On Architecture Independent Design and Analysis of Parallel Programs *

Ananth Grama[†], Vipin Kumar[†], Sanjay Ranka[*], and Vineet Singh [**]

[†] Department of Computer Science,

University of Minnesota

Minneapolis, MN.

{ananth, kumar}@cs.umn.edu

http://www.cs.umn.edu/ãnanth

http://www.cs.umn.edu/k̃umar

[*] Department of Computer Science,

University of Florida

Gainesville, FL.

ranka@cis.ufl.edu

[**] Hewlett Packard Labs,

1500 Page Mill Road

Palo Alto, CA.

vsingh@hpl.hp.com

Abstract

The presence of a universal machine model for serial algorithm design, namely the von Neumann model, has been one of the key ingredients of the success of uniprocessors. The presence of such a model makes it possible for an algorithm to be ported across a wide range of uniprocessors efficiently. Universal models proposed for parallel computers however tend to be highly limited in the range of parallel computers they can efficiently cover. Consequently, portability of parallel programs is attained at the expense of loss of efficiency.

In this paper, we explore desirable and attainable properties of universal models of architecture independent parallel program design. We study various models that have been proposed, classify them based on important machine parameters and study their limitations. We examine the role of aggregate communication operations. Based on this, we present a framework for analyzing communication overheads in parallel programs. We show that a wide variety of programs designed in the framework of aggregate communication operations can be ported to a range of architectures with minimal loss of efficiency. Furthermore, the analytical expressions derived from our framework for a class of bandwidth insensitive aggregate communication operations remain unchanged across platforms and are indeed a good reflection of the actual runtime. This methodology affords programmers ease of analysis and comes closer to existing programming environments based on MPI and HPF.

1 Introduction

The framework for an architecture independent programming model is illustrated in Figure 1. Algorithm designer A designs algorithms for solving problem P on abstract machine model I. A translator T exists that can take this algorithm and a description of a physical

*†This work was supported by IST/BMDO through Army Research Office contract DA/DAAH04-95-1-0538, NSF grant NSG/1RI-9216941, and by Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Related papers are available via WWW at URL: http://www.cs.umn.edu/users/kumar/papers.html. *The work of Sanjay Ranka was supported in part by NSF under ASC-9213821 and AFMC and ARPA under contract #F19628-94-C-0057. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

architecture, and develop a machine code for solving problem P on architecture A in a reasonable amount of time. This framework makes it possible to design algorithms in an architecture independent manner if the runtime of all algorithms on model I is asymptotically the same as its runtime on physical architectures. The ability of the model to determine the runtime of an algorithm on a physical architecture is referred to as its **predictive property**. Such an architecture independent programming model makes it possible to develop an algorithm for a given problem P that can run (asymptotically) optimally on a variety of physical architectures.

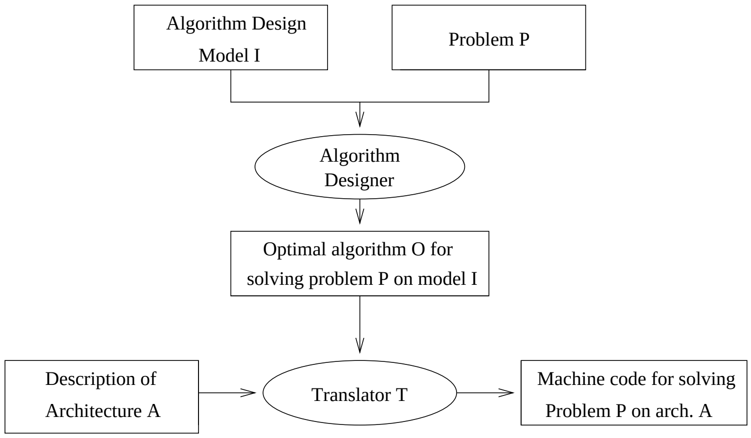

Figure 1: The role of algorithm design model I in solving problem P on architecture A.

For serial algorithm design, the von Neumann model serves as model I. Although the processor architecture and memory organization of sequential computers differ, they all conform to the same abstract model. Consequently, a program designed for the von Neumann model can be compiled and run relatively efficiently on any sequential computer. In particular, the asymptotic time complexity of an algorithm on different implementations of the von Neumann model is the same. Indeed, the existence of a universal model has played a key role in the impressive growth of the application of sequential computers. If algorithms had to be redesigned for every new sequential architecture, the cost of software development would have been much higher, greatly limiting the cost-effective application of computers.

For parallel program design, no satisfactory model I has been found that can be used to design programs for a variety of architectures, despite extensive efforts. An algorithm design model for parallel computers must exhibit two characteristics. First, it must be sufficiently general to capture the salient properties of a large class of parallel computers. Second, programs designed for this abstract model must execute efficiently on actual parallel computers. These characteristics represent conflicting goals. For example, a universal parallel programming model must make assumptions about the degree of connectivity among the processors. Networks such as linear arrays and meshes have a relatively low degree of connectivity compared to networks such as hypercubes. If the model assumes a low degree of connectivity, algorithms designed for it will be optimal for parallel computers with this characteristic. However, the same algorithms may be sub-optimal for architectures with a higher degree of connectivity, because the communication resources of the computer may be under-utilized. On the other hand, if the model assumes a high degree of connectivity among processors, then algorithms designed for the model may be sub-optimal on an architecture such as a mesh, yet may be optimal on a hypercube. Thus, regardless of the connectivity of the parallel model, it may yield sub-optimal performance for some architectures. Trading accuracy and generality appropriately is critical to any such model.

In this paper, we study a number of architecture independent parallel programming models that have been proposed. We show the inter-relationships between these models in a comprehensive taxonomy based on characteristics that lead to important performance optimizations. Applications that do not exploit these characteristics may perform very poorly on many important parallel architectures. We do not propose another universal machine model based on arbitrary data redistributions. Instead, we recognize that a large number of scientific, engineering and computer science applications have been solved using collective communication operations [11, 7]. Based on this rich experience, we propose a strategy for algorithm analysis based on aggregate communication operations. A limited number of these aggregate communication operations cover the vast majority of high-performance applications. Application developers can simply design algorithms using these aggregate operations. The algorithms can be analyzed using a table lookup to determine the time taken for a specific aggregate operation on a specific architecture. In particular, developers can ignore the complexity and architecture-dependence of the aggregate operations' implementations. Furthermore, when problem size is sufficiently large, additional architecture-independence and simplicity may be obtained by using an approximation of our model. Our model for architecture-independent parallel program design is consistent with the functionality of MPI and HPF. A limitation of our model is that it constrains the parallel algorithm to be synchronous.

2 Overview of Conventional Parallel Computers

Evolutionary trends of parallel computers have converged to the general architecture illustrated in Figure 2. It consists of a set of processing elements connected through an interconnection network. Scalable architectures have not been able to support true shared memory. The memory is physically distributed across the processors. Interaction between processors is either through message passing or through a shared address space. Processing nodes consist of a processor with on chip cache, a second level cache and a large memory (typically ranging from 16 - 128 megabytes). The clock speeds range up to a several hundred MHz. Memory hierarchy plays a critical role at such clock rates. Locality of data reference even within a processor is necessary to achieve good per-processor performance. By optimally utilizing instruction and data prefetch queues and the functional units, processors are capable of instruction cycle times in the range of a few nanoseconds. Many issues that are critical to the design of parallel algorithms are also relevant for extracting good performance from individual processing elements.

The interconnection network of conventional parallel computers range from low degree networks such as 2-D meshes to high degree networks such as hypercubes. These belong to the general class of networks referred to as k:d meshes. Here, d refers to the dimension of the mesh and k refers to the number of processors along any dimension. A 2-D mesh is therefore a $\sqrt{p}$:2 mesh and a hypercube is a 2:$\log p$ mesh. We assume throughout this paper for simplicity that the meshes have wraparound connections. (It must be noted that the hypercubes do not have wraparound connections.) The corresponding cross-section bandwidth of these networks varies between $\Theta(\sqrt{p})$ for a 2-D mesh to $\Theta(p)$ for a hypercube. The cross-section bandwidth of the interconnection network is critical to the performance of some data communication patterns such as matrix transpose, FFT, and sorting. There is also a large class of communication patterns whose performance is not critically impacted by the cross-section bandwidth. Some important examples are one-to-all broadcast and its variants, all-to-all broadcast, and one-to-all personalized communications [11].

Low dimensional networks such as 2-D meshes are cheaper to construct than high dimensional networks such as hypercube (and related networks such as multistage networks and fat trees), but they have smaller bisection width. For many communication operations that are not bisection width insensitive, low dimension networks are more cost effective than hypercubes and related networks [11, 5].

Popular interconnection topologies are buses (SGI Challenge), 2D meshes (Paragon, Delta), 3D meshes (Cray T3D), hypercubes (nCUBE), fat tree (CM5), hierarchical networks (cedar, DASH).

2.1 The Communication Model

The communication delays arise from various sources. These delays can be categorized into three classes:

- Message size dependent delay: This delay is a consequence of copying the message into buffers and due to the finite capacity of the interconnection network. We capture these delays in a constant t_w, henceforth referred to as the per-word transfer time.

- Path length dependent delay: This delay is a consequence of the overhead at intermediate network interfaces and the signal propagation time on each link. This delay is represented by constant t_h, also referred to as the per-hop time.

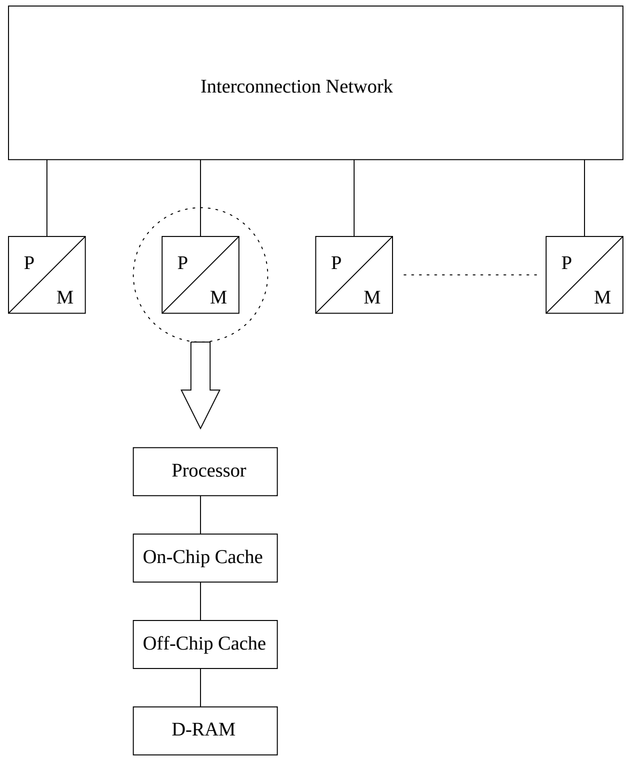

Figure 2: Organization of a conventional parallel computer.

- Startup latency: This delay is independent of message length and number of hops. It is incurred due to handshaking between processors, buffer management and routing. This is represented by t_s.

A message of size m traversing d hops of a cut-through (CT) routed network incurs a communication delay given by

$$T_{comm}^{CT} = t_s + t_h d + t_w m$$

The corresponding cost for a store-and-forward (SF) network is

$$T_{comm}^{SF} = t_s + (t_h + t_w m)d$$

Most conventional computers have cut-through routed networks. In the rest of this paper, we will restrict ourselves to cut-through networks only.

The startup time t_s is often large, and can be several hundred machine cycles or more. The per-word transfer time t_w is determined by the link bandwidth. Lower degree networks typically have higher link bandwidth (and consequently lower t_w). This is because the individual channels can be made fatter for the same overall cost [11, 5]. The per-word transfer time t_w is often higher than t_c, the time to do a unit computation on data available in the cache. The per-hop component $t_h d$ can often be subsumed into the startup time t_s without significant loss of efficiency. This is because the diameter of the network, which is the upper bound on distance d is relatively small for most practical sized machines, and t_h also tends to be small.

The above expressions adequately model communication time for lightly loaded networks. However, as the network becomes more congested, the finite network capacity becomes a bottleneck. Multiple messages attempting to traverse a particular link on the network are serialized. A good measure of the capacity of the network is its cross-section bandwidth (also referred to as the bisection width). This is defined as the minimum number of independent paths available between two equal partitions of the machine. For a hypercube, the bisection width is $p/2$, for a wraparound mesh $2\sqrt{p}$, and for a shared bus 1.

The values of constants associated with small fixed size messages can be somewhat different. For example, using active messages[18], it is possible to reduce the startup time for message transfer compared to arbitrary sized messages. This is because fixed size messages use simple buffer allocation protocols. However, each packet must now carry message tags and control information. This combined with interrupt overheads increase effective per-word transfer time t_w.

On shared address space systems, reading m consecutive words from a remote processor's memory also takes time $t_s + t_h d + t_w m$. Often, $t_h d$ is constant for all memory modules, and can be subsumed into t_s. The value of t_s tends to be much smaller than its counterpart in message passing systems. As before, the communication time corresponds to lightly loaded networks. If there are more memory transfers in progress than the bisection width of the network, the time for these operations will be higher.

3 Taxonomy of Parallel Algorithm Design Models

Conventional parallel computers reward certain algorithmic features and penalize others. Desirable features of parallel algorithms include seclusive memory access, minimum remote data access and bulk data access. In addition, the interconnection networks of some parallel computers also reward certain types of structured communication operations. Algorithm design models mirror these attributes of parallel computers to varying degrees (see Figure 3). This determines the accuracy and coverage of the model. In this section, we discuss various factors that differentiate the models and how they render models more or less realizable in the space of real machines.

3.1 Global Memory Access Versus Seclusive Memory Access

The strongest models of parallel algorithm design allow each word in a part of the memory to be addressed independent of all others (*i.e.*, assume a global p-ported memory). Access to a certain word in this global memory by a processor does not block access to any other word by another processor. Unfortunately, such a strong model comes with significant costs. If the total number of words in global memory is m and the number of processors is p, a completely non-blocking switch has a complexity $\Theta(mp)$. For reasonable values of m ($> 10^7$) and p (10^3), switch complexities of $\Theta(mp)$ are not currently realizable. Furthermore, the presence of processor caches imposes stronger requirements of the global memory (these implications will be touched upon later in this section). Therefore, models based on global p-ported memory take a much stronger view of the architecture than is realizable for reasonable machine sizes. Most PRAM models (PRAM, LPRAM [1], BPRAM [2]) assume a global memory.

The switch complexity can be reduced by organizing memory into banks (p banks of single-ported memories). When a processor is accessing a bank, access to all other words in the bank by other processors is blocked. If the switched memory is organized into b banks, the switch complexity of a non-blocking (at the bank level) switch is $\Theta(pb)$. This

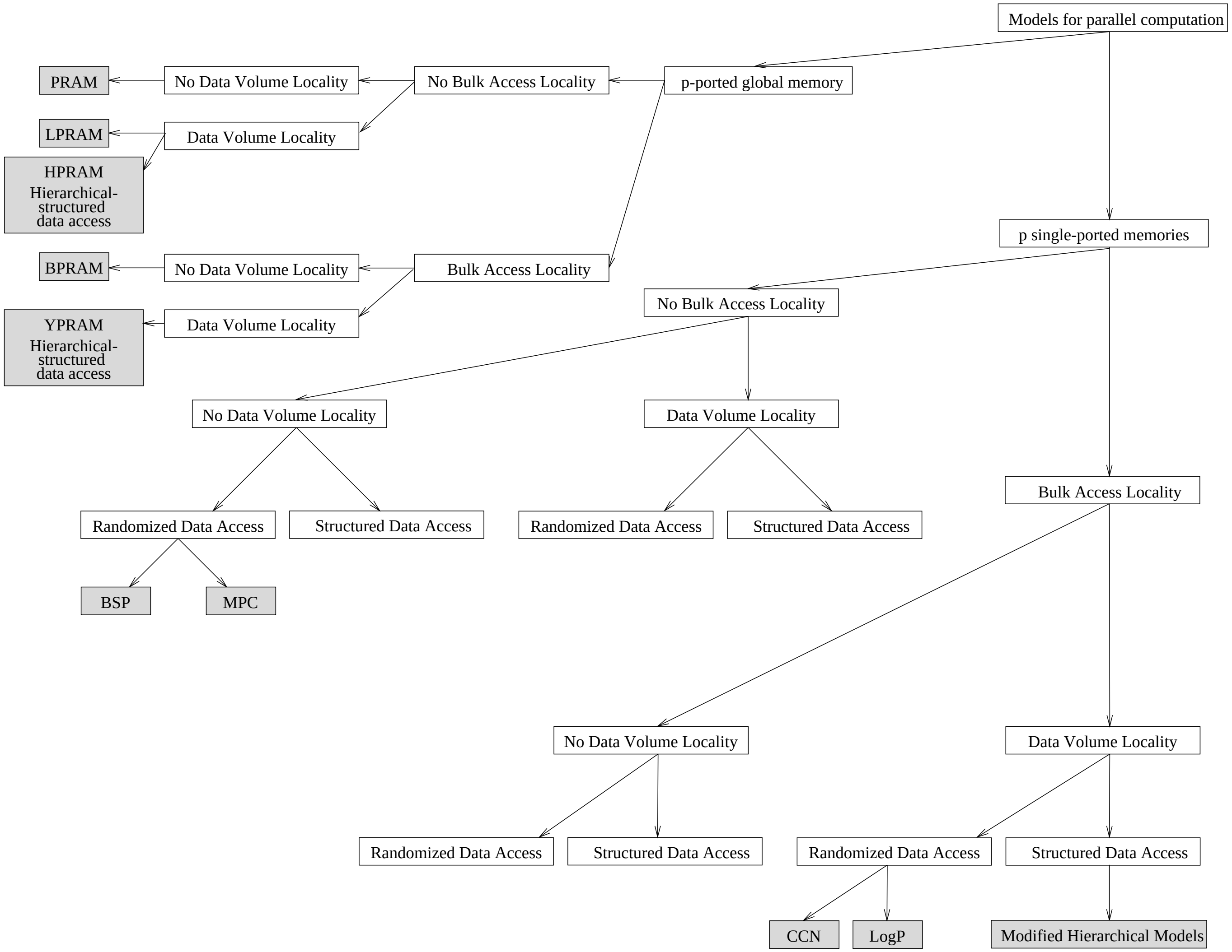

Figure 3: Taxonomy of models for parallel programming.

complexity is realizable at the current state of the art. Such a model is referred to as a Seclusive Memory model. When combined with appropriate communication cost models, seclusive memory models approximate machine architectures much better than global memory models. Algorithm design models based on seclusive memory models include MPC[12], BSP(XPRAM)[17], CCN, and LogP[4]. All practical parallel computers (including shared address space and message passing machines) use single ported memory and thus belong to the class of seclusive memory machines.

3.2 Bulk Access Locality

In conventional parallel computers, it takes a certain amount of time to access the first word of a message. Subsequent words of the message can typically be fetched much faster. This underscores the need for making data accesses in bulk. Bulk data access locality in a parallel algorithm is rewarded by the startup latency (t_s) term in the communication cost model. A dominant part of the startup latency is spent in packaging data, computing routing information, establishing protocols and programming routing hardware. This time varies significantly depending on the protocols used. Nevertheless, it is significantly higher than the processor cycle time in all cases. Although startup latency may not seem to be a major architecture parameter, it is in fact responsible in significant part for enforcing a degree of granularity in communication (and hence computation). Algorithms design models can be classified into those that reward bulk data access and those that do not. Models in the latter category are benign to algorithms that do not incorporate granularity. For such algorithms, the startup latency may have to be paid for each remote access on a real machine. Consequently, algorithms designed for such models perform poorly on real machines. Most of the current machines fall into the category of p single-ported memory machines with non-negligible startup latencies.

Bulk access locality plays an important role even for serial computers with memory hierarchy. Most processors use caches with line size greater than one word. In the event of a cache miss, an entire cache line is moved from the memory to the cache. Bulk access locality in a program implies that it is more likely that all the words in the line will be used.

3.3 Data Volume Locality

Access to words in non-local memory typically costs more than local memory. This indicates that the volume of non-local data accessed should be minimized. Data volume locality is rewarded by the per-word transfer time (t_w) in the communication cost model and overall bandwidth of the network. Such locality is particularly important for machines in which per-word transfer time of the communication link t_w is higher than local memory access time. In many other machines, the degree of connectivity of the interconnection network is limited. For such machines, the effective per-word transfer time for many communication operations must be scaled up to accommodate the limited cross-section bandwidth. Machines with t_w comparable to local memory access time, may have higher remote access times when compensated for cross-section bandwidth. Data volume locality is also important in such cases.

We have considered cases in which the startup latency t_s is non-negligible (requiring bulk access) and those in which t_w is higher than local memory access time. However, even when startup latency can be neglected and communication bandwidth is equal to local memory bandwidth, we need bulk access. This is because the total latency for remote access is determined by both network latency and memory latency. Masking memory latency at the level of uniprocessors (by using cache) requires bulk access equal to cache line size. The same bulk access will be required for remote access even in the absence of network latency for networks with identical communication and local memory bandwidths.

3.4 Structured Communication Operations

Data communication patterns in most applications have a definite structure. It is possible to exploit these structures by imposing these intelligently on to the underlying machine topology. It is therefore desirable that models reward parallel algorithms that induce structured communication patterns.

Typical applications can be viewed as a set of interacting subtasks executing on different processors. Interaction is by means of exchange of messages. It is often possible to formulate these tasks in such a way that the computation proceeds in interleaved steps of computation and communication. All processors (or subsets thereof) participate in the communication step. Since the communication involves more than a single pair of processors (aggregates or groups), such communication operations are also referred to as aggregate or collective communication operations. All parallel programs written in data-parallel languages such as HPF fall in this category. Programs based on MPI that use only aggregate communication operations also fall in this category.

It is possible to view the communication in a parallel algorithm at various levels of abstractions. In this section, we will investigate these levels of abstractions and their inter-relationships. All processors in a machine can be divided into logical non-overlapping groups or aggregates. We define the following communication operations within a processor group:

Aggregate Communication Operations: An aggregate communication operation is the most general communication operation. Each processor has a set of data items- each of which needs to be sent to a subset of processors. (Note that the data items can be of different sizes.) The total amount of data leaving a processor is called its *data fan-out*. The total amount of data entering a processor is called its *data fan-in*. Aggregate communication operations can be either personalized or broadcast operations.

Personalized Operations: In these operations, each data item has a unique destination processor associated with it. It is useful to classify personalized operations on the basis of the data fan-in and data fan-out. In the first category, every processor has identical data fan-out and data fan-in. Instances of personalized operations in this category include NEWS, general permutations, and all-to-all personalized communications. In the second category, no restriction is placed on the data fan-out and data fan-in. An example of such an operation is one-to-all personalized communication. These operations are described in detail in [11, 14].

Broadcast Operations: In these operations, each data item goes to all the processors in the group. Instances of these operations include one-to-all, k-to-all, and all-to-all broadcasts. These operations are described in detail in [11].

Data-redistribution Operation: In a data redistribution operation, each processor has one data item that needs to be communicated to another processor. Each data item has a unique destination processor. The size of the data items can vary. If all data items are of the same size, the operation is a general permutation. An interesting sub-class of general permutations is the k-shift operation. The k-shift is a shift of stride k. This operation forms the basis of the NEWS shift (where the stride is one and $\sqrt{p}$) and the Butterfly (where the stride is 2^i, for i varying from 0 to $\log p$).

Figure 4 illustrates the interrelationship between various communication operations. Arrows in the figure indicate that a lower entity is an instance of the higher entity. Successors of any node in the figure are neither necessarily exhaustive nor mutually exclusive.

It is easy to see that all communication operations can be designed as composites of the data redistribution operation (highlighted boxes in Figure 4). The underlying pattern of the redistribution operation can have critical impact on the runtime of the composite communication operation. For many aggregate operations, these patterns are very well defined and can be exploited for optimal performance. For instance, consider a NEWS operation on a 2-D mesh. The cost of this aggregate communication operation is less than a random communication across the network. If represented appropriately, it is possible to exploit the structure in these redistributions in a portable manner using aggregate communication libraries. Algorithms can be designed using operations higher in the hierarchy in Figure 4.

Most current parallel computers are capable of exploiting the structure of data-redistribution operations. However, many models ignore this structure. They assume that all redistributions take equal time. A very common assumption made is that if the maximum amount of data entering or leaving a processor in a particular redistribution is m then the communication time is proportional to m and is independent of the number of processors p. This can be a very misleading assumption. Consider a random permutation in which each processor

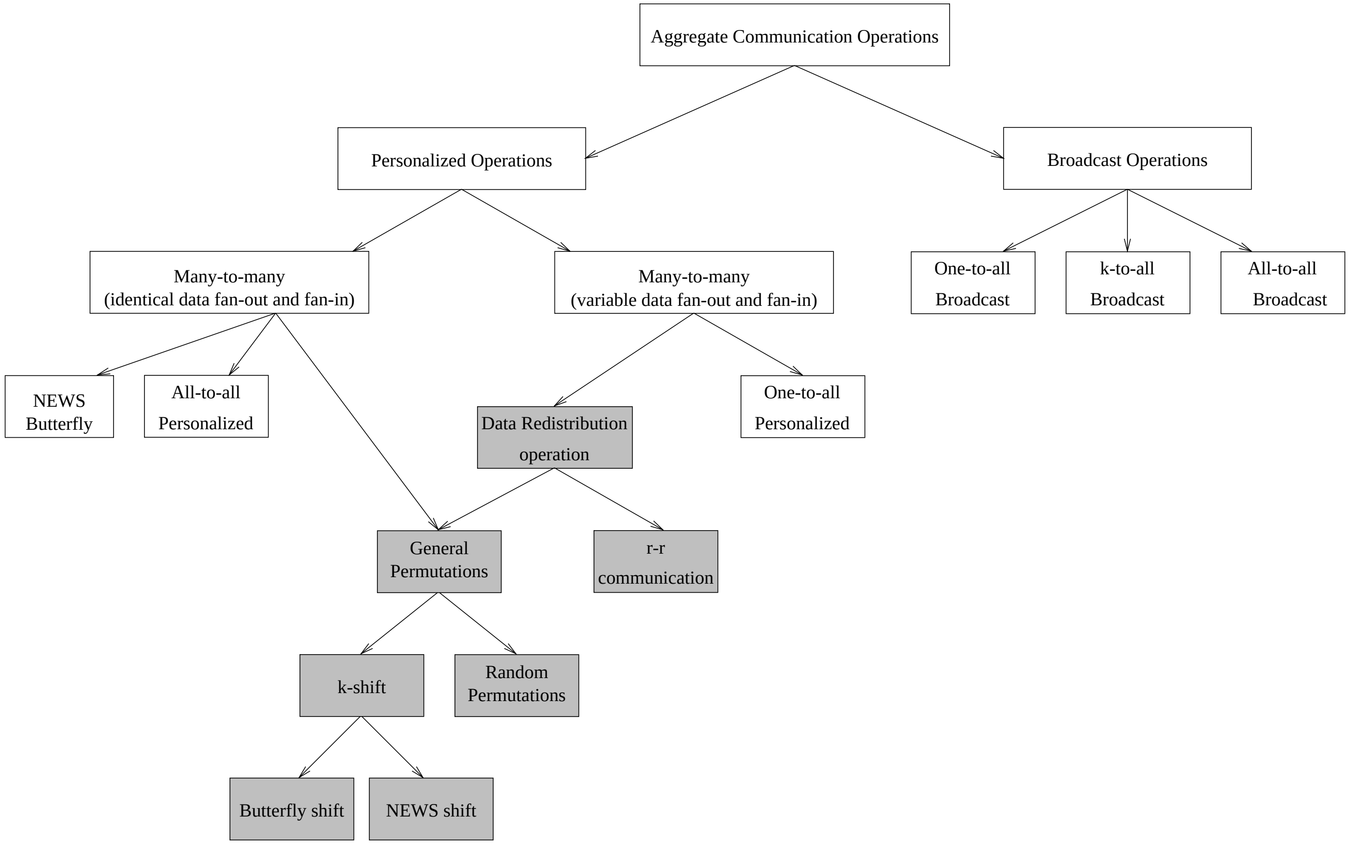

Figure 4: Communication operations and their interrelationships.

has a data item of size m which needs to be communicated to a unique processor. The basic prerequisite for executing a random permutation in time $\Theta(m)$ (independent of the number of processors p) is that the cross-section bandwidth of the network is $\Theta(p)$. Real machine topologies in this class are the hypercube, fat-tree, butterfly, and the crossbar network. It is possible to perform this operation in $\Theta(m)$ time only in the presence of slack and sufficient cross-section bandwidth [14, 15].

Models which assume that communication cost is independent of the number of processors and structure of communication are called equal cost redistribution models. All other models are called variable redistribution cost models. We can see that the accuracy of equal redistribution cost models can be poor and the class of real machines covered is very small.

Many communication operations can be implemented as a sequence of data redistributions. At each level in the hierarchy, data redistribution involves processors within clusters only. Thus, at the top level, the redistribution involves all processors. At the next level, the set of processors is partitioned into two parts. Redistributions can now involve processors in these two halves. Furthermore, it is assumed that redistributions within each level have no structure. Hierarchical models of parallel computers (YPRAM, HPRAM) are capable of accurately predicting the cost of these hierarchical sequence of data redistribution steps.

It is desirable that the programming model assumes p single-ported memories and rewards bulk access, data volume locality, and structure inherent in a large class of data access patterns.

4 Limitations of Existing Models for Parallel Programming

A variety of algorithm design models have been proposed. All of them are limited in one or more of the following aspects:

1. They are too strong, and hence impossible to realize at the current state of the art;

2. They are too specific and do not cover a broad enough class of architectures;

3. They have poor predictive powers.

In this section, we will study the limitations of some important models.

4.1 PRAM

Among the earliest and most extensively studied models of parallel computing is the PRAM model of computing. It falls into the class of global memory models. It assumes the startup latency to be zero and a bandwidth to global memory identical to local memory bandwidth. It is therefore the strongest model of parallel computers and also most difficult to realize in conventional architectures. Although shared-bus based architectures approximate the PRAM model for small number of processors (uncongested bus), the presence of caches puts locality constraints even for these architectures.

PRAMs are useful in extracting the maximum concurrency from algorithms. They are a natural progression from the serial RAM model of computers and are easier to design algorithms for than all the other models of parallel computers. However, they lack a cost model for communication. Data locality is therefore not an issue while designing PRAM algorithms. This is a major drawback since communication costs are often the dominant overhead in parallel algorithms. In fact, many PRAM algorithms are not even cost optimal when ported to real architectures.

Ranade [13] had earlier demonstrated how it is possible to simulate a p processor PRAM on a p processor seclusive memory machine with a butterfly interconnect. Ranade's technique, however, pays a $\log p$ cost for each instruction. This excess cost is often unacceptable. In addition to the $\log p$ cost, there are also constant overheads involved in the simulation. These render the simulation technique less useful in practice.

4.2 LPRAM

The LPRAM [1] model of computing is a global memory model. The model is based on the premise that non-local memory accesses are more expensive than local memory accesses. It therefore tries to minimize the volume of remote data accessed and rewards data volume locality.

LPRAM ignores the startup latency associated with communication in conventional machines. In doing so, it fails to reward bulk access locality. Furthermore, it does not reward structured data accesses. Algorithms designed for this model may potentially pay significant startup overheads and make poor use of available communication bandwidth. Architectural features assumed by the model (negligible startup overhead, global non-blocking memory) render it unrealizable in real parallel computers.

4.3 BPRAM

The BPRAM [2] model of computing is a global memory model. This model is based on the premise that startup communication latencies of conventional computers are often high. It therefore rewards bulk access of remote data (bulk access locality).

BPRAM assumes that the communication bandwidth is identical to local memory bandwidth. In doing so, it fails to capture data volume locality. Furthermore, like LPRAM, it does not exploit structures in underlying communication patterns and thus makes poor use of available communication bandwidth. Although it is possible to balance the communication and local memory bandwidth in conventional parallel computers, the presence of hierarchical memory structures on processors complicates this. Assuming a high cache hit ratio, the effective local memory bandwidth is in fact the cache bandwidth. The communication bandwidth must therefore match the bandwidth to cache in this model. This is difficult to realize unless communication is highly clustered.

Architectural assumptions such as non-blocking global memory and balanced communication and local memory bandwidth make this model difficult to construct.

4.4 Hierarchical PRAM models (HPRAM and YPRAM)

The HPRAM [10, 9] and YPRAM [6] models of computation allow for (recursively) decomposing the machine till the algorithm only uses one processor and executes sequentially.

For more than one processor, the communication cost charged is a function of the number of processors available. Assigning different cost to data redistributions in which communication is limited to smaller subsets of processors allow for designing algorithms which are more effective in utilizing locality especially for divide and conquer type applications. Despite the availability of variable cost redistributions, HPRAM [10, 6] suffers from limitations similar to the BSP model in terms of slack and worst case bandwidth requirements for every step. This is because HPRAM does not reward bulk access locality. Note that YPRAM does not suffer from this drawback since it rewards bulk access locality. Analyzing time requirements using these models is difficult for non-recursively decomposable algorithms. Further, important communication primitives such as NEWS are difficult to analyze in these models. Both HPRAM and YPRAM can accurately predict the cost of those aggregate communication operations that can be optimally implemented as a hierarchical sequence of data redistributions.

4.5 Bulk Synchronous Parallel

Valiant proposed the XPRAM [17] model that allows PRAM algorithms to run within a constant factor of optimal as long as there is sufficient slack. The XPRAM model forms the basis for Bulk Synchronous Parallelism (BSP). It assumes a seclusive memory and for efficient operation, assumes startup latency to be negligible and per-word access time to be similar to cycle time. Although such a model is not as difficult to construct as global memory models, the restrictions on startup latency and per-word transfer time are difficult to realize in real machines. The main feature of BSP is that it allows simulation of $p \log p$ PRAM processors on p seclusive memory processors in asymptotically the same time as on p PRAM processors. Recall that a p processor PRAM can be emulated on a p processor seclusive memory machine (with a butterfly interconnect) incurring a $\log p$ cost for each instruction [13]. This emulation does not require any slack.

Although BSP [16] is an attractive theoretical model, it has several drawbacks. The cost of making a bulk access of h words by a processor in a superstep is assumed to be $s + g'h$.

Here, s is the startup cost and g' is the effective per-word transfer time. The startup cost s in the expression is not the same as the startup latency t_s of the network. In a single superstep, a processor may be accessing $h = \log p$ words each from a different processor. This implies that if the messages are sent directly, the startup latency t_s must be paid $\log p$ times. This time must be factored into the parameter g'. Indeed, theoretically it is possible to do so. However, as we had discussed earlier that the startup latency t_s is typically at least several machine cycles (for many practical machines it is several hundred machine cycles). For efficient simulation, the cost of realizing a h-relation must be comparable to the cost of h local memory accesses. Most conventional processing nodes have hierarchical memories. Assuming a high cache hit rate, the effective local memory access time is in fact the time to access a piece of data in the cache (and not the memory). Clearly, time $s + g'h$ cannot be in the range of time required for h cache accesses. To remedy this situation, the model must amortize startup latency t_s by requesting more than one word per processor. This indicates the need for greater slack. Let us assume that if k words are accessed for each startup cost paid, it is possible to mask the startup latency. In the original model, it was enough to access $\log p$ words. However, if $k \log p$ words were accessed, it is likely that not all the words will reside in the same $\log p$ processors. Since other processors have also been requested, the startup cost must be paid (and hence masked) for these accesses. Clearly, the slack required to mask the startup cost is substantially higher than $\log p$. As an example, if a total of $p \log p$ random memory requests were made by each processor, each processor would receive $\Theta(p \log p)$ requests on an average. If access to $\log p$ words at a processor is adequate for masking startup latency, then a slack of $p \log p$ is required for a meaningful BSP simulation. Note that this slack does not alleviate the problem associated with caches since the $\log p$ words accessed from a processor will not be contiguous. Furthermore, even with this slack, there will be significant loss in efficiency resulting from constant overheads. The slack can be reduced significantly by using alternate models that reward bulk access locality.

4.6 CCN and LogP Models

The Completely Connected Network (CCN) model of computing is a seclusive memory model. Although CCN has never been formally proposed as a model for architecture independent programming, its variants have been used extensively [3, 11, 7]. It assumes the presence of p processing elements connected over a completely connected network. The cost of sending a message of size m from one processor to any other is given by $t_s + t_w m$. Here, t_s is the startup latency and t_w is the bandwidth term. It therefore rewards bulk access and data volume locality. The startup latency t_s tends to be large if processors have to exchange arbitrary sized messages. As mentioned in Section 2 this time arises from buffer management, establishing protocols, routing, and programming communication hardware. In fact, on many machines, t_s is as high as thousands of machine cycles. However, on many architectures, small, fixed size messages can be sent with a small overhead using active messages. This forms the basis for the LogP model of computation. In the LogP model processors communicate using small fixed size messages of size m (or pipelined aggregates thereof). The model rewards bulk access at the level of small messages and data volume locality (using parameter g).

Larger messages are sent in LogP by breaking them into smaller messages and pipelining them. For each of the smaller messages, the processor incurs an overhead of o at the sending end. The message can then be injected into the network. Subsequent messages can be injected at gaps of g. Here, g is determined by the network bandwidth. If a sequence of short messages need to be sent, the gap of the first message can be overlapped with the preparation overhead o of the second message. Depending on values of g and o, two situations arise. In the first case $g > o$: in this case, the sending processor can prepare the messages faster than the network can accept them. In this case, the network bandwidth will not be wasted. In the second case $o > g$: in this case, the processor cannot inject messages into the network fast enough to utilize available bandwidth. Therefore, it is desirable that the size of the small fixed message m be such that $g > o$. Note that m is the fourth parameter of the LogP model (that is not often mentioned). The LogP model rewards bulk access of up to m words.

Messages can be injected into the network at intervals of g. Of this time, o is spent by the processor preparing the next message. The remaining time $g - o$ can potentially be used to perform useful computation. It can be seen that the processor overhead associated with sending a long message grows linearly with message size. This implies that the message size m should be as large as possible. The fixed message size m must also be a parameter of the model since it impacts the performance of the algorithm on real machines.

The CCN model does not suffer from either of these drawbacks. Since the startup cost is paid just once for the message, it is able to utilize available bandwidth completely. Furthermore, the processor overhead is independent of the message size. Therefore, it is possible to overlap the entire message transit time $t_w m$ with useful computation.

In addition to these, both models suffer from certain drawbacks. These stem from their inability to exploit structure in underlying communication patterns. Neither model is capable of distinguishing benign data redistributions from those that congest the network. They assume that all redistributions can be accomplished in time proportional to message size per processor. Let us study the implications of this assumption closely. Assume that each processor has a data item of size m that it needs to communicate to a unique, randomly chosen destination (random permutation). Both models assume that this permutation can be performed in $\Theta(m)$ time. However, there are no known algorithms that can perform the permutation in this time even on $\Theta(p)$ cross-section bandwidth networks such as hypercubes without slack. (The transportation primitive of Shankar and Ranka [14, 15] performs this permutation in $\Theta(m)$ but it requires a slack of $\Theta(p^2)$.) Therefore, the cost model is a lower bound for data redistributions on all architectures. Now consider an adaptation of these models for a mesh. The effective bandwidth is scaled down by the cross-section bandwidth to account for the limited bandwidth of the mesh while performing generalized redistributions. However, aggregate communication operations such as NEWS and broadcasts consist entirely of redistributions that do not congest the network and are therefore capable of extracting the same network performance from a mesh as from a hypercube. In these cases, the communication cost of both models is an upper bound[1]. The communication time predicted by the models is therefore neither an upper bound nor a lower bound. When applied to communication operations such as broadcast and NEWS, the time predicted by LogP is an upper bound since it assumes worst case bandwidth utilization (all messages cross the bisection). However, when applied to some bandwidth sensitive operations (such as the transportation primitive), the time predicted by is actually a lower bound. To remedy these problems the LogP model itself suggests use of different values of gap g for different communication patterns. This reflects the motivation for our model of algorithm design.

5 A Model for Architecture Independent Programming and Analysis

In this section, we propose a model for parallel programming based on abstraction of communication inherent in the algorithm. Many existing models incorporate bulk access locality and data volume locality. Their main drawback has been their inability to identify favorable data redistributions and their costs. In our model, we remedy this by classifying redistributions into categories and assigning appropriate costs to them on k:d cubes.

Figure 4 illustrates the problem of parallel programming and analysis. Whereas it is desirable to program higher in the hierarchy for portability, it is important to look lower in the hierarchy for analyzing algorithms. LogP goes as far down as the data-redistribution operation and ignores the rich hierarchy underneath it. One solution is a library based approach. Since the total number of commonly used aggregate communication operations is limited, it is possible to package these into libraries and use a table to look up their communication costs. This forms the basis of our model.

We assume in our model that the programming paradigm is synchronous. Each pro-

[1]The LogP model allows for using different g for different communication patterns but it does not specify how to do so.

cessor goes through phases of local computation followed by communication for non-local data. At any given snapshot, all processors are in the same phase (*i.e.*, computing or communicating). Communication is in the form of aggregate communication operations. As has been noted earlier, most parallel algorithms are comprised of a small set of aggregate communication operations. We have identified the following set of aggregate communication operations that appear most frequently: many-to-many broadcast (and their dual - reduce, and variations - scan), many-to-many personalized, k-shifts, 2-D grid based communication - NEWS (and its generalization to $\log p$ dimensions - butterfly). For a more detailed description of these operations, the reader is referred to Kumar et al. [11] and Shankar and Ranka [14, 15]. Personalized communication appears in such applications as matrix transpose, hash joins, and fast fourier transforms. Broadcasts and scans are ubiquitous. NEWS communication is required in finite difference and finite element methods, and a variety of image processing applications.

The aggregate communication operations identified above can be expressed in the form of composites of data-redistribution operations. The performance properties of aggregate communication operations can be ascertained from the underlying redistribution operations. We use this to develop cost models for quantifying the communication overhead of parallel algorithms. We present two cost models: an accurate cost model, and a simpler approximate cost model that can be applied in the presence of adequate slack. Slack is defined as the total number of data units per-processor participating in the aggregate communication operation. We will demonstrate in Section 6 the use of these models for performance prediction. We show that the approximate cost model is adequate in the presence of slack and yields very good bounds on the performance.

5.1 Accurate Cost Model

The accurate cost model is based on a tabulation of the aggregate communication operations and their exact cost on each architecture. The analysis is based on a table-lookup for the appropriate operation-architecture combination. The runtime of the entire parallel algorithm can be computed in this manner. Table 1 tabulates the various aggregate communication operations along with their cost on k:d cubes. Here, t_s is the startup latency, t_h is the per-hop time and t_w is the per-word transfer time.

5.2 Two-Level Approximate Model

A closer examination of the above table yields interesting insights. The space of data-redistribution operations can be partitioned into two parts: operations sensitive to the cross-section bandwidth, and those insensitive to the cross-section bandwidth. Sensitivity to cross-section bandwidth is determined by the volume of data and the redistribution pattern. Generalized data-redistribution operations (*eg.*, random permutations) and some structured redistributions (*eg.*, k-shifts) are cross-section bandwidth sensitive. This forms the basis of the approximate cost model. In the presence of adequate slack, it is possible to approximate the cost of various communication operations. The slack required is different for different algorithms and architectures. A slack of $\Theta(p)$ is adequate for all cases illustrated in Table 1. In many of these cases, a lower slack is adequate. For *eg.*, one-to-all broadcast on a mesh needs $\Theta(\sqrt{p})$ slack, and on a hypercube, it needs $\Theta(\log(p)$ slack. In the presence of this slack, the communication times can be approximated as follows:

Aggregate Communication Operations Sensitive to Cross-Section Bandwidth Aggregate communication operations such as many-to-many personalized and k-shifts are based on cross-section bandwidth sensitive redistributions. Let m be the maximum amount of data leaving or entering any processor and c be the cross-section bandwidth of the network. Each link in the network has an effective bandwidth of $1/\mu$. This corresponds to a per-word link time of μ seconds / byte. Then the time taken by any aggregate communication operation in this class is given by

$$T_{comm} = \mu m(p/c) \tag{1}$$

Operation	Linear Array	2-D Mesh (wraparound, square)	Hypercube
One-to-all broadcast	$(t_s + t_w m)\log p$ $+t_h(p-1)$	$(t_s + t_w m)\log p$ $+2t_h(\sqrt{p}-1)$	$(t_s + t_w m)\log p$
One-to-all broadcast with $\Theta(p)$ slack	$2(t_s p + t_w m)$	$2(2t_s\sqrt{p} + t_w m)$	$2(t_s \log p + t_w m)$
All-to-all broadcast	$(t_s + t_w m)(p-1)$	$2t_s(\sqrt{p}-1) + t_w m(p-1)$	$t_s \log p + t_w m(p-1)$
One-to-all personalized	$(t_s + t_w m)(p-1)$	$2t_s(\sqrt{p}-1) + t_w m(p-1)$	$t_s \log p + t_w m(p-1)$
All-to-all personalized	$(t_s + t_w mp/2)(p-1)$	$(2t_s + t_w mp)(\sqrt{p}-1)$	$(t_s + t_w m)(p-1)$ $+(t_h/2)p\log p$
Circular *q*-shift	$(t_s + t_w m)\lfloor p/2 \rfloor$	$(t_s + t_w m)(2\lfloor\sqrt{p}/2\rfloor + 1)$	$t_s + t_w m$ $+t_h \log p$

Table 1: Various communication operations and their cost on a k:d cube.

Specifically, for a k:d cube, the cross-section bandwidth is given by $c = p^{\frac{d-1}{d}}$. Substituting the value of c in Equation 1, we get

$$T_{comm} = \mu m p^{\frac{1}{d}} \tag{2}$$

Equation 2 gives the time for a aggregate communication operation in this class for a k:d cube.

Aggregate Communication Operations Insensitive to Cross-Section Bandwidth Aggregate communication operations such as many-to-many broadcast and NEWS belong to this class (NEWS is defined as 2-D grid structured communication. The communication operation is bandwidth insensitive on the family of k:d meshes). The cost of these operations is largely independent of the number of processors. If m is the maximum amount of data entering or leaving any processor, the time for these operations is given by:

$$T_{comm} = \mu m \tag{3}$$

A parallel algorithm can be analyzed for a k:d cube by using these two communication costs applied to appropriate aggregate communication operations.

In both these cases, the approximate model ignores the impact of the startup latency and the per-hop time. This restricts the applicability of the model to cases where there is adequate slack. In other cases, it yields lower bounds on the communication time for the operation.

6 Analysis of Algorithms

In this section, we summarize results of analysis of a variety of parallel algorithms. We compare the approximate and LogP models to the accurate model. We show that in each of the cases the approximate model yields accurate asymptotic estimates of communication overhead. The LogP model fails to do so for the mesh for all the algorithms. Furthermore, although it may seem that the LogP model gives accurate runtimes for the hypercube, it gives no hints about placement of tasks on processors. Randomly placing subtasks on processors may cause contention over the network and degradation of performance and worse runtimes. Detailed analysis of these algorithms is presented in Appendix A. From the appendix, it also becomes clear that our models are much simpler to analyze algorithms for. Table 2 presents communication overheads predicted by each model for Fast Fourier Transforms (FFT), matrix-matrix product, sample sort and Gaussian elimination.

Algorithm	Hypercube			Mesh		
	Approx	Exact	LogP	Approx	Exact	LogP
FFT	$\mu \frac{n}{p} \log p$	$t_s \log p + t_w \frac{n}{p} \log p$	$\log p(L + t_w n/p)$	$2\mu n/\sqrt{p}$	$t_s \log p + 2t_w \frac{n}{\sqrt{p}}$	$\log p(L + t_w n/\sqrt{p})$
Matrix Product	$2\mu \frac{n^2}{\sqrt{p}}$	$2(t_s \log p + t_w \frac{n^2}{\sqrt{p}})$	$2(L \log p + t_w \frac{n^2}{\sqrt{p}})$	$2\mu \frac{n^2}{\sqrt{p}}$	$2t_s\sqrt{p} + 2t_w \frac{n^2}{\sqrt{p}}$	$2(L \log p + t_w n^2)$
Sample sort	$\mu p \log^2 p$ $+2\mu n/p$	$\log^2 p(t_s + t_w p)$ $+2t_s p + 4t_w n/p$	$\log^2 p(L + t_w p)$ $+Lp + 2t_w n/p$	$2\mu p^{1.5}$ $+\mu 2n/\sqrt{p}$	$2(t_s\sqrt{p} + t_w p^{1.5})$ $+4t_s\sqrt{p} + 4t_w n/\sqrt{p}$	$\log^2 p(L + t_w p^{1.5})$ $+Lp + 2t_w n/\sqrt{p}$
Gauss. Elim.	$2\mu n/\sqrt{p}$	$4(t_s \log p + t_w n/\sqrt{p})$	$4(L \log p + t_w n/\sqrt{p})$	$2\mu n/\sqrt{p}$	$4(2t_s\sqrt{p} + t_w n/\sqrt{p})$	$4(2L + t_w n)$

Table 2: A comparison of communication overheads predicted by the approximate, accurate and LogP models. It is clear that the approximate model predicts the overhead accurately in asymptotics for every algorithm. The LogP model however fails to do so for the mesh architecture for every algorithm.

7 Concluding Remarks

Algorithm design based on a small set of collective communication operations offers several advantages:

- Algorithms are portable across a variety of platforms with a minimal loss of efficiency.
- The algorithm design model hides the complexity of writing efficient communication routines for different machines from the user.
- Programming parallel machines using this paradigm is much simpler. The analogue in serial computers is one of programming based on libraries as opposed to programming in assembly.
- The model brings the algorithm design process closer to the functionality of MPI and HPF.
- The model provides a means of determining good data partitioning and placement strategies. Data placement and the communication operations they induce together determine the figure of merit for an algorithm. The key to designing efficient algorithms is to identify favorable combinations of the two. By limiting the number of communication operations, the model prunes the space of possible algorithms incurring minimal loss in efficiency.

Constraining a parallel algorithm to be a synchronous parallel algorithm may potentially cause some loss in efficiency. There are also some algorithms that are entirely asynchronous in nature and these algorithms are difficult to express in our framework. Examples of such applications are in discrete event simulation and branch-and-bound tree search. For these algorithms, the analysis will have to be performed at the level of individual messages.

The approximate model distinguishes the bandwidth sensitive and insensitive operations and assumes identical communication costs for operations in each category. Although it ignores the startup costs, it provides a quick and easy analysis technique that works for most algorithms.

Appendix A: Analysis of Algorithms

Here, we present detailed analysis of the parallel algorithms considered in Section 6. We assume knowledge of the algorithms used for basic communication operations. Whereas this knowledge is not necessary for analyzing algorithms in our framework, it is necessary for analyzing them for the LogP model. The reader is referred to [11] for detailed descriptions of these analyses.

7.1 Fast Fourier Transforms

There are two main parallel formulations of the Cooley and Tukey Fast Fourier Transform: binary-exchange and transpose based algorithm. In this section, we analyze the binary exchange algorithm in detail.

7.1.1 Performance on a Hypercube

Accurate Analysis Consider implementing the binary-exchange algorithm to compute an n-point FFT on a hypercube with p processors, where $n > p$. Assume that both n and p are powers of two. As Figure 5 shows, we partition the sequences into blocks of n/p contiguous elements and assign one block to each processor. Assume that the hypercube is d-dimensional (that is, $p = 2^d$) and $n = 2^r$.

Figure 5 shows that elements with indices differing in their d (= 2) most significant bits are mapped onto different processors. However, all elements with indices having the same d most significant bits are mapped onto the same processor. An n-point FFT requires $r = \log n$ iterations. In the m^{th} iteration, elements with indices differing in the m^{th} most significant bit are combined. As a result, elements combined during the first d iterations reside on different processors, and pairs of elements combined during the last $(r - d)$

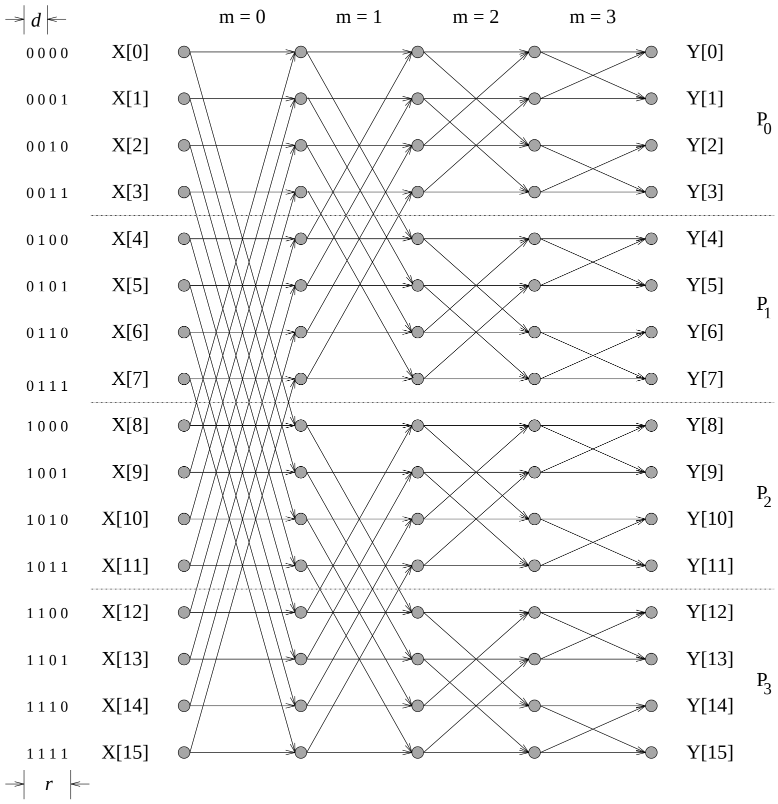

Figure 5: A 16-point FFT on four processors. P$_i$ denotes the processor labeled i. In general, the number of processors is $p = 2^d$ and the length of the input sequence is $n = 2^r$.

iterations reside on the same processors. Hence, this parallel FFT algorithm performs interprocessor communication only during the first $d = \log p$ of the $\log n$ iterations. There is no communication during the last $r - d$ iterations. Furthermore, in i^{th} of the first d iterations, all the elements that a processor requires come from exactly one other processor—the one whose label is different in the i^{th} most significant bit.

Each communication operation exchanges n/p words of data. Since all communication takes place between directly-connected processors, the total communication time does not depend on the type of routing. Thus, the time spent in communication in the entire algorithm is $t_s \log p + t_w(n/p) \log p$. A processor updates n/p elements of R during each of the $\log n$ iterations. If a complex multiplication and addition pair takes time t_c, then the parallel run time of the binary-exchange algorithm for n-point FFT on a p-processor hypercube is

$$T_P = t_c \frac{n}{p} \log n + t_s \log p + t_w \frac{n}{p} \log p. \tag{4}$$

Analysis Using Approximate Model The basic communication operation inside an FFT is a butterfly. The amount of data at each step is n/p. This operation takes time $\mu\frac{n}{p}\log p$.

Analysis Using LogP Model Assume $g > o$ and that messages can be pipelined in the form of short messages. The butterfly communication pattern is accomplished by $\log p$ permutations each of n/p data. The time for this operation is given by $\log p(L + gn/p)$. The gap g for a hypercube is determined by the link bandwidth t_w. Therefore, the communication overhead is $\log p(L + t_w n/p)$.

Comments We have assumed that neither model allows for overlapping computation and communication. Indeed, it is possible to overlap the t_w term in the accurate model with useful computation. The overlap in LogP is however limited to $g - o$ for each short message sent. All three models are able to correctly predict the scalability as measured by the isoefficiency function[2].

7.1.2 Performance on a 2-D Mesh

Accurate Analysis Assume that an n-point FFT is computed on a p-processor mesh with $\sqrt{p}$ rows and $\sqrt{p}$ columns, and that $\sqrt{p}$ is a power of two. Let $n = 2^r$ and $p = 2^d$. Also assume that the processors are labeled in a row-major fashion and that the data are distributed in the same manner as for the hypercube; that is, an element with index $(b_0 b_1 \cdots b_{r-1})$ is mapped onto the processor labeled $(b_0 \cdots b_{d-1})$.

As in case of the hypercube, communication takes place only during the first $\log p$ iterations between processors whose labels differ in one bit. However, unlike the hypercube, the communicating processors are not directly connected in a mesh. Consequently, messages travel over multiple links. Figure 6 shows the messages sent and received by processors 0 and 37 during an FFT computation on a 64-processor mesh. As the figure shows, proces-

[2]Isoefficiency is a scalability metric. It relates the rate of growth of problem size to the number of processors for maintaining constant efficiency. For a detailed discussion on the isoefficiency metric, the reader is referred to [11, 8]

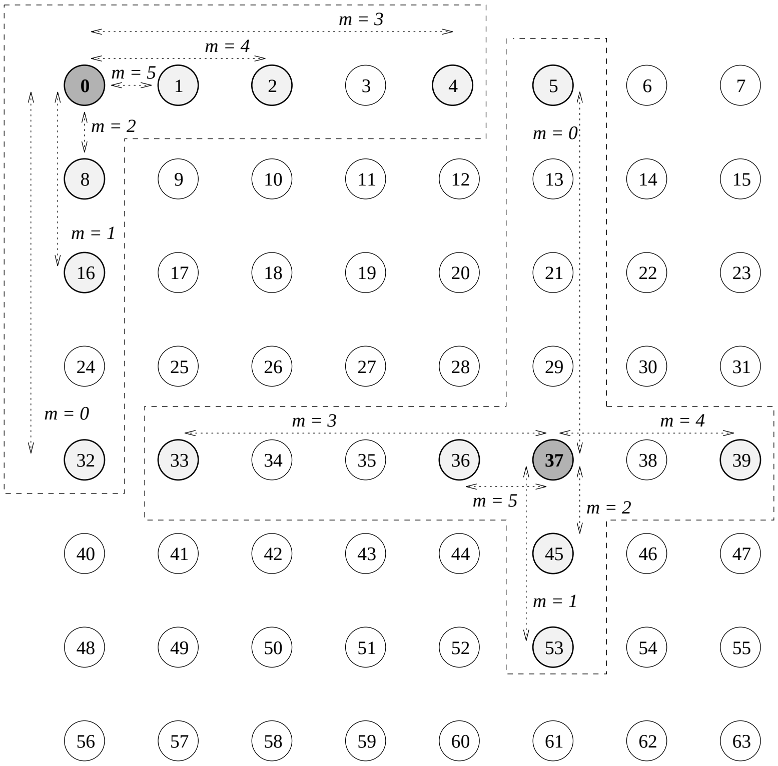

Figure 6: Data communication during an FFT computation on a square mesh of 64 processors. The figure shows all the processors with which the processors labeled 0 and 37 exchange data.

sor 0 communicates with processors 1, 2, 4, 8, 16, and 32. Note that all these processors lie in the same row or column of the mesh as that of processor 0. Processors 1, 2, and 4 lie in the same row as processor 0 at distances of 1, 2, and 4 links, respectively. Processors 8, 16, and 32 lie in the same column, again at distances of 1, 2, and 4 links. More precisely, in $\log\sqrt{p}$ of the $\log p$ steps that require communication, the communicating processors are in the same row, and in the remaining $\log\sqrt{p}$ steps, they are in the same column. The distance between the communicating processors in a row or a column grows from one link to $\sqrt{p}/2$ links, doubling in each of the $\log\sqrt{p}$ steps. The same relationship holds for any processor in the mesh, such as processor 37 shown in Figure 6. Thus, the total time spent in performing row-wise communication is $\Sigma_{m=0}^{d/2-1}(t_s + t_w(n/p)2^m)$ for a mesh with store-and-forward routing. An equal amount of time is spent in column-wise communication. Recall that we assumed that a complex multiplication and addition pair takes time t_c. Since a processor performs n/p such calculations in each of the $\log n$ iterations, the overall parallel run time is given by the following equation:

$$\begin{aligned} T_P &= t_c \frac{n}{p} \log n + 2 \sum_{m=0}^{d/2-1} (t_s + t_w \frac{n}{p} 2^m) \\ &= t_c \frac{n}{p} \log n + 2(t_s \log \sqrt{p} + t_w \frac{n}{p} (\sqrt{p} - 1)) \\ &= t_c \frac{n}{p} \log n + 2(t_s \log \sqrt{p} + t_w \frac{n}{p} (\sqrt{p} - 1)) \\ &\approx t_c \frac{n}{p} \log n + t_s \log p + 2t_w \frac{n}{\sqrt{p}} \end{aligned} \tag{5}$$

Analysis Using Approximate Model The butterfly communication on the mesh is a k-stride NEWS communication (in two directions), where k varies as $1, 2, 4, ..., \sqrt{p}/2$. A stride-k NEWS operation can be viewed as k stride-one NEWS operations. Therefore, the time for this operation in a single direction is given by:

$$\begin{aligned} T_{comm,approx,mesh} &= \sum_{i=0}^{\log(p/2)-1} 2^i \mu \frac{n}{p} \\ &= \mu \frac{n}{p} \sum_{i=0}^{\log(p/2)-1} 2^i \\ &= \mu \frac{n}{\sqrt{p}} \end{aligned} \tag{6}$$

The time for communication along the second dimension is identical to this. Therefore, the total communication predicted by the approximate model for the mesh is given by $2\mu n/\sqrt{p}$.

Analysis Using LogP Model The time for the LogP model remains the same as before *i.e.*, $\log p(L + gn/p)$. However, the gap g must be scaled up by the cross-section bandwidth factor. Therefore $g = t_w \sqrt{p}$, and the communication time given by $\log p(L + t_w n/\sqrt{p})$. The communication time is larger by a $\log p$ factor compared to that predicted by the accurate model.

Comments The approximate model predicts the performance of the algorithm correctly in asymptotics. For a mesh, the LogP model overestimates the runtime of parallel FFT.

The above discussion is applicable to the binary exchange algorithm, which is the best algorithm when the granularity of computation available per processor is small or if t_s is high. For large problem sizes ($n > p^2$) one can use a transpose based algorithm. There are three steps in this algorithm:

1. Perform a local FFT
2. Transpose
3. Perform a local FFT

Since transpose requires and all-to-all personalized communication and utilizes the cross section bandwidth fully, the approximate model and log P model will give the same results as direct analysis for both hypercube and mesh.

7.2 Dense Matrix-Matrix Product

Consider two $n \times n$ matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ $(0 \leq i, j < \sqrt{p})$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each. These blocks are mapped onto a $\sqrt{p} \times \sqrt{p}$ logical mesh of processors. The processors are labeled from $\mathrm{P}_{0,0}$ to $\mathrm{P}_{\sqrt{p}-1,\sqrt{p}-1}$. Processor $\mathrm{P}_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix. Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < \sqrt{p}$. To acquire all the required blocks, an all-to-all broadcast of matrix A's blocks is performed in each row of processors, and an all-to-all broadcast of matrix B's blocks is performed in each column. After $\mathrm{P}_{i,j}$ acquires $A_{i,0}, A_{i,1}, \ldots, A_{i,\sqrt{p}-1}$ and $B_{0,j}, B_{1,j}, \ldots, B_{\sqrt{p}-1,j}$, it performs the submatrix multiplications and additions to compute $C_{i,j}$.

7.2.1 Performance on a Hypercube

Accurate Analysis Assume that the logical mesh of processors is embedded into a p-processor hypercube. The algorithm requires two all-to-all broadcast steps (each consisting of $\sqrt{p}$ concurrent broadcasts in all rows and columns of the processor mesh) among groups of $\sqrt{p}$ processors. The messages consist of submatrices of n^2/p elements. The total communication time is $2(t_s \log p + t_w(n^2/p)(\sqrt{p}-1))$ for the hypercube. After the communication step, each processor computes a submatrix $C_{i,j}$, which requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices. This takes a total of $\sqrt{p} \times (n/\sqrt{p})^3 t_c = n^3/pt_c$ time. Thus, the parallel run time for multiplying two $n \times n$ matrices using this algorithm on a p-processor hypercube is approximately

$$T_P = \frac{n^3}{p} t_c + 2t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}. \tag{7}$$

Analysis Using Approximate Model As noted earlier, all-to-all broadcast is not a bandwidth sensitive operation. Therefore, time taken is μm, where m is the maximum amount of data entering or leaving any processor. For matrix multiplication, each all-to-all broadcast along the columns or rows requires $\frac{n^2}{p} \times \sqrt{p} = \frac{n^2}{\sqrt{p}}$ data to enter each processor. Therefore,

$$T_P = \frac{n^3}{p} t_c + 2\mu \frac{n^2}{\sqrt{p}}$$

Analysis Using LogP Model Assume that $g > o$ making it possible to pipeline messages. The analysis for the LogP model is identical to that for the accurate model. The communication overhead is given by $2(L \log p + g\frac{n^2}{\sqrt{p}})$. For a hypercube g is determined by the link bandwidth t_w. Therefore, the communication overhead predicted by LogP is $2(L \log p + t_w \frac{n^2}{\sqrt{p}})$.

Comments Both LogP and the approximate model predict the communication overhead of the parallel algorithm accurately in asymptotic terms. The accurate and approximate models are however much simpler to use. The LogP model actually requires knowledge of the all-to-all broadcast algorithm to derive the communication time.

7.2.2 Performance on a 2-D Mesh

Accurate Analysis If we use a mesh instead of a hypercube, only the term associated with t_s in the parallel run time (Equation 7) of this matrix multiplication algorithm is affected.

On a wraparound mesh with store-and-forward routing, each all-to-all broadcast among the $\sqrt{p}$ processors of a row or column of the mesh takes approximately $t_s\sqrt{p} + t_w n^2/\sqrt{p}$ time. Thus, the total parallel run time is

$$T_P = \frac{n^3}{p} t_c + 2t_s\sqrt{p} + 2t_w \frac{n^2}{\sqrt{p}}. \tag{8}$$

Analysis Using Approximate Model Since the all-to-all broadcast is not a bandwidth sensitive operation, the approximate analysis for 2-D mesh is the same as the approximate analysis for the hypercube. Therefore, the predicted communication overhead is still $2\mu\frac{n^2}{\sqrt{p}}$.

Analysis Using LogP Model Again, assume that $g > o$ making it possible to pipeline messages. As before, the communication overhead predicted by the LogP model is $2(L \log p + g\frac{n^2}{\sqrt{p}})$. The gap g must however be scaled up to compensate for the lower bisection width of the mesh. Therefore $g = t_w\sqrt{p}$. The communication overhead predicted by the LogP model is therefore $2(L \log p + t_w n^2)$.

Comments Whereas the approximate model predicts the communication overhead accurately in asymptotics, the LogP model overestimates it by a factor of $\sqrt{p}$. This is because it assumes that all-to-all communication is a bandwidth sensitive operation, which is not the case.

7.3 Sample Sort

Sample sort is a commonly used technique for sorting keys [11]. This technique is based on partitioning the list into buckets that can be independently sorted. Since the distribution of particles is a-priori unknown, it is difficult to determine evenly spaced splitters. Sample sort selects a sample of size s from the n-element sequence, and the range of the buckets is determined by sorting the sample and choosing $m-1$ elements from the result. The performance of sample sort depends on the sample size s and the way it is selected from the n-element sequence.

A commonly used technique for selecting splitters is as follows: Let n be the number of elements to be sorted and m be the number of buckets. The scheme divides the n elements into m blocks of size n/m each and sorts each block using quicksort. From each sorted block it chooses $m-1$ evenly spaced elements. The $m(m-1)$ elements selected from all the blocks represent the sample used to determine the buckets. This scheme guarantees that the number of elements ending up in each bucket is less than $2n/m$.

A simple parallelization of this sorting algorithm can be derived by assigning a bucket to each of the p processors (*i.e.* $m = p$). Each processor is initially assigned n/p elements, which it sorts sequentially. It then chooses $p-1$ evenly spaced elements from the sorted block. The total number of elements selected across all processors is $p(p-1)$. These elements are sorted using a bitonic sort. After the global bitonic sort, each processor has $(p-1)$ sorted elements. Each processor picks one of these as a splitter. The p splitters thus picked are communicated to all processors using an all-to-all broadcast. Each processor then splits its local sample into p parts based on the splitters and communicates elements to designated processors. This is done using a single all-to-all personalized communication operation [11]. The elements received by each processor as a result of this operation are accrued appropriately into a sorted sequence.

Accurate Cost Model The computational costs of the algorithm are as follows: initial sort of n/p elements in time $t_c(n/p)\log(n/p)$; selection of $p-1$ sample elements in time $t_c(p-1)$; sorting $p(p-1)$ candidate splitters in parallel using bitonic sort in time $t_c p \log^2 p$; selecting a splitter in time t_c; inserting $(p-1)$ splitters into local sorted lists using binary search in time $t_c(p-1)\log(n/p)$; and the final assembly step can be computed in time $2t_c n/p$ since each processor can receive at most $2n/p$ elements. Since these costs are identical for all models, the rest of this section concentrates on modeling the communication overhead of the algorithm.

There are three main communication steps in the algorithm: communication involved in bitonic sort of p^2 elements ($\log p$ iterations of butterfly-shift of p elements); all-to-all broadcast of p splitters; and all-to-all personalized communication of elements to be sorted. The personalized communication is such that all processors send out n/p elements, but a processor may receive up to $2n/p$ elements.

The times for these three steps on the hypercube are as follows: The communication involved in bitonic sort takes time $\log^2 p \times (t_s + t_w p)$. The all-to-all broadcast takes time $t_s \log p + t_w p$. The third step is an unbalanced personalized communication. Traditional algorithms for personalized communication may yield very poor performance. We therefore resort to the transportation primitive of Shankar and Ranka [14, 15]. Using this primitive, the communication time can be bounded by $2(t_s + t_w \times 2n/p^2) \times p$. This expression holds provided $n/p > p^2$. The total communication time on a hypercube is therefore given by:

$$T_{comm,hypercube} = \log^2 p \times (t_s + t_w p) + t_s \log p + t_w p + 2(t_s + t_w \times \frac{2n}{p^2}) \times p$$

Neglecting the startup latencies, the asymptotic terms for the communication time reduce the expression to:

$$T_{comm,hypercube} = t_w (p \log^2 p + 4\frac{n}{p})$$

The communication time for the above steps on a mesh are given by the following: The communication involved in bitonic sort translates to a k-stride NEWS operation (in only two directions instead of the regular 4 directions). This can be reduced to k stride-one NEWS operations. The value of k varies as $\sqrt{p}/2, \sqrt{p}/4, ..., 1$. The time for this operation is given by

$$\sum_{k=0}^{k=\log(\sqrt{p}/2)} (2^k (t_s + t_w p)) = t_s \sqrt{p} + t_w p^{1.5}$$

There are $\log p$ iterations of this but the size of the mesh decreases by a factor of two each time. Therefore, the entire time is bounded by $2(t_s \sqrt{p} + t_w p^{1.5})$. The all-to-all broadcast takes time $2t_s \sqrt{p} + t_w p$. The personalized communication takes time $2(2t_s + t_w 2n/p)\sqrt{p}$. The total communication time on a mesh is therefore given by:

$$T_{comm,mesh} = 2(t_s \sqrt{p} + t_w p^{1.5}) + t_s \sqrt{p} + t_w p + 2(2t_s + 2t_w \frac{n}{p})\sqrt{p}$$

This expression asymptotically approximates to:

$$T_{comm,mesh} = 2t_w p^{1.5} + 4t_w \frac{n}{\sqrt{p}}$$

LogP Model The computation costs remain the same. The communication costs for the three stages are as follows: the communication in bitonic sort ($\log p$ butterfly stages) is performed in $\log^2 p$ stages. Each stage communicates p words of data. Assuming that the overhead o is negligible and gap g is greater than o, the time for each communication takes time $L + gp$. The total time for this operation is therefore $\log^2 p(L + gp)$. Assuming the same all-to-all broadcast algorithm as in the accurate model, the time for the broadcast is given by $L \log p + gp$. It is not clear though how the algorithm would be reached using the LogP model. Since we are primarily concerned with predictive properties of the models, we will assume that the algorithm is given to us. The time for personalized communication of $2n/p$ elements is $Lp + 2gn/p$. Therefore the total time is given by:

$$T_{comm,LogP} = \log^2 p(L + gp) + L \log p + gp + Lp + 2gn/p$$

For a hypercube, the gap g is equal to t_w. Therefore, the total time for a hypercube as predicted by the $\log P$ model is given by

$$T_{comm,hypercube,LogP} = \log^2 p(L + t_w p) + L \log p + t_w p + Lp + 2t_w n/p$$

Assuming that the bandwidth term dominates latency L, this time reduces to

$$T_{comm,hypercube,LogP} = t_w(p \log^2 p + 2n/p)$$

For a mesh, the gap g is determined by the link bandwidth scaled down by the cross-section bandwidth factor. Therefore, $g = \sqrt{p} t_w$ for the mesh. The asymptotic time for the mesh is therefore given by

$$T_{comm,mesh,LogP} = t_w(p^{1.5} \log^2 p + 2n/\sqrt{p})$$

Approximate Model The communication times predicted by our approximate model for the hypercube are as follows: the communication involved in bitonic sort is $\log p$ stages of a butterfly. Each message is of size p. The time for each stage is given by $\mu p \log p$. There are $\log p$ stages, therefore the total time is given by $\mu p \log^2 p$. The all-to-all broadcast is not bandwidth sensitive and the total amount of data entering any processor is p words. Therefore the time is given by μp. The personalized communication of $n/2p$ words is bandwidth sensitive and takes time $2\mu n/p$. Therefore, the total time is given by:

$$T_{comm,hypercube,approx} = \mu(p \log^2 p + p + 2\frac{n}{p})$$

The corresponding times for the mesh are as follows: The k-stride NEWS is implemented as k single-stride NEWS communications. The time for this can be derived in a manner similar to the exact model and equals $2\mu p^{1.5}$. The all-to-all broadcast takes the same time as on a hypercube, i.e. μp, since the operation is not bandwidth sensitive. The personalized communication of $2n/p$ elements takes time $\mu 2n/\sqrt{p}$. Therefore, the total time is given by:

$$T_{comm,mesh,approx} = \mu(2p^{1.5} + p + \frac{2n}{\sqrt{p}}$$

Communication	Hypercube			Mesh		
	Approx	Exact	LogP	Approx	Exact	LogP
Communication (Bitonic Sort)	$\mu p \log^2 p$	$\log^2 p(t_s + t_w p)$	$\log^2 p(L + t_w p)$	$2\mu p^{1.5}$	$2(t_s\sqrt{p} + t_w p^{1.5})$	$\log^2 p(L + t_w p^{1.5})$
All-to-all Broadcast	μp	$t_s \log p + t_w p$	$L \log p + t_w p$	μp	$2t_s\sqrt{p} + t_w p$	$L \log p + t_w p^{1.5}$
Personalized Communication (data = $2n/p$)	$\frac{2\mu n}{p}$	$2t_s p + 4t_w \frac{n}{p}$	$L + 2t_w \frac{n}{p}$	$\mu 2 \frac{n}{\sqrt{p}}$	$4t_s\sqrt{p} + 4t_w \frac{n}{\sqrt{p}}$	$L + 2t_w \frac{n}{\sqrt{p}}$

Table 3: Communication times for various models for sample sort.

Table 3 summarizes the communication cost of the algorithm predicted by various models. From this table, it is clear that the approximate model gives accurate results when the startup latencies are small. The LogP model, however, underestimates available bandwidth and consequently overestimates the total time for the mesh. Furthermore, by ignoring placement, each strided NEWS operation is treated as a general permutation. This leads to a further overestimation of the runtime.

7.4 Gaussian Elimination for Solving Dense Linear Systems

Gaussian elimination is used for solving a linear system of equations by reducing it to its upper triangular form and determining solutions by back-substitution. For a detailed description of the algorithm and its parallel formulations, the reader is referred to Kumar et. al. [11]. For a dense linear system with n unknowns, Gaussian elimination involves approximately $n^2/2$ divisions and approximately $(n^3/3) - (n^2/2)$ subtractions and multiplications. Assuming that each scalar arithmetic operation takes unit time, the sequential run time of the procedure is approximately $2n^3/3$ (for large n).

Parallel formulations of Gaussian elimination are based on partitioning the matrix. Popular partitioning techniques include striping, checkerboarding, and cyclic mappings [11]. In this section, we compare various models on the basis of their performance prediction of the checkerboard version of parallel Gaussian elimination.

Given a linear system $Ax = b$ with n unknowns, the $n \times n$ matrix A is mapped to a $\sqrt{p} \times \sqrt{p}$ logical grid of processors using a checkerboard mapping. Gaussian elimination proceeds in the following steps: (i) row-broadcast of $n/\sqrt{p}$ elements corresponding to column of the variable currently being eliminated; (ii) n^2/p divisions corresponding to dividing all the numbers assigned to a processor by the coefficient of the variable being eliminated; (iii) column broadcast of $n/\sqrt{p}$ elements from the equation used to eliminate the variable; (iv) n^2/p subtractions corresponding to subtracting each element by the row received in the column broadcast.

Since the computational cost predicted by each model is identical, we concentrate on the communication cost predicted by them. Since the only communication operation is a row / column broadcast, it is possible to determine the cost of one of these operations and multiply it by two (assuming that there is no pipelining of communication). In a hypercube, each row / column forms a subcube. Therefore the exact cost of this operation is given by $(t_s + t_w n/\sqrt{p}) \log \sqrt{p}$. The combined cost of the row and column broadcasts is therefore given by $(t_s + t_w n/\sqrt{p}) \log p$. The corresponding cost for the mesh can be derived to be $(t_s + t_w n/\sqrt{p}) \log p + 4t_h(\sqrt{p} - 1)$. It is possible to lose the $\log p$ factor in each of these times if we can break the broadcast message into parts. This requires the original message to be of size greater than $\log p$ for the hypercube and $\sqrt{p}$ for the mesh. This implies that $n/\sqrt{p} > \sqrt{p}$, or $n > p$. This is indeed a reasonable condition. In this situation, the communication overhead for a hypercube is given by $4(t_s \log p + t_w n/\sqrt{p})$ and that for a mesh is given by $4(2t_s\sqrt{p} + t_w n/\sqrt{p})$.

For the approximate model, the one-to-all broadcast operation is not bandwidth sensitive. Furthermore, the maximum amount of data leaving a processor is $\frac{n}{\sqrt{p}}$. Therefore, the cost of this broadcast operation is $\mu\frac{n}{\sqrt{p}}$ and the combined cost of both operations is $2\mu\frac{n}{\sqrt{p}}$ on both the mesh and the hypercube.

The timings for the LogP model can be derived in a manner similar to the exact model. As before, we assume that $g > o$ to utilize link bandwidth. The time for sending a single message of size $n/\sqrt{p}$ is $(L + g \times n/\sqrt{p})$. The time for a broadcast to $\sqrt{p}$ processors corresponds to the time for sending $\log \sqrt{p}$ such messages. Therefore the time is $(L + g \times n/\sqrt{p}) \log \sqrt{p}$, and the total time is given by $(L + g \times n/\sqrt{p}) \log p$. Assuming that the message size is large enough that the term corresponding to latency L can be dropped, this time becomes $g\frac{n}{\sqrt{p}} \log p$. The values of g for a hypercube and a mesh are t_w and $t_w\sqrt{p}$ respectively. Substituting these values, we can see that the communication overhead predicted by the LogP model for the hypercube is $t_w\frac{n}{\sqrt{p}} \log p$ and that for the mesh is $t_w n \log p$.

The approximate model predicts the communication overhead to be off from the accurate model by a factor of $\log p$. However, as we have mentioned, it is possible to eliminate the $\log p$ factor by using a faster broadcast algorithm (which requires broadcast message to be of size p or more). In such a situation, ignoring the per-hop and startup times, the approximate model yields accurate results for both the mesh and the hypercube. The LogP model overestimates the time for a mesh by a factor of $\sqrt{p}$. Furthermore, it is apparent from this example that whereas it is easy to analyze algorithms in our framework, it is more difficult to do so using the LogP model since it lacks the base of aggregate communication operations.

References

[1] A. Agarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. Technical Report RC 14998 (No.64644), IBM T.J. Watson Research Center, Yorktown Heights, NY, 1989.

[2] A. Agarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computations. Technical Report RC 14973 (No.66882), IBM T.J. Watson Research Center, Yorktown Heights, NY, 1989.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[4] D. Culler, R. Karp, D. Patterson, and et al. Logp: Towards a realistic model of parallel computation. 1992.

[5] W. J. Dally. Analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6), June 1990.

[6] Pilar de la Torre and Clyde P. Kruskal. Towards a single model of efficient computation in real parallel machines. In *Future Generation Computer Systems*, pages 395 – 408, 8(1992).

[7] G. C. Fox, M. Johnson, G. Lyzenga, S. W. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors: Vol. 1*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[8] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, 1 (3):12–21, 1993. Also available as Technical Report TR-93-24, Department of Computer Science, University of Minnesota and from anonymous ftp site ftp.cs.umn.edu, file users/kumar/isoeff-tutorial.ps.

[9] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation. i. the model. *Journal of Parallel and Distributed Computing*, 16(3):212–32, Nov. 1992.

[10] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation. ii. binary tree and fft algorithms. *Journal of Parallel and Distributed Computing*, 16(3):233–49, Nov. 1992.

[11] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Algorithm Design and Analysis*. Benjamin Cummings/ Addison Wesley, Redwod City, 1994.

[12] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21(4):339–374, November 1984.

[13] A. G. Ranade. How to emulate shared memory. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science*, pages 185 – 194, 1987.

[14] R. Shankar and S. Ranka. Random data access on a coarse grained parallel machine i. one-to-one mappings. Technical report, School of Computer and Information Science, Syracuse University, Syracuse, NY, 13244, 1995.

[15] R. Shankar and S. Ranka. Random data access on a coarse grained parallel machine ii. one-to-many and many-to-one mappings. Technical report, School of Computer and Information Science, Syracuse University, Syracuse, NY, 13244, 1995.

[16] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

[17] L. G. Valiant. General purpose parallel architectures. *Handbook of Theoretical Computer Science*, 1990.

[18] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.