The Block Distributed Memory Model

Joseph F. JáJá[1,3] **Kwan Woo Ryu**[1,2,3]

Dept. of Electrical Engineering
Inst. for Advanced Comp. Studies
Inst. for Systems Research

Dept. of Computer Engineering
Kyungpook Natl Univ., KOREA
Inst. for Advanced Comp. Studies

University of Maryland
College Park, MD 20742

Abstract

We introduce a computation model for developing and analyzing parallel algorithms on distributed memory machines. The model allows the design of algorithms using a single address space and does not assume any particular interconnection topology. We capture performance by incorporating a cost measure for interprocessor communication induced by remote memory accesses. The cost measure includes parameters reflecting memory latency, communication bandwidth, and spatial locality. Our model allows the initial placement of the input data and pipelined prefetching.

We use our model to develop parallel algorithms for various data rearrangement problems, load balancing, sorting, FFT, and matrix multiplication. We show that most of these algorithms achieve optimal or near optimal communication complexity while simultaneously guaranteeing an optimal speed-up in computational complexity.

[1]Partially supported by the National Science Foundation, Grant No. CCR-9103135, and by the National Science Foundation Engineering Research Center Program NSFD CD 8803012.

[2]Supported in part by NONDIRECTED RESEARCH FUND, Korea Research Foundation, 1992.

[3]The email addresses of the authors are joseph@src.umd.edu and ryu@umiacs.umd.edu.

1 Introduction

Parallel processing promises to offer a quantum leap in computational power that is likely to have a substantial impact on various aspects of the computing field, and that in particular can be exploited to investigate a wide range of what has been called "grand challenge" problems in science and engineering. It is widely recognized [23] that an important ingredient for the success of this technology is the emergence of computational models that can be used for algorithms development and for accurately predicting the performance of these algorithms on real machines. We take a similar view as in [24] in that the computation model should be a "bridging model" that links the two layers of hardware and software. Existing computation models tend to be biased towards one or the other layer, except for very few exceptions. The Bulk Synchronous Parallel (BSP) model advocated by Valiant [24] is one of the few exceptions. In this paper, we introduce a computation model that specifically attempts to be a bridging model between the shared memory (single address) programming model and the distributed-memory message passing architectures. Distributed memory systems configured as a single address space are usually referred to as (scalable) *shared memory multiprocessors*. These machines achieve the scalability of distributed memory architectures and the simple programming style provided by the single address space. Our model can also be used for predicting performance of data parallel algorithms running on distributed memory architectures.

Since a computation model should predict performance on real machines, we start with a discussion on the basis of our measure of communication costs incurred by accessing remote data. As indicated in [8], the hardware organizations of massively parallel processors (MPPs) seem to be converging towards a collection of powerful processors connected by a communication network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. According to this common organization, the communication between the different processors is handled by point-to-point messages whose routing times are controlled by parameters related to the network latency, processor communication bandwidth, overhead in preparing a message, and network capacity. Such a model avoids a description of the exact structure of the network since algorithms exploiting specific features of the network are less likely to be robust enough to work well on a variety of architectures and to adapt easily to possible future technological changes. However programming the machine at the message-passing level imposes a heavy burden on the programmer and makes algorithms development and evaluation quite complicated. On the other hand, the data-parallel and the shared-memory programming models are appealing in terms of their ease of use and in terms of their close relationship to sequential programming. Both models assume a single address space.

The Block Distributed Memory (BDM) model introduced in the next section captures the performance of shared memory (single address space) algorithms by incorporating a cost measure for interprocessor communication caused by remote memory accesses. The cost is modeled using the *latency* and the *communication bandwidth* of each processor. Since a remote memory access involves the transmission of a packet that typically contains a number of consecutive words, our model encourages the use of *spatial locality* by incorporating a parameter m that represents a cost associated with accessing up to m consecutive words; this cost will be incurred even if a single word is needed. Our model allows the *initial placement* of input data and includes the memory latency hiding technique of *pipelined prefetching*. Since we measure the amount of local computation and the amount of communication separately, we are able to normalize the communication cost and drop one parameter so as to make the analysis of the corresponding algorithms simpler. We use our model to develop parallel algorithms for various data rearrangement problems, load balancing, sorting, the Fast Fourier Transform (FFT) computation, and matrix multiplication. We show that most of these algorithms achieve optimal or near optimal communication complexity while simultaneously guaranteeing an optimal speed-up in computational complexity.

In the next section, we provide the details of our model while Section 3 describes a collection of algorithms for handling data rearrangements that occur frequently in shared memory algorithms. The load balancing problem is addressed in Section 4 where a communication efficient algorithm is presented, and Section 5 is devoted to the presentation of efficient algorithms for sorting, FFT, and matrix multiplication. Most of the resulting algorithms seem to share a common structure with high-performance algorithms that have been tested on real machines.

2 The Block Distributed Memory (BDM) Model

Our computation model, the Block Distributed Memory (BDM), will be defined in terms of four parameters p, τ, σ, and m. As we will see later, the parameter σ can be dropped without loss of generality. The parameter p refers to the number of processors; each such processor is viewed as a unit cost random access machine (RAM). In addition, each processor has an interface unit to the interconnection network that handles communication among the different processors. Data are communicated between processors via point-to-point messages; each message consists of a packet that holds m words from *consecutive* locations of a local processor memory. Since we are assuming the shared memory programming model, each request to a remote location involves the preparation of a request packet, the injection of the packet into the network, the reception of the packet at the destination processor, and finally the sending of a packet containing the contents of m consecutive locations, including the requested value, back to the requesting processor. We will model the cost of handling the request to a remote location (read or write) by the formula $\tau + m\sigma$, where τ is the maximum latency time it takes for a requesting processor to receive the appropriate packet, and σ is the rate at which a processor can inject (receive) a word into (from) the network. Moreover, no processor can send or receive more than one packet at a time. As a result we note the following two observations. First, if π is any permutation on p elements, then a remote memory request issued by processor P_i and destined for processor $P_{\pi(i)}$ can be completed in $\tau + m\sigma$ time for all processors P_i, $0 \leq i \leq p-1$, simultaneously. Second, k remote access requests issued by k

distinct processors and destined to the same processor will require $k(\tau + m\sigma)$ time to be completed; in addition, we do not make any assumption on the relative order in which these requests will be completed.

Most current interconnection networks for multiprocessors use several hardware and software techniques for hiding memory latency. In our model, we allow pipelined *prefetching* for hiding memory latency. In particular, k prefetch read operations issued by a processor can be completed in $\tau + km\sigma$ time.

The underlying communication model for BDM is consistent with the LogP and the postal models [8, 13, 5] but with the addition of the parameter m that incorporates spatial locality. However, our model does not allow low-level handling of message passing primitives except implicitly through data accesses. In particular, an algorithm written in our model can specify the initial data placement among the local memories of the p processors, can use the processor id to refer to specific data items, and can use synchronization barriers to synchronize the activities of various processors whenever necessary. Remote data accesses are charged according to the communication model specified above. As for synchronization barriers, we make the assumption that, on the BDM model, they are provided as primitive operations. There are two main reasons for making this assumption. The first is that barriers can be implemented in hardware efficiently at a relatively small cost. The second is that we can make the latency parameter τ large enough to account for synchronization costs. The resulting communication costs will be on the conservative side but that should not affect the overall structure of the resulting algorithms.

The complexity of a parallel algorithm on the BDM model will be evaluated in terms of two measures: the computation time T_{comp}, and the communication time T_{comm}. The measure T_{comp} refers to the maximum of the local computation performed on any processor as measured on the standard sequential RAM model. The communication time T_{comm} refers to the total amount of communication time spent by the overall algorithm in accessing remote data. Our main goal is the design of parallel algorithms that achieve optimal or near-optimal computational speedups, that is, $T_{comp} \approx O(\frac{T_{seq}}{p})$, where T_{seq} is the sequential complexity of the problem under consideration, in such a way that the total *communication time* T_{comm} *is minimized.*

Since T_{comm} is treated separately from T_{comp}, we can normalize this measure by dividing it by σ. The underlying communication model for BDM can now be viewed as the postal model [5] but with the added parameter m reflecting spatial locality. Hence an access operation to a remote location takes $\tau + m$ time, and k prefetch read operations can be executed in $\tau + km$ time. Note that the parameter τ should now be viewed as an upper bound on the capacity of the interconnection network, i.e., an upper bound on the maximum number of words in transit from or to a processor. In our estimates of the bounds on the communication time, we make the simplifying (and reasonable) assumption that τ is an integral multiple of m.

We believe that locality is an important factor that has to be taken into consideration when designing parallel algorithms for large scale multiprocessors. We have incorporated the parameter m into our model to emphasize the importance of spatial locality. The notion of *processor locality* also seems to be important in current multiprocessor architectures; these architectures tend to be hierarchical, and hence the latency τ is much higher for accessing processors that are further up in the hierarchy than those that are "close by". This feature can be incorporated into our model by modifying the value of τ to reflect the cost associated with the level of hierarchy that needs to be used for a remote memory access. This can be done in a similar fashion as in the memory hierarchy model studied in [2] for sequential processors. However in this paper we have opted for simplicity and decided not to include the processor locality into consideration.

Several models that have been discussed in the literature, other than the LogP and the postal models referred to earlier, are related to our BDM model. However there are significant differences between our model and each of these models. For example, both the Asynchronous PRAM [9] and the Block PRAM [1] assume the presence of a shared memory where intermediate results can be held; in particular, they both assume that the data is initially stored in this shared memory. This makes data movement operations considerably simpler than in our model. Another example is the Direct Connection Machine (DCM) with latency [14] that uses message passing primitives; in particular, this model does not allow pipelined prefetching as we do in the BDM model.

3 Basic Algorithms for Data Movements

The design of communication efficient parallel algorithms depends on the existence of efficient schemes for handling frequently occurring transformations on data layouts. In this section, we consider data layouts that can be specified by a two-dimensional array A, say of size $q \times p$, where column i of A contains a subarray stored in the local memory of processor P_i, where $0 \leq i \leq p-1$. A transformation Π on the layout A will map the elements of A into the layout $\Pi(A)$ not necessarily of the same size. We present optimal or near optimal algorithms to handle several such transformations including broadcasting operations, matrix transposition, and data permutation. All the algorithms described are deterministic except for the algorithm to perform a general permutation.

We start by addressing several broadcasting operations. The simplest case is to broadcast a single item to a number of remote locations. Hence the layout A can be described as a one-dimensional array and we assume that the element $A[0]$ has to be copied into the remaining entries of A. This can be viewed as a concurrent read operation from location $A[0]$ executed by processors $P_1, P_2, \ldots, P_{p-1}$. The next lemma provides a simple algorithm to solve this problem; we later use this algorithm to derive an optimal broadcasting algorithm.

Lemma 3.1 *Given a p-processor BDM and an array $A[0:p-1]$ where $A[j]$ resides in processor P_j, the element $A[0]$ can be copied into the remaining entries of A in $\tau + m(p-1)$ communication time.*

Proof: A simple algorithm consists of $p-1$ rounds that can be pipelined. During the rth round, each processor P_j reads $A[(j+r) \bmod p]$, for $1 \leq r \leq p-1$; however, only $A[0]$ is copied into A[j]. Since these rounds can be realized with $p-1$ pipelined prefetch read operations, the resulting communication complexity is $\tau + m(p-1)$. □

We are now ready for the following theorem that essentially establishes the fact that a k-ary balanced tree broadcasting algorithm is the best possible for $k = \frac{\tau}{m} + 1$ (recall that we earlier made the assumption that τ is an integral multiple of m).

Theorem 3.1 *Given a p-processor BDM, an item in a processor can be broadcast to the remaining processors in* $\leq 2\tau \lceil \frac{\log p}{\log(\frac{\tau}{m}+1)} \rceil$ *communication time. On the other hand, any broadcasting algorithm that only uses read, write, and synchronization barrier instructions requires at least* $(\tau \frac{\log p}{\log(\frac{\tau}{m}+2)} + m \log p)$ *communication complexity.*[4]

Proof: We start by describing the algorithm. Let k be an integer to be determined later. The algorithm can be viewed as a k-ary tree rooted at location $A[0]$; there are $\lceil \log_k p \rceil$ rounds. During the first round, $A[0]$ is broadcast to locations $A[1], A[2], \ldots, A[k-1]$, using the algorithm described in Lemma 3.1, followed by a synchronization barrier. Then during the second round, each element in locations $A[0], A[1], \ldots, A[k-1]$ is broadcast to a distinct set of $k-1$ locations, and so on. The communication cost incurred during each round is given by $\tau + (k-1)m$ (Lemma 3.1). Therefore the total communication cost is $T_{comm} \leq (\tau + (k-1)m)\lceil \log_k p \rceil$. If we set $k = \frac{\tau}{m} + 1$, then $T_{comm} \leq (\tau + \frac{\tau}{m} m)\lceil \frac{\log p}{\log(\frac{\tau}{m}+1)} \rceil = 2\tau \lceil \frac{\log p}{\log(\frac{\tau}{m}+1)} \rceil$.

We next establish the lower bound stated in the theorem. Any broadcasting algorithm using only read, write, and synchronization barrier instructions can be viewed as operating in phases, where each phase ends with a synchronization barrier (whenever there are more than a single phase). Suppose there are s phases. The amount of communication to execute phase i is at least $\tau + k_i m$, where k_i is the maximum number of copies read from any processor during phase i. Hence the total amount of communication required is at least $\sum_{i=1}^{s} (\tau + k_i m)$. Note that by the end of phase i, the desired item has reached at most $(k_1+1)(k_2+1)\cdots(k_i+1)$ remote locations. It follows that, if by the end of phase s, the desired item has reached all the processors, we must have $p \leq (k_1+1)(k_2+1)\cdots(k_s+1)$. The communication time $\sum_{i=1}^{s}(\tau + k_i m)$ is minimized when $k_1 = k_2 = \cdots = k_s = k$, and hence $(k+1)^s \geq p$. Therefore $s \geq \frac{\log p}{\log(k+1)}$ and the communication time is at least $s(\tau + km) \geq (\tau + km)\frac{\log p}{\log(k+1)}$. We complete the proof of this theorem by proving the following claim.

Claim: $\frac{\tau + km}{\log(k+1)} \geq \frac{\tau}{\log(\frac{\tau}{m}+2)} + m$, for any $k \geq 1$.

Proof of the Claim: Let $r = \frac{\tau}{m}$, $f_1(k) = \frac{\tau}{\log(k+1)}$, $f_2(k) = \frac{km}{\log(k+1)}$, and $f(k) = f_1(k) + f_2(k) = \frac{\tau + km}{\log(k+1)}$. Then,

$$f'(k) = \frac{m(k+1)\log(k+1) - (\log e)(\tau + km)}{(k+1)\log^2(k+1)}.$$

(Case 1) $(1 \leq k \leq r+1)$: Since $f_1(k)$ is decreasing and $f_2(k)$ is increasing in this range, the claims follows easily by noting that $f_1(k) \geq f_1(r+1) = \frac{\tau}{\log(\frac{\tau}{m}+2)}$ and

[4]All logarithms are to the base 2 unless otherwise stated.

$f_2(k) \geq f_2(1) = m$.
(Case 2) $(k > r+1)$: We show that $f(k)$ is increasing when $k > r+1$ by showing that $f'(k) > 0$ for all integers $k \geq r+1$. Note that since $k \geq \frac{\tau}{m} + 1$, we have that $m(k+1)\log(k+1) - (\log e)(\tau + km)$ is at least as large as $m[(k+1)\log(k+1) - (\log e)(2k-1)]$ which is positive for all nonzero integer values of k. Hence $f(k) \geq f(\frac{\tau}{m}+1)$ and the claim follows. □

The *sum* of p elements on a p-processor BDM can be computed in at most $2\tau\lceil\frac{\log p}{\log(\frac{\tau}{m}+1)}\rceil$ communication time by using a similar strategy. Based on this observation, it is easy to show the following theorem.

Theorem 3.2 *Given n numbers distributed equally among the p processors of a BDM, we can compute their sum in $O(\frac{n}{p} + \frac{\tau \log p}{m \log \frac{\tau}{m}})$ computation time and at most $2\tau\lceil\frac{\log p}{\log(\frac{\tau}{m}+1)}\rceil$ communication time. The computation time reduces to $O(\frac{n}{p})$ whenever $p\log p \leq \frac{n}{\tau}$.* □

Another simple broadcasting operation is when each processor has to broadcast an item to all the remaining processors. This operation can be executed in $\min\{\tau + m(p-1), 2\tau + (m^2 + p)\}$ communication time as shown in the next lemma.

Lemma 3.2 *Given a p-processor BDM and an array $A[0:p-1]$ distributed one element per processor, the problem of broadcasting each element of A to all the processors can be done in $\min\{\tau + m(p-1), 2\tau + m^2 + p\}$ communication time.*

Proof: The bound of $\tau + m(p-1)$ follows from the simple algorithm described in Lemma 3.1. If p is significantly larger than m, then we can use the following strategy. We use the previous algorithm until each processor has m elements. Next, each block of m elements is broadcast in a circular fashion to the appropriate $(\lceil\frac{p}{m}\rceil - 1)$ processors. One can verify that the resulting communication complexity is $2\tau + m^2 + p$. □

Our next data movement operation is the matrix transposition that can be defined as follows. Let $q \geq p$ and let p divide q evenly without loss of generality. The data layout described by A is supposed to be rearranged into the layout A' so that the first column of A' contains the first q/p consecutive rows of A laid out in row major order form, the second column of A' contains the second set of q/p consecutive rows of A, and so on. Clearly, if $q = p$, this corresponds to the usual notion of matrix transpose.

An efficient algorithm to perform matrix transposition on the BDM model is similar to the algorithm reported in [8]. There are $p-1$ rounds that can be fully pipelined by using prefetch read operations. During the first round, the appropriate block of q/p elements in the ith column of A is read by processor $P_{(i+1)\bmod p}$ into the appropriate locations, for $0 \leq i \leq p-1$. During the second round, the appropriate block of data in column i is read by processor $P_{(i+2)\bmod p}$, and so on. The resulting total communication time is given by $T_{comm} = \tau + (p-1)m\lceil\frac{q}{pm}\rceil \leq \tau + (q - \frac{q}{p}) + (p-1)m$ and the amount of local computation is $O(q)$. Clearly this algorithm is optimal whenever pm divides q. Hence we have the following lemma.

Lemma 3.3 *A $q \times p$ matrix transposition can be performed on a p-processor BDM in $\tau + (p-1)m\lceil \frac{q}{pm} \rceil$; this bound reduces to $\tau + (q - \frac{q}{p})$ whenever pm divides q.* $\Box$

We next discuss the broadcasting operation of a block of n elements residing on a single processor to p processors. We describe two algorithms, the first is suitable when the number n of elements is relatively small, and the second is more suitable for large values of n. Both algorithms are based on circular data movement as used in the matrix transposition algorithm. The details are given in the proof of the next theorem.

Theorem 3.3 *The problem of broadcasting n elements from a processor to p processors can be completed in at most $2[2\tau\lceil \frac{\log p}{\log(\frac{\tau}{m}+1)} \rceil + n]$ communication time by using a k-ary balanced tree algorithm. On the other hand, this problem can be solved in at most $2[\tau + (p-1)m\lceil \frac{n}{pm} \rceil]$ communication time by using the matrix transposition algorithm.*

Proof: For the first algorithm, we use a k-ary tree as in the single item broadcasting algorithm described in Theorem 3.1, where $k = \frac{\tau}{m}+1$. Using the matrix transposition strategy, distribute the n elements to be broadcast among k processors, where each processor receives a contiguous block of size $\frac{n}{k}$. We now view the p processors as partitioned into k groups, where each group includes exactly one of the processors that contains a block of the items to be broadcast. The procedure is repeated within each group and so on. A similar reverse process can gradually read all the n items into each processor. Each forward or backward phase is carried out by using the cyclic data movement of the matrix transposition algorithm. One can check that the communication time can be bounded as follows.

$$\begin{aligned} T_{comm} &\leq 2[(\tau + (k-1)m\lceil \frac{n}{km} \rceil) + (\tau + (k-1)m\lceil \frac{n}{k^2 m} \rceil) + \cdots] \\ &\leq 2[2\tau\lceil \frac{\log p}{\log(\frac{\tau}{m}+1)} \rceil + n] \end{aligned}$$

If $n > pm$, we can broadcast the n elements in $2[\tau + (p-1)m\lceil \frac{n}{pm} \rceil]$ communication time using the matrix transposition algorithm of Lemma 3.3 twice, once to distribute the n elements among the p processors where each processor receives a block of size $\frac{n}{p}$, and the second time to circulate these blocks to all the processors. $\Box$

The problem of *distributing* n elements from a single processor can be solved by using the first half of either of the above two broadcasting algorithms. Hence we have the following corollary.

Corollary 3.1 *The problem of distributing n elements from one processor to $p-1$ processors such that each processor receives $\frac{n}{p}$ elements can be completed in at most* $\min\{(2\tau\lceil \frac{\log p}{\log(\frac{\tau}{m}+1)} \rceil + n), (\tau + (p-1)m\lceil \frac{n}{pm} \rceil)\}$ *communication time.* $\Box$

We finally address the following general routing problem. Let A be an $\frac{n}{p} \times p$ array of n elements initially stored one column per processor in a p-processor BDM machine. Each element of A consists of a pair (data,i), where i is the index of the processor to which the data has to be relocated. We assume that at most $\alpha\frac{n}{p}$ elements have to be routed to any single processor for some constant $\alpha \geq 1$. We describe in what follows a randomized algorithm that completes the routing in $2(\tau + c\frac{n}{p})$ communication time and $O(\frac{n}{p})$ computation time, where c is any constant larger than $\max\{1+\frac{1}{\sqrt{2}}, \alpha+\frac{\sqrt{\alpha}}{2}\}$. The complexity bounds are guaranteed to hold with high probability, that is, with probability $\geq 1 - n^{-\epsilon}$, for some positive constant ϵ, as long as $p^2 < \frac{n}{6\ln n}$, where ln is the logarithm to the base e.

The overall idea of the algorithm has been used in various randomized routing algorithms on the mesh. Here we follow more closely the scheme described in [20] for randomized routing on the mesh with bounded queue size.

Before describing our algorithm, we introduce some terminology. We use an auxiliary array A' of size $\frac{cn}{p} \times p$ for manipulating the data during the intermediate stages and for holding the final output, where $c > \max\{1 + \frac{1}{\sqrt{2}}, \alpha + \frac{\sqrt{\alpha}}{2}\}$. Each column of A' will be held in a processor. The array A' can be divided into p equal size slices, each *slice* consisting of $\frac{cn}{p^2}$ consecutive rows of A'. Hence a slice contains a set of $\frac{cn}{p^2}$ consecutive elements from each column and such a set is referred to as a *slot*. We are ready to describe our algorithm.

Algorithm Randomized_Routing

Input: An input array $A[0 : \frac{n}{p} - 1, 0 : p - 1]$ such that each element of A consists of a pair (data,i), where i is the processor index to which the data has to be routed. No processor is the destination of more than $\alpha\frac{n}{p}$ elements for some constant α.

Output: An output array $A'[0 : \frac{cn}{p} - 1, 0 : p - 1]$ holding the routed data, where c is any constant larger than $\max\{1 + \frac{1}{\sqrt{2}}, \alpha + \frac{\sqrt{\alpha}}{2}\}$.

begin

[Step 1] Each processor P_j distributes randomly its $\frac{n}{p}$ elements into the p slots of the jth column of A'.

[Step 2] Transpose A' so that the jth slice will be stored in the jth processor, for $0 \leq j \leq p - 1$.

[Step 3] Each processor P_j distributes locally its $\leq \frac{cn}{p}$ elements such that every element of the form (*,i) resides in slot i, for $0 \leq i \leq p - 1$.

[Step 4] Perform a matrix transposition on A' (hence the jth slice of the layout generated at the end of Step 3 now resides in P_j).

end

The next two facts will allow us to derive the complexity bounds for our randomized routing algorithm. For the analysis, we assume that $p^2 < \frac{n}{6\ln n}$.

Lemma 3.4 *At the completion of Step 1, the number of elements in each slot is no more than $\frac{cn}{p^2}$ with high probability, for any $c > 1 + \frac{1}{\sqrt{2}}$.*

Proof: The procedure performed by each processor is similar to the experiment of throwing $\frac{n}{p}$ balls into p bins. Hence the probability that exactly $\frac{cn}{p^2}$ balls are placed in any particular bin is given by the binomial distribution

$$b(k; N, q) = \left(\begin{array}{c} N \\ k \end{array} \right) q^k (1-q)^{N-k},$$

where $k = \frac{cn}{p^2}$, $N = \frac{n}{p}$, and $q = \frac{1}{p}$. Using the following Chernoff bound for estimating the tail of the binomial distribution

$$\sum_{j \geq (1+\epsilon)Nq} b(j; N, q) \leq e^{-\epsilon^2 Nq/3},$$

we obtain that the probability that a particular bin has more than $\frac{cn}{p^2}$ balls is upper bounded by

$$e^{-(c-1)^2 \frac{n}{3p^2}}.$$

Therefore the probability that any of the bins has more than $\frac{cn}{p^2}$ balls is bounded by $p^2 e^{-(c-1)^2 \frac{n}{3p^2}}$ and the lemma follows. $\Box$

Lemma 3.5 *At the completion of Step 3, the number of elements in any processor which are destined to the same processor is at most $\frac{cn}{p^2}$ with high probability, for any $c > \alpha + \frac{\sqrt{\alpha}}{2}$.*

Proof: The probability that an element is assigned to the jth slice by the end of Step 1 is $\frac{1}{p}$. Hence the probability that $\frac{cn}{p^2}$ elements destined for a single processor fall in the jth slice is bounded by $b(\frac{cn}{p^2}; \frac{\alpha n}{p}, \frac{1}{p})$ since no processor is the destination of more than $\frac{\alpha n}{p}$ elements. Since there are p slices, the probability that more than $\frac{cn}{p^2}$ elements in any processor are destined for the same processor is bounded by

$$pe^{-(\frac{c}{\alpha}-1)^2 \frac{\alpha n}{3p^2}},$$

and hence the lemma follows. $\Box$

¿From the previous two lemmas, it is easy to show the following theorem.

Theorem 3.4 *The routing of n elements stored initially in an $\frac{n}{p} \times p$ array A of a p-processor BDM such that at most $\alpha\frac{n}{p}$ elements are destined to the same processor can be completed with high probability in $2(\tau + c\frac{n}{p})$ communication time and $O(\frac{n}{p})$ computation time, where c is any constant larger than $\max\{1 + \frac{1}{\sqrt{2}}, \alpha + \frac{\sqrt{\alpha}}{2}\}$, and $p^2 < \frac{n}{6 \ln n}$.* $\Box$

Remark: Since we are assuming that $p^2 < \frac{n}{6 \ln n}$, the effect of the parameter m is dominated by the bound $c\frac{n}{p}$ (as $\frac{n}{p} > 6p \ln n \geq mp$, assuming $m \leq 6 \ln n$). $\Box$

4 Load Balancing

Balancing load among processors is very important since poor balance of load generally causes poor processor utilization [19]. The load balancing problem is also important in developing fast solutions for basic combinatorial problems such as sorting, selection, list ranking, and graph problems [12, 21].

This problem can be defined as follows. The load in each processor P_i is given by an array $A_i[0 : n_i - 1]$, where n_i represents the number of useful elements in P_i such that $\max_{j=0}^{p-1}\{n_j\} = M$ and $\sum_{j=0}^{p-1} n_j = n$. We are supposed to redistribute the n elements over the p processors such that $\frac{n}{p}$ elements are stored in each processor, where we have assumed without loss of generality that p divides n.

In this section, we develop a simple and efficient load balancing algorithm for the BDM model. The corresponding communication time is given by $T_{comm} \leq 5\tau + M + \frac{n}{p} + p + 2m^2$. The overall strategy is described next.

Let $n_{i,1} = \lfloor \frac{n_i}{m} \rfloor m$ and $n_{i,2} = n_i - n_{i,1}$, for $0 \leq i \leq p-1$. Then, the overall strategy of the load balancing algorithm can be described as follows: First, the load balancing problem of the $(n_{0,1} + n_{1,1} + \cdots + n_{p-1,1})$ elements stored in the p arrays $A_i[0 : n_{i,1} - 1]$, $0 \leq i \leq p-1$, is considered and hence an output array $A'_i[0 : \frac{n}{p} - 1]$ of P_i is generated. The array A'_i may have km or $(k-1)m$ useful elements, where $k = \frac{n}{pm}$ (steps 2 and 3). Next, processors with $(k-1)m$ useful elements read m elements from appropriate processors (Step 4). The details are given in the next algorithm. We assume for simplicity that all of n, p, and m are powers of two.

Algorithm Load_Balancing

Input : Each processor P_i contains an input array $A_i[0 : n_i - 1]$.
Output : The elements are redistributed in such a way that $\frac{n}{p}$ elements are stored in the output array $A'_i[0 : \frac{n}{p} - 1]$ of P_i, for $0 \leq i \leq p-1$.

begin

[Step 1] Each processor P_i reads the $p-1$ values $n_0, \ldots, n_{i-1}, n_{i+1}, \ldots, n_{p-1}$ held in the remaining processors. This step can be performed in at most $2\tau + m^2 + p$ communication time by Lemma 3.2.

[Step 2] Processor P_i performs the following local computations:

```
2.1 for $j = 0$ to $p-1$ do
        { $n_{j,1} = \lfloor \frac{n_j}{m} \rfloor m$;     $n_{j,2} = n_j - n_{j,1}$;}
2.2 Compute the prefix sums $s_0, s_1, \ldots, s_{p-1}$ of $n_{0,1}, n_{1,1}, \ldots, n_{p-1,1}$, and
    compute $t = \lfloor \frac{s_{p-1} - n + pm}{m} \rfloor$;
2.3 if $(i < t)$ then
        { $l_i = \min \{j | i\frac{n}{p} < s_j\}$;
          $r_i = \min \{j | (i+1)\frac{n}{p} \leq s_j\}$;
        }
    else
        { $l_i = \min \{j | i(\frac{n}{p} - m) + tm < s_j\}$;
        $r_i = \min\{j | (i+1)(\frac{n}{p} - m) + tm \le s_j\}$;
    }
```

Remark: The index t is chosen in such a way that, for $i < t$, processor P_i will read $\frac{n}{p}$ elements, and for $i \ge t$, P_i will read $\frac{n}{p} - m$ elements. The indices l_i and r_i will be used in the next step to determine the locations of the $\frac{n}{p}$ or $\frac{n}{p} - m$ elements that will be moved to P_i. Notice that this step takes $O(p)$ computation time.

[Step 3] Processor P_i reads $A_{l_i+1}, A_{l_i+2}, \ldots, A_{r_i-1}$, and reads appropriate numbers of elements from A_{l_i} and A_{r_i} respectively.

Remark: This step needs a special attention since there are cases when a set of consecutive processors read their elements from one processor, say P_i. Assume that h processors, $P_{i'}, P_{i'+1}, \ldots, P_{i'+h-1}$, have to read some appropriate elements from P_i. Notice that $h \le \lceil \frac{M}{n/p-m} \rceil + 1$. Then this step can be divided into two substeps as follows: In the first substep, $h-1$ processors, $P_{i'}, P_{i'+1}, \ldots, P_{i'+h-2}$, read their elements from each such P_i; this substep can be done in $\tau + M$ communication time by applying Corollary 3.1. In the second substep, the rest of the routing is performed by using a sequence of read prefetch operations since the remaining elements in each processor are accessed only by a single processor. Hence the total communication time required by this step is $(\tau + M) + (\tau + \frac{n}{p}) = 2\tau + M + \frac{n}{p}$.

[Step 4] Processor P_i, $(i \ge t)$, reads the remaining m elements from the appropriate processors; the corresponding indices l'_i and r'_i can be computed locally as in Step 2.

Remark: This step can be completed in $(\tau + m^2)$ communication time since each processor reads its m elements from at most m processors, and these reads can be prefetched.

end

Thus, one can show the following theorem.

Theorem 4.1 *The load balancing of n elements over p processors, such that at most M elements reside in any processor, can be realized in $5\tau + M + \frac{n}{p} + p + 2m^2$ communication time .* □

5 Sorting, FFT, and Matrix Multiplication

In this section, we consider the three basic computational problems of sorting, FFT, and matrix multiplication, and present communication efficient algorithms to solve these problems on the BDM model. The basic strategies used are well-known but the implementations on our model require a careful attention to several technical details.

5.1 Sorting

We first consider the sorting problem on the BDM model. Three strategies seem to perform best on our model; these are (1) column sort [15], (2) sample sort (see e.g. [6] and related references), and (3) rotate sort [18]. It turns out that the column sort algorithm is best when $n \geq 2p(p-1)^2$, and that the sample sort and the rotate sort are better when $p^2 \leq n < 2p(p-1)^2$.

The column sort algorithm is particularly useful if $n \geq 2p(p-1)^2$; it can be implemented in at most $(4\tau + 3\frac{n}{p} + 2pm)$ communication time with $O(\frac{n \log n}{p})$ computation time. When $n < 2p(p-1)^2$, the column sort algorithm is not practical since its constant term grows exponentially as n decreases. The sample sort algorithm is provably efficient when $p^2 < \frac{n}{6 \ln n}$; it can be implemented in $(3\tau + (p-1)\lceil \frac{5 \ln n}{m} \rceil m + 6\frac{n}{p})$ communication time and $O(\frac{n \log n}{p})$ computation time with high probability. The rotate sort algorithm can be implemented in $8(\tau + \frac{n}{p} + pm)$ communication time with $O(\frac{n \log n}{p})$ computation time, whenever $n \geq 6p^2$.

We begin our description with the column sort algorithm.

Column Sort

The column sort algorithm is a generalization of odd-even mergesort and can be described as a series of elementary matrix operations. Let A be an $q \times p$ matrix of elements where $qp = n$, p divides q, and $q \geq 2(p-1)^2$. Initially, each entry of the matrix is one of the n elements to be sorted. After the completion of the algorithm, A will be sorted in column major order form. The column sort algorithm has eight steps. In steps 1, 3, 5, and 7, the elements within each column are sorted. In steps 2 and 4, the entries of the matrix are permuted. Each of the permutations is similar to matrix transposition of Lemma 3.3. Since $q = \frac{n}{p}$ in this case, these two steps can be done in $2(\tau + (p-1)\lceil \frac{n}{mp^2} \rceil m) \leq 2(\tau + \frac{n}{p} + pm)$ communication time. Each of steps 6 and 8 consists of a $\frac{q}{2}$-shift operation which can be clearly done in $\tau + \frac{n}{2p}$ communication time. Hence the column sort algorithm can be implemented on our model within $(4\tau + 3\frac{n}{p} + 2pm)$ communication time and $O(\frac{n \log n}{p})$ computation time. Thus, we have the following theorem.

Theorem 5.1 *Given $n \geq 2p(p-1)^2$ elements such that $\frac{n}{p}$ elements are stored in each of the local memories of a set of p processors, the n elements can be sorted in column major order form in at most $(4\tau + 3\frac{n}{p} + 2pm)$ communication time and $O(\frac{n \log n}{p})$ computation time.* $\square$

Sample Sort

The second sorting algorithm that we consider in this section is the sample sort algorithm which is a randomized algorithm whose running time does not depend on the input distribution of keys but only depends on the output of a random number generator. We describe a version of the sample sort algorithm that sorts on the BDM model in at most $(3\tau + (p-1)\lceil \frac{5 \ln n}{m} \rceil m + 6\frac{n}{p})$ communication time and $O(\frac{n \log n}{p})$ computation time whenever $p^2 < \frac{n}{6 \ln n}$. The complexity bounds are guaranteed with high probability if we use the randomized routing algorithm described in Section 3.

The overall idea of the algorithm has been used in various sample sort algorithms. Our algorithm described below follows more closely the scheme described in [6] for sorting on the connection machine CM-2; however the first three steps are different.

Algorithm Sample_Sort

Input: n elements distributed evenly over a p-processor BDM such that $p^2 < \frac{n}{6 \ln n}$.
Output: The n elements sorted in column major order.

begin

[Step 1] Each processor P_i randomly picks a list of $5 \ln n$ elements from its local memory.

[Step 2] Each processor P_i reads all the samples from all the other processors; hence each processor will have $5p \ln n$ samples after the execution of this step.

[Step 3] Each processor P_i sorts the list of $5p \ln n$ samples and pick $(5 \ln n + 1)$st, $(10 \ln n + 1)$st, ... samples as the $p - 1$ pivots.

[Step 4] Each processor P_i partitions its $\frac{n}{p}$ elements into p sets, $S_{i,0}, S_{i,1}, \ldots, S_{i,p-1}$, such that the elements in set $S_{i,j}$ belong to the interval between jth pivot and $(j+1)$st pivot, where 0th pivot is $-\infty$, pth pivot is $+\infty$, and $0 \leq j \leq p-1$.

[Step 5] Each processor P_i reads all the elements in the p sets, $S_{0,i}, S_{1,i}, \ldots, S_{p-1,i}$, by using Algorithm Randomized_Routing.

[Step 6] Each processor P_i sorts the elements in its local memory.

end

The following lemma can be immediately deduced from the results of [6].

Lemma 5.1 *For any $\alpha > 0$, the probability that any processor contains more than $\alpha \frac{n}{p}$ elements after Step 5 is at most*

$$n e^{-(1-\frac{1}{\alpha})^2 \frac{5\alpha \ln n}{2}}. \qquad \square$$

Next, we show the following theorem.

Theorem 5.2 *Algorithm Sample_Sort can be implemented on the p-processor BDM in $O(\frac{n \log n}{p})$ computation time and in at most $(3\tau + (p-1)\lceil \frac{5 \ln n}{m} \rceil m + 6\frac{n}{p})$ communication time with high probability, if $p^2 < \frac{n}{6 \ln n}$.*

Proof: Step 2 can be done in $\tau + (p-1)\lceil \frac{5 \ln n}{m} \rceil m$ communication time by using a technique similar to that used to prove Lemma 3.2. By Lemma 5.1, the total number of elements that each processor reads at Step 5 is at most $2\frac{n}{p}$ elements with high probability. Hence, Step 5 can be implemented in $2\tau + 6\frac{n}{p}$ communication time with high probability using Theorem 3.4. The computation time for all the steps is clearly $O(\frac{n \log n}{p})$ with high probability if $p^2 < \frac{n}{6 \ln n}$, and the theorem follows. $\square$

Rotate Sort

The rotate sort algorithm [18] sorts elements on a mesh by alternately applying transformations on the rows and columns. The algorithm runs in a constant number of row-transformation and column-transformation phases (16 phases). We assume here that $n \geq 6p^2$.

Naive implementation of the original algorithm on our model requires 14 simple permutations similar to matrix transpositions, and 14 local sortings within each processor. We slightly modify the algorithm so that the algorithm can be implemented on our model with 8 simple permutations and at most 14 local sortings within each processor. Since each such simple permutation can be performed on our model in $(\tau + \frac{n}{p} + pm)$ communication time, this algorithm can be implemented in $8(\tau + \frac{n}{p} + pm)$ communication time and $O(\frac{n \log n}{p})$ computation time on the BDM model.

For simplicity, we assume that $\frac{n}{p} = 2^s$ and $p = 2^{2t}$, where $s \geq 2t$. The results can be generalized to other values of n and p. A *slice* is a subarray of size $\sqrt{p} \times p$, consisting of all rows i such that $l\sqrt{p} \leq i \leq (l+1)\sqrt{p} - 1$ for some $l \geq 0$. A *block* is a subarray of size $\sqrt{p} \times \sqrt{p}$, consisting of all positions (i, j) such that $l\sqrt{p} \leq i \leq (l+1)\sqrt{p} - 1$ and $r\sqrt{p} \leq j \leq (r+1)\sqrt{p} - 1$ for some $l \geq 0$ and $r \geq 0$.

We now describe the algorithm briefly; all the details appear in [18]. We begin by specifying three procedures, which serve as building blocks for the main algorithm. Each procedure consists of a sequence of phases that accomplish a specific transformation on the array.

Procedure BALANCE: Input array is of size $v \times w$.
(a) Sort all the columns downwards.
(b) Rotate each row i (i mod w) positions to the right.
(c) Sort all the columns downwards.

Procedure UNBLOCK:
(a) Rotate each row i ((i mod $\sqrt{p}$)$\sqrt{p}$) positions to the right.
(b) Sort all the columns downwards.

Procedure SHEAR:
(a) Sort all even-numbered columns downwards and all the odd-numbered columns upwards.
(b) Sort all the rows to the right.

The overall sorting algorithm is the following.

Algorithm ROTATESORT
1. BALANCE the input array of size $\frac{n}{p} \times p$.
2. Sort all the rows to the right.
3. UNBLOCK the array.
4. BALANCE each slice as if it were a $p \times \sqrt{p}$ array lying on its side.
5. UNBLOCK the array.
6. "Transpose" the array.

7. SHEAR the array.
8. Sort all the columns downwards.

For the complete correctness proof of the algorithm, see [18]. We can easily prove that this algorithm can be performed in at most 14 local sorting steps within each processor. We can also prove that each of the simple permutations can be done in $(\tau + \frac{n}{p} + pm)$ communication time in a similar way as in Lemma 3.3. Steps 1, 3, 5, and 6 can each be done with one simple permutation. Steps 2 and 4 also can each be done with one simple permutation by overlapping their second permutations with the first permutations of steps 3 and 5 respectively. Originally, Step 7 is "Repeat SHEAR three times" which is designed for removing six "dirty rows" that are left after Step 5; hence, this step requires 6 simple permutations on our model. Since we assumed $\frac{n}{p} \geq 6p$, the length of each column is larger than that of each row, and we can reduce the number of the applications of SHEAR procedure in Step 7 by transposing the matrix in Step 6. Thus, since the assumption $\frac{n}{p} \geq 6p$ implies that there are at most two dirty columns after the execution of Step 6, one application of procedure SHEAR is enough in Step 7 for removing the two dirty columns and we have the following theorem.

Theorem 5.3 *Given $n \geq 6p^2$ elements, $\frac{n}{p}$ elements in each of the p processors of the BDM model, the n elements can be sorted in column major order form in $8(\tau+\frac{n}{p}+pm)$ communication time and $O(\frac{n \log n}{p})$ computation time.* □

Notice that if $p^2 \leq n < 6p^2$, we need to repeat SHEAR $\lceil \log(1 + \frac{6p^2}{n}) \rceil$ times at Step 7 for removing the dirty columns, and the communication time for Algorithm ROTATESORT is at most $k(\tau + \frac{n}{p} + pm)$, where $k = 6 + 2\lceil \log(1 + \frac{6p^2}{n}) \rceil$.

Other Sorting Algorithms

When the given elements are integers between 0 and $p^{O(1)}$, the local sorting needed in each of the previous algorithms can be done in $O(\frac{n}{p})$ computation time by applying radix sort. Hence we have the following corollary.

Corollary 5.1 *On a p-processor BDM machine, n integers, each between 0 and $p^{O(1)}$, can be sorted in column major order form in $O(\frac{n}{p})$ computation time and in (1) $(4\tau + 3\frac{n}{p} + 2pm)$ communication time if $n \geq 2p(p-1)^2$, (2) $k(\tau + \frac{n}{p} + pm)$ communication time if $n \geq p^2$, where $k = 6 + 2\lceil \log(1 + \frac{6p^2}{n}) \rceil$, or (3) $(3\tau + (p-1)\lceil \frac{5 \ln n}{m} \rceil m + 6\frac{n}{p})$ communication time with high probability, if $p^2 < \frac{n}{6 \ln n}$.* □

Two other sorting algorithms are worth considering: (1) radix sort (see e.g. [6] and related references) and (2) approximate median sort [2, 22]. The radix sort can be performed on our model in $O((\frac{b}{r})\frac{n}{p})$ computation time and $(\frac{b}{r})T_{comm}(n,p)$ communication time, where b is the number of bits in the representation of the keys, r is such that the algorithm examines the keys to be sorted r-bits at a time, and $T_{comm}(n,p)$ is the communication time for routing a general permutation on n elements (and hence the bounds in the above corollary apply). The approximate median sort, which is similar to sample sort with no randomization used but with $p-1$ elements picked from each processor after sorting the elements in each processor, can be done on our model in $O(\frac{n \log n}{p})$ computation time and in at most $3\tau + p^2 + 2\frac{n}{p} + T_{comm}(n,p)$ communication time, if $n \geq 2p(p-1)^2$.

5.2 Fast Fourier Transform

We next consider the Fast Fourier Transform (FFT) computation. This algorithm computes the discrete Fourier transform (DFT) of an n-dimensional complex vector x, defined by $y = W_n x$, where $W_n(j,k) = w_n^{jk}$, for $0 \leq j,k \leq n-1$, and $w_n = e^{i\frac{2\pi}{n}} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$, $i = \sqrt{-1}$, in $O(n \log n)$ arithmetic operations.

Our implementation on the BDM model is based on the following well-known fact (e.g. see [17]). Let the n-dimensional vector x be stored in the $\frac{n}{p} \times p$ matrix X in row-major order form, where p is an arbitrary integer that divides n. Then the DFT of the vector x is given by

$$W_p[\overline{W}_n * W_{\frac{n}{p}} X]^T, \qquad (1)$$

where $\overline{W}_n$ is the submatrix of W_n consisting of the first $\frac{n}{p}$ rows and the first p columns (twiddle-factor scaling), and $*$ is elementwise multiplication. Notice that the resulting output is a $p \times \frac{n}{p}$ matrix holding the vector $y = W_n x$ in column major order form. Equation (1) can be interpreted as computing DFT($\frac{n}{p}$) on each column of X, followed by a twiddle-factor scaling, and finally computing DFT(p) on each row of the resulting matrix.

Let the BDM machine have p processors such that p divides n and $n \geq p^2$. The initial data layout corresponds to the row major order form of the data , i.e., the local memory of processor P_i will hold $x_i, x_{i+p}, x_{i+2p}, \ldots$, $0 \leq i \leq p-1$. Then the algorithm suggested by (1) can be performed by the following three stages. The first stage involves a local computation of a DFT of size $\frac{n}{p}$ in each processor, followed by the twiddle-factor scaling (elementwise multiplication by $\overline{W}_n$). The second stage is a communication step that involves a matrix transposition as outlined in Lemma 3.3. Finally, $\frac{n}{p^2}$ local FFTs each of size p are sufficient to complete the overall FFT computation on n points. Therefore we have the following theorem.

Theorem 5.4 *Computing an n-point FFT can be done in $O(\frac{n \log n}{p})$ computation time and $(\tau + (p-1)\lceil\frac{n}{p^2 m}\rceil m)$ communication time if $n \geq p^2$. When mp^2 divides n, the communication time reduces to $\tau + (\frac{n}{p} - \frac{n}{p^2})$.* □

Remark: The algorithm described in [8] can also be implemented on our model within the same complexity bounds. However our algorithm is somewhat simpler to implement. □

5.3 Matrix Multiplication

We finally consider the problem of multiplying two $n \times n$ matrices A and B. We assume that $p \leq \frac{n^3}{\log n}$. We partition each of the matrices A and B into $p^{\frac{2}{3}}$ submatrices, say

$A = (A_{i,j})_{0 \le i,j \le p^{1/3}-1}$ and $B = (B_{i,j})_{0 \le i,j \le p^{1/3}-1}$, where each of $A_{i,j}$ and $B_{i,j}$ is of size $\frac{n}{p^{1/3}} \times \frac{n}{p^{1/3}}$ assuming without loss of generality that $p^{\frac{1}{3}}$ is an integer that divides n. For simplicity we view the processors indices are arranged as a cube of size $p^{\frac{1}{3}} \times p^{\frac{1}{3}} \times p^{\frac{1}{3}}$, that is, they are given by $P_{i,j,k}$, where $0 \le i,j,k \le p^{\frac{1}{3}} - 1$. We show that product $C = AB$ can be computed in $O(\frac{n^3}{p})$ computation time and $6(2\tau\lceil\frac{\log p}{3\log(\frac{\tau}{m}+1)}\rceil + \frac{n^2}{p^{2/3}})$ communication time. The overall strategy is similar to the one used in [1, 10], where some related experimental results supporting the efficiency of the algorithm appear in [10]. Before presenting the algorithm, we establish the following lemma.

Lemma 5.2 *Given p matrices each of size $n \times n$ distributed one matrix per processor, their sum can be computed in $O(n^2)$ computation time and $2(2\tau\lceil\frac{\log p}{\log(\frac{\tau}{m}+1)}\rceil + n^2)$ communication time.*

Proof: We partition the p processors into $\frac{p}{k}$ groups such that each group contains k processors, where $k = \frac{\tau}{m} + 1$. Using the matrix transposition algorithm, we put the first set of n/k rows of each matrix in a group into the first processor of that group, the second set of n/k rows into the second processor, and so on. Then for each processor, we add the k submatrices locally. At this point, each of the processors in a group holds an $\frac{n}{k} \times n$ portion of the sum matrix corresponding to the initial matrices stored in these processors. We continue with the same strategy by adding each set of k submatrices within a group of k processors. However this time the submatrices are partitioned along the columns resulting in submatrices of size $\frac{n}{k} \times \frac{n}{k}$. We repeat the procedure $\lceil\frac{\log p}{\log(\frac{\tau}{m}+1)}\rceil$ times at which time each processor has an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ portion of the overall sum matrix. We then collect all the submatrices into a single processor. The complexity bounds follow as in the proof of Theorem 3.3. □

Algorithm Matrix_Multiplication

Input: Two $n \times n$ matrices A and B such that $p \le \frac{n^3}{\log n}$. Initially, submatrices $A_{i,j}$ and $B_{i,j}$ are stored in processor $P_{i,j,0}$, for each $0 \le i,j \le p^{\frac{1}{3}} - 1$.

Output: Processor $P_{i,j,0}$ holds the submatrix $C_{i,j}$, where $C = A \times B = (C_{i,j})_{0 \le i,j \le p^{1/3}-1}$, and each $C_{i,j}$ is of size $\frac{n}{p^{1/3}} \times \frac{n}{p^{1/3}}$.

begin

[Step 1] Processor $P_{i,j,k}$ reads blocks $A_{i,j}$ and $B_{j,k}$ that are initially stored in processors $P_{i,j,0}$ and $P_{j,k,0}$ respectively. Each such block will be read concurrently by $p^{1/3}$ processors. This step can be performed in $4(2\tau\lceil\frac{\log p}{3\log(\frac{\tau}{m}+1)}\rceil + \frac{n^2}{p^{2/3}})$ communication time by using the first algorithm described in Theorem 3.3.

[Step 2] Each processor multiplies the two submatrices stored in its local memory. This step can be done in $O(\frac{n^3}{p})$ computation time.

[Step 3] For each $0 \le i,j \le p^{\frac{1}{3}} - 1$, the sum of the product submatrices in the $p^{\frac{1}{3}}$ processors $P_{i,j,k}$, for $k = 0, \ldots, p^{\frac{1}{3}} - 1$, is computed and stored in processor $P_{i,j,0}$.

This step can be done in $O(\frac{n^2}{p^{2/3}})$ computation time and in $2(2\tau\lceil\frac{\log p}{3\log(\frac{\tau}{m}+1)}\rceil + \frac{n^2}{p^{2/3}})$ communication time using the algorithm described in Lemma 5.2.

end

Therefore we have the following theorem.

Theorem 5.5 *Multiplying two* $n \times n$ *matrices can be completed in* $O(\frac{n^3}{p})$ *computation time and* $6(2\tau\lceil\frac{\log p}{3\log(\frac{\tau}{m}+1)}\rceil + \frac{n^2}{p^{2/3}})$ *communication time on the p-processor BDM model, whenever* $p \le \frac{n^3}{\log n}$. $\square$

Remark: We could have used the second algorithm described in Theorem 3.3 to execute Steps 1 and 3 of the matrix multiplication algorithm. The resulting communication bound would be at most $6[\tau + \frac{n^2}{p^{2/3}} + p^{1/3}m]$. $\square$

References

[1] A. Aggarwal, A. Chandra, and M. Snir, *On Communication Latency in PRAM Computations*, Proc. 1st ACM Symp. on Parallel Algorithms and Architectures, pp. 11-21, June 1989.

[2] A. Aggarwal, A.K. Chandra, and M. Snir, *Hierarchical Memory with Block Transfer*, Proc. 28th Annual Symp. on Foundations of Computer Science, pp. 204-216, Oct. 1987.

[3] Anant Agarwal, B.-H. Lim, D. Kranzs, and J. Kubiatowicz, *APRIL: A Processor Architecture for Multiprocessing*, Proc. of the 17th Annual International Symp. on Computer Architecture, pp. 104-114, May 1990.

[4] A. Bar-Noy and S. Kipnis, *Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems*, Proc. 4th Symp. on Parallel Algorithms and Architectures, pp. 13-22, July 1992.

[5] A. Bar-Noy and S. Kipnis, *Multiple Message Broadcasting in the Postal Model*, Proc. 7th International Parallel Processing Symp., pp. 463-470, April 1993.

[6] G.E. Blelloch et.al., *A Comparison of Sorting Algorithms for the Connection Machine CM-2*, Proc. 3th Symp. on Parallel Algorithms and Architectures, pp. 3-16, July 1991.

[7] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie, *Overview of the KSRI Computer System*, TR KSR_TR_9202001, Kendall Square Rescard, Boston, Feb. 1992.

[8] D. Culler et. al., *LogP: Toward a Realistic Model of Parallel Computation*, Proc. 4th ACM PPOPP, pp. 1-12, May 1993.

[9] P.B. Gibbons, *Asynchronous PRAM Algorithms*, a chapter in Synthesis of Parallel Algorithms, J.H. Reif, editor, Morgan-Kaufman, 1990.

[10] A. Gupta and V. Kumar, *Scalability of Parallel Algorithms for the Matrix Multiplication*, 1993 International Conference on Parallel Processing, Vol. III, pp. 115-123.

[11] E. Hagersten, S. Haridi, and D. Warren, *The Cache-Coherence Protocol of the Data Diffusion Machine*, M. Dubois and S. Thakkar, editors, Cache and Interconnect Architectures in Multiprocessors, Kluwer Academic Publishers, 1990.

[12] J. JáJá and K.W. Ryu, *Load Balancing and Routing on the Hypercube and Related Networks*, Journal of Parallel and Distributed Computing 14, pp. 431-435, 1992.

[13] R.M. Karp, A. Sahay, E.E. Santos, K.E. Schauser, *Optimal Broadcast and Summation in the LogP Model*, Proc. 5th Symp. on Parallel Algorithms and Architectures, pp. 142-153, July 1993.

[14] C.P. Kruskal, L. Rudolph, and M. Snir, *A Complexity Theory of Efficient Parallel Algorithms*, Theoretical Computer Science 71, pp. 95-132, 1990.

[15] T. Leighton, *Tight bounds on the Complexity of Parallel Sorting*, IEEE Trans. Comp. C-34(4), pp. 344-354, April 1985.

[16] D. Lenoski et. al., *The Stanford Dash Multiprocessor*, IEEE Computer 25(3), pp. 63-79, March 1992.

[17] C.V. Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, 1992.

[18] J.M. Marberg and E. Gafni, *Sorting in Constant Number of Row and Column Phases on a Mesh*, Algorithmica 3, pp. 561-572, 1988.

[19] K. Mehrotra, S. Ranka, and J.-C. Wang, *A Probabilistic Analysis of a Locality Maintaining Load Balancing Algorithm*, Proc. 7th International Parallel Processing Symp., pp. 369-373, April 1993.

[20] S. Rajasekaran and T. Tsantilas, *Optimal Routing Algorithms for Mesh-connected Processor Arrays*, Algorithmica (8), pp. 21-38, 1992.

[21] K.W. Ryu and J. JáJá, *Efficient Algorithms for List Ranking and for Solving Graph Problems on the Hypercube*, IEEE Trans. Parallel and Distributed Systems 1(1), pp. 83-90, Jan. 1990.

[22] H. Shi and J. Schaeffer, *Parallel Sorting by Regular Sampling*, J. of Parallel and Distributed Computing 14, pp. 361-372, 1992.

[23] H.J. Siegel et. al., *Report of the Purdue Workshop in Grand Challenges in Computer Architecture for the Support of High Performance Computing*, J. of Parallel and Distributed Computing 16(3), pp. 198-211, 1992.

[24] L. G. Valiant, *A Bridging Model for Parallel Computation*, CACM 33(8), pp. 103-111, Aug. 1990.