PARALLEL ALGORITHMS AND ARCHITECTURES [1]

W.F.McCOLL
Programming Research Group
Oxford University
11 Keble Road
Oxford OX1 3QD
England

Abstract

In this paper we consider some of the central issues involved in the design of parallel algorithms. We describe several efficient algorithms for idealised shared memory architectures and draw some conclusions as to what would be required to implement them on a realistic physical architecture, i.e. one with distributed memory. We also describe some systolic algorithms for matrix computations, sequence comparison and molecular modelling, and briefly discuss their implementation on arrays of transputers. In the final section we discuss the question of whether the current preoccupation with architectural details in parallel algorithm design is likely to persist. We briefly describe some techniques which show that a physically realistic general purpose parallel architecture based on distributed memory can be constructed which will execute any shared memory parallel algorithm with no significant overhead due to communication. We thus have the attractive prospect in the very near future of architectural independence in parallel algorithm design.

Keywords : algorithms, complexity, computer architecture, molecular modelling, parallel computation, routing, sequence comparison, systolic algorithms.

[1] This work was supported by the Science and Engineering Research Council under grant GR/E01010.

1 Introduction

An extensive set of general techniques for the design and analysis of efficient sequential algorithms has been developed over the last three decades, see, e.g. [1, 13, 38, 40, 41, 42, 59, 67, 69]. The successs of this development is in large measure due to the existence of theoretical models of computation (Random Access Machine, Turing Machine) and natural complexity measures for those models (e.g. time, space) which, since they accurately reflect the true costs of sequential computations, have been almost universally adopted in discussions of algorithmic efficiency. For example, we are able to assert that a new $O(n^2)$ time sequential algorithm is superior to a known $O(n^3)$ one, without giving elaborate details of the various physical representations of data structures within the machine. Only in extreme cases does this simplistic approach to complexity give way to a much more detailed analysis of the physical data representations used. For example, in considering operations on very large databases [22, 70] one typically has to take into account the representation of the database within a hierarchical memory system.

The situation in parallel computing at the present time is dramatically different. Almost all parallel algorithms are designed with a particular architecture (theoretical or practical) in mind. We thus have a wide variety of different types of parallel algorithm. These include PRAM algorithms [18, 29, 55], circuit algorithms (Boolean and arithmetic) [4, 8, 12, 14, 21, 23, 28, 52, 77], comparison networks [2, 7, 42], VLSI algorithms [71], systolic algorithms [47, 51], mesh algorithms [6, 66], hypercube algorithms [26, 33], pyramid algorithms [50] and various types of asynchronous distributed algorithms [9, 27]. Even within a single class, such as the PRAM algorithms, we have further subdivisions on the basis of properties of the shared memory, e.g. concurrent or exclusive read , concurrent or exclusive write etc. This plethora of computational models inhibits the development of a set of general methods for the design of efficient parallel algorithms.

In this paper we consider some of the central issues involved in the design of parallel algorithms. We describe several efficient algorithms for idealised shared memory architectures and draw some conclusions as to what would be required to implement them on a realistic physical architecture, i.e. one with distributed memory. We also describe some systolic algorithms for matrix computations, sequence comparison and molecular modelling, and briefly discuss their implementation on arrays of transputers. In the final section we discuss the question of whether the current preoccupation with architectural details in parallel algorithm design is likely to persist. We briefly describe some techniques which show that a physically realistic general purpose parallel architecture based on distributed memory can be constructed which will execute any shared memory parallel algorithm with no significant overhead due to communication. We thus have the attractive prospect in the very near future of architectural independence in parallel algorithm design. This should greatly assist us in the pursuit of a set of general techniques for the design, analysis and verification of parallel algorithms to match those which we currently

have for sequential algorithms.

For an introduction to the areas of parallel programming and parallel algorithms, see e.g. [3, 9, 16, 26, 29, 55].

2 Shared Memory Parallel Algorithms

Two standard models for shared memory parallel computation are the PRAM and the circuit.

A *parallel random access machine (PRAM)* consists of a collection of processors which compute synchronously in parallel and which communicate with a common global random access memory. In one time step, each processor can do (any subset of) the following - read two values from memory, perform a simple two-argument operation, write a value to memory. There is no explicit communication between processors. Processors can only communicate by writing to, and reading from, the shared memory. We make the following simplifying assumptions.

- The memory size is unbounded.
- The inputs to the computation are stored in the memory when the algorithm is started.
- The number of processors is unbounded.
- Each processor has only enough memory to store the argument and result values.

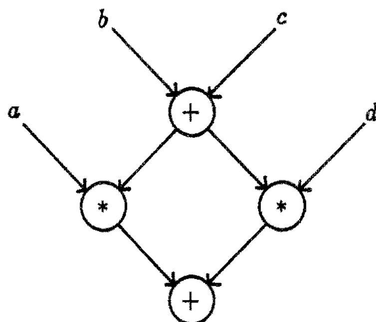
In a *Concurrent Read Concurrent Write (CRCW) PRAM*, any number of processors can read from, or write to, a given memory cell in a single time step. In a *Concurrent Read Exclusive Write (CREW) PRAM*, at most one processor can write to a given memory cell at any one time.

A *circuit* is a directed acyclic graph with n input nodes (in-degree 0) corresponding to the n inputs to the problem, and a number of *gates* (in-degree 2) corresponding to two-argument functions. In a Boolean circuit, the gates are labelled with one of the binary Boolean functions $NAND, \wedge, NOR, \vee, \rightarrow, \oplus$ etc. In a typical arithmetic circuit, the input nodes are labelled with some value from $\mathbb{Q}$, the set of rational numbers, and the gates are labelled with some operation from the set $\{+, -, *, /\}$.

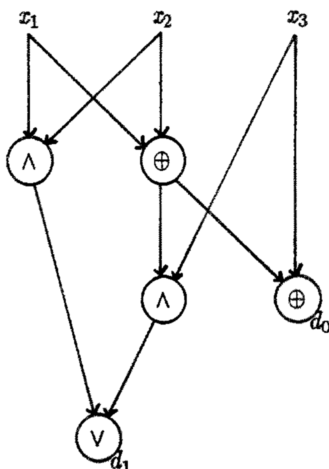
Example 2.1 *PRAM computation of* $ab + ac + bd + cd$ *from inputs* a, b, c, d.

Time step 1	p_1	$b + c \Rightarrow x$
Time step 2	p_1	$a * x \Rightarrow y$
	p_2	$x * d \Rightarrow z$
Time step 3	p_1	$y + z \Rightarrow$ result

Example 2.2 *Arithmetic circuit for $ab + ac + bd + cd$.*



Example 2.3 *Boolean circuit which computes the two binary digits $< d_1, d_0 >$ of $x_1 + x_2 + x_3$.*



The parallel complexity of a PRAM algorithm is the number of steps. The parallel complexity of a circuit is the depth of the circuit, i.e. the maximum number of gates on any directed path. The parallel complexity of Examples 2.1, 2.2 and 2.3 are all three.

2.1 Addition

Let $ADD_n(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$ where $x_i \in \mathbf{Q}$. A circuit of depth $\lceil log_2 n \rceil$ for ADD_n can easily be obtained by constructing a balanced binary tree of +-gates, with n leaves corresponding to the arguments $x_1, \ldots, x_n$. The optimality of this construction, in terms of depth, follows from the functional dependency of ADD_n on each of its n arguments. If we now define $OR_n(x_1, \ldots, x_n) = \bigvee_{i=1}^{n} x_i$ where $x_i \in \{0, 1\}$, then we have a very similar problem to that of computing ADD_n. We

can easily obtain a PRAM algorithm of complexity $\lceil log_2 n \rceil$ and a Boolean circuit of depth $\lceil log_2 n \rceil$ for OR_n. That this circuit depth is optimal follows from functional dependency. However, as Cook and Dwork [19] observed, there is rather more to the question of the PRAM complexity of OR_n. If we allow concurrent write then OR_n can be computed in one parallel step in an obvious way. Processor i reads x_i from memory location i and if $x_i = 1$ it writes a 1 into location 0. Cook and Dwork [19] show that even on an exclusive read/exclusive write PRAM, OR_n can be computed in less than $\lceil log_2 n \rceil$ steps. They derive an upper bound of $0 \cdot 72 log_2 n$ on the number of steps required. However, they also show that a lower bound of $\Omega(log_2 n)$ holds and thus only a constant factor improvement is possible.

2.2 Polynomial Evaluation

Let $P_n(a_0, a_1, \ldots, a_n, x) = \sum_{i=0}^{n} a_i x^i$ where $a_i, x \in \mathbb{Q}$. The standard sequential algorithm for polynomial evaluation is *Horner's Rule* where to calculate P_n we successively compute

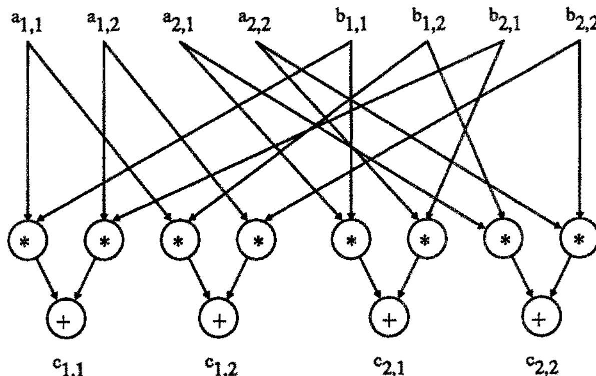
$$\begin{aligned} p_n &= a_n \\ p_i &= (p_{i+1} * x) + a_i \quad for \;\; i = n-1, n-2, \ldots, 1, 0. \end{aligned}$$

Then $P_n = p_0$. The sequential complexity of polynomial evaluation has been studied for many years. It is known that $2n$ arithmetic operations are required to evaluate a general polynomial of degree n, given by its coefficients [12]. Thus, in terms of sequential complexity, Horner's Rule is optimal. However, it is very unsuitable for parallel computation since at every step in the computation the immediately preceding subresult is required. If instead, we evaluate each term $a_i x^i$ of the polynomial independently, in parallel, using a balanced binary tree of $*$-gates and we then sum the values of the terms using a balanced binary tree of $+$-gates then we have a circuit of depth $2\lceil log_2(n+1) \rceil$. This circuit is *exponentially better*, in terms of depth, than a circuit based on Horner's Rule, although the number of gates (sequential complexity) is now $O(n^2)$ rather than $O(n)$. The $2\lceil log_2(n+1) \rceil$ upper bound can be further improved to $log_2 n + O(\sqrt{log_2 n})$ by using a simple recursive parallel algorithm due to Munro and Paterson [53] which splits the polynomial into consecutive blocks of terms and factors out the appropriate power of x. Kosaraju [43] has shown that the algorithm of Munro and Paterson is optimal.

2.3 Matrix Multiplication

Let A, B be two $n \times n$ matrices of rational numbers. Then the product of A, B is an $n \times n$ matrix C, where $c_{i,j} = \sum_{k=1}^{n} a_{i,k} * b_{k,j}$. The exact determination of the sequential complexity of matrix multiplication is a major open problem in the field of computational complexity [17, 54, 68]. At the present time, the best known algorithm (asymptotically, as $n \to \infty$) requires only $O(n^{2 \cdot 376})$ arithmetic operations [20] as opposed to the standard $O(n^3)$. No lower bound larger than the trivial $\Omega(n^2)$ is known.

In contrast, determining the shared memory parallel complexity of matrix multiplication is trivial. We can evaluate each $c_{i,j}$ term independently, in parallel, by a balanced binary tree of depth $\lceil log_2 n \rceil + 1$. Functional dependency shows this bound for $c_{i,j}$ to be optimal and so we have an optimal bound for parallel matrix multiplication. Despite having obtained an optimal algorithm for an idealised machine (PRAM, circuit) we still have some way to go to produce an efficient algorithm which can be physically realised. The crux of the problem can be seen by considering the circuit graph for 2×2 matrices.



If we make the (reasonable) assumption that the arguments to the problem are not replicated within the physical parallel machine, then the above diagram suggests that more time might be spent in moving the arguments around than in performing the arithmetic operations. Precise results of this form can, in fact, be obtained using information-theoretic arguments. For example, motivated by VLSI considerations, we might restrict our circuits to be planar. Using information flow arguments we can establish lower bounds of $\Omega(n^4)$ and $\Omega(n^2)$ on the size (number of gates) and depth respectively, of any planar Boolean matrix multiplication circuit [65]. Such results, indicating the dominance of the cost of communication in parallel computations, are by no means unusual, they are very common indeed. The problem of managing communication is a central task in the design of efficient parallel algorithms.

2.4 Linear Recurrences

The mth Fibonacci number f_m is given by the second order linear recurrence

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_m &= f_{m-1} + f_{m-2} \quad for \quad m \geq 2 \end{aligned}$$

This definition can be directly translated into an arithmetic circuit with $m-1$ gates (and depth $m-1$) which successively computes $f_2, f_3, \ldots, f_n$. As in the case of Horner's Rule we have a circuit with no direct parallel speedup. If instead, we use the unconventional definition

$$(f_{m-1}\, f_m) \;=\; (f_0\, f_1)\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{m-1}$$

then we see immediately that f_m can be calculated by an arithmetic circuit of *size and depth* $O(log_2 m)$ if we compute the matrix power efficiently by repeated squaring.

The above result for Fibonacci numbers is a special case of the following more general result by Greenberg et al. [30] on the parallel evaluation of kth order linear recurrrences. If we have $F = (f_0\, f_1\, \ldots\, f_{k-1})$ and $f_m = \sum_{j=1}^{k} a_{k-j} * f_{m-j}$ for $m \geq k$, then $(f_{m-k+1}\, \ldots\, f_m) = F * M^{m-k+1}$ where M is the $k \times k$ matrix

$$\left(\begin{array}{c|c} 0 \cdots 0 & a_0 \\ \hline & a_1 \\ I & \vdots \\ & a_{k-1} \end{array}\right)$$

and therefore the parallel complexity of computing f_m is at most $O(log_2 k \cdot log_2(m-k))$.

For a practical application of this result we consider the problem of solving linear systems. Let B be an $n \times n$ non-singular, lower triangular matrix, and $\underline{c}$ be an n-element vector. In solving the linear system $B\underline{x} = \underline{c}$ by 'back substitution' we use the recurrence $x_i = (c_i - \sum_{j=1}^{i-1} b_{i,j} * x_j)/b_{i,i}$ for $1 \leq i \leq n$. If we let $x_i = 0$ for $i < 1$ then we can rewrite this recurrence in the form $(x_i\, x_{i-1}\, \ldots\, x_{i-n+1}\, 1) = (x_{i-1}\, x_{i-2}\, \ldots\, x_{i-n}\, 1) * M_i$ where

$$M_i = \left(\begin{array}{c|ccc|c} -\frac{b_{i,i-1}}{b_{i,i}} & & & & \\ \vdots & & & & \\ -\frac{b_{i,1}}{b_{i,i}} & & \mathrm{I} & & 0 \\ 0 & & & & \\ \vdots & & & & \\ 0 & & & & \\ \hline 0 & 0 & \cdots\cdots\cdots & 0 & 0 \\ -\frac{c_i}{b_{i,i}} & 0 & \cdots\cdots\cdots & 0 & 1 \end{array}\right)$$

Therefore we can design an arithmetic circuit of depth $O((log_2 n)^2)$ which solves $B\underline{x} = \underline{c}$ to obtain $\underline{x}$.

2.5 Sequence Comparison

Let $X_m = x_1 x_2 \ldots x_m$ and $Y_n = y_1 y_2 \ldots y_n$ be two sequences. We are interested in the minimum number of insertions and deletions required to change X_m into Y_n. We assume that each insertion or deletion costs 1. Very large instances of this problem arise in the area of molecular biology where the strings correspond to nucleic acid sequences [64]. The edit distance is a measure of the similarity of the two sequences

When $m = 0$ we have $Cost(X_m, Y_n) = n$. Similarly, $n = 0$ gives $Cost(X_m, Y_n) = m$. If $m > 0$, $n > 0$ then we have

$$\begin{aligned} Cost(X_m, Y_n) &\leq Cost(X_{m-1}, Y_{n-1}) && \text{if } x_m = y_n \\ Cost(X_m, Y_n) &\leq 1 + Cost(X_{m-1}, Y_n) && [\text{ delete } x_m] \\ Cost(X_m, Y_n) &\leq 1 + Cost(X_m, Y_{n-1}) && [\text{ insert } y_n] \end{aligned}$$

Using dynamic programming we can tabulate the values of *Cost* in a straightforward way.

Example 2.4 *DEFINE* ⇒ *DESIGN*

	∅	D	E	F	I	N	E
∅	[0]	1	2	3	4	5	6
D	1	[0]	1	2	3	4	5
E	2	1	[0]	[1]	2	3	4
S	3	2	1	[2]	3	4	5
I	4	3	2	3	[2]	3	4
G	5	4	3	4	[3]	4	5
N	6	5	4	5	4	[3]	[4]

The number below x_i and to the right of y_j is $Cost(X_i, Y_j)$. The chain of bracketed numbers indicate that we should perform the following sequence of edits to change DEFINE into DESIGN : Delete F, insert S, insert G, delete E. An arithmetic circuit of size $O(m*n)$ and depth $O(m+n)$ can be obtained by performing the calculations on the ith diagonal, in parallel, at the ith level in the circuit. We will return to the sequence comparison problem in Section 3.3.

3 Systolic Algorithms

H.T.Kung [45, 46] has persuasively argued the case for the design of parallel algorithms which carefully balance the input/output, communication and processing in parallel computations to avoid bottlenecks and hence increase throughput. A systolic parallel algorithm typically has the following properties

- computation proceeds synchronously
- only local computation is required
- operations are pipelined to balance input/output, computation and processing
- the processor architecture has a simple, regular (usually mesh) structure.

The synchronous property is not an essential feature and, indeed, many asynchronous systolic (or wavefront) algorithms have been developed [47].

3.1 Matrix Multiplication

For our first example of a systolic algorithm we consider again the problem of multiplying two $n \times n$ matrices A, B to produce C. The systolic matrix multiplication algorithm is based on the following simple program.

$\underline{Program_{i,j}}$

Time step 0	:	$c_{i,j} := 0;$
Time step $i+j+t-2,\ 0 < t \leq n$	:	Input$(a_{i,t}, b_{t,j})$; $c_{i,j} := c_{i,j} + (a_{i,t} * b_{t,j});$ Output$(a_{i,t}, b_{t,j})$;

Program$_{i,j}$ is implemented on processor $P_{i,j}$ in an $n \times n$ array of processors corresponding to the elements of C. To ensure the correct synchronous dataflow we need to do the following.

- Provide a communication channel from $P_{i,j}$ to $P_{i,j+1}$ and from $P_{i,j}$ to $P_{i+1,j+1}$.
- Supply $a_{i,j}$ as input to processor $P_{i,1}$ at time $i+j-1$.
- Supply $b_{i,j}$ as input to processor $P_{1,j}$ at time $i+j-1$.

At time step $3n-1$ the product matrix C is stored in the $n \times n$ processor array and can be output using the communication channels.

3.2 Algebraic Path Problem

Let G be a complete directed graph on n nodes and A be a matrix of values corresponding to the arcs of G. The *Algebraic Path Problem (APP)* is the problem of computing $A^* = I \oplus A \oplus A^2 \oplus A^3 \oplus \cdots$ where matrix product is defined in terms of two operations $\oplus$ and $\otimes$.

The APP is a problem of major importance in a variety of areas and has been extensively studied in recent years. Some simple examples of the APP are the following :

A	$\oplus$	$\otimes$	Problem
$\{0,1\}$	$\vee$	$\wedge$	Transitive closure of a directed graph. (Is node i connected to node j by a directed path ?) Warshall [76]
$\mathbb{R} \cup \{+\infty\}$	*min*	$+$	Shortest path from node i to node j. Floyd [25]
$\mathbb{R} \cup \{+\infty\}$	*min*	*max*	Minimum cost spanning tree in a connected undirected graph. Maggs and Plotkin [49]

The APP also finds applications in areas such as parsing and logic programming, and can be used as the basis of algorithms for matrix inversion. For simplicity in our discussion here, we restrict ourselves to the instance of the APP corresponding to the transitive closure of a directed graph. In this case, letting $A^0 = I$, we have $A^* = \bigvee_{i \geq 0} A^i$. Noting that node i is connected to node j if and only if it is connected by a path of length $\leq n-1$, we have

$$\begin{aligned} A^* &= \bigvee_{i=0}^{n-1} A^i \\ &= (A^0 \vee A)^{n-1} \\ &= (A^0 \vee A)^{2^k} \quad \textit{whenever } 2^k \geq n-1 \end{aligned}$$

Therefore, to obtain an efficient shared memory parallel algorithm for the computation of the transitive closure of A we need only set the main diagonal to 1 and repeatedly square the resulting matrix until we have a sufficiently large power. For an $n \times n$ Boolean matrix A, this method yields an arithmetic circuit of depth $O(log_2 n)^2)$. This 'repeated squaring' approach, when combined with the systolic matrix multiplication algorithm yields an $O(n\, log_2 n)$ time systolic algorithm for transitive closure on an $n \times n$ processor array. The problem of designing more efficient systolic algorithms for the APP has been intensively studied in recent years. Guibas et al. [31] describe a simple "three pass" extension of the systolic matrix multiplication algorithm which yields the transitive closure in $O(n)$ steps on an $n \times n$ array. A detailed description of the correctness proof for this algorithm can be found in the text by Ullman [71]. Since the appearance of the paper by Guibas et al.[31], Robert, Rote and others [62, 63] have improved and generalised this algorithm in a number of ways.

3.3 Sequence Comparison

The sequence comparison problem can be efficiently solved using a linear systolic array with two-way communication in which the two sequences are compared as they pass through the array moving in opposite directions. Let $X_m = x_1 x_2 \cdots x_m$ and $Y_n = y_1 y_2 \cdots y_n$ be the two sequences to be compared. We use a linear array of processors. Each processor P_i has

input and output channels (In.L,Out.L) to the processor on the left, P_{i-1}, and similar channels (In.R,Out.R) to the processor on the right, P_{i+1} . We input the sequences $y_1\,1\,y_2\,2\,y_3\,3\,\cdots\,y_n\,n$ and $x_1\,1\,x_2\,2\,x_3\,3\,\cdots\,x_m\,m$ sequentially from the left and right respectively. Once the sequences have been input to the array, the computation proceeds with each processor repeatedly performing the following set of operations every two time steps.

Step 1 : In.R(x_i);
In.L(y_j);
Bool:=($x_i = y_j$);
Out.R(y_j);
Out.L(x_i);

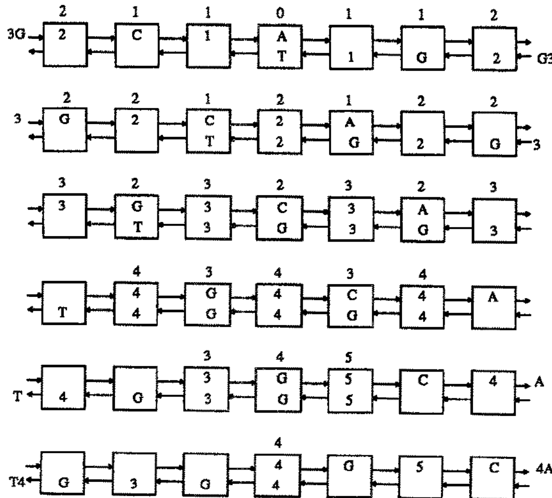
Step 2 : In.R($Cost(X_i, Y_{j-1})$);
In.L($Cost(X_{i-1}, Y_j)$);

$$Cost(X_i, Y_j) = minimum \begin{cases} 1 + Cost(X_i, Y_{j-1}) \\ 1 + Cost(X_{i-1}, Y_j) \\ Cost(X_{i-1}, Y_{j-1}) \text{ if } Bool \end{cases}$$

Out.R($Cost(X_i, Y_j)$);
Out.L($Cost(X_i, Y_j)$);

The following simple example illustrates how the values in the array change from one time step to the next.

	Ø	T	G	G
Ø	0	1	2	3
A	1	2	3	4
C	2	3	4	5
G	3	4	3	4



A simple inductive argument shows that the matrix of edit distances has the property that horizontally and vertically adjacent pairs of values always differ by 1. Because of this property it is sufficient to keep the remainder modulo 4 of the matrix entry. If we implement the above systolic algorithm but keep only such remainders then we can still reconstruct the true value of the edit distance from either of the two sequences of values $c_{m,1}, c_{m,2}, \ldots, c_{m,n}$ and $c_{1,n}, c_{2,n}, \ldots, c_{m,n}$ which are output by the array ($c_{i,j} = Cost(X_i, Y_j)$ *mod* 4) since

$$c_{m,0} = m \quad , \quad (c_{m,i} = c_{m,i-1} - 1) \vee (c_{m,i} = c_{m,i-1} + 1)$$

$$c_{0,n} = n \quad , \quad (c_{i,n} = c_{i-1,n} - 1) \vee (c_{i,n} = c_{i-1,n} + 1)$$

At Oxford we have implemented this algorithm on a small array of 40 T800 transputers, each with 1/4 MByte memory. In matching nucleic acid sequences (four letter alphabet) we need only two bits per character and, as we have seen, two bits per value. This greatly reduces the communication and storage costs and significantly improves the performance of the implementation. Nucleic acid sequence databases are already growing in size at an astonishing rate. The Human Genome Project promises to further accelerate this growth. Algorithmic techniques for pattern matching and database searching, such as those described, will be crucial to future developments in this area.

3.4 Molecular Modelling

Molecular modelling is one of the most important practical applications of parallel architectures. Ab initio methods [34] are based on quantum-mechanical calculations of molecular structure. The computation times required for such calculations, even with highly parallel architectures, are at present prohibitive for all but the smallest molecules. However, as the power of parallel machines increases we can expect a much wider use of this approach.

At present, most calculations of the structure and dynamics of proteins, carbohydrates and nucleic acids are carried out using molecular mechanics and molecular dynamics techniques. These methods adopt a force field approach to calculating the potential energy of a molecule in a given conformation. The *AMBER* force field is quite typical. Let $i \Leftrightarrow j$ denote the (symmetric) relation that atom i is bonded to atom j. Then in *AMBER* the potential energy is calculated as a sum of the following types of term.

- Bond energies for all $i, j,\ i \Leftrightarrow j$
- Angle energies for all $i, j, k,\ (i \Leftrightarrow k) \wedge (k \Leftrightarrow j)$
- Torsional energies for all $i, j, k, l,\ (k \Leftrightarrow i) \wedge (i \Leftrightarrow j) \wedge (j \Leftrightarrow l)$
- Coulombic potential for all $i, j,\ (i \not\Leftrightarrow j)$
- Lennard-Jones potential for all $i, j,\ (i \not\Leftrightarrow j)$
- Hydrogen bonding term for all $i, j,\ (i \not\Leftrightarrow j)$

The problem of calculating the potential energy of a molecular conformation is similar in many ways to the sequence comparison problem. They are both instances of "all pairs" computations. For any such computation, a linear systolic array provides a convenient pattern of dataflow. In the case of the potential energy calculation we simply pass one copy of the atomic coordinates and bond information of the structure past another in the array. When the coordinate and bond information for atom i meets that for atom j, we execute the following program.

if	$i \Leftrightarrow j$	then	calculate bond term; for all $k, l,\ (k \Leftrightarrow i) \wedge (i \Leftrightarrow j) \wedge (j \Leftrightarrow l)$, calculate torsional terms;
elseif	$(i \Leftrightarrow k) \wedge (k \Leftrightarrow j)$	then	calculate angle energy;
		else	calculate Coulombic energy; calculate Lennard-Jones potential; calculate hydrogen bonding term;

Let $d_{i,j}$ denote the distance between atoms i, j. The contributions from the Lennard-Jones

and hydrogen bonding terms rapidly decline with increasing $d_{i,j}$ and can safely be neglected beyond a certain predefined cut-off distance. Unfortunately, the Coulombic potential does not decline quickly, and it is the calculation of these Coulombic terms for all non-bonded pairs which dominates the energy calculation for large molecules.

We have used this approach to produce a highly parallel imlementation of *AMBER* which runs on an array of transputers attached to a SUN workstation.

3.5 Systolic Programming

The systolic approach was initially conceived as a means of producing high-performance special-purpose hardware systems. However, most of the systolic algorithms which have been developed in recent years can be quite naturally viewed as parallel programs to be implemented on standard communicating process architectures such as transputer arrays.

The range of applications for which efficient systolic algorithms are known is growing steadily each year [47, 51]. We are also starting to see the development of a theory of systolic parallel programming [15, 16, 35, 47, 60] which will assist in the future design and verification of systolic algorithms.

4 General Purpose Parallel Computers

In the previous sections we have described a number of parallel algorithms for idealised shared memory parallel machines and for distributed memory architectures such as transputer arrays. In this final section we consider some more general questions concerning developments in the areas of algorithms, architectures and programming languages for parallel computation.

There have been many notable successes in recent years in the design of new highly efficient parallel algorithms through the explicit control of input/output, communication and processing. Some of the systolic algorithms are particularly impressive. It should also be noted that for many practical applications in the physical sciences, relatively little detailed design is required to produce an efficient parallel algorithm. Many such problems are "embarrassingly parallel" by virtue of the underlying physical structure.

On the basis of such observations, some would argue that parallel programming need not (and should not) change significantly. The arguments are essentially the same as those used in support of low-level languages for sequential programming, i.e. greater control over physical resources. The arguments against this approach, e.g. the possibility of architecture-independent parallel software, are , however, very strong. It should also be noted that there are many application areas where no parallel algorithms with simple, regular data structures and process structures are known, although efficient algorithms for shared memory machines have been developed [18, 29].

Thus far, the absence of any commercially available parallel systems which simultaneously offered large scale parallelism and a shared memory programming model has resulted in the communicating process approach [37] prevailing by default. However, a number of recent results in theoretical computer science indicate how such such a system could be designed, and the continuing developments in VLSI technology give confidence that very shortly such systems will be widely available. Before pursuing this issue further, we briefly discuss two topics related to general-purpose parallel computers which are now quite well understood, namely the possible communication structures (interconnection networks) for such machines, and the efficiency with which one parallel machine can simulate another.

4.1 Communication Structures

A very large number of papers have been written on the topic of how to connect n nodes (processors) together to produce a parallel architecture which efficiently supports the communicating process programming model. This is a standard problem from the field of algebraic graph theory [10] and is quite well understood. If we restrict ourselves to regular structures with exactly d connections to each node then even with small fixed d, e.g. $d \leq 6$, we have a wide range of possible families of graphs such as toroidal meshes (1D,2D,3D), trees, and shuffle-exchange type graphs such as the Cube-Connected Cycles (CCC) [58].

Ease of programming and avoidance of bottlenecks both suggest that a useful feature for any such communication graph structure is that it should look isomorphic from any node, i.e. it should be vertex-transitive [10]. This property tends to rule out trees as a communication structure, although they have many other desirable features.

We can compare meshes and shuffle-exchange type graphs in a variety of ways. Perhaps the main advantage of the meshes (up to 3D!) is that they can be directly embedded in 3D physical space. The main advantage of the shuffle graphs, on the other hand, is that they have diameter $O(log_2 n)$, i.e. the distance between any two nodes is at most $O(log_2 n)$. For a k-dimensional mesh, the diameter is $O(\sqrt[k]{n})$.

If we allow a nonconstant number of connections per node, then we can, for example, construct a $(log_2 n)$-dimensional hypercube. This structure has $log_2 n$ connections per node, has diameter $log_2 n$, and is vertex-transitive. The main advantage of the hypercube over fixed-degree shuffle graphs such as the CCC is the increased communication bandwidth afforded by the increase in the number of connections per node.

For more on the topic of communication structures, see e.g. [9, 39, 71, 74].

4.2 Embedding

Another issue which has received a great deal of attention in recent years is efficient graph embedding, i.e. the task of efficiently mapping a given graph onto some standard host graph.

There are a number of situations in parallel computing where one encounters this problem. In laying out VLSI circuits one has often to efficiently embed some arrangement of wires, e.g. a binary tree of wires, onto a planar grid structure. Another example is where one has a standard multiprocessor architecture, e.g. a hypercube, and one requires some other process structure, perhaps a tree or a mesh. The final example is closely related to the previous one , but is worthy of explicit mention. It is where one has a large virtual process structure and a much smaller physical structure of the same form. For example , one might have an algorithmic problem which naturally gives rise to a 100×50 mesh of processes, but have only an 8×8 physical mesh on which to implement it. Such situations arise frequently when programming in a language such as *occam*. There will typically be a number of possible ways of allocating the large set of virtual processes to the much smaller set of physical processors. Which of these possible process mappings is most appropriate will often depend crucially on the balance between communication and computation in the processes.

For more details on solutions to some of these problems , see e.g. [5, 24, 32, 44, 48, 57].

4.3 Routing

A major challenge for contemporary computer science is to determine the extent to which one can have architectural independence in parallel algorithms and parallel programming languages. The central task in this direction is to design highly parallel distributed memory architectures which efficiently implement the idealised shared memory model, using only a 'sparse' communication structure, e.g. at most $log_2 n$ connections per node.

Over the last decade, a series of results in theoretical computer science by Valiant, Brebner, Upfal, Pippenger, Ranade and others [56, 61, 72, 74, 73, 75] have demonstrated conclusively that there is no theoretical impediment to providing such a system. They have shown that using ideas such as randomised two-phase routing one can implement shared memory algorithms on architectures such as the hypercube with no significant loss of efficiency. In randomised two-phase routing, one sends a message from node i to node j not by sending it directly by a shortest path, but by sending it initially from i to some randomly chosen node k in the structure (by a shortest path) , and from there to j (again by a shortest path).

Some parallel computer systems, such as the Ametek system and the Intel iPSC-2, currently provide hardware support for routing non-local messages. The next step is to provide enough communication bandwidth and memory management hardware to directly support shared memory. We can confidently expect that within five years such features will be standard on most parallel architectures. This, of course, raises the question of what else might be provided as "primitive" on future parallel architectures. There is a strong case for examining the extent to which it is cost-effective to provide hardware support for parallel operations on complex data structures. This topic is likely to receive a lot of attention over the next few years [11, 36].

5 Conclusion

We have described some simple parallel algorithms for two idealised shared memory models, PRAMs and circuits. We have also shown some systolic algorithms which can be efficiently implemented either directly in hardware or using arrays of transputers.

The comparative ease with which one can design algorithms for a shared memory machine is likely to be a decisive factor in determining the form of future parallel architectures. Hardware support for routing and shared memory management will result in general-purpose parallel architectures which are efficient for a wide range of problems. However, in those application areas where no overhead whatsoever is considered acceptable and where the financial and technical resources are available, dedicated special-purpose systems will continue to be developed (at a cost) using programming notations which allow one to explicitly control input/output, communication etc. as in systolic programs.

The "special-purpose systems approach" will probably continue to be adopted for some time in many of the areas which currently absorb large amounts of time on parallel architectures, e.g. fluid dynamics, molecular modelling, signal processing, image processing, computer graphics. The advantages gained by adopting this approach are likely to diminish, however, with increases in the scale of parallelism.

Our conclusion then is that the current degree of diversity in architectures for parallel computation will be greatly reduced over the next few years. We can at last look forward to parallelism having an impact in application areas where one has complex data structures and/or much less regular patterns of computation, e.g. combinatorial optimisation, sparse matrix computations, symbolic computation, artificial intelligence. We can also expect to see the emergence of a set of general techniques for the design, analysis and verification of efficient parallel algorithms to match those which we currently have for sequential algorithms.

References

[1] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] AKL, S. G. *Parallel Sorting Algorithms.* Academic Press, 1985.

[3] AKL, S. G. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[4] ALT, H. Comparing the combinational complexities of arithmetic functions. *Journal of the ACM 35*, 2 (Apr. 1988), 447–460.

[5] ATALLAH, M. J., AND HAMBRUSCH, S. E. Solving tree problems on a mesh-connected processor array. In *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science* (1985), pp. 222–231.

[6] ATALLAH, M. J., AND KOSARAJU, S. R. Graph problems on a mesh-connected processor array. In *Proc. 14th Annual ACM Symposium on Theory of Computing* (1982), pp. 345–353.

[7] BATCHER, K. E. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference* (1968), pp. 307–314.

[8] BERKOWITZ, S. J. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters 18* (1984), 147–150.

[9] BERTSEKAS, D. P., AND TSITSIKLIS, J. N. *Parallel and Distributed Computation - Numerical Methods.* Prentice Hall, 1989.

[10] BIGGS, N. *Algebraic Graph Theory.* Cambridge University Press, 1974.

[11] BLELLOCH, G. Scans as primitive parallel operations. In *Proc. International Conference on Parallel Processing* (1987).

[12] BORODIN, A., AND MUNRO, I. J. *The Computational Complexity of Algebraic and Numeric Problems.* Theory of Computation Series. American Elsevier, 1975.

[13] BRASSARD, G., AND BRATLEY, P. *Algorithmics - Theory and Practice.* Prentice Hall, 1988.

[14] BRENT, R. P. The parallel evaluation of general arithmetic expressions. *Journal of the ACM 21* (1974), 201–206.

[15] CAPPELLO, P. R., AND STEIGLITZ, K. Unifying VLSI array design with linear transformations of space-time. *Advances in Computing Research 2* (1984), 23–65. Jai Press Inc.

[16] Chandy, K. M., and Misra, J. *Parallel Program Design : A Foundation.* Addison-Wesley, 1988.

[17] Cook, S. A. An overview of computational complexity. *Communications of the ACM 26*, 6 (1983), 400–408.

[18] Cook, S. A. A taxonomy of problems with fast parallel algorithms. *Information and Control 64*, (1-3) (1985), 2–22.

[19] Cook, S. A., and Dwork, C. Bounds on the time for parallel RAM's to compute simple functions. In *Proc. 14th Annual ACM Symposium on Theory of Computing* (1982), pp. 231–233.

[20] Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. In *Proc. 19th Annual ACM Symposium on Theory of Computing* (1987), pp. 1–6.

[21] Csanky, L. Fast parallel matrix inversion algorithms. *SIAM Journal on Computing 5* (1976), 618–623.

[22] Date, C. J. *An Introduction to Database Systems*, fourth ed., vol. 1 of *Systems Programming Series.* Addison-Wesley, 1986.

[23] Dunne, P. E. *The Complexity of Boolean Networks*, vol. 29 of *A.P.I.C. Studies in Data Processing.* Academic Press, 1988.

[24] Fiat, A., and Shamir, A. Polymorphic arrays : A novel VLSI layout for systolic computers. In *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science* (1984), pp. 37–45.

[25] Floyd, R. W. Algorithm 97 : Shortest path. *Communications of the ACM 5*, 6 (1962), 345.

[26] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W. *Solving Problems on Concurrent Processors : Volume 1. General Techniques and Regular Problems.* Prentice Hall, 1988.

[27] Gallager, R. G., Humblet, P. A., and Spira, P. M. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems 5* (1983), 66–77.

[28] Gathen, von zur, J. Parallel arithmetic computations : A survey. In *Proc. Mathematical Foundations of Computer Science1986, LNCS Vol. 233* (1986), Springer-Verlag, pp. 93–112.

[29] Gibbons, A. M., and Rytter, W. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[30] Greenberg, A. C., Ladner, R. E., Paterson, M. S., and Galil, Z. Efficient parallel algorithms for linear recurrence computation. *Information Processing Letters 15*, 1 (Aug. 1982), 31–35.

[31] Guibas, L. J., Kung, H. T., and Thompson, C. D. Direct VLSI implementation of combinatorial algorithms. In *Proc. Caltech Conference on VLSI* (1979), C. Seitz, Ed., pp. 509–525.

[32] Gupta, A. K., and Hambrusch, S. E. Optimal three-dimensional layouts of complete binary trees. *Information Processing Letters 26* (1987), 99–104.

[33] Heath, M. T., Ed. *Hypercube Multiprocessors 1986.* SIAM, Philadelphia, 1986.

[34] Hehre, W. J., Radom, L., v.R Schleyer, P., and Pople, J. A. *Ab Initio Molecular Orbital Theory.* John Wiley and Sons, 1986.

[35] Hennessy, M. Proving systolic systems correct. *ACM Transactions on Programming Languages and Systems 8*, 3 (1986), 344–387.

[36] Hillis, W. D. *The Connection Machine.* MIT Press, 1985.

[37] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice Hall, 1985.

[38] Horowitz, E., and Sahni, S. *Fundamentals of Computer Algorithms.* Pitman, 1978.

[39] Jerrum, M. R., and Skyum, S. Families of fixed degree graphs for processor interconnection. *IEEE Transactions on Computers 33* (1984), 190–194.

[40] Knuth, D. E. *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming.* Addison-Wesley, 1968. (2nd Edition, 1973).

[41] Knuth, D. E. *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming.* Addison-Wesley, 1969. (2nd Edition, 1981).

[42] Knuth, D. E. *Sorting and Searching*, vol. 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[43] Kosaraju, S. R. Parallel evaluation of division-free arithmetic expressions. In *Proc. 18th Annual ACM Symposium on Theory of Computing* (1986), pp. 231–239.

[44] Kosaraju, S. R., and Atallah, M. J. Optimal simulations between mesh-connected arrays of processors. In *Proc. 18th Annual ACM Symposium on Theory of Computing* (1986), pp. 264–272.

[45] Kung, H. T. Why systolic architectures? *IEEE Computer 15*, 1 (Jan. 1982), 37–46.

[46] KUNG, H. T. Memory requirements for balanced computer architectures. *Journal of Complexity 1*, 1 (Oct. 1985), 147–157.

[47] KUNG, S. Y. *VLSI Array Processors.* Prentice Hall, 1988.

[48] LEIGHTON, F. T. *Complexity Issues in VLSI : Optimal Layouts for the Shuffle-Exchange Graph and Other Networks.* MIT Press, 1983.

[49] MAGGS, B. M., AND PLOTKIN, S. A. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters 26* (1988), 291–293.

[50] MILLER, R., AND STOUT, Q. F. Data movement techniques for the pyramid computer. *SIAM Journal on Computing 16*, 1 (1987), 38–60.

[51] MOORE, W., MCCABE, A., AND URQUHART, R., Eds. *Systolic Arrays.* Adam Hilger, 1987.

[52] MULLER, D. E., AND PREPARATA, F. P. Bounds to complexities of networks for sorting and switching. *Journal of the ACM 22*, 2 (1975), 195–201.

[53] MUNRO, I. J., AND PATERSON, M. S. Optimal algorithms for parallel polynomial evaluation. *Journal of Computer and System Sciences* (1973), 189–198.

[54] PAN, V. *How to Multiply Matrices Faster*, vol. 179 of *Lecture Notes in Computer Science.* Springer-Verlag, 1984.

[55] PARBERRY, I. *Parallel Complexity Theory.* Pitman, 1987.

[56] PIPPENGER, N. J. Parallel communication with limited buffers. In *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science* (1984), pp. 127–136.

[57] PREPARATA, F. P. Optimal three-dimensional VLSI layouts. *Mathematical Systems Theory 16* (1983), 1–8.

[58] PREPARATA, F. P., AND VUILLEMIN, J. The Cube-Connected Cycles : A versatile network for parallel computation. *Communications of the ACM 24*, 5 (1981), 300–309.

[59] PURDOM JR., P. W., AND BROWN, C. A. *The Analysis of Algorithms.* Holt, Rinehart and Winston, 1985.

[60] QUINTON, P. The systematic design of systolic arrays. In *Automata Networks in Computer Science Theory and Applications* (1987), F. F. Soulie, Y. robert, and M. Tchuente, Eds., Manchester University Press, pp. 229–260.

[61] RANADE, A. G. How to emulate shared memory. In *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science* (1987), pp. 185–194.

[62] ROBERT, Y., AND TRYSTRAM, D. Systolic solution of the algebraic path problem. In *Systolic Arrays* (1986), W. Moore, A. McCabe, and R. Urquhart, Eds., Adam Hilger, pp. 171–180.

[63] ROTE, G. A systolic array algorithm for the algebraic path problem (shortest paths ; matrix inversion). *Computing 34* (1985), 191–219.

[64] SANKOFF, D., AND KRUSKAL, J. B., Eds. *Time Warps, String Edits, and Macromolecules : The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.

[65] SAVAGE, J. E. Planar circuit complexity and the performance of VLSI algorithms. In *VLSI Systems and Computations* (1981), H. T. Kung, B. Sproull, and G. Steele, Eds., Computer Science Press, pp. 61–68. (Expanded version appears as INRIA Report No.77 (1981).).

[66] SCHNORR, C. P., AND SHAMIR, A. An optimal sorting algorithm for mesh connected computers. In *Proc. 18th Annual ACM Symposium on Theory of Computing* (1986), pp. 255–263.

[67] SEDGEWICK, R. *Algorithms*, second ed. Addison-Wesley, 1988.

[68] STRASSEN, V. Gaussian elimination is not optimal. *Numerische Mathematik 13* (1969), 354–356.

[69] TARJAN, R. E. *Data Structures and Network Algorithms.* SIAM, 1983.

[70] ULLMAN, J. D. *Principles of Database Systems*, second ed. Pitman, 1982.

[71] ULLMAN, J. D. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[72] UPFAL, E., AND WIGDERSON, A. How to share memory in a distributed system. In *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science* (1984), pp. 171–180.

[73] VALIANT, L. G. A scheme for fast parallel communication. *SIAM Journal on Computing 11*, 2 (1982), 350–361.

[74] VALIANT, L. G. General purpose parallel architectures. In *Handbook of Theoretical Computer Science* (To appear), J. van Leeuwen, Ed., North Holland.

[75] VALIANT, L. G., AND BREBNER, G. J. Universal schemes for parallel communication. In *Proc. 13th Annual ACM Symposium on Theory of Computing* (1981), pp. 263–277.

[76] WARSHALL, S. A theorem on Boolean matrices. *Journal of the ACM 9*, 1 (1962), 11–12.

[77] WEGENER, I. *The Complexity of Boolean Functions.* John Wiley and Sons, 1987.