Modelling and Analysis of Communication Overhead for Parallel Matrix Algorithms

XIAODONG WANG
Department of Electrical Engineering
Texas A&M University, College Station, TX 77843, U.S.A.

V. P. ROYCHOWDHURY
Department of Electrical Engineering
University of California, Los Angeles, CA 90095, U.S.A.

(Received and accepted January 2000)

Abstract—We develop and analyze novel algorithms that make efficient use of the communication system in distributed memory architectures with processing elements interconnected by a hypercube network. The algorithms studied here include the parallel Gauss-Jordan (GJ) matrix inversion algorithm and the Gaussian Elimination (GE) algorithm for LU factorization. We first propose a new broadcasting algorithm on the hypercube multiprocessor for the parallel GJ algorithm. This algorithm ensures that the data items are sent out from the source and arrive at the destinations at the earliest possible time. We then present a parallel GJ inversion algorithm using row partitioning. This algorithm exploits a compute-and-send-ahead strategy for achieving overlapping of communication and computation, and the resulting framework leads to rigorous analytical and model-based numerical performance analysis of our parallel algorithms. In particular, we prove a lower bound on the matrix size such that data transmission is fully overlapped by computation. We also prove that the message queue length in the input buffer of each processor is at most two. We next consider the GJ algorithm under submatrix partitioning, with or without pivoting. We show that when submatrix partitioning is used, even when communication is fully overlapped by computation, the communication overhead is larger than when using row partitioning. Thus, we show that by minimizing the 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. © 2000 Elsevier Science Ltd. All rights reserved.

Keywords—Communication overhead, Parallel matrix algorithms, Overlap of communication and computation, Performance modelling and analysis, Hypercubes.

1. INTRODUCTION

One of the challenges for large-scale parallel processing is to minimize communication overhead, see [1–5]. An efficient approach to reducing communication overhead is to allow communication to overlap computation. 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 [3] and Monsoon [6], expend a significant amount of hardware to integrate communication with computation, and exhibit impressive communication
performance. For the current message passing architectures, a communication mechanism, active message, has been proposed [2], which minimizes the software overhead in message passing machines and achieves overlap of communication and computation.

The approaches mentioned above deal with the issue of reducing communication overhead from a general engineering perspective. We, however, 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 paper is analytic, which provides rigorous quantitative evaluation of the performance of the parallel algorithm. We first consider the parallel Gauss-Jordan matrix inversion algorithm on message passing hypercube multiprocessors, and then extend the idea of overlapping communication and computation to the parallel LU factorization algorithm.

1.1. Assumptions on the Machine Model

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 channels.
2. It is a multiple instruction multiple data (MIMD) parallel machine.
3. The interprocessor communication is through DMA controlled asynchronous message passing.
4. The software supports user-specified interrupt handling.

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 the communication phase requires time linear in the size of the message, plus a setup cost [2,7]. 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. Then, the time to run a program is

\[ T = T(\text{compute}) + T(\text{communicate}) = \sum_{i=1}^{N_c} (t_s + t_w L_i), \]

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 for each processor. 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., \( T(\text{compute}) \geq 0.9T(\text{communicate}) \).

A high performance network is required to minimize the communication time, and it sits 90% idle [2].

On the other hand, if communication and computation are fully overlapped, the situation is very different. The time to run a program becomes

\[ T = \max(T(\text{compute}) + N_c t_s, \sum_{i=1}^{N_c} t_w L_i). \]

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., \( T(\text{compute}) \gg N_c t_s \) [2].

Next, we introduce the communication model used in this paper, and the concept of full overlap of communication and computation. Assume 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 \( P_2 \) are 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 that 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 \( T(\text{idle}) = T_1 + (t_s + t_w m) - T_2 \).
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 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 [4,8-10]. 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 [11].

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., iPSC/860 and nCUBE 2 [8,13]. In [13], some of the algorithms representing recent developments in both classes are reviewed.

It was shown in [14] 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 [13], a medium-grain parallel algorithm for implementing GJ inversion with column interchanges on a hypercube was proposed. The authors claimed that submatrix partition had communication advantages not shared by partitions limited to rows or columns. However in their algorithm, *no attempt was made to overlap communication and computation*.

1.4. Motivation for a New GJ Inversion Algorithm

1.4.1. Sequential GJ inversion algorithm with column interchanges

We first summarize 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, $\sigma_i = i, i = 1,2, \ldots, N$. 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 values of $\sigma_k$ and $\sigma_{\text{pivot}}$. Note that in the algorithm we actually do not interchange columns, but rather, we just address the permuted columns using the information saved in $\sigma$. 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 following algorithm to $A$, $a_{i,j} = b_{k,l}$, where $k = \sigma_i$ and $\sigma_l = j$.

\begin{align*}
\text{for } k = 1 \text{ to } N \\
\quad \text{pivot} = k \\
\quad \text{for } j = k + 1 \text{ to } N \\
\quad \quad \text{if } |a_{k,\sigma_j}| > |a_{k,\sigma_{\text{pivot}}}| \text{ then pivot} = j \\
\quad \quad \sigma_k \leftrightarrow \sigma_{\text{pivot}}
\end{align*}
for \( j = 1 \) to \( N \) and \( j \neq k \)
\[
\sigma_k, \sigma_j = \frac{\sigma_k, \sigma_j}{\sigma_k, \sigma_k}
\]
for \( i = 1 \) to \( N \) and \( i \neq k \)
\[
\sigma_i, \sigma_j = \sigma_i, \sigma_j - \sigma_i, \sigma_k \cdot \sigma_k, \sigma_j
\]
for \( i = 1 \) to \( N \) and \( i \neq k \)
\[
\sigma_i, \sigma_k = -\frac{\sigma_i, \sigma_k}{\sigma_k, \sigma_k}
\]
\[
\sigma_k, \sigma_k = \frac{1}{\sigma_k, \sigma_k}.
\]


We next briefly review the GJ inversion algorithm proposed in [13] (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 the 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 determine 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 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 comprises \( d_1 = 1/2 \log p \) one-hop transfers of a message of size \( N/\sqrt{p} \). The selection and permutation of the pivot column comprises \( d_2 = 1/2 \log p \) one-hop exchanges of a message of size \( N/\sqrt{p} \). Therefore, the total communication overhead for each processor incurred in this algorithm is
\[
T_{\text{comm}} = N(d_1 + d_2)(t_s + t_w(N/\sqrt{p})) = N \log p(t_s + t_w(N/\sqrt{p})).
\]

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

1.5. Strategy for Overlapping of Communication and Computation

Two popular optimization techniques applied to parallel matrix computations for improving concurrency are the send-ahead strategy that attempts to communicate data as early as possible, and the compute-and-send-ahead strategy that computes in advance data which have to be communicated [15–18]. The aim is to avoid the idle time of the processors by overlapping communication and computation. In [15], an attempt was made to apply these two optimization strategies to parallel LU decomposition algorithms. But no efficient communication mechanism to achieve overlap of communication and computation was developed in [15]. Instead, standard
Table 1. Summary of key theoretical results in this paper, on the conditions of full overlap of communication and computation, and the total communication overhead of the parallel GJ algorithm when full overlap is achieved, under the row and submatrix partitioning (with or without pivoting). When full overlap is achieved, row partitioning has the smallest communication overhead. Moreover, for the algorithm using row partitioning, with pivoting, and the algorithm using submatrix partitioning, without pivoting, when full overlap is achieved, the total communication overhead is independent of the machine size $p$, and linearly increases with the problem size $N$.

<table>
<thead>
<tr>
<th>Partition Strategy</th>
<th>Condition</th>
<th>Total Comm. Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row, w/ Pivoting</td>
<td>$\left(\frac{N}{3} - 3\right) N f \geq \frac{1}{2} p t_s + 2 (t_s + t_w N) \log p$</td>
<td>$\frac{1}{2} N t_s$</td>
</tr>
<tr>
<td>Submatrix, w/o Pivoting</td>
<td>$\left(\frac{N^2}{p} - \frac{3N}{\sqrt{p}}\right) f \geq 2 \sqrt{p} t_s + 2 \left( t_s + t_w \frac{N}{\sqrt{p}} \right) \log p$</td>
<td>$N t_s$</td>
</tr>
<tr>
<td>Submatrix, w/ Pivoting</td>
<td></td>
<td>$\geq \frac{1}{2} N (1 + \log p) t_s$</td>
</tr>
</tbody>
</table>

Table 2. The communication overhead for the parallel GJ inversion algorithms 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 setup overhead, $t_w$ is the data transmission rate, $p$ is the number of processors. Clearly, submatrix partitioning incurs less communication overhead in this case.

<table>
<thead>
<tr>
<th>Partition Strategy</th>
<th>Total Comm. Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row Partitioning, with Pivoting</td>
<td>$N (t_s + t_w N) \log p$</td>
</tr>
<tr>
<td>Submatrix Partitioning, with Pivoting</td>
<td>$N \left( t_s + t_w \frac{N}{\sqrt{p}} \right) \log p$</td>
</tr>
</tbody>
</table>

communication subroutines provided by the software are used in their parallel algorithms. Moreover, no analytical performance analysis is provided in [15]. The experimental results also showed the superiority of the submatrix partition strategy.

To accomplish full overlap of data transmission and 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 the interrupt routine is executed, 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. Throughout this paper, we assume that the underlying machine model has this feature of communication interrupt handling.

2. SUMMARY OF RESULTS

In this paper, we first propose a new one-to-all broadcast algorithm on hypercube that helps to achieve the overlap of communication and computation. We then propose a set of asynchronous 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 bound on the matrix size such that data transmission is fully overlapped by computation. We also give 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, then the row partitioning strategy has the lowest communication overhead. Table 1 summarizes some theoretical results in this paper on the GJ algorithm; for the sake of comparison, Table 2 lists the corresponding communication complexities when no overlap of communication and computation is attempted [13,19].
Figure 1 depicts some numerical results on the total communication overhead for different GJ algorithms. Note that the vertical axis is log scaled. It is clear from this plot that overlapping 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.

In Section 7, we further apply the idea of overlapping communication and computation to the parallel LU factorization algorithm. Unlike the parallel GJ inversion algorithm, the computation load of each processor decreases as the computation proceeds. Therefore, full overlap of communication and computation cannot be achieved throughout the whole computation process. Nevertheless, we show that for any number \( c \), where \( 0 < c < 1 \), full overlapping of communication and computation can be achieved up to the \( cN^{th} \) iteration step of the LU factorization algorithm, given the matrix size \( N \) is large enough (see Theorem 10).

3. ALGORITHM I: ROW PARTITIONING WITH PIVOTING

In this section, we describe our parallel GJ inversion algorithm with column interchanges, using row partitioning. Suppose that we want to invert an \( N \times N \) matrix \( A \) on a \( d \)-dimensional hypercube containing \( p = 2^d \) processors. Assume that \( N = np \). We partition the matrix by rows and use wrap-mapping to distribute the rows of \( A \) among processors, i.e., rows \( k, p + k, 2p + k, \ldots, N - p + k \) 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 \((k - 1) \mod p + 1\), thus \( 1 \leq [k] \leq p \).
When the matrix is partitioned by rows, each row is stored 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 $k^{th}$ iteration, 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 communication and computation, the pivot row should be sent out at the earliest possible time, so that when a processor needs 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 $P[k+1]$ update the $(k+1)^{th}$ row as the first task during the $k^{th}$ iteration. Then $P[k+1]$ determines the pivot element and normalizes the updated row (compute-ahead). After that, it broadcasts this $(k+1)^{th}$ pivot row and the pivot element to all the other processors, and then resumes updating the other $(n - 1)$ rows using the $k^{th}$ pivot row (send-ahead). By using this compute-and-send-ahead strategy, the $(k+1)^{th}$ pivot row is transmitted through the communication channels while all the processors are still in the $k^{th}$ iteration.

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 present this algorithm in the next section. Whenever a message arrives at a processor, the processor is interrupted from the computation. By looking at 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 (i.e., it is a leaf node in the tree), 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 $k^{th}$ iteration, the $(k+1)^{th}$ pivot row still has not arrived at the processor, then the processor has to wait for it before starting the $(k+1)^{th}$ iteration.

The following is the pseudo code for the $k^{th}$ iteration of Algorithm I:

```plaintext
if (P = P[k+1]) then
    read in the $k^{th}$ pivot row and pivot element;
    update the $(k + 1)^{th}$ row of $A$;
    find the $(k + 1)^{th}$ pivot element;
    normalize the $(k + 1)^{th}$ pivot row;
    broadcast the $(k + 1)^{th}$ 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 $k^{th}$ row);
if (P ≠ P[k+1] and P ≠ P[k]) then
    read in the $k^{th}$ 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$_j$ 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 step 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 \((k+1)^{th}\) row of matrix \( A \) is broadcast by processor \( P_{k+1} \) immediately after it is updated and normalized in the \(k^{th}\) iteration of the parallel GJ inversion algorithm, before \( P_{k+1} \) starts updating the other rows using the \(k^{th}\) pivot row. To broadcast the \((k+1)^{th}\) row 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 setup time overhead, and the \((k+1)^{th}\) 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 show that such a set of \( p \) broadcasting trees can be obtained by appropriately modifying the well-known SBT (spanning binomial tree) broadcasting algorithm on hypercubes discussed in [20]. A number of properties of such a set of trees are established as well.

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 the physical address of each processor in a binary-reflected Gray code order with the starting address as 0 [20]. Let the \( d\)-bit code of \( P = 2^d \) integers be \( \text{Gray}(d) \), i.e., \( \text{Gray}(d) = (g_1^d, g_2^d, \ldots, g_{2^d-1}^d, g_{2^d}^d) \), where \( g_i^d \in \{0, 1\}^d \) is the code for processor \( i \). The binary-reflected Gray code on \( d \) bits is recursively defined as follows:

\[
\text{Gray}(d + 1) = (0g_1^d, 0g_2^d, \ldots, 0g_{2^d-1}^d, 1g_{2^d-1}^d, 1g_{2^d}^d, \ldots, 1g_1^d).
\]

Alternatively,

\[
\text{Gray}(d + 1) = (g_1^d0, g_1^d1, g_2^d1, g_2^d0, g_3^d0, g_3^d1, \ldots, g_{2^d-1}^d1, g_{2^d}^d0).
\]

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 paper, we will just use \( P_i \) to denote the physical address \( g_i^d \).

In [20], 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 \( c_m = 0 \), \( \forall m \in M(c) = \{q + 1, q + 2, \ldots, d - 1\} \), 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

\[
\text{children}(i) = \{(i_{d-1}i_{d-2} \ldots i_m \ldots i_0)\}, \quad \forall m \in M(c),
\]

\[
\text{parent}(i) = \begin{cases} \phi, & \text{if } i = s, \\ (i_{d-1}i_{d-2} \ldots i_q \ldots i_0), & \text{if } i \neq s. \end{cases}
\]

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 \( j \) [20]. A node is a leaf node if it has no children, i.e., \( q = d - 1 \).

**Lemma 1.** (See [19].) 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 \( 0 \leq n_c \leq d - H(s, i) \).

At the \( k^{th} \) broadcasting step 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 at the high-order \((d - 1)^{th}\) bit.
On the hypercube, if node $P$ and node $P'$ differ only at 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 that $P_{[k]}$ and $P_{[k+1]}$ differ at the $j^{th}$ bit, then we need to construct an SBT rooted at $P_{[k]}$ and with $P_{[k+1]}$ as a leaf node, such a tree is defined as follows.

**Definition 1.** SBT$_j$(s). An SBT$_j$(s) in a $d$-cube is an SBT rooted at node $s$ and the $j^{th}$ neighbor node of $s$ is a leaf node. (Note the SBT(s) defined in [20] is, therefore, SBT$_{d-1}(s)$.)

**Definition 2.** Shuffle and Inverse Shuffle Operation on a Node. The shuffle operation on a node $i = (i_d \ldots i_1 i_0)$ is defined as $S(i) = (i_{d-1} i_{d-3} \ldots i_0 i_1 i_2 \ldots i_d)$. The inverse shuffle operation on $i$ is defined as $S^{-1}(i) = (i_0 i_d i_{d-2} \ldots i_1)$. Moreover, $S^k(i)$ represents applying shuffle operation $k$ times on $i$, and $S^{-k}(i)$ represents applying inverse shuffle operation $k$ times on $i$.

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

**Lemma 2.** (See [19,]) 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 $(l + k) \mod n$. The inverse shuffle operation $S^{-k}$ maps every edge in dimension $l$ to dimension $(l - k) \mod n$.

**Corollary 1.** (See [19,]) The topology of a graph remains unchanged under shuffle and inverse shuffle operations.

The next lemma shows that SBT$_j$(s) (see Definition 1) can be obtained by applying shuffle and inverse shuffle operation on SBT$_{d-1}(s)$.

**Lemma 3.**

$$\text{SBT}_j(s) = S^{-(d-1-j)} \left[ \text{SBT}_{d-1}\left(S^{d-1-j}(s)\right) \right].$$

**Proof.** By Corollary 1, $S^{-(d-1-j)}[\text{SBT}_{d-1}(S^{d-1-j}(s))]$ is a spanning binomial tree rooted at $S^{-(d-1-j)}[\text{SBT}_{d-1}(S^{d-1-j}(s))] = s$. Let $i'$ be the leaf node at the first level of $\text{SBT}_{d-1}(S^{d-1-j}(s))$, then by definition, $i'$ differs with $S^{d-1-j}(s)$ by the $(d - 1)^{th}$ bit. Let $i$ be the corresponding node of $i'$ in $S^{-(d-1-j)}[\text{SBT}_{d-1}(S^{d-1-j}(s))]$, by Lemma 2, $i$ and $s$ differ at the $(d - 1) - (d - 1 - j) = j^{th}$ bit. Therefore, this tree is SBT$_j$(s).

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

**Theorem 1.** In SBT$_j$(s), let $i$ be any node, and $c = i \oplus s$. Let $q$ be such that $c_q = 1$ and $c_m = 0$, $\forall m \in M(c) = \{q + 1 \mod p, q + 2 \mod p, \ldots, j \mod p\}$, and let $q = j - d$ if $c = 0$. Then

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

$$\text{parent}(i) = \{ (i_{d-1} i_{d-2} \ldots \tilde{i}_q \ldots i_0) \}, \quad \text{if } i \neq s.$$

**Proof.** Let $s' = S^{d-1-j}(s)$, $i' = S^{d-1-j}(i)$, $c' = i' \oplus s'$. Clearly, $c' = S^{d-1-j}(c)$. Let $q'$ be such that $c'_{q'} = 1$ and $c'_{m} = 0$, $\forall m \in M(c') = \{q' + 1, q' + 2, \ldots, d - 1\}$. Then by definition, in SBT$_{d-1}(s')$, the edges that connect $i'$ and its children nodes are those in dimension $d - 1$, $d - 2$, $\ldots, d' + 1$. And the edge that connects $i'$ and its parent node is the one in dimension $q'$. Let $q = q' - (d - 1 - j) \mod d$. From the definition of $q'$ and the relation $c' = S^{d-1-j}(c)$, it is easy to verify that $q$ satisfies the constraints given in the theorem. By Lemma 3, SBT$_j$(s) is obtained by inverse shuffle SBT$_{d-1}(s')$ $(d - 1 - j)$ times. Then by Lemma 2, in SBT$_j$(s), the edges that
Figure 2. \( \text{SBT}'(\text{Gray}(3)) \) is a family of \( p = 2^3 = 8 \) \( \text{SBT}_j \)s. The root node of the \([k + 1]^{\text{th}} \text{SBT}_j \) is a leaf node in the \( k^{\text{th}} \text{SBT}_j \) at the first level, for all \( 1 \leq k \leq p \).

connect \( i \) and its children nodes are those in dimension \( d - 1 - (d - 1 - j) - j, j - 1 \mod d, \ldots, q' + 1 - (d - 1 - j) \mod d = q + 1 \mod d \). And the edge that connects \( i \) and its parent node is the one in dimension \( q \).

The next theorem shows the structure of \( \text{SBT}_j(s) \).

**Theorem 2.** In \( \text{SBT}_j(s) \), 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 \( \binom{l}{j} \), and the height of the tree is \( d \). The number of children nodes of any node \( i \) is \( (j - q) \mod (d + 1) \). The number of leaf nodes is \( 2^{d-1} \).

**Proof.** The first three statements follow directly from Lemma 1, Lemma 2, and Corollary 1. By Theorem 1, in \( \text{SBT}_j(s) \) node \( i \) is a leaf node if and only if the \( j^{\text{th}} \) bit of \( i \) (\( i_j \)) differs from the \( j^{\text{th}} \) bit of \( s \) (\( s_j \)). For a given root \( s \), there are \( 2^{d-1} \) nodes \( i \) such that \( i_j \neq s_j \). Therefore, there are \( 2^{d-1} \) leaf nodes in \( \text{SBT}_j(s) \).

Next, we give the construction of the \( p \) broadcasting trees needed in Algorithm I. As defined below, each tree in the family is an \( \text{SBT}_j \), such that the root node of the \([k + 1]^{\text{th}} \text{SBT}_j \) is a leaf node in the \( k^{\text{th}} \text{SBT}_j \) at the first level, for all \( 1 \leq k \leq p \).
DEFINITION 4. A FAMILY OF \( p = 2^d \) BROADCASTING TREES \( \text{SBT}(\text{Gray}(d)) \). Given \( \text{Gray}(d) = (g_1^d, g_2^d, \ldots, g_{2^d-1}^d, g_{2^d}^d), g_i^d \in \{0,1\}^d \), \( \text{SBT}(\text{Gray}(d)) \) is a family of \( p = 2^d \) \( \text{SBT}_j \)'s:

\[
\text{SBT}(\text{Gray}(d)) = (\text{SBT}(1), \text{SBT}(2), \ldots, \text{SBT}(2^d)),
\]

where \( \text{SBT}(i) = \text{SBT}_j(g_i^d), \) and \( j(i) \) is the bit where \( g_i^d \) and \( g_{i+1}^d \) differ at.

As an example, consider \( d = 3 \). We have

\[
\text{Gray}(3) = (000, 001, 011, 010, 110, 111, 101, 100),
\]

and Figure 2 shows the corresponding \( \text{SBT}_j \)'s of \( \text{SBT}(\text{Gray}(3)) \).

The next theorem shows the structure of \( \text{SBT}(\text{Gray}(d)) \), and leads to the setup overhead complexity of the broadcasting algorithm.

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

PROOF. See Appendix A.

The setup overhead complexity of the Algorithm I is as follows.

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

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

5. ANALYSIS OF ALGORITHM I

While the previous two sections describe the parallel GJ inversion algorithm and the broadcasting protocol, in this section we analyze the performance of the Algorithm I given in Section 3. In particular, we address the following issues.

1. Contention Free Broadcasting. We show (see Theorem 4) that if the \( \text{SBT}_j \) broadcasting algorithm is used in Algorithm I, then there will be no conflict between successive broadcasting steps, that is, whenever a message arrives at an intermediate processor, it can be forwarded immediately without delay.

2. Full Overlap of Communication and Computation. We show (see Theorem 5) that when the matrix size \( N \) is sufficiently large, the idle time is zero for each processor.

3. Message Queue Length. We show (see Theorem 6) that the message queue length is at most 2 when full overlap is achieved.

4. Numerical Results. We also present the numerical results on the communication overhead of Algorithm I for partial and full overlap.

We begin our analysis by considering the computation load of each processor at the \( k \)-th iteration. The updating of each element not on the \( k \)-th pivot column involves one multiplication and one subtraction, and the updating of each element on the \( k \)-th pivot column involves one division. For simplicity, we assume that the time to update each element of \( A \) is \( f \). For processor \( P_{k+1} \), since it has already normalized the \( k \)-th pivot row and found the pivot element in the previous \( (k - 1) \)-th iteration, it needs to update the rest \( n - 1 \) rows it has using the \( k \)-th pivot row and pivot element. We call this computational task \( \Pi_2 \), and the corresponding computation time is \( T_1 \), where \( T_1 = (n - 1)Nf \). For processor \( P_{k+1} \), its computational task in the \( k \)-th iteration can be split into two parts, \( \Pi_2 \) and \( \Pi_3 \). \( \Pi_2 \) corresponds to updating the \((k + 1) \)-th row of \( A \) using...
the $k^{th}$ pivot row, normalizing this row and searching the pivot element of the $(k+1)^{th}$ pivot row. The time for updating is $N_f$. We assume that it takes another $N_f$ to determine the pivot element and normalize this row. Therefore, the time for $\Pi_2$ is $T_2 = 2N_f$. After completing $\Pi_2$ processor $P_{[k+1]}$ immediately broadcasts the $(k+1)^{th}$ pivot row. Then it resumes to update the other $(n-1)$ rows using the $k^{th}$ pivot row, we call this task $\Pi_3$, and the corresponding time $T_3 = T_2 = (n-1)N_f$. For all the other processors, the computational task during the $k^{th}$ iteration is $\Pi_4$, which corresponds to updating all the $n$ rows using the $k^{th}$ pivoting row. The time for $\Pi_4$ is $T_4 = nN_f$. Therefore, the computational load of processor $P$ at the $k^{th}$ iteration of Algorithm 1 is

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

Note that there is a computational load imbalance among the processors in the $k^{th}$ iteration. $P_{[k+1]}$ 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 $E$. A node in $G$ represents one of the four computational tasks $\Pi_1$, $\Pi_2$, $\Pi_3$, and $\Pi_4$. 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 end 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 $8 \times 8$ matrix using four 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 $k^{th}$ iteration of the parallel GJ inversion.

**PROPERTY 3.** In the $k^{th}$ stage of $G$, the node in column $[k]$ is a $\Pi_1$ node; in column $[k+1]$ is a $\Pi_2$ node followed by a $\Pi_3$ node; and the nodes in all other columns are $\Pi_4$ nodes.

**PROPERTY 4.** All nodes at the first stage of $G$ has in-degree 1. At stages other than the first, a $\Pi_1$ or $\Pi_3$ node has in-degree 1, and a $\Pi_2$ or $\Pi_4$ node has in-degree 2.

**PROPERTY 5.** A $\Pi_2$ 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 $G$: "vertical" arcs and "nonvertical" 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 $G$. 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 $(k+1)^{th}$ iteration only after it completes the $k^{th}$ iteration. The communication delay introduced by a vertical arc is the communication setup 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 setup overhead and this overhead is represented by the vertical arc. A nonvertical arc is an arc from a $\Pi_2$ node to a $\Pi_3$ node or to a $\Pi_4$ node at the next stage in $G$. The precedence constraint it represents is that the
Table 3. Parameters used in the analysis of Algorithm I. $t^k(P)$ can be calculated as follows: 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, $t^k(P) = H(P_{[k]}, P)(t_s + t_w N)$. $s^k(P)$ is determined as follows: at stage $k$ in $G$, a $\Pi_2$ or a $\Pi_4$ node has two incoming arcs, the delay introduced by the vertical arc is $s^k(P)$ and the delay introduced by the nonvertical 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. A $\Pi_1$ node has one vertical incoming arc, which introduces no delay, i.e., $s^k(P) = 0$. A $\Pi_3$ node has one vertical incoming arc, which introduces delay $s^k(P) = t_s$.

<table>
<thead>
<tr>
<th>$T^k_{\text{start}}(P)$</th>
<th>The time processor $P$ starts the $k$th iteration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T^k_{\text{complete}}(P)$</td>
<td>The time processor $P$ completes the $k$th iteration.</td>
</tr>
<tr>
<td>$T^k(P)$</td>
<td>The computation load of processor $P$ at the $k$th iteration.</td>
</tr>
<tr>
<td>$T^k_{\text{send}}$</td>
<td>The time the $k$th pivot row is sent out.</td>
</tr>
<tr>
<td>$T^k_{\text{arrive}}(P)$</td>
<td>The time the $k$th pivot row arrives at $P$.</td>
</tr>
<tr>
<td>$s^k(P)$</td>
<td>The setup overhead of $P$ at the $k$th stage.</td>
</tr>
<tr>
<td>$t^k(P)$</td>
<td>The time required by the $k$th message to arrive at processor $P$ after it is sent out by $P_{[k]}$, assuming no delay at any intermediate processor.</td>
</tr>
</tbody>
</table>
computation in the head node cannot start before it receives the message from the tail node, i.e., a processor cannot start the \((k + 1)^{\text{th}}\) iteration before it gets the \((k + 1)^{\text{th}}\) pivot row from \(P_{[k+1]}\). The communication delay introduced by a nonvertical 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 in Table 3, and the relationships among these variables are illustrated in Figure 4.

**LEMMA 4.** We have the following recurrent relationships:

\[
T_{\text{start}}^k(P) = \max \left\{ T_{\text{complete}}^{k-1}(P), T_{\text{arrive}}^k(P) \right\} + s^k(P), \quad \text{if } P \neq P_{[k]},
\]

\[
T_{\text{complete}}^k(P) = T_{\text{start}}^k(P) + T^k(P),
\]

\[
T_{\text{send}}^k = T_{\text{start}}^k(P_{[k]}) + T_2.
\]

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

The next theorem states that a message cannot be delayed at an intermediate processor during the broadcast process in Algorithm I.
THEOREM 4. In Algorithm I, when the SBT\textsubscript{j} broadcast algorithm is used, then

\[ T_{\text{arrive}}^k(P) = T_{\text{send}}^k + t^k(P). \]

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 forwarding the current 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 \( T_{\text{arrive}}^k(P) - T_{\text{arrive}}^{k-1}(P) > t_s \), then the 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 \((k - 1)\)th SBT\textsubscript{j}, then as long as \( T_{\text{arrive}}^k(P) - T_{\text{arrive}}^{k-1}(P) > 0 \), message \( k \) will not be delayed at processor \( P \), since \( P \) can immediately forward message \( k \).

Assume that \( H(P_{[k-1]}, P) = m \), 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 that \( P_{[k]} \) and \( P_{[k-1]} \) differ at 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

\[ T_{\text{send}}^k - T_{\text{send}}^{k-1} = [T^k_{\text{start}}(P_{[k]}) + T_2] - T_{\text{send}}^{k-1} \geq (T_{\text{send}}^{k-1} + t_s + t_wN + T_2) - T_{\text{send}}^{k-1} = (t_s + t_wN) + T_2. \]

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 that message \((k - 1)\) is not delayed at any processor \( P \), i.e.,

\[ T_{\text{arrive}}^{k-1}(P) = T_{\text{send}}^{k-1} + t^{k-1}(P) = T_{\text{send}}^{k-1} + m(t_s + t_wN). \]

We need to show message \( k \) also cannot be delayed. We consider the following two cases.

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

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

\[ T_{\text{comp}}(P) = \sum_{k=1}^{N} T^k(P) \]

and

\[ T_{\text{comm}}(P) = T_{\text{complete}}^N(P) - T_{\text{comp}}(P) - T_{\text{complete}}^N(P) - \sum_{k=1}^{N} T^k(P). \]

It can be seen from Figure 4 that at the \((k - 1)\)th iteration, if \( T_{\text{arrive}}^k(P) > T_{\text{complete}}^{k-1}(P) \), then processor \( P \) will be idle for a time \( t_{\text{idle}}^{k-1}(P) = T_{\text{arrive}}^k(P) - T_{\text{complete}}^{k-1}(P) \); otherwise \( t_{\text{idle}}^{k-1}(P) = 0 \).

If \( t_{\text{idle}}^k(P) = 0 \) 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. Full Overlap of Communication and Computation.** We say that the communication is fully overlapped by computation at processor \( P \) in Algorithm I if

\[
T^{k-1}_{\text{complete}}(P) \geq T^k_{\text{send}} + t^k(P), \quad 2 \leq k \leq N,
\]
or equivalently,

\[
T^k_{\text{start}}(P) = T^{k-1}_{\text{complete}}(P) + s^k(P), \quad 2 \leq k \leq N.
\]

That is, the \( k \)th pivot row has already arrived at processor \( P \) when \( P \) completes the \((k-1)\)th iteration.

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 determines the pivot element (which takes time \( Nf \)), before sending it out, we get

\[
d_0(P) = H(P_1,P)(t_s + t_w N) + Nf.
\]

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 any particular time.

**Definition 6. Message Queue Length.** Let \( Q^k(P) \) be the message queue length of processor \( P \) at stage \( k \), then

\[
Q^k(P) = 1, \quad \text{if} \quad T^{k-1}_{\text{send}} + t^k(P) < T^k_{\text{complete}}(P) < T^{k+1}_{\text{send}} + t^{k+1}(P).
\]

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

**An Intuitive Argument on the Condition for Full Overlap.** We next consider the condition for full overlap of communication and computation in Algorithm I. The minimum computational load at each iteration is \((n-1)Nf\). Intuitively, this time should be greater than the maximum possible message delay, to achieve full overlap. Since the algorithm is asynchronous, the maximum message delay consists of three parts:

1. the initial delay \( d_0(P) \) which can be up to \((t_s + t_w N) \log p\);
2. the total previous setup overheads, which can be up to \((1/2)pt_s\) (see Appendix B); and
3. the message transmission delay, which can be up to \((t_s + t_w N) \log p\).

Therefore, an intuitive condition for full overlap is \((n-1)Nf > (1/2)pt_s + 2(t_s + t_w N) \log p\).

The next theorem, however, gives a rigorous sufficient condition for full overlap; the proof is constructed by showing that for large enough \( N \), the conditions in Definition 5 are always satisfied.

**Theorem 5.** Full overlap of communication and computation can be achieved in Algorithm I if \( N \geq N_0 \), 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.** See Appendix B.

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

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

**Proof.** The proof is by contradiction. Suppose that there exist \( k \) and \( P \) such that \( Q^k(P) = l \geq 3 \). By the definition of the queue length \( Q^k(P) \), and using Lemma 4, we have

\[
T^k_{\text{complete}}(P) \geq T^{k+l}_{\text{send}} + t^{k+l}(P) = [T^{k+l-1}_{\text{start}}(P_{k+l}) + T_2] + t^{k+l}(P)
\]

\[
\geq [T^{k+l-2}_{\text{complete}}(P_{k+l}) + s^{k+l-1}(P_{k+l})] + T_2 + t^{k+l}(P).
\]
Since $N \geq N_0$, by Theorem 5, communication is fully overlapped by computation, thus $T_{\text{complete}}^k(P) = d_0(P) + \sum_{i=1}^{k} [T^i(P) + s^i(P)]$. Hence, substituting this in the above equation, we have

$$
d_0(P) + \sum_{i=1}^{k} [T^i(P) + s^i(P)] \geq d_0(P_{k+l})
$$

$$
\sum_{i=1}^{k} [T^i(P_{k+l}) + s^i(P_{k+l})] + T_2 + t^{k+l}(P).
$$

Hence,

$$
\sum_{i=1}^{k+l-2} T^i(P_{k+l}) - \sum_{i=1}^{k} T^i(P) + T_2 \leq (d_0(P) - d_0(P_{k+l})) - t^{k+l}(P)
$$

$$
- \sum_{i=1}^{k+l-1} s^i(P_{k+l}) + \sum_{i=1}^{k} s^i(P) \leq (t_s + t_w N) \log p + \frac{1}{2} t_s.
$$

But when $l \geq 3$,

$$
\sum_{i=1}^{k+l-2} T^k(P_{k+l}) - \sum_{i=1}^{k} T^k(P) + T_2 \geq T_1 + T_2 = \left(\frac{N}{p} + 1\right) N_f > \left(\frac{N}{p} - 3\right) N_f
$$

$$
> 2 (t_s + t_w N) \log p + \frac{1}{2} pt_s
$$

$$
> (t_s + t_w N) \log p + \frac{1}{2} pt_s.
$$

Equations (1) and (2) contradict each other. Therefore, $Q^k(P) \leq 2$. 

**Comm. Overhead $T_{\text{comm}}$**

![Figure 5. Communication overhead of Algorithm I (number of processors $p = 16$).](image-url)
**COROLLARY 3:** When \( N \geq N_0 \), the communication overhead of processor \( P \) in Algorithm I is \( d_0(P) + (N/2)t_s \).

**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 by 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 all the processors during the rest of the updating process \((2 \leq k \leq N)\). 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 \((N/2)t_s\). Therefore, the total communication overhead is \( d_0(P) + (N/2)t_s \).

Next, we give the simulation results on the communication overhead 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 \( t_s = 150 \) and \( t_w = 3 \). These parameters are close to that of the nCUBE 2 machine [21]. Since the communication overhead may be different for different processors, the data depicted in the figure are the largest overhead among the processors. Figure 5 depicts the communication overhead \( T_{\text{comm}} \) vs. the matrix size \( N \) when the machine size \( p = 16 \). Curve 1 is the total communication overhead; Curve 2 is the total communication overhead minus the initial delays (i.e., \( d_0(P) \)); for the case of Curve 3, we let each processor initially store the first row of \( A \), such that all processors can start without the initial delay. It can be seen that when there is an initial delay, then for \( N > 250 \) the communication is "almost" overlapped by computation, but not completely until \( N > 450 \). When there is no initial delay, the communication is completely overlapped by computation when \( N > 250 \).

Figure 6 shows the communication overhead 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 sizes (if the initial delay is not considered), and linearly increases with the problem size (i.e., $T_{\text{comm}} = (1/2)Nt_a$).

6. ALGORITHMS USING SUBMATRIX PARTITIONING

The main advantages of the submatrix partitioning are shorter message lengths, i.e., each message is of length $N/\sqrt{p}$ instead of $N$; and shorter communication propagation delay, i.e., each broadcasting tree is of height $d/2$ instead of $d$. However, as discussed in Section 1.4.2, the synchronization barrier caused by recursive doubling when searching for the pivot element makes it difficult to overlap communication and computation. In Section 6.1, we discuss Algorithm II, which uses submatrix partitioning, but without partial pivoting. We show that when the matrix size is large enough, full overlap can be achieved. But when full overlap is achieved, the communication overhead is twice that of Algorithm I. In Section 6.2, we discuss Algorithm III, which uses submatrix partitioning and with partial pivoting. We show that because of the synchronization barrier caused by the recursive doubling, it is difficult to achieve full overlap. Furthermore, the setup overhead alone of Algorithm III is much larger than the total communication overhead of Algorithm II and Algorithm I.

6.1. Algorithm II: Submatrix Partitioning Without Pivoting

The processors are configured as a $\sqrt{p} \times \sqrt{p}$ array. The physical address of each processor is determined by the 2D Gray code mapping. We still use the compute-and-send-ahead strategy to achieve overlap of communication and computation. In Algorithm II, there are concurrent broadcasts within subcubes at each iteration: first, there are $\sqrt{p}$ concurrent broadcasts of segments of multipliers within the subcube rows; then there are $\sqrt{p}$ concurrent broadcasts of segments of the pivot row within the subcube columns. Figure 7 shows the data flow for Algorithm II where $p = 16$. Let $(I, J)$ be the coordinates for processor $P$ in the processor array, where $1 < I < p_r$ and $1 \leq J \leq p_c$. The pseudo code for the $k$th step of the Algorithm II is as following:

![Figure 7. Data flow of Algorithm II.](image-url)
if \((I \neq [k + 1] \text{ and } J \neq [k + 1])\) 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,
  it to its children nodes in the column subcube, if there is any;

if \((I = [k + 1] \text{ and } J = [k + 1])\) 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 \(A\);

**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 \((I \neq [k + 1] \text{ and } J = [k + 1])\) 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 column
  pass it to its children nodes in the column subcube, if there is any;

if \((I = [k + 1] \text{ and } J \neq [k + 1])\) 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;

In what follows, we give the performance analysis of Algorithm II. For simplicity, we only consider the case the hypercube dimension \(d\) is even, i.e., \(p_c = p_r = \sqrt{p} = 2^{d/2}\). 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.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T^k_{\text{start}}(P_{ij}))</td>
<td>The time (P_{ij}) starts the (k^{th}) loop.</td>
</tr>
<tr>
<td>(T^k_{\text{complete}}(P_{ij}))</td>
<td>The time (P_{ij}) completes the (k^{th}) loop.</td>
</tr>
<tr>
<td>(T^k(P_{ij}))</td>
<td>The computation load of (P_{ij}) at the (k^{th}) loop.</td>
</tr>
<tr>
<td>(T')</td>
<td>The time to update or normalize one segment of the multipliers or the pivot row elements.</td>
</tr>
<tr>
<td>(T^k_{\text{send-row}}(P_{ij}))</td>
<td>The time that the segment of (k^{th}) row of (A) is sent by (P_{ij}).</td>
</tr>
<tr>
<td>(T^k_{\text{send-column}}(P_{ij}))</td>
<td>The time that the segment of the (k^{th}) column of (A) is sent by (P_{ij}).</td>
</tr>
<tr>
<td>(t^k_{\text{row}}(P_{ij}))</td>
<td>The time it takes for this segment of the (k^{th}) row to reach (P_{ij}).</td>
</tr>
<tr>
<td>(t^k_{\text{column}}(P_{ij}))</td>
<td>The time it takes for this segment of the (k^{th}) column to reach (P_{ij}).</td>
</tr>
<tr>
<td>(s^k_{\text{setup}}(P_{ij}))</td>
<td>The setup overhead of (P_{ij}) when broadcasting the (k^{th}) row segment.</td>
</tr>
<tr>
<td>(s^k_{\text{setup}}(P_{ij}))</td>
<td>The setup overhead of (P_{ij}) when broadcasting the (k^{th}) column segment.</td>
</tr>
</tbody>
</table>
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 $N t_s$.

**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 SBTs, for consecutive $\sqrt{p}$ broadcasting steps, each processor incurs setup overhead of $(1/2)Nt_s$. Therefore, the total setup overhead for each processor is $2 \cdot (1/2)Nt_s = N t_s$. 

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 cannot be immediately pass over. Instead, it will be delayed until the processor finishes setting up the channel for the previous message. This situation cannot happen in Algorithm I, as proved in Theorem 4. In the worst case, a message is delayed for a time of $t_s$ at an intermediate processor. As a result, we have

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

**LEMMA 5.** We have the following recurrent relationship:

$$T^k_{\text{start}}(P_{ij}) = \begin{cases} \max \left\{ T^{k-1}_{\text{complete}}(P_{ij}) + s^k(P_{ij}) + s^k(P_{ij}), T^k_{\text{send-c}}(P_{[k]i}) \right. \\ + t_s^k(P_{ij}) + s^k(P_{ij}), T^k_{\text{send-r}}(P_{[k]j}) + t_s^k(P_{ij}) + s^k(P_{ij}) \right\}, & i \neq [k], j \neq [k], \\
\max \left\{ T^{k-1}_{\text{complete}}(P_{[k]i}) + s^k(P_{[k]i}) + s^k(P_{[k]i}), T^k_{\text{send-c}}(P_{[k]i}) \right. \\ + t_s^k(P_{[k]i}) + s^k(P_{[k]i}), T^k_{\text{send-r}}(P_{[k]j}) + t_s^k(P_{[k]i}) + s^k(P_{[k]i}) \right\}, & i = [k], j \neq [k], \\
\max \left\{ T^{k-1}_{\text{complete}}(P_{[k]i}) + s^k(P_{[k]i}) + s^k(P_{[k]i}), T^k_{\text{send-c}}(P_{[k]i}) \right. \\ + t_s^k(P_{[k]i}) + s^k(P_{[k]i}), T^k_{\text{send-r}}(P_{[k]i}) + t_s^k(P_{[k]i}) + s^k(P_{[k]i}) \right\}, & i \neq [k], j = [k], \\
T^{k-1}_{\text{complete}}(P_{[k]i}) + s^k(P_{[k]i}) + s^k(P_{[k]i}), & i = [k], j = [k]. \end{cases}$$

We say that communication is fully overlapped by computation if

$$T^k_{\text{start}}(P_{ij}) - T^{k-1}_{\text{complete}}(P_{ij}) + s^k(P_{ij}) + s^k(P_{ij}), \quad 2 \leq k \leq N, \quad \forall i, j.$$ 

The next theorem gives a sufficient condition on full overlap of communication and computation in Algorithm II.

**THEOREM 8.** Full overlap of communication and computation can be achieved if $N \geq N_0$, where $N_0$ is the positive root of the following quadratic equation:

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

**PROOF.** See Appendix C. At the beginning of Algorithm II, each processor except $P_{11}$ must be initially idle until it gets the multipliers and/or the first pivot row elements. Let $d_0(P_{ij})$ be the initial delay of $P_{ij}$.
(a) This figure shows communication overheads of Algorithm II (number of processors $p = 16$).

(b) This figure shows communication overheads of Algorithm II with different machine sizes.
COROLLARY 4. When \( N \geq N_0 \), the communication overhead of processor \( P_{ij} \) in Algorithm II is \( d_0(P_{ij}) + N t_s \).

Next, we give the simulation results on the communication overhead of Algorithm II. Figure 8a depicts the communication overhead \( T_{\text{comm}} \) vs. the matrix size \( N \) when the machine size \( p = 16 \). As a comparison, we put the corresponding curve of Algorithm I (Curve 1 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 \( N \) gets larger, 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 8b shows the communication overhead 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., \( T_{\text{comm}} = N t_s \)).

6.2. 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 cannot be done by a single processor. Instead, a row of processors each determines the local pivot element and then finds the pivot element through recursive doubling. As noted earlier, recursive doubling causes synchronization barriers among the processors and incurs a much higher setup overheads. During the \( k \)th iteration, the \((k + 1)\)th pivot row elements can be computed ahead and sent ahead by a row of processors; but the multipliers cannot 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 update its share of the submatrix column by column. As soon as the message containing the pivot row elements arrives, the processor will be interrupted and will start searching for 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 \( d/2 \) write-read pairs, each processor will get the pivot element and the corresponding segments of multipliers. Then it will normalize the pivot row elements using the pivot element.

Therefore, at each iteration of Algorithm III, each processor will be interrupted \((d/2) + 1\) times: one for broadcasting the pivot row elements; \(d/2\) 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 \((1/2)(1 + \log p) N t_s\).

PROOF. The one-to-all broadcasting steps of the pivot row elements within the subcube-columns causes setup overhead of \((1/2)N t_s\). At each iteration, the permutation within the subcube-rows to find the pivot elements causes each processor a setup overhead of \((1/2)t_s \log p\), and there are totally \( N \) iterations. So the overall setup overhead for each processor is \((1/2)(1 + \log p) N t_s\).

Therefore, Algorithm III incurs much larger setup overhead than Algorithm I and Algorithm II. Furthermore, 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.

7. LU FACTORIZATION

In this section, we apply the ideas of overlapping communication and computation to the parallel LU factorization algorithm [17,22,23], using the row partitioning strategy and column interchanges for partial pivoting.

The pseudocode for the sequential row oriented LU factorization algorithm is shown below:

\[
\text{for } k = 1 \text{ to } N - 1
\]
\[
\text{for } i = k + 1 \text{ to } N
\]
\[ l_{jk} = \frac{a_{jk}}{a_{kk}} \]

for \( j = k + 1 \) to \( N \)
\[ a_{ij} = a_{ij} - l_{ik}a_{kj}. \]

The \( j \) loop subtracts multipliers of the \( k \)th row of the current \( A \) from succeeding rows. At the \( k \)th 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 \( k \)th 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 \[16\]. Here the rows are distributed as illustrated in Figure 9 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.

\begin{align*}
\text{rows 1, 8, 9, 16} & \quad \text{rows 2, 7, 10, 15} & \quad \text{rows 3, 6, 11, 14} & \quad \text{rows 4, 5, 12, 13}
\end{align*}

Figure 9. 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 \( k \)th stage. Let \( q = \lfloor k/p \rfloor \), then the computation load of processor \( P \) at the \( k \)th stage is
\[ T^k(P) = \begin{cases} 
(n - q - 1)(N - k)f, & \text{if } P \in \{P_{(pq+1)}, P_{(pq+2)}, \ldots, P_{(k)}\}, \\
(n - q + 1)(N - k)f, & \text{if } P = P_{(k+1)}, \\
(n - q)(N - k)f, & \text{if } P \in \{P_{(k+2)}, \ldots, P_{(p(q+1))}\}. 
\end{cases} \]

The following lemma shows the computation load difference between any two processors at any time is bounded by a linear function of \( N \).

**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.** See Appendix D.

Unlike the parallel GJ inversion algorithm, in the parallel LU factorization 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 cannot be achieved throughout the whole computation process. Nevertheless, it can be shown that for any number \( c \), where \( 0 < c < 1 \), in the parallel LU factorization algorithm, up to stage \( cN \), full overlapping of communication and computation can be achieved, given the matrix size \( N \) is large enough, as stated by the following theorem.
THEOREM 10. In the parallel LU factorization algorithm using row partitioning, with column interchange for partial pivoting, full overlap of communication and computation can be achieved for up to the cNth iteration (0 < c < 1), where \( N_0 \) is the positive root of the following quadratic equation:

\[
N_0 = \left[ \frac{N (1-c^2)}{p} - 3p \right] N f = \frac{1}{2} pt_s + 2 (t_s + t_w N) \log p.
\]

PROOF. Using Lemma 6, the proof is similar to the proof of Theorem 5 and hence, is omitted here.

Next, we give the simulation results on the processor idle time vs. the number of iterations for the parallel LU factorization algorithm. The machine parameters are the same as in the previous sections. In Figure 10, 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.

8. CONCLUDING REMARKS

Most parallel matrix algorithms proposed in the literature do not attempt to overlap interprocessor 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 [13,15] 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 simulations show that when the algorithms proposed in this paper are used, the parallel algorithms incurs much less communication overhead compared with the algorithm in [13], even when the communication is not completely overlapped by computation. Finally, we observe that the ideas in this paper can be applied to other parallel algorithms as well, e.g., parallel LU factorization algorithm.

APPENDIX A

PROOF OF THEOREM 3

PROOF. By Theorem 2, in SBT_j(s), any node i is at level \( H(s, i) \). For any fixed i, there are \( \binom{d}{2} \) ss such that \( H(s, i) = l \). And in the corresponding SBTs rooted at these nodes, i is at level l.

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

BASE CASE. d = 1. The proposition is obviously true.

INDUCTION STEP. Assume that the proposition is true for dimension d, we show it is also true for dimension \( d + 1 \).

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

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

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

For SBT(\( g_2, \ldots, g_2, g_1 \)), let

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

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(\( 0g_1, \ldots, 0g_{2^d}, 1g_2, \ldots, 1g_1 \)), for any node 0i, we have

\[
\sum_{m=1}^{2^{d+1}} \phi_{d+1}(0i, g_m^{d+1}) = \sum_{m=1}^{2^{d-1}} \phi_d(i, g_m) + 0 + \sum_{m=2^d}^{2^d} \phi'_d(i, g_m) + 1
\]

\[
= \sum_{m=1}^{2^d} \phi_d(i, g_m) + \sum_{m=2^d}^{2^d} \phi'_d(i, g_m) + 1 - (\phi_d(i, g_2^d) + \phi'_d(i, g_1))
\]

\[
= 2^{d-1} + 2^{d-1} + 1 - (\phi_d(i, g_2^d) + \phi'_d(i, g_1)).
\]
Suppose that \(g_2\) and \(g_1\) differ by the \(j\)th bit. If \(\phi_d(i, g_2) = 1\), then the \(j\)th bits of \(i\) and \(g_2\) are different, so the \(j\)th bits of \(i\) and \(g_1\) are the same, and \(\phi_d(i, g_1) = 0\). Similarly, if \(\phi_d(i, g_1) = 1\), \(\phi_d(i, g_2) = 0\). Therefore, \(1 - (\phi_d(i, g_2) + \phi_d(i, g_1)) = 0\). Thus, \(\sum_{m=1}^{2^{d+1}} \phi_d(0i, g_m^{d+1}) = 2^d\).

\[
\sum_{m=1}^{2^{d+1}} \phi_d(1i, g_m^{d+1}) = \sum_{m=1}^{2^d} \phi_d(i, g_m) + 1 + \sum_{m=2^d}^{2^{d+1}} \phi_d(i, g_m) + 0
\]

\[
= \sum_{m=1}^{2^d} \phi_d(i, g_m) + 1 - (\phi_d(i, g_2) + \phi_d(i, g_1))
\]

\[
= 2^{d-1} + 2^{d-1} + 1 - (\phi_d(i, g_2) + \phi_d(i, g_1)) = 2^d.
\]

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

**APPENDIX B**

**PROOF OF THEOREM 5**

**PROOF.** We rewrite the quadratic equation in the standard form

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

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

\[
\left(\frac{N}{p} - 3\right) N f \geq \frac{1}{2} p t_s + 2 (t_s + t_w N) \log p.
\]

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

**BASE CASE.** \(k = 2\).

\[
\Delta T^2(P) = T_{\text{complete}}^2(P) - T_{\text{end}}^2 - t^2(P) = [T_{\text{start}}^1(P) + T^1(P)] - [T_{\text{start}}^2(P_2) + T_2] - t^2(P)
\]

\[
= (d_0(P) - T_2) - (d_0(P_2) - d_0(P)) - t^2(P),
\]

where

\[
T^1(P) - T_2 = \begin{cases}
T_1 - T_2 = (n - 3)Nf, & P = P_1, \\
T_4 - T_2 = nNf - 2Nf = (n - 2)Nf, & P \neq P_1,
\end{cases}
\]

and

\[
d_0(P_2) - d_0(P) \leq (t_s + t_w N) \log p, \quad t^2(P) \leq (t_s + t_w N) \log p.
\]

Therefore,

\[
\Delta T^2(P) \geq (n - 3)Nf - 2(t_s + t_w N) \log p = \left(\frac{N}{p} - 3\right) N f - 2(t_s + t_w N) \log p > 0,
\]

when \(N \geq N_0\).
**INDUCTION STEP.** Assume that the proposition is true for \(2 \leq l \leq k\). We prove it is also true for \(l = k + 1\).

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

\[
\Delta T^{k+1}(P) = T^{k}_{\text{complete}}(P) - T^{k+1}_{\text{send}} - t^{k+1}(P)
\]

\[
= \left[ T^{k}_{\text{start}}(P) + T^{k}(P) \right] - \left[ T^{k}_{\text{start}}(P_{[k+1]}) + T_{2} \right] - t^{k+1}(P)
\]

\[
= \left[ d_{0}(P) + \sum_{i=1}^{k} (T^{i}(P) + s^{i}(P)) \right]
\]

\[
- \left[ d_{0}(P_{[k+1]}) + \sum_{i=1}^{k} (T^{i}(P_{[k+1]}) + s^{i}(P_{[k+1]})) - T^{k}(P_{[k+1]}) + T_{2} \right] - t^{k+1}(P)
\]

\[
= \sum_{i=1}^{k} \left[ T^{i}(P) - T^{i}(P_{[k+1]}) \right] + (T_{[k+1]} - T_{2}) - [d_{0}(P_{[k+1]}) - d_{0}(P)]
\]

\[
- \left[ \sum_{i=1}^{k} s^{i}(P_{[k+1]}) - \sum_{i=1}^{k} s^{i}(P) \right] - t^{k+1}(P)
\]

\[
\geq \sum_{i=1}^{k} \left[ (t_{i} + t_{w} N) \log p - \frac{1}{2} p t_{s} - (t_{s} + t_{w} N) \log p \right]
\]

\[
- \left( \frac{N}{p} - 3 \right) N f - \frac{1}{2} p t_{s} - 2(t_{s} + t_{w} N) \log p \geq 0,
\]

when \(N \geq N_{0}\).

Here we used the fact that for any two processors \(P\) and \(P^{'}\), and any \(1 \leq k \leq N\), \(\sum_{i=1}^{k} T^{i}(P) - \sum_{i=1}^{k} T^{i}(P^{'}) \leq 2N f\).

We also used the fact that \(\sum_{i=1}^{k} s^{i}(P_{[k+1]}) - \sum_{i=1}^{k} s^{i}(P) = [k/p]((1/2)p t_{s} - (1/2)p t_{s}) + \sum_{i=1}^{[k]} s^{i}(P_{[k+1]}) - \sum_{i=1}^{[k]} s^{i}(P) \leq (1/2)p t_{s}\).

**APPENDIX C**

**PROOF OF THEOREM 8**

**PROOF.** Let

\[
\Delta T^{k}_{c}(P_{ij}) = \left( T^{k-1}_{\text{complete}}(P_{ij}) + s^{k}_{c}(P_{ij}) + s^{k}_{s}(P_{ij}) \right) - \left( T^{k}_{\text{send}}(P_{ij}) + t^{k}_{c}(P_{ij}) + s^{k}_{s}(P_{ij}) \right),
\]

\[
\Delta T^{k}_{r}(P_{ij}) = \left( T^{k-1}_{\text{complete}}(P_{ij}) + s^{k}_{c}(P_{ij}) + s^{k}_{s}(P_{ij}) \right) - \left( T^{k}_{\text{send}}(P_{ij}) + t^{k}_{c}(P_{ij}) + s^{k}_{s}(P_{ij}) \right),
\]

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

**BASE CASE.** \(k = 2\). First, consider the case \(i \neq 2\) and \(j \neq 2\).

\[
\Delta T^{2}_{c}(P_{ij}) = \left( T^{1}_{\text{complete}}(P_{ij}) + s^{2}_{c}(P_{ij}) + s^{2}_{s}(P_{ij}) \right) - \left( T^{2}_{\text{send}}(P_{ij}) + t^{2}_{c}(P_{ij}) + s^{2}_{s}(P_{ij}) \right).
\]

If \(T^{2}_{\text{send}}(P_{ij}) - T^{2}_{\text{send}}(P_{22}) + t^{2}_{c}(P_{22}) + s^{2}_{s}(P_{22}) + T^{'}\), then

\[
\Delta T^{2}_{r}(P_{ij}) = \left( T^{1}_{\text{start}}(P_{ij}) + t^{2}_{c}(P_{ij}) + s^{2}_{s}(P_{ij}) \right) - \left( T^{2}_{\text{send}}(P_{22}) + t^{2}_{c}(P_{22}) + s^{2}_{s}(P_{22}) + T^{'} + t^{2}_{r}(P_{ij}) \right).
\]

\[
\geq \left( \frac{N^{2}}{p} - 2N \frac{f}{\sqrt{p}} \right) f - (N^{2} f + (t_{s} + T' N \sqrt{p}) \log_{2} p) - (t_{s} + t_{w} N \sqrt{p}) \log_{2} p
\]

\[
= \left( \frac{N^{2}}{p} - 3N \frac{f}{\sqrt{p}} \right) f - 2(t_{s} + t_{w} N \sqrt{p}) \log_{2} p \geq 0.
\]
If $T_{\text{send-}r}^2(P_{2j}) = T_{\text{start}}^1(P_{2j}) + 2T'$, then

$$
\Delta T_{r}^2(P_{ij}) = (T_{\text{start}}^1(P_{ij}) + T^1(P_{ij}) + s_r^1(P_{ij})) - (T_{\text{start}}^1(P_{2j}) + 2T' + t_r^2(P_{ij}))
$$

$$
= (T^1(P_{ij}) - 2T') + s_r^1(P_{ij}) - (T_{\text{start}}^1(P_{2j}) - T_{\text{start}}^1(P_{ij})) - t_r^2(P_{ij})
$$

$$
\geq \left( \frac{N^2}{p} - \frac{2N}{\sqrt{p}} \right) f - \left( \frac{N}{\sqrt{p}} f + \left( t_s + t_w \frac{N}{\sqrt{p}} \right) \log_2 p \right) - \frac{1}{2} \left( t_s + t_w \frac{N}{\sqrt{p}} \right) \log_2 p
$$

$$
= \left( \frac{N^2}{p} - \frac{3N}{\sqrt{p}} \right) f - \frac{3}{2} \left( t_s + t_w \frac{N}{\sqrt{p}} \right) \log_2 p \geq 0.
$$

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

**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 $T_{\text{send-}r}^{k+1} = T_{\text{send-}c}^{k+1}(P_{[k+1][k+1]} + t_c^{k+1}(P_{[k+1]j})) + s_c^{k+1}(P_{[k+1]j}) + T'$. The proofs for the other cases are similar.

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

$$
\Delta T_{r}^{k+1}(P_{ij}) = (T_{\text{complete}}^k(P_{ij}) + s_c^{k+1}(P_{ij}) + s_r^{k+1}(P_{ij})) - (T_{\text{send-}r}^{k+1}(P_{ij}) + s_r^{k+1}(P_{ij}))
$$

$$
= (T_{\text{start}}^1(P_{ij}) + T^{k}(P_{ij}) + s_c^{k+1}(P_{ij})) - (T_{\text{send-}c}^{k+1}(P_{[k+1][k+1]}) + t_c^{k+1}(P_{[k+1]j}))
$$

$$
+ s_c^{k+1}(P_{[k+1]j}) + T' + t_r^{k+1}(P_{ij})
$$

$$
= \left( T_{\text{start}}^1(P_{ij}) + \sum_{l=1}^{k} T^{l}(P_{ij}) + \sum_{l=1}^{k} (s_c^l(P_{ij}) + s_r^l(P_{ij})) + s_r^{k+1}(P_{ij}) \right)
$$

$$
- \left( T_{\text{start}}^{k}(P_{[k+1][k+1]}) + \sum_{l=1}^{k} T^{l}(P_{[k+1][k+1]}) - T^{k}(P_{[k+1][k+1]}) + T' \right)
$$

$$
- \left( \sum_{l=1}^{k} (s_c^l(P_{[k+1][k+1]}) + s_r^l(P_{[k+1][k+1]})) + t_c^{k+1}(P_{[k+1]j}) + s_c^{k+1}(P_{[k+1]j}) \right)
$$

$$
+ s_c^{k+1}(P_{[k+1]j}) + T' + t_r^{k+1}(P_{ij})
$$

$$
= T^{k}(P_{[k+1][k+1]}) + \sum_{l=1}^{k} (T^{l}(P_{ij}) - T^{l}(P_{[k+1][k+1]})) - 2T'
$$

$$
- (T_{\text{start}}^1(P_{[k+1][k+1]}) - T_{\text{start}}^1(P_{ij}))
$$

$$
- \sum_{l=1}^{k} (s_c^l(P_{[k+1][k+1]}) + s_r^l(P_{[k+1][k+1]}) - (s_c^l(P_{ij}) + s_r^l(P_{ij}))
$$

$$
- (t_c^{k+1}(P_{[k+1]j}) + s_c^{k+1}(P_{[k+1]j})) - t_r^{k+1}(P_{ij}) + s_r^{k+1}(P_{ij})
$$

$$
\geq \left( \frac{N^2}{p} - \frac{2N}{\sqrt{p}} \right) f - \left( \frac{N}{\sqrt{p}} f + \left( t_s + t_w \frac{N}{\sqrt{p}} \right) \log_2 p \right)
$$

$$
- 2\sqrt{p}t_s - \left( t_s + t_w \frac{N}{\sqrt{p}} \right) \log_2 p
$$

$$
= \left( \frac{N^2}{p} - \frac{3N}{\sqrt{p}} \right) f - 2\sqrt{p}t_s - 2 \left( t_s + t_w \frac{N}{\sqrt{p}} \right) \log_2 p \geq 0.
$$

Similarly, we can show that $\Delta T_{r}^{k+1}(P_{ij}) \geq 0$, $\forall i, j$. □
PROOF OF LEMMA 6

PROOF. Without loss of 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

$$\sum_{k=1}^{p} T^k(P) = \sum_{k=1}^{i-2} T^k(P) + T^{i-1}(P) + \sum_{k=i}^{p} T^k(P)$$

$$= \sum_{k=1}^{i-2} n(N-k)f + (n+1)[N-(i-1)]f + \sum_{k=i}^{p} (n-1)(N-k)f$$

$$= \sum_{k=1}^{p} (n-1)(N-k)f + [N-(i-1)]f + \sum_{k=1}^{i-1} (N-k)f.$$

$$\sum_{k=p+1}^{2p} T^k(P) = \sum_{k=p+1}^{p+(p-i)-1} T^k(P) + T^{p+(p-i)}(P) + \sum_{k=p+(p-i)+1}^{2p} T^k(P)$$

$$= \sum_{k=p+1}^{p+(p-i)-1} (n-1)(N-k)f + n[N-(2p-i)]f + \sum_{k=p+(p-i)+1}^{2p} (n-2)(N-k)f$$

$$= \sum_{k=p+1}^{2p} (n-2)(N-k)f + [N-(2p-i)]f + \sum_{k=p+1}^{p+(p-i)} (N-k)f.$$

Thus,

$$\sum_{k=1}^{2p} T^k(P) = \sum_{k=1}^{p} (n-1)(N-k)f + \sum_{k=p+1}^{2p} (n-2)(N-k)f + (2N+1-2p)f$$

$$+ \sum_{k=1}^{i-1} (N-k)f + \sum_{k=p+1}^{p+(p-i)} (N-k)f.$$

If $i = 1$, then

$$\sum_{k=1}^{2p} T^k(P) = \sum_{k=1}^{p} (n-1)(N-k)f + \sum_{k=p+1}^{2p} (n-1)(N-k)f + n[N-(2p-1)]f + (n-1)[N-(2p)]f$$

$$= \sum_{k=1}^{p} (n-1)(N-k)f + \sum_{k=p+1}^{2p} (n-2)(N-k)f + (2N+1-4p)f + \sum_{k=p+1}^{p+(p-i)} (N-k)f.$$

Therefore, we have

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

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

$$\left| \sum_{k=2pr}^{2p(r+1)} T^k(P) - \sum_{k=2pr}^{2p(r+1)} T^k(P') \right| \leq 2p^2f.$$
Therefore,
\[
\left| \sum_{k=1}^{K} T^k(P) - \sum_{k=1}^{K} T^k(P') \right| \leq 2p^2 f \left( \frac{K}{2p} \right) + \sum_{k=[K/2p]+1}^{K} \left| T^k(P) - T^k(P') \right| < 2p^2 \frac{K}{2p} + 2pfN \leq 3Nf.
\]

REFERENCES

7. A. Grama, A. Gupta and V. Kumar, Isoefficiency measuring the scalability of parallel algorithms and architectures, IEEE Parallel and Distributed Technology 1 (3), 12–21, (August 1993).