Programming for Parallelism

Alan H. Karp

IBM Palo Alto Scientific Center

The state of the art of parallel programming and what a sorry state that art is in.

In the last few years we have seen an explosion in the interest in and availability of parallel processors[1] and a corresponding expansion in applications programming activity. Clearly, applications programmers need tools to express the parallelism, either in the form of subroutine libraries or language extensions. There has been no shortage of providers of such tools.[2-7] These tools allow the applications programmer to express the parallelism explicitly. There are also projects underway to provide automatic parallelization of sequential code.[8-10]

While this article presents language extensions in use at the end of 1985, it is not comprehensive. I will concentrate on the following aspects of the problem:

- Scientific programming: Characterized by a high degree of floating-point computation, usually coded in Fortran. The discussion is not relevant to logic programming such as AI applications expressed in Lisp or Prolog.
- General-purpose machines: Machines capable of running a wide variety of applications with reasonable efficiency. In special-purpose machines such as image, signal, and array processors, and systolic arrays the algorithms are coded into the hardware. Even general-purpose array processors rarely exhibit the parallelism of interest here.
- MIMD processors: Multiple-instruction, multiple-data machines, not vector processors, which are single-instruction, multiple-data (SIMD) machines. I treat the vector unit, if available, in much the way I treat the floating-point multiplier, as another functional unit. I will not discuss parallel SIMD machines here, like the Illiac IV,[11] ICL DAP,[12] MPP,[13] and Connection Machine.[14] Nor will I discuss microinstruction machines. For example, each processor in a systolic array[15] is not a complete computer in the sense that an entire application cannot be run on it.
- Moderately parallel systems: No more than tens of processors. The tools described leave it up to the programmer to distribute the work, even to the extent of having a different program running on each processor. Massively parallel systems of the order of a thousand or more processors are quite different. At this time there are no general-purpose, MIMD machines in this class that are widely available, so no one has experience programming them.
- Fortran: The most commonly used language for scientific computers. Since most of the work to date has been done in Fortran, and because most scientific programmers are familiar with it, I use Fortran for all the examples. It is possible to use these constructs in other languages, but "modern" languages like Pascal and Ada that attempt to eliminate side effects are quite different. Side effects represented by Fortran COMMON data present important advantages and problems for the application programmer.
- Explicitly declared parallelism: Parallelism controlled by the programmer. I describe a style in which the programmer controls the parallelism. Even though great strides have been made in automatic parallelization, the only automatic system available to date[10] is limited to individual

0018-9162/87/0500-0043$01.00©1987IEEE

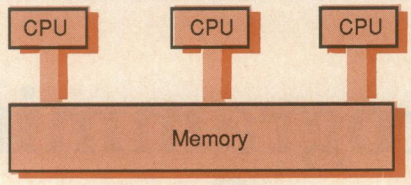


Figure 1. Schematic of a shared memory system.

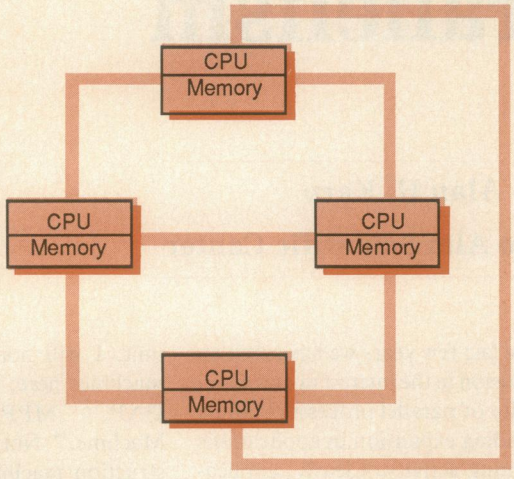


Figure 2. Schematic of a fully interconnected message passing system.

loops. Parallelism at a larger granularity must still be specified by the programmer.

Rather than enumerate all possible ways of describing a particular type of parallelism, I use generic notation. This notation indicates how the parallelism is described without worrying about details of implementation. For example, I will not discuss locks, semaphores, and monitors that can be used to implement the constructs I describe. Actual implementations are, and must be, somewhat more complicated than those described here.

Taxonomy

A common myth is that the programmer does not need to understand the hardware being used. Like all myths, it contains a grain of truth. However, when performance becomes critical, programmers have used their knowledge of paging, cache size, vector lengths, and so forth to tune their programs.

The situation is worse for parallel processors than for uniprocessors because of the wider variety of architectures. Although a number of efforts are being made to write portable programs for parallel processors,[7,16] some algorithms will run poorly on certain architectures. In addition, the coding style used will frequently depend on the type of parallelism in the hardware.

Rather than attempt a complete taxonomy, I restrict the classification to those aspects that affect coding style. I put all systems into one of three classes: shared memory, message passing, or hybrid. While I don't expect such a simple scheme to describe the variety of machines possible, it is sufficient to demonstrate the variety of programming styles used.

It is common to distinguish between processes and processors. A process is a running program; a processor is a computer. In many operating systems it is possible to have many processes running on one processor. For the remainder of this article, I ignore this distinction and assume each processor has only one process. Such an assumption is reasonable for scientific codes running on a stand-alone parallel processor.

Shared memory. A shared memory machine has a single global memory accessible to all processors. The simplest configuration is shown in Figure 1. Each processor may have some local memory, such as registers as on the Cray X-MP,[17] or the cache on the IBM 3090,[18] but I assume the operating system presents to the user the image of totally shared memory. For example, although the cache on the IBM 3090 is local to a processor, cross-cache validation makes the cache transparent to the user. The user need not worry that a piece of data is in the cache of the wrong processor, since the operating system makes sure that the correct value is delivered. In fact, the programmer is not allowed to explicitly use the cache in any way, making the system look like it has a single, global memory.

Other types of shared memory organizations are possible. For example, the Alliant FX/8[10] has several processors run out of a common cache. A completely different approach was taken by the NYU Ultra machine.[19] Although the memory is distributed, the system fits my definition of a shared memory machine because any processor module can be connected to any memory module.

A key feature of shared memory systems is that the access time to a piece of data is independent of the processor making the request. If the code running on two processors can be swapped without affecting performance, the system has a true, shared memory. This is not to say that there is no memory contention. Such issues as page faults and memory bank conflicts still affect performance, just as in uniprocessors. Clearly, the aggregate memory bandwidth will limit the number of processors that can be accommodated on the system.

Message passing. Message passing systems are configured so that some memory is local to each processor but none is globally accessible. The only way for the application to share data among processors is for the programmer to explicitly code commands to move data from one node to another. The time it takes for a processor to access data depends on its distance from the processor that currently has the data in its local memory. Therefore, in contrast to the shared memory systems, the performance of an algorithm will depend on how well the location of data matches up with its use.

Figure 2 shows a fully interconnected message passing system, with each processor having a direct connection to every other processor. Such a scheme is impractical when there are a large number of processors. Therefore, designers of message passing systems are forced to pick less dense wirings. The particular choice made has an important influence on the algorithms to be run on the machine. An algorithm designed for one machine can

perform very badly on a different machine.

A large number of connection schemes have been used.[20] The simplest approach is to connect the machines in a ring with each processor talking to its two nearest neighbors. In such a machine, it takes a time proportional to the number of processors to send data to each processor.

A machine with a denser wiring is a mesh machine in which the processors are connected in a two-dimensional grid. Each processor talks to its north, south, east, and west neighbors. If there are p processors in the machine, it takes time proportional to $\sqrt{p}$ to send data to each processor.

An even denser wiring is provided by a hypercube interconnection, one of today's most popular designs.[21-23] Each processor is assigned a binary id number $0 \leq n \leq 2^d - 1$, where d is called the *dimension* of the cube. Two processors are connected through a port if their identification numbers differ only in the corresponding bit in the id number. Thus, processor 0000 is connected to processor 1000 through the left-most communications port of each processor. The machine is called a hypercube because the connection scheme can be pictured as a cube for $d = 3$. Each node is placed at a corner and the direct links become the edges of the cube. The advantage of this configuration is that the largest distance between processors is proportional to $\log_2 p$.

There exists a very simple connection system with a maximum distance of two from any processor to any other processor: a star connection. In such a configuration, a central processor connects to every other machine. Because of the large amount of traffic that the center machine must handle, it often differs from the rest of the processors.[24] Although the maximum distance between processors is independent of p, the processor in the middle must be able to handle all the traffic generated, which limits the number of processors in such a system.

Hybrid. Hybrid systems have some of the properties of shared memory systems and some of the properties of message passing. As illustrated in Figure 3, all memory is local to a given processor, but the operating system makes the machine look like it has a single, global memory. Thus, programs are written as if for a shared memory system. However, data must be laid out as if for a message passing system if the best performance is to be obtained, since the access time depends on the distance between the owner of the data and the requester. The IBM RP3,[25] BBN Butterfly,[26] and Cedar[27] are examples of hybrid machines.

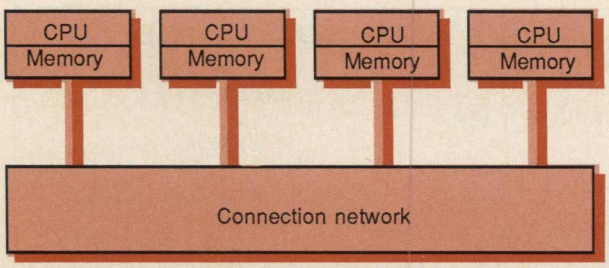


Figure 3. Schematic of a hybrid machine.

As far as the programmer is concerned, hybrid systems are coded like shared memory systems. Even the bugs in programs are the same as for shared memory. However, the performance considerations resemble those of message passing machines. Fortunately, the penalties for poor data layout are often considerably smaller on hybrid systems. Message passing systems typically take hundreds to thousands of machine cycles to deliver a message, while hybrid systems deliver data from a remote memory in tens of cycles. Even so, data layout is key to algorithm performance, and the aggregate communications speed is a limit on the number of processors that can be accommodated.

Comments on taxonomy. While Hockney[28] was careful to distinguish architecture from function, I am more interested in the programmer's view of the system. Such a view necessarily mixes the actual hardware with the picture of the machine presented by the available software. For example, a group of processors sharing a common memory would be classified as a message passing system if the software tools only used the shared memory for message buffers. However, a clever programmer who managed to use these buffers to share data would think of the machine as a shared memory system.

My classification scheme also includes questions of performance, something intentionally left out of other taxonomies. In particular, the principal difference between a hybrid and a shared memory system is one of performance. A programmer interested only in correct operation can treat a hybrid system as if it were a shared memory system. This approach is reasonable if the penalty for accessing data out of the local memory is small. However, the programmer must be aware of the details of the hardware implementation in order to produce efficient code. Even a factor of two delay in getting data can seriously degrade performance.

Software tools

Parallel processing hardware does the programmer no good without a means of describing the parallelism to the system. In large measure, the type of parallelism selected by the programmer depends as much on the software tools as on the underlying hardware. This section describes some of the tools that can be used to make a program run on several processors.

Message passing. Message passing systems need only two basic functions added to the standard language support, SEND and RECEIVE. Of course, most implementations include a variety of mode setting and query functions not discussed here.

SEND is used to send a message containing data from one processor to another. There are actually two forms of SEND. One continues processing immediately on dispatching the message; the other waits to make sure the message has arrived (but not necessarily been read into the memory of the recipient). The latter is called a *blocked* SEND and the former, an *unblocked* SEND. At a minimum, the arguments include a destination, the message length, and an array containing the message. Other arguments often used are a status word, a port address or routing information, and a flag to indicate whether or not to wait for an acknowledgment.

In general, a blocked SEND is used only in the presence of unreliable communications. For example, if the processors do not maintain message queues, a message will be rejected until the previous message has been received. If it is important that all messages be sent in a particular order, a

```
      parallel do 10 I=1,3
        IF (I.EQ.1) THEN
          .....
        ELSE IF (I.EQ.2) THEN
          .....
        ELSE IF (I.EQ.3) THEN
          .....
        ENDIF
   10 CONTINUE
```

```
      parallel case
          ....
        parallel
          ....
        parallel
          ....
      end parallel case
```

Figure 4. PARALLEL CASE code.

blocked SEND must be used. (I assume that the operating system will continually try to transmit a rejected message from a blocked SEND.)

A RECEIVE is used to read a message sent from another processor. It can also be blocked or unblocked. At a minimum, the arguments include an array to contain the message and the length of this array. Other options frequently used are the processor id of the sender or the input port to be read, a status flag, and an indication of whether or not to send an acknowledgment.

Both blocked and unblocked modes are needed. If the algorithm requires a specific piece of data from another processor, the programmer codes a blocked RECEIVE. The receiver then waits for the data to arrive. Unfortunately, blocked RECEIVE can lead to a deadlock in which two processors are waiting for data from each other.

Unblocked RECEIVE has two uses. The most common use is to implement a global receive, in which the processor needs to receive messages on several input ports but the order doesn't matter. If the port id is an argument to the RECEIVE, an unblocked RECEIVE allows the program to continually test the input ports and read the data as it arrives. This requirement could be met by providing a RECEIVE FROM ANY construct.

The other use for unblocked RECEIVE is asynchronous input. Here the program continually checks the input port for a message. If there is no message, one piece of work is done; if there is a message, a different piece of work is done. In general, such a scheme is useful for load balancing. If the message carries data needed by high priority work, it is still possible to do low priority work until the data arrives. If a blocked RECEIVE were used, the processor would be idle until the message arrived.

Shared memory. For a number of reasons, the language extensions needed by shared memory systems are much more extensive than those for message passing machines. First, there is the need to distinguish which data is private to each processor and which is known to all processors. Second, because data is shared, synchronization is needed to prevent out-of-sequence access to memory. Primarily, though, the shared memory allows an entirely different style of programming that has several modes of operation, each requiring a different set of language extensions.

There are two commonly used ways to divide up the work in a shared memory system. In the fork-join style, a process will spawn subprocesses, a FORK, and wait for them to finish, a JOIN. In the SPMD (single program, multiple data) style, each processor runs the same program but executes different code depending on its processor id or data in shared memory.

Both fork-join and SPMD programs need a means to restrict access to code. A *critical section* contains code that gets executed by all processors one at a time. It is usually used for reduction operations such as summing into a global variable. A *serial section* is code to be executed by only one processor and skipped by all others. It is usually used to initialize global data.

Synchronization is also needed. In message passing systems a blocked RECEIVE synchronizes; in fork-join, the JOIN serves this purpose. In SPMD programming, other constructs are needed. A *barrier* is a point in the code where all processors wait for the last one to arrive. A barrier is dangerous because a processor might branch around it, which causes all the other processors to wait until the program terminates. Another way of synchronizing is the WAIT UNTIL construct. Here each processor continually checks shared memory to see if some particular condition is met. In contrast to the barrier where the processors wait at a specific line of code, each processor can WAIT UNTIL the same condition is met from different parts of the program. One common use is to wait until an input or output operation finishes before continuing the computation.

Probably the most common SPMD construct is the PARALLEL DO. Since the cost of sharing data is very small in shared memory systems, programmers often parallelize their code at the DO-loop level. If all iterations of a loop are independent, each processor can run a different subset of the loop index range as long as each value of the index gets used exactly once.

Two mechanisms are used to distribute the work in the loop to the processors. A *self-scheduled* PARALLEL DO works by giving the first value of the loop index to the first processor to arrive, the second index to the second processor, and so forth. A processor that reaches the end of the loop returns to the top to get more work. A *prescheduled* PARALLEL DO works by partitioning the loop ahead of time so that each processor will do a certain set of loop indices, no matter how long each one takes.

Self-scheduled PARALLEL DO provides for automatic load balancing, since processors get more to do as soon as they have finished with their work. However, assuring that each processor gets a unique value of the loop index forces some synchronization not needed for prescheduled loops.

The choice between self- and prescheduled loops depends on both the application and the hardware. If the work is naturally load balanced, and the synchronization cost is high, then prescheduling is preferred. If the amount of work depends on the loop index or if the synchronization cost is low, then self-scheduling will perform better.

Another useful construct is the PARALLEL CASE. Its structure is similar to a standard CASE statement available in several languages. It differs from the usual definition in that all cases get executed in parallel. It can be implemented with a PARALLEL DO, but PARALLEL CASE leads to more readable code as shown in Figure 4.

Simple problem

The best way to get a feel for what the programming tools described in the previous section imply is to look at a simple example, summing a list of 1,000,000 numbers. Actually, this problem is nontrivial if the best performance is to be achieved, and numerous algorithms have been proposed.[12] Since we are interested

in coding style, I will discuss only the simplest method.

Figure 5 shows a program that could be run on a uniprocessor. The main program reads in the numbers to be summed, calls a subroutine to do the arithmetic, and prints the result. The subroutine initializes the sum, adds the numbers, and returns. I have separated the work in this way to illustrate several of the coding styles.

A simple notation has been used in the sample codes. Code shown in uppercase is standard Fortran; code in lowercase represents language extensions. While these codes have not been run, a desk check has been done.

```
      PARAMETER ( N = 1000000 )
      DIMENSION A(N)
      READ (*) A
      CALL SUMSUB (SUM,A,N)
      WRITE(*) SUM
      END
C
      SUBROUTINE SUMSUB (SUM,A,N)
      DIMENSION A(N)
      SUM = 0.0
      DO 10 I = 1, N
         SUM = SUM + A(I)
   10 CONTINUE
      RETURN
      END
```

Figure 5. Uniprocessor version.

Shared memory. One of the first things tried on a parallel processor was the fork-join approach shown in Figure 6a. We can see how the constructs introduced in "Software tools" are used to get the program running on more than one processor. First, we declare the shared data GLOBAL. Each processor will have a private copy of any variable not explicitly declared as shared. In order to divide up the work equally, we need to know how many processors are available. It is common practice to code all programs independently of the number of processors. This convention not only enhances the portability of the code, but makes it easier to debug on a single processor. I assume the system will provide the number of processors through the system variable NPROCS.

Work is divided among the processors using the CREATE function. Each time a CREATE statement is executed, a task is generated that runs the indicated subroutine, here SUMSUB. Each task works on a different part of the array, as indicated by A(IS). It is important that the address of A(IS) be computed before the CREATE is completed. If we rely on the created task to fetch the address of A(IS) from shared memory, we may find that the main routine has changed it before the created task gets started.

Once the main program has finished distributing the work, it does the remaining part of the job. When the main program reaches the JOIN statement, it waits until all other tasks have completed, at which time the result can be printed.

All the computational work is done in the subroutine SUMSUB. Here we see the use of a serial section and a critical section. Since SUM is a global variable, we could get wrong results if all processors were allowed to initialize the variable so we use

```
      PARAMETER ( N = 1000000 )
      global A(N), SUM, N
      READ (*) A
      INC = (N+nprocs-1)/nprocs
      IS = 1
      DO 10 J = 1, nprocs-1
         create ( 'SUMSUB', SUM, A(IS), INC)
         IS = IS + INC
   10 CONTINUE
      CALL SUMSUB ( SUM, A(IS), N-IS+1 )
      join
      WRITE (*) SUM
      END
C
      SUBROUTINE SUMSUB ( SUM, A, N )
      DIMENSION A(N)
      serial section
      SUM = 0.0
      end serial section
      DO 10 I = 1, N
      critical section
         SUM = SUM + A(I)
      end critical section
   10 CONTINUE
      RETURN
      END
```

(a)

```
      SUBROUTINE SUMSUB (SUM,A,N)
      DIMENSION A(N)
      serial section
         SUM = 0.0
      end serial section
      SUMLOC = 0.0
      DO 10 I = 1, N
         SUMLOC = SUMLOC + A(I)
   10 CONTINUE
      critical section
         SUM = SUM + SUMLOC
      end critical section
      RETURN
      END
```

(b)

Figure 6. Fork-join (a) and fork-join with good performance (b).

```
      PARAMETER ( N = 1000000 )
      DIMENSION A(N)
      READ (*) A
      INC = (N+nprocs-1)/nprocs
      IS = 1
      DO 10 J = 1, nprocs-1
         send ( J, 'INC:1, A(IS):INC' )
         IS = IS + INC
10    CONTINUE
      CALL SUMSUB ( SUM, A(IS), N-IS+1 )
      DO 20 J = 1, nprocs-1
         receive from any ( 'SUMJ:1' )
         SUM = SUM + SUMJ
20    CONTINUE
      WRITE (*) SUM
      END
```

(a)

```
      PARAMETER (N = 1000000,INCR=(N+nprocs-1)/nprocs)
      DIMENSION A(INCR)
      receive ( 0, 'INC:1, A:INC' )
      CALL SUMSUB ( SUM, A, INC )
      send ( 0, 'SUM:1' )
      END
```

(b)

```
      PARAMETER ( N = 1000000 )
      DIMENSION A(N)
      INC = (N+nprocs-1)/nprocs
      IF ( procid .EQ. 0 ) THEN
         READ (*) A
         IS = 1
         DO 10 J = 1, nprocs-1
            send ( J, 'A(IS):INC' )
            IS = IS + INC
10       CONTINUE
         CALL SUMSUB ( SUM, A(IS), N-IS+1 )
         DO 20 J = 1, nprocs-1
            receive from any ( 'SUMJ:1' )
            SUM = SUM + SUMJ
20       CONTINUE
         WRITE (*) SUM
      ELSE
         receive ( 0, 'A:INC' )
         CALL SUMSUB ( SUM, A, INC )
         send ( 0, 'SUM:1' )
      ENDIF
      END
```

(c)

Figure 7. Message passing code for processor 0 (a), message passing code for processors other than 0 (b), and message passing for SPMD code (c).

a serial section. We could have set SUM = 0 in the main routine, but perhaps we don't want to rely on the user remembering to initialize. (The real reason is that I didn't want to pass up the opportunity to illustrate a serial section.)

We need the critical section in the loop because SUM is a global variable. If processor 1 and processor 2 each fetch the old value, add their respective contributions, and store the new value, the program will produce an incorrect result. The critical section guarantees that only one processor can update the global variable SUM at any instant.

Even though the code in Figure 6a will produce correct results, it suffers from what is called a *performance bug*; it runs much slower than it should. The reason is the critical section within the loop. Since only one processor is allowed to update the value at any time, only one addition can be done at a time. Therefore, we have lost almost all the possible parallelism in the code. The code in Figure 6a will run slower than the uniprocessor version because of the synchronization overhead incurred at the critical section.

The code in Figure 6b, where we sum the data into a local variable, will perform much better. Since each processor has its own copy of SUMLOC, all the arithmetic in the loop can be done in parallel. At the end of the loop each processor adds its contribution to the global sum in a critical section. Now the critical section is encountered only once per processor instead of once per addition.

This routine has a curious kind of side effect. Fortran programmers are used to calling subroutines and having the values of variables in the calling sequence and in COMMON changed. Normally though, except for knowing the data types of the variables in the calling sequence and COMMON, the person writing the subroutine need not know what is happening in the calling routine. Such is not the case here. The person writing SUMSUB must know that SUM is a global variable. The absence of any such indication within the subroutine is a serious problem for debugging and maintenance.

Message passing. The coding style on a message passing system is quite different from that used on a shared memory system for two main reasons. First, data must be explicitly moved from the memory of one processor to the memory of another. Second, there is often no master processor to spawn tasks as in Figure 6a. This second difference is due in part to the distributed memory. For a master processor to create tasks, code would have to be physically moved from the master processor to each slave. The communications cost of such a transfer makes this approach impractical. Therefore, it is usual to load the code for each processor once at the start of the job. Figure 7a shows the program that runs on processor 0, and Figure 7b shows the code running on all other processors.

Processor 0 reads the data and distributes it using the SEND command. The arguments can be interpreted as saying

> SEND a message to processor J consisting of one word starting at the address of INC and INC words starting at the address of A(IS).

Next, processor 0 calls SUMSUB to add up the rest of the numbers. After that is done, it goes into a loop to get the partial sums from the other processors. The RECEIVE

FROM ANY says to read one word from any processor into SUMJ. Once all partial sums have been received, the total is printed.

All the other processors run the program shown in Figure 7b. As soon as the data arrives at the communications port of a processor waiting at a RECEIVE, it can copy the data into its local memory and proceed. Each receives a message from processor 0 consisting of one word to be stored as INC and INC words to be stored into array A. Each then calls a version of SUMSUB identical to that in Figure 5. On return from SUMSUB, each processor sends its contribution to processor 0 and exits. Since this is the main routine on these processors, the job terminates. In order to compute more than one sum, the programmer would have to code a loop with an appropriate termination condition.

Another way to manage the code in such an environment is to program in the single program, multiple data style. When coded this way, each processor runs the same code unless a processor-dependent control is used. An example of SPMD programming is shown in Figure 7c.

At the start of the job, each processor is in the same state except for a unique identifier, the PROCID. Each processor first computes its own copy of INC. On reaching the first IF statement, all processors but number 0 skip immediately to the RECEIVE where they wait for data to arrive. Processor 0 reads the data and distributes it using the SEND command. On finishing with SUMSUB, each processor (other than processor 0) sends its contribution to processor 0 and exits. Processor 0 sums the partial sums and prints the result.

Notice that the main routine is somewhat more complicated than the shared memory version. To compensate for this complexity, SUMSUB is much simpler and independent of the architecture of the machine. In addition, there are no problems of data dependence, and all synchronization is handled by the SEND and RECEIVE commands. However, this example does not require any global synchronization, which would be more complicated than on a shared memory system.

Several features of this code affect the elapsed time needed to run the program. First, there is the cost of communication. There will be some overhead in routing the data through the interconnection network in addition to the time taken to physically move the data. This overhead may be a small part of the communication cost of sending a long list of array values, but it is certain to dominate the cost of sending the partial sums back to processor 0. Second, there is the problem of load balancing. It is conceivable that some of the processors will have completed their work before the last one has received all its data. We say the load balancing is poor because some processors are idle while useful work remains to be done.

```
PARAMETER ( N = 1000000 )
global A(N), SUM, N, JSYNC
DATA JSYNC/0/
IF ( procid .EQ. 0 ) THEN
   READ (*) A
   SUM = 0.0
   JSYNC = 1
ENDIF
INC = (N+nprocs-1)/nprocs
IS = INC*procid
INCR = MIN ( INC, N-IS )
wait-until ( JSYNC .NE. 0 )
CALL SUMSUB (SUM,A(IS+1),INCR)
barrier
IF ( procid .EQ. 0 ) WRITE (*) SUM
END
```

Figure 8. SPMD code.

SPMD shared memory. The SPMD coding style is not limited to message passing systems.[6,29,30] Figure 8 shows how SPMD could be used on a shared memory system. Subroutine SUMSUB is identical to that in Figure 6b.

As with the previous shared memory example, some of the data must explicitly be declared GLOBAL. As with the message passing example, all processors but number 0 skip the code that reads the data. However, we do not have the SEND-RECEIVE mechanism to synchronize the processors so we use the WAIT UNTIL. The WAIT UNTIL construct is an example of *event* synchronization. In this example, each processor continually checks global variable JSYNC until it takes on a value different from 0. On some systems the memory location containing JSYNC will be a *hot spot*. Special hardware is often needed to prevent such hot spots from degrading system performance.[19,25,31]

While they are waiting for JSYNC to be set, all processors compute their own copies of the local variables INC, INCR, and IS. While these variables could be made global and computed in a serial section, the overhead in any extra synchronizations and additional memory contention would outweigh any possible savings.

Once JSYNC = 1, the waiting processors call SUMSUB. Next, we synchronize with a barrier to prevent processor 0 from writing SUM until all processors have finished adding their contributions. As soon as the last processor reaches the barrier, they all continue processing.

Realistic problem

While a simple problem like summing a series of numbers would appear to be a trivial task, we have seen that there are some subtle points to consider on a parallel processor. In this section, we will look at these points again with a nontrivial problem, solving systems of linear equations.

We want to solve the system of equations $\mathbf{A}\ \mathbf{x} = \mathbf{b}$. The best method for general, dense matrices is Gaussian elimination to perform an LU decomposition followed by a forward elimination and a backward substitution. The procedure is

(1) Use Gaussian elimination to find two triangular matrices **L** and **U** such that $\mathbf{A} = \mathbf{LU}$.

(2) Solve the lower triangular linear system $\mathbf{L}\ \mathbf{y} = \mathbf{b}$ by forward elimination.

(3) Solve the upper triangular linear system $\mathbf{U}\ \mathbf{x} = \mathbf{y}$ by back substitution.

The time needed to factor a matrix of order N into its LU components is proportional to N^3, while the time for the forward elimination and back substitution only increases as N^2. For matrices with $N > 100$, the LU decomposition accounts for over 90 percent of the execution time. Therefore, I discuss only the factorization step.

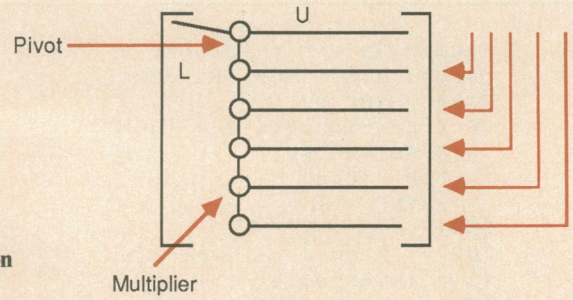


Figure 9. LU decomposition by Gaussian elimination.

```
      SUBROUTINE FACTOR(A,N,IPVT,INFO)
      DIMENSION A(N,N),IPVT(N)
      INFO = 0
      DO 20 K = 1, N-1
         IPVT(K) = ISAMAX(N-K+1,A(K,K),1) + K - 1
         CALL SSWAP (N-K+1,A(K,K),N,A(IPVT(K),K),N)
         IF ( A(K,K) .EQ. 0.0 ) THEN
            INFO = K
         ELSE
            T = -1.0/A(K,K)
            CALL SSCAL (N-K,T,A(K+1,K),1)
            DO 10 J = K+1, N
               CALL SAXPY(N-K,A(K,J),A(K+1,K),1,A(K+1,J),1)
10          CONTINUE
         ENDIF
20    CONTINUE
      IF ( A(N,N) .EQ. 0.0 ) INFO = N
      RETURN
      END
```

Figure 10. LU decomposition.

Figure 9 illustrates the decomposition process. The following procedure is used for each column in turn:

(1) Search the elements on or below the diagonal of the current column.

(2) Interchange rows to move the largest of these elements to the diagonal. This new diagonal element is called the *pivot*.

(3) Divide the elements below the diagonal by the pivot to produce a set of *multipliers*.

(4) Multiply the part of the pivot row to the right of the diagonal times each multiplier and subtract the product from the corresponding part of each row.

The code in Figure 10 embodies the LU decomposition algorithm. It is nearly identical to SGEFA from Linpack and calls several subroutines from the BLAS.[32] As used here they mean

ISAMAX: Search elements K to N of column K and return the index of the element having the largest magnitude.

SSWAP: Interchange elements K to N of rows K and IPVT(K).

SSCAL: Multiply elements K+1 to N of column K by T.

SAXPY: Multiply elements K+1 to N of column K by A(K,J) and add the result to the corresponding elements of column J.

The arguments in the calling sequence are

A: the matrix to be factored;

N: the order of the matrix;

IPVT: an array to save the order of interchanges needed for pivoting; and

INFO: a flag that indicates if a pivot is identically zero.

Shared memory. Figure 11a shows the FACTOR routine as it might be coded on a shared memory system using the fork-join approach. Assume that the arrays A and IPVT have been declared global by the calling routine. There are only two differences between the sample codes in Figures 10 and 11a. Figure 11a uses a CREATE function instead of a CALL to routine SAXPY, and a JOIN to synchronize the processes. Nothing else changes.

This program creates a task for each column to be processed. This approach is inefficient if there are fewer processors than columns due to the unavoidable overhead in starting the tasks. A better approach is shown in Figure 11b, where we divide the work up equally among the available processors. The additional argument in SAXPY tells each processor the number of columns to process.

Programming this problem SPMD on a shared memory system is slightly more complicated, as shown in Figure 12a. First, we need to assume that A and IPVT have both been declared global by the calling routine. We need one serial section to find the pivot and interchange rows. The barrier assures that none of the processors tests A(K,K) before the pivot step is complete. Next, everyone else waits at the second barrier while a processor scales the subdiagonal part of the current column. Finally, we have a PARALLEL DO. The WAIT UNTIL ensures that the next column has been computed before the search for the pivot element begins.

This example illustrates one of the disadvantages of SPMD programming. In general, synchronization must be done before most program branches. Not only does this requirement introduce extra overhead, but it is a source of potential errors. In practice, the extra overhead is small because most of the processors go immediately to the synchronization point. In addition, errors are made that involve skipping barriers in the program. These errors are easier to find than most because processors end up at a barrier waiting for a processor that never shows up. The programmer has available the exact location where the processors are waiting, which helps in finding the error.

It is also possible to provide a PRESCHEDULED DO. One way of prescheduling a loop is shown in Figure 12b. The PARALLEL DO has been replaced by a conventional DO that depends on the processor id and the number of processors. Each processor entering the loop is guaranteed to get a unique value of J, and each value of J will be taken by some processor.

Message passing. When discussing the shared memory implementations of LU decomposition, we assumed the data was already in shared memory. We cannot

```
      SUBROUTINE FACTOR(A,N,IPVT,INFO)
      DIMENSION A(N,N), IPVT(N)
      INFO = 0
      DO 20 K = 1, N-1
         IPVT(K)=ISAMAX(N-K+1,A(K,K),1)+K-1
         CALL SSWAP (N-K+1,A(K,K),N,
     *               A(IPVT(K),K),N)
         IF ( A(K,K) .EQ. 0.0 ) THEN
            INFO = K
         ELSE
            T = -1.0/A(K,K)
            CALL SSCAL (N-K,T,A(K+1,K),1)
            DO 10 J = K+1, N
              create('SAXPY',N-K,A(K,J),
     *                A(K+1,K),1,A(K+1,J),1)
10          CONTINUE
            join
         ENDIF
20    CONTINUE
      IF ( A(N,N) .EQ. 0.0 ) INFO = N
      RETURN
      END
```

(a)

```
      SUBROUTINE FACTOR(A,N,IPVT,INFO)
      DIMENSION A(N,N), IPVT(N)
      INFO = 0
      DO 20 K = 1, N-1
         IPVT(K)=ISAMAX(N-K+1,A(K,K),1)+K-1
         CALL SSWAP (N-K+1,A(K,K),N,
     *               A(IPVT(K),K),N)
         IF ( A(K,K) .EQ. 0.0 ) THEN
            INFO = K
         ELSE
            T = -1.0/A(K,K)
            CALL SSCAL (N-K,T,A(K+1,K),1)
            NC = (N-K+nprocs-1)/nprocs
            J = 1
            DO 10 I = 1, nprocs
               NC = MIN ( NC, N-K-J+1 )
               create('SAXPY',NC,N-K,A(K,J),
     *                A(K+1,K),1,A(K+1,J),1)
               J = J + NC
10          CONTINUE
            join
         ENDIF
20    CONTINUE
      IF ( A(N,N) .EQ  0.0 ) INFO = N
      RETURN
(b)   END
```

Figure 11. Fork-join LU, one processor per column (a) and fork-join LU, several columns per process (b).

ignore the step of getting the data into memory on a message passing system. Figure 13a shows one way of distributing the data to the contributing processors.

We have used the trick of having processor 0 send data to itself. Notice that the columns of the matrix are distributed to the processors much as one would deal cards in a bridge game: processor 0 gets columns $1, p+1, 2p+1, \ldots$; processor 1 gets columns $2, p+2, 2p+2, \ldots$; and so forth. If instead we gave processor 0 the first group of columns, processor 1 the next set, and so forth, we would get very poor load balancing. Referring to Figure 9 shows that after the first group of columns had been reduced, processor 0 would have no more work to do. Eventually, only one processor would be doing all the work.

In Figure 13b we see how the work is distributed. We have included additional arguments in the calling sequence: M, the number of columns to be held by each processor, and W, a work array used to hold the pivot column sent from another processor. On entering the DO 20 loop in routine FACTOR, the processor that owns the current pivot column finds the index of the maximum and scales the column. It then broadcasts the index of the pivot and the multipliers to all other processors. As soon as one of the other processors receives

```
      SUBROUTINE FACTOR(A,N,IPVT,INFO)
      DIMENSION A(N,N), IPVT(N)
      global JSYNC
      INFO = 0
      DO 20 K = 1, N-1
         serial section
            IPVT(K) = ISAMAX(N-K+1,A(K,K),1) + K - 1
            CALL SSWAP (N-K+1,A(K,K),N,A(IPVT(K),K),N)
         end serial section
         barrier
         IF ( A(K,K) .EQ. 0.0 ) THEN
            INFO = K
         ELSE
            serial section
               T = -1.0/A(K,K)
               CALL SSCAL (N-K,T,A(K+1,K),1)
               JSYNC = 0
            end serial section
            barrier
            parallel do 10 J = K+1, N
               CALL SAXPY (N-K,A(K,J),A(K+1,K),1,A(K+1,J),1)
               JSYNC = J
10          CONTINUE
            wait until ( JSYNC .GE. K+1 )
         ENDIF
20    CONTINUE
      IF ( A(N,N) .EQ. 0.0 ) INFO = N
      RETURN
(a)   END
```

```
         ---
         DO 10 J = K+1+procid, N, nprocs
            CALL SAXPY (N-K,A(K,J),A(K+1,K),1,A(K+1,J),1)
10       CONTINUE
(b)      ---
```

Figure 12. SPMD LU decomposition (a) and SPMD LU decomposition, prescheduled DO (b).

its data, it does the row interchange and forms the appropriate linear combinations for the columns it owns.

There is no need for a barrier at the end of the loop. Any processor that finishes early will quickly find itself at the RECEIVE waiting for the new pivot column to send the required data. If the owner of the pivot column finishes first, it immediately starts work on finding the pivot and producing the multipliers. The RECEIVE command guarantees the needed synchronization.

```
      PARAMETER ( N=1000, M=(N+nprocs-1)/nprocs )
      DIMENSION A(N,M), IPVT(N), COL(N), W(N), IP(N)
      IF ( procid .EQ. 0 ) THEN
         DO 10 I = 0, N-1
            READ (*) COL
            ICOL = MOD(I,nprocs)
            send ( ICOL, 'COL:N' )
10       CONTINUE
      ENDIF
      DO 20 K = 1, M
         receive ( 0, 'A(1,K):N' )
20    CONTINUE
      CALL FACTOR ( A, N, M, IP, W, IN )
      IF ( procid .EQ. 0 ) THEN
         DO 40 K = 0, nprocs-1
            receive ( K, 'INFO:1, IP:N' )
            IN = MAX (IN,INFO)
            DO 30 J = 0, N-1
               IF (MOD(J,nprocs).EQ.K) IPVT(J)=IP(J)
30          CONTINUE
40       CONTINUE
      ENDIF
      RETURN
      END
```

(a)

```
      SUBROUTINE FACTOR ( A, N, M, IP, W, IN )
      DIMENSION A(N,M), IP(N), W(N)
      IN = 0
      IC = 1
      DO 20 K = 1, N-1
         IT = MOD(K-1,nprocs)
         IF ( procid .EQ. IT ) THEN
            L = ISAMAX(N-K+1,A(K,IC),1) + K - 1
            IF ( A(L,IC) .EQ. 0.0 ) THEN
               IN = K
            ELSE
               T = -1.0/A(L,IC)
               CALL SSCAL (N-K,T,A(K+1,IC),1)
            ENDIF
            send (*, 'IN:1, L:1, A(K+1,IC):N-K' )
         ENDIF
         receive ( IT, 'IN:1, L:1, W(K+1):N-K' )
         IP(K) = L
         CALL SSWAP (M-IC+1,A(K,IC),N,A(L,IC),N)
         IF ( procid .EQ. IT ) IC = IC + 1
         IF ( IN .NE. K ) THEN
            DO 10 J = IC, M
               CALL SAXPY (N-K,A(K,J),W(K+1),1,A(K+1,J),1)
10          CONTINUE
         ENDIF
20    CONTINUE
      IT = MOD(N-1,nprocs)
      IF ( A(N,M) .EQ. 0.0 .AND. procid .EQ. IT ) IN = N
      send ( 0, 'IN:1, IP:N' )
      RETURN
      END
```

(b)

Figure 13. Message passing data distribution (a) and message passing computation (b).

Hybrid systems. Hybrid systems are programmed like shared memory systems, but have data access delays like message passing systems. These delays are usually much smaller than on message passing systems, but they can be significant. For example, the design of the IBM RP3[25] calls for local data to be delivered in two machine cyles while remote data will take 10 machine cycles to reach the functional unit.

Although the code in Figure 12a could be used on a hybrid system, there is a potential performance bug. Assume the data is distributed among the processors as done for the message passing system. A single processor will own the column containing the pivot and the multipliers. As coded, the first processor to reach the serial section will search for the pivot, and another will form the multipliers. If the processor performing these tasks is not the owner of the data, the program will run slower than it should.

The hybrid system may not need to explicitly move the data if the operating system can be directed to map the data as needed. One way to achieve this end without direct support from the operating system or the compiler is shown in Figure 14. Here we assume the data is read into the memory local to processor 0 and that A is declared global in the calling routine.

The matrix gets distributed to the other processors by being copied into a local variable, AL. Two work arrays, one local and one global, are used to move the multipliers between the local memories of the processors. First, the owner of the current column copies the multipliers from its private memory, AL, into the global work array W. After the barrier, each processor copies the global work array into its local work array, WL. This approach saves time since the multipliers are used many times in the DO 20 loop. At the end of the factorization, the entire matrix gets reassembled in the DO 40 loop. An undesirable side effect of this approach is that memory is wasted because two copies of the array are needed.

Clearly, this code for the hybrid system is more complex and harder to debug than either the shared memory or message passing versions. Note, however, that this complication is needed only to improve performance. The program in Figure 12a will run correctly on a hybrid system. If the ratio of the access times for local and

remote data is near unity, there will be no need to worry about the location of the data.

Published examples

Solving systems of linear equations is an important part of many applications and often accounts for a large part of the computer time. Because the algorithm can be written in a very compact program, it is commonly used as an example.[33,34]

This section presents four versions of this algorithm that illustrate some of the parallel processing constructs used. These programs are based on subroutine SGEFA from Linpack.[32] Liberties were taken with the published codes to make them more readable and more like my examples. The reader is referred to the cited publications for the full programs.

VM/EPEX. I wrote the first example, Figure 15, to show SPMD programming using the VM/EPEX software available for use within IBM for experimental purposes.[6] VM/EPEX works with the VM/CMS operating system using unprotected shared segments to provide a shared memory area. Each processor is a distinct virtual machine, and all synchronization is handled by semaphores in shared memory.

Although the hardware can have one, two, or four processors, the software makes it possible to simulate any number of processors. VM/EPEX comes with a preprocessor that scans a source file for constructs beginning with @. These constructs get converted into in-line code and calls to subroutines that provide the required functions.

The VM/EPEX code is very similar to the code in Figure 12a. The @SHARED in the example is translated into COMMON and marked for loading into the shared memory. The construct is equivalent to declaring the variables A, IPVT, and N to be GLOBAL. @SERIAL BEGIN PROCESS = 1 defines a serial section to be executed by the processor with PROCID = 1. It is terminated with @SERIAL END WAIT, which is equivalent to the END SERIAL SECTION followed by a barrier. The @DO is a self-scheduled, PARALLEL DO terminated by the @ENDDO WAIT. Again, the WAIT is equivalent to putting a barrier following the loop. The option CHUNK is used to reduce the overhead in assigning loop indi-

```
      SUBROUTINE FACTOR(A,AL,N,M,IP,W,WL,IN)
      DIMENSION A(N,N),AL(N,M),IP(N),W(N),WL(N)
      global L, PIVOT, W
      INFO = 0
      IC = 0
      DO 10 K = procid, N, nprocs
         IC = IC + 1
         CALL SCOPY (N,A(1,K),1,AL(1,IC),1)
10    CONTINUE
      IC = 1
      DO 30 K = 1, N-1
         IT = MOD(K-1,nprocs)
         IF ( procid .EQ. IT ) THEN
            IP(K) = ISAMAX (N-K+1,AL(K,IC),N) + K - 1
            L = IP(K)
            PIVOT = AL(L,IC)
         ENDIF
         barrier
         CALL SSWAP ( M-IC+1, AL(K,IC), N, AL(L,IC), N )
         IF ( PIVOT .EQ. 0.0 ) THEN
            IN = K
         ELSE
            IF ( procid .EQ. IT ) THEN
               T = -1.0/PIVOT
               CALL SSCAL (N-K,T,AL(K,IC),1)
               CALL SCOPY (N-K,AL(K+1,IC),1,W(K+1),1)
               IC = IC + 1
            ENDIF
            barrier
            CALL SCOPY (N-K,W(K+1),1,WL(K+1),1)
            DO 20 J = IC, M
               CALL SAXPY(N-K,AL(K,J),WL(K+1),1,AL(K+1,J),1)
20          CONTINUE
         ENDIF
30    CONTINUE
      IC = 0
      DO 40 K = procid, N, nprocs
         IC = IC + 1
         CALL SCOPY (N,AL(1,IC),1,A(1,K),1)
40    CONTINUE
      IF ( A(N,N) .EQ. 0.0 ) IN = N
      RETURN
      END
```

Figure 14. SPMD LU decomposition of a hybrid.

```
      SUBROUTINE DGEFA(LDA,INFO)
      @SHARED/ARRAY/A(100,100),IPVT(100),N
      INFO = 0
      DO 60 K = 1, N-1
         @SERIAL BEGIN PROCESS = 1
            L = ISAMAX(N-K+1,A(K,K),1)+K-1
            IPVT(K) = L
            IF (A(L,K) .EQ. 0.0D0) THEN
               INFO = K
            ELSE
               T = -1.0D0/A(L,K)
               CALL SSCAL(N-K+1,T,A(K+1,K),1)
            ENDIF
         @SERIAL END WAIT
         L = IPVT(K)
         IF ( A(L,K) .NE. 0.D0 ) THEN
            CALL SSWAP ( N-K+1, A(L,K), N, A(K,K), N )
            @DO 30 J = K+1, N, CHUNK = 10
               CALL SAXPY(N-K,A(K,J),A(K+1,K),1,A(K+1,J),1)
30          @ENDDO WAIT
         ELSE
            INFO = K
         ENDIF
60    CONTINUE
      IPVT(N) = N
      IF (A(N,N) .EQ. 0.0D0) INFO = N
      END
```

Figure 15. VM/EPEX code.

```
      forcesub SGEFA(LDA,INFO) of NPROC on NPEM ident ME
         global A(1000,1000), IPVT(1000), N, INFO
      end header
      INFO = 0
      DO 50 K = 1, N-1
         barrier
            L = ISAMAX ( N-K+1, A(K,K), 1 ) + K - 1
         end barrier
         IF ( A(L,K) .EQ. 0.0 ) THEN
            INFO = K
         ELSE
            barrier
               IPVT(K) = L
               CALL SSWAP (N-K+1,A(K,K),N,A(IPVT(K),K),N)
               T = -1.0/A(K,K)
               CALL SSCAL (N-K,T,A(K+1,K),1)
            end barrier
            selfsched DO 40 J = K+1, N
               CALL SAXPY (N-K,A(K,J),A(K+1,K),1,A(K+1,J),1)
40          end selfsched DO
         ENDIF
50    CONTINUE
      END
```

Figure 16. Force code.

```
      REAL S
      do parallel ( I = 1, N )
         begin
            REAL T
            ...
         end
      end parallel
```

Figure 17. BEGIN-END code.

```
      PARAMETER (ND=100)
      DIMENSION A(ND,ND), TEMP(ND)
      READ (5,*) N
      CALL SETRNG(A,N,N)
      CALL SETRNG(TEMP,N)
      DO 20 IC = 1, N-1
         CMAX = MAX(A(IC:N,IC))
         IMAX    = LOCMAX(A(IC:N,IC))
         TEMP = A(IMAX,:)
         A(IMAX,:) = A(IC,:)
         A(IC,:) = TEMP
         do parallel (JR=IC+1,N)
            begin
               A(IC,JR) = A(IC,JR)/CMAX
               do parallel (K = IC+1:N)
                  A(JR,K) = A(JR,K) - A(IC,K)*A(JR,IC)
               end parallel
            end
         end parallel
   20 CONTINUE
      END
```

Figure 18. Protran code.

ces to processors. Here it tells the system to give each processor 10 values of the loop index each time they get work. Because of the synchronization needed to get unique loop indices, the reduction in elapsed time is significant.

The Force. This example, Figure 16, shows the SPMD programming style using the Force.[2] The basic concept of the Force is very similar to that of VM/EPEX although they were developed independently. I have modified the published code to make it closer to the examples shown earlier. As published, the program does both the search for the pivot element and the scaling of the pivot column in parallel. Since these steps represent only a small part of the work if the matrix is large, I have chosen to do them on a single processor.

A Force subroutine has a header that indicates how many processes, NPROC, are to be run on how many processors, NPEM, and the local variable used to store the process id, ME. The header also contains the declaration of global data. Like VM/EPEX, COMMON is used to implement data sharing.

The Force uses a generalized concept of a barrier. All processes stop at the barrier until the last one has arrived. That process then executes the code up to the END BARRIER. Once the first process has reached this point, all processes continue executing at the line following the END BARRIER. Thus, this construct is equivalent to our serial section bracketed by our BARRIER. If there is no code between BARRIER and END BARRIER, then this construct is equivalent to our BARRIER. The SELFSCHED DO is equivalent to our PARALLEL DO.

Protran. A collection of problems has been put together to test language extensions to support vector and parallel processors.[35] One of these extensions to Fortran was developed by IMSL, Inc., and is called Protran.[36] In addition to vector and matrix operations, it has a number of problem solving statements. Extensions include some that make the language look more like the proposed Fortran 8X array extensions[37] and others that add parallel processing constructs.

The parallel processing constructs added to Protran are DO PARALLEL, CRITICAL, and BEGIN-END. The first two extensions do not need further explanation. Since this language is intended for use on shared memory systems, the BEGIN-

END is the only way of providing private data among processors operating within a DO PARALLEL. Although the sample code presented does not need any local variables, if needed they can be declared within the scope of the BEGIN-END. In the example of Figure 17, each processor executing the DO PARALLEL has the same value of S and a distinct value of T. In addition, T does not exist outside the BEGIN-END.

In the Protran version of the factorization code shown in Figure 18, routine SETRNG is used to set up a dope vector containing the lengths of the arrays being used. These lengths need not be the same as the dimensions of the arrays. This code also shows several of the array extensions proposed for Fortran 8X.

Note the use of a nested DO PARALLEL. It is left to the compiler to decide what work can be done in parallel. For example, the inner DO PARALLEL cannot be started until the column scaling is complete. Depending on the hardware, the scaling could be done on one processor or distributed over processors. In this case, we let the compiler contain the knowledge of the hardware instead of the programmer. Such a procedure puts more pressure on compiler writers, but leads to more portable code.

```
      SUBROUTINE PGEFA(A,LDA,N,M,IP,CID,ID,IPVT,BUF)
      DIMENSION A(LDA,M), BUF(N), IPVT(M)
      L = 1
      DO 20 K = 1, N
         IR = MOD(K-1,IP)
         IF ( IR .EQ. ID ) THEN
            KP = ISAMAX ( N-K+1, A(K,L), 1 ) + K + 1
            IF ( A(KP,L) .EQ. 0.D0 ) KP = 0
            IPVT(L) = KP
            IF( KP .NE. 0 ) THEN
              T = -1.0/A(K,L)
              CALL SSCAL ( N-K, T, A(K+1,L) , 1 )
            ENDIF
            CALL SCOPY(N-K,A(K+1,L),1,BUF,1)
            BUF(N-K+1) = KP
            L = L + 1
         ENDIF
         CALL GSEND(ID,IR,K,IP,CID,BUF,N-K+1)
         KP = BUF(N-K+1)
         IF ( KP .NE. 0 ) THEN
            CALL SSWAP ( M-L+1, A(IPVT(K),L),N,A(K,L),N )
            DO 10 J = L, M
               CALL SAXPY(N-K,A(K,L),BUF,1,A(K+1,J),1)
10          CONTINUE
         ENDIF
20    CONTINUE
      END
```

Figure 19. iPSC code.

Hypercube. The code presented in Figure 19 is part of the Linpack library being written for the iPSC,[21] a hypercube message passing system marketed by Intel Corporation. It is quite similar to the program shown in Figure 13b as subroutine FACTOR. Although the distribution of the data is not shown, a scheme similar to that shown in Figure 13a could be used.

Each node of the hypercube runs the same subroutine, so this code is an example of SPMD programming. In this system, both the number of processors, IP, and the process id, ID, are passed through the argument list. The GSEND routine is similar to the SEND(*,... construct used in Figure 13b. GSEND also contains code to receive the data. Thus, all data routing and synchronization is handled in this routine. If the program were moved to a different message passing system, all that need be changed is GSEND.

The point of parallel processing is to reduce the elapsed time to complete the job. If the program that sums 1,000,000 numbers were run on a 1-Mflop uniprocessor, it would finish in about one second. The time the program takes on a parallel processor will depend on the coding style, the architecture of the machine, and the hardware implementation. However, run on 10 processors each capable of 1 Mflop, the job will certainly take longer than 0.1 seconds to complete. The job of the system designers, compiler and library writers, and application programmers is to get the actual time for the job as close as possible to the ideal.

I have identified three classes of parallel architectures: shared memory, message passing, and hybrid. I have also illustrated two programming styles: fork-join and SPMD. Each programming style can be used on each of the parallel architectures depending on the tools provided to the applications programmer.

Algorithms are easy to design for shared memory systems. One simply puts the data in memory as if running on a uniprocessor. On the other hand, programs are hard to debug. An error usually involves picking up wrong data from a global variable. The processor then continues computing, with the bad data producing an erroneous final result. There is no indication of when the error occurred.

Message passing systems are different. Algorithm design is hard because the data must be distributed so that communications traffic is minimized. Debugging is easier than in shared memory systems because errors normally cause the system to stop at the point of the error. Thus, the programmer knows the complete state of the machine at the point where the error occurred. This is not to say that debugging is easy. There are many times where it is very hard to track down the cause of the error, but at least the programmer knows where to start looking.

Hybrid systems are the worst of both worlds. Errors are hard to find because they are the same ones made on shared memory systems. In addition, care is needed in organizing the data in order to reduce the amount of data to be moved. All is not lost, however, since hybrid systems may be easier to build than either shared memory or message passing systems.

From these comments we see that data organization is the key to parallel algorithms even on shared memory systems. Unfortunately, Fortran programmers have such simple data structures available to them, just scalars and arrays, that we tend to con-

centrate on program flow. It will take some retraining to get Fortran programmers to plan their data first and their program flow later.

The importance of data management is also a problem for people writing automatic parallelization compilers. To date, our compiler technology has been directed toward optimizing control flow. Such features as common expression elimination, code movement, and dependence analysis for vectorization have been used for many years. Even today, when hierarchical memories make program performance a function of data organization, no compiler in existence changes the data addresses specified by the programmer to improve performance. If such compilers are to be successful, particularly on message passing and hybrid systems, a new kind of analysis will have to be developed. This analysis will have to match the data structures to the executable code in order to minimize memory traffic.

A more fundamental problem on shared memory systems arises when the parallelism is nested. Most parallel processors allow only a single level of parallelism, which greatly simplifies the data sharing specification. In these systems, a master process is allowed to spawn subprocesses, but the subprocesses may not themselves spawn processes. Alternatively, all processors run the same code as their peers. In both cases, data is either known to all processes or is private.

The situation is more complicated when a subprocess is allowed to spawn subprocesses of its own. Consider a parallel job currently running two subprocesses that each call a library routine to work on some private data. If the library routine spawns subprocesses of its own, a simple global/local dichotomy for the data will not suffice. Each of the library routine invocations must share a different set of data among its subprocesses. Clearly, some form of scoping of the global data is required. The data scoping rules in such block-structured languages as Algol and PL/I provide a useful model, but to date no systems have addressed this issue.

As I stated in the introduction, my goal was to present the state of the art of parallel programming. I believe I have shown what a sorry state that art is in. We are just beginning to define the appropriate set of language extensions. Much more work is needed in compiler-assisted dependence analysis and in developing debugging tools.□

Acknowledgments

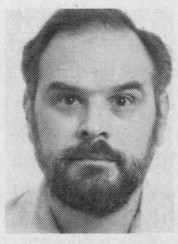
I would like to thank an anonymous referee for suggesting important improvements to the organization of this paper.

References

1. J.J. Dongarra and I.S. Duff, "Advanced Computer Architectures," Mathematics and Computer Science Div. Report TM-57, Argonne National Laboratory, Argonne, Ill., 1985.
2. H.F. Jordan, "Structuring Parallel Algorithms in an MIMD, Shared Memory Environment," Report Number CSDG 84-2, Dept. of Electrical and Computer Engineering, University of Colorado, 1984.
3. P. Mehrotra and J. Van Rosendale, "The BLAZE Language: A Parallel Language for Scientific Programming," ICASE Report Number 85-29, NASA Langley Research Center, Langley, Va., 1985.
4. S. Shin et al., "Parallel Computation on the Loosely Coupled Array of Processors: A Guide to the Preprocessor," IBM Report # KGN-42, Kingston, N.Y., 1985.
5. T.W. Pratt, "PISCES: An Environment for Parallel Scientific Computation," ICASE Report Number 85-12, NASA Langley Research Center, Langley, Va., 1985.
6. F. Darema-Rogers et al., "An Environment for Parallel Execution," IBM Research Report #11225, Yorktown Heights, N.Y., 1985.
7. E.L. Lusk and R.A. Overbeek, "Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors," Report Number ANL-83-97, Argonne National Laboratory, Argonne, Ill., 1983.
8. D. Kuck, "High Speed Multiprocessing and Compilation Techniques," *IEEE Trans. on Computers*, C-29, Piscataway, N.J., 1980, pp. 763-776.
9. J.R. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," *Proc. IBM Conf. Parallel Computers in Scientific Computations*, Rome, Italy, 1982. Also published in *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, Md., 1984.
10. R. Perron and C. Mundie, "The Architecture of the Alliant FX/8 Computer," *Digest of Papers, Compcon, Spring 86*, A.G. Bell, ed., IEEE Computer Society Press, Washington, D.C., 1986.
11. G.H. Barnes et al., "The Illiac IV Computer," C-17, *IEEE Trans. on Computers*, Piscataway, N.J., 1968, pp. 746-757.
12. R.W. Hockney and C.R. Jesshope, *Parallel Computers*, Adam Hilger, Ltd., Bristol, 1981.
13. K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. on Computers*, C-29, Piscataway, N.J., 1980, pp. 836-340. Also published in *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, Md., 1984.
14. W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.
15. S.Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proc. IEEE*, Vol. 72, Piscataway, N.J., 1984, pp. 867-884. Also published in *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, Md., 1984.
16. J.J. Dongarra, B.T. Smith, and D.C. Sorensen, "Algorithm Design for Different Computer Architectures," in "New Directions for Advanced Computer Architectures," Report Number ANL/MCS-TM-32, Argonne National Laboratory, Argonne, Ill., 1984.
17. J.L. Larson, "An Introduction to Multitasking on the Cray X-MP-2 Multiprocessor," *Computer*, July 19□84, pp. 62-69.
18. IBM Corp., "IBM 3090 System Summary—Engineering/Scientific," announcement letter 185-120, 1985.
19. A. Gottlieb et al., "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. on Computers*, C-32, Piscataway, N.J., 1983, pp. 175-189. Also published in *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, Md., 1984.
20. T. Feng, "A Survey of Interconnection Networks," *Computer*, Dec. 1981, pp. 12-27. Aiso published in *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, Md., 1984.
21. J. Rattner, "Concurrent Processing: A New Direction in Scientific Computing," *AFIPS Conference Proc.*, 54, AFIPS Press, Reston, Va., 1985, pp. 159-166.
22. C.L. Seitz, "The Cosmic Cube," *Comm. ACM*, 28, New York, N.Y., 1985, pp. 22-33.
23. J.F. Palmer, "A VLSI Parallel Computer," *Digest of Papers, Compcon, Spring 86*, A.G. Bell, ed., IEEE Computer Society Press, Washington, D.C., 1986.
24. E. Clementi and D. Logan, "Parallel Processing with the Loosely Coupled Array of Processors System," IBM Report # KGN-43, Kingston, N.Y., 1985.
25. G.F. Pfister et al., "The IBM Research Parallel Processor Prototype (RP3)," *Int'l Conf. Parallel Processing*, IEEE Computer Society Press, Washington, D.C., 1985.
26. W. Crowther et al., "Performance Measurements on a 128-node Butterfly Parallel Processor," *Int'l Conf. Parallel Processing Proc.*, IEEE Computer Society Press, Washington, D.C., 1985, pp. 531-535.
27. D. Gajski et al., "CEDAR," *Digest of Papers, Compcon, Spring 86*, A.G. Bell, ed., IEEE Computer Society Press, Washington, D.C., 1986.
28. R.W. Hockney, "MIMD Computing in the USA—1984," *Parallel Computing*, Vol. 2, 1985, pp. 119-136.
29. H.F. Jordan, "Parallel Computation with the Force," Report Number 85-45, ICASE, NASA Langley, Hampton, Va., 1985.
30. M. Booth and K. Misegades, "Microtasking: A New Way to Harness Multiproces-

sors," *Cray Channels*, Summer 1986, pp. 24-27.

31. R. Rettberg and R. Thomas, "Contention Is No Obstacle to Shared-Memory Multiprocessing," *Comm. ACM*, Vol. 29, New York, N.Y., 1986, pp. 1202-1212.
32. J.J. Dongarra et al., *Linpack Users' Guide*, Society of Industrial and Applied Mathematics, Philadelphia, 1979.
33. J.J. Dongarra, F.G. Gustavson, and A.H. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Review*, Vol. 26, 1984, pp. 91-112.
34. J.J. Dongarra, "Performance of Various Computers using Standard Linear Equations Software in a FORTRAN Environment," Mathematics and Computer Science Div. Report TM-23, Argonne National Laboratory, Argonne, Ill., 1986.
35. J.R. Rice, "Problems to Test Parallel and Vector Languages," Report Number CSD-TR 516, Dept. of Computer Sciences, Purdue University, W. Lafayette, Ind., 1985.
36. J.R. Rice, *Numerical Methods, Software and Analysis*, McGraw-Hill, New York, 1983.
37. G. Paul, "Vectran and the Proposed Vector/Array Extensions to ANSI Fortran for Scientific and Engineering Computation," *Proc. IBM Conference on Parallel Computers in Scientific Computations*, Rome, Italy, 1982. Also published in *Supercomputers: Design and Applications*, K. Hwang, ed., IEEE Computer Society Press, Silver Spring, Md., 1984.



Alan Karp is a staff member at IBM's Palo Alto Scientific Center. He has worked on problems of radiative transfer in moving stellar matter and in planetary atmospheres, hydrodynamics problems in pulsating stars and in enhanced oil recovery, and numerical methods for parallel processors. He is currently studying the interface between programmers and parallel processors, with special attention to debugging parallel algorithms.

Karp received his PhD in astronomy from the University of Maryland in 1974. He has served on the editorial boards of the *Journal of Quantitative Spectroscopy and Radiative Transfer* and the *Journal of Transport Theory and Statistical Physics*.

Readers may write to Karp at the IBM Scientific Center, 1530 Page Mill Road, Palo Alto, CA 94304.