Architecture-Independent Parallel Algorithm Design for Distributed-Memory Architectures*

Feng Gao

Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada V6T 1W5

This paper introduces an architecture-independent, hierarchical approach to algorithm design on 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 efficiency. In our framework, there are three levels of algorithm design: design of a network-independent algorithm in a network-independent programming environment, design of virtual networks (virtual architectures) for the algorithm, and design of emulations of the virtual networks on physical networks. In its organizational principle, this methodology is analogous to the abstract data structure approach to sequential algorithm design. We propose 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 networks, can be implemented on a wide spectrum of physical networks to achieve optimality on each of them with respect to both computation and communication. We illustrate this thesis with an analysis of the algorithm design for ordinary matrix multiplication. In a paper by Gao, a general theory of portable optimality of parallel algorithms is presented. Besides its implications to the methodology of parallel algorithm design, our framework also suggests new questions for theoretical research in parallel computation on interconnection networks.

1. INTRODUCTION

In this paper, we introduce a framework of algorithm design and analysis for parallel computing on distributed-memory architectures. The main goal of introducing this framework 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 serious 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 it will lie as much in portability of algorithms as in their efficiency.

Hypothetically, architecture independence of parallel algorithms may be achieved through either automatic detection of parallelism or emulation of a general high level parallel programming environment such as the PRAM (cf. Karp and Ramachandran [18]) on realistic architectures. However, due to their tremendous complexity it is unlikely that such unrestricted mechanisms could be handled efficiently by automated systems. We contend that a more realistic approach in this direction should be the development of a restricted machine environment and a structured algorithmic paradigm in much the same fashion as the (abstract) data structure programming environment and algorithmic paradigm for sequential computing. We defer more discussions on this issue to Section 7.

As a first step in this direction towards architecture independence of parallel algorithms, we propose a framework of hierarchical algorithm design which emphasizes the following principles:

(A) abstraction of the notion of algorithm from its implementation on a machine;
(B) separation of issues of computation parallelism from issues of data communication; and
(C) discrimination between the data communication requirement of an algorithm due to computation dependency and actual data routing in an interconnection network.

More specifically, in our framework, the design, analysis, and implementation of a parallel algorithm for a given
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 partitioning 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 small collection of virtual architectures (virtual networks) 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 physical networks. More precisely, they are selected on the basis of their capability to support the communication-oriented primitive procedure calls in the algorithm as well as their flexibility for efficient emulation on different physical networks. The third level is the implementation of this architecture-independent algorithm on a wide spectrum of interconnection networks, by emulating a properly chosen virtual network on each physical 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.

This organizational principle for handling data communication in parallel algorithm design is analogous to the abstract data structure approach for handling data organization in sequential algorithm design. Its goal is to permit the algorithm designer to develop efficient algorithms in an environment free of details of actual data routing, in a way similar to what the abstract data structure approach achieves in detaching sequential algorithm design from details of actual data access.

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 counterintuitive that one algorithm could run well on several different networks.

To show that the answer is, however, affirmative, we propose the following thesis: architecture-independent optimality of an algorithm can lead to portable optimality of the algorithm. Namely, a single algorithm, obtained via architecture-independent optimization, with the support of properly chosen virtual networks, can be implemented on a wide spectrum of architectures to achieve optimality on each of them. In this paper, this thesis is illustrated by an analysis of the example of designing a coarse- or medium-grain parallel algorithm for ordinary matrix multiplication (OMM). In a forthcoming paper Gao [13], we will present a general theory that uses graph-theoretic concepts to capture and synthesize the generality of this thesis and will apply the methodology to more examples.

In addition, we show that this approach allows one to develop not only portable but structured programs: when local computation is distinguished from communication-oriented tasks organized around primitive procedure calls, one gains in program modularity.

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-oriented stages is asymptotically optimal in an architecture-independent sense, i.e., simultaneously achieving optimal computation parallelism, minimizing the number of rounds of communication, and minimizing the amount of data communication due to computation dependency. Each communication-oriented stage uses one procedure call on a generic communication-oriented primitive. The two generic primitives used are limited-complete-broadcast and limited-histogramming (Section 3).

(b) A collection of virtual networks, the 3D mesh, the 3D meshes of s-D meshes (s > 1) and the 3D mesh of hypercubes, are selected on which to implement this 3D algorithm (Section 4).

(c) Through emulation of these virtual networks, this 3D algorithm can be implemented on the 1D, 2D, and 3D meshes, the 3s-D meshes (s > 1) 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. The 3D algorithm is also implemented on the (3s - r)-D meshes (s > 1, r = 1, 2) through emulation of virtual networks, but optimality of these implementations has not been established (Section 5).

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

Besides the process of algorithm design, our results have implications for theoretical research in parallel computation. For instance, Theorem 3 (Section 5) in particular establishes the communication optimality of the 3D 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 general matrix multiplication, it is in the spirit of traditional complexity theory 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 5) which makes use of embeddings of the 3D mesh in the 1D and 2D meshes. The notion of portable optimality motivates 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 (Proposition 1, Section 6).

The rest of the paper is organized as follows: Section 2 presents the model of computation. Section 3 presents the high level architecture-independent algorithm design and analysis. Section 4 concerns design of virtual networks. Section 5 concerns implementations of one algorithm on different networks through emulation of virtual networks. Section 6 contains proofs for some of the technical lemmas. And Section 7 contains conclusions and a brief discussion of issues concerning a structured approach to parallel computing.

2. MODEL OF COMPUTATION

We study parallel processing of a description of computation. This description of computation can be in the form of a (sequential, PRAM, etc.) program, a computation dependency graph in which nodes represent inputs, outputs, and elementary operations and edges represent data dependencies, or a set of mathematical relations describing the computation. For the case study in this paper, the computation dependency graph is acyclic, and its size depends only on the input size (number of input nodes in the graph). The input size $N$ is one of the two asymptotic variables. We assume that the value of an input, output, or computation node is an indivisible data item, or a word. A similar model was used in Papadimitriou and Ullman [22].

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 and edges represent processor links. The size $p$ of this graph is the second asymptotic variable. We impose the condition that $p \to \infty$ as $N \to \infty$. We assume in this paper that $p$ is small compared to $N$. More precisely, when we present results for an algorithm we will specify a function $f(N)$ such that the results are valid whenever $p \leq f(N)$ (or $p = O(f(N))$).

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 bidirectional; 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 [10] or adaptive cut-through [21].

We view parallel computation on a network of processors as consisting of both local computation within processors and data communication between processors. Therefore, the time of an algorithm is determined by both computation and communication. In other words, the efficiency of an algorithm on a 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 $t_c$, and that to send or receive a message (packet) of size $m$ ($m$ words) by a processor through a link takes time $mt_c + t_g$ to transfer the data where $t_c$ is the unit of communication bandwidth time and $t_g$ is the unit of communication start-up time. Time is charged accordingly when a processor is idle waiting for data. Thus, there are three time measures to characterize the performance of an algorithm on an architecture: parallel (computation) time—the number of parallel steps (ignoring communication) to execute all the elementary operations, communication start-up time—the maximum of the communication start-up times of all the processors, and communication bandwidth time—the maximum of communication bandwidth times of all the processors. The three units $t_c, t_g$, and $t_a$ are constants, but in this paper we assume they are incomparable; i.e., we treat them as completely unrelated parameters that cannot be compared. We adopt this assumption at a theoretical level because these measures are determined by different aspects of technology and their ratios vary on different machines; any one of them can be a significant performance factor for machines of realistic sizes. In particular, the computation time and the two communication times will be estimated separately.

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 setting because the extra saving in time in the best case of full overlapping would be no more than a factor of $\frac{1}{4}$.

In addition, following Stout and Wager [28], batching of several packets into one packet or splitting of one packet into several packets in a processor is permitted and is assumed to be cost-free. As pointed out in Stout and Wager [28], this can lead to a reduction in the start-up time of a communication scheme. But the bandwidth time is generally not sensitive to this assumption. Whenever a result does not depend on this assumption it will be stated explicitly.
The notion of parallel time has long been a standard measure of parallelism. Variants of our measures of communication have in the past few years been widely adopted by researches in the hypercube computation community (see, e.g., [17, 25, 28]). Some statistical data were gathered to determine the units $t_c$ and $t_p$ for hypercube machines (e.g., [17]). They can also be viewed as a generalization of the message complexity measure used in theoretical analysis of routing in interconnection networks (e.g., [33, 8, 24]).

The two measures of communication time, i.e., start-up time and bandwidth time, are network-dependent. Without assuming the existence of 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 also need architecture-independent measures of communication. They are defined by assuming direct processor communication; i.e., that there is a complete network connecting the processors. We introduce the following two measures of communication time: (network-independent) communication latency of an algorithm—the maximum, over all the processors, of the number of communication start-ups; and (network-independent) communication cost of an algorithm—the maximum, over all the processors, of the communication cost of a processor, where the communication cost of a processor is the total number of different words transferred by the processor over all start-ups. As with the architecture-dependent communication measures, time is charged accordingly when a processor is idle waiting for data.

At a high level of algorithm design, this model avoids the details of data routing and allows communication to be characterized only by the number of rounds of communication and the amount of data a processor communicates due to computation dependency. These two new communication time measures together with parallel (computation) time will be used to guide the first level of design—design of an architecture-independent algorithm.

Note that the start-up time of an algorithm on any network cannot be lower than its communication latency. However, the bandwidth time of an algorithm on an unbounded-degree network may be lower than its communication cost. This is because the different words transferred by a processor in the definition of the latter may be fanned in or out through an unbounded number of links.

Also note that without an assumption on distribution of input data or on parallel time the communication cost can always be zero—just let a single processor do all the computation. Also without an assumption on parallel time the communication latency can always be one—just let the processors exchange all the input data first and then compute independently but redundantly. Our approach is the following: we first make the assumption of even distribution of input data and look for an algorithm that simultaneously achieves optimal parallel time, optimal communication cost, and optimal communication latency under this assumption; when this is not possible, we impose the assumption of optimal parallel time and look for an algorithm that simultaneously achieves optimal communication cost and optimal communication latency under this assumption. For OMM, the assumption of even distribution of input data will suffice (see Theorem 1, Section 3 for a precise statement of the assumption). Examples in which the trade-offs between the architecture-independent time measures are more complex and for which even the second assumption does not suffice to pose an optimal algorithm, will be discussed in Gao [13].

3. ARCHITECTURE-INDEPENDENT ALGORITHM DESIGN AND ANALYSIS

We illustrate our architecture-independent algorithm design and analysis with the example of designing an algorithm for ordinary matrix multiplication $C = AB$, where $A = (a_{ij})_{n \times m}$, $B = (b_{ij})_{m \times n}$, and $C = (c_{ij})_{n \times n}$. A description of computation is

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

Note that the input size is $N = 2n^2$ and that 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. In the conventional approach, one partitions and schedules the computation tasks according to the specifics of the architecture.

First, consider the 1D mesh network of processors connected in the order $P_1, P_2, ..., P_m$, with $P_r$ also connected to $P_1$ (in this paper the term “1D mesh” stands for a 1D ring, “2D mesh” stands for a 2D torus, etc.; the term ring is reserved for a 1D ring in one of the dimensions of a higher dimensional mesh (torus)). The following one-dimensional partitioning of tasks is often said to be natural for this network (e.g., [17]): assume $p \leq n$; let processor $P_t$, $1 \leq t \leq p$, be responsible for multiplication of the columns $m, (l-1)(n/p) < m \leq l(n/p)$, of $B$ with the matrix $A$. The rows $m, (l-1)(n/p) < m \leq l(n/p)$, of $A$ and the columns $m, (l-1)(n/p) < m \leq l(n/p)$, of $B$ are assigned to processor $P_r$ as input data. 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 $2n^3/p$. The processors communicate for data before computation begins. Each processor sends a packet...
of its rows of A onto the ring. These packets move in lock-step around the ring once. The network starts up communication \( p - 1 \) times, each time transmitting a packet of size \( n^2/p \) along each link. This takes communication start-up time \( O(p) \) and communication bandwidth time \( O(n^2) \). It can be easily shown that both times are optimal for the 1D mesh (Corollary 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. Suppose one has a 2D mesh, with the \( p \) processors connected using the 2D labeling \( P_{i,j} \), \( 1 \leq i_1, i_2 \leq \sqrt{p} \); namely, two processors are neighbors if and only if their labels differ by \( 1 \pmod{\sqrt{p}} \) in exactly one of the indices. Consider the following two-dimensional partitioning of computation tasks which is considered natural for the 2D mesh [17]: assume \( p \leq n^2 \); processor \( P_{i,j} \) is responsible for multiplying the rows \( m \), \( (i_1 - 1) \sqrt{p} < m \leq i_1 \sqrt{p} \), of \( A \) with the columns \( m \), \( (i_2 - 1) \sqrt{p} < m \leq i_2 \sqrt{p} \), of \( B \). Initially, processor \( P_{i,j} \) holds elements

\[
a_{ij} ; \quad (i_1 - 1) \frac{n}{\sqrt{p}} < i \leq i_1 \frac{n}{\sqrt{p}} \quad \text{and} \quad (i_2 - 1) \frac{n}{\sqrt{p}} < j \leq i_2 \frac{n}{\sqrt{p}}.
\]

Again computation can be scheduled to achieve the optimal parallel time \( 2n^2/p \). Communicationwise, for every 1D ring in the \( i \)-dimension or the \( j \)-dimension of the 2D mesh, the \( \sqrt{p} \) processors on this ring have to share a block of either \( n/\sqrt{p} \) rows of \( A \) or \( n/\sqrt{p} \) columns of \( B \). Each block is divided into \( \sqrt{p} \) number of \( n/\sqrt{p} \times n/\sqrt{p} \) submatrices, distributed among the \( \sqrt{p} \) processors on the ring. They can be sent around the ring just as with the one-dimensional partitioning on the 1D 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 2D mesh is thus the same as on one such ring: \( O(\sqrt{p}) \) for start-up time and \( O(n^2/\sqrt{p}) \) for bandwidth time. Both communication times are optimal for the 2D mesh (Corollary 2). These times are better than those for the one-dimensional partitioning on the 1D mesh.

It is now easier to see the drawbacks of this conventional approach. First, 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. Second, one designs a different "natural" algorithm for a different architecture these algorithms are not easily portable. Third, although one can compare different architectures for parallelizing the same description of computation, analysis and comparison of algorithms are problematic due to the lack of a common ground: different algorithms are designed for and their performances are measured on different architectures. Efficiency of algorithms is achieved at the expense of uniformity and machine independence of algorithms.

We now introduce our architecture-independent approach. First, we separate the notion of algorithm from the underlying architecture. We call the following the 1D-algorithm. The computation tasks are partitioned according to the previously mentioned one-dimensional partitioning. The schedule of the algorithm consists of two stages. In Stage 1, input elements of \( A \) residing in each processor are made available to every other processor (this is only a functional description of the task, independent of what the network is and how it is executed 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 sending data in one processor 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 1D algorithm can then be written in the following pseudo-code using one communication procedure call:

\[
\text{COMPLETE-BROADCAST (of the elements of } A \text{ to be shared)}
\]

\[
\text{for all processors do (concurrently)}
\]

\[
\text{local computation}
\]

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

\[
\text{for all rows and columns of processors in the 2D labeling do (concurrently)}
\]

\[
\text{LIMITED-COMPLETE-BROADCAST (of the rows of } A \text{ and columns of } B \text{ to be shared)}
\]

\[
\text{for all processors do (concurrently)}
\]

\[
\text{local computation}
\]

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 computation time plus the time to execute the communication procedures on the network.

Also note that even though the algorithm is given in a somewhat synchronous form its execution need not be. In fact the algorithm at this high level 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 1D algorithm, communication latency is \( O(1) \), since only one step of communication suffices to execute complete-broadcast when processors communicate directly; communication cost is \( O(n^2) \), since
in one step of communication the elements of \( A \) are exactly what each processor sends or receives. Similarly, for the 2D algorithm communication latency is also \( O(1) \) while communication cost is \( O(n^2/\sqrt{p}) \).

We can now say that the 2D algorithm is better, in the sense that its communication cost is asymptotically lower. That is, the amount of data communication due to computation dependency is smaller. This is a fair comparison because there is no bias introduced by the choice of a network.

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 \textit{optimal communication cost} (latency). Our hope, as reflected by the thesis of portable optimality we propose, is that if the number of rounds of communication is optimized and the amount of data a processor communicates due to computation dependency is optimized, then it can be translated into good performance on a wide spectrum of physical networks. This will eventually be substantiated in Section 5, using developments in this and the next sections.

\textbf{Theorem 1.} Assume \( p \leq n^3 \) and that each of the \( p \) processors holds \( O(n^2/p) \) of the input data exclusively. The \textit{optimal communication cost} of ordinary matrix multiplication is \( \Theta(n^2/p^3) \) and the \textit{optimal latency} is \( \Theta(1) \). Furthermore, there exists an algorithm which simultaneously achieves optimal parallel time, optimal communication cost, and optimal communication latency.

\textbf{Proof.} The lower bound \( \Omega(n^3/p) \) for parallel time is obvious. We establish a lower bound on communication cost by estimating the minimum amount of data that the worst-case processor must communicate due to computation dependency. The lower bound on communication latency (i.e., it is not zero) is a consequence of the non-zero 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 \( 2n^3 \) input nodes each representing an element of \( A \) or \( B \); there are \( n^2 \) output nodes each representing an element of \( C \); there are \( n^3 \) nodes in the graph that are neither input nodes nor output nodes, representing the \( n^3 \) multiplications \( a_{ik} b_{kj} \), \( i, j, k = 1, 2, \ldots, n \). Each node \( a_{ik} b_{kj} \) has two input nodes \( a_{ik} \) and \( b_{kj} \) as children and one output node \( c_{ij} \) as 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 (representing the additions) 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 \( \mathbb{R}^3 \) in the following way: for \( 1 \leq i, j, k \leq n \), input node \( a_{ik} \) is identified with the \( 1 \times 1 \) square on the \( xz \)-coordinate plane centered at the lattice point \( (i, 0, k) \); input node \( b_{kj} \) is identified with the \( 1 \times 1 \) square on the \( yz \)-coordinate plane centered at the lattice point \( (0, j, k) \); and output node \( c_{ij} \) is identified with the \( 1 \times 1 \) square on the \( xy \)-coordinate plane centered at the lattice point \( (i, j, 0) \); multiplication node \( a_{ik} b_{kj} \) is identified with the \( 1 \times 1 \times 1 \) cube centered at the lattice point \( (i, j, k) \). 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 \( U_i \) of the multiplication nodes (cubes) is assigned to a processor \( P_j \), then the volume of this set of cubes gives the number of multiplications the processor executes. The minimal amount of input data the processor needs is given by the sum of the areas of projections of \( U_i \) onto the \( xz \)-coordinate plane and the \( yz \)-coordinate plane. This minus \( O(n^2/p) \)—the amount of input data the processor holds initially—gives the amount of input data it needs to receive from the other processors to execute its share of multiplications. The minimal amount of data the processor needs to communicate with others to output \( C \) is given by projecting \( U_i \) and its complement \( U - U_i \) onto the \( xy \)-plane and measure the area of their intersection. We thus arrive at the following lemma.

\textbf{Lemma 1.} Let \( U_i \) be the subset of cubes assigned to processor \( P_j \). The \textit{optimal communication cost} of OMM is

\[
\Omega \left( \max_{1 \leq i \leq p} \{ \text{AREA}\{ P_{xy}(U_i) \} + \text{AREA}\{ P_{xz}(U_i) \} \} \right) - O(n^2/p),
\]

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

In the context of I/O complexity of OMM, the next lemma was given in [16]. It was also used in Aggarwal et al. [1] to derive a lower bound for OMM on a shared-memory PRAM, where the inputs \( A \), \( B \) before computation and the outputs \( C \) after computation are assumed to be in the shared memory. In that model, the first two terms in Lemma 2 represent the cost of moving input data from the shared memory to the local memory while the third term represents the cost of moving the output data to the shared memory. Our assumption is somewhat weaker since we do not have any requirement on the distribution of outputs \( C \). As a result, the lower bound in Lemma 2 does not translate
automatically to a lower bound for our quantity in Lemma 1. But it can be used to obtain the lower bound we desire (Lemma 3).

**Lemma 2.** For any subset \( U_i \) of \( U \),

\[
\text{AREA} \{ P_{\times}(U_i) \} + \text{AREA} \{ P_{\cdot}(U_i) \} + \text{AREA} \{ P_{\cdot\cdot}(U_i) \} = \Omega((\text{VOLUME}(U_i))^{2/3}).
\]

Since there is at least one processor \( P \), that does \( n^3/p \) multiplications, \( \text{VOLUME}(U_i) = \Omega(n^2/p) \) for this processor, which means that for \( P \), the quantity theory in Lemma 2 is \( \Omega(n^2/p^{2/3}) \). We then use Lemma 2 to prove that the quantity in Lemma 1 is also \( \Omega(n^2/p^{2/3}) \). This estimate, given in the following lemma, is proved in Section 6.

**Lemma 3.** For any processor \( P \), that executes \( \Omega(n^3/p) \) multiplications,

\[
\text{AREA} \{ P_{\times}(U_i) \} + \text{AREA} \{ P_{\cdot}(U_i) \} + \text{AREA} \{ P_{\cdot\cdot}(U_i) \} = \Omega(n^2/p^{2/3}).
\]

Lemma 1 and Lemma 3 together yield our lower bound on communication cost. 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 the 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 partitioning of computation tasks: partition the \( n \times n \times n \) lattice \( U \) into \( p \) subsets of size \( n^2/p \), organized three-dimensionally, i.e., into \( p^{1/3} \times p^{1/3} \times p^{1/3} \) cubic blocks of lattice points, each of dimensions \( n/p^{1/3} \times n/p^{1/3} \times n/p^{1/3} \). The lower bound quantity in Lemma 1 for each of these subsets is \( 3n^2/p^{2/3} \). More specifically, processor \( P_{\text{blk}i} \), \( 1 \leq i, j, k \leq p^{1/3} \), is assigned multiplication tasks

\[
(a_{ij} b_{ij}) (i-1)n \leq j \leq i n/p^{1/3}, \quad (j-1)n/p^{1/3} \leq k \leq jn/p^{1/3}, \quad (k-1)n \leq l \leq kn/p^{1/3}.
\]

Each processor is also responsible for computing partial sums of the multiplications assigned to it. The summing of the partial-sum values distributed among the processors will be part of a generic communication-oriented primitive called limited-histogramming (see below). For input data distribution, divide each of the \( n/p^{1/3} \times n/p^{1/3} \) submatrices of \( A \); \( b_{h,k} : (i-1)n/p^{1/3} \leq i \leq (i-1)n/p^{1/3} \), \( (j-1)n/p^{1/3} \leq k \leq (j-1)n/p^{1/3} \), \( (k-1)n \leq l \leq kn/p^{1/3} \), which are to be shared by a \( j \)-row (or an \( i \)-row) of processors in the three-dimensional labeling \{ \( P_{\text{blk}} \) \}, into \( p^{1/3} \) smaller subsets of equal size, and distribute them among the \( j \)-row (or \( i \)-row) of \( p^{1/3} \) processors.

The algorithm has three stages. Stage 1 is again a limited-complete-broadcast to move data in place, concurrently among processors along each \( i \)- or \( j \)-row in the three-dimensional labeling of the 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 three-dimensional labeling have to sum up the partial-sum values for each of the \( n^2/p^{2/3} \) output elements they cooperate to compute. Thus Stage 3 is a limited-histogramming concurrently among such row of processors. Histogramming of \( m \) numbers by \( q \) processors, as defined in Stout and Wager [28], 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 subsets of partial-sum values among each such row of processors—plus fully concurrent local computation (since \( p \leq n^2 \), the \( n^2/p^{2/3} \) elements of \( C \) to be output by a \( k \)-row of \( p^{1/3} \) processors can be divided into \( p^{1/3} \) subsets each to be output by one processor). In each of these two steps of communication the total size of different packets that any one processor transfers is \( O(n^2/p^{2/3}) \). Thus the algorithm achieves optimal parallel time, optimal communication latency, and optimal communication cost.

Q.E.D.

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

for each \( i \)-row and \( j \)-row of processors do (concurrently)

LIMITED-COMPLETE-BROADCAST (of the subsets of elements of \( A \) or \( B \) to be shared)

for all processors do (concurrently)

multiplications and partial-sums

for each \( k \)-row of processors do (concurrently)

LIMITED-HISTOGRAMMING (of the partial-sum values)

The three-dimensional partitioning is also discussed in [17]. When \( p \) is a power of 8, it turns out to be just the 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 3D algorithm over the 2D algorithm in terms of communication cost is not necessarily significant.

Other works on architecture-independent analysis include Papadimitriou and Ullman [22] and Papadimitriou and Yannakakis [23]. The analysis in George et al. [15] is also
4. VIRTUAL NETWORK DESIGN

In this section we illustrate the design of virtual networks to serve as the interface between an algorithm and a wide spectrum of physical networks that it is implemented on. There are two main criteria in choosing the virtual architecture: (a) they offer good support for the generic communication primitives of the algorithm; and (b) they have the flexibility to be emulated efficiently on different physical networks.

The flow of data in an algorithm is characterized by its (abstract) data communication structure. In this paper, the data communication structure of an algorithm is defined as a hypergraph (cf. [4]): it consists of a set of $p$ nodes each representing a processor and a family of subsets of nodes each representing a relation among the nodes in the subset; a subset of nodes is in the family if and only if the subset is the domain of a communication-oriented primitive procedure call. Here we call such a subset a hyperedge of the hypergraph.

For our 3D algorithm, the hyperedges in the data communication structure are all the $i$, $j$, and $k$-rows of $p^{1/3}$ processors in the 3D labeling of processors.

Our somewhat heuristic approach to virtual network design is as follows: A virtual network will basically be a virtual network realization of the data communication structure, derived by specifying an interconnection pattern among the nodes in each hyperedge of the data communication structure. It has the property that there is a good interconnection among the nodes in every hyperedge in order to execute the corresponding communication-oriented primitive procedure call efficiently. What the interconnection patterns are depends on the nature of the primitives as well as on the characteristics of the physical network(s) on which the virtual network will be emulated. Through these virtual networks we achieve uniformity at the algorithm level: a single algorithm with the associated data communication structure is implemented on many different physical networks. We also would like to achieve uniformity at the virtual network level. Namely, we would like, if possible, to use a small number of virtual networks to achieve portable optimality on many different physical networks.

In Gao [13], complexity-theoretic characterizations of good virtual networks will be studied.

Since the data communication structure of the 3D algorithm is three-dimensional, our virtual networks will be 3D networks. We call a network a 3D network of certain graphs if the network is obtained by replacing every ring connection in each of the three dimensions of the 3D mesh with such a graph. For example, a 3D mesh of hypercubes is a network which is obtained by replacing every ring connection in each of the three dimensions of the 3D mesh with a hypercube connection (which by Lemma 6 is a hypercube). Similarly, there are the 3D mesh of $s$-D meshes (the same as the $3s$-D mesh) and the 3D mesh of trees (with every ring in each of the three dimensions replaced with a balanced binary tree; different from the conventional one in, e.g., [32]).

The physical networks that we consider are the meshes of arbitrary dimension and the hypercube. We choose the following 3D networks as virtual networks: the 3D mesh, the 3D mesh of hypercubes, and the 3D meshes of $s$-D meshes, $s > 1$.

Before presenting the theorem concerning implementations of the 3D algorithm on the virtual networks, we give two lemmas that are needed in the proof of the theorem. Proofs for the two lemmas can be found in Section 6.

**Lemma 4.** Assume that a network of size $p$ has the property that for any subset of nodes of size $\Omega\left(\log p\right)$ the maximum distance between any pair of nodes in this subset is asymptotically no smaller than the diameter of the network. Then under the assumption of Theorem 1, the start-up time of any algorithm for general matrix multiplication on the network is asymptotically no smaller than the diameter of the network.

**Corollary 1.** Under the assumption of Theorem 1, the start-up time of any algorithm for general matrix multiplication on a mesh of arbitrary dimension or on a hypercube is asymptotically no smaller than the diameter of the network.

**Proof.** By Lemma 4, the proof consists of verifying for everyone of these networks the property that for any subset of nodes of size $\Omega\left(\log p\right)$ the maximum distance between any pair of nodes in this subset is asymptotically no smaller than the diameter of the network. The statement is easy to verify for a mesh of arbitrary dimension $k$. For the hypercube it follows from the fact that the diameter of the hypercube is $\log p$ and the result that the radius of any sphere of $\Omega\left(p\right)$ nodes in the hypercube is $\Omega\left(\log p\right)$. The latter can be obtained through estimating partial sums of a binomial series. We omit the details here. Q.E.D.

The next lemma uses the notion of the minimal bisection width of a network, which is defined as the size of the smallest cut-set of the network graph. A cut-set is a subset of edges whose removal splits the graph into two equal halves.

**Lemma 5.** Under the assumption of Theorem 1, the bandwidth time of any algorithm for general matrix multiplication on any of the following networks is $\Omega\left(p^{2}/\omega(p)\right)$, where $\omega(p)$ is the minimal bisection width of the network: the 1D, 2D, and 3D meshes.
Corollary 2. For \( s = 1 \) or \( 2 \), the \( s-D \) algorithm implemented on the \( s-D \) mesh as in Section 3, without the use of batching or splitting of packets, is optimal for the network.

Proof. It follows from Corollary 1, Lemma 5, and the fact that, for \( s = 1 \) or \( 2 \), the diameter of the \( s-D \) mesh is \( O(p^{1/3}) \) and the minimal bisection width of the \( s-D \) mesh is \( O(p^{(s-1)/3}) \).

Theorem 2. The \( 3-D \) algorithm can be implemented on the \( 3-D \) mesh, without the use of batching or splitting of packets, to achieve optimal start-up time \( O(p^{1/3}) \) and optimal bandwidth time \( O(n^2/p^{2/3}) \). It can be implemented on the \( 3-D \) mesh of hypercubes to achieve optimal start-up time \( O(\log p) \) and optimal bandwidth time \( O(n^2/p^{2/3} \log p) \). It can be implemented on the \( 3-D \) mesh of \( s-D \) meshes (\( s > 1 \)) to achieve optimal start-up time \( O(p^{1/3}) \) and optimal bandwidth time \( O(n^2/p^{2/3}) \).

Proof. We consider the \( 3-D \) mesh, the \( 3-D \) mesh of hypercubes, and the \( 3-D \) mesh of \( s-D \) meshes (\( s > 1 \)), in that order.

The same technique used in Section 3 to implement the 1D and 2D algorithms on the 1D and 2D meshes can be used to implement the limited-complete-broadcast primitive of the \( 3-D \) algorithm on the \( 3-D \) mesh, executed concurrently among every 1D ring in the \( i \)- and \( j \)-dimensions. This achieves start-up time \( O(p^{1/3}) \) and bandwidth time \( O(n^2/p^{2/3}) \). The following technique is used to implement the limited-histogramming primitive among every 1D ring in the \( k \) dimension: after the local computation stage, each processor in such a ring organizes the \( O(n^2/p^{2/3}) \) partial-sum values it holds into \( p^{1/3} \) packets of size \( O(n^2/p) \), each destined for a different processor on the ring (including one for itself); to start the procedure, each 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 \( O(n^2/p^{2/3}) \). Note that no batching or splitting of packets is used.

Optimality of start-up time follows from Lemma 4 and the fact that the diameter of the \( 3-D \) mesh is \( O(p^{1/3}) \). Optimality of the bandwidth time follows from Lemma 5 and the fact that the minimum bisection width of the \( 3-D \) mesh is \( O(p^{2/3}) \).

For the \( 3-D \) mesh of hypercubes, Stage 1 of the \( 3-D \) algorithm, limited-complete-broadcast, can be carried out by concurrently calling the hypercube complete-broadcast procedure due to Stout and Wager [28] in every sub-hypercube 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 \( O(mp^{1/3}/\log p^{1/3}) \), where \( m \) is the size of a packet (see [28]). Since the size of a packet is \( \Theta(r^2/p) \), the bandwidth time is \( O(n^2/p^{2/3} \log p) \). 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 and Wager [28], 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.

Optimality of start-up time follows from Lemma 4 and the facts that the \( 3-D \) mesh of hypercubes is a hypercube (Lemma 6) and that the diameter of the hypercubes is \( O(\log p) \). We now show that the bandwidth time on the \( 3-D \) mesh of hypercubes is also optimal. By Theorem 1, at least one processor has to transfer \( \Omega(n^2/p^{2/3}) \) 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 \( \Omega(n^2/p^{2/3} \log p) \) holds for bandwidth time.

For the \( 3-D \) mesh of \( s-D \) meshes (\( s > 1 \)), both limited-complete-broadcast and limited-histogramming can be executed in every appropriate \( s-D \) submesh to achieve start-up time \( O(p^{1/3}) \) and bandwidth time \( O(n^2/p^{2/3}) \). The main idea is to run complete broadcast or histogramming \( s \) times. Each time it is run among every ring of \( p^{1/3} \) processors in a particular dimension of the \( s-D \) submesh, one dimension at a time. After each run, packets are reorganized appropriately by batching or splitting. Each run is executed in the same way that it is executed among a ring of processors in the implementation of the \( 3-D \) algorithm on the \( 3-D \) mesh. We omit the details here.

Optimality of the start-up time on the \( 3-D \) mesh of \( s-D \) meshes follows from Lemma 4 and the fact that the diameter of the \( 3-D \) mesh of \( s-D \) meshes is \( \Theta(p^{1/3}) \). Optimality of the bandwidth time follows from Theorem 1 and the fact that each processor in the network has only a bounded number of links.

Q.E.D.

5. Emulation of Virtual Architectures

In this section we implement the \( 3-D \) algorithm on a wide spectrum of physical networks by selecting a suitable virtual network for each physical network and finding a good emulation of the former on the latter. Now we need to utilize properties of the physical network.

The physical networks we consider are the meshes of arbitrary dimension and the hypercube. More specifically, we implement the \( 3-D \) algorithm on the 1D, 2D, and 3D meshes through emulation of the \( 3-D \) mesh, and on the hypercube through emulation of the \( 3-D \) mesh of hypercubes. Implementation of the \( 3-D \) algorithm on the \( (3r - r) \)-D mesh (\( s > 1, r = 0, 1, 2 \)) is done through emulation of the \( 3-D \) mesh
of s-D meshes. We prove optimality for all cases but the (3s−r)-D meshes, where s > 1, r = 1, 2.

We present implementation and analysis for the hypercube and then for the meshes. We first review the standard reflected Gray 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^d for some positive integer d, we can decompose the binary string into three segments of equal length. Then the same Gray code defines a subhypercube whenever values of two of the three substrings are fixed. This gives a 3D mesh of hypercube. Note that in this coding all the links of the hypercube are also links of the 3D mesh of hypercubes. We thus have the following.

**Lemma 6.** The 3D mesh of hypercubes is a hypercube.

This lemma and Theorem 2 lead immediately to the following theorem.

**Theorem 3.** Let p = 2^d, d > 0. The 3D algorithm can be implemented on the hypercube through emulation of the 3D mesh of hypercubes virtual network to achieve optimal parallel time 2n^p, optimal start-up time \(\Theta(\log p)\) and optimal bandwidth time \(\Theta(n^p/p^{d/3} \log p)\).

**Theorem 4.** The 3D algorithm can be implemented on the 1D, 2D, and 3D meshes through emulation of the 3D mesh virtual network, without the use of batching or splitting of packets, to yield simultaneous optimal parallel time, optimal start-up time, and optimal bandwidth time on each of these networks.

Theorem 4 is a consequence of the next lemma which is more general and is proved in Section 6. 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.

**Lemma 7.** 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 3D algorithm on the 3D mesh takes optimal start-up time \(O(p^{1/3})\), i.e., \(O(p^{1/3})\) steps of packet movements, and optimal bandwidth time \(O(n^p/p^{d/3})\) because the size of each packet is \(O(n^p/p)\) (see proof of Theorem 2). By Lemma 7, these same packet movements can be done in \(O(p)\) steps in the 1D mesh and \(O(\sqrt{p})\) steps on the 2D mesh. Therefore the algorithm achieves a start-up time of \(O(p)\) and a bandwidth time of \(O(n^p)\) on the 1D mesh, and it achieves a start-up time of \(O(\sqrt{p})\) and a bandwidth time of \(O(n^p/\sqrt{p})\) on the 2D mesh. All are optimal for the respective networks (Corollary 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 used in the emulation or in the earlier implementation of the 3D algorithm on the 3D mesh.

Q.E.D.

Lemma 7 also yields the following.

**Theorem 5.** The 3D algorithm can be implemented on the (3s−r)D mesh, s > 1, r = 0, 1, 2, through emulation of the 3D mesh of s-D meshes virtual network, to yield simultaneously optimal parallel time, optimal start-up time \(O(p^{1/3s−r−1})\), and bandwidth time \(O(n^p/p^{d/3−r−3s/(3s−r)})\). The bandwidth time is also optimal when \(r = 0\).

**Proof.** The 3D mesh of s-D meshes is the same as the 3s-D mesh. Emulate the 3D mesh of s-D meshes virtual network on the 3s-D mesh which is in turn emulated on the (3s−r)D mesh, and apply Theorem 2 and Lemma 7. Optimality of the implementations on the 3s-D meshes follows from Theorem 2.

Q.E.D.

We do not know whether our implementations of the 3D algorithm on the (3s−r)D meshes when s > 1 and r = 1, 2 are optimal. It is easy to see that the implementations are better than any implementations of the 1D or 2D algorithm. Also note that the 3D algorithm is implemented on each 3s-D mesh individually in Section 4. We conjecture that there do not exist one algorithm and one virtual network which can be implemented on all the 3s-D meshes to achieve portable optimality for OMM. The following results show that this is not the case when one minimizes only the bandwidth time or only the start-up time.

**Theorem 6.** The 3D algorithm can be implemented on the (3s−r)D meshes for s > 1, r = 0, 1, 2, through emulation of the 3D mesh alone, to achieve optimal parallel time, bandwidth time \(O(n^p/p^{d/3−r−3s/(3s−r)})\), and start-up time \(O(p^{1/3−r−3s/(3s−r)})\). The bandwidth time is optimal when \(r = 0\).

**Proof.** Emulate the 3D mesh on the 3s-D mesh, which is in turn emulated on the (3s−r)D mesh. The first emulation uses the fact that the 3D mesh (torus) is a subgraph of the 3s-D mesh (torus), and so both bandwidth time and start-up time of the 3D algorithm on the 3D mesh carry over to the 3s-D mesh. To verify this fact, one needs to show that a ring is a subgraph of a s-D torus; i.e., each s-D torus contains a hamiltonian cycle (cf. [4] for the meaning of the term). This can be verified through induction on the
Theorem 7. The 3D algorithm can be implemented on the \((3s-r)D\) meshes for all \(s \geq 1\), through emulation of one virtual network, the 3D mesh of trees, to achieve optimal parallel time and optimal start-up times \(O(p^{(3s-r)})\).

Proof Sketch. The 3D algorithm can be implemented on the 3D mesh of trees to achieve an optimal bandwidth time of \(O(n^2/p^{2/3})\) and an optimal start-up time of \(O(\log p)\). We first describe a simpler implementation that yields a larger bandwidth time and then we describe the main idea to improve it. To implement limited-complete-broadcast, simply let the packets of data to be shared by the \(p^{1/3}\) processors in a balanced binary tree be sent from the leaf nodes up to the root node, batched up along the way with one another and with packets to be sent from the internal nodes, and then broadcast down as a single packet to every processor. For limited-histogramming, summing-up be done at different levels of the tree at the same time. We omit the details. The proof of optimality for the implementation is similar to that in Theorem 2 for the implementation on the 3D mesh using Lemmas 4 and 5. We omit it here.

The 3D mesh of trees can then be emulated on the \((3s-r)D\) meshes (\(r = 0, 1, 2\)) as in Lemma 7. It is not hard to verify that the desired start-up time of \(O(p^{(3s-r)})\) is achieved.

Note that in the proof, in order to achieve the desired small start-up time, batching and splitting of packets are used both in the implementation of the algorithm on the virtual network and in the emulation. The emulation is thus not in the same model as the ones in Lemma 7, where batching or splitting of packets is not allowed. Also, the bandwidth time is very large due to the large congestion of the embedding which we did not estimate.

So, minimizing both the bandwidth time and the start-up time may sacrifice uniformity at the virtual network level. This means that implementation of the communication primitives may have to be more architecture-dependent to be optimal. But we point out that in a practical situation, given concrete \(t_s\) and \(t_r\), it may not be necessary to minimize both communication times since one of them may dominate the other.

6. PROOFS OF SOME LEMMAS AND RELATED RESULTS

Lemma 3. For any processor \(P_i\) for which \(\text{VOLUME}(U_i) = \Omega(n^3/p)\),

\[
\text{AREA}(P_{xy}(U_i)) + \text{AREA}(P_{xz}(U_i)) + \text{AREA}(P_{xy}(U_i) \cap P_{xz}(U_i - U_j)) = \Omega(n^2/p^{2/3}).
\]

Proof. Let \(P_i\) be any processor for which \(\text{VOLUME}(U_i) = \Omega(n^3/p)\). If

\[
\text{AREA}(P_{xy}(U_i)) + \text{AREA}(P_{xz}(U_i)) = \Omega(n^2/p^{2/3})
\]

we are done. So suppose

\[
\text{AREA}(P_{xy}(U_i)) + \text{AREA}(P_{xz}(U_i)) = o(n^2/p^{2/3}).
\]

Then

\[
\text{AREA}(P_{xy}(U_i)) = \Omega(n^2/p^{2/3})
\]

by Lemma 2. This implies that

\[
\text{AREA}(P_{xy}(U_i) \cap P_{xz}(U - U_j)) = \Omega(n^2/p^{2/3}),
\]

for otherwise (6.3) and

\[
\text{AREA}(P_{xy}(U_i) \cap P_{xz}(U - U_j)) = o(n^2/p^{2/3})
\]
holds due to Thompson [31]. It is shown in Fisher [12] for $|w|_{0}$ words that have to be communicated between them is $\Omega(n^2/p^2)$ input data but not all the processors necessarily hold input data. What we need is a partition of the network into two parts such that each part holds $\Theta(n^2/p^2)$ input data and the cut-set is of size $O(\alpha(p))$, given that each of $\Omega(p)$ of them holds $\Theta(n^2/p^2)$ input data. For the 1D mesh this is easy. For the 2D and 3D meshes, we use an argument based on the partition tree in Section 3.3 of Ullman [32] to partition each network into two subsets each holding between $2n^2/3$ and $4n^2/3$ input data, with the size of the cut-set $O(\alpha(p))$.

We omit the details here.

Q.E.D.

**Lemma 4.** Assume that a network of size $p$ has the property that for any subset of nodes of size $\Omega(p)$ the maximum distance between any pair of nodes in this subset is asymptotically no smaller than the diameter of the network. Then under the assumption of Theorem 1, the start-up time of any algorithm for general matrix multiplication on the network is asymptotically no smaller than the diameter of the network.

**Proof.** The proof uses the following argument due to Gentleman [14]. Take two arbitrary distinct processors. Suppose one holds some input element $a_{i,j}$ and the other some input element $b_{i,j}$. Consider a third processor that will hold the output element $c_{i,j}$. From each of the first two processors a piece of information must travel to the third one. This implies that one piece of information must traverse a path whose length is at least half of the distance between the first two processors.

Under the assumption of Theorem 1, at least $\Omega(p)$ processors hold input elements exclusively. For any two distinct processors one of which holds some $a_{i,j}$ and the other some $b_{i,j}$, the above argument is valid. If two distinct processors only hold elements of the same matrix, without loss of generality, say $A$, then the above argument is valid regarding the distance between each of them to any third processor that holds some element of $B$. This implies that one piece of information must traverse a path whose length is at least $\frac{1}{2}$ the distance between the first two processors. Thus, one piece of information must traverse a distance asymptotically no smaller than the diameter of the network under the assumption of the lemma. Hence the lower bound on start-up time.

Q.E.D.

**Lemma 5.** Under the assumption of Theorem 1, the bandwidth time of any algorithm for general matrix multiplication on any of the following networks is $\Omega(n^2/\alpha(p))$, where $\alpha(p)$ is the minimal bisection width of the network: the 1D, 2D, and 3D meshes.

**Proof.** The proof uses the following type of argument due to Thompson [31]. It is shown in Fisher [12] for general matrix multiplication that if one partitions the processors into two arbitrary subsets such that each subset holds $\Theta(n^2)$ input data exclusively, then the number of words that have to be communicated between them is $\Omega(n^2)$. For a network with minimal bisection width $\alpha(p)$, at least one of the links in the minimal cut-set has to transfer $\Omega(n^2/\alpha(p))$ words.

A straightforward partitioning of the $p$ processors of a network into two subsets of equal size would not allow application of the above argument. This is because under the assumption of Theorem 1 each processor holds $O(n^2/p)$ input data but not all the processors necessarily hold input data. What we need is a partition of the network into two parts such that each part holds $\Theta(n^2/p)$ input data and the cut-set is of size $O(\alpha(p))$, given that each of $\Omega(p)$ of them holds $\Theta(n^2/p)$ input data. For the 1D mesh this is easy. For the 2D and 3D meshes, we use an argument based on the partition tree in Section 3.3 of Ullman [32] to partition each network into two subsets each holding between $2n^2/3$ and $4n^2/3$ input data, with the size of the cut-set $O(\alpha(p))$.

We omit the details here.

Q.E.D.

Before proving Lemma 7 on mesh embedding, we present a negative result concerning product of dilation and congestion (see below) as an upper bound for the number of emulation steps. This is to motivate the approach taken in the proof of the lemma. Most previous results on emulation between interconnection networks have used the product of dilation and congestion as an upper bound on the number of emulation steps. For the meshes we consider, the following proposition shows that no embedding can produce such an upper bound good enough for our purpose.

We thus will have to go down to a finer level of analysis of the embeddings in the proof. This shows the limitation of considering only product of dilation and congestion in studying emulation between interconnection networks.

The *dilation* of an embedding of a guest graph $G$ in a host graph $H$ is defined as the maximum length of the paths in $H$ any edge in $G$ is mapped onto. The *congestion* of the embedding is defined as the maximum number of edges in $G$ any edge in $H$ supports (cf. [6]).

**Proposition 1.** Let $m > q$ be positive integers. Any embedding of the $m$-D mesh in the $q$-D mesh has a congestion of $O(p^{q/(q-1)m})$ and a dilation of $O(p^{q/(q-1)m})$.

**Proof.** The lower bound on congestion follows from the fact that the minimal bisection widths of the $m$-D and $q$-D meshes are $\Theta(p^{3m-1}/m)$ and $\Theta(p^{q-1}/q)$, respectively, and that their ratio $\Theta(p^{q/(q-1)m})$ 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. The argument for the lower bound on dilation goes as follows. The diameter of the $m$-D mesh and the $q$-D mesh are $\Theta(p^{1/m})$ and $\Theta(p^{1/q})$, respectively. Let $P_1$ and $P_2$ be two processors which have a distance of $\Omega(p^{1/m})$ between them in the $q$-D mesh. Consider any path of length $O(p^{1/m})$ between them in the $m$-D mesh. At least one edge on this path must be embedded in a path of length $\Omega(p^{q/(q-1)m})$ in the $q$-D mesh.

Q.E.D.

The proposition means that there is no hope to obtain the upper bound we want on the optimal number of emulation steps by estimating dilation and congestion separately.
The approach we take below is to find an embedding with optimal dilation and then schedule the token movement on the host graph in such a way that congestion does not cause more than a constant factor of delay in the emulation.

**Lemma 7.** Let \( m > q \) be positive integers and \( \log p \) be divisible by the least common multiple of \( m \) and \( q \). The \( m \times D \) mesh can be embedded in the \( q \times D \) mesh in such a way that one step of token movement on the former can be emulated by an optimal \( O(p^{1/q} - 1/\omega) \) step of token movement on the latter.

**Proof.** We only give a 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 the end.

The embedding of a mesh in another can be determined by specifying how every 1D 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 \( \{0, 1, 2, \ldots, L - 1\} \) can be represented by the set of all binary strings of length \( \log L \) in which 0 is represented by \( (0, 0, \ldots, 0) \) and the representation of \( M > 0 \) is obtained by taking the binary string which represents \( M - 1 \) 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.

**Emulation of 3D Mesh Token Movement on 1D Mesh**

Let \( \log p = 3d \), where \( d \) is a positive integer. The 1D mesh of size \( p \) will be coded as above by the set of binary strings of length \( 3d \): \( \{czd, czd-1, \ldots, cz1\} \). The 3D mesh of size \( p \) will be coded by the same set of binary strings \( \{(zxz, \ldots, z1, ydy, \ldots, y1, xdx, \ldots, x1)\} \). Here each of the three substrings of length \( d \) codes rings in each of the three dimensions of the 3D mesh, also using the binary representation of natural numbers. The nodes of the 3D mesh will be mapped to the nodes of the 1D 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 3D mesh the series of legal moves on the 1D mesh which emulate it. Without loss of generality, we shall consider only forward legal moves on the 3D 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 3D mesh can be executed fully concurrently, and so the emulation of one step of token movement on the 3D mesh can be done in less than \( 6p^{2/3} \) steps on the 1D mesh.

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

\[
(z'x, \ldots, z'1, y'd, \ldots, y'1, x'd, \ldots, x'1) \\
\Rightarrow (z'x, \ldots, z'1, y'd, \ldots, y'1, x'd, \ldots, x'1)
\]

and \((x'd, \ldots, x'1) \neq (1, \ldots, 1)\). Then the flipping of bits which represents the forward legal move on the 3D mesh also represents a forward legal move in the 1D mesh. Now suppose \((x'd, \ldots, x'1) = (1, \ldots, 1)\). Then the flipping of bits which represents the forward legal move on the 3D mesh no longer represents a legal move on the 1D mesh because for such a move to be legal on the latter we would require also flipping some of the more significant bits beyond the range \( x'd, \ldots, x'1 \).

However, we can still emulate this move by a series of \( p^{1/3} \) backward legal moves on the 1D mesh, equivalent to moving backwards along the ring on the \( x \)-direction of the 3D mesh:

\[
(z'x, \ldots, z'1, y'd, \ldots, y'1, 1, \ldots, 1, 0) \\
\Rightarrow (z'x, \ldots, z'1, y'd, \ldots, y'1, 0, \ldots, 0, 0, 0).
\]

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

\[
(z'y, \ldots, z'y1, y'd, \ldots, y'y1, x'd, \ldots, x'y1) \\
\Rightarrow (z'y, \ldots, z'y1, y'd, \ldots, y'y1, x'd, \ldots, x'y1)
\]

and \((y'd, \ldots, y'y1) \neq (1, \ldots, 1)\). Then the forward legal move on the 3D mesh is no longer legal on the 1D mesh because for such a move to be legal on the latter would require also the substring \((x'd, \ldots, x'y')\) be equal to \((1, \ldots, 1)\) and to be flipped to \((0, \ldots, 0)\). However, we can emulate it by a series of \( p^{1/3} \) forward legal moves on the 1D mesh, flipping through all possible values of the substring \((x'd, \ldots, x'y')\): first move forward all the way from \((z'y, \ldots, z'y1, y'd, \ldots, y'y1, x'd, \ldots, x'y1)\) to \((z'y, \ldots, z'y1, y'd, \ldots, y'y1, 1, \ldots, 1)\); then make the legal move

\[
(z'y, \ldots, z'y1, y'd, \ldots, y'y1, 1, \ldots, 1) \\
\Rightarrow (z'y, \ldots, z'y1, y'd, \ldots, y'y1, 0, \ldots, 0, 0, \ldots, 0)
\]

and finally move forward all the way from \((z'y, \ldots, z'y1, y'd, \ldots, y'y1, 0, \ldots, 0, 0)\) to \((z'y, \ldots, z'y1, y'd, \ldots, y'y1, x'd, \ldots, x'y1)\). Now suppose \((y'd, \ldots, y'y1) = (1, \ldots, 1)\). We emulate this forward legal move by moving backwards on the 1D mesh all the way from \((z'y, \ldots, z'y1, 1, \ldots, 1, x'd, \ldots, x'y1)\) to \((z'y, \ldots, z'y1, 0, \ldots, 0, x'd, \ldots, x'y1)\). This requires flipping through almost all possible values of the substring \((y'd, \ldots, y'y1, x'd, \ldots, x'y1)\), a total of no more than \(p^{2/3} \) backward moves.
In exactly the same way as in the case of a forward legal move along the y-direction when \((y_{1}^{\prime}, \ldots, y_{t}^{\prime}) \neq (1, \ldots, 1)\), any forward legal move along the z-direction, 
\[
(z_{d}^{\prime}, \ldots, z_{1}^{\prime}, y_{d}^{\prime}, \ldots, y_{1}^{\prime}, x_{d}^{\prime}, \ldots, x_{1}^{\prime})
\]
including when \((z_{d}^{\prime}, \ldots, z_{1}^{\prime}) = (1, \ldots, 1)\), can be emulated by a series of \(p^{2/3}\) forward legal moves on the 1D mesh by flipping through all possible values of the substring 
\[(y_{1}^{\prime}, \ldots, y_{1}, x_{d}^{\prime}, \ldots, x_{1}),\]
We have shown that the dilation of the embedding is \(O(p^{2/3})\). One can show that the congestion of the embedding is also \(O(p^{2/3})\). We need to show that congestion does not delay token movement by more than a constant factor in this embedding. The key lies in the observation that a legal move on the 3D mesh is emulated by a series of forward legal moves only or backward legal moves only on the 1D mesh; i.e, a series of emulating moves does not reverse direction. For an arbitrary step of token movement on the 3D mesh consider the part of token movement along a single direction. It can be emulated on the 1D mesh in lockstep movement. Namely, each token starts at a different node and they flow orderly along either a forward or backward direction of the 1D mesh for no longer than \(p^{2/3}\) steps, never running into or overtaking one another. Emulation of one step of token movement on the 3D mesh can therefore be done in \(6p^{2/3}\) steps of token movement on the 1D mesh, by emulating the six directions of the 3D mesh one by one. The bound can be reduced to \(2p^{2/3} + 2p^{1/3} + 2\) when one keeps track of the details.

(b) Emulation of 3D Mesh Token Movement on the 2D Mesh

Let \(p = 2^{ad}\) for some positive integer \(d\). The 2D mesh of size \(p\) is coded, in the standard binary representation of natural numbers, by the set of binary strings of length \(\log p\): 
\[\{(b_{a1}, \ldots, b_{1}, a_{a1}, \ldots, a_{1})\},\]
where the two substrings code rings in the \(a-\) and \(b-\)dimensions of the 2D mesh, respectively. The 3D mesh of the same size is coded in the same binary representation of natural numbers by the same set of binary strings 
\[\{(y_{2d}, \ldots, y_{1}, z_{2d}, \ldots, z_{d+1}, x_{2d}, \ldots, x_{1})\},\]
where the two substrings \((x_{2d}, \ldots, x_{1})\) and \((y_{2d}, \ldots, y_{1})\) code rings in the \(x-\) and \(y-\)dimensions of the 3D mesh, respectively, and 
\[(z_{2d}, \ldots, z_{1})\]
the concatenation of \((z_{2d}, \ldots, z_{d+1})\) with \((z_{d}, \ldots, z_{1})\), codes rings in the \(z-\)dimension of the 3D mesh. Again the nodes of the 3D mesh are mapped to the nodes of the 2D mesh under the identity mapping. We specify embedding of the edges by describing for each legal move on the 3D mesh the emulating series of legal moves on the 2D mesh. Again without loss of generality only the 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 3D 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 3D mesh by \(3p^{1/6}\) steps on the 1D mesh.

First consider forward legal moves along \(x-\) and \(y-\)directions. This is analogous to the case of emulating legal moves along the \(z-\)direction of the 3D mesh on the 1D mesh. A ring in the \(x-\)dimension of the 3D mesh is embedded in a single ring in the \(a-\)dimension of the 2D mesh and is coded by the 2\(d\) most significant bits of the substring \((a_{a1}, \ldots, a_{1})\). Any forward legal move along the \(x-\)direction on the 3D mesh can thus be emulated by a series of \(p^{1/6}\) forward legal moves in the \(a-\)dimension of the 2D mesh by flipping through all possible values of the substring \((a_{a1}, \ldots, a_{1})\). Therefore, each ring in the \(x-\)dimension of the 3D mesh is evenly stretched along the forward direction to fit into a ring in the \(a-\)dimension of the 2D mesh, and each ring in the \(a-\)dimension of the 2D mesh supports \(p^{1/6}\) rings in the \(x-\)dimension of the 3D mesh. By the same argument as in emulating the 3D mesh on the 1D 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 2D mesh. By symmetry, one step of token movement consisting of moves (forward and backward) along the \(y-\)direction of the 3D mesh can be emulated by \(p^{1/6}\) steps of moves along the \(b-\)direction on the 2D mesh. Since 1D rings in different dimensions of the 2D mesh are edge-disjoint, these two \(p^{1/6}\) steps of moves can be executed concurrently.

Now consider forward legal moves in the \(z-\)direction of the 3D mesh. First consider the case where the move consists of flipping bits only among \(z_{d}, \ldots, z_{1}\). Since these bits are the same as the \(d\) least significant bits \(a_{d}, \ldots, a_{1}\) which code rings in the \(a-\)dimension of the 2D mesh, the move is still a forward legal move along the \(a-\)direction of the 2D mesh. Apparently, two different such moves on the 3D mesh remain different moves on the 2D mesh. A step of token movement on the 3D mesh consisting of only such moves can thus be made in one step on the 2D mesh.

Now consider the case where the move requires flipping also bits among \(z_{2d}, \ldots, z_{d+1}\). These bits are the same as the \(d\) least significant bits \(b_{d}, \ldots, b_{1}\) that code rings in the \(b-\)dimension of the 2D mesh. In this case \((z_{d}, \ldots, z_{1})\) must be \((1, \ldots, 1)\) and be flipped to \((0, \ldots, 0)\). Let the move be 
\[
(y_{2d}, \ldots, y_{1}, z_{2d}, \ldots, z_{d+1}, z_{2d}, \ldots, z_{d+1}, x_{2d}, \ldots, x_{1}, 1, \ldots, 1)
\]
\[
(y_{2d}, \ldots, y_{1}, z_{2d}, \ldots, z_{d+1}, x_{2d}, \ldots, x_{1}, 0, \ldots, 0).
\]
First suppose \((z_{2}\delta, \ldots, z_{2}\delta+1) \neq (1, \ldots, 1)\). Then the move can be emulated on the 2D 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 \((y_{2}\delta, \ldots, y_{1}, z_{2}\delta, \ldots, z_{2}\delta+1, x_{2}\delta, \ldots, 1, \ldots, 1)\), and then move backwards along the \(a\)-direction all the way from
\[
(y_{2}\delta, \ldots, y_{1}, z_{2}\delta, \ldots, z_{2}\delta+1, x_{2}\delta, \ldots, x_{1}' , 1, \ldots, 1)
\]
to
\[
(y_{2}\delta, \ldots, y_{1}', z_{2}\delta, \ldots, z_{2}\delta+1, x_{2}\delta, \ldots, x_{1}', 0, \ldots, 0).
\]
Note that two different such moves on the 3D mesh are emulated by two different series of moves on the 2D mesh which do not share edges, so a step of token movement on the 3D mesh consisting of only such moves can be emulated in \(p^{1/6}\) steps on the 2D mesh. Now suppose \((z_{2}\delta, \ldots, z_{2}\delta+1) = (1, \ldots, 1)\). Then the move
\[
(y_{2}\delta, \ldots, y_{1}, 1, \ldots, 1, x_{2}\delta, \ldots, x_{1}', 1, \ldots, 1)
\]
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 \((y_{2}\delta, \ldots, y_{1}, 0, \ldots, 0, x_{2}\delta, \ldots, x_{1}', 1, \ldots, 1)\), and then move backwards along the \(a\)-direction all the way to \((y_{2}\delta, \ldots, y_{1}', 0, \ldots, 0, x_{2}\delta, \ldots, x_{1}', 0, \ldots, 0)\). 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 3D mesh can be done in \(2p^{1/6}\) steps on the 2D mesh. By careful examination we can fully overlap execution of all emulating series in all these cases concerning forward legal moves along the \(z\)-direction of the 3D mesh. We observe that only the first two emulating steps on the 2D mesh involve forward legal moves. So, by symmetry, backward legal moves along the \(z\)-direction of the 3D mesh can be emulated essentially concurrently. A step of token movement consisting of moves along the \(z\)-direction of the 3D mesh can therefore be emulated by \(2p^{1/6}\) steps of token movement on the 2D mesh. The total number of steps of token movement on the 2D mesh to emulate one step of token movement on the 3D 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 substrings of length \(\log p/q\) in coding the \(q\)-D mesh while these substrings 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 \(O(p^{1/6} - 1)^{m}\) follows from Proposition 1 and the fact that the dilation of any embedding is a lower bound on the number of emulation steps needed using that embedding. Q.E.D.

7. TOWARDS STRUCTURED PARALLEL COMPUTING

In this section we summarize our conclusions and discuss briefly the issues concerning a structured approach to parallel computing. In this paper, we propose a framework of architecture-independent algorithm design for parallel computing on distributed-memory architectures. In our framework, there are three levels of design: architecture-independent algorithm design, virtual network design, and design of emulations of virtual networks on physical networks.

At the level of architecture-independent algorithm design, we adopt an environment in which the algorithm designer explicitly handles computation parallelism as well as communication at the level of data dependency. No specific interconnection network is assumed at this level. 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. We introduce complexity measures for architecture-independent analysis of algorithms.

Upon obtaining an algorithm with optimal computation parallelism and minimal communication at the architecture-independent level, we design virtual networks that are tailored to fit the communication need of the algorithm. These virtual networks are then emulated on different physical networks. For the example of ordinary matrix multiplication, we show that our methodology leads to portable optimality of a single algorithm on a wide spectrum of physical networks. In Gao [13], we will present a general theory of portable optimality that applies to a wide range of computational problems.

The technical results for the two lower levels of design are of greater generality and can be used for designing algorithms for other applications.

Parallel computing has thus far failed to become a realistic means of general computing despite the availability of parallel machines and the substantial research activities in parallel computation. This is due to the lack of a good programming environment that would hide the vast, constantly changing details of parallel architectures from the algorithm designer and the lack of a good algorithm design paradigm that would make efficient algorithms less sensitive to an architecture, so as to have greater generality and longevity.

Sequential computing, historically, has gone through a similar stage in its development. In the early 1950s, most of the effort a programmer spent in developing a program was devoted to managing details of floating-point and indexing operations. Early automated systems intended to provide a higher level programming environment to the programmer in general did not produce efficient programs. An important step forward in the evolution of sequential computing was the introduction of successful high level programming
structured parallel computing

languages such as FORTRAN and ALGOL. These high level programming languages relieved the programmer of many red-tape issues in developing a program so that the programmer could focus on more algorithmic issues. But the success of these high level programming languages was made possible only by the realization that the key to their acceptance was good compilers capable of translating a high level program into a machine code that is as efficient as a hand-coded one [3], and by effective programming methodologies that stressed the importance of structuring programs according to well-formulated principles and around well-defined abstract data structures [9]. Uniformity at a high level of abstraction was achieved without sacrificing efficiency at the machine level.

The abstraction of the notion of algorithm from specific machines and programming languages and the use of asymptotic complexity to measure the “goodness” of algorithms designed on abstract data structures, utilized the full potential of this structured, hierarchical approach to sequential computing [2, 30]. By developing theoretically efficient algorithms, designing good data structures which fit their need for data access, and finding efficient emulation of the data structures on the machine memory, one was able to reconcile machine independence of algorithms and algorithm design at the high level with efficient computation and memory access at the machine level. In particular, the algorithm designer was relieved from the details of data access without sacrificing its efficiency.

And here is where one finds the relevance of this historical digression. The motivation for parallel computers was elimination of the memory access bottleneck. But the problem has not gone away with the coming of parallel computers. The speed of a large number of processors in a machine can be utilized only when memory access—distributed memory access in this case—is efficient enough to support computation. Data communication in parallel computing thus plays the same crucial role as memory access in sequential computing. Therefore, the success of parallel computing as a means of general computing will hinge on the development of not only good high level programming languages that will take care of red-tape issues for the programmer and good programming methodologies to make programs manageable, but also a good algorithm design paradigm which will reconcile uniformity and machine independence of algorithms and algorithm design at a high level of abstraction with efficiency of computation and data access (communication) at the machine level. What we demonstrate in this paper is the same organization principle—use of abstract data structures for data organization and access—which has proven to be successful for sequential computing, is very much relevant to parallel computing.

In this paper, we only considered a distributed memory environment for the high level algorithm design. It is conceivable that this methodology may lead to powerful parallel data structures that support other environments for algorithm design.

In the past few years, there have been a number of theoretical works that advocate the use of machine-independent programming environments. For example, both Valiant [34] and Kruskal et al. [19] proposed the emulation of variants of the PRAM on distributed-memory machines. In such a framework, an unrestricted machine environment (basically a complete connection network) is emulated on an actual machine through hashing and routing. In contrast, we only emulate a restricted machine environment for each algorithm (a virtual network tailored to fit the communication pattern of the algorithm). From a practical point of view, their approaches have disadvantages, which include large emulation overhead that needs to be masked by not very natural tricks, large “constant” overhead associated with probabilistic techniques used such as hashing, and applicability only on physical networks of small diameter such as the hypercube. Recently, there have begun to be works independent of ours which consider emulating restricted machine environments on distributed memory machines through use of a set of primitives. For example, Dally and Willis [11] had an ambitious goal of using a set of primitives to emulate several different machine environments on the J-machine. Skillicorn [26] discussed emulation of a restricted shared memory environment through a set of primitives on a variety of parallel machines. 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.

ACKNOWLEDGMENTS

I thank David Kirkpatrick for valuable discussions on relevant theoretical issues as well as on a draft of this paper. I thank Maria Klawe and Nick Pippenger for valuable comments and for criticism on a draft of this paper. I also thank Alan Wagner for very helpful suggestions including the hypergraph interpretation of data communication structures and for pointers to the literature. Discussions with Uri Ascher, Zhaqjun Bai, David Chin, Geng Lin, and Jim Little were also very helpful. I appreciate the valuable comments from Lenore Blum, Velvel Kahan, Dick Karp, Beresford Parlett, and Steve Smale during the preliminary stage of this research at the University of California, Berkeley. I also acknowledge the many detailed, helpful comments from the referee.

REFERENCES


