An Introduction to Parallel Computation

Abhiram Ranade

The most powerful computers in existence today are parallel computers. How are these parallel computers built? How are they programmed? This article provides an introduction.

Abhiram Ranade is an Associate Professor of Computer Science and Engineering at the Indian Institute of Technology, Mumbai. His fields of research include parallel computation, algorithm design and computer architecture.

A parallel computer is a network of processors built for the purpose of solving *cooperatively* large computational problems as fast as possible. Several such computers have been built recently, and have been used to solve problems much faster than a single processor. A spectacular example is a computation recently carried out by a research group at the T J Watson Research Center in Yorktown Heights, USA. They wanted to solve a problem in Quantum Chromo Dynamics which would have taken hundreds of years on a single processor, clearly a project that would never have been attempted. By using a parallel computer having 448 processors, they were able to solve the problem in a continuous run lasting 2 years. Of course, high speed computation is necessary also in other more mundane areas. For instance, parallel computers are routinely used for predicting the weather. But for it, forecasting would perhaps never be on time! Parallel computers have also been very useful for solving problems in engineering and biological sciences as well as in business data processing.

This article provides a brief introduction to parallel computing. How do you build parallel computers? How are they programmed? How can you distribute the computations required to solve a program among several processors so as to reduce the total time? These are some of the questions considered in this article.

Parallel Computer Organization

A parallel computer is built by suitably connecting together processors, memory, and *switches*. A switch is simply a hardware

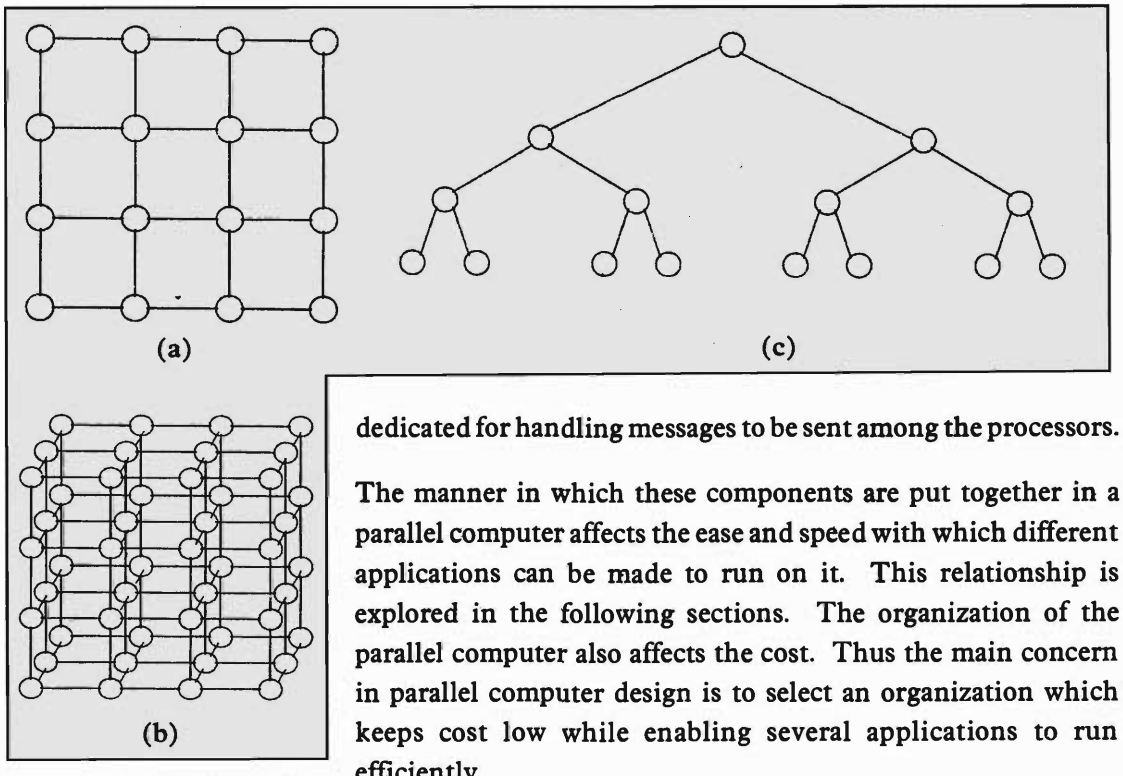

Figure 1. Parallel Computer Organizations.

dedicated for handling messages to be sent among the processors.

The manner in which these components are put together in a parallel computer affects the ease and speed with which different applications can be made to run on it. This relationship is explored in the following sections. The organization of the parallel computer also affects the cost. Thus the main concern in parallel computer design is to select an organization which keeps cost low while enabling several applications to run efficiently.

Figure 1 schematically shows the organization of several parallel computers that have been built over the years. Circles in *Figure 1* represent a *node* which might consist of a processor, some memory and a switch. The memory in a node is commonly referred to as the local memory of the processor in the node. Nodes are connected together by communication links, represented by lines in the figure. One or several nodes in the parallel computer are typically connected to a *host*, which is a standard sequential computer, e.g. a PC. Users control the parallel computer through the host.

Figure 1a shows an organization called a two dimensional array, i.e. the nodes can be thought of as placed at integer cartesian coordinates in the plane, with neighbouring nodes connected together by a communication link. This is a very popular organization, because of its simplicity, and as we will see, because of the ease of implementing certain algorithms on it. It has been

Readers might wonder why this is called a tree; hint: turn it upside down! Continuing the analogy, the lone processor at the top is called the *root*, and the processors in the bottom most level (8 in this case) are called *leaves*! The other processors are called the *internal nodes*.

used in several parallel computers, including 'The Paragon', parallel computer built by Intel Corporation. Higher dimensional arrays are also used. For example, the Cray T3E parallel computer is interconnected as a 3 dimensional array of processors *Figure 1b*.

Figure 1c shows the binary tree [1] organization. This has been used in several computers e.g. the Database Machine parallel computer built by Teradata Corporation, or the Columbia University DADO experimental parallel computer.

Besides two and three dimensional arrays and binary trees, several other organizations have been used for interconnecting parallel computers. Comprehensive descriptions of these can be found in the Suggested Reading given at the end.

Basic Programming Model

A parallel computer can be programmed by providing a program for each processor in it. In most common parallel computer organizations, a processor can only access its local memory. The program provided to each processor may perform operations on data stored in its local memory, much as in a conventional single processor computer. But in addition, processors in a parallel computer can also send and receive messages from processors to which they are connected by communication links. So, for example, each processor in *Figure 1a* can send messages to upto 4 processors, the ones in *Figure 1b* to upto 6 processors, and in *Figure 1c* to upto 3 processors.

For estimating the execution time of the programs, it is customary to assume that the programs can perform elementary computational operations (e.g. addition or multiplication) on data stored in local memory in a single step. It is also customary to assume that it takes a single time step to send one word of data to a neighbouring processor.[2]

In order to solve an application problem on a parallel computer we must divide the computation required among the available

[2] If a processor needs to send a message to a processor that is not directly connected to it, the message must be explicitly forwarded through the intermediate processors. In some parallel computers the switches are intelligent and can themselves forward the messages without involving the main processors. In such computers, more complex models are needed to estimate the time taken for communication.

processors and provide the programs that run on the processors. How should one divide the computations and manage the communication among the processors? This is explored in the following section. How the programs for the individual processors are expressed by the user is a question concerning *parallel programming languages* which is also discussed later.

Parallel Algorithm Design

Parallel algorithm design is a vast field. Over the years, parallel-algorithms have been designed for almost every conceivable computational problem. Some of these algorithms are simple, some laborious, some very clever. I will provide a very brief introduction to the field using two examples: matrix multiplication and a problem called prefix computation. These two examples will in no way cover the variety of techniques used for parallel algorithm design, but I hope that they will illustrate some of the basic issues.

The first step in designing a parallel algorithm is to understand how the problem might have been solved on a conventional single processor computer. Often this might reveal that certain operations could potentially be performed in parallel on different processors. The next step is to decide which processor will perform which operations, where the input data will be read and how the data structures of the program will be stored among the different processors. For high performance, it is desirable that no processor should be assigned too much work else it will lag behind the others and delay the completion. Also the processors should not waste time waiting for data to arrive from other processors: whatever data is needed by them should ideally be available in their local memories, or arrive from near-by processors.

Presented below is the algorithm for matrix multiplication running on a 2 dimensional array, and that for prefix computation on a tree of processors. This does not mean that the

```
procedure matmult (A, B, C, n)
dimension A (n, n), B (n, n), C (n, n)
do i=1, n
    do j=1, n
            C(i, j) = 0
            do k-1, n
                    C(i, j) = C(i, j) +A (i,k) * B (k, j)
            enddo
    enddo
enddo
end
```

Figure 2. Sequential matrix multiplication.

problems cannot be solved on other organizations, just that these respective organizations are the most convenient.

Matrix Multiplication

I shall consider square matrices for simplicity. The sequential program for this problem is shown in *Figure 2*.

The program is very simple: it is based directly on the definition of matrix multiplication ($c_{ij} = \Sigma_k a_{ik} b_{kj}$). Most sophisticated algorithms are known for matrix multiplication, but this is the most commonly used algorithm. It is clear from the code that all the n^2 elements of C can be calculated in parallel! This idea is used in the parallel program given in *Figure 3*.

An $n \times n$ two dimensional array of processors is used, with processors numbered as shown. All processors execute the program shown in the figure. Notice that each basic iteration takes 3 steps (after data is ready on the communication links), I will call this a macrostep.

Matrix B is fed from the top, with processor $1i$ receiving column i of the matrix, one element per macrostep, starting at macrostep i. Processor 11 thus starts receiving its column in macrostep 1, processor 12 in macrostep 2, and so on. This is suggested in *Figure 3* by staggering the columns of B. Likewise, matrix A

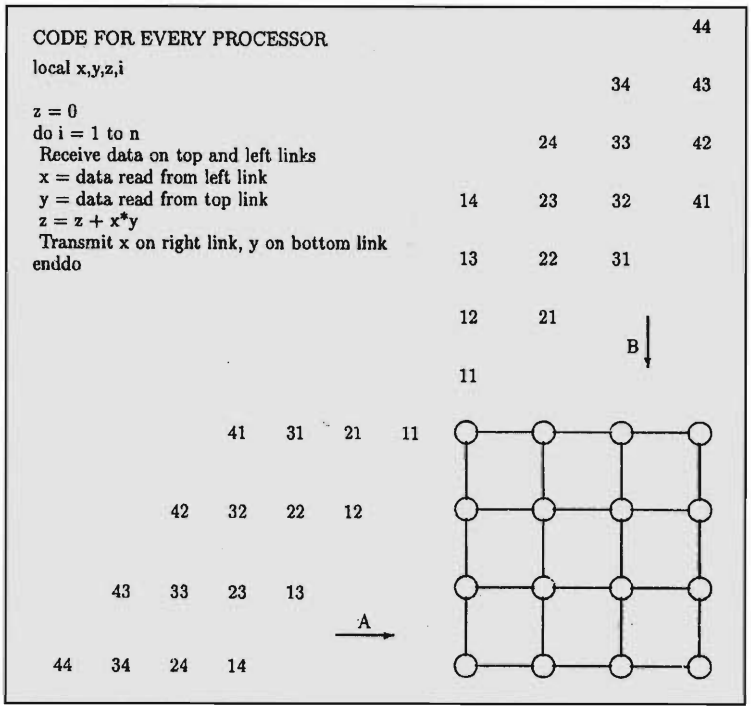

Figure 3. Parallel matrix multiplication.

is fed from the left, with processor $j1$ receiving row j, one element per macrostep, starting at macrostep j. The code for each processor is exceedingly simple. Each processor maintains a local variable z, which it initializes to zero. Then the following operations are repeated n times. Each processor waits until data is available on the left and top links. The numbers read on the two links are multiplied and the product added to the local variable z. Finally, the data read on the left link is sent out on the right link, and the data on the top sent to the bottom link.

I will show that processor ij in the array will compute element $C(i, j)$ in its local variable z. To see this notice that every element $A(i, k)$ gets sent to every processor in row i of the array. In fact, observe that processor ij receives elements $A(i, k)$ on its left link at macrostep $i+j+k-1$. Similarly processor ij also receives $B(k, j)$ on its top link at the same step! Processor ij multiplies these elements, and notice that the resulting product $A(i, k) * B(k, j)$ is accumulated in its local variable z. Thus it may be seen that by macrostep $i+j+n-1$, processor ij has completely

calculated $C(i, j)$. Thus all processors finish computation by macrostep $3n - 1$ (the last processor to finish is nn). At the end every processor ij holds element $C(i, j)$.

The total time taken is $3n - 1$ macrosteps (or $9n - 3$ steps), which is substantially smaller than the approximately cn^3 steps required on a sequential computer (c is a constant that will depend upon machine characteristics). The parallel algorithm is thus faster by a factor proportional to n^2 than the sequential version. Notice that since we only have n^2 processors, the best we can expect is a speed up of n^2 over the sequential. Thus the algorithm has achieved the best time to within constant factors.

At this point, quite possibly the readers are saying to themselves, "This is all very fine, but what if the input data was stored in the processors themselves, e.g. could we design a matrix multiplication algorithm if $A(i, j)$ and $B(i, j)$ were stored in processor ij initially and not read from the outside?" It turns out that it is possible to design fast algorithms for this initial data distribution (and several other natural distributions) but the algorithm gets a bit more complicated.

Prefix Computation

The input to the prefix problem is an n element vector x. The output is an n element vector y where we require that $y(i)=x(1)+x(2)+\ldots+x(i)$. This problem is named prefix computation because we compute all prefixes of the expression $x(1)+x(2)+x(3)+\ldots+x(n)$. It turns out that prefix computation problems arise in the design of parallel algorithms for sorting, pattern matching and others, and also in the design of arithmetic and logic units (ALUs).

On a uniprocessor, the prefix computation problem can be solved very easily, in linear time, using the code fragment shown in *Figure 4*. Unfortunately, this procedure is inherently sequential. The value $y(i)$ computed in iteration i depends upon the

Figure 4. Sequential prefix computation.

```
procedure prefix (x, y, n)
dimension x (n), y (n)
y (1) = x (1)
doi =2, n
y (i) = y (i – 1) + x (i)
enddo
end
```

value $y(i-1)$ computed in the $i-1$th iteration. Thus, if we are to base any algorithm on this procedure, we cannot compute elements of y in parallel. This is a case where we need to think afresh.

Thinking afresh helps. Surprisingly enough, we can develop a very fast parallel algorithm to compute prefixes. This algorithm called the *parallel prefix algorithm*, is presented in *Figure 5*. It runs on a complete binary tree of processors with n leaves, i.e. $2n - 1$ processors overall.

The code to be executed by the processors is shown in *Figure 5*. The code is different for the leaves, internal nodes and the root. As will be seen, initially all processors except the leaf processors execute receive statements which implicitly cause them to wait for data to arrive from their children. The leaf processors read in data, with the value of $x(i)$ fed to leaf i from the left. The leaf processors then send this value to their parents. After this the leaves execute a receive statement, which implicitly causes them to wait until data becomes available from their parents.

For each internal node, the data eventually arrives from both children. These values are added up and then sent to its own parent. After this the processors wait for data to be available from their parent.

The root waits for data to arrive from its children, and after this happens, sends the value 0 to the left child, and the value received from its left child to its right child. It does not use the value sent by the right child.

```
procedure for leaf processors
local val, pval
read val
send val to parent
Receive data from parent
pval = data received from parent
write val+pval
end
```

```
procedure for internal nodes
local lval, rval, pval
Receive data from left and right children
lval = data received from left child
rval = data received from right child
send lval+rval to parent
Receive data from parent
pval = data received from parent
send pval to left child
send pval+lval to right child
end
```

```
procedure for root
local lval, rval
Receive data from left and right children
lval = data received from left child
rval = data received from right child
send 0 to left child
send lval to right child
end
```

Figure 5. Parallel Prefix Computation.

The data sent by the root enables its children to proceed further. These children in turn execute the subsequent steps of their code and send data to their own children and so on. Eventually, the leaves also receive data from their parents, the values received are added to the values read earlier and the output.

Effectively, the algorithm runs in two phases: in the 'up' phases all processors except the root send values towards their parent. In the 'down' phase all processors except the root receive values

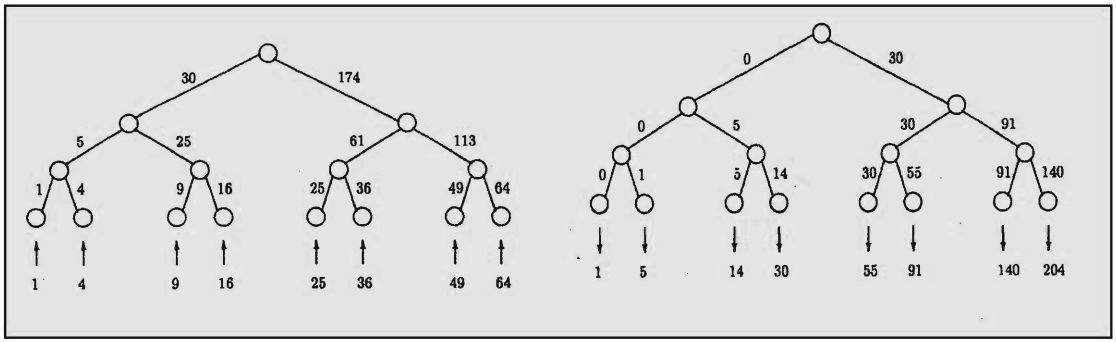

Figure 6. Execution example for parallel prefix.

from their parent. *Figure 6* shows an example of the algorithm in execution. As input we have used $x(i)=i^2$. The top picture shows values read at the leaves and also the values communicated to the parents by each processor. The bottom picture shows the values sent to the children, and the values output. *Figure 6* verifies the correctness of the algorithm for an example, but it is not hard to do so in general. The two main insights are: each internal node sends to its parent the sum of the values read in its descendant leaves, and each non root node receives from its parent the sum of the values read to the left of its descendant leaves. Both these observations can be proved by induction.

In the 'up' phase, the leaves are active only in the first step, their parents only at step 2, their parents in turn only at step 3, and so on. The root receives values from its children in step $\log n$, and sends data back to them at step $(\log n)+1$. This effectively initiates the 'down' phase. The down-phase mirrors the up-phase in that the activity is triggered by the root, and the leaves are the last to get activated. Thus the entire algorithm finishes in $2 \log n$ steps. Compare this to the sequential algorithm which took about n steps to complete. $2\log n$ is much smaller![3]

In conclusion even though at first glance it seemed that prefix computation was inherently sequential, it was possible to invent a very fast parallel algorithm.

[3] We can devise an algorithm that runs in essentially the same time using only $p=n/\log n$ processors. This algorithm is more complex than the one described, but it is optimal in the sense that it is faster than the sequential version by about a factor p.

Parallel Programming Languages

Most parallel computers support libraries with procedures that perform communication operations. With such libraries, it is possible to program the parallel computer by writing programs very similar to the ones described in the previous section. Notice that this style of programming is fairly low level – the programmer precisely needs to know details of the machine like the number of processors and the interconnection pattern. A program written for one parallel computer cannot be made to run on others, even another parallel computer with just a different number of processors.

Over the last few years, programming languages have been developed which (attempt to) allow users to write *portable* parallel programs. Some of these languages require the user only to identify what operations can be performed in parallel, without identifying explicitly which processor performs which operations and when. The hope is that the compiler will work like an expert system (it will have to know what we know about parallel algorithm design techniques) and translate the high level user code into programs for individual processors. Obviously, devising such compilers is a challenging task.

As an example, *Figure 7* shows a program for computing dot products in the language HPF (High Performance FORTRAN). HPF allows parallelism to be expressed through operations on arrays. For example, the first statement "$c = a * b$" concisely specifies a parallel operation in which the n elements of the arrays a and b are multiplied together and assigned to corresponding elements of the array c. Other parallel operations on arrays (e.g. addition, and even more complex operations) are also predefined in the language. It is not always necessary to operate on corresponding elements; for example "a [1:5] $*$ b [6:10]" would cause a pairwise multiplication between the first five elements of array a and elements 6 through 10 of array b. The 'sum' operator in the second statement is a primitive operator

Figure 7. Dot product in HPF.

```
procedure    dot(a,b,r,n)
dimension a[n],b[n],c[n]
c = a * b
r = sum(c)
end
```

provided by HPF and it causes the elements of c to be summed together.

Notice that this program very much resembles a conventional FORTRAN program; there is no reference to multiple processors, nor to sending and receiving data. It is the job of the compiler to distribute the parallel operations specified in the program among available processors. The compiler is also required to determine a storage scheme for the arrays a, b and c. For example, if the parallel computer has p processors, a natural storage scheme might be to store n/p elements of each array on each processor. Finally, notice that the 'sum' operator expresses communication as well as addition. To compute the sum it is necessary to collect together the partial sums generated on different processors. As may be observed, compiling an HPF program to efficiently run on an arbitrary parallel computer is a formidable task. But substantial strides have been made in this direction.

Several other parallel programming languages have also been developed; many are extensions of common uniprocessor languages like C, Prolog, and Lisp. Writing efficient compilers for these languages (like HPF) is a very challenging task and is the subject of intense research.

The most ambitious approach to parallel computing is to develop a 'supercompiler' that takes as input, programs written in an ordinary language like FORTRAN or C and generates code for multiple processors that perform the same computation as expressed in the original program, only faster. Notice that the task of such a compiler is much harder than a compiler for a language like HPF. The HPF compiler knows directly that an operation such as "a=b*c" can be executed in parallel if a, b, and c are arrays. In ordinary FORTRAN the equivalent code (which would consist of a *do loop*) would have to be analyzed by the compiler to infer that the code is expressing component wise multiplication, and therefore can be parallelized. A lot of work

has been done in developing such supercompilers, but they are not always able to generate a code that executes fast on parallel computers.

It might appear from the preceding discussion that a baffling array of options confronts a user who wishes to program a parallel computer. How does the user decide? The answer depends upon whether high speed is the only objective, or whether program portability, ease of program development are also important. For achieving very high speed, programming using send-receive like statements is very likely the best choice; otherwise using a high level programming language might be better. A combination of the two alternatives might also be possible sometimes.

Concluding Remarks

Parallel computing became the subject of intense research in the late 1970s. A major motivating factor was the realization that it is very difficult to keep on improving the speed with which electrical circuits operated. Thus computer designers turned to parallel computing: using several circuits (processors) to get more computation done in a single step.

Over the years parallel computing has become a mature field. Modern parallel computers can perform computations at breath-taking speed. An Intel Paragon system with about 7000 processors recently executed an industry standard benchmark performing 281 billion floating point operations each second[4]. Today parallel computers (having from tens to thousands of processors) are routinely used for the purpose of weather prediction, for designing airplanes, for analysing and predicting stock market fluctuations, for making strikingly realistic animations for use in movies and for wading through large databases.

In spite of the spectacular progress, parallel computing remains a subject of intense research in universities and also in industry.

Suggested Reading

- FT Leighton. *Introduction to parallel algorithms and architectures*. Morgan-Kaufman Publishers, 1991.
- Joseph Ja' Ja'. *An introduction to Parallel Algorithms*. Addison-Wesley Publishers, 1992.
- V Kumar, A Grama, A Gupta and G Karypis. *Introduction to Parallel Computing*. The Benjamin-Cummings Publishing Company, 1994.

[4] Each processor was an Intel i860 processor, which could perform computations at the peak rate of 75 million floating point operations per second.

Even in India, several parallel computers have been built at the Center for the Development of Advanced Computing (C-DAC), Pune, and also at national research labs. The most recent offering from C-DAC is the PARAM Open Frame architecture and this has already demonstrated its usefulness in weather prediction and in molecular modelling.

This was a very brief introduction to the topic of parallel computing. It will have served its purpose if it has acquainted the reader with some of the basic challenges of parallel computing and has aroused curiosity enough to want to read more. The references mentioned (see Suggested Reading) are excellent for beginners.

Address for Correspondence
Abhiram Ranade
Department of Computer Science and Engineering
Indian Institute of Technology
Powai, Mumbai 400076, India

After the first atomic bomb test explosion on July16, 1945 at Alamogordo, New Mexico, USA

"People were transfixed with fright at the power of the explosion. Oppenheimer was clinging to one of the uprights in the control room. A passage from the Bhagavad Gita, the sacred epic of the Hindus, flashed into his mind:

If the radiance of a thousand suns
were to burst into the sky,
that would be like
the splendour of the Mighty One –

Yet, when the sinister and gigantic cloud rose up in the far distance over Point Zero, he was reminded of another line from the same source:
I am become Death, the shatterer of worlds
Sri Krishna, the Exalted One, lord of the fate of mortals, had uttered the phrase."

from: Robert Jungk, *Brighter than a Thousand Suns*