6-1-1995

Minimizing Communication Overhead for Matrix Inversion Algorithms on Hypercubes

Xiaodong Wang

Purdue University School of Electrical and Computer Engineering

Vwani P. Roychowdhury

Purdue University School of Electrical and Computer Engineering

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

Wang, Xiaodong and Roychowdhury, Vwani P., "Minimizing Communication Overhead for Matrix Inversion Algorithms on Hypercubes" (1995). *ECE Technical Reports*. Paper 138.
<http://docs.lib.purdue.edu/ecetr/138>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

MINIMIZING COMMUNICATION OVERHEAD FOR MATRIX INVERSION ALGORITHMS ON HYPERCUBES

XIAODONG WANG
VWANI P. ROYCHOWDHURY

TR-ECE 95-17
JUNE 1995

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

Minimizing Communication Overhead for Matrix Inversion Algorithms on Hypercubes

Xiaodong Wang¹ and Vwani P. Roychowdhury²
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, USA

¹ **xiaodong@ecn.purdue.edu**

² **vwani@ecn.purdue.edu**

Abstract

The main contribution of this report is the development of novel algorithms that make efficient use of the communication system in distributed memory architectures with processing elements interconnected by a hypercube network. These algorithms achieve almost optimal overlap of communication delays by computation, leading to a minimization of communication overhead. Rigorous analytical and numerical performance analysis of our parallel algorithms are presented as well. The parallel algorithm under study in this report is the parallel Gauss-Jordan matrix inversion algorithm. Many of the ideas introduced in this report, however, apply to other linear algebra algorithms as well.

Parallel Gauss-Jordan matrix inversion algorithms on the hypercube multiprocessors have been extensively studied in the literature. Two common data partitioning strategies for matrix algorithms are row-wise partitioning and submatrix partitioning. It has been claimed that for the parallel GJ inversion algorithm, submatrix partitioning scheme exhibit communication overhead advantages not shared by partitions limited to rows or columns. Most parallel algorithms proposed in the literature, however, do not attempt to overlap inter-processor communication by computation. As a result, the formula $\text{execution time} = \text{computation time} + \text{communication time}$ is used to analyze the complexity of the parallel algorithm. However, during most of the communication time, the processors are actually idle, waiting for the data to arrive.

Most commercially available parallel machines provide communication interrupt handling capability. By utilizing this feature, we believe a lot of parallel matrix algorithms can be improved by overlapping interprocessor communication and computation. In this report we propose and analyze new parallel GJ inversion algorithms under different data partitioning strategies, with or without partial pivoting.

We first propose a new broadcasting algorithm on the hypercube multiprocessor for parallel GJ algorithm. This algorithm ensures that the data are sent out from the source and arrives at the destinations at the earliest possible time. We then give the parallel GJ inversion algorithm using row partitioning. The strategy to overlap communication by computation is for each processor to compute and send out the data needed by the other processors as early as possible. We prove a lower bound on the matrix size such that data transmission is fully overlapped by computation. We also prove that the message length in the input buffer of each processor is at most 2.

We also consider the algorithms under submatrix partitioning, with or without pivoting. We show that when submatrix partitioning is used, even when the communication is fully overlapped by computation, the communication overhead is larger than when using row partitioning. Thus, we show that by minimizing communication overhead, the row partitioning scheme can indeed have better overall performance than the submatrix partitioning scheme.

Finally we extend the idea of overlapping communication and computation to the parallel LU factorization algorithm.

Contents

1	Introduction	1
1.1	Assumptions on the Machine Model	1
1.2	Algorithmic Communication Model	2
1.3	Related Work on Parallel Gauss-Jordan Matrix Inversion Algorithm	3
1.4	Motivation for a New GJ Inversion Algorithm	4
1.4.1	Sequential GJ inversion algorithm with column interchanges	4
1.4.2	A brief review of the subcube-grid parallel GJ inversion algorithm in [1] . . .	4
1.5	Strategy for Overlapping of Communication and Computation	5
2	Summary of Results	7
3	Algorithm I: Row Partitioning With Pivoting	9
4	The SBT; Broadcasting Algorithm	11
5	Analysis of Algorithm I	17
6	Algorithm II: Submatrix Partitioning, Without Pivoting	29
7	Analysis of Algorithm II	33
8	Algorithm III: Submatrix Partitioning, With Pivoting	37
9	Scalability Analysis	39
9.1	The Isoefficiency Metric of Scalability	39
9.2	The Isospeed Metric of Scalability	39
10	LU Factorization	41
11	Concluding Remarks	45

1 Introduction

One of the challenges for large-scale parallel processing is to minimize communication overhead. Although one can attempt to reduce it via architectural innovations, physical reality dictates that remote **access** will always be significantly slower than local access. Software **techniques** for reducing overhead are therefore essential.

An efficient approach to reducing communication overhead is to allow communication to overlap computation. This helps reducing overhead in two ways: First, when one process is waiting for data from a remote process, another ready process maybe scheduled for execution. Secondly, even a single process may wait for multiple data items simultaneously, and continue execution whenever any of the expected items arrive [32]. Research projects addressing overlap of communication and computation have been carried out at both hardware and software levels. Research prototypes of message driven architectures, such as the J-Machine [33] and Monsoon [34], expend a significant amount of hardware to integrate communication with computation. The message driven execution style promotes the overlap of communication and computation, and exhibits impressive communication performance. For the current message passing **architectures**, a communication mechanism, active message, has been proposed [31]. This model minimizes the software overhead in message passing machines and achieves overlap of communication and computation.

The approaches mentioned above deal with the communication overhead from a general engineering perspective. We, however, in this report, treat the problem of minimizing communication overhead from an algorithmic perspective. That is, given a machine model **and** a computational task, how **can** one develop an efficient parallel algorithm which achieves overlap of communication and **computation**, and incurs minimum communication overhead. The technique used in this report is **analytic**, which provides rigorous quantitative **evaluation** on **the** performance of the parallel algorithm. We consider the parallel Gauss-Jordan matrix inversion algorithm on message passing hypercube multiprocessors.

1.1 Assumptions on the Machine Model

Throughout this report, we make the following assumptions about the machine model:

1. The parallel machine has binary-cube network with all-port **communication** capability, **i.e.**, the **hardware** supports simultaneous communication on all the channel;. Machines with hypercube network that support all-port **communication** include CM-2, Chi-200 and **nCUBE** 2.
2. It is **a** multiple instruction multiple data (MIMD) parallel machine. **MIMD** machines include **nCUBE** 2, **iPSC**, CM 2, Paragon, etc.
3. The interprocessor communication is through DMA **controlled** asynchronous message passing. Machines with this feature include **nCUBE** 2, **iPSC**, Paragon, etc.

4. The software supports user specified interrupt handling. Intel iPSC, Intel Paragon and CM 5 have this feature.

1.2 Algorithmic Communication Model

The most common cost model used in algorithm design for large-scale **multiprocessor** assumes that the program alternates between computation and communication phases, and **that** communication requires time linear in the size of the message, plus a setup cost. Consider a simple scenario in which all **the** processors have the same computation and communication loads, and the program alternates between computation and communication phases in a synchronous manner. Thus, the time **to run** a program is $T = T_{compute} + T_{communicate}$ and $T_{communicate} = \sum_{i=1}^{N_c} (t_s + t_w L_i)$, where t_s is the **setup** cost, t_w is the time per byte, L_i is the i^{th} message length, and N_c is the total number of communications. To achieve 90% of the peak processor performance, the algorithm must be tailored to achieve a sufficiently high ratio of computation to communication, i.e., $T_{compute} \geq 9T_{communicate}$. A high performance network is required to minimize the **communication** time, and it sits 90% idle [31].

If **communication** and computation are fully overlapped, the situation is very different. The time to run a program becomes $T = \max(T_{compute} + N_c t_s, \sum_{i=1}^{N_c} t_w L_i)$. Thus, to achieve high processor efficiency, the data transmission and computation time need only balance, **and** the computation time need only swamp the setup overhead, i.e., $T_{compute} \gg N_c t_s$ [31].

Next we introduce the communication model used this **report**, and the concept of full overlap of **communication** and computation. Assuming that at time T_1 , processor P_1 starts sending a message of length m to its *adjacent* processor P_2 (i.e., P_1 and P_2 is directly connected by communication channel). P_1 will first spend time t_s to set up the **communication** channel. The message will then take additional time $t_w m$ to reach P_2 , i.e., the message arrives at P_2 at time $T_1 + t_s + t_w m$. P_1 can return to computation after setting up the **communication** channel, i.e., at time $T_1 + t_s$. Now assume that at time T_2 , processor P_2 needs to read this message from P_1 . If this message has already been in the input buffer of P_2 , i.e., this message was sent out by P_1 at time T_1 such that $T_1 + (t_s + t_w m) \leq T_2$, then P_2 can read this message **without** any delay. Here we assume it takes negligible time to read the message in the input buffer. **Otherwise**, P_2 will remain idle until the message **arrives** in its input buffer. The idle time is $t_{idle} = T_1 + (t_s + t_w m) - T_2$. Therefore the **communication** overhead of each processor contains two parts: **the setup** time, **when** the processor sets up the communication channel, and the idle time, **when the** processor is idle, waiting for message **to arrive**.

In a **parallel** computation process on a multiprocessor, if for every processor, the idle time is always **zero** throughout the process (that is, each processor is **either** doing computation, or setting up the communication channel, but never idle), then we say *communication is fully overlapped by computation* in this process. Hence our objective is to accomplish this **full** overlap of communication and **computation**, and at the same time, to minimize the **total setup** overhead.

1.3 Related Work on Parallel Gauss-Jordan Matrix Inversion Algorithm

Matrix operation, such as inversion and factorization, is a standard part of any linear algebra library. Such libraries are critical for the evolution of parallel computer into useful tools for large scale scientific computations. Two common sequential algorithms for dense matrix inversion are Gaussian Elimination (GE) and Gauss-Jordan (GJ) Elimination. They have the same operation count (FLOPs) for inversion. It has been shown that GJ is significantly more efficient in parallel than GE, because parallel GJ has a more homogeneous work load distribution and incurs less communication overhead [12]. Furthermore, GJ inversion is an in-place matrix inversion algorithm. Thus no additional memory is required as work place during the process of GJ elimination.

Several parallel algorithms have been developed for implementing GJ elimination with or without pivoting on distributed-memory architectures. Some of them are fine-grain algorithms developed for hypercubes assumed to have little setup cost and high communication bandwidth, e.g., the Caltech Hypercube [11, 12]. The others are medium-grain algorithms developed for hypercubes with substantial message passing latency, e.g., the commercially available Intel iPSC/1, iPSC/2, iPSC/860 and nCUBE 2 hypercubes [1, 17]. In [1] some of the algorithms representing recent developments in both classes are reviewed.

It was shown in [2] that the GJ algorithm has essentially different properties when using column interchanges instead of row interchanges for improving numerical stability. A more satisfactory bound for the residual norm can be derived for solutions obtained by GJ with column interchanges instead of row interchanges. In [1] a medium-grain parallel algorithm for implementing GJ inversion with column interchanges on a hypercube multiprocessor was proposed. The hypercube network was configured as a two-dimensional subcube-grid to support submatrix partitionings. The authors claimed that for GJ inversion with column interchanges on hypercubes with substantial communication latency, submatrix partitions had communication advantages not shared by partitions limited to rows or columns. In their algorithm, no attempt was made to overlap communication and computation.

Two popular optimization techniques applied to parallel matrix computations for improving concurrence are the send-ahead strategy that attempt to communicate data as early as possible, and the *compute-and-send-ahead* strategy that computes in advance data which have to be communicated [3, 4, 5]. Their most important aim is to avoid the processors idleness due to the data waiting time by overlapping communication and computation. In [3] these two optimization strategies were applied to parallel LU decomposition algorithms, with or without partial pivoting. In their implementation, they used the communication subroutine provided by the software, and it is not clear how the overlap is achieved. No analytical performance analysis was provided in their paper. Their experimental results also showed the superiority of the square scattered decomposition strategy.

1.4 Motivation for a New GJ Inversion Algorithm

1.4.1 Sequential GJ inversion algorithm with column interchanges

We first give the sequential GJ algorithm with column interchanges for inverting an $N \times N$ matrix $A = [a_{i,j}]$. This algorithm performs in-place inversion. An array $\sigma_{1:N}$ is used to record the permutation of the columns. Initially $\sigma_i = i$, $i = 1, 2, \dots, N$. At the k -th step of GJ reduction, if the pivot column is found to be column $pivot$, then columns k and $pivot$ need to be interchanged, and we record this interchange by swapping the value of σ_k and σ_{pivot} . Note that in the algorithm we actually do not interchange columns, but rather, we just address the permuted columns using the information saved in σ . In this way, some intermediate data movement can be saved. As a result, A is overwritten by a row and column permuted form of A^{-1} . Let $B = [b_{i,j}] = A^{-1}$, then after applying the algorithm to A , $a_{i,j} = b_{k,l}$, where $k = \sigma_j$, and $\sigma_l = j$.

```

for k = 1 to N
    pivot = k
    for j = k + 1 to N
        if  $|a_{k,\sigma_j}| > |a_{k,\sigma_{pivot}}|$  then pivot = j
     $\sigma_k \longleftrightarrow \sigma_{pivot}$ 

    for j = 1 to N and j  $\neq$  k
         $a_{k,\sigma_j} = a_{k,\sigma_j} / a_{k,\sigma_k}$ 
        for i = 1 to N and i  $\neq$  k
             $a_{i,\sigma_j} = a_{i,\sigma_j} - a_{i,\sigma_k} * a_{k,\sigma_j}$ 

        for i = 1 to N and i  $\neq$  k
             $a_{i,\sigma_k} = -a_{i,\sigma_k} / a_{k,\sigma_k}$ 
         $a_{k,\sigma_k} = 1 / a_{k,\sigma_k}$ 

```

1.4.2 A brief review of the subcube-grid parallel GJ inversion algorithm in [1]

We next briefly review the GJ inversion algorithm proposed in [1] (Thereby we call this algorithm Algorithm 0). The matrix is partitioned into submatrices, and the data are distributed among the processors using full column and row wrap-mapping. Each column and each row of processors form a subcube. Within the main loop, there are three blocks of code corresponding to the communication of the pivot row; the selection and permutation of the pivot column; and the GJ elimination on the processor's share of matrix.

A hypercube of dimension d is configured as a $\gamma_1 \times \gamma_2 = p = 2^d$ subcube-grid. Each subcube-column has dimension $d_1 = \log \gamma_1$ and each subcube-row has dimension $d_2 = \log \gamma_2$. d_1 and d_2 differ at most by one for optimal performance. At each iteration, there are two types of communication

required for this algorithm: one to broadcast the pivot row and another to find and broadcast the pivot column. The former involves one-to-all broadcast within the subcube-column, and the latter involves permutations within the subcube-row. After every processor receives the appropriate pivot row segment, they each determine the local pivot element and the corresponding pivot column. Every processor has the pivot element and the corresponding segment of the final pivot column after permutations (recursive doubling) within the subcube-row. Since recursive doubling within a subcube of dimension d_2 involves d_2 steps of write-read pairs, this portion of the algorithm causes a *synchronization barrier* for all processors. No processor can start computation before the entire communication is completed. Therefore this algorithm is implemented in a synchronous manner, i.e., in every iteration, a processor starts performing its computation only after the entire data communication is completed.

Assuming that $\gamma_1 = \gamma_2 = \sqrt{p}$, then the communication of the pivot row amounts $d_1 = \frac{1}{2} \log p$ one-hop transfers of a message of size $N/\sqrt{p}$. The selection and permutation of the pivot column amounts to $d_2 = \frac{1}{2} \log p$ one-hop exchanges of a message of size $N/\sqrt{p}$. Therefore, the total communication overhead for each processor incurred in this algorithm is $T_{comm} = N(d_1 + d_2)(t_s + t_w \frac{N}{\sqrt{p}}) = N \log p(t_s + t_w \frac{N}{\sqrt{p}})$.

Since no communication is overlapped by computation, T_{comm} appears as a whole in the total execution time of the algorithm, i.e., $T = T_{comp} + T_{comm}$, where T_{comp} is the total computation time of each processor.

1.5 Strategy for Overlapping of Communication and Computation

The strategy for overlap of communication and computation is to let each processor to compute and send out the data needed by the other processors as early as possible, so that these data would have arrived at the destinations when they are to be used. This is called *compute-and-send-ahead* strategy and we will elaborate on it later when we give the parallel GJ inversion algorithms. The best we can do to this end is to make the idle time of each processor zero, and to make the total setup overhead of each processor as small as possible.

In [3] the *send-ahead* and the *compute-and-send-ahead* strategies have been applied to the parallel LU factorization algorithm to achieve overlap of communication and computation. The main communication operation of the algorithm is broadcasting the pivot rows and the multipliers. Since the data is expected to arrive at the destinations at the earliest possible time, the data broadcast process is of the most importance and should be carefully designed. However, in their paper, the authors took the broadcast within a subcube (which sends data from one processor to all others within the subcube) as a communication primitive, and did not specify the details of this process. And no analytical performance analysis was given for the algorithms in the paper.

To accomplish full overlap of data transmission by computation, we need the high level languages for the parallel machines to provide the interrupt handling capability. Whenever a message arrives at the input buffer of a processor, the processor is interrupted from the current computation and

starts executing the interrupt handling subroutine, which involves forwarding the message to some adjacent processors. After that the processor resumes the previous interrupted computation. This feature guarantees that the message is passed along the communication tree in the hypercube without **delay** and reaches the destinations at the earliest possible time. For example, for one-to-all broadcasting, when the message arrives at a processor, and if this processor is **not** a leaf node of the broadcasting tree, then this processor is interrupted from the current computation and forwards the message to the adjacent nodes on the hypercube which has not got the message, before it resumes **the** interrupted computation. This feature of interrupt handling is crucial for the overlap of communication and computation. Throughout this report we assume the underlying machine model has this feature.

2 Summary of Results

In this report, we show that by utilizing the interrupt handling capability of the parallel machine, a set of **asynchronous** algorithms can be designed such that the data transmission can be partially or fully overlapped by computation. We propose a new **one-to-all** broadcast **algorithm** on hypercube that helps to achieve overlap of communication and computation for the parallel GJ inversion algorithm.

We **give** the new parallel GJ inversion algorithms under different data partitioning strategies (i.e., row partitioning and submatrix partitioning), with or without partial pivoting. For each **algorithm**, we prove a lower bound on the matrix size such that data transmission is fully overlapped by computation, and the total communication overhead when full overlap is achieved. Our conclusion is that when the matrix size is large enough such that data transmission is **fully** overlapped by computation, *the row partitioning strategy has the lowest communication overhead*. We also give the **scalability** analysis for the algorithms discussed in **this** report. Furthermore, we also apply the idea of overlap communication and computation to the parallel LU factorization algorithm.

Partition Strategy	Total Comm. Overhead
row partitioning	$N(t_s + t_w N) \log p$
submatrix partitioning	$N(t_s + t_w \frac{N}{\sqrt{p}}) \log p$

Table 1: **The** communication overhead for the parallel GJ inversion algorithm under the row partitioning and **submatrix** partitioning, when no overlap of communication and computation is **attempted**. The matrix size is $N \times N$, t_s is the communication startup overhead, t_w is the data transmission rate, p is the number of **processors**.

Partition Strategy	Condition	Total Comm. Overhead
row	$\left(\frac{N}{p} - 3\right) N f \geq \frac{1}{2} p t_s + 2(t_s + t_w N) \log p$	$\frac{1}{2} N t_s$
submatrix, w/o pivoting	$\left(\frac{N^2}{p} - \frac{3N}{\sqrt{p}}\right) f \geq 2\sqrt{p} t_s + 2(t_s + t_w \frac{N}{\sqrt{p}}) \log p$	$N t_s$
submatrix, w/ pivoting	—	$\frac{1}{2} N (1 + \log p) t_s$

Table 2: **Summary** of key theoretical results in this report, for the conditions of full overlap of communication by **computation**, and the total communication overhead of the **algorithm** when **full overlap** is achieved, under the row or **submatrix** partitioning, with or **without** pivoting.

Table 1 lists the communication complexities for the parallel GJ inversion algorithm under the row partitioning and submatrix partitioning, where no overlap of **communication** and computation is attempted [1, 9]. Table 2 summarizes some theoretical results in this report on the conditions of full overlap and the total communication overhead when full overlay is **achieved**.

Figure 1 depicts some numerical results on **the** total **communication** overhead of different **algo-**

rithms. Note that the vertical axis is log scaled. It is clear from this plot that overlapping data communication and computation can greatly reduce the communication overhead of the parallel algorithm. Furthermore, when the matrix size is large enough, row partitioning is superior to submatrix partitioning.

Figure 1: Comparison of the communication overheads of four parallel GJ matrix inversion algorithms under the row or submatrix partitioning, with or without overlap (Number of processors $p=16$).

This report is organized as follows: In section 3 we give our parallel GJ inversion algorithm using row partitioning, with pivoting (Algorithm I). In section 4 we propose a new one-to-all broadcast algorithm on hypercube that helps to achieve overlap of communication and computation. In section 5, we give the performance analysis of Algorithm I. In section 6 we give Algorithm II, which uses submatrix partitioning and without pivoting. The analysis of Algorithm II is given in section 7. In section 8, we discuss Algorithm III, which also uses submatrix partitioning but with pivoting. In section 9, we give the scalability analysis of our parallel algorithms. In section 10 we extend the idea of overlapping communication and computation to parallel LU factorization algorithm. Section 11 concludes this report.

3 Algorithm I: Row Partitioning With Pivoting

In this section, we give our algorithm of parallel GJ inversion with column interchanges, using row partitioning. Suppose we want to invert an $N \times N$ matrix A on a d dimensional hypercube containing, $p = 2^d$ processors. Assume $N = np$. We partition the matrix by rows and use wrap-mapping to distribute the rows of A among processors, i.e., rows $k, p+k, 2p+k, \dots, N-p+k$ are assigned to processor P_k , $1 \leq k \leq p$. We will discuss how to determine the physical address of processor P_k on the hypercube in the next section. We use notation $[k]$ to represent $[(k-1) \bmod p] + 1$, thus $1 \leq [k] \leq p$.

When the matrix is partitioned by rows, each row is entirely within one processor. Thus the search for the pivot element can be done by the single processor which has the pivot row, and the recursive doubling is no longer needed. In the algorithm, during the k -th loop, each processor first gets the k -th pivot row of A and the pivot element from processor $P_{[k]}$, and then updates all the rows assigned to it. In order to overlap the communication by computation, the pivot row should be sent out at the earliest possible time, so that when a processor is to use it, it would have already arrived at that processor. Since processor $P_{[k+1]}$ is assigned row $(k+1)$ of A , we let it update this row at first during the k -th loop. Then it finds the pivot element and normalizes this row (compute-ahead). After that it broadcasts this $(k+1)$ st pivot row and the pivot element to all the other processors, before it resumes to update the other $n-1$ rows using the k -th pivot row (send-ahead). By using this compute-and-send-ahead strategy, the $(k+1)$ st pivot row is transmitted through the communication channels while all the processors are still updating in the k th loop. Thus the overlap of communication and computation is achieved.

Since the message may take several hops to reach the destinations, it needs to be routed to the next processor without delay upon arrival at an intermediate processor. This is realized by a local message routing algorithm and the communication interrupt handling capability of the machine. By the local message routing algorithm, each processor knows where to forward the incoming message to. We will discuss this algorithm in the next section. Whenever a message arrives at a processor, the processor is interrupted from the computation. By identifying the header of the message the processor knows which pivot row this message contains. It will then determine where to forward this message to. If it does not need to forward it, it resumes its computation. Otherwise, it has to forward the message to some of its adjacent processors, and incurs a setup overhead of t_s , before it resumes its computation. If after completing the k -th loop of updating, the $(k+1)$ st pivot row still has not arrived at the processor, then the processor has to wait for it before it can start the $(k+1)$ st updating loop.

The following is pseudo code for the k th step of the parallel GJ inversion algorithm with column interchanges, using row partitioning:

```

if (  $P \neq P_{[k+1]}$  ) then
    read in the  $k$ th pivot row and pivot element;

```


update the $(k + 1)$ st row of A;
find the $(k + 1)$ st pivot element;
 normalize the $(k + 1)$ st pivot row;
 broadcast the $(k + 1)$ st pivot row;
update the rest $(n - 1)$ rows of A;
 if $(P = P_{[k]})$ then
 update the $(n - 1)$ rows of A (other than the kth row);
 if $(P \neq P_{[k+1]} \text{ and } P \neq P_{[k]})$ then
 read in the kth pivot row and pivot element;
 update the n rows of A;
Interrupt handling:
 read the incoming data message;
 forward the incoming message to appropriate processors, if needed;

4 The *SBT*;Broadcasting Algorithm

In order to implement the *compute-and-send-ahead* strategy discussed in the previous section, we need to construct a set of p broadcasting trees in the hypercube. At the k th broadcasting in Algorithm I, the root of the tree is $P_{[k]}$. Furthermore, each broadcasting tree should have the following two features:

1. $P_{[k+1]}$ and $P_{[k]}$ should be directly linked.

As discussed in the previous section, the $(k+1)$ st row of matrix A is broadcast by processor $P_{[k+1]}$ immediately after it is updated and normalized in the k th stage of the parallel GJ inversion algorithm, before $P_{[k+1]}$ starts updating the other rows using the k th pivot row. To broadcast the $(k+1)$ st at the earliest possible time, processor $P_{[k+1]}$ should receive the k th pivot row from processor $P_{[k]}$ at the earliest possible time. This implies that $P_{[k+1]}$ and $P_{[k]}$ should be adjacent on the hypercube.

2. $P_{[k+1]}$ should be a leaf node in the k th broadcasting tree.

After receiving the k th pivot row, $P_{[k+1]}$ should not be involved in forwarding the message to some other processors, so that its computation is not delayed by the startup time overhead, and the $(k+1)$ st row can be sent out at the earliest possible time. This implies that $P_{[k+1]}$ should be a leaf node in the k th broadcasting tree.

We next discuss how to determine the physical address of each processor on the hypercube. Since the logically consecutively numbered processors need to be physically adjacent, i.e., processors $P_{[k+1]}$ and $P_{[k]}$ be adjacent on the hypercube, we address the processors in a binary-reflected Gray code order with the starting address as 0 [7]. Let the d -bit code of $p = 2^d$ integers be *Gray*(d). The binary-reflected Gray code on d bits is recursively defined as follows.

Let $Gray(d) = (g_1^d, g_2^d, \dots, g_{2^{d-1}}^d, g_{2^d}^d)$. Then

$$Gray(d+1) = (0g_1^d, 0g_2^d, \dots, 0g_{2^{d-1}}^d, 1g_2^d, 1g_{2^{d-1}}^d, \dots, 1g_1^d).$$

Alternatively,

$$Gray(d+1) = (g_1^d 0, g_1^d 1, g_2^d 1, g_2^d 0, g_3^d 0, g_3^d 1, \dots, g_{2^{d-1}}^d 1, g_{2^{d-1}}^d 0).$$

We use this binary-reflected Gray code mapping to associate each processor's logical address with its physical address, i.e., let the physical address of processor P_i be g_i^d . For the rest of the report, we will just use P_i to denote the physical address g_i^d .

In [7] the spanning binomial tree (SBT) of a d -cube rooted at node s is defined as follows. Let i be any node, and let $c = i \oplus s$, where $\oplus$ denotes bitwise exclusive OR operation. Let q be such that $c_q = 1$ and $c_m = 0$, $\forall m \in M(c) = \{q+1, q+2, \dots, d-1\}$, and let $q = -1$ if $c = 0$. The set $M(c)$ is the set of leading zeros of c . Then in $SBT(s)$, the children nodes and the parent node of a given node i are

$$\text{children}(i) = \{(i_{d-1}i_{d-2} \dots \bar{i}_m \dots i_0)\}, \forall m \in M(c),$$

$$\text{parent}(i) = \begin{cases} \phi, & i = s, \\ (i_{d-1}i_{d-2} \cdots \bar{i}_q \cdots i_0), & i \neq s. \end{cases}$$

It can be easily verified that the children and parent functions defined above are consistent, i.e., that node j is a child of node i iff node i is the parent of node j [7]. A node is a leaf node if it has no children, i.e., $q = d - 1$.

Lemma 1. In $SBT(s)$, any node i is at level $H(s, i)$, where $H(s, i)$ is the **Hamming** distance between s and i . The number of nodes at level l is $\binom{d}{l}$, and the height of the tree is d . The number of children nodes of any node i is $n_c = d - 1 - q$, and $0 \leq n_c \leq d - H(s, i)$. [7] $\square$

At the Eth broadcasting in Algorithm I, the root of the SBT is $P_{[k]}$. And because of the **binary-reflected** Gray code mapping, $P_{[k+1]}$ is adjacent to $P_{[k]}$ and **therefore** is at the first level of the SBT. However, with the SBT defined above, it is not guaranteed that $P_{[k+1]}$ is a leaf node, since by definition, $P_{[k+1]}$ is a leaf node in $SBT(P_{[k]})$ only if they differ by the high order $(d - 1)$ st bit.

On the hypercube, if node P and node P' differ only by the j th bit (and therefore they are directly linked), $0 \leq j < d$, then we say P is the j th neighbor of P' . Suppose $P_{[k]}$ and $P_{[k+1]}$ differ by the j th bit, we need to construct a SBT rooted at $P_{[k]}$ and with $P_{[k+1]}$ as a leaf node.

Definition 1 A $SBT_j(s)$ in a d -cube is a SBT rooted at node s and the j th neighbor node of s is a leaf node. (Note the $SBT(s)$ defined in [7] is therefore $SBT_{d-1}(s)$.)

Definition 2 The shuffle operation on a node $i = (i_{d-1}i_{d-2} \cdots i_1i_0)$ is defined as $S(i) = (i_{d-2}i_{d-3} \cdots i_1i_0i_{d-1})$. The inverse shuffle operation on i is defined as $S^{-1}(i) = (i_0i_{d-1}i_{d-2} \cdots i_1)$. Moreover, $S^k(i)$ represents applying shuffle operation k times on i , and $S^{-k}(i)$ represents applying inverse shuffle operation E times on i . $\square$

Definition 3 The shuffle operation on a graph $G(V, E)$ is defined as $S(G) = G(S(V), S(E))$, where $S(V) = \{S(i) | \forall i \in V\}$ and $S(E) = \{(S(i), S(j)) | \forall (i, j) \in E\}$. Similarly we define the inverse shuffle operation on a graph $S^{-1}(G) = G(S^{-1}(V), S^{-1}(E))$. $\square$

Lemma 2 Shuffle and inverse shuffle operations of a graph preserve the **Hamming** distance between nodes. The shuffle operation S^k maps every edge in dimension l to dimension $(l + k) \bmod n$. The inverse shuffle operation S^{-k} maps every edge in dimension l to dimension $(l - k) \bmod n$. [7] $\square$

Corollary 1 The topology of a graph remains unchanged under shuffle and inverse shuffle operations. [7] $\square$

The next lemma shows that $SBT_k(s)$ can be obtained by applying shuffle and inverse shuffle operation on $SBT_{d-1}(s)$.

Figure 2: $SBT(Gray(\mathcal{G}))$ is a family of $2^3 = 8$ SBT's. The root node of the $[i+1]$ -st SBT is a leaf node of the i -th SBT at the first level.

Lemma 3

$$SBT_k(s) = S^{-(d-1-k)} \left[SBT_{d-1}(S^{d-1-k}(s)) \right].$$

Proof: By Corollary 1, $S^{-(d-1-k)} \left[SBT_{d-1}(S^{d-1-k}(s)) \right]$ is a SBT rooted at $S^{-(d-1-k)} \left[S^{d-1-k}(s) \right] = s$. Let i' be the leaf node at the first level of $SBT_{d-1}(S^{d-1-k}(s))$, then by definition, i' differs with $S^{d-1-k}(s)$ by the $(d-1)$ st bit. Let i be the corresponding node of i' in $S^{-(d-1-k)} \left[SBT_{d-1}(S^{d-1-k}(s)) \right]$, by Lemma 2, i and s differ by the $(d-1) - (d-1-k) = k$ th bit. Therefore this SBT is $SBT_k(s)$. $\square$

The following theorem shows how to the construct $SBT_k(s)$ directly, by specifying the parent and children functions.

Theorem 1 In $SBT_k(s)$, let q be such that $c_q = 1$ and $c_m = 0$, $\forall m \in M(c) = \{q+1 \bmod p, q+2 \bmod p, \dots, k \bmod p\}$, and let $q = k-d$ if $c = 0$. Then

$$\begin{aligned} \text{children}(i) &= \{(i_{d-1}i_{d-2} \dots \bar{i}_m \dots i_0)\}, \forall m \in M(c), \\ \text{parent}(i) &= \begin{cases} \phi, & i = s, \\ (i_{d-1}i_{d-2} \dots \bar{i}_q \dots i_0), & i \neq s. \end{cases} \end{aligned}$$

Proof: Let $s' = S^{d-1-k}(s)$, $i' = S^{d-1-k}(i)$, $c' = i' \oplus s'$. Clearly $c' = S^{d-1-k}(c)$. Let q' be such that $c'_{q'} = 1$ and $c'_m = 0$, $\forall m \in M(c') = \{q'+1, q'+2, \dots, d-1\}$. Then by definition, in $SBT_{d-1}(s')$, the edges that connect i' and its children nodes are those in dimension $d-1, d-2, \dots, q'+1$. And the edge that connects i' and its parent node is the one in dimension q' . Let $q = q' - (d-1-k) \bmod d$. From the definition of q' and the relation $c' = S^{d-1-k}(c)$, it is easy to verify that q satisfies the constraints given in the theorem. By Lemma 3, $SBT_k(s)$ is obtained by inverse shuffle $SBT_{d-1}(s')$ $(d-1-k)$ times. Then by Lemma 2, in $SBT_k(s)$, the edges that connect i and its children nodes are those in dimension $d-1 - (d-1-k) \bmod d, \dots, q'+1 - (d-1-k) \bmod d = q+1 \bmod d$. And the edge that connects i and its parent node is the one in dimension q . $\square$

The next theorem shows the structure of $SBT_k(s)$.

Theorem 2 In $SBT_k(s)$, any node i is at level $H(s, i)$, where $H(s, i)$ is the Hamming distance between s and i . The number of nodes at level l is $\binom{d}{l}$, and the height of the tree is d . The number of children nodes of any node i is $(k-q) \bmod (d+1)$. The number of leaf nodes is 2^{d-1} .

Proof: The first three statements follow directly Lemma 1, Lemma 2 and Corollary 1. By Theorem 1, in $SBT_k(s)$ node i is a leaf node if and only if the k th bit of i (i_k) differs from the k th bit of s (s_k). For a given root s , there are 2^{d-1} nodes i such that $i_k \neq s_k$. Therefore there are 2^{d-1} leaf nodes in $SBT_k(s)$. $\square$

Next we give the construction of the p broadcasting trees used in Algorithm I.

Definition 4 $SBT(Gray(d))$ is a family of $p = 2^d$ SBT's:

$$SBT(Gray(d)) = (SBT^{(1)}, SBT^{(2)}, \dots, SBT^{(2^d)}),$$

where $Gray(d) = (g_1^d, g_2^d, \dots, g_{2^d-1}^d, g_{2^d}^d)$, $SBT^{(i)} = SBT_{j(i)}(g_i^d)$, where g_i^d and $g_{[i+1]}^d$ differ by the $j(i)$ -th bit. $\square$

As an example, consider $d = 3$. We have

$$Gray(3) = (000, 001, 011, 010, 110, 111, 101, 100),$$

and Figure 2 shows the corresponding SBT's of $SBT(Gray(3))$.

The next theorem shows the structure of the set of trees, and will lead to the setup overhead complexity of the broadcasting algorithm.

Theorem. 3 In $SBT(Gray(d))$, each node appears at the l -th level of $\binom{d}{l}$ SBT's. Moreover, each node appears as a leaf node in 2^{d-1} SBT's.

Proof: By Theorem 2, in $SBT_k(s)$, any node i is at level $H(s, i)$. For any fixed i , there are $\binom{d}{l}$ s 's such that $H(s, i) = l$. And in the corresponding SBT 's rooted at these nodes, i is at level l .

To prove each node appears as a leaf node in 2^{d-1} SBT 's, we use proof by induction on the hypercube dimension d . We use the first definition of binary-reflected Gray code. Induction hypothesis: In $SBT(g_1, g_2, \dots, g_{2^d})$ or $SBT(g_{2^d}, \dots, g_2, g_1)$, any node i appears as leaf node in 2^{d-1} SBT 's.

Base case: $d = 1$. The proposition is obviously true.

Induction Step: Assume the proposition is true for dimension d , we show it is also true for dimension $d + 1$.

For $SBT(g_1, g_2, \dots, g_{2^d})$, let

$$\phi_d(i, g_m) = \begin{cases} 1, & i \text{ and } g_m \text{ differ by the } j(m)\text{th bit,} \\ 0, & \text{otherwise.} \end{cases}$$

where g_m and $g_{[m+1]}$ differ by the $j(m)$ th bit.

And for $SBT(g_{2^d}, \dots, g_2, g_1)$, let

$$\phi'_d(i, g_m) = \begin{cases} 1, & i \text{ and } g_m \text{ differ by the } j'(m)\text{th bit,} \\ 0, & \text{otherwise.} \end{cases}$$

where g_m and $g_{[m-1]}$ differ by the $j'(m)$ th bit.

Then by the induction hypothesis,

$$\sum_{m=1}^{2^d} \phi_d(i, g_m) = \sum_{m=2^d}^1 \phi'_d(i, g_m) = 2^{d-1}$$

For the $d + 1$ dimension, we have $SBT(0g_1, \dots, 0g_{2^d}, 1g_2^d, \dots, 1g_1)$, for any node O_i , we have

$$\begin{aligned} & \sum_{m=1}^{2^{d+1}} \phi_{d+1}(0i, g_m^{d+1}) \\ = & \sum_{m=1}^{2^d-1} \phi_d(i, g_m) + 0 + \sum_{m=2^d}^2 \phi'_d(i, g_m) + 1 \\ = & \sum_{m=1}^{2^d} \phi_d(i, g_m) + \sum_{m=2^d}^1 \phi'_d(i, g_m) + 1 - (\phi_d(i, g_{2^d}) + \phi'_d(i, g_1)) \\ = & 2^{d-1} + 2^{d-1} + 1 - (\phi_d(i, g_{2^d}) + \phi'_d(i, g_1)). \end{aligned}$$

Suppose g_{2^d} and g_1 differ by the j th bit. If $\phi_d(i, g_{2^d}) = 1$ then the j th bits of i and g_{2^d} are different, so the j th bits of i and g_1 are the same, and $\phi'_d(i, g_1) = 0$. Similarly, if $\phi'_d(i, g_1) = 1$,

$\phi_d(i, g_{2^d}) := 0$. Therefore $1 - (\phi_d(i, g_{2^d}) + \phi'_d(i, g_1)) = 0$. Thus $\sum_{m=1}^{2^{d+1}} \phi_{d+1}(0i, g_m^{d+1}) = 2^d$.

$$\begin{aligned}
& \sum_{m=1}^{2^{d+1}} \phi_{d+1}(1i, g_m^{d+1}) \\
&= \sum_{m=1}^{2^d-1} \phi_d(i, g_m) + 1 + \sum_{m=2^d}^2 \phi'_d(i, g_m) + 0 \\
&= \sum_{m=1}^{2^d} \phi_d(i, g_m) + \sum_{m=2^d}^1 \phi'_d(i, g_m) + 1 - (\phi_d(i, g_{2^d}) + \phi'_d(i, g_1)) \\
&= 2^{d-1} + 2^{d-1} + 1 - (\phi_d(i, g_{2^d}) + \phi'_d(i, g_1)) \\
&= 2^d.
\end{aligned}$$

Therefore in $SBT(g_1^{d+1}, g_2^{d+1}, \dots, g_{2^{d+1}}^{d+1})$, any node appears as leaf node 2^d times. Using the similar argument, we can show that in $SBT(g_{2^{d+1}}^{d+1}, \dots, g_2^{d+1}, g_1^{d+1})$, any node appears as leaf node 2^d times. $\square$

The following corollary gives the setup overhead complexity of the **broadcasting** algorithm.

Corollary 2 In Algorithm 1 (row partitioning, with pivoting), when $SBT(Gray(d))$ is used to broadcast the pivot rows, the total setup overhead of each processor is $\frac{1}{2}Nt_s$.

Proof: Assume $N = np$. There are totally N one-to-all broadcasts, and $SBT(Gray(d))$ is used $n = N/p$ times. By Theorem 3, in $SBT(Gray(d))$, each node appears as leaf node $p/2$ times. That is, for consecutive p broadcastings, each processor incurs setup overhead of $\frac{1}{2}pt_s$. Therefore the total setup overhead for each processor in the N broadcasts is $\frac{1}{2}pt_s \frac{N}{p} = \frac{1}{2}Nt_s$. $\square$

5 Analysis of Algorithm I

In this section we analyze the performance of the Algorithm I given in Section 3. The objective is to show the following:

1. A sufficient condition for full overlap of computation and communication.
2. The message queue length is bounded by a small number when full overlap is achieved.
3. Numerical results on the communication overhead of Algorithm I for **partial** or full overlap.

First consider the computation load of each processor at the Cth loop. The updating of each element **not** on the Cth column involves one multiplication and one subtraction, and the updating of each element on the Cth column involves one division. For simplicity, we assume the time to update each element of A is f . For processor $P_{[k]}$, since it has already normalized the Cth pivot row and found the pivot element in the previous $(C - 1)$ st loop, it needs to update the rest $(n - 1)$ rows it has using the Cth pivot row and pivot element. We call this computational task Π_1 , and the corresponding computation time is T_1 , where $T_1 = (n - 1)Nf$. For processor $P_{[k+1]}$, its computational task in the Cth loop can be split into two parts, Π_2 and Π_3 . Π_2 corresponds to updating the $C + 1$ st row of A using the Cth pivot row, normalizing this row **and** finding the pivot element of the $k + 1$ st pivot row. The time for updating is Nf . We **assume** it takes another Nf to find the pivot element and normalize this row. Therefore the time for Π_2 is $T_2 = 2Nf$. After completing Π_2 processor $P_{[k+1]}$ immediately broadcasts the $(k + 1)$ st pivot row. Then it resumes to update the other $(n - 1)$ rows using the Cth pivot row, we call this task Π_3 , and the corresponding time $T_3 = T_1 = (n - 1)Nf$. For all the other processors, the computational task during the Cth loop is Π_4 , which corresponds to updating all the n rows using the k th pivoting row. The time for Π_4 is $T_4 = nNf$.

Therefore the computational load of processor P at the Cth loop of Algorithm I is given by

$$T^k(P) = \begin{cases} T_1 = (n - 1)Nf, & P = P_{[k]}, \\ T_2 + T_3 = (n + 1)Nf, & P = P_{[k+1]}, \\ T_4 = nNf, & \text{otherwise.} \end{cases}$$

Note there is computational load imbalance among the processors in the k th loop. $P_{[k+1]}$ has the largest load and $P_{[k]}$ has the smallest load. By using the wrap-mapping to distribute the rows of A **among** processors, every processor becomes $P_{[k]}$ (and $P_{[k+1]}$) alternatively and thus the **computation** loads are balanced over the whole process of GJ inversion.

In **order** to explain and analyze our parallel GJ inversion algorithm, we introduce a task scheduling **diagram**, which is instrumental in our analysis of the performance of the proposed algorithm and allows us to conveniently formalize the notion of overlap **communication** and computation. A task scheduling diagram is a directed graph $G(V, E)$ consisting of a node set V and an arc set E .

Figure 3: Task scheduling diagram for Algorithm I ($p = 4$ and $N = 8$).

A node in G represents one of the four computational tasks Π_1, Π_2, Π_3 and Π_4 . An arc in G represents precedence relations that exist between two computational tasks, subject to the condition that **certain** amount of time is allocated to each arc to account for **the** actual **communication** time between the two computational tasks represented by the two leaf nodes of **the** arc.

An example of a task scheduling diagram is given in Figure 3. It represents the parallel GJ inversion process of a 8×8 matrix using 4 processors. In general the task scheduling diagram G has the following properties:

Property 1: The nodes of G are ordered horizontally into p columns, each **column** l corresponds to the **updating** operation sequence of processor P_l during the parallel GJ inversion.

Property 2: Vertically G is constructed in N stages, each stage k corresponds to the concurrent updating **operations** of all the processors at the k th loop of the parallel GJ inversion.

Property 3: In the k th stage of G , the node in column $[k]$ is a T_1 node; in column $[k + 1]$ is a T_2 node followed **by** a T_3 node; and the nodes in all other **columns** are T_4 nodes.

Property 4: All nodes at the first stage of G has in-degree 1. At stages other than the first, a T_1 or T_3 node has in-degree 1, and a T_2 or T_4 node has in-degree 2.

Property 5: A T_2 node has out-degree p and the each of the other three type of nodes has out-degree 1, except in the last stage.

As can be seen in Figure 3, there are two kinds of arcs in C : "vertical" arcs and "non-vertical" arcs. They represent two kinds of precedence constraints and communication delays. A vertical arc is an arc connecting two adjacent nodes in the same column in C . The precedence constraint it represents is that the computation in the tail node must precede that of the head node, i.e., a processor can start the $(k+1)$ st loop only after it completes the k th loop. The communication delay introduced by a vertical arc is the communication startup delay. As discussed earlier, a processor may be interrupted by an incoming message and it may need to forward the message to other processors. This may introduce communication startup overhead and this overhead is represented by the vertical arc. A non-vertical arc is an arc from a T_2 node to a T_2 node or to a T_4 node at the next stage in G . The precedence constraint it represent is that the computation in the head node can not start before it receives the message from the tail node, i.e., a processor can not start the $(k+1)$ st loop before it gets the $(k+1)$ st pivot row from $P_{[k+1]}$. The communication delay introduced by a non-vertical arc is the time between the message is sent out and the message reaches the processor.

Next we give an analytical model for the performance of **Algorithm I**. First some notations are listed as following.

- $T_{start}^k(P)$: The time processor P starts the k th loop.
- $T_{complete}^k(P)$: The time processor P completes the k th loop.
- $T^k(P)$: The computation load at the k th loop of processor P .
- T_{send}^k : The time the k th pivot row is sent out.
- $T_{arrive}^k(P)$: The time the k th pivot row arrives at P .
- $s^k(P)$: The setup overhead of P at the k th stage.
- $t^k(P)$: The time between the k th message is sent out by $P_{[k]}$ and the time it arrives at processor P , assuming no delay at each intermediate processor.

At stage k in G , a T_2 or a T_4 node has two incoming arcs, the delay introduced by the vertical arc is $s^k(P)$ and the delay introduced by the non-vertical arc is $t^k(P)$. From Section 4 we know that $s^k(P)$ is either 0 or t_s , depending on whether or not P is a leaf node in the k th broadcasting tree. Let $H(P_{[k]}, P)$ be the Hamming distance between the physical addresses of $P_{[k]}$ and P . Then the message will take $H(P_{[k]}, P)$ hops to reach P from $P_{[k]}$. Therefore $t^k(P) = H(P_{[k]}, P)(t_s + t_w N)$. A T_1 node has one vertical incoming arc, which introduces no delay, i.e., $s^k(P) = 0$. A T_3 node has one vertical incoming arc, which introduces delay $s^k(P) = t_s$.

Figure 4 illustrates the relationship among the parameters defined above, which can be formally stated by the following lemma.

Lemma 4 We have the following *recurrent* relationships:

Figure 4: Illustration of the relationship among the parameters defined above.

$$T_{start}^k(P) = \begin{cases} \max\{T_{complete}^{k-1}(P), T_{arrive}^k(P)\} + s^k(P), & P \neq P_{[k]}, \\ T_{complete}^{k-1}(P) + s^k(P), & P = P_{[k]}. \end{cases}$$

$$T_{complete}^k(P) = T_{start}^k(P) + T^k(P).$$

$$T_{send}^k = T_{start}^{k-1}(P_{[k]}) + T_2.$$

Proof: These relationships are obvious from the definition of $T_{start}^k(P)$, $T_{complete}^k(P)$ and T_{send}^k , and the task scheduling graph, as illustrated in Figure 4. $\square$

The next theorem states that a message can not be delayed at an intermediate processor during the broadcast process in Algorithm I.

Theorem.4 In Algorithm I, when the SBT_i broadcast algorithm is used, then

$$T_{arrive}^k(P) = T_{send}^k + t^k(P).$$

Proof: We need to show that whenever a message arrives at an intermediate processor, the processor can forward this message immediately without delay. A delay is incurred if when the message arrives at the processor, the processor is setting up the communication channel for a previous message. Then only after the processor finishes setting up the **communication** channel for the **previous** message can it start forward this message. In the worst case the **message** can be delayed for up to time t_s .

We have the following two simple observations:

(1) If $T_{arrive}^k(P) - T_{arrive}^{k-1}(P) > t_s$, then message k will not be delayed at processor P , because when **message** k arrives at P , P has already finished setting up **communication** channel for message $k - 1$, and therefore can forward message k immediately.

(2) If P does not forward message $k - 1$, i.e., P is a leaf node of the $(k - 1)$ st SBT, then as long as $T_{arrive}^k(P) - T_{arrive}^{k-1}(P) > 0$, message k will not be delayed at processor P , since P can immediately forward message k .

Assume $H(P_{[k-1]}, P) = m$, since $H(P_{[k]}, P_{[k-1]}) = 1$, we have either $H(P_{[k]}, P) = m + 1$ or $H(P_{[k]}, P) = m - 1$. Suppose $P_{[k]}$ and $P_{[k-1]}$ differ by the j th bit, if P and $P_{[k-1]}$ have the same j th bit, then P and $P_{[k]}$ have different j th bits, and $H(P_{[k]}, P) = m + 1$; otherwise P and $P_{[k-1]}$ have different j th bits, then P and $P_{[k]}$ have the same j th bit, and $H(P_{[k]}, P) = m - 1$.

Notice that

$$\begin{aligned} & T_{send}^k - T_{send}^{k-1} \\ &= [T_{start}^{k-1}(P_{[k]}) + T_2] - T_{send}^{k-1} \\ &\geq (T_{send}^{k-1} + t_s + t_w N + T_2) - T_{send}^{k-1} \\ &= (t_s + t_w N) + T_2. \end{aligned}$$

The proof is by induction on k . For $k = 1$ obviously the message will not be delayed at any processor since there is no other previous message. Assume message $(k - 1)$ is not delayed at any processor P , i.e.,

$$T_{arrive}^{k-1}(P) = T_{send}^{k-1} + t^{k-1}(P) = T_{send}^{k-1} + m(t_s + t_w N).$$

We need to show message k also can not be delayed. We consider two cases:

(1) $H(P_{[k]}, P) = m + 1$. Then $T_{arrive}^k(P) - T_{arrive}^{k-1}(P) \geq T_{send}^k + (m + 1)(t_s + t_w N) - T_{arrive}^{k-1}(P) = 2(t_s + t_w N) + T_2 > t_s$. Therefore message k will not be delayed at processor P .

(2) $H(P_{[k]}, P) = m - 1$. Then $T_{arrive}^k(P) - T_{arrive}^{k-1}(P) \geq T_{send}^k + (m - 1)(t_s + t_w N) - T_{arrive}^{k-1}(P) = T_2 > 0$. Since P and $P_{[k-1]}$ have different j th bits, P is a leaf node in the $(k - 1)$ st SBT, therefore P does not forward message $(k - 1)$. So message k will not be delayed at processor P . $\square$

Let $T_{comp}(P)$ be the total computation load of processor P , and $T_{comm}(P)$ be the total communication overhead of processor P , then

$$\begin{aligned} T_{comp}(P) &= \sum_{k=1}^N T^k(P), \\ T_{comm}(P) &= T_{complete}^N(P) - T_{comp}(P) \\ &= T_{complete}^N(P) - \sum_{k=1}^N T^k(P). \end{aligned}$$

It can be seen in Figure 4 that at the $(k-1)$ st iteration, if $T_{arrive}^k(P) > T_{complete}^{k-1}(P)$, then processor P will be idle for a time $t_{idle}^{k-1}(P) = T_{arrive}^k(P) - T_{complete}^{k-1}(P)$; otherwise $t_{idle}^{k-1}(P) = 0$. If $t_{idle}^k(P) = 0$ for all $2 \leq k \leq N$, then full overlap is achieved at P . Now we can give a formal definition of full overlap of communication and computation.

Definition 5 We say that the communication is fully overlapped by computation for processor P in Algorithm I if

$$T_{complete}^{k-1}(P) \geq T_{send}^k + t^k(P), \quad 2 \leq k \leq N,$$

or equivalently

$$T_{start}^k(P) = T_{complete}^{k-1}(P) + s^k(P), \quad 2 \leq k \leq N.$$

That is, the k th pivot row has already arrived at processor P when P completes computation of the $k-1$ st stage of the parallel GJ inversion algorithm. $\square$

At the beginning of the algorithm, i.e., $k=1$, all the processors except P_1 are idle until the first pivot row arrives, as illustrated in Figure 3. Let $d_0(P)$ be the initial idle time of processor P . Since P_1 first normalizes the first row of A and finds the pivot element, which takes time Nf , before it sends it out, then $d_0(P) = H(P_1, P)(t_s + t_w N) + Nf$.

Since our objective is to let the message arrive at all the processors before it is needed, another parameter of interest is the number of messages in the input buffer of a processor at a particular time.

Definition 6 Let $Q^k(P)$ be the message queue length of processor P at stage k , then $Q^k(P) = l$, if

$$T_{send}^{k+l} + t^{k+l}(P) \leq T_{complete}^k(P) < T_{send}^{k+l+1} + t^{k+l+1}(P),$$

That is, after processor P completes stage k and is to start stage $k+1$, messages for stage $k+1, k+2, \dots, k+l$ have already arrived at P . $\square$

The next theorem gives a sufficient condition for full overlap communication and computation in Algorithm I.

Theorem 5 When $N \geq N_0$, full overlap of communication and computation *can* be achieved, where N_0 is the positive root of the quadratic equation

$$\left(\frac{N}{p} - 3\right)Nf = \frac{1}{2}pt_s + 2(t_s + t_w N) \log p.$$

Proof: We rewrite the quadratic equation in the standard form:

$$\left(\frac{f}{p}\right)N^2 - (3f + 2t_w \log p)N - \left(\frac{1}{2}p + 2 \log p\right)t_s = 0.$$

Obviously this equation has one positive root and one negative root. Let the positive root be N_0 . When $N \geq N_0$, we have

$$\left(\frac{N}{p} - 3\right)Nf \geq \frac{1}{2}pt_s + 2(t_s + t_w N) \log p.$$

Let $\Delta T^k(P) = T_{complete}^{k-1}(P) - T_{send}^k - t^k(P)$, we will show that $\Delta T^k(P) \geq 0$ for $2 \leq k \leq N$ and $P \in \{P_1, P_2, \dots, P_p\} - P_k$. The proof is by induction.

(1) Base case: $k = 2$,

$$\begin{aligned} \Delta T^2(P) &= T_{complete}^1(P) - T_{send}^2 - t^2(P) \\ &= [T_{start}^1(P) + T^1(P)] - [T_{start}^1(P_2) + T_2] - t^2(P) \\ &= (d_0(P) + T^1(P)) - (d_0(P_2) + T_2) - t^2(P) \\ &= (T^1(P) - T_2) - (d_0(P_2) - d_0(P)) - t^2(P). \end{aligned}$$

where $T^1(P) - T_2$

$$= \begin{cases} T_1 - T_2 = (n-3)Nf, & P = P_1, \\ T_4 - T_2 = nNf - 2Nf = (n-2)Nf, & P \neq P_1. \end{cases}$$

$$d_0(P_2) - d_0(P) \leq (t_s + t_w N) \log p.$$

$$t^2(P) \leq (t_s + t_w N) \log p.$$

Therefore

$$\begin{aligned} \Delta T^2(P) &\geq (n-3)Nf - 2(t_s + t_w N) \log p - Nf \\ &= \left(\frac{N}{p} - 3\right)Nf - 2(t_s + t_w N) \log p \\ &> 0, \end{aligned}$$

when $N \geq N_0$.

(2) Induction step: Assume the proposition is true for $2 \leq l \leq k$. We prove it is also true for $l = k + 1$.

By the induction hypothesis, $T_{start}^l(P) = T_{complete}^{l-1}(P) + s^l(P)$, for $2 \leq l \leq k$. Therefore

$$\begin{aligned}
& \Delta T^{k+1}(P) \\
&= T_{complete}^k(P) - T_{send}^{k+1} - t^{k+1}(P) \\
&= [T_{start}^k(P) + T^k(P)] - [T_{start}^k(P_{[k+1]}) + T_2] - t^{k+1}(P) \\
&= \left[d_0(P) + \sum_{i=1}^k (T^i(P) + s^i(P)) \right] - \left[d_0(P_{[k+1]}) + \sum_{i=1}^k (T^i(P_{[k+1]}) + s^i(P_{[k+1]})) - T^k(P_{[k+1]}) + T_2 \right] \\
&\quad - t^{k+1}(P) \\
&= \sum_{i=1}^k [T^i(P) - T^i(P_{[k+1]})] + (T_{[k+1]}^k - T_2) - [d_0(P_{[k+1]}) - d_0(P)] \\
&\quad - \left[\sum_{i=1}^k s^i(P_{[k+1]}) - \sum_{i=1}^k s^i(P) \right] - t^{k+1}(P) \\
&> \sum_{i=1}^k [T^i(P) - T^i(P_{[k+1]})] + (T_2 + T_3 - T_2) - (t_s + t_w N) \log p - \frac{1}{2} p t_s - (t_s + t_w N) \log p \\
&= -Nf + (n-l)Nf - \frac{1}{2} p t_s - 2(t_s + t_w N) \log p \\
&= \left(\frac{N}{p} - 2 \right) Nf - \frac{1}{2} p t_s - 2(t_s + t_w N) \log p \\
&\geq 0,
\end{aligned}$$

when $N \geq N_0$. Here we used the fact that if let $r = \left\lceil \frac{k}{p} \right\rceil$, then

$$\sum_{i=1}^k T^i(P) = \begin{cases} (T_1 + T_2 + T_3)r + T_4(k - 2r), & P \in \{P_1, P_2, \dots, P_{[k]}\}, \\ (T_1 + T_2 + T_3)r - T_1 + T_4(k - 2r + 1), & P = P_{[k+1]}, \\ (T_1 + T_2 + T_3)(r - 1) + T_4(k - 2r + 2), & P \in \{P_{[k+2]}, \dots, P_p\}. \end{cases}$$

And therefore

$$\sum_{i=1}^k [T^i(P) - T^i(P_{[k+1]})] = \begin{cases} T_1 - T_4 = -Nf, & P \in \{P_1, P_2, \dots, P_{[k]}\}, \\ T_4 - (T_2 + T_3) = -Nf, & P \in \{P_{[k+2]}, \dots, P_p\}. \end{cases}$$

We also used the fact that

$$\begin{aligned}
& \sum_{i=1}^k s^i(P_{[k+1]}) - \sum_{i=1}^k s^i(P) \\
&= \left\lceil \frac{k}{p} \right\rceil \left(\frac{1}{2} p t_s - \frac{1}{2} p t_s \right) + \sum_{i=1}^{[k]} s^i(P_{[k+1]}) - \sum_{i=1}^{[k]} s^i(P) \\
&\leq \frac{1}{2} p t_s.
\end{aligned}$$

□

The next theorem states that the message queue length at each processor is bounded by 2.

Theorem 6 When $N \geq N_0$, the message queue length is at most 2, i.e., $Q^k(P) \leq 2$.

Proof: By contradiction. Otherwise there exist k and P such that $Q^k(P) = l \geq 3$. Since $N \geq N_0$, by Theorem 5, communication is fully overlapped by computation. By the definition of the queue length $Q^k(P)$, we have

$$\begin{aligned} T_{complete}^k(P) &\geq T_{send}^{k+l} + t^{k+l}(P) \\ &= T_{start}^{k+l-1}(P_{[k+l]}) + T_2 + t^{k+l}(P) \\ &= T_{complete}^{k+l-2}(P_{[k+l]}) + s^{k+l-1}(P_{[k+l]}) + T_2 + t^{k+l}(P). \end{aligned}$$

Or

$$\begin{aligned} d_0(P) + \sum_{i=1}^k [T^i(P) + s^i(P)] &\geq d_0(P_{[k+l]}) + \sum_{i=1}^{k+l-2} [T^i(P_{[k+l]}) + s^i(P_{[k+l]})] \\ &\quad + s^{k+l-1}(P_{[k+l]}) + T_2 + t^{k+l}(P) \end{aligned}$$

$$\begin{aligned} \sum_{i=1}^{k+l-2} T^i(P_{[k+l]}) - \sum_{i=1}^k T^i(P) + T_2 &\leq (d_0(P) - d_0(P_{[k+l]})) - t^{k+l}(P) - \sum_{i=1}^{k+l-1} s^i(P_{[k+l]}) + \sum_{i=1}^k s^i(P) \\ &\leq (t_s + t_w N) \log p + \frac{1}{2} t_s. \end{aligned} \quad (1)$$

But when $l \geq 3$,

$$\begin{aligned} \sum_{i=1}^{k+l-2} T^i(P_{[k+l]}) - \sum_{i=1}^k T^i(P) + T_2 &\geq T_1 + T_2 \\ &= \left(\frac{N}{p} + 1\right) N f \\ &> \left(\frac{N}{p} - 3\right) N f \\ &\geq 2(t_s + t_w N) \log p + \frac{1}{2} p t_s \\ &> (t_s + t_w N) \log p + \frac{1}{2} p t_s. \end{aligned} \quad (2)$$

(1) and (2) contradicts each other. Therefore $Q^k(P) \leq 2$. $\square$

Corollary 3 When $N \geq N_0$, the communication overhead of processor P in Algorithm 1 is $d_0(P) + \frac{N}{2} t_s$.

Proof: At the beginning of the algorithm ($k = 1$), the processors other than P_1 must wait to get the first pivot row from P_1 , since no computation can be done for these processors before getting the first pivot row. Thus a communication overhead (initial idle time) of $d_0(P)$ is incurred. After that, since $N \geq N_0$ by Theorem 5, the data transmission is fully overlapped by computation for

Figure 5: Communication overheads of Algorithm I (Number of processors $p=16$).

all the processors during the rest of the updating process (for $2 \leq k \leq N$). Therefore the total idle time for processor P in Algorithm I is $d_0(P)$. By Corollary 2, the total setup overhead of each processor in Algorithm I is $\frac{N}{2}t_s$. So the total communication overhead is $d_0(P) + \frac{N}{2}t_s$. $\square$

Next we give some simulation results on the parallel overheads of Algorithm I, obtained by numerically evaluating the recurrent relationships given in Lemma 4. The time is scaled to f , i.e., let $f = 1$. We use the machine parameter $t_c = 150$ and $t_w = 3$. These parameters are close to that of the nCUBE 2 machine [28]. Since the communication overheads may be different for different processors, the data depicted in the figure are the largest overheads among the processors. Figure 5 depicts the communication overhead T_{comm} vs the matrix size N when the machine size $p = 16$. Curve 1 is the total communication overheads; curve 2 is the total communication overheads minus the initial delays; for the case of curve 3, we let each processor initially holds the first row of A , such that all processors can start without the initial delay. It can be seen that when there is initial delay, when $N > 260$ the communication is “almost” overlapped by computation, but not completely until $N > 460$. When there is no initial delay, the communication is completely overlapped by computation when $N > 260$.

Figure 6 shows the communication overheads with different machine sizes. The initial delays are subtracted so that it is easy to see where full overlap is achieved. We can see that when full overlap is achieved, the communication overhead is independent of the machine size (if the initial delay is not considered), and is linearly increasing with the problem size (i.e., $T_{comm} = \frac{1}{2}Nt_s$).

Figure 6: Communication overheads of Algorithm I with different machine sizes.

6 Algorithm II: Submatrix Partitioning, Without Pivoting

In this section, we apply the idea of overlap communication and computation to the parallel GJ matrix inversion algorithm using submatrix partitioning. The main advantages of the submatrix **partitioning** are shorter message length, i.e., the message is of length $\frac{N}{\sqrt{p}}$ instead of N ; and shorter **communication** propagation delay, i.e., each broadcasting tree is of height $\frac{d}{2}$ instead of d . However, the disadvantage is that the searching of pivot element can not be done by a single processor. Instead $\sqrt{p}$ processors individually find their local pivot element and compare with each other to find the final pivot element through recursive doubling. As a result, at every iteration there is a **synchronization barrier** among all the processors which makes it difficult to overlap communication and computation. In this section, we study the algorithm without pivoting.

Let $p = 2^d$ be the number of processors in a d -cube. These processors are configured as a two dimensional array of processors. Let p_r and p_c be the number of processor rows and columns respectively, where $p_r = p_c = \sqrt{p} = 2^{d/2}$ when d is even, and $p_r = \sqrt{2p} = 2^{(d+1)/2}$, $p_c = \sqrt{p/2} = 2^{(d-1)/2}$ when d is odd. The physical address of each processor is determined by the 2-d Gray code mapping. Let $d_r = \log p_r$, $d_c = \log p_c$, and $d = \max(d_r, d_c)$. The d -bit Gray code is $Gray(d) = (g_1^d, g_2^d, \dots, g_d^d)$. We use notation P_{ij} to denote the processor with coordinates (i, j) in the processor array, where $1 \leq i \leq p_r$ and $1 \leq j \leq p_c$. Then the physical address of P_{ij} is $g_i^d g_j^d$. It is easy to see that under this mapping strategy each column and each row processors form a subcube and the adjacent processors on the grid are directly connected.

The matrix elements are distributed among the processors by full column and row wrap-mapping, i.e., matrix element a_{ij} is placed in processor row I and column J according to $I = (i - 1) \bmod p_r + 1$ and $J = (j - 1) \bmod p_c + 1$. Thus $1 \leq I \leq p_r$ and $1 \leq J \leq p_c$. This mapping strategy improves the computation load balance when interchanges are performed [1]. In our implementation, the objective is to achieve the overlap of communication and computation. This full column and row wrap-mapping strategy will balance both the computation and communication loads.

Under the submatrix partitioning strategy, there are two types of communication required at every iteration: one to pass the pivot row and another to pass the multipliers. The multipliers are the matrix column that is to be set to zero in that step. These data are available in one column of processors and the multipliers must be passed to the remaining processor columns. Therefore the communication path for passing multipliers is across processor columns.

The pivot row is contained initially in a single row of processors. The communication path for passing pivot row is across processors rows. Before sent out, each element of the pivot row need to be normalized by the pivot element. The pivot element is contained in the multipliers passed to the processor. Therefore the normalization and the broadcast of the pivot row can not start until the processors get the multipliers.

We still use the *compute-and-send-ahead* strategy to achieve the overlap of communication and computation, i.e., at the k -th iteration, the data elements of the $k+1$ st pivot row and multipliers are first updated and then sent out. Unlike in Algorithm I, where there is only one broadcast within

the hypercube at every iteration, in Algorithm II, there are concurrent **broadcasts** within **subcubes** at each iteration: first there are p_r concurrent broadcasts of segments of multipliers within the **subcube rows**; then there are p_c concurrent broadcasts of segments of the pivot row within the **subcube columns**. For simplicity, we assume $p_r = p_c = \sqrt{p}$. We use the notation $[k]$ to represent $(k-1) \bmod p_r + 1$. Then at the k th iteration, processors in the $[k+1]$ -st column and the $[k+1]$ -st row are roots of the broadcast trees. As an example, Figure 7 shows the data flow for Algorithm II where $p = 16$ and $[k+1] = 2$.

Figure 7: Data flow of Algorithm II.

Let (I, J) be the coordinates for processor P in the processor array, where $1 \leq I \leq p_r$ and $1 \leq J \leq p_c$. The pseudo code for the k th step of the Algorithm II is as following:

if $(I \neq [k+1])$ and $J \neq [k+1]$ then

update all the elements it has using the corresponding **segments** of the k th pivot row and multipliers;

 Interrupt handling:

 If the message is from a processor in the same processor row,
 pass it to its children nodes in the row **subcube**, if there is any;

 If the message is from a processor in the same processor column,
 pass it to its children nodes in the column **subcube**, if there is any ;

if $(I = [k+1])$ and $J = [k+1]$ then

update the corresponding segment of the $k+1$ st column of A

broadcast this column segment in the row **subcube**;

update the corresponding segment of the $k + 1$ st row of A;
and normalize it using the $k + 1$ st pivot element;
broadcast this row segment and the pivot element in the column **subcube**;
 update the rest of the elements of A;

if $(I \neq [k + 1])$ and $J = [k + 1]$ then

update the corresponding segment of the $k + 1$ st column of A;
 broadcast this column segment in the row **subcube**;
update the rest of the elements of A.

Interrupt handling:

If the message is from a processor in the same processor row,
 pass it to its children nodes in the row **subcube**, if there is any;

if $(I = [k + 1])$ and $J \neq [k + 1]$ then

update the corresponding segment of the $k + 1$ st row of A;
 update the rest of the elements of A;

Interrupt handling:

If the message is from a processor in the same processor **column**
 pass it to its children nodes in the column **subcube**, if there is any;
 normalize the corresponding segment of the $k + 1$ st row A
 using the $k + 1$ st pivot element;
 broadcast this row segment and the pivot element in the row **subcube**;

In the above algorithm, whenever a message arrives at the processor, if the processor is doing **computation**, it will be interrupted and start executing the communication interrupt handling. If on the other hand, the processor has finished all the **computation** at a particular step and the message **has** not yet arrived, it will become idle until the message arrives, and then start executing the interrupt handling code before it goes to the next **step** of the loop.

The algorithm for each broadcast within a subcube-row or **subcube-column** is essentially the same as the broadcasting algorithm proposed in Section 4. Consider the i -th processor row **con-**sisting processors with physical addresses $g_i^d g_1^d, g_i^d g_2^d, \dots, g_i^d g_{2^d}^d$. From Section 4 we know now to construct a family of SBT's:

$$SBT(Gray(d)) = (SBT^{(1)}, SBT^{(2)}, \dots, SBT^{(2^d)}),$$

where in $SBT^{(k)}$, g_k^d is the root node and g_{k+1}^d is a leaf node at the first level.

A family of SBT's in the i -th processor row can be obtained by making the **following** transformation on each node of each SBT in $SBT(Gray(d))$: if the physical address of a node is g_j , then replace it **with** $g_i^d g_j$, $1 \leq j \leq 2^d$. Similarly a family of SBT's in the j -th processor column can

be obtained by the following transformation on each node of each SBT in $SBT(Gray(d))$: if the physical address of a node is g_i , then replace it with $g_i g_j^d$, $1 \leq i \leq 2^d$.

7 Analysis of Algorithm II

In this section we give the performance analysis of Algorithm II. For **simplicity** we only consider the case the hypercube dimension d is even, i.e., $p_c = p, = \sqrt{p} = 2^{\frac{d}{2}}$. Let P_{ij} be the processor in the i th processor row and j th processor column, $1 \leq i \leq \sqrt{p}, 1 \leq j \leq \sqrt{p}$. The following notations are used in the analysis:

$T_{start}^k(P_{ij})$:	The time P_{ij} starts the k th loop.
$T_{complete}^k(P_{ij})$:	The time P_{ij} completes the k th loop.
$T^k(P_{ij})$:	The computation load of P_{ij} at the k th loop.
T' :	The time to update or normalize one segment of the multipliers or the pivot row elements.
$T_{send-r}^k(P_{[k]j})$:	The time that the segment of k th row of A is sent by $P_{[k]j}$.
$t_r^k(P_{ij})$:	The time it takes for this segment of the k th row to reach P_{ij} .
$T_{send-c}^k(P_{i[k]})$:	The time that the segment of the k th column of A is sent by $P_{i[k]}$.
$t_c^k(P_{ij})$:	The time it takes for this segment of the k th column to reach P_{ij} .
$s_r^k(P_{ij})$:	The setup overhead of P_{ij} when broadcasting the k th row segment.
$s_c^k(P_{ij})$:	The setup overhead of P_{ij} when broadcasting the k th column segment.

The following theorem shows that the total setup overhead of Algorithm II is twice that of Algorithm I.

Theorem 7 *In Algorithm II, the total setup overhead of each processor is Nt_s .*

Proof: In Algorithm II, each processor involves N one-to-all broadcasts in its **subcube** column and N **one-to-all** broadcasts in its **subcube** row. By the construction of the **SBT**'s, for consecutive $\sqrt{p}$ broadcastings, each processor incurs setup overhead of $\frac{1}{2}\sqrt{p}t_s$. Therefore the total setup overhead for each processor is $2 \cdot \frac{1}{2}\sqrt{p}t_s \frac{N}{\sqrt{p}} = Nt_s$. $\square$

Each processor involves two broadcastings at every iteration. The situation, might occur that when a message containing the segment of the pivot row arrives at an intermediate processor, this processor is in the process of setting up the communication channel for a **message** containing the segment of the multipliers. Therefore the message can not be immediately pass over. Instead, it will be delayed until the processor finishes setting up the channel for the previous message. This situation can not happen in Algorithm I, as proved in Theorem 4. In the worst case, a message is delayed for a time of t , at an intermediate processor. As a result,, we have

$$H(P_{[k]j}, P_{ij})(t_s + t_w \frac{N}{\sqrt{p}}) \leq t_r^k(P_{ij}), t_c^k(P_{ij}) \leq H(P_{[k]j}, P_{ij})(2t_s + t_w \frac{N}{\sqrt{p}})$$

Lemma 5 *We have the following recurrent relationship:*

$$T_{start}^k(P_{ij}) = \begin{cases} \max\{T_{complete}^{k-1}(P_{ij}) + s_c^k(P_{ij}) + s_r^k(P_{ij}), T_{send-c}^k(P_{i[k]}) \\ + t_c^k(P_{ij}) + s_c^k(P_{ij}), T_{send-r}^k(P_{[k]j}) + t_r^k(P_{ij}) + s_r^k(P_{ij})\}, & i \neq [k], j \neq [k], \\ \max\{T_{complete}^{k-1}(P_{[k]j}) + s_c^k(P_{[k]j}) + s_r^k(P_{[k]j}), T_{send-c}^k(P_{[k][k]}) \\ + t_c^k(P_{[k]j}) + s_c^k(P_{[k]j})\}, & i = [k], j \neq [k], \\ \max\{T_{complete}^{k-1}(P_{i[k]}) + s_c^k(P_{i[k]}) + s_r^k(P_{i[k]}), T_{send-r}^k(P_{[k][k]}) \\ + t_r^k(P_{i[k]}) + s_r^k(P_{i[k]})\}, & i \neq [k], j = [k], \\ T_{complete}^{k-1}(P_{[k][k]}) + s_c^k(P_{[k][k]}) + s_r^k(P_{[k][k]}), & i = [k], j = [k]. \end{cases}$$

$$T_{complete}^k(P_{ij}) = T_{start}^k(P_{ij}) + T^k(P_{ij}).$$

$$T_{send-c}^k(P_{i[k]}) = T_{start}^{k-1}(P_{i[k]}) + T'$$

$$T_{send-r}^k(P_{[k]j}) = \max\{T_{start}^{k-1}(P_{[k]j}) + T', T_{send-c}^k(P_{[k][k]}) + t_c^k(P_{[k]j}) + s_c^k(P_{[k]j})\} + T'.$$

□

We say that communication is fully overlapped by computation if

$$T_{start}^k(P_{ij}) = T_{complete}^{k-1}(P_{ij}) + s_c^k(P_{ij}) + s_r^k(P_{ij}), \quad 2 \leq k \leq N, \quad \forall i, j.$$

Theorem 8 *When $N \geq N_0$, the communication can be fully overlapped by computation, where N_0 is the positive root of the following quadratic equation:*

$$\left(\frac{N^2}{p} - \frac{3N}{\sqrt{p}}\right) f = 2\sqrt{p}t_s + 2(2t_s + t_w \frac{N}{\sqrt{p}}) \log p.$$

□

Proof: See Appendix.

At the beginning of Algorithm II, each processor except P_{11} must be initially idle until it got the multipliers and/or the first pivot row elements. Let $d_0(P_{ij})$ be the initial delay of P_{ij} .

Corollary 4 *When $N \geq N_0$, the communication overhead of processor P_{ij} in Algorithm II is $d_0(P_{ij}) + Nt_s$.*

Next we give the simulation results on the parallel overhead of Algorithm II, obtained by numerically evaluating the recurrent relationships give in Lemma 5. The parameters (f , t , and t) are the same as in Section 5. Figure 8 depicts the communication overhead T_{comm} vs the matrix size

N when the machine size $p = 16$. As a comparison, we put the corresponding curve of Algorithm I (curve I. in Figure 5) in the same figure. We can see that when the matrix size N is small, Algorithm II incurs less overhead than Algorithm I; but when N becomes large, Algorithm I incurs much less overhead than Algorithm II. We can also see that in Algorithm II the: matrix size N_0 for full overlap is smaller than that in Algorithm I. Figure 9 shows the **communication** overheads with different **machine** sizes. We can see that when full overlap is achieved, the **communication** overhead is independent of the machine size (if we neglect the initial delay), and linearly increases with the problem size (i.e., $T_{comm} = Nt$).

Figure 8: Communication overheads of Algorithm II (Number of processor $p=16$).

Figure 9: Communication overheads of Algorithm II with different **machine** sizes.

8 Algorithm III: Submatrix Partitioning, With Pivoting

We consider the parallel GJ inversion algorithm with column interchanges. Since the matrix is partitioned by submatrix, the searching of pivot element can not be done by a single processor. Instead, a row of processors each determine the local pivot element and then find the pivot element through recursive doubling. During the E -th iteration, the $(E + 1)$ st pivot row elements can be computed ahead and sent ahead by a row of processors; but the multipliers can not be computed before the processor gets the segment of the pivot row and finds out the local pivot element. In Algorithm III, we let each processor updates its share of the submatrix column by column. As soon as the message containing the pivot row elements arrives, it will be interrupted and find the local pivot element. Then it will first update the column segment corresponding to this local pivot element and then participate permutation among the processors in the same subcube-row. After $\frac{d}{2}$ write-read pairs, each processor will get the pivot element and the corresponding segments of multiplier. Then it will normalize the pivot row elements using the pivot element.

Therefore at each iteration of Algorithm III, each processor will be interrupted $\frac{d}{2} + 1$ times: one for the broadcasting of the pivot row elements; $\frac{d}{2}$ for the permutations. We have the following theorem regarding the total setup overhead of Algorithm III.

Theorem 9 In Algorithm III, the setup overhead for each processor is $\frac{1}{2}(1 + \log p)Nt_s$.

Proof: The one-to-all broadcastings of the pivot row elements within the subcube-columns causes setup overhead of $\frac{1}{2}Nt_s$. At each iteration, the permutation within the subcube-rows to find the pivot elements causes each processor a setup overhead of $\frac{1}{2} \log p t_s$, and there are totally N iterations. So the overall setup overhead for each processor is $\frac{1}{2}(1 + \log p)Nt_s$. $\square$

Therefore Algorithm III incurs much larger setup overhead than Algorithm I and Algorithm II. Further more, Since the recursive doubling essentially causes synchronization barrier for all the processors, although we can use the interrupt handling capability to let the processor do some computation while waiting for the message, it is difficult to achieve full overlap.

if $(I = [E + 1])$ then

 update the corresponding segment of the $(E + 1)$ st row of A

 broadcast this row segment in the subcube-column;

 update the rest of the submatrix of A column by column;

if $(I \neq [E + 1])$ then

 update the share of the submatrix of A column by column;

Interrupt handling:

 if the message is from a processor in the subcube column, then

 forward this message if needed;

 find the local pivot element;

 update the segment column corresponding to the local pivot;

write this updated segment to its first neighbor in the **subcube-row**;

if the message is from the j th neighbor in the **subcube-row**, then

if $j < \frac{d}{2}$ then find the larger pivot element and send the corresponding segment to its $j+1$ -th neighbor;

if $j = \frac{d}{2}$ then find the larger pivot element and the corresponding segment is the **multiplier**.

9 Scalability Analysis

The scalability of a parallel algorithm on a parallel architecture is a measure of its capability to effectively utilize the processors. Several metrics have been proposed in the literature to measure the scalability of algorithms and machine systems [28, 29, 30]. In this section we adopt two metrics, namely, the *isoefficiency* metric [28] and the [30] metric to study the **scalability** of the parallel Gauss – Jordan matrix inversion algorithm proposed in this report.

9.1 The Isoefficiency Metric of Scalability

Let p be the number of processors, W be the problem size which is the total amount of computation to be done, and T_o be the total parallel overhead of all the processors. Then the execution time on p processors T_p satisfies $W + T_o = pT_p$. Speedup is defined as $S = \frac{W}{T_p}$. Efficiency is defined as $E = \frac{S}{p} = \frac{W}{pT_p} = \frac{W}{W + T_o} = \frac{1}{1 + \frac{T_o}{W}}$.

Gupta and Kumar defined an isoefficiency function $f_E(p)$ to measure the scalability of parallel algorithm and architecture combinations [28, 29], where $f_E(p)$ is the amount of work needed to maintain a constant efficiency E when p processors are used.

Theorem. 10 *For the parallel GJ inversion algorithm on a hypercube architecture, when the matrix is partitioned as submatrices and there is no overlap between communication and computation (i.e., Algorithm 0), then the isoefficiency function $f_E(p) = \Theta(p^{\frac{3}{2}}(\log p)^3)$.*

Proof: When submatrix partitioning is used and if there is no overlapping between communication and computation, the communication overhead for each processor is $N(t_s + t_w \frac{N}{\sqrt{p}}) \log p$. Therefore the total communication overhead is $T_o = pN(t_s + t_w \frac{N}{\sqrt{p}}) \log p$. The work load of the GJ inversion algorithm is $W = N^3 f$. To keep the efficiency E fixed, we have $f_E(p) = W = KT_o$, where $K = \frac{E}{1-E}$. Therefore $N^3 f = KpN(t_s + t_w \frac{N}{\sqrt{p}}) \log p$, or $fN^2 - Kt_w \sqrt{p} \log p N - Kt_s p \log p = 0$. Solving for N we get $N = \Theta(\sqrt{p} \log p)$. Thus $f_E(p) = W = N^3 f = \Theta(p^{\frac{3}{2}}(\log p)^3)$. $\square$

Theorem 11 *For the parallel GJ inversion algorithm on a hypercube architecture, when the matrix is partitioned by rows and when communication is fully overlapped by computation (i.e., Algorithm I), then the isoefficiency function $f_E(p) = \Theta(p^{\frac{3}{2}})$.*

Proof: When the matrix is partitioned by rows and when communication is fully overlapped by computation, the overhead for each processor is $\frac{1}{2}Nt_s$. Therefore the total overhead is $T_o = \frac{1}{2}pNt_s$. To keep the efficiency E fixed, we have $f_E(p) = W = KT_o$, where $K = \frac{E}{1-E}$. Therefore $N^3 f = \frac{1}{2}KpNt_s$. Solving for N we get $N = \sqrt{\frac{Kt_s}{2f}p} = \Theta(\sqrt{p})$. Thus $f_E(p) = W = N^3 f = \Theta(p^{\frac{3}{2}})$. $\square$

9.2 The Isospeed Metric of Scalability

In [30] the scalability of a parallel algorithm–machine combination is defined based on the *isospeed* metric. The average speed is the achieved speed of the given computing system divided by p ,

the number of processors. Let W be the amount of work of an algorithm when p processors are employed, and W' be the amount of work of the algorithm when $p' > p$ processors are employed to maintain the average speed, then the *scalability from system size p to system size p'* of the algorithm — machine combination is defined as: $\phi(p, p') = \frac{W}{W'}$. The work W' is determined by the isospeed constraint.

Theorem 12 *For Algorithm I, when the problem size is such that communication is fully overlapped by computation, then the scalability from system size p to system size p' ($p' > p$) is $\phi(p, p') = \sqrt{\frac{p}{p'}}$.*

Proof: By the isospeed constraint, we have $\frac{W}{pT_p} = \frac{W'}{p'T_{p'}}$, or $\frac{W}{W+T_o} = \frac{W'}{W'+T_o'}$, or $\frac{1}{1+\frac{T_o}{W}} = \frac{1}{1+\frac{T_o'}{W'}}$. Therefore $\frac{T_o}{W} = \frac{T_o'}{W'}$, or $\frac{\frac{1}{2}pNt_s}{\frac{1}{2}N^3f} = \frac{\frac{1}{2}p'N't_s}{\frac{1}{2}N'^3f}$. Thus $\frac{N'}{N} = \sqrt{\frac{p'}{p}}$. Therefore $\phi(p, p') = \frac{W}{W'} = \frac{p'}{p} \left(\frac{N}{N'}\right)^3 = \frac{p'}{p} \left(\frac{p}{p'}\right)^{\frac{3}{2}} = \sqrt{\frac{p}{p'}}$. $\square$

10 LU Factorization

In this **section**, we apply the ideas of overlapping communication and **computation**, to the parallel LU **factorization** algorithm, using the row partitioning strategy and column interchanges for partial pivoting.

The **pseudocode** for the sequential row oriented LU decomposition algorithm is given below:

```

for  $k = 1$  to  $N - 1$ 
  for  $i = k + 1$  to  $N$ 
     $l_{ik} = a_{ik}/a_{kk}$ 
    for  $j = k + 1$  to  $N$ 
       $a_{ij} = a_{ij} - l_{ik}a_{kj}$ 

```

The **j** loop subtracts multipliers of the **kth** row of the **current** A from succeeding rows. At the **kth** stage of the algorithm, there are $(N - k)$ row **updatings**, and the **length** of each row to be updated is $(N - k)$. (Recall in the GJ inversion algorithm, at **the kth stage**, there are N row **updatings**.,and each row is of length N .)

When implemented in parallel, to keep the computational load balanced among the processors, we use a variation of the row wrapped storage — reflection wrapped storage. Here the rows are distributed as illustrated in Figure 10 for the case of $p = 4$ and $N = 16$. In general, the first p rows are distributed to the p processors in order, the next p rows are distributed in reverse order, and so on. We use the notation $P_{\{k\}}$ to denote the processor that has the k th row.

Figure 10: Reflection wrapped row storage.

The parallel LU factorization algorithm using row partitioning is similar to the Algorithm I in Section 3. During the k -th iteration, each processor first **gets** the k -th pivot row of A and the pivot element **from** processor $P_{\{k\}}$, and then does the row **updatings**. To overlap communication and **computation**, processor $P_{\{k+1\}}$ first updates the $(k+1)$ -st row of A **during** the k -th iteration. Then it finds **the** pivot element and normalizes this row (*compute-ahead*). **After** that it broadcasts this $(k+1)$ -st **pivot** row and the pivot element to all the other processors, before it resumes to update the other rows using the k -th pivot row (send-ahead).

Consider the computation load of each processor at the **kth** stage. Let $q = \lfloor \frac{k}{p} \rfloor$, then the **computation** load of processor **P** at the k th stage is

$$T^k(P) = \begin{cases} (n - q - 1)(N - k)f, & P \in \{P_{\{pq+1\}}, P_{\{pq+2\}}, \dots, P_{\{k\}}\}, \\ (n - q + 1)(N - k)f, & P = P_{\{k+1\}}, \\ (n - q)(N - k)f, & P \in \{P_{\{k+2\}}, \dots, P_{\{p(q+1)\}}\}. \end{cases}$$

Lemma 6 For any $1 \leq K \leq N$, and for any two processors P and P' ,

$$\left| \sum_{k=1}^K T^k(P) - \sum_{k=1}^K T^k(P') \right| < 3pNf$$

Proof: Without loss generality we assume $P = P_{\{i\}}$ and $P' = P_{\{j\}}$, where $1 \leq i < j \leq p$. Consider $\sum_{k=1}^{2p} T^k(P)$.

If $i \neq 1$ and $i \neq p$, then

$$\begin{aligned} \sum_{k=1}^p T^k(P) &= \sum_{k=1}^{i-2} T^k(P) + T^{i-1}(P) + \sum_{k=i}^p T^k(P) \\ &= \sum_{k=1}^{i-2} n(N-k)f + (n+1)[N-(i-1)]f + \sum_{k=i}^p (n-1)(N-k)f \\ &= \sum_{k=1}^p (n-1)(N-k)f + [N-(i-1)]f + \sum_{k=1}^{i-1} (N-k)f. \end{aligned}$$

$$\begin{aligned} \sum_{k=p+1}^{2p} T^k(P) &= \sum_{k=p+1}^{p+(p-i)-1} T^k(P) + T^{p+(p-i)}(P) + \sum_{k=p+(p-i)+1}^{2p} T^k(P) \\ &= \sum_{k=p+1}^{p+(p-i)-1} (n-1)(N-k)f + n[N-(2p-i)]f + \sum_{k=p+(p-i)+1}^{2p} (n-2)(N-k)f \\ &= \sum_{k=p+1}^{2p} (n-2)(N-k)f + [N-(2p-i)]f + \sum_{k=p+1}^{p+(p-i)} (N-k)f. \end{aligned}$$

Thus

$$\begin{aligned} \sum_{k=1}^{2p} T^k(P) &= \sum_{k=1}^p (n-1)(N-k)f + \sum_{k=p+1}^{2p} (n-2)(N-k)f + (2N+1-2p)f \\ &\quad + \sum_{k=1}^{i-1} (N-k)f + \sum_{k=p+1}^{p+(p-i)} (N-k)f. \end{aligned}$$

If $i = 1$ then

$$\begin{aligned} \sum_{k=1}^{2p} T^k(P) &= \sum_{k=1}^p (n-1)(N-k)f + \sum_{k=p+1}^{2p-2} (n-1)(N-k)f + n[N-(2p-1)]f + (n-1)[N-(2p)]f \\ &= \sum_{k=1}^p (n-1)(N-k)f + \sum_{k=p+1}^{2p} (n-2)(N-k)f + (2N+1-4p)f \\ &\quad + \sum_{k=p+1}^{p+(p-i)} (N-k)f. \end{aligned}$$

Therefore we have

$$\begin{aligned}
& \left| \sum_{k=1}^{2p} T^k(P) - \sum_{k=1}^{2p} T^k(P') \right| \\
&= \begin{cases} |-2p(j-i)f| & \text{if } i \neq 1, \\ |-2p(j-i+1)f| & \text{if } i = 1. \end{cases} \\
&\leq 2p^2 f.
\end{aligned}$$

Similarly we can show that for any $r \geq 0$,

$$\left| \sum_{k=2pr}^{2p(r+1)} T^k(P) - \sum_{k=2pr}^{2p(r+1)} T^k(P') \right| \leq 2p^2 f.$$

Therefore

$$\begin{aligned}
& \left| \sum_{k=1}^K T^k(P) - \sum_{k=1}^K T^k(P') \right| \\
&\leq 2p^2 f \left\lfloor \frac{K}{2p} \right\rfloor + \left| \sum_{k=\lfloor \frac{K}{2p} \rfloor 2p+1}^K [T^k(P) - T^k(P')] \right| \\
&< 2p^2 \frac{K}{2p} + 2pNf \\
&\leq 3Nf.
\end{aligned}$$

□

Unlike the parallel GJ inversion algorithm, in the parallel LU decomposition algorithm, the computation load of each processor decreases as the computation proceeds. When the computation load decreases to a certain level, there will not be enough computation to do to compensate for the processor idle time caused by data communication. Therefore full overlap of communication and computation can not be achieved throughout the whole computation process. Nevertheless, it can be shown that for a certain number c , where $0 < c < 1$, in the parallel LU factorization algorithm, up to stage cN , full overlap of communication and computation can be achieved, given the matrix size N is large enough, as stated by the following theorem.

Theorem 13 When $N > N_0$, up to the cN th iteration, where $0 < c < 1$, full overlap of communication and computation can be achieved, where N_0 is the positive root of the following quadratic equation.

$$\left[\frac{N(1-c^2)}{p} - 3p \right] Nf = \frac{1}{2}pt_s + 2(t_s + t_w N) \log p.$$

Proof: Using the above lemma, the proof is similar to the proof of Theorem 5 and therefore omitted here. □

Next we give the simulation results on the processor idle time vs the number of iterations for the parallel LU decomposition algorithm. The machine parameters are the same as in the previous sections. In Figure 11 the two curves correspond to $N = 160$ and $N = 320$ respectively, and $p = 8$. We can see for $N = 160$, up to $k = 55$, full overlap is achieved; while for $N = 320$, up to $k = 215$, full overlap is achieved.

Figure 11: The accumulated processor idle time vs. the number of iterations in the parallel LU decomposition algorithm ($p=8$).

11 Concluding Remarks

Most **parallel** matrix algorithms proposed in the literature do not attempt to overlap inter-processor communication by computation. This leads to increased communication overhead, and might lead to conclusions that are not valid when communication overhead is **systematically** minimized. For example, it is claimed in [1, 3] that the submatrix partitioning strategy for GJ algorithm is superior to the **row/column** partitioning strategy because of lower communication overhead. We, however, show that if communication and computation are efficiently overlapped, then **the** row partitioning strategy has a **lower communication overhead** than the submatrix partitioning strategy.

We first proposed a new broadcasting algorithm on the hypercube multiprocessor for parallel **GJ algorithm**. This algorithm ensures that the data are sent out from the **source** and arrives at the destinations at the earliest possible time. We then gave the parallel GJ inversion algorithm using row partitioning. We prove a lower bound on the matrix size **such** that data transmission is fully overlapped by computation. We also prove that the message length in the input buffer of each **processor** is at most 2.

We also consider the algorithms under submatrix partitioning, with or without pivoting. We show that when submatrix partitioning is used, even when **the** communication is fully overlapped by **computation**, the communication overhead is larger than when **using** row partitioning.

Our numerical simulation shows that when the algorithms proposed in this report is used, the parallel algorithms incurs much less communication overhead compared with **the** algorithm in [1], even when the communication is not completely overlapped by computation. Finally, we observe that the ideas in this report can be applied to other parallel algorithms as well, such **as** parallel LU factorization algorithm.

Appendix: Proof of Theorem 8

Proof: Let

$$\Delta T_c^k(P_{ij}) = (T_{complete}^{k-1}(P_{ij}) + s_c^k(P_{ij}) + s_r^k(P_{ij})) - (T_{send-c}^k(P_{i[k]}) + t_c^k(P_{ij}) + s_c^k(P_{ij})),$$

$$\Delta T_r^k(P_{ij}) = (T_{complete}^{k-1}(P_{ij}) + s_c^k(P_{ij}) + s_r^k(P_{ij})) - (T_{send-r}^k(P_{[k]j}) + t_r^k(P_{ij}) + s_r^k(P_{ij})),$$

We will show $\Delta T_c^k(P_{ij}) \geq 0$ and $\Delta T_r^k(P_{ij}) \geq 0$, $\forall i, j, 2 \leq k \leq N$. The proof is by induction on k .

(1) *Base case:* $k=2$.

First consider the case $i \neq 2$ and $j \neq 2$.

$$\Delta T_r^2(P_{ij}) = (T_{complete}^1(P_{ij}) + s_c^2(P_{ij}) + s_r^2(P_{ij})) - (T_{send-r}^2(P_{2j}) + t_r^2(P_{ij}) + s_r^2(P_{ij}))$$

If $T_{send-r}^2(P_{2j}) = T_{send-c}^2(P_{22}) + t_c^2(P_{2j}) + s_c^2(P_{2j}) + T'$, then

$$\begin{aligned} \Delta T_r^2(P_{ij}) &= (T_{start}^1(P_{ij}) + T^1(P_{ij}) + s_c^2(P_{ij})) - (T_{send-c}^2(P_{22}) + t_c^2(P_{2j}) + s_c^2(P_{2j}) + T' + t_r^2(P_{ij})) \\ &= (T_{start}^1(P_{ij}) + T^1(P_{ij}) + s_c^2(P_{ij})) - (T_{start}^1(P_{22}) + T' + t_c^2(P_{2j}) + s_c^2(P_{2j}) + T' + t_r^2(P_{ij})) \\ &= (T^1(P_{ij}) - 2T') + s_c^2(P_{ij}) - (T_{start}^1(P_{22}) - T_{start}^1(P_{ij})) - (t_c^2(P_{2j}) + s_c^2(P_{2j}) + t_r^2(P_{ij})) \\ &\geq \left(\frac{N^2}{p} - \frac{2N}{\sqrt{p}} \right) f - \left(\frac{N}{\sqrt{p}} f + (t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \right) - (t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \\ &= \left(\frac{N^2}{p} - \frac{3N}{\sqrt{p}} \right) f - 2(t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \\ &\geq 0. \end{aligned}$$

If $T_{send-r}^2(P_{2j}) = T_{start}^1(P_{2j}) + 2T'$, then

$$\begin{aligned} \Delta T_r^2(P_{ij}) &= (T_{start}^1(P_{ij}) + T^1(P_{ij}) + s_c^2(P_{ij})) - (T_{start}^1(P_{2j}) + 2T' + t_r^2(P_{ij})) \\ &= (T^1(P_{ij}) - 2T') + s_c^2(P_{ij}) - (T_{start}^1(P_{2j}) - T_{start}^1(P_{ij})) - t_r^2(P_{ij}) \\ &\geq \left(\frac{N^2}{p} - \frac{2N}{\sqrt{p}} \right) f - \left(\frac{N}{\sqrt{p}} f + (t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \right) - \frac{1}{2}(t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \\ &= \left(\frac{N^2}{p} - \frac{3N}{\sqrt{p}} \right) f - \frac{3}{2}(t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \\ &\geq 0. \end{aligned}$$

Similarly we can show $\Delta T_r^2(P_{ij}) \geq 0$ when $i = 2$ or $j = 2$. And similarly we can show $\Delta T_c^2(P_{ij}) \geq 0$, $\forall i, j$.

(2) *Induction Step:* Assume the proposition is true for $2 \leq l \leq k$. We prove it is also true for $l = k+1$. We just show the case $i \neq k+1$ and $j \neq k+1$, and $T_{send-r}^{k+1} = T_{send-c}^{k+1}(P_{[k+1][k+1]}) + t_c^{k+1}(P_{[k+1]j}) + s_c^{k+1}(P_{[k+1]j}) + T$. The proofs for the other cases are similar.

By the induction hypothesis, $T_{start}^l(P_{ij}) = T_{complete}^{l-1}(P_{ij}) + s_c^k(P_{ij}) + s_r^k(P_{ij})$, for $2 \leq l \leq k$, $\forall i, j$.

$$\begin{aligned}
& \Delta T_r^{k+1}(P_{ij}) \\
&= (T_{complete}^k(P_{ij}) + s_c^{k+1}(P_{ij}) + s_r^{k+1}(P_{ij})) - (T_{send-r}^{k+1}(P_{ij}) + s_r^{k+1}(P_{ij})) \\
&= (T_{start}^k(P_{ij}) + T^k(P_{ij}) + s_c^{k+1}(P_{ij})) - (T_{send-c}^{k+1}(P_{[k+1][k+1]}) + t_c^{k+1}(P_{[k+1]j}) \\
&\quad + s_c^{k+1}(P_{[k+1]j}) + T' + t_r^{k+1}(P_{ij})) \\
&= \left(T_{start}^1(P_{ij}) + \sum_{l=1}^k T^l(P_{ij}) + \sum_{l=1}^k (s_c^l(P_{ij}) + s_r^l(P_{ij})) + s_r^{k+1}(P_{ij}) \right) \\
&\quad - \left(T_{start}^1(P_{[k+1][k+1]}) + \sum_{l=1}^k T^l(P_{[k+1][k+1]}) - T^k(P_{[k+1][k+1]}) + T' \right) \\
&\quad - \left(\sum_{l=1}^k (s_c^l(P_{[k+1][k+1]}) + s_r^l(P_{[k+1][k+1]})) + t_c^{k+1}(P_{[k+1]j}) + s_c^{k+1}(P_{[k+1]j}) + T' + t_r^{k+1}(P_{ij}) \right) \\
&= T^k(P_{[k+1][k+1]}) + \sum_{l=1}^k (T^l(P_{ij}) - T^l(P_{[k+1][k+1]})) - 2T' - (T_{start}^1(P_{[k+1][k+1]}) - T_{start}^1(P_{ij})) \\
&\quad - \sum_{l=1}^k \left[(s_c^l(P_{[k+1][k+1]}) + s_r^l(P_{[k+1][k+1]}) - (s_c^l(P_{ij}) + s_r^l(P_{ij}))) \right] \\
&\quad - (t_c^{k+1}(P_{[k+1]j}) + s_c^{k+1}(P_{[k+1]j})) - t_r^{k+1}(P_{ij}) + s_r^{k+1}(P_{ij}) \\
&\geq \left(\frac{N^2}{p} - \frac{2N}{\sqrt{p}} \right) f - \left(\frac{N}{\sqrt{p}} f + (t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \right) - 2\sqrt{p}t_s - (t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \\
&= \left(\frac{N^2}{p} - \frac{3N}{\sqrt{p}} \right) f - 2\sqrt{p}t_s - 2(t_s + t_w \frac{N}{\sqrt{p}}) \log_2 p \\
&\geq 0.
\end{aligned}$$

Similarly we can show that $\Delta T_c^{k+1}(P_{ij}) \geq 0$, $\forall i, j$. $\square$

References

- [1] E. Chu, A. George and D. Quesnel, "Parallel matrix inversion on a subcube-grid ", *Parallel Computing*, 19, 1993, pp.243-256.
- [2] T.J. Dekker and W.Hoffmann, "Rehabilitation of the Gauss-Jordan algorithm ", *Numer. Math.*, 54, 1989, pp.591-599.
- [3] M. Angelaccio and M. Colajanni, "Unifying and optimizing parallel linear algebra algorithms ", *IEEE Transactions on Parallel And Distributed Systems*, Vol.4, No.12, Dec 1993, pp.1382-1397.
- [4] J.M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, New York: Plenum, 1988.
- [5] J.M. Ortega and C.H. Romine, "The ijk forms of **factorization methods II**. Parallel systems ", *Parallel Computing*, 7, 1988, pp.149-162.
- [6] S.L. Johnsson, "Minimizing the communication time for matrix multiplication on multiprocessors ", *Parallel Computing*, 19, 1993, pp.1235-1257.
- [7] S.L. Johnsson and C-T Ho, "Optimum broadcasting and personalized **communication** in hypercubes ", *IEEE Transactions on Computers*, Vol.38, No.9. Sept 1989, pp.1249-1268.
- [8] E. Chu and A. George, "A balanced **submatrix** merging algorithm for multiprocessor architectures ", *Parallel Computing*, 18, 1992, pp.1-10.
- [9] A.C. Elster and A.P. Reeves, "Block-matrix operations rising orthogonal trees ", *Third Conf. on Hypercube Concurrent Computers and Applications*, ACM, 1988, pp.1554-1561.
- [10] A. Gerasoulis, I. Nelken, N. Missirlis and R. Peskin, "**Implementing** Gauss-Jordan on a hypercube multiprocessor ", *Third Conf. on Hypercube Concurrent Computers and Applications*, ACM, 1988, pp.1569-1576.
- [11] G.C. Fox, W. Furmanski and D.W. Walker, "Optimal matrix **algorithms** on homogeneous hypercubes ", *Third Conf. on Hypercube Concurrent Computers and Applications*, ACM, 1988, pp.1554-1561.
- [12] P.G. Hipes and A. Kuppermann, "Gauss-Jordan inversion with pivoting on the Caltech Mark II hypercube ", *Third Conf. on Hypercube Concurrent Computers and Applications*, ACM, 1988, pp.1621-1634.
- [13] G.A. Geist and Michael T. Heath, "Matrix **factorization** on a hypercube multiprocessor ", *Hypercube Multiprocessors 1986*, SIAM, (Ed. M.T. Heath, 1986), pp.161-180.

- [14] Cleve Moler, "Matrix computation on distributed memory **multiprocessors** ", Hypercube **Multiprocessors 1986**, SIAM, (Ed. M.T. Heath, 1986), pp.181-195.
- [15] C.H. **Romine**, "The Parallel solution of triangular system on a hypercube ", Hypercube Multiprocessors **1987**, SIAM, (Ed. M.T. Heath, 1987), pp.552-559.
- [16] S. Lakshmivarahan and Sudarshan .K. **Dhall**, "A lower bound on the communication complexity of solving linear tridiagonal systems on cube architectures ", **Hypercube** Multiiprocessors **1987**, SIAM, (Ed. M.T. Heath, 1987), pp.560-568.
- [17] **Apostolos** Gerasoulis and Izzy Nelken, "Scheduling linear algebra parallel algorithms on MIMD Architectures ", Proceedings of the 4th SIAM Conference of Parallel **Processing** for Scientific **Computing**, 1989, pp.68-95.
- [18] U. **Meier**, "A parallel partition method for solving banded **systems** of linear **equations**", Parallel **Computing** , 2, 1985, pp.33-43.
- [19] S. **Lennart** Johnsson, "Solving tridiagonal systems on ensemble architectures ", SIAM Journal of **Scientific** and **Statistical** Computing, Vol.8, No.3, May 1987, pp.354-392.
- [20] M.D. **Levin** and D.J. Evans, "The inversion of matrices by the double-bordering algorithm on MIMD computers ", Parallel Computing, 17, 1991, pp.591-602.
- [21] Alan Geroge, Joseph W.H. Liu and Esmond Ng, "Communication results for parallel sparse Cholesky factorization on a hypercube ", Parallel Computing, 10, **1989**, pp.287-298.
- [22] S. **Lennart** Johnsson, "Communication efficient basic linear algebra computations on hypercube architectures ", Journal of Parallel And Distributed **Computing** , 4, 1987, pp.133-172.
- [23] Robert F. Lucas, Tom Blank and Jerome J. Tiemann, "A parallel **solution** method for large sparse! systems of equations ", IEEE Transactions on Computer-Aided Design, Vol.CAD-6, No.6, Nov 1987, pp.981-991.
- [24] **Ching-Tien** Ho, "Optimal broadcasting on SIMD hypercubes without indirect addressing capability ", Journal of Parallel and Distributed **Computing**, 13, 1991, pp.246-255.
- [25] Alan George, Michael T. Heath, Joseph Liu and Esmond Ng, "Sparse Cholesky factorization on a local-memory multiprocessor ", SIAM Journal of Scientific and Statistical Computing, Vol.9, No.2, Mar 1988, pp.327-340.
- [26] John L. Gustafson, Gary R. Montry and Robert E. Benner, "Development of parallel methods for a 1024-processor hypercube ", SIAM Journal of Scientific and Statistical Computing, Vol.9, No.4, Jul 1988, pp.609-638.

- [27] Anshul Gupta and Vipin Kumar, "Scalability of parallel algorithms for matrix multiplication ", **Technical Report TR91-54**, Computer Science Department, University of Minnesota.
- [28] Anshul Gupta and Vipin Kumar, "Scalability of FFT on parallel **computers** ", IEEE *Transactions* on Parallel and Distributed Systems, **Vol.4**, No.8, Aug 1993, pp.922-932.
- [29] Ananth Grama, Anshul Gupta and Vipin Kumar, "Isoefficiency measuring the scalability of parallel algorithms and architectures ", IEEE Parallel and Distributed Technology, Vol.1, No.3, Aug1993, pp.12-21.
- [30] X-H Sun and D.T. **Rover**, "Scalability of parallel algorithm-Machine **combinations** ", IEEE *Transactions* on Parallel and Distributed Systems, **Vol.5**, No.6, Jun 1994, pp.599-613.
- [31] **T.von** Eicken, D.E. Culler, S.C. Goldstein and K.E. Schauer, "Active messages: a mechanism for integrated communication and computation ", Proc. of the *19th Int'l Symp. on Computer Architecture*, Australia, May 1992, pp.256-266.
- [32] A. Gursoy and L.V. Kale, "Dagger: combining **benefits** of **synchronous** and asynchronous communication styles ", Proc. of the *8th Int'l Parallel Processing Symp.*, Mexico, 1994, pp.590-596.
- [33] **W.Dally** and et al., "Architecture of a message-driven processor ", Proc. of the 14th Annual Int'l *Symp.* on Comp. Arch., June 1987, pp.189-196.
- [34] G.M. Papadopoulos and D.E. Culler, "Monsoon: an explicit token-store architecture ", Proc. of *the 17th Annual Int'l Symp.* on Comp. Arch., **Seattle, Washington**, May 1990.