Towards Structured Parallel Computing
Part I
A Theory of Algorithm Design and Analysis for Distributed-Memory Architectures

by

Feng Gao

Technical Report 89-27
December, 1989

Computer Science Department

University of British Columbia

Vancouver, B.C. V6T 1W5 Canada

TOWARDS STRUCTURED PARALLEL COMPUTING
Part I
A Theory of Algorithm Design and Analysis for Distributed-Memory Architectures

Feng Gao

Computer Science Department
University of British Columbia
Vancouver, B.C. V6T 1W5 Canada
December, 1989

ABSTRACT

This paper advocates a architecture-independent, hierarchical approach to algorithm design and analysis for distributed-memory architectures, in contrast to the current trend of tailoring algorithms towards specific architectures. We show that, rather surprisingly, this new approach can achieve uniformity without sacrificing optimality. In our particular framework there are three levels of algorithm design: design of a network-independent algorithm in a network-independent programming environment, design of virtual architectures for the algorithm, and design of emulations of the virtual architectures on physical architectures. We propose and substantiate through a complete complexity analysis of the example of ordinary matrix multiplication, the following thesis: architecture-independent optimality can lead to portable optimality. Namely, a single network-independent algorithm, when optimized network-independently, with the support of properly chosen virtual architectures, can be implemented on a wide spectrum of networks to achieve optimality on each of them with respect to both computation and communication. Besides its implications to the methodology of parallel algorithm design, our theory also suggests new questions for theoretical research in parallel computation on interconnection networks.

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada under Grant OGP0041639, and by the U.S. National Science Foundation under Grant CCR8712121 during the preliminary stage.

1. Introduction

In this paper, we develop a theory of algorithm design and analysis for coarse- and medium-grain parallel computation on distributed-memory architectures. The main goal of developing this theory is to advocate a departure from the current trend of tailoring parallel algorithms towards specific architectures, or interconnection networks (the two terms will be synonymous in this paper when no confusion arises). This conventional approach has several drawbacks: a) the design and programming of parallel algorithms involve a great deal of machine details; b) architecture-driven algorithms are not easily portable across different machines; and c) a common ground for analysis and comparison of algorithms is lacking.

We feel that the success of parallel computing will not lie so much, at the level of algorithm design, in utilizing every detail of a machine as in the ease of algorithm development and analysis, and will lie as much in portability of algorithms as in their optimality.

As a step towards a more structured, hierarchical approach to parallel computing, we propose a framework for algorithm design and analysis which takes a more architecture-independent view. We abstract out the notion of an algorithm from its implementation on a machine, and separate the notion of optimality of the former from that of the latter.

In our theory, the design, analysis and implementation of a parallel algorithm for a given application consists of three levels, each involving optimization. The highest level is independent of any possible underlying interconnection network. An algorithm is designed using data-dependency analysis to achieve a type of *architecture-independent optimality* with respect to both computation and communication. It is also specified in an architecture-independent context, by a partition of the computation tasks and a schedule of the computation tasks as well as the communication-oriented tasks. The communication-oriented tasks are organized around a set of generic primitives specified by their functionality. At the next level, a collection of virtual architectures are chosen to implement the algorithm on (more precisely, to implement the generic primitives on). They are the ones that are tailored, to suit both the algorithm and the hardware architectures. More specifically, they are selected on the basis of their capability to support the generic communication-oriented primitives of the algorithm as well as

their flexibility for emulation on different hardware networks. For special-purpose computing, the third level is the hardware realization of a properly chosen virtual architecture for the algorithm. For general-purpose computing, the third level is the implementation of this architecture-independent algorithm on a spectrum of interconnection networks, by emulating a properly chosen virtual architecture on each hardware network. The two lower levels combine our architecture-independent complexity analysis with tools from current areas of research in parallel computation: parallel algorithms for specific architectures, communication algorithms for specific architectures, and emulation between architectures.

A high level algorithm designed this way is network-independent and can be ported to various architectures. The question is whether the implementations of this algorithm on different architectures will still be competitive with ones tailored for specific networks. At the first sight, it would seem counter-intuitive that one algorithm could run well on several different networks.

To show that the answer is however affirmative, we propose and substantiate the following thesis: architecture-independent optimality of an algorithm can lead to *portable optimality* of the algorithm. Namely, a single algorithm, when optimized architecture-independently, with the support of properly chosen virtual architectures, can be implemented on a wide spectrum of architectures to achieve optimality on each of them.

In addition, we will show that this approach allows the development of an algorithm to be carried out in a more structured as well as network-independent programming environment: when local computation is distinguished from communication-oriented tasks organized around certain generic primitives, one gains in modularity.

We will also show that this approach leads to a framework of optimal interconnection network design for special-purpose computing -- design of sparse networks that can do almost as well as any dense network including the complete-connection network, for their special purposes.

The theory will be illustrated throughout the paper with a model problem -- design of an algorithm for ordinary matrix multiplication (OMM). In subsequent papers, it will be demonstrated that this approach can be successfully applied to more complex applications such as dynamic programming and matrix

factorization, where more complicated resource-tradeoffs arise.

Our specific findings for OMM can be summarized as follows:

a) An algorithm consisting of a three-dimensional partitioning of computation tasks and a schedule of one local computation stage and two communication stages each using one communication-oriented primitive, is asymptotically optimal in an architecture-independent sense, i.e., simultaneously achieving optimal computation parallelism, minimizing the number of rounds of communication, and minimizing data interdependency. The two primitives used are limited-complete-broadcast and limited-histogramming (see Section 3 for their description).

b) Three virtual architectures, the 3-D mesh, a variant of the 3-D mesh of trees, and the 3-D mesh of hypercubes which is a hypercube itself, are selected to implement this 3-d algorithm on. Each is proved to have certain optimality property.

c) For special-purpose network design, the 3-D mesh or the variant of the 3-D mesh of trees can be realized as hardware networks to implement this 3-d algorithm on and yield performance almost as good as that of any network including the complete-connection network.

d) For general-purpose computing, this 3-d algorithm can be implemented on the 1-D mesh, 2-D mesh, 3-D mesh and the hypercube networks to achieve portable optimality, that is, to be asymptotically as good as any algorithm designed for any of these networks. Here optimality is with respect to both computation and communication. These optimal implementations are obtained through optimal emulations of virtual architectures: the 3-D mesh virtual architecture on the 1-D mesh, 2-D mesh and the 3-D mesh itself, and the (3-D mesh of) hypercube(s) virtual architecture on the hypercube itself.

Results concerning the generic communication-oriented primitives, the virtual architectures they are implemented on, and emulation of these virtual architectures are of greater generality and can be used to support algorithms for other applications.

Besides to the process of algorithm design, our theory has implications to theoretical research in parallel computation. For instance, Theorem 3 (Section 6) in particular establishes the communication optimality of the 3-d algorithm on the hypercube for OMM. Most previous results concerning optimality of

communication on the hypercube have been for communication problems rather than computational problems. Although our lower bounds apply only to OMM rather than the general problem of matrix multiplication, it is in the spirit of the traditional theory of algorithms and complexity to investigate the complexity of communication for computation. The proofs for the lower bounds also demonstrate the importance of an architecture-independent analysis. Another instance is Theorem 4 (Section 6) which makes use of embeddings of the 3-D mesh in the 1-D and 2-D meshes. Motivated by the notion of portable optimality, it requires the embedding of a denser graph in sparser ones. It also requires a tight estimate on the optimal number of emulation steps which cannot be obtained by just using the product of dilation and congestion as an upper bound as most previous graph-embedding works have done.

The rest of the paper is organized as follows: Section 2 presents the model of computation. Section 3 presents the general theory for the architecture-independent algorithm design and analysis. Section 4 concerns design of virtual architectures. Section 5 discusses special-purpose computing. Section 6 discusses general-purpose computing, i.e., implementation of one algorithm on several networks to achieve portable optimality. Section 7 contains a summary and a brief discussion of issues concerning a structured, hierarchical approach to parallel computing.

2. Model of Computation

We study parallelization of a *description of computation.* This description of computation can be in the form of a (sequential, PRAM (cf. Karp & Ramachandran (1988)), etc.) program, a computation dependency graph in which nodes represent inputs, outputs and elementary operations while edges represent data dependency, or a set of mathematical relations describing the computation. For the case study in this paper, the computation dependency graph is acyclic and depends only on the size of the input. The input size N is one of the two asymptotic variables. We assume that the value of an input node, output node or computation node is an indivisible data item, or a *word.* A similar model was used in Papadimitriou & Ullman (1984).

Since a description of computation usually only represents a class of computational schemes to solve a computational problem, the lower bounds obtained do

not necessarily give the intrinsic complexity of solving a computational problem. However, the notion of description of computation is at an appropriate level of abstraction for studying algorithm design on realistic parallel architectures.

An interconnection network is represented by a connected undirected graph in which nodes represent processors while edges represent processor links. The size p of this graph is the second asymptotic variable. We impose the condition that $p \to \infty$ as $N \to \infty$, and that p is small compared to N. In addition, we make the network regularity assumption that as $p \to \infty$ the ratio of the maximal node degree to the minimal node degree in the network graph is bounded above by a constant.

As to the communication capability of a network, we assume that the links have equal transfer rate, that all links to a processor can communicate simultaneously, and that the links are bi-directional, i.e., data can travel in both directions simultaneously. We also assume that the mechanism of communication is *store-and-forward,* also known as *packet-switched.* One can also consider other mechanisms such as wormhole routing (Dally(1987)). Following Stout & Wager (1987), we allow cost-free *batching* of several packets into one packet or *splitting* of one packet into several packets at a processor. However, our results concerning algorithms on meshes do not require this assumption. One can also consider models in which batching and splitting of packets are not allowed (e.g., Valiant (1982), Borodin & Hopcroft (1985), and Bertsekas et. al. (1989)).

We view parallel computation on such a network of processors as consisting of both local computation within processors and transfer of data between processors. Therefore, the time of an algorithm is determined by both computation and communication. In other words, the efficiency of an algorithm on the machine depends on the degree of computation parallelism as well as the degree of communication parallelism.

For measures of time, we assume that an elementary operation takes unit time t_a , and that to send a packet of size m (m words) through a link takes time mt_b to transfer the data where t_b is the unit transfer rate, and additional time t_s to start up and terminate communication. There are therefore three time measures to characterize the performance of an algorithm on an architecture: *parallel time* -- the number of parallel steps (ignoring communication) to execute all the elementary operations, *communication start-up time* -- the number of parallel

communication steps, or start-ups of the network (a start-up is the transfer of one packet along a link), and *communication bandwidth time* -- the sum, over all parallel communication steps, of the size of the largest packet transferred in that step. The three units t_a, t_s and t_b are treated as constants but are considered incomparable. Thus, the three time measures will be estimated individually, even though one of them may dominate asymptotically. This is because these units are determined by different aspects of technology and their ratios vary on different machines, and any one of them can be a significant performance factor for machines of realistic sizes. A consequence of this incomparability assumption is that we cannot consider the benefit of overlapping communication with computation in the same processor. This is justified in an asymptotic analysis because the extra saving in time in the best case of full overlapping would be no more than a factor of $\frac{1}{2}$.

The notion of parallel time has long been a standard measure of parallelism. Variants of the two measures of communication have recently been widely adopted in the parallel computation community (see, e.g., Johnsson & Ho (1987), Saad & Schultz (1986), and Stout & Wager (1987)). Some statistical data were gathered to determine the units t_b and t_s for hypercube machines (e.g., Johnsson & Ho (1987)).

The two measures of communication time, i.e., start-up time and bandwidth time, are architecture-dependent. Without assuming an underlying interconnection network, they cannot be used to measure the 'goodness' of an algorithm. To be able to design and analyze algorithms architecture-independently, we introduce two related architecture-independent measures of communication: (network-independent) *communication latency* of an algorithm -- the number of parallel communication steps, under the assumption that the processors communicate with one another directly; and (network-independent) *communication cost* of an algorithm -- the sum, over all parallel communication steps, of the maximal number of words transferred by any one processor in that step, again under the assumption of direction communication between processors.

These two new communication measures together with parallel time will be used to guide the first level of design -- design of an architecture-independent algorithm.

Note that without an assumption on distribution of input data or on parallel time the communication cost can always be 0 -- just let a single processor do all the computation. Also without an assumption on parallel time the communication latency can always be 1 -- just let the processors exchange all input data first and then compute independently but redundantly. For OMM, we impose the assumption of even distribution of input data (see Theorem 1 for a precise statement of the assumption).

One can also view this network-independent algorithm design as design on a complete-connection network with processor-bound bandwidth. Namely, the two new parameters are the start-up time and the bandwidth time, respectively on a complete-connection network, where the bandwidth time is modified as the sum, over all parallel communication steps, of the maximal number of words transferred by any one processor in that step. At this high level of algorithm design, the assumption of direct processor communication avoids the complication of handling message-forwarding while that of processor-bound bandwidth discourages unrealistic use of the large fan-in and fan-out degrees.

We also adopt the following model of architectural cost. The *architectural cost* of a network is measured by the number of links in the network. Many of the networks we consider have only $\Theta(p)$ links, i.e., are bounded-degree networks under the network regularity assumption. For these networks the number of links no longer distinguishes them. In this case we use the minimal bisection width of the network graph to measure the architectural cost of a network. The minimal bisection width of a graph is the minimal number of edges whose removal separates the graph into two graphs of equal size.

The number of links in a network has traditionally been a measure of cost for multiprocessor design (see, e.g., Stone (1987)). The minimal bisection width is an important factor that determines the area of a chip in VLSI layout (Thompson (1979)), and has also been used in different ways to study multiprocessor organizations (e.g., Dally (1987)).

3. Architecture-Independent Algorithm Design

We illustrate the network-independent algorithm design and analysis with the example of ordinary matrix multiplication $C = AB$, where $A = (a_{ij})_{n\times n}$, $B = (b_{ij})_{n\times n}$ and $C = (c_{ij})_{n\times n}$. A description of computation is as follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad i,j = 1, 2, \cdots, n.$$

Note that the input size is $N = 2n^2$, and there are a total of $2n^3$ elementary operations (additions and multiplications), ignoring lower order terms.

To motivate our approach, we first look at the conventional approach.

For the moment suppose communication is negligible. Achieving optimal parallel time in this idealized case is equivalent to obtaining optimal parallel speed-up. The following simple algorithm achieves this: label the p processors P_1 through P_p and make processor P_l, $1 \leq l \leq p$, responsible for multiplication of the lth block of $\frac{n}{p}$ columns of B with the matrix A. Apparently, computations on different processors are independent and can be done concurrently if the necessary data are in place. This partitioning of computation tasks can thus be scheduled to achieve an optimal parallel time of $\frac{2n^3}{p}$.

Now take communication into account. Suppose the processors are connected linearly as a 1-D mesh P_1 through P_p and then P_1. Here and throughout the paper a 1-D mesh stands for a 1-D ring, i.e., the last processor on the mesh is connected to the first, a 2-D mesh stands for a 2-D torus, and so on for the higher dimensional meshes; the term *ring* is reserved for a 1-D ring in one of the dimensions of a 2-D, 3-D or higher dimensional mesh. On this 1-D mesh network we can assign to P_l as input data, the lth block of $\frac{n}{p}$ columns of B which is used by this processor only, and the lth block of $\frac{n}{p}$ rows of A which all processors need to use. Before computation begins, elements of A can be spread around by having every processor send a packet of its rows of A around the ring once (in the same direction). The network starts up communication p times, each time sending a packet of size $\frac{n^2}{p}$ along each link. This takes communication start-up time p and communication bandwidth time n^2. It can be easily shown that both times are optimal on the 1-D mesh (Corollaries 1 & 2).

In this conventional approach, it is then natural to ask whether one can do better with respect to communication on a more lavish network. Consider the following algorithm: the p processors are labeled (l_1,l_2), $1 \leq l_1,l_2 \leq \sqrt{p}$; processor (l_1,l_2) is responsible for multiplying the l_1th block of $\frac{n}{\sqrt{p}}$ rows of A with the l_2th

block of $\frac{n}{\sqrt{p}}$ columns of B. Initially, processor (l_1,l_2) holds elements of A which lie on the intersection of the l_1th block of $\frac{n}{\sqrt{p}}$ rows with the l_2th block of $\frac{n}{\sqrt{p}}$ columns, and holds elements of B which lie on the intersection of the l_2th block of $\frac{n}{\sqrt{p}}$ rows with the l_1th block of $\frac{n}{\sqrt{p}}$ columns. Again obviously the parallel time is the optimal $\frac{2n^3}{p}$. Now suppose the processors are connected as a 2-D mesh according to the above labeling, i.e., two processors are neighbors if and only if their labels differ in one component (modulo $\sqrt{p}$). For every 1-D ring in the x-dimension or the y-dimension of the 2-D mesh, the $\sqrt{p}$ processors on this ring have to share a block of either $\frac{n}{\sqrt{p}}$ rows of A or $\frac{n}{\sqrt{p}}$ columns of B. Each block is divided into $\sqrt{p}$ number of $\frac{n}{\sqrt{p}}\times\frac{n}{\sqrt{p}}$ submatrices distributed among the $\sqrt{p}$ processors on the ring, and can be sent around the ring just as with the 1-D mesh network. Note that no two such rings share a link, so communication on different rings can be fully concurrent. The communication times on the 2-D mesh is thus the same as on one such ring: $\sqrt{p}$ for start-up time and $\frac{n^2}{\sqrt{p}}$ for bandwidth time, which are better than those for the previous algorithm on the 1-D mesh. Again both communication times are optimal on the 2-D mesh (Corollaries 1 & 2).

So, in this approach one tailors the partitioning of computation tasks to match as well as possible the interconnection network topology in order to optimize communication. In this context, the 1-dimensional partitioning of computation tasks for OMM is often said to be natural for the 1-D mesh architecture while the 2-dimensional partitioning is said to be natural for the 2-D mesh (Johnsson & Ho (1987)).

It is now easier to see what we stated in the introduction about the drawbacks of this approach. Firstly, the algorithm designer has to be familiar with the specifics of the underlying interconnection network and has to work in a programming environment involving these details. Secondly, since one designs a different 'natural' algorithm for a different architecture these algorithms are not easily portable. Thirdly, although one can compare different architectures for parallelizing the same description of computation, analysis and comparison of algorithms are problematic due to lack of a common ground: different algorithms are designed for, and their performance measured on different architectures.

We now proceed to introduce our architecture-independent approach.

First, we separate the notion of an algorithm from the underlying architecture. We call the following the 1-d algorithm. The computation tasks are partitioned according to the above-mentioned one-dimensional partitioning. The schedule of the algorithm consists of two stages. In Stage 1, all input elements of A residing in each processor is made known to every other processor (this is only a functional description of the task, independent of what the network is and how it is done on the network). In Stage 2, every processor executes its share of computation tasks concurrently. We use the term *broadcast* to denote the event of making data in one processor known to every other processor, *complete-broadcast* to denote broadcast from every processor simultaneously, and *limited-complete-broadcast* to denote complete-broadcast within a subset of processors. The 1-d algorithm can then be written in the following pseudo-code using one communication primitive:

Complete–Broadcast (*of the blocks of rows of A to be shared*)

for *all processors do* (*concurrently*)

local computation

In the context of matrix computation Stage 2 can also be written in terms of block matrix multiplication, which does not concern us here.

Similarly, the 2-d algorithm consists of the above-mentioned two-dimensional partitioning and a schedule given by the following pseudo-code:

for *all rows* and *columns of processors in the 2–dimensional labeling do* (*concurrently*)

Limited–Complete–Broadcast (*of the submatrices of A or B to be shared*)

for *all processors do* (*concurrently*)

local computation

Note that the algorithms are now independent of the interconnection topology of the network. The time of an algorithm on whatever network will be the parallel time plus the time to execute the communication primitives on the network. This is the basis for selecting virtual architectures for an algorithm in Section 4 and implementing one algorithm on different networks in Section 6.

Also note that even though the algorithm is given in a somewhat synchronous form its execution need not be. In fact the algorithm is independent of any assumption on the model of communication except that of a distributed-memory.

We now evaluate the two architecture-independent algorithms using our architecture-independent measures of communication. For the 1-d algorithm, communication latency is 1 since only one step of communication suffices to execute complete-broadcast when processors communicate directly; communication cost is n^2 since in this one step of communication the elements of A are exactly what each processor needs to send or receive. Similarly, for the 2-d algorithm communication latency is also 1 while communication cost is $\frac{n^2}{\sqrt{p}}$.

We can now say that the 2-d algorithm is better, in the sense that its communication cost is asymptotically lower. This is a fair comparison because there is no bias introduced by the choice of an architecture.

From this architecture-independent perspective, the following question naturally arises: How much can the communication cost and latency be reduced for OMM? We call the smallest possible communication cost (latency) for a description of computation its *optimal communication cost (latency).* Our hope, as reflected by the thesis we propose, is that if the number of calls on the generic communication primitives is optimized and the amount of data a processor needs to communicate due to data dependency is optimized, then it can be translated into good performance on various networks. Nothing like this has been proved up to this point in the paper. This will eventually be substantiated in Section 6, using developments in this and the next sections.

Theorem 1 Under the assumption that each of the p processors except possibly $o(p)$ of them holds $\Theta(\frac{n^2}{p})$, and no processor holds more than $O(\frac{n^2}{p})$ of the input data exclusively, the optimal communication cost of ordinary matrix multiplication is $\Theta(\frac{n^2}{p^{2/3}})$ and the optimal latency is $\Theta(1)$. Furthermore, there exists an algorithm which simultaneously achieves optimal parallel time, optimal communication cost and optimal latency.

Proof The lower bound $\Omega(\frac{n^3}{p})$ for parallel time is obvious while the lower bound for communication latency (i.e., it is not 0) is a consequence of the nonzero lower bound on communication cost. We sketch a proof for the lower bound on

communication cost. We also give a matching upper bound algorithm.

We construct the following directed acyclic graph (DAG) which models computation dependency for the n^3 multiplications: there are $2n^2$ input nodes representing the elements of A and B; there are n^2 output nodes representing elements of C; there are n^3 nodes in the graph that are neither input nodes nor output nodes, corresponding to the n^3 multiplications $a_{ik}b_{kj}$, $i, j, k = 1, 2, \cdots, n$. Each node $a_{ik}b_{kj}$ has two input nodes a_{ik} and b_{kj} as children and one output node c_{ij} as a parent.

Note that this representation ignores the additions. As a result, the output nodes have fan-in n. To remedy this, we assume that a node with m children ($m > 2$) is equivalent to any binary tree with m-1 nonleaf nodes (for addition) and m leaves which are the original children of this node. This models the fact that we can do additions in arbitrary orders.

To help visualize data dependency, we embed the DAG in the Euclidean space R^3 in the following way: for $1 \leq i, j, k \leq n$, input node a_{ik} is identified with the lattice point $(i,0,k)$, input node b_{kj} is identified with the lattice point $(0,j,k)$, output node c_{ij} is identified with the lattice point $(i,j,0)$, and multiplication node $a_{ik}b_{kj}$ is identified with the lattice point (i,j,k), which by convention is also identified with the $1\times1\times1$ cube centered at the lattice point. All the multiplication nodes thus form a three-dimensional lattice U of size $n\times n\times n$ with the input and output nodes lying on the three coordinate planes.

Now data dependency can be completely characterized as follows: a node (i,j,k) needs the value of $(i,0,k)$ and the value of $(0,j,k)$, and any two nodes (i,j,k_1) and (i,j,k_2) where $k_1 \neq k_2$ both contribute to the value of $(i,j,0)$. If a subset U_l of the multiplication nodes (cubes) is assigned to a processor P_l, then the volume of this set of cubes gives the number of multiplications the processor does. The minimal amount of input data the processor needs is given by the sum of the areas of projections of U_l onto the zx-coordinate plane and the yz-coordinate plane. This minus $O(\frac{n^2}{p})$ -- the amount of input data the processor holds initially -- gives the amount of input data it needs to receive from other processors. The minimal amount of data the processor needs to communicate with others to output C is given by projecting U_l and its complement $U \backslash U_l$ onto the xy-plane and measure the area of their intersection. We thus arrive at the following lemma.

Lemma 1 Let U_l be the subset of cubes assigned to processor P_l. The optimal communication cost of OMM is

$$\Omega(\max_{1\le l\le p}\{\, AREA\{P_{yz}(U_l)\} + AREA\{P_{xz}(U_l)\} + AREA\{P_{xy}(U_l)\bigcap P_{xy}(U\setminus U_l)\}\,\}) \;-\; O(\frac{n^2}{p})$$

where P_{xy}, P_{yz} and P_{xz} are the projection operators of R^3 onto the three coordinate planes, respectively.

The next lemma is a simple fact in elementary geometry. In the context of I/O complexity of OMM, a similar result was proved in Hong & Kung (1981). In the discrete setting, namely that we count the number of lattice points on the coordinate planes which are projected onto, rather than measuring the area, results of this type can also be found in Chung, et. al. (1986).

Lemma 2

$$AREA\{P_{yz}(U_l)\} + AREA\{P_{xz}(U_l)\} + AREA\{P_{xy}(U_l)\} \;=\; \Omega(\,(VOLUME\{U_l\})^{2/3}\,)$$

Since there is at least one processor P_l that does $\frac{n^3}{p}$ multiplications, for this processor $Volume\ (U_l) = \Omega(\frac{n^3}{p})$, which means that the quantity in Lemma 2 is $\Omega(\frac{n^2}{p^{2/3}})$. We then use Lemma 2 to prove that the quantity in Lemma 1 is also $\Omega(\frac{n^2}{p^{2/3}})$. This estimate, given in the following lemma, is proved in the Appendix.

Lemma 3 For any processor P_l that executes $\Omega(\frac{n^3}{p})$ multiplications,

$$AREA\{P_{yz}(U_l)\} + AREA\{P_{xz}(U_l)\} + AREA\{P_{xy}(U_l)\bigcap P_{xy}(U\setminus U_l)\} \;=\; \Omega(\frac{n^2}{p^{2/3}})$$

Lemma 1 and Lemma 3 together yield our lower bound.

Lemma 1 shows why the two algorithms mentioned earlier are not optimal w.r.t. communication cost. They correspond to assigning thin slices of the lattice points to processors and long square-cylinder subsets of the lattice points to the processors, respectively. Neither type of subset has a small area-sum when projected onto the three coordinate planes. To achieve optimality, a subset has to be truly three-dimensional. Thus the lower bound analysis suggests the following partition of computation tasks: partition the $n\times n\times n$ lattice into p subsets of size $\frac{n^3}{p}$, organized three-dimensionally, i.e., into $p^{1/3}\times p^{1/3}\times p^{1/3}$ cubic blocks of lat-

tice points each of dimensions $\frac{n}{p^{1/3}} \times \frac{n}{p^{1/3}} \times \frac{n}{p^{1/3}}$. The lower bound quantity in Lemma 1 for each of these subsets is $\frac{3n^2}{p^{2/3}}$. More specifically, processor (i,j,k), $1 \leq i, j, k \leq p^{1/3}$, is assigned multiplication tasks

$$a_{l_i l_k} b_{l_k l_j}: \quad \frac{(i-1)n}{p} < l_i \leq \frac{in}{p}, \frac{(j-1)n}{p} < l_j \leq \frac{jn}{p}, \frac{(k-1)n}{p} < l_k \leq \frac{kn}{p}$$

Each processor is also responsible for partial sums of the multiplications assigned to it. The summing of the partial-sum values distributed among the processors will be considered part of a communication-oriented primitive called limited-histogramming (see below).

The algorithm has three stages. Stage 1 is again a limited-complete-broadcast to distribute input data, concurrently among processors along each row in the i- or j-direction in the 3-dimensional labeling. Here it is assumed that each of the $\frac{n}{p^{1/3}} \times \frac{n}{p^{1/3}}$ submatrices of A (or B) whose elements are to be shared by an i-row (or j-row) of processors is further divided into $p^{1/3}$ smaller blocks of equal size, distributed among the row of $p^{1/3}$ processors. This stage can be done in one communication step on a complete-connection network. Stage 2 is the concurrent execution of the multiplications and partial sums within each processor. In Stage 3, the $p^{1/3}$ processors along each row in the k-direction in the 3-dimensional labeling have to sum up the partial-sum values for each of the $\frac{n^2}{p^{2/3}}$ output elements they cooperate to compute. Thus Stage 3 is a *limited-histograming* concurrently among each such row of processors. *Histograming* of m numbers by q processors, as defined in Stout & Wager (1987), is the communication-computation task to produce m numbers each of which is summed up from a value in every processor, with the output evenly distributed among the q processors. We used limited-histogramming to denote histogramming among a subset of processors. Stage 3 can also be done in one communication step on the complete-connection network -- a permutation of data among each such row of processors -- plus fully concurrent local computation. In each of these two steps of communication the number of words any processor transfers is $O(\frac{n^2}{p^{2/3}})$. Thus the algorithm has optimal parallel time, optimal communication latency and optimal communication cost. q.e.d.

The following is a pseudo-code for the 3-d algorithm:

for *each i–row* and *j–row of processors do* (*concurrently*)

Limited–Complete–Broadcast (*of the blocks of elements of A or B to be shared*)

for *all processors do* (*concurrently*)

local computation

for *each k–row of processors do* (*concurrently*)

Limited–Histogramming (*of the partial–sum values*)

The 3-dimensional partitioning is also discussed in Johnsson & Ho (1987). When p is a power of 8, it turns out to be just the recursive (divide-and-conquer) block matrix multiplication partitioning.

We point out that when the number of processors p is not very large the advantage of the 3-d algorithm over the 2-d algorithm in terms of communication cost is not necessarily significant in practice.

Other works on architecture-independent analysis include Papadimitriou & Ullman (1984). The analysis in George, et. al. (1987) was also quite architecture-independent although it was intended for the hypercube. In general, architecture-independent analysis has not received enough attention among the parallel computation community.

4. Virtual Architecture Design

In this section we illustrate how to select a collection of virtual architectures to serve as the interface between an algorithm and the hardware networks it will be implemented on. There are two main criteria in choosing the virtual architectures: a) they are good for supporting the generic communication primitives of the algorithm; b) they have a low architectural cost in the case of special-purpose network design, or they have low architectural cost as well as the flexibility to be emulated efficiently on different hardware networks in the case of general-purpose algorithm design.

Since the information flow of the 3-d algorithm is three-dimensional, we naturally set our sight on several 3-D networks. They are the 3-D mesh, the 3-D mesh of trees, the 3-D mesh of hypercubes, and the 3-D mesh of cliques. The 3-D mesh of cliques (complete subgraphs) is obtained by replacing every 1-D ring in

each of the three dimensions of the 3-D mesh with a clique. Similarly, the 3-D mesh of trees is obtained by replacing every ring in each of the three dimensions with a balanced binary tree. Note that this mesh of trees is different from the standard one (Ullman (1984)) where a tree is built with the nodes of an original 1-D ring being the leaf nodes of the tree. The 3-D mesh of hypercubes is obtained by replacing every ring in each of the three dimensions of the mesh with a hypercube. This last turns out to be just the same network as the hypercube (Proposition 4).

A network is called an *ideal architecture* for a description of computation w.r.t. bandwidth time (start-up time, or both) if
i) there is an algorithm that can be implemented on it to achieve optimal parallel time and a bandwidth time (start-up time, or both) asymptotically at least as small as the optimal communication cost (optimal communication latency, or both); and
ii) no network of asymptotically lower architectural cost can achieve i).

We refer to an architecture simply as an ideal architecture if it is ideal w.r.t. both communication time measures.

Note that the start-up time of any algorithm on any network cannot be lower than the optimal communication latency, but the bandwidth time of an algorithm may be lower than the optimal communication cost on a network of unbounded node-degrees without the assumption of processor-bound bandwidth.

Proposition 1 The 3-D mesh is ideal for OMM w.r.t. bandwidth time.

Proof The same technique used for implementing the 1-d and 2-d algorithms on the 1-D and 2-D meshes, which we discussed in Section 3, is now used to implement the limited-complete-broadcast primitive of the 3-d algorithm on the 3-D mesh, concurrent among every 1-D ring in the i- and j-dimensions. This achieves start-up time $O(p^{1/3})$ and bandwidth time $O(\frac{n^2}{p^{2/3}})$. The following technique is used to implement the limited-histogramming primitive among every 1-D ring in the k-dimension: every processor in such a ring organizes the $O(\frac{n^2}{p^{2/3}})$ partial-sum values it holds after local computation stage into $p^{1/3}$ packets of size $O(\frac{n^2}{p})$, each destined for a different processor on the ring (including one for itself); to start the procedure, every processor sends to its neighbor in the forward

direction of the ring the packet destined for its neighbor in the backward direction of the ring; afterwards, at each step every processor on the ring receives a packet from the neighbor behind, adds the values to those in its own packet with the same destination, and forwards it to the neighbor in front; the process ends after each processor receives the packet destined for it and adds the values to its own. This again takes start-up time $O(p^{1/3})$ and bandwidth time $O(\frac{n^2}{p^{2/3}})$. So, condition i) for an ideal architecture is satisfied w.r.t. bandwidth time. For condition ii), note that the network has the smallest possible number of links $\Theta(p)$ so the minimal bisection width is the measure of architectural cost here. To show that no network of smaller minimal bisection width can achieve i) , we use the following argument due to Thompson (1979). It is shown in Fisher (1987) for general matrix multiplication, and can be easily seen in our proof of Theorem 1 for ordinary matrix multiplication, that between any two halves of a network each of which holds $\Theta(n^2)$ input data exclusively, the number of words that have to be communicated between them is $\Theta(n^2)$. For a network with minimal bisection width $\omega(p)$, at least one of the links has to transfer $\Omega(\frac{n^2}{\omega(p)})$ words. Therefore, to have a bandwidth time of $O(\frac{n^2}{p^{2/3}})$ requires a network to have a minimal bisection width of $\Omega(p^{2/3})$ which is the minimal bisection width of the 3-D mesh. q.e.d.

Note that no batching or splitting of messages is used in the above and earlier implementations of the 1-d, 2-d and 3-d algorithms on the 1-D, 2-D and 3-D meshes, respectively.

As a consequence of Thompson's argument used in the proof we also have the following.

Corollary 1 For k = 1, 2 or 3, The bandwidth time of the k-d algorithm implemented on the k-D mesh without the use of batching or splitting of messages is optimal for the network.

Note that the start-up time $O(p^{1/3})$ of the 3-d algorithm on the 3-D mesh is however far larger that the optimal communication latency $\Theta(1)$ of OMM.

Proposition 2 The graph of any network that can support our 3-d algorithm to achieve start-up time 2 contains the 3-D mesh of cliques as a subgraph. More generally, an ideal architecture for OMM must have $\Omega(n^{1+\epsilon})$ links for a positive constant ϵ.

Proof Apparently, the 3-D mesh of cliques achieves the desired start-up time because every subset of processors among which limited-complete-broadcast or limited-histogramming has to be executed is connected as a clique. To achieve a start-up time of 2 for the 3-d algorithm requires each of the two communication primitives be executed in one communication step. Each of these primitives defines communication-oriented tasks among subsets of processors which have the property that the outputs in every processor depend on the inputs held by every processor involved in the same task. To execute such a task in one communication step requires a complete connection among the processors involved.

For the second statement of the proposition, generalization of an argument for general matrix multiplication due to Gentleman (1978) shows that at least one piece of information concerning one of the input elements has to traverse a path of at least $\frac{1}{4}$ of the length of the diameter of the network graph (intermediate processing may happen along the way). Thus the start-up time on the network is at least $\frac{1}{4}$ of the diameter of the graph. To achieve a constant start-up time means that the diameter of the graph must be constant. The well-known Moore bound (cf. Chung (1987)) states that for a graph to achieve a diameter D its maximal degree K must satisfy $K = \Omega(p^{1/D})$. Thus the maximal degree of the network graph is at least $\Omega(p^{1/D})$. By our network regularity assumption, every node has at least this degree and the number of links in the network is $\Omega(p^{1+1/D})$. q.e.d.

As a consequence of the generalization of Gentleman's argument used in the proof we have the following.

Corollary 2 For k = 1, 2, or 3, the start-up time of the k-d algorithm implemented on the k-D mesh without the use of batching or splitting of messages is optimal for the network.

Bounded-degree networks have only $\Theta(p)$ links, and the hypercube has only $\Theta(p \log p)$ links. A large hardware network with more than $O(p\ polylog(p))$ ($polylog(p)$ stands for any polynomial of $\log p$) links is considered unrealistic given the state-of-the-art of technology. Even as a virtual architecture a network with so many links is not convenient to emulate on a sparse network. Proposition 2 therefore shows that the notion of ideal architecture can be too stringent because of the requirement i). The requirement ii) can also be too stringent. For example, for

OMM with respect to bandwidth time it rules out any network of unbounded degrees and thus of higher architectural cost, on which one may achieve a bandwidth time asymptotically lower than the optimal communication cost by utilizing the large number of links per node. These considerations therefore motivate the following weaker notion.

A network is called a *natural architecture* for a description of computation w.r.t. bandwidth time (start-up time, or both) if
i') there is an algorithm that can be implemented on it to achieve optimal parallel time and bandwidth time (start-up time, or both) to
within a $polylog(p)$ factor of the optimal communication latency (optimal communication cost, or both); and
ii') any network with lower architectural cost cannot achieve as low a start-up time (bandwidth time, or both) as this network can.

Again it is simply called a natural architecture if it is natural w.r.t. both communication times.

We point out that this particular definition of natural architecture is somewhat model-sensitive in the sense that if cost-free batching and splitting of messages are not permitted in the model of communication then the requirement i') may still be too stringent.

Theorem 2 The 3-D mesh of trees and the 3-D mesh of hypercubes are both natural architectures for OMM.

Proof We first verify conditions i') and ii') for the 3-D mesh of trees. For this network we first show that a simple implementation of the 3-d algorithm satisfies i') but not ii'), and then briefly describe an improved implementation that satisfies both i') and ii'). Consider the following simple implementation. To implement the limited-complete-broadcast primitive, simply let the packets to be shared by the $p^{1/3}$ processors in a balanced binary tree be sent from the leaf nodes up, batched up along the way to reach the root node, and then sent down as a single packet to every processor. Bandwidth time is $O(\frac{n^2 \log p}{p^{2/3}})$ with the sending-down dominating. For the limited-histogramming primitive, summing-up from the leaf nodes of a balanced binary tree to the root node does not require batching, and to have an even distribution of output data if needed requires packets to be sent from the root down and split into halves at each of the $\log p^{1/3}$ steps. Here

bandwidth time is again $O(\frac{n^2 \log p}{p^{2/3}})$ with the summing-up dominating. Start-up time for the two stages is $O(\log p^{1/3})$. The extra parallel time for additions during summing-up is $O(\frac{n^2 \log p}{p^{2/3}})$. Under our assumption $p \leq 2n^2$ this is a lower order term compared with the optimal parallel time $\frac{2n^3}{p}$. So, condition i') is satisfied. Now consider condition ii'). The network has $O(p)$ links and the same minimal bisection width as the 3-D mesh. Yet the bandwidth time for our simple implementation is a factor of $\log p$ larger than that on the 3-D mesh which is optimal given the minimal bisection width (Proposition 1). This is due to the fact that at any one communication step only links at one level of a tree are busy. To remove this $\log p$ factor, i.e., to obtain an implementation which satisfies i') as well as ii'), we introduce pipelining at all levels of a tree to modify the above simple implementation. The main idea is to let packets at all levels of a tree start simultaneously moving up to the root and then down to the leaves, and let batching, splitting and summing-up be done at different levels of the tree at the same time. We omit the details here. With this modification, the parallel time and start-up time are unchanged while the bandwidth time can be reduced to the optimal $O(\frac{n^2}{p^{2/3}})$.

For the 3-D mesh of hypercubes, condition i') is easily seen to be satisfied since the network graph contains the 3-D mesh of trees as a subgraph. Consider condition ii'). Implementation of the 3-d algorithm on the 3-D mesh of hypercubes will be deferred to Theorem 3 in Section 6. There it will be shown that the bandwidth time of the 3-d algorithm on the 3-D mesh of hypercubes is $O(\frac{n^2}{p^{2/3} \log p})$. Now take any network that can do as well as the 3-D mesh of hypercubes and we show that it must have $\Omega(p \log p)$ links, the number that the 3-D mesh of hypercubes has. Since optimal parallel time has to be achieved by condition i), no processor can execute more than $O(\frac{n^3}{p})$ multiplications. This means there are $\Theta(p)$ processors each of which executes $\Theta(\frac{n^3}{p})$ multiplications. By Lemma 3, every of these processors must communicate $\Omega(\frac{n^2}{p^{2/3}})$ words. To achieve the same bandwidth time as on the 3-D mesh of hypercubes every of these processors must have $\Omega(\log p)$ links, a total of $\Omega(p \log p)$ links for the network. q.e.d.

As a consequence of the proof we also have the following.

Corollary 3 The 3-D mesh of trees is an ideal architecture for OMM w.r.t. bandwidth time.

These three networks, the 3-D mesh, the 3-D mesh of trees and the 3-D mesh of hypercubes form our collection of virtual architectures for the 3-d algorithm. Again, to achieve modularity at this level implementations of limited-complete-broadcast and limited-histogramming can be organized around message-passing primitives between neighbors in a virtual architecture. We do not attempt to give the pseudo-codes here.

5. Special-Purpose Computing: Network Design

Consider the following situation: We are given a large number of processors to be used for parallelization of a description of computation which has a very large number of computation tasks; we want to design both an algorithm and a network topology to connect the processors together in such a way that high performance can be achieved with low network cost.

Assumption: The total bandwidth of a processor -- the number of words that can be transferred per time unit by the processor -- is bounded above by a constant. In other words, a processor can have a small number of large links or a large number of small links, but the total number of words that can be pumped into or out of the processor per time unit is fixed.

This is a reasonable assumption since the total energy a given chip can produce per time unit is bounded. Note that under this assumption, the bandwidth time of any algorithm on any network cannot be lower than the optimal communication cost of a description of computation since unbounded fan-in or fan-out at a processor does not reduce bandwidth time. Because of this, if the processor are connected into a dense network and yet many of the links are not used for the particular application, then a good portion of the network bandwidth is wasted. In such a case, it will be better-off to connect the processors into a sparser network with larger links. This is the motivation for the following notion.

A network is called an *optimal architecture* for a description of computation w.r.t. bandwidth time (start-up time, or both) if
i) there is an algorithm that can be implemented on this architecture to achieve optimal parallel time and a bandwidth time (start-up time, or both)

asymptotically as low as can be achieved on any network; and
ii) no network of lower architectural cost can achieve i).

With the above assumption, the notion of ideal architecture translates immediately into that of optimal architecture.

Proposition 3 Under the above assumption, an ideal architecture for a description of computation w.r.t. bandwidth time (start-up time, or both) is an optimal architecture w.r.t. bandwidth time (start-up time, or both).

Corollary 4 the 3-D mesh and the 3-D mesh of trees are optimal architectures w.r.t. bandwidth time for OMM.

Conclusions with practical implications can now be drawn from the above. For example, if the start-up time unit t_s is not too large compared with the bandwidth time unit t_b, and $p \ll n^2$, then $\frac{n^2}{p^{2/3}} t_b \gg p^{1/3} t_s$, which means that the 3-D mesh is very close to optimal with respect to total communication. In this case, a 3-D mesh is an economical way to organize the processors to achieve good performance. For the 3-D mesh of trees, this is true even when t_s is larger or when $p \approx n^2$; however, one must bear in mind that efficient mechanism for packet batching and splitting is necessary to achieve this.

6. General-Purpose Computing: Emulation of Virtual Architectures

In contrast to the situation in Section 5 we are now given not just a number of processors but a network of processors with a fixed topology. Now processor-bound bandwidth is not assumed. The links are already there, and so it makes sense to try to utilize as many of them as possible. In this case, unbounded fan-in or fan-out may reduce the bandwidth time to lower than the optimal communication cost of a description of computation.

We already designed a network-independent algorithm in Section 3, and selected a collection of virtual architectures in Section 4. We now want to implement the algorithm on this given architecture by picking a suitable virtual architecture from the collection and finding a good emulation of it on the network. Now we need to utilize properties of the network.

In this section we consider four networks: 1-D mesh, 2-D mesh, 3-D mesh, and the hypercube. More specifically, we show that the 3-d algorithm for OMM can be implemented on the 1-D mesh, the 2-D mesh, and the hypercube to

achieve simultaneously optimal parallel time, optimal start-up time and optimal bandwidth time on each of them. Note that optimality of the 3-d algorithm on the 3-D mesh was already established in Section 4 (Corollaries 1 & 2).

We present the implementation and analysis first for the hypercube and then for the meshes.

We first review the standard reflected Grey code representation of the hypercube: the p (p a power of two) hypercube nodes are represented by the set of all binary strings of length $\log p$; there is a link between two nodes if and only if their strings differ in exactly one bit. When $p = 2^{3d}$ for some positive integer d, we can decompose the binary string into three segments of equal length. Then the same Grey code defines a sub-hypercube whenever values of two of the three substrings are fixed. This gives a 3-D mesh of hypercube. Note that in this coding all the links of the hypercube are also links of the 3-D mesh of hypercubes. We thus have the following.

Proposition 4 The 3-D mesh of hypercubes is a hypercube.

The following theorem in particular establishes the communication optimality of a computational algorithm for a description of computation on the hypercube. Previous results concerning communication on the hypercube have mostly been about communication problems or graph-embedding problems, not computational problems. Although our theorem is only for a description of computation rather than a computational problem, we hope that it is a step closer to the spirit of the traditional theory of algorithms and complexity. From the proof of the theorem one can see the importance of the role an architecture-independent analysis plays in establishing a result of this type.

Theorem 3 Let $p = 2^{3d}$, $d > 0$. The 3-d algorithm can be implemented on the hypercube to achieve optimal parallel time $\frac{2n^3}{p}$, optimal start-up time $\Theta(\log p)$, and optimal bandwidth time $\Theta(\frac{n^2}{p^{2/3}\log p})$.

Proof The particular implementation that achieves optimality is obtained by picking the 3-D mesh of hypercubes as the virtual architecture and emulating it on the hypercube. Note that this virtual architecture happens to be the same as the hypercube so no work is needed for emulation. Recall that in the proof of Theorem 2 we deferred to present the implementation of the 3-d algorithm on the 3-D mesh of hypercubes. So, the following implementation and proof of

optimality should actually be viewed as for the 3-D mesh of hypercubes. Optimality on the hypercube then follows since the two networks are identical.

We prove the lower bounds first. By Theorem 1, at least one processor has to transfer $\Omega(\frac{n^2}{p^{2/3}})$ words for its share of computation regardless of the network topology. These data have to be transferred through at most $\log p$ incident links. Thus the lower bound $\Omega(\frac{n^2}{p^{2/3}\log p})$ holds for bandwidth time. The lower bound $\Omega(\log p)$ on start-up time follows from a generalization of the argument in Gentleman (1978) mentioned in the proof of Proposition 2 and the fact that the diameter of the hypercube is $\log p$.

We now establish the upper bounds. Think of the network as the 3-D mesh of hypercubes. Stage 1 of the 3-d algorithm, limited complete broadcast, can now be carried out by concurrently calling the hypercube complete broadcast procedure due to Stout & Wager (1987) in every subcube connecting an i-row or j-row of $p^{1/3}$ processors. This takes start-up time $O(\log p^{1/3}) = O(\log p)$ and bandwidth time $O(\frac{mp^{1/3}}{\log p^{1/3}})$ where m is the size of a packet (see Stout & Wager (1987), where all packest are assumed to be of the same size m and m is large). Since the size of a packet is $\Theta(\frac{n^2}{p})$ (for convenience we assume the $o(p)$ processors that may not hold this amount of data send out packets padded to make up the proper size), the bandwidth time is $O(\frac{n^2}{p^{2/3}\log p})$.

Stage 3 of the 3-d algorithm, the limited-histogramming stage, can be carried out concurrently in every subcube connecting a k-row of $p^{1/3}$ processors by calling the histogramming procedure due to Stout & Wager (1987), which yields the same upper bounds. However, arithmetic computation (additions during summing-up) was assumed to be of no cost in their paper. A careful examination of their procedure shows that the arithmetic computation is indeed fully parallel. q.e.d.

In the next two implementations of the 3-d algorithm on the 1-D mesh and 2-D mesh, we use the 3-D mesh as the virtual architecture.

Theorem 4 The 3-d algorithm can be implemented on the 1-D mesh and the 2-D mesh, respectively without the use of batching or splitting of messages, to yield simultaneous optimal parallel time, optimal start-up time and optimal

bandwidth time on these networks.

Theorem 4 is a consequence of the next lemma which is more general and is proved in the Appendix. Consider the following situation: There are a number of tokens distributed among the p processors in a network. *A step of token movement* is a transformation of the initial configuration of the tokens to another configuration by moving some of these tokens from the processors they reside in to the neighboring processors, with the restriction that no more than one token is allowed to cross a link in each direction.

The lemma concerns the number of steps of token movement necessary on one mesh to emulate an arbitrary step of token movement on another mesh. Most previous results on emulation between interconnection networks have used the product of dilation and congestion (cf. Bhatt, et. al. (1986)) as an upper bound for the number of emulation steps. For the meshes we consider, it can be shown that no embedding can produce such an upper bound good enough for our purpose. We thus have to go down to a finer level of analysis of the embeddings in the proof of the lemma in the Appendix. This shows the limitation of only considering dilation and congestion in studying emulation between interconnection networks.

Lemma 4 Let $m > q$ be positive integers, and $\log p$ be divisible by the least common multiple of m and q. The m-D mesh can be embedded in the q-D mesh in such a way that one step of token movement on the former can be emulated by an optimal $\Theta(p^{(1/q-1/m)})$ steps of token movement on the latter.

Proof of Theorem 4 Our implementation of the 3-d algorithm on the 3-D mesh takes start-up time $O(p^{1/3})$, i.e., $O(p^{1/3})$ steps of packet movements, and bandwidth time $O(\frac{n^2}{p^{2/3}})$ because the size of each packet is $O(\frac{n^2}{p})$ (Proposition 1). By Lemma 4, these same packet movements can be done in $O(p)$ steps on the 1-D mesh and $O(\sqrt{p})$ steps on the 2-D mesh. Therefore the algorithm achieves a start-up time of $O(p)$ and a bandwidth time of $O(n^2)$ on the 1-D mesh, and it achieves a start-up time of $O(\sqrt{p})$ and a bandwidth time of $O(\frac{n^2}{\sqrt{p}})$ on the 2-D mesh. All are optimal for the respective networks (Corollary 1 & 2). It is not hard to see that such an emulation does not decrease the computation parallelism either in the local computation stage or during the limited-histogramming stage when communication and computation are interleaved. No batching or splitting

of messages is needed since neither is used in the emulation or in the earlier implementation of the the 3-d algorithm on the 3-D mesh. q.e.d.

The emulations in this section provides the basis for implementing message-passing primitives between neighbors of a virtual architecture in terms of message-passing primitives on the physical architectures. Again, we will not give the pseudo-codes here.

7. Towards Structured Parallel Computing

In this section we summarize our ideas and discuss briefly issues concerning a structured, hierarchical approach to parallel computing.

In this paper, we propose a framework of hierarchical design and analysis for parallel algorithms on distributed-memory architectures. In our theory, there are three levels of design (and analysis): architecture-independent algorithm design, virtual architecture design, and design of emulations of virtual architectures on hardware architectures.

At the level of architecture-independent algorithm design, we adopt an environment which allows processors to communicate directly and use parallel (computation) time, the number of communication rounds and data interdependency as the complexity measures. We also require, in an algorithm, separation of local computation tasks from communication-oriented tasks and organization of the latter around a set of generic primitives.

It might be feared that allowing direct communication between processors may lead to an algorithm whose implementation on a hardware architecture would suffer from excessive traffic. We have shown that this does not have to be the case. Firstly, minimization of data dependency forces computation tasks to be partitioned in a way to preserve locality. Secondly, instead of emulating the unrestricted environment of direct processor communication on a hardware architecture, we use an intermediate level of design to select a collection of virtual architectures to implement the algorithm on. Each of these virtual architectures is tailored to meet the communication need of the algorithm in a minimal way, and the whole collection offers flexibility to match a wide spectrum of potential hardware architectures.

At the third level of design, for each hardware architecture among a wide spectrum, we properly choose a virtual architecture from the collection and emulate it optimally on the hardware architecture.

A complete analysis shows that for the case study of ordinary matrix multiplication, the above methodology leads to a network-independent algorithm which can be implemented on a wide spectrum of architectures to achieve optimality on each of them, thus achieving portable optimality. The technical results for the two lower levels of design are of greater generality and can be used for designing algorithms for other applications.

From a theoretical standpoint, our framework offers a common ground for network-independent analysis and comparison of parallel algorithms. From a practical standpoint, it has the advantage that the high level algorithm can be developed in an environment that does not involve details of any possible underlying architecture and is portable across different architectures. Moreover, when a set of generic communication-oriented primitives is identified to support a wide class of applications, it will provide incentive for establishing software support to handle the two lower levels of design, making them transparent to the algorithm designer.

The search for scientific methodologies to achieve both uniformity and efficiency for computing is a major theme of computer science. Its success has been demonstrated, from structured programming in the 1960's, to the multilayer network architectures for data communications in the 1980's. The most relevant comparison here is perhaps the design of (sequential) algorithms and data structures. The separation of the notion of an algorithm from specific machines and programming languages, and the introduction of the notion of abstract data structure as a layer between the algorithm and the machine, have proven to be instrumental to the design of efficient algorithms that were able to utilize the vast computing power offered by the modern (serial) computers to tackle complex problems (Tarjan (1987)). In a sense, this is a motivation for our approach. If one views the virtual architectures in our theory as global (parallel) data structures (as opposed to local data structures within a processor) then the generic communication-oriented primitives around which an algorithm is organized are just data structure manipulation primitives. What we advocate for parallel computing on distributed-memory architectures is the separation of the

notion of a parallel algorithm from the underlying architectures and the use of parallel data structures as an interface between an algorithm and the underlying architectures. It is conceivable that this methodology may lead to powerful parallel data structures that support other environments for algorithm design than what we study in this paper.

Due to its tremendous complexity, parallel computing has not become a practical means of computing despite the availability of parallel machines and the amount of research efforts directed to the area. We hope that a more structured approach to algorithm design will be a step in the right direction towards making parallel computing a practical reality.

8. Appendix

Lemma 3 For any one processor P_l for which $VOLUME\{U_l\} = \Omega(\frac{n^3}{p})$,

$$AREA\{P_{yz}(U_l)\} + AREA\{P_{xz}(U_l)\} + AREA\{P_{xy}(U_l) \bigcap P_{xy}(U \setminus U_l)\} = \Omega(\frac{n^2}{p^{2/3}}) \quad (A.1)$$

Proof Let P_l be any processor for which $VOLUME(U_l) = \Omega(\frac{n^3}{p})$. If

$$AREA\{P_{yz}(U_l)\} + AREA\{P_{xz}(U_l)\} = \Omega(\frac{n^2}{p^{2/3}})$$

we are done. So suppose

$$AREA\{P_{yz}(U_l)\} + AREA\{P_{xz}(U_l)\} = o(\frac{n^2}{p^{2/3}}) \quad (A.2)$$

Then

$$AREA\{P_{xy}(U_l)\} = \Omega(\frac{n^2}{p^{2/3}}) \quad (A.3)$$

by Lemma 2. This implies that

$$AREA\{P_{xy}(U_l) \bigcap P_{xy}(U \setminus U_l)\} \} = \Omega(\frac{n^2}{p^{2/3}})$$

for otherwise (A.3) and

$$AREA\{P_{xy}(U_l) \bigcap P_{xy}(U \setminus U_l)\} \} = o(\frac{n^2}{p^{2/3}})$$

imply

$$AREA\{P_{yz}(U_l)\} + AREA\{P_{xz}(U_l)\} = \Omega(\, n\,(\sqrt{\frac{n^2}{p^{2/3}}})\,) \;=\; \Omega(\frac{n^2}{p^{1/3}})$$

contradicting (A.2). Thus Lemma 3 holds. q.e.d.

Lemma 4 Let $m > q$ be positive integers, and $\log p$ be divisible by the least common multiple of m and q. The m-D mesh can be embedded in the q-D mesh in such a way that one step of token movement on the former can be emulated by an optimal $\Theta(p^{(1/q-1/m)})$ steps of token movement on the latter.

Proof Here we only give the proof for the cases $m = 3$, $q = 1$ and $m = 3$, $q = 2$ which are needed for proving Theorem 4. The general proof is just a generalization of the proof for these cases and will only be hinted at at the end.

The embedding of a mesh in another can be determined by specifying how every 1-D ring in each of the dimensions of the guest mesh is embedded. For this purpose, we first review the standard binary representation of natural numbers. A set of positive integers $\{0, 1, 2, \cdots, L\text{–}1\}$ can be represented by the set of all binary strings of length $\log L$ in which 0 is represented by $(0, 0, \cdots, 0)$ and the representation of k is obtained by taking the binary string which represents k-1 and flipping the least-significant (rightmost) 0-bit to 1 and all bits to its right from 1 to 0. When these integers are used to label consecutive processors on a ring of size L, we say that this flipping of bits in the binary string represents a *forward legal move* -- crossing of a link in the forward direction of the ring. Similarly, taking the binary string and flipping the rightmost 1-bit to 0 and all bits to its right from 0 to 1 represents a *backward legal move* -- crossing of a link in the backward direction of the ring.

A) Emulation of 3-D mesh token movement on 1-D mesh:

Let $\log p = 3d$, where d is a positive integer. The 1-D mesh of size p will be coded as above by the set of binary strings of length $\log p$: $\{\, (c_{3d}\ c_{3d-1}, \cdots, c_1)\,\}$. The 3-D mesh of size p will be coded by the same set of binary strings $\{\, (z_d, \cdots, z_1, y_d, \cdots, y_1, x_d \cdots, x_1)\,\}$. Here each of the three substrings of length d codes rings in each of the three dimensions of the 3-D mesh, also using the binary representation of natural numbers. The nodes of the 3-D mesh will be mapped to the nodes of the 1-D mesh under the identity mapping between the two identical sets of binary strings. For embedding of the edges, we specify for each legal move on the 3-D mesh the series of legal moves on the 1-D mesh which emulate it. Without loss of generality we shall consider only forward legal moves on the 3-D

mesh. We shall see that the maximal length of any such emulating series is $p^{2/3}$. We then show that all series emulating legal moves along a single direction of the 3-D mesh can be executed fully concurrently, and so the emulation of one step of token movement on the 3-D mesh can be done in less than $6p^{2/3}$ steps on the 1-D mesh.

First consider any forward legal move on the 3-D mesh along the x-direction. Suppose the move is

$$(z'_d \cdots, z'_1, y'_d \cdots, y'_1, x'_d \cdots, x'_1) \rightarrow (z'_d \cdots, z'_1, y'_d \cdots, y'_1, x''_d \cdots, x''_1)$$

and $(x'_d \cdots, x'_1) \neq (1, \cdots, 1)$. Then the flipping of bits which represent the forward legal move on the 3-D mesh also represents a forward legal move in the 1-D mesh. Now suppose $(x'_d \cdots, x'_1) = (1, \cdots, 1)$. Then the flipping of bits which represents the forward legal move on the 3-D mesh no longer represents a legal move on the 1-D mesh because for such a move to be legal on the latter would require also flipping some of the more significant bits beyond the range $x_d \cdots, x_1$. However, we can still emulate this move by a series of $p^{1/3}$ backward legal moves on the 1-D mesh, equivalent to moving backwards along the ring in the x-direction on the 3-D mesh:

$$\begin{aligned}
&(z'_d \cdots, z'_1, y'_d \cdots, y'_1, 1, \cdots, 1, 1) \\
\rightarrow\ &(z'_d \cdots, z'_1, y'_d \cdots, y'_1, 1, \cdots, 1, 0) \\
\rightarrow \cdots \rightarrow\ &(z'_d \cdots, z'_1, y'_d \cdots, y'_1, 0, \cdots, 0, 0)
\end{aligned}$$

Next consider any forward legal move on the 3-D mesh along the y-direction. Suppose the move is

$$(z'_d \cdots, z'_1, y'_d \cdots, y'_1, x'_d \cdots, x'_1) \rightarrow (z'_d \cdots, z'_1, y''_d \cdots, y''_1, x'_d \cdots, x'_1)$$

and $(y'_d \cdots, y'_1) \neq (1, \cdots, 1)$. Then the forward legal move on the 3-D mesh is no longer legal on the 1-D mesh because for such a move to be legal on the latter would require also the sub-string $(x'_d \cdots, x'_1)$ be equal to $(1, \cdots, 1)$ and be flipped to $(0, \cdots, 0)$. However, we can emulate it by a series of $p^{1/3}$ forward legal moves on the 1-D mesh, flipping through all possible values of the sub-string $(x_d \cdots, x_1)$: first move forward all the way from $(z'_d \cdots, z'_1, y'_d \cdots, y'_1, x'_d \cdots, x'_1)$ to $(z'_d \cdots, z'_1, y'_d \cdots, y'_1, 1, \cdots, 1)$; then make the legal move

$$(z'_d \cdots, z'_1, y'_d \cdots, y'_1, 1, \cdots, 1) \rightarrow (z'_d \cdots, z'_1, y''_d \cdots, y''_1, 0, \cdots, 0)$$

and finally move forward all the way from $(z'_{d} \cdots, z'_1, y''_{d} \cdots, y''_1, 0, \cdots, 0)$ to $(z'_{d} \cdots, z'_1, y''_{d} \cdots, y''_1, x'_{d} \cdots, x'_1)$. Now suppose $(y'_{d} \cdots, y'_1) = (1, \cdots, 1)$. We emulate this forward legal move by moving backward on the 1-D mesh all the way from $(z'_{d} \cdots, z'_1, 1, \cdots, 1, x'_{d} \cdots, x'_1)$ to $(z'_{d} \cdots, z'_1, 0, \cdots, 0, x'_{d} \cdots, x'_1)$. This requires flipping through almost all possible values of the sub-string $(y_{d}, \cdots, y_1, x_{d} \cdots, x_1)$, a total of $p^{2/3}-p^{1/3}$ backward moves.

In exactly the same way as in the case of a forward legal move along the y-direction when $(y'_{d} \cdots, y'_1) \neq (1, \cdots, 1)$, any forward legal move along the z-direction:

$$(z'_{d} \cdots, z'_1, y'_{d} \cdots, y'_1, x'_{d} \cdots, x'_1) \rightarrow (z''_{d} \cdots, z''_1, y'_{d} \cdots, y'_1, x'_{d} \cdots, x'_1)$$

including when $(z'_{d} \cdots, z'_1) = (1, \cdots, 1)$, can be emulated by a series of $p^{2/3}$ forward legal moves on the 1-D mesh by flipping through all possible values of the sub-string $(y_{d} \cdots, y_1, x_{d}, \cdots, x_1)$.

In the standard terminology of graph embedding (see, e.g., Bhatt, et. al. (1986)), what we have shown is that the dilation of the embedding -- the longest 1-D mesh path a 3-D mesh edge is embedded in -- is $p^{2/3}$. One can show that the congestion of the embedding -- the maximal number of 3-D mesh edges any 1-D mesh edge supports -- is also $p^{2/3}$. Most previous studies use the product of the two parameters as an upper bound for the number of emulation steps. However, this would not be good enough for our purpose. In fact, it can be shown that no embedding can offer such a product good enough for our purpose. Therefore, we have to go down to a finer level of analysis to show that congestion does not play a crucial role in delaying token movement in this embedding. The key lies in the observation that a legal move on the 3-D mesh is emulated by a series of forward legal moves only or backward moves only on the 1-D mesh, i.e., a series of emulating moves do not reverse direction. For an arbitrary step of token movement on the 3-D mesh consider all tokens which are to be moved in a single direction. When emulated on the 1-D mesh each will start at a different node and they will flow orderly along either forward or backward direction of the 1-D mesh for no longer than $p^{2/3}$ steps, never running into or overtaking one another. Emulation of a step of token movement on the 3-D mesh can therefore be done in $6p^{2/3}$ steps of token movement on the 1-D mesh, by emulating the six directions of the 3-D mesh one by one. The bound is only $2p^{2/3}+2p^{1/3}+2$ when one keeps track of the details.

B) Emulation of 3-D mesh token movement on the 2-D mesh:

Let $p = 2^{6d}$ for some positive integer d. The 2-D mesh of size p is coded, in the standard binary representation of natural numbers, by the set of binary strings of length $\log p$: $\{ (b_{3d} \cdots, b_1, a_{3d} \cdots, a_1) \}$, where the two substrings code rings in the a- and b-dimensions of the 2-D mesh, respectively. The 3-D mesh of the same size is coded in the same binary representation of natural numbers by the same set of binary strings $\{ (y_{2d} \cdots, y_1, z_{2d} \cdots, z_{d+1}, x_{2d} \cdots, x_1, z_d \cdots, z_1) \}$, where the two substrings $(x_{2d} \cdots, x_1)$, $(y_{2d} \cdots, y_1)$ code rings in the x- and y-dimensions of the 3-D mesh, respectively, and $(z_{2d} \cdots, z_1)$, the concatenation of $(z_{2d} \cdots, z_{d+1})$ with $(z_d \cdots, z_1)$, codes rings in the z-dimension of the 3-D mesh. Again the nodes of the 3-D mesh are mapped to the nodes of the 2-D mesh under the identity mapping. We specify embedding of the edges by describing for each legal move on the 3-D mesh the emulating series of legal moves on the 2-D mesh. Again without loss of generality only emulation of forward legal moves will be given. We will show that the maximal length of any series emulating a move along the x- or y-direction is $p^{1/6}$, and the maximal length of any series emulating a move along the z-direction is $2p^{1/6}$. We will also show that all series emulating legal moves (both backward and forward) along x- and y-directions of the 3-D mesh can be executed fully concurrently, and all those emulating legal moves (both backward and forward) along the z-direction can be executed fully concurrently. This leads to emulation of one step of token movement on the 3-D mesh by $3p^{1/6}$ steps on the 1-D mesh.

First consider forward legal moves along x- and y-directions. This is analogous to emulating legal moves along the z-direction of the 3-D mesh on the 1-D mesh. A ring in the x-dimension of the 3-D mesh is embedded in a single ring in the a-dimension of the 2-D mesh and is coded by the $2d$ most significant bits of the sub-string $(a_{3d} \cdots, a_1)$. Any forward legal move along the x-direction on the 3-D mesh can thus be emulated by a series of $p^{1/6}$ forward legal moves in the a-direction of the 2-D mesh by flipping through all possible values of the sub-string $(a_d \cdots, a_1)$. Therefore, each ring in the x-dimension of the 3-D mesh is evenly stretched along the forward direction to fit into a ring in the a-dimension of the 2-D mesh, and each ring in the a-dimension of the 2-D mesh supports $p^{1/6}$ rings in the x-dimension of the 3-D mesh. By the same argument as in emulating the 3-D mesh on the 1-D mesh, all series emulating legal moves along the x-direction can

be executed fully concurrently. Thus one step of token movement consisting of moves (forward and backward) along the x-direction can be emulated in $p^{1/6}$ steps on the 2-D mesh. By symmetry, one step of token movement consisting of moves (forward and backward) along the y-direction of the 3-D mesh can be emulated by $p^{1/6}$ steps of moves along the b-direction on the 2-D mesh. Since 1-D rings in different dimensions of the 2-D mesh are edge-disjoint, these two $p^{1/6}$ steps of moves can be executed at the same time.

Now consider forward legal moves in the z-direction of the 3-D mesh.

First consider the case where the move consists of flipping bits only among $z_d \cdots, z_1$. Since these bits are the same as the d least significant bits $a_d \cdots, a_1$ that code rings in the a-dimension of the 2-D mesh, the move is still a forward legal move along the a-direction of the 2-D mesh. Apparently, two different such moves on the 3-D mesh remain different moves on the 2-D mesh. A step of token movement on the 3-D mesh consisting of only such moves can thus be made in one step on the 2-D mesh.

Now consider the case where the move requires flipping also bits among $z_{2d} \cdots, z_{d+1}$. These bits are the same as the least significant bits $b_d \cdots, b_1$ that code rings in the b-dimension of the 2-D mesh. In this case $(z_d \cdots, z_1)$ must be $(1, \cdots, 1)$ and be flipped to $(0, \cdots, 0)$. Let the move be

$$(y'_{2d} \cdots, y'_1, z'_{2d} \cdots, z'_{d+1}, x'_{2d} \cdots, x'_1, 1, \cdots, 1)$$

$$\rightarrow (y'_{2d} \cdots, y'_1, z''_{2d} \cdots, z''_{d+1}, x'_{2d} \cdots, x'_1, 0, \cdots, 0)$$

First suppose $(z'_{2d} \cdots, z'_{d+1}) \neq (1, \cdots, 1)$. Then the move can be emulated on the 2-D mesh by one forward legal move along the b-direction plus a series of $p^{1/6}-1$ backward legal moves along the a-direction: first make one move along the b-direction to $(y'_{2d} \cdots, y'_1, z''_{2d} \cdots, z''_{d+1}, x'_{2d} \cdots, x'_1, 1, \cdots, 1)$, and then move backwards along the a-direction all the way from

$$(y'_{2d} \cdots, y'_1, z''_{2d} \cdots, z''_{d+1}, x'_{2d} \cdots, x'_1, 1, \cdots, 1)$$

to

$$(y'_{2d} \cdots, y'_1, z''_{2d} \cdots, z''_{d+1}, x'_{2d} \cdots, x'_1, 0, \cdots, 0).$$

Note that two different such moves on the 3-D mesh are emulated by two different series of moves on the 2-D mesh which do not share edges, so a step of token movement on the 3-D mesh consisting of such moves only can be emulated

in $p^{1/6}$ steps on the 2-D mesh. Now suppose $(x'_{2d} \cdots, x'_{d+1}) = (1, \cdots, 1)$. Then the move

$$(y'_{2d} \cdots, y'_1, 1, \cdots, 1, x'_{2d} \cdots, x'_1, 1, \cdots, 1)$$

$$\rightarrow (y'_{2d} \cdots, y'_1, 0, \cdots, 0, x'_{2d} \cdots, x'_1, 0, \cdots, 0)$$

can be emulated by a series of $p^{1/6}$–1 backward moves along the b-direction plus a series of $p^{1/6}$–1 backward moves along the a-direction. First move backwards along the b-direction all the way to $(y'_{2d} \cdots, y'_1, 0, \cdots, 0, x'_{2d} \cdots, x'_1, 1, \cdots, 1)$, and then move backwards along the a-direction all the way to $(y'_{2d} \cdots, y'_1, 0, \cdots, 0, x'_{2d} \cdots, x'_1, 0, \cdots, 0)$. Again two different such moves do not contend for edges when emulated and a step of token movement consisting of only such moves on the 3-D mesh can be done in $2p^{1/6}$ steps on the 2-D mesh. By careful examination we can fully overlap execution of all emulating series in all these cases for forward legal moves along the z-direction of the 3-D mesh. We observe that only the first two emulating steps on the 2-D mesh involve forward legal moves. So, by symmetry, backward legal moves along the z-direction of the 3-d mesh can be emulated essentially concurrently. A step of token movement consisting of moves along the z-direction of the 3-D mesh can therefore be emulated by $2p^{1/6}$ steps of token movement on the 2-D mesh.

The total number of steps of token movement on the 2-D mesh to emulate one step of token movement on the 3-D mesh is therefore $3p^{1/6}$.

Proof for the general case of embedding the m-D mesh in the q-D mesh is just a generalization of the above proof. A binary string of length $\log p$ is divided into q consecutive sub-strings of length $\frac{\log p}{q}$ in coding the q-D mesh while these sub-strings are further divided into segments to code the m-D mesh as in A) or in B). We omit the details.

Optimality of the bounds $O(p^{(1/q-1/m)})$ follows from the fact that the minimal bisection widths of the m-D and q-D meshes are $\Theta(p^{\frac{m-1}{m}})$ and $\Theta(p^{\frac{q-1}{q}})$, respectively, and that their ratio $\Theta(p^{1/q-1/m})$ is a lower bound on the worst-case number of m-D mesh edges a q-D mesh edge has to support in any embedding. q.e.d.

Acknowledgements I would like to thank David Kirkpatrick for valuable discussions on relevant theoretical issues as well as on a draft of this paper. I would like to thank Maria Klawe and Nick Pippenger for valuable comments and for criticism on a draft of this paper. I would also like to thank Alan Wagner for very helpful suggestions and for pointers to the literature. Discussions with Uri Ascher, Zhaojun Bai, Geng Lin, Jim Little and Jim Varah were also very helpful. I appreciate the valuable comments from Lenore Blum, William Kahan, Richard Karp, Beresford Parlett, and Steve Smale during the preliminary stage of this research at the University of California, Berkeley.

References

Bertsekas, D. P., Ozveren, C., Stamoulis, G. D., Tseng, P, and Tsitsiklis, J. N. (1989), Optimal communication algorithms for hypercubes. LIDS-P-1847, Laboratory for Information and Decision Systems, M.I.T.

Bhatt, S., Chung, F., Leighton, T., and Rosenberg, A. (1986), Optimal simulation of tree machines. *Proc. 27-th IEEE Annual Symp. on Foundations of Computer Science,* 274-282.

Borodin, A, and Hopcroft, J. E. (1985), Routing, merging, and sorting on parallel models of computation. *J. Computer and System Sciences,* **30,** 130-145.

Chung, F. R. K. (1987), Diameters of graphs: old problems and new results. *Proc. 18-th Southeastern Conference on Combinatorics, Graph Theory, and Computing, Congressus Numerantium,* Vol. 60, 295-317.

Chung, F. R. K., Graham, R. L., Frankl, P., and Shearer J. B. (1986), Some intersection theorems for ordered sets and graphs. *J. Combinatorial Theory,* Series A **43,** 23-37.

Dally, W. J. (1987), Wire-efficient VLSI multiprocessor communication networks. *Proc. 1987 Stanford Conference on Advanced Research in VLSI,* MIT Press, Cambridge, Massachusetts, 391-415.

Fisher, D. C. (1987), Communication complexity of matrix multiplication. preprint.

Gentleman, W. M. (1978), Some complexity results for matrix computations. *J. ACM,* Vol. 25, No. 1, 112-115.

George, J. A., Liu, W-H, and Ng, E. G-Y (1987), Communication results for parallel sparse Cholesky factorization on a hypercube. Technical Report CS-87-03, York University.

Hong, J-W. and Kung, H. T. (1981), I/O complexity: the red-blue pebble game. *Proc. 19-th ACM Annual Symp. on Theory of Computing,* 326-333.

Johnsson, S. L. and Ho, C-T (1987), Matrix multiplication on Boolean cubes using generic communication primitives. YALEU/DCS/TR-530, Department of Computer Science, Yale University.

Karp, R. M., and Ramachandran, V. (1988), A survey of parallel algorithms for shared-memory machines. Report No. UCB/CSD 88/408, Computer Science Division, University of California, Berkeley.

Papadimitriou, C. H., and Ullman, J. D. (1984), A communication-time tradeoff. *Proc. IEEE 25-th Annual Symp. on Foundations of Computer Science,* 84-88.

Saad, Y. and Schultz, M. H. (1986), Data communication in parallel architectures. YALEU/DCS/RR-461, Department of Computer Science, Yale University.

Stone, H. S. (1987), *High-Performance Computer Architecture.* Addison-Wesley, Reading, Massachusetts.

Stout, Q. F. and Wager, B. (1987), Intensive hypercube communication I: prearranged communication in link-bound machines. CRL-TR-9-87, Computing Research Laboratory, University of Michigan.

Tarjan, R. E. (1987), Algorithm Design. *CACM,* Vol. 30, No. 3, 205-212.

Thompson, C. D. (1979), Area-time complexity for VLSI. *Proc. ACM 11-th Annual Symp. on Theory of Computing,* 81-88.

Ullman, J. D. (1984), *Computational Aspects of VLSI.* Computer Science Press, Rockville, Maryland.

Valiant, L. G. (1982), A scheme for fast parallel communication. *SIAM J. Computing,* Vol. 11, No. 2, 350-361.