Lecture 2

Communication and computation in some parallel algorithms*

*Six lectures on Algorithms, Trinity term 1999.

 lec02

Summary

- The importance of communication in hardware
  - Area-time bounds for VLSI chips
- Linear algebra
  - Matrix multiplication
  - Solution of linear systems
  - The SVD and eigenvalue problems
- Non-numerical problems
  - Merging
  - Sorting

Area-time bounds for VLSI

It is interesting to consider algorithms implemented in *hardware* before we consider those implemented in *software*.

Specifically, consider the model of VLSI circuits described in Ullman's book *Computational Aspects of VLSI* [13]. Variations on this model were used by Thompson [9], Brent and Kung [5], Brent and Goldschlager [4], etc to obtain lower bounds on the *area* and *time* required for some fundamental computations, e.g. binary multiplication, sorting, the FFT, evaluation of propositional calculus formulae, set equality, context-free language recognition, etc.

Sketch of the model

- A computation is performed in a planar, convex region R of area A.
- Wires of finite width λ are used for communication within R.
- I/O ports of finite size on the boundary of R are used for input and output (communication with the outside world).
- Wires are allowed to overlap, but the degree of overlap is bounded by ν, e.g. $\nu = 2$ is sufficient.
- Each input is read only once.
- Storage of one bit of information takes a fixed area.
- The circuit is synchronous with a fixed cycle time τ.
- Wires can transmit one bit in time τ.

For details, see Ullman [13] or the original papers.

Bisection

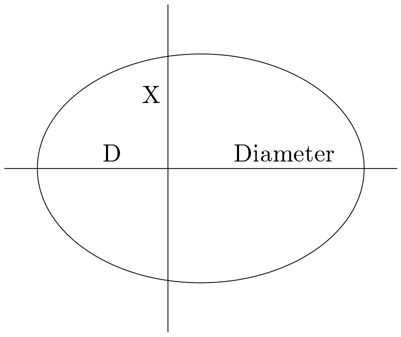


Figure 1: A VLSI chip

The ellipse is the boundary of a VLSI chip. D is the diameter and X is a chord of length L perpendicular to the diameter.

Given a nonempty set V of processing elements (ports and/or gates), X is chosen so that some of the elements of V lie on each side of it. This is called a *bisection* of V.

Information transfer

The (maximal) information transfer across X during a computation of time T is

$$I = WT/\tau \ ,$$

where W is the number of wires which intersect X.

Theorem. If there is a bisection of V with information transfer I then

$$AT^2 = \Omega(I^2) \ .$$

Idea of proof: $A = \Omega(L^2)$ and $L \geq \lambda W/\nu$, so

$$AT^2 = \Omega(W^2T^2) = \Omega(I^2) \ .$$

This theorem is from Brent and Goldschlager [4]. There are similar results (with slightly different definitions) in Brent and Kung [5] and Thompson [9].

Applications

To apply the theorem, we use a "crossing sequence" argument to give a lower bound on I, the information which has to cross X. Typically we obtain

$$I = \Omega(n) \ ,$$

where n is a measure of the size of the problem. Thus, we can conclude that

$$AT^2 = \Omega(n^2) \ .$$

For example, this bound applies to n-bit binary multiplication, evaluation of a propositional calculus formula (optionally in disjunctive normal form), and for testing set equality.

AT bounds

Often we can obtain an independent lower bound on the area, typically

$$A = \Omega(n) \ .$$

Then, multiplying the two bounds and taking a square root, we have

$$AT = \Omega(n^{3/2}) \ .$$

A lower bound on AT seems to be more natural than a bound on AT^2.

For example, the $AT = \Omega(n^{3/2})$ bound applies to binary multiplication, and shows that in a sense

multiplication is harder than addition

because we can perform binary addition in the same model with

$$AT = O(n) \ .$$

Parallel machines

There are many examples of parallel computer architectures (a few alive and well, many extinct). They all involve multiple processors but differ in other important respects –

- Memory may be local or shared (or actually local but apparently shared, or some combination, ...)
- The processors (real or virtual) may have independent instruction streams (MIMD) or a common instruction stream (SIMD). If MIMD, we can program them using a common program (SPMD) or different programs in each processor.
- The processors and memories may be connected in various different ways, which may or may not be visible to the programmer. For example, rings, tori, hypercubes, cliques (crossbars), trees, ...
- There may be a small number of powerful processors (perhaps vector processors) or a larger number of feeble processors.

Machine-independent models

It is because of these differences that machine-independent models such as BSP have been introduced. However, no standard has emerged yet, and programmers still resort to low-level features of machines in order to get higher efficiency (typically for benchmarks such as the *Linpack* benchmark, because they help to sell machines).

Data distribution

Given a parallel machine with p processors and a problem with input data D, we have to distribute D over the processors in some manner. (Of course, with a smart enough compiler, the data distribution might be invisible to the programmer.) The result may end up distributed over the processors and we have to specify how this is to occur[1].

[1] Otherwise problems such as sorting are trivial !

Linear algebra on parallel machines

Linear algebra problems with dense matrices provide nice, regular examples and are (relatively) easy to solve with high efficiency on parallel machines. If only all problems were so well-defined and regular !

Matrix multiplication

Consider, for example, the problem of forming the product C of two $n \times n$ matrices A and B. (The rectangular case is interesting and important, but we consider the square case for simplicity.)

Data distribution

We could distribute A over the processors by *rows, columns*, or *blocks*, and similarly for B and C. Whichever way we partition A, B and C, some communication between processors is necessary (unless everything is done on one processor and the others remain idle).

We assume that the classical, $O(n^3)$ matrix multiplication algorithm is used, and that the time for one multiply and add on a single processor (with data in cache etc) is τ. Thus, it would take time $T_1 \approx n^3\tau$ to solve the problem on a single processor. We hope to solve the problem p times faster on p processors, i.e. we hope that the *speedup*

$$S = \frac{T_1}{T_p}$$

is close to p, or that the *efficiency*

$$E = \frac{S}{p} = \frac{T_1}{pT_p}$$

is close to 1. Here T_p is the time required to solve the problem on p processors.

Communication versus computation

With current technology, communication between processors is typically much slower than communication/computation within a processor. To communicate a message of w words might take time

$$(Gw + H)\tau \ ,$$

where G and H are constants, typically much greater than 1.

We can interpret

- $1/(G\tau)$ as the processor to processor *communication bandwidth*, and
- $H\tau$ as the *startup cost* of a communication.

In practice such a simple linear model is inaccurate, but we use it for lack of anything better.

Block distribution

In order to minimise communication costs, the best way to distribute A and B is by blocks. Suppose that $p = s^2$ is a perfect square and $s|n$, for the sake of simplicity[2].

We partition A, B and C into $s \times s$ block matrices, where each block is of size $n/s \times n/s$.

For example, if $s = 2$, $p = 4$, we partition A as

$$A = \left[\begin{array}{c|c} A_{0,0} & A_{0,1} \\ \hline A_{1,0} & A_{1,1} \end{array}\right] ,$$

where each $A_{i,j}$ is a matrix of size $n/2 \times n/2$. (Here and below indices run from zero.)

[2] Such assumptions are usually made by lecturers; only programmers have to consider the difficult cases.

Estimate of efficiency

The processor assigned the block with indices (i, k) has to accumulate the sum

$$C_{i,k} = \sum_j A_{i,j} B_{j,k} \ ,$$

so it needs data from the "block row" i of A and the "block column" k of B.

For each $n/s \times n/s$ matrix product, taking time $(n/s)^3\tau$, the processor needs $2(n/s)^2$ words of data, which can to be transferred in time $2G(n/s)^2 + 2H$. If we neglect H and low order terms, we see that

$$T_p \approx \frac{n^3}{s^2}\left(1 + \frac{Gs}{n}\right) \ ,$$

so

$$E \approx 1/(1 + Gs/n) \ .$$

Hence, E is close to 1 only if

$$n \gg Gs \ .$$

Interpretation

Since G is typically in the range 100 to 1000, the inequality

$$n \gg Gs$$

means that n has to be large relative to $s = \sqrt{p}$.

Thus, we can not make efficient use of a large number of processors unless n is huge.

Gaussian elimination

In practice, it is more likely that we want to solve a linear system

$$Ax = b$$

than multiply two matrices. Consider the method of Gaussian elimination (without pivoting, for the time being). If we partition A into $p = s^2$ blocks as before, a problem of *load balance* rears its head.

As the elimination proceeds, all the "action" moves to the bottom right corner, and more and more processors have nothing to do. This is because A is gradually transformed into the upper triangular matrix U in the matrix factorisation $A = LU$. After the j-th iteration the first j columns are zero below the diagonal (or traditionally are used to store columns of L).

Gaussian elimination

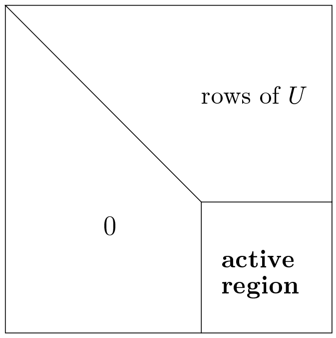


Figure 2: Decomposition $A \to LU$

Solution - scattered decomposition

A solution to the load balance problem is to use a *scattered* or *cyclic* distribution of the data A. Here matrix element $a_{i,j}$ is stored in processor

$$(i \bmod s, j \bmod s)$$

instead of in processor

$$(\lfloor is/n \rfloor, \lfloor js/n \rfloor) \ .$$

For such mappings it is convenient to take indices running from 0, as in C, rather than from 1, because the range of the "mod s" function is $\{0, 1, \ldots, s-1\}$.

Assume a cyclic data distribution. As the computation proceeds, each processor has a matrix of roughly the same "shape" (tending towards upper triangular), and we gain a factor of almost three in efficiency over that obtained using the block decomposition.

Scattered versus block decomposition

We have seen that the scattered decomposition gives better load balance than the block decomposition, at least for Gaussian elimination. (A little thought shows that they are equivalent for matrix multiplication.)

Another advantage of the scattered decomposition is that the location of element $a_{i,j}$ of the matrix A depends only on (i, j) and is *independent* of the dimensions of A. This is a great advantage if we are writing software to operate on matrices of arbitrary shape (not necessarily known at compile time).

Partial pivoting

Except for special classes of matrices, Gaussian elimination is unstable unless we perform pivoting to constrain the size of the multipliers (elements of L).

For example, the use of partial pivoting corresponds the a matrix factorisation

$$PA = LU ,$$

where P is a permutation matrix, chosen so that the elements $m_{i,j}$ of L are at most 1 in absolute value.

Communication overhead of pivoting

On a parallel machine, pivoting introduces additional communication overheads. At the k-th step we need to find the best pivot element in the k-th column, and this requires communication between the processors which have access to elements of the column. The volume of communication is small, but the startup costs are significant. To find the maximum of a vector stored in $s = \sqrt{p}$ processors, using a binary tree, and to broadcast the result to the processors, costs us about $H\tau \lg(p)$ just in startup costs, so a term

$$H\tau n \lg(p)$$

will occur in the estimate of T_p.

H may be about 100 (if special communication hardware is provided) or 10^6 or more (if communication is done entirely in software using interrupts), so the startup cost may well dominate for small and moderate n.

Distribution by columns

Instead of distributing both rows and columns in a scattered (cyclic) manner, it is tempting to distribute just the columns of A in this way. More precisely, element $a_{i,j}$ could be stored in processor

$$j \bmod p ,$$

where the processors are numbered $0, 1, \ldots, p-1$.

The advantage of this "cyclic by column" distribution is that a single processor has access to a whole column, so no communication is required to find the pivot element in that column. Information about the pivot element and its location still has to be broadcast to the other processors.

Disadvantages of distribution by columns

The distribution by columns has significant disadvantages.

- Communication in the horizontal direction has bandwidth reduced by a factor $s = \sqrt{p}$, from $s/(G\tau)$ to $1/(G\tau)$.
- For matrix multiplication $C \leftarrow AB$, do we distribute B by columns (for consistency) or by rows (for compatibility with the definition of matrix multiplication) ?

Generally, it seems preferable to use a distribution where rows and columns are treated in the same way.

Memory references per flop

On many machines it is impossible to achieve close to peak performance if Gaussian elimination is performed in the obvious way (via saxpys or rank-1 updates). This is because performance is limited by memory accesses rather than by floating-point arithmetic. Saxyps and rank-1 updates have a high ratio of memory references to floating point operations.

Close to peak performance can be obtained for matrix-vector or (better) matrix-matrix multiplication which (if implemented properly) have a lower ratio of memory references to floating-point operations.

Blocking

It is possible to reformulate Gaussian elimination so that most of the floating-point arithmetic is performed in matrix-matrix multiplications (level 3 BLAS). The idea is to introduce a "blocksize" or "bandwidth" parameter ω. Gaussian elimination is performed via saxpys or rank-1 updates in vertical strips of width ω. Once ω pivots have been chosen, a horizontal strip of height ω can be updated. At this point, a matrix-matrix multiplication can be used to update the lower right corner of A. The optimal choice of ω is usually about $\sqrt{n}$.

It is important to note that the introduction of the parameter w is *independent* of the data distribution on a parallel machine. There is certainly no need to distribute A in $\omega \times \omega$ blocks. For more on this and related topics, see my paper "The Linpack benchmark on the AP1000" [3].

Illustration

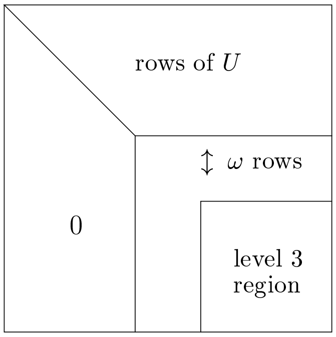


Figure 3: Blocked LU decomposition

The SVD

Suppose $m \geq n$. A singular value decomposition (SVD) of a real $m \times n$ matrix A is its factorisation into the product of three matrices

$$A = U\Sigma V^T ,$$

where U is $m \times n$ with orthonormal columns, Σ is an $n \times n$ non-negative diagonal matrix, and V is an $n \times n$ orthogonal matrix.

The diagonal elements σ_i of Σ are the *singular values* of A. The SVD has many applications (see Golub and Van Loan [7]).

Computing the SVD in parallel

The SVD is usually computed by a two-sided orthogonalisation process, e.g. by two-sided reduction to bidiagonal form followed by the QR algorithm. It is difficult to implement this Golub-Kahan-Reinsch algorithm efficiently on a parallel machine. It is much simpler to use a one-sided orthogonalisation method due to Hestenes.

The idea of Hestenes is to generate an orthogonal matrix V such that AV has orthogonal columns. Normalising the Euclidean length of each non-null column to unity, we get

$$AV = \tilde{U}\Sigma$$

As a null column of $\tilde{U}$ is always associated with a zero diagonal element of Σ, this gives the SVD of A.

Parallel implementation of the Hestenes method

Let $A_1 = A$ and $V_1 = I$. The Hestenes method uses a sequence of plane rotations Q_k chosen to orthogonalise two columns in

$$A_{k+1} = A_k Q_k \ .$$

If the matrix V is required, the plane rotations are accumulated using $V_{k+1} = V_k Q_k$. Under suitable conditions $\lim Q_k = I$, $\lim V_k = V$ and $\lim A_k = AV$. The matrix A_{k+1} differs from A_k only in two columns, say columns i and j, and the new columns are obtained from the old columns by a plane rotation through a certain angle θ, where $|\theta| \le \pi/4$.

It is desirable for a "sweep" of $n(n-1)/2$ rotations to include all pairs (i, j) with $i < j$.

The chess tournament analogy

On a parallel machine we would like to orthogonalise several pairs of columns simultaneously. This should be possible so long as no column occurs in more than one pair. The problem is similar to that of organising a round-robin tournament between n players.

A game between players i and j corresponds to orthogonalising columns i and j, a round of several games played at the same time corresponds to orthogonalising several pairs of (disjoint) columns, and a tournament where each player plays each other player once corresponds to a sweep in which each pair of columns is orthogonalised. Thus, schemes which are well-known to chess players can be used to give orderings amenable to parallel computation.

It is desirable to minimise the number of parallel steps in a sweep, which corresponds to the number of rounds in the tournament.

Lazy players

On a parallel machine with restricted communication paths there are constraints on the orderings which we can implement efficiently. A useful analogy is a tournament of lazy chess players. After each round the players want to walk only a short distance to the board where they are to play the next round.

Using this analogy, suppose that each chess board corresponds to a processor and each player corresponds to a column of the matrix. A game between two players corresponds to orthogonalisation of the corresponding columns. If the chess boards (processors) are arranged in a linear array with nearest-neighbour communication paths, then the players should have to walk (at most) to an adjacent board between the end of one round and the beginning of the next round, i.e. columns of the matrix should have to be exchanged only between adjacent processors. Several orderings satisfying these conditions are known.

The number of steps in one sweep

Since A has n columns and at most $\lfloor n/2 \rfloor$ pairs can be orthogonalised in parallel, a sweep requires as least $n-1$ parallel steps (n even) or n parallel steps (n odd). The ordering of Brent and Luk [6] attains this minimum, and convergence can be guaranteed.

Data distribution

As described, each processor deals with two columns, so the column-wrapped representation is convenient. However, the block or scattered representations can also be used. The block representation involves less communication between processors than does the scattered representation if the standard orderings are used. However, the two representations are equivalent if different orderings are used.

The scattered representation does not have a load-balancing advantage here, since the matrix does not change shape.

The symmetric eigenvalue problem

There is a close connection between the Hestenes method for finding the SVD of a matrix A and the Jacobi method for finding the eigenvalues of the symmetric matrix $B = A^T A$.

Important differences are that the formulas defining the rotation angle θ involve elements $b_{i,j}$ of B rather than inner products of columns of A, and transformations must be performed on the left and right instead of just on the right (since $(AV)^T(AV) = V^T BV$).

Instead of permuting columns of A, we have to apply the same permutation to both rows and columns of B.

An implementation on a square systolic array of $n/2$ by $n/2$ processors is possible, and can be adapted to other parallel architectures. Again, a blocked data representation is desirable to minimise communication costs.

Parallel merging and sorting

To conclude, we consider a "non-numerical" problem – sorting data into order. It is assumed that each item of data has a key and the keys are totally ordered (an example is lexicographic ordering).

First, consider a simpler problem: merging data held on two processors (the solution of this problem will be useful for sorting).

The merge-exchange problem

Suppose processors P and Q each have n items of data and the data on each processor is already sorted. For example, if $n = 4$, processor P might have (A, B, F, Z) and processor Q might have (C, D, E, G).

The problem is to merge the $2n$ items of data (we assume they are all distinct) into one sorted list, and end with the first half of the list on processor P and the last half on processor Q. In our example, P should end up with (A, B, C, D) and Q with (E, F, G, Z).

We could transfer Q's data to P, merge it with P's data, then send the last half of the merged list back to Q. However, this is inefficient because P does all the work (Q is idle while P is merging), and some data may be transferred unnecessarily. Also, P needs more memory than is required just to store $2n$ items.

A solution – first find the median

Suppose the final sorted list is $(A_1, A_2, \ldots, A_{2n}$. Thus, P ends up with $(A_1, \ldots, A_n)$ and Q ends up with $(A_{n+1}, \ldots, A_{2n})$. If the processors can determine, by a small amount of communication and local computation, the value of the *median* element A_n, then a more efficient solution is possible –

P sends to Q all of its elements which are greater than A_n, and Q sends to P all of its elements which are less than or equal to A_n. Then, all each processor has to do is a local merge.

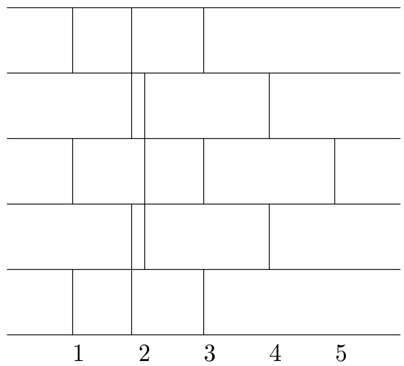
Finding the median can be done by binary search. At each stage, if the candidate median is M say, each processor counts how many of its elements exceed M, and sends the count to the other processor. Each processor now can determine how many elements in the final sorted list will exceed M. If this number is greater than n then M is increased, if less than n then M is decreased, …

The communication tradeoff

Overall, there are about $\lg(n)$ communication steps to find the median. Thus on average, we trade the time required to communicate $O(n)$ elements for $O(\lg n)$ communication startup times. This is worthwhile if n is sufficiently large.

Sorting networks

A *sorting network* is a sorting circuit built from *comparators*, which are circuit elements which can sort two inputs (for the definition, see Knuth, Vol. 3). For example:



The network shown will sort 6 items in 5 steps using 12 comparators.

A generalisation of sorting networks

In a sorting network, we can replace each comparator by a "generalised comparator" which takes two sorted lists, merges them, and outputs the lower and upper halves of the result (as in the merge-exchange problem considered previously).

It can be shown that, *provided* the input and output lists of the generalised comparators are all of the same size k, and the initial inputs are sorted lists of size k, then the generalised network will sort correctly.

The restriction on input sizes is necessary, as examples show, but can be circumvented by the use of "virtual" elements, so is not a problem in practice (see Tridgell and Brent [10]).

Generic parallel sorting algorithm

We can take any serial algorithm which can be implemented as a sorting network (e.g. Batcher's merge-exchange algorithm, see Knuth [8, Algorithm M]), and convert it into a parallel algorithm which uses the merge-exchange operation.

Practical parallel sorting

Simply extending Batcher's algorithm is inefficient. A practical algorithm could add the steps of pre-balancing, fast internal sorting, and perhaps "almost sorting". For details see Tridgell and Brent [10].

Other parallel sorting algorithms

There are many serial sorting algorithms, and even more parallel algorithms. The main competitors appear to be

- Algorithms based on merge-exchange, as above.
- Algorithms based on sample-sort.
- Algorithms based on radix sort.

A disadvantage of the parallel algorithms based on sample-sort and radix-sort is that they require *all to all* communication, whereas algorithms based on merge-exchange require only processor to processor communication.

Another disadvantage of algorithms based on radix sort is that the keys must have fixed length and the ordering of keys can not be defined by the user.

For more on parallel sorting, see Andrew Tridgell's thesis [12].

References

[1] R. P. Brent, Parallel algorithms for digital signal processing, in *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms* Springer-Verlag, 1991, 93–110.

[2] R. P. Brent, Parallel algorithms in linear algebra, *Algorithms and Architectures: Proc. Second NEC Research Symposium* SIAM, Philadelphia, 1993, 54–72.

[3] R. P. Brent, The LINPACK benchmark on the AP 1000, *Proceedings of Frontiers '92* (McLean, Virginia, October 1992), IEEE Press, 1992, 128–135.

[4] R. P. Brent and L. M. Goldschlager, Some area-time tradeoffs for VLSI, *SIAM J. Computing* 11 (1982), 737-747.

[5] R. P. Brent and H. T. Kung, The area-time complexity of binary multiplication, *J. ACM* 28 (1981), 521–534.

[6] R. P. Brent and F. T. Luk, The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays, *SISSC* 6 (1985), 69–84.

[7] G. H. Golub and C. Van Loan, *Matrix Computations*, second edition, Johns Hopkins Press, Baltimore Maryland, 1989.

[8] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Menlo Park, 1981.

[9] C. D. Thompson, Area-time complexity for VLSI, *Proc. 11th ACM Symp. on Theory of Computing*, 1979, 81–88.

[10] A. Tridgell and R. P. Brent, A general-purpose parallel sorting algorithm, *International J. of High Speed Computing* 7 (1995), 285–301.

[11] A. Tridgell, R. P. Brent and B. D. McKay, *Parallel Integer Sorting*, Tech. Report TR-CS-97-10, CSL, ANU, May 1997, 32 pp.

[12] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization*, Ph. D. thesis, Australian National University, 1999.

[13] J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Maryland, 1984.

[14] B. B. Zhou, R. P. Brent and A. Tridgell, Efficient implementation of sorting algorithms on asynchronous distributed-memory machines, *Proc. ICPDS'94*, IEEE CS Press, Los Alamitos, California, 1994, 102–106.