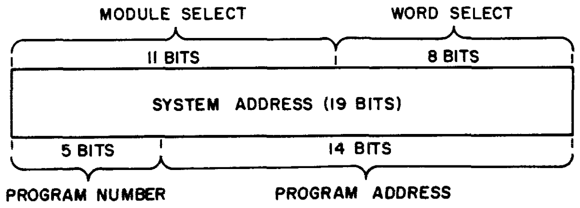

Figure 4. Showing the origin and use of th϶ 19-bit system address.

The storage allocation function of the executive consists of writing appropriate words into the associative memory, thereby assigning storage modules to programs. Notice that there is no inherent order or adjacency to the storage modules, so that scavenging unused modules for assignment to a new program does not require moving any existing programs.

In a multiprocessor system each processor would have its own associative memory, and conflict resolution would be accomplished by a switch called the Memory Exchange in Figure 6.

The arrangement of Figure 6 meets the three storage problems, as explained below.

1. The "system address" notion, together with the ground rule that no two programs can coexist within a module, insure that a program can access only its assigned modules.

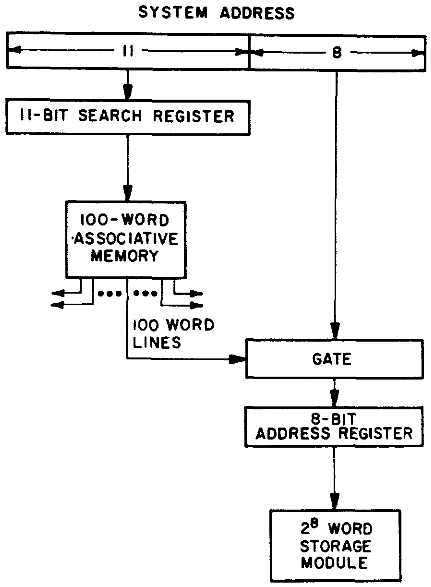

Figure 5. The associative memory selects a storage module from the eleven high-order bits of the system address.

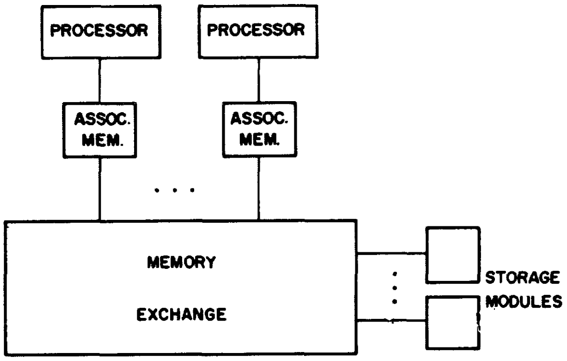

Figure 6. A storage subsystem.

2. The allocation of storage requires no movemet of memory conntents, only the simultaneous writing in all associative memories of as many words as there are modules being allocated.

3. If there is only one problem in the system and it contains many simultaneous references to the same module (as might happen with matrix operations) then this memory system provides no advantage. The other extreme, in which the memory is no bottleneck, is that wherein the system contains many serially coded programs.

A SYSTEM CONFIGURATION

One more major change over the system of Figure 3 remains to be made: the control and main memories will be consolidated.

The control memory has the following properties.

1. It contains logic for queuing state words.
2. Although it is active only when a FORK or JOIN is being executed, there are brief times when it might be very busy.

It should also have the following property.

3. The executive should have the facility to allocate modules from the system "storage pool" to the control and main memories as required. In this case the control memory would belong to program zero, the executive.

The system is made homogenous by isolating the logic functions of control memory into a second kind of processor, the control processor (CP). The control processor has a fixed program, presents the same interface to the memory as an arithmetic processor (AP, formerly called processor), and provides a communication path for state words between AP's and memory. Figure 7 shows the system configuration. The Dispatcher is a switching device for connecting AP's and CP's.

Figure 8 shows the flow of state words. Path A gives the flow of a state word after AP$_1$ executes a FORK. Path B gives the flow of a state word after AP$_2$ executes a nonfinal JOIN. The system should accommodate a number of CPs' which will be in balance with the number of AP's.

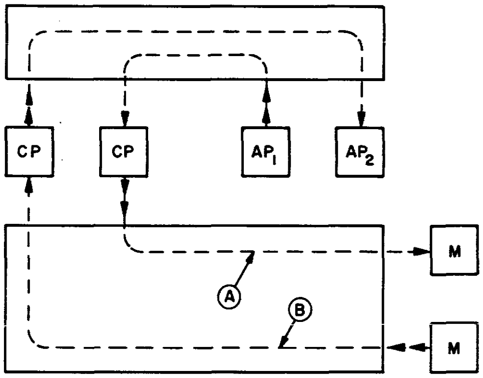

Figure 8. Path of state words.

Finally, an I/O processor (IP) provides a communication channel to the outside world. When an AP decodes an I/O instruction it sends the state word and the decoded instruction down the Dispatcher to the appropriate IP. This frees the AP. When the IP finishes the I/O instruction it sends the state word to control memory or to an AP. Figure 9 shows the final system configuration.

SOME PROGRAMMING CONSIDERATIONS

One of the most intriguing aspects of programming the system developed here arises from the possibility that the same section of code can be executed simultaneously in two parallel paths. This might happen n^2 times in the addition of two n $\times$ n matrices, or it could happen a small number of times in the simultaneous use of the same cosine routine in a tracking calculation. The usefulness of this possibility (indeed, the difficulty of avoiding

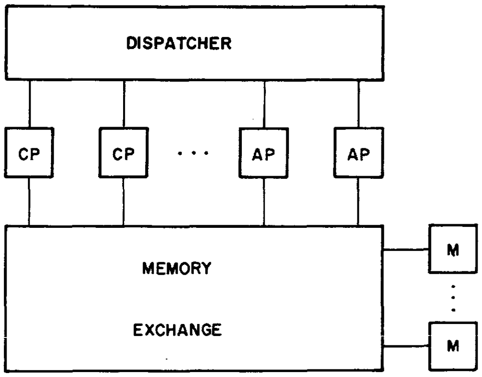

Figure 7. A more homogeneous system configuration.

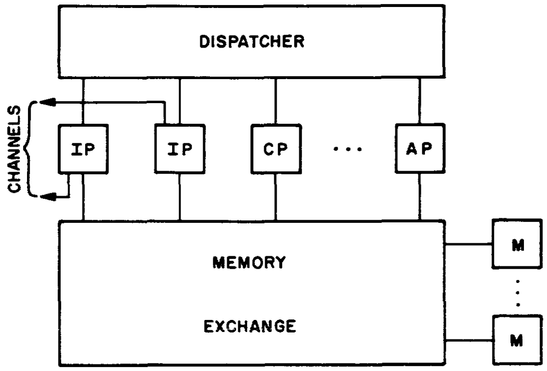

Figure 9. Final system configuration.

it) provides arguments for including index registers in the state word.

Consider the multiple use of the cosine subroutine. The exit address cannot be stored in a fixed memory location; it could be wiped out at any time by another call to the subroutine. A little reflection reveals that if there is any information unique to the subroutine call at the time of entrance to the subroutine it is in the state word. (Consider that the two calls to the subroutine might be the same instruction.) In the more general case, consider the subroutine's use of parameters and temporary storage. Neither can any of these be in fixed memory locations. One answer is to stack parameters (including the exit address) in memory and to use an index register to point to the stack. This is not a very good answer, however, because the stack is not in general last-in-first-out. Another possibility which could bear investigation is the use of control memory for subroutine parameters.

Now consider the n × n matrix addition. In a FORTRAN expression of this process the DO implies serial repetition of the addition coding. This process could be done in parallel, and so we arrive at the parallel DO instruction whose implementation generates n state words, each with a different value in a specified index register. When the loop is short the use of a junction counter to determine the end of the loop would create a bad traffic jam in the memory; this and other reasons make the parallel DO instruction impractical in spite of its appeal. However, if there are enough instructions in the scope of the loop, it would be justified to use a DO or simply to loop on a FORK instruction.

CONCLUDING REMARKS

Fundamental to the concepts presented here is the principle, not yet commonly accepted, that parallel paths in a program need not bear fixed relationships to the processors of a multiprocessor system executing that program. In many applications, if the number of parallel paths generally exceeds the number of processors, adding a processor will increase the system's effective speed. This fact emphasizes that there are two research objectives whose fulfillment will render the system design presented here a practical improvement over the present state of affairs.

1. A search should be made for parallelism in commonly used algorithms. The effort of such a search would be greatly reduced by the addition of the equivalent of FORK and JOIN to the common publication languages, for example, ALGOL.

2. Memories permitting simultaneous access to any set of words should be developed. As long as memories are slower than processors, simultaneous access is the only alternative to higher memory speed for increasing overall processing rates.

REFERENCES

1. CONWAY, M., Design of a Separable Transition-Diagram Compiler, *Comm. ACM* 6 (July 1963), p. 396.
2. CHEATHAM, T. E., JR., and LEONARD, G. F., An Introduction to the CL–II Programming System, Computer Associates Incorporated Report No. CA-63-7-SD, August 1963.
3. THOMPSON, R., and WILKINSON, J., The D825 Automatic Operating and Scheduling Program, *Proc. SJCC,* 1963, p. 41.
4. DREYFUS, P., Programming on a Concurrent Digital Computer, Notes of University of Michigan 1961 Engineering Summer conference, *Theory of Computing Machine Design.*
5. RICHARDS, P., Parallel Programming, Technical Operations Incorporated Report No. TO–B 60–27, August 1960, p. 4.
6. *Ibid.*, p. 6.
7. BUCHHOLZ, W. (Ed.), Planning a Computer System: Project Stretch, McGraw-Hill, 1962, p. 136.