On the Impact of Communication Latencies on Distributed Sparse LU Factorization

R. Wunderling, H. Ch. Hege and M. Grammel

Konrad Zuse-Zentrum für Informationstechnik Berlin (ZIB)

Heilbronner Str. 10

E-Mail: {wunderling,hege,grammel}@zib-berlin.de

Abstract

Sparse LU factorization offers some potential for parallelism, but at a level of very fine granularity. However, most current distributed memory MIMD architectures have too high communication latencies for exploiting all parallelism available. To cope with this, latencies must be avoided by coarsening the granularity and by message fusion. However, both techniques limit the concurrency, thereby reducing the scalability. In this paper, an implementation of a parallel LU decomposition algorithm for linear programming bases is presented for distributed memory parallel computers with noticable communication latencies. Several design decisions due to latencies, including data distribution and load balancing techniques, are discussed. An approximate performance model is set up for the algorithm, which allows to quantify the impact of latencies on its performance. Finally, experimental results for an Intel iPSC/860 parallel computer are reported and discussed.

1 Introduction

We consider the solution of linear systems of equations

$$Ax = b, \tag{1}$$

where A is an unsymmetric, nonsingular sparse $n \times n$ matrix with no particular nonzero pattern. In this paper we deal with matrices coming from linear programming bases. Matrices that are routinely solved today have $1 - 20$ nonzeros per row or column, possibly with a few exceptions, and their dimension range up to $10^4 - 10^6$ for current problems.

In simplex based linear programming methods, equation (**??**) is usually solved through LU factorization [**?**]. This requires five steps:

1. Factorization: Find permutations π, ρ and lower and upper triangular matrices L and U respectively, such that
$$A_{\pi_i \rho_j} = (LU)_{ij} \quad \forall (i,j), 1 \leq i,j \leq n. \tag{2}$$
2. Permute vector b to $d_i = b_{\pi_i} \quad \forall i, 1 \leq i \leq n$.
3. Forward substitution: Solve $Ly = d$.
4. Backward substitution: Solve $Uz = y$.
5. Permute vector z to the solution of (**??**) $x_{\rho_j} = z_j \quad \forall i, 1 \leq i \leq n$.

In this paper, only the factorization step is considered. Our target architecture is a local memory MIMD parallel computer with significant communication latencies.

Parallel LU factorization algorithms for unsymmetric matrices have been developed and implemented for shared memory architectures [**?**]. More recently, van der Stappen, Bisseling and van de Vorst presented a distributed algorithm for a mesh network of transputers [**?**]. This architecture has communication latencies that may easily be hidden through asynchronous message passing (see section **??**). The amount of communicated data for various data distribution schemes, has been investigated in [**?**]. To the knowledge of the authors, the impact of communication latencies has not been studied in detail yet. However, for many current architectures, communication latencies are in the range of $10^3 - 10^4$ floating point operations and have a significant impact on the performance. The aim of this work was to develop, implement and analyze a factorization algorithm suitable for such parallel architectures.

The outline of this paper is as follows. Section **??** describes the basic factorization algorithm and how the issues of numerical *stability* and *sparsity* are handled in the pivot selection algorithm. From this, a parallel algorithm is derived (section **??**). The amount of available parallelism is shown and our implementation is described with special emphasis on the design decisions due to communication latencies. Using an approximate *performance model* the impact of latencies on our algorithm is discussed in section **??**. In section **??** experimental results are reported and discussed for an Intel iPSC/860, leading to some conclusions in section **??**.

2 The Basic LU Decomposition Algorithm

The general factorization algorithm is given in figure **??**. It consists of a sequence of $n-1$ rank-1-updates of A, referred to as *pivot elimination* steps:

$$A = A^1 \to A^2 \to \ldots \to A^n.$$

In each elimination step a nonzero matrix element, called the *pivot element*, is selected. Starting with the index sets $I^1 = \{1, \ldots, n\} = J^1$ the factorization algorithm recursively defines sets I^s and J^s by removing the row and column index of the pivot element, respectively. The matrix $A^s_{I^s J^s}$, consisting of rows I^s and columns J^s of A^s, is called the *active submatrix* at stage s. Pivot elements are selected from the active submatrix and permuted to its upper left corner. Then, the L-loop and update-loop are processed.

```
Pivot-loop:            For $s := 1$ to $n-1$:
    Pivot selection:   Select pivot element, i.e., choose
                       $i^s \in I^s, j^s \in J^s$, such that $A^s_{i^s j^s} \neq 0$.
    Permutation:       Set $\pi_{i^s} := \rho_{j^s} := s$.
    L-loop:            For $i \in I^s$:
                           Set $L'_{ij_s} := A^s_{ij^s} / A^s_{i^s j^s}$.
                       Set $I^{s+1} := I^s \setminus \{i^s\}$, $J^{s+1} := J^s \setminus \{j^s\}$.
    Update-loop:       For $i \in I^{s+1}$, $j \in J^{s+1}$:
                           Set $A^{s+1}_{ij} := A^s_{ij} - L'_{ij^s} \cdot A^s_{i^s j}$.
Termination:           $L_{ij} := L'_{\pi_i \rho_j}$, $U_{ij} := A^n_{\pi_i \rho_j}$.
```

Figure 1: *A generic LU factorization algorithm.*

In order to obtain accurate solutions from the LU factors, numerical stability must be maintained during the factorization. One approach is *threshold pivoting* [?]. Matrix elements A^s_{ij} are accepted as pivot elements only, if they satisfy the *threshold condition*

$$|A^s_{ij}| \geq u \cdot \max_{l \in I^s} |A^s_{lj}| \quad , \tag{3}$$

where $0 \leq u \leq 1$ is an adjustable threshold parameter. For linear programming bases a good default value for u is 0.01. Usually this is increased to 0.1

or min($2u$,1) when instability is detected [?]. In the sequel, matrix elements satisfying equation ?? are referred to as *eligible*.

Sparsity is exploited by restricting the L-loop and update-loop in figure ?? to indices with nonzero matrix elements. This can be done efficiently, if appropriate storage schemes are employed for the sparse matrix [?]. However, new nonzero matrix elements may be created in the update-loop. Such elements are referred to as *fill-ins*. In order to further take advantage of sparsity fill-ins must be avoided during the factorization.

It is well known, that the pivot selection strategy has a dramatical impact on the amount of fill and hence the performance of the factorization [?]. Since the problem of finding a sequence of pivot elements with minimum fill is $\mathcal{NP}$-complete [?], a number of heuristics have been developed [?]. In state of the art simplex based LP solvers the *Markowitz pivot selection* scheme proved to be very efficient [?, ?]. Pivot elements with low *Markowitz number*

$$M^s_{ij} = (r^s_i - 1)(c^s_j - 1) \tag{4}$$

are chosen, where r^s_i and c^s_j denote the number of nonzeros in row i and column j of the active submatrix $A^s_{I^sJ^s}$, respectively. It can easily be seen, that M^s_{ij} provides an upper bound to the fill occurring in the next elimination step, when choosing A^s_{ij} as pivot element.

Following [?] our pivot selection algorithm relies on a combination of both, threshold pivoting for stability and Markowitz pivot selection for fill reduction (cf. figure ??). For fast access to rows and columns with few nonzeros, we keep $2n$ double linked lists labeled $R_1, \ldots, R_n$ and $C_1, \ldots, C_n$. Each row index i and column index j of the active submatrix is stored in the list $R_{r^s_i}$ and $C_{c^s_j}$, respectively.

Rows or columns with only one matrix element are referred to as *singletons*. If singletons exist, they are taken as pivot elements right away. Otherwise, eligible matrix elements with low Markowitz number are searched. Since this is expensive, the search is restricted to p columns or rows with minimal number of nonzeros, where p is a program parameter. If during this search an eligible element with minimal Markowitz number is found, the search is stopped and the element selected as pivot element.

Since inequality (??) is a rowwise condition, some columns may contain no eligible element. Hence, after checking p columns no eligible element may be found. If this happens, the search is continued until an eligible element is found. Columns with no eligible element are marked to not search them again for eligible elements.

For checking the threshold condition (??) the largest absolute value in a row must be computed. We save this value in an array `max`. If row i is modified during the factorization, `max[i]` is set to -1 to indicate, that this value is out of date.

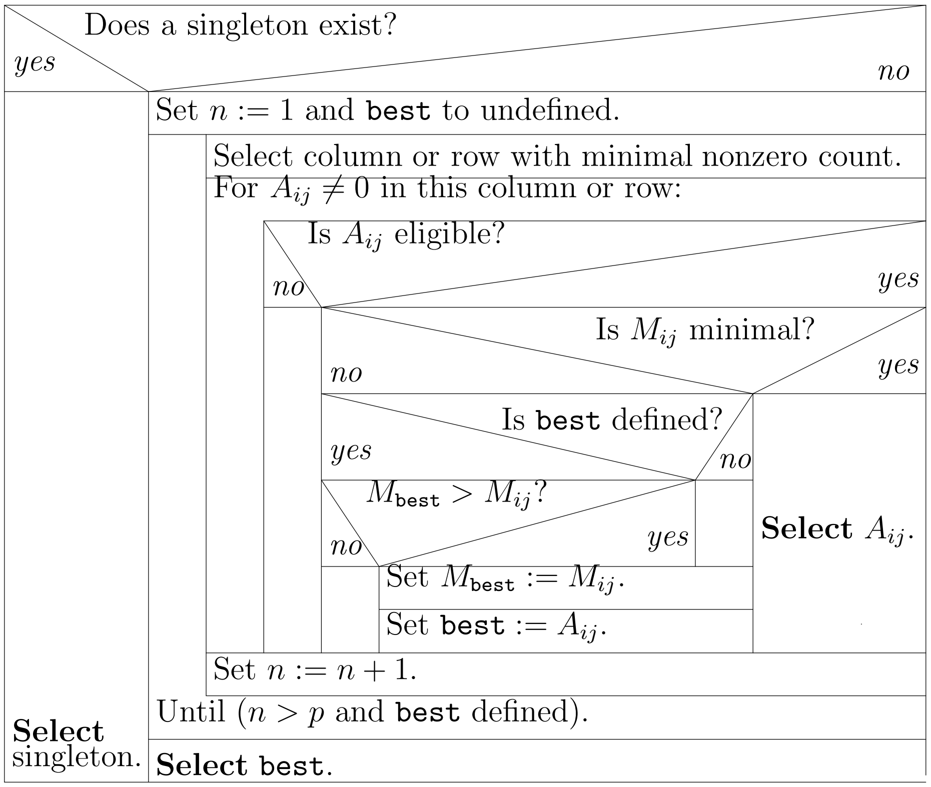

Does a singleton exist?
yes
no
Set $n := 1$ and `best` to undefined.
Select column or row with minimal nonzero count.
For $A_{ij} \neq 0$ in this column or row:
Is A_{ij} eligible?
no
yes
Is M_{ij} minimal?
no
yes
Is `best` defined?
yes
no
$M_{\texttt{best}} > M_{ij}$?
no
yes
Set $M_{\texttt{best}} := M_{ij}$.
Set `best` $:= A_{ij}$.
Select A_{ij}.
Set $n := n + 1$.
Until ($n > p$ and `best` defined).
Select singleton.
Select `best`.

Figure 2: *Selection of pivot elements using threshold pivoting and Markowitz numbers.*

3 Parallel Algorithm

In this section our parallel algorithm is presented. First, the concurrency generally offered by the LU factorization is shown. In the second subsection, the distribution of data is discussed as the first decision, where communication latencies are considered. Subsection 3 sketches the four phases of our parallel algorithm, that are described in detail in the following subsections.

3.1 Parallelization Opportunities

Figure **??** reveals, that both inner loops, the L-loop and the update-loop, may be parallelized. Further, the loops in the pivot selection algorithm (figure **??**) may be parallelized, as well as the determination of values `max[`j`]`, needed for checking the threshold condition (**??**).

In general, the instances of the pivot-loop cannot be executed concurrently, since active submatrices depend on previous iterations. However, for sparse matrices this is not strictly true. It is indeed possible to perform several pivot steps in parallel, if *compatible* pivot elements are selected [**?**]. Nonzero elements

$A^s_{i_1 j_1} \ldots A^s_{i_m j_m}$ are called *compatible* or *independent*, if they satisfy

$$A_{i_k j_l} = 0, \quad \forall 1 \leq k \neq l \leq m, \tag{5}$$

i.e., if $A^s_{i_1 \ldots i_m, j_1 \ldots j_m}$ forms a diagonal submatrix of $A^s_{I^s J^s}$. However, determination of compatible pivot elements requires additional computation and — for the distributed case — communication.

There are two general approaches to exploit this parallelism due to sparsity. One operates on the symmetrized nonzero pattern of the matrix $A + A^T$ or $A^T A$ [?, ?], whereas the other approach works directly on the unsymmetric matrix. The first approach allows to take advantage of the techniques developed for symmetric sparse matrices such as elimination trees [?, ?, ?]. However, as shown in [?], this may reduce the amount of parallelism for matrices with unsymmetric nonzero pattern, since $A + A^T$ and $A^T A$ may be much denser than A itself. Therefore, we decided to work directly on the unsymmetric matrix.

3.2 Distribution of Data

For MIMD multiprocessors with distributed memory, distribution of data is an issue of major importance for the amount of exploitable parallelism. Generally, there are two approaches to this problem. One is, to determine a distribution at runtime by analyzing the sparsity pattern and distributing matrix rows or columns to achieve a maximum of parallelism with local memory access [?]. The other approach is a fixed assignment of rows, columns or both to processors [?, ?]. We chose the latter approach, because it can do without additional communication for data redistribution.

For maximum exploitation of parallelism, it has been suggested to distribute both, rows and columns, using a *grid distribution* [?]. Such a distribution also yields low communication complexity for LU decomposition of dense matrices. However, it requires "many" communications in one parallel pivot elimination step, leading to an accumulation of communication latencies. Indeed, a tentative implementation using the grid distribution, yielded a decrease in performance even for only 4 processors of an Intel iPSC/860.

This shows how important it is for this kind of problem to design algorithms that minimize the impact of latencies. This can be achieved by *avoiding latencies* as well as *latency hiding*. Our algorithm uses both. Latencies are hidden by using assynchronous communication. Latencies are avoided by reducing the number of messages per pivot elimination step. This is done with *message fusion* and *coarsening* the granularity of the algorithm. However, both reduce the amount of concurrency, yielding poor scaling properties.

To achieve coarser granularity, a rowwise or columnwise distribution can be used instead of a grid distribution. We adopted a random assignment of matrix rows to processing elements, that remains fixed during the factorization. This distribution allows concurrency in both, the L-loop and the update-loop. However, it does not allow for parallelism within the rowwise part of the update-loop. Further, the threshold condition may be checked locally on demand, i.e., without requiring communication.

For load balancing, the rows should be evenly distributed among the processors, although the algorithm does not rely on that. We shall discuss the issue of load balancing in section **??**.

3.3 Outline of the Algorithm

Our algorithm is divided into four phases (cf. figure **??**). Matrices occurring in linear programming typically contain many singletons. Elimination of singletons does not require the same amount of computation as general pivot elements. Hence, it is advisable to handle them with a separate algorithm, that can do without all the communication required for the general case.

For row singletons $A^s_{i^s j} = 0$, $\forall j \in J^{s+1}$, only the L-loop must be computed, while the update-loop has length 0. Note, that row singletons of any number are always compatible and can, hence, be eliminated in parallel. This is done in phase I of our parallel algorithm and will be described in section **??**.

Phase II (see section **??**) handles the elimination of column singletons, that occur in LP basis matrices as well. In this case neither the L-loop nor the update-loop require computation. Again, column singletons are always compatible and may be processed in parallel.

Section **??** describes phase III, the actual parallel sparse LU factorization, based on the concept of compatible pivot elements. Iteratively, a set of compatible pivot elements with "small" Markowitz numbers is generated and eliminated asynchronously, until the entire matrix is factorized or becomes "too dense".

Finally, phase IV is a semi-dense parallel factorization and will be described in **??**. In this phase, only one pivot element at a time is eliminated.

3.4 Elimination of Row Singletons

The rowwise distribution of the matrix allows every processor to detect row singletons locally. Since row singletons are always compatible, all row singletons available at one step may be eliminated in a single parallel iteration.

The algorithm for the elimination of row singletons is shown in figure **??** for one processor out of P. First, all row singletons locally available are selected and the (local) variable l is set to the number of row singletons. These row singletons are broadcast to the other processors in only one message (message fusion). Then the local part of the L-loop is done for these pivot elements.

In the loop over n, the row singletons selected by the other $P-1$ processors are handled asynchronously in the order the corresponding messages come in. The number of row singletons received in a message is stored in l'. Then, the

```
Phase I    While (there are row singletons):
               Eliminate row singletons.
Phase II   While (there are column singletons):
               Eliminate column singletons.
Phase III  While (active submatrix is “sparse”):
               Select set of compatible pivot elements
               and eliminate them.
Phase IV   While (matrix is not factorized):
               Select one pivot element
               and eliminate it.
```

Figure 3: *The four phases of the parallel factorization algorithm.*

local part of the L-loop is processed for the l' pivot elements in this message and l is incremented by l'. Hence, after the loop over n, l contains the total number of row singletons that have been (concurrently) selected in this iteration. This is iterated until $l < 1$, i.e., there are no more row singletons in the matrix. Experiments showed, that it has no significant effect on the overall performance to stop the outer loop, as soon as only “few” row singletons where found, i.e., $l_p <$“few”.

```
    Select all $l$ (local) row singletons.
    Broadcast $l$ pivot elements.
    For $i := 1$ to $l$:
        Do local part of L-loop for singleton $i$.
    For $p := 2$ to $P$:
        Receive $l'$ row singletons.
        For $i' := 1$ to $l'$:
            Do local part of L-loop for singleton $i'$.
        Set $l := l + l'$.
Until ($l < 1$).
```

Figure 4: *Parallel elimination of row singletons (for one processor out of P).*

Note, that it is correct to perform the loop over i at any time after the selection of local row singletons. We decided to perform it right after broadcasting the pivot elements in order to overlap communication and computation for hiding latencies.

3.5 Elimination of Column Singletons

Since every column is distributed over all processors, determination of column singletons may not happen locally. Instead, all processors must cooperate in computing the nonzero numbers c_j for all columns j. The structure of this task is often referred to as *gossiping*: Information that is distributed among all processors, must be merged with some operation and the result made accessible to all processors [?].

Collective Information Exchange. The gossiping algorithm with fewest communications depends on the topology of the underlying communication network. It can easily be seen, that the diameter of the network graph is a lower bound on the number of communication steps [?]. For a hypercube of dimension d, the algorithm given in figure **??** takes d (bidirectional) communications and is, hence, optimal in terms of number of messages.

```
For $i := 1$ to $d$:
    Send data to processor $p_i$.
    Receive data from processor $p_i$.
    Merge received data.
```

Figure 5: *"Gossiping" for processor p in a hypercube of dimension d.*

In figure **??**, p_i denotes the processor with the same coordinates as processor p, except for coordinate i, where $i = 1, \ldots, d$. The algorithm can be viewed as of P fan-in algorithms running in parallel, such that at the end every processor has the merged information. Note, that the merging operation is performed in parallel.

Elimination of Column Singletons. With the gossiping algorithm from figure **??**, we are now able to describe the algorithm of phase II for eliminating column singletons. It is sketched in figure **??**. At the beginning, the number of nonzeros per column, in the sequel refered to as *column counts*, are computed with a gossiping algorithm. Then, all column singletons are eliminated without further communication. For some columns the nonzero counts will have changed and must be updated. To do this, every processor stores the local changes of the column counts. These changes are distributed with a gossip, such that every processor updates the column counts correctly.

```
Gossip: Compute column nonzero counts.
    For all column singletons $j$:
        Eliminate pivot column $j$.
        Is pivot element on this processor?
        no |  yes
           |  Eliminate pivot row.
           |  Store column count changes.
    Gossip: Update column counts.
Until (no column singletons found).
```

Figure 6: *Parallel elimination of column singletons.*

3.6 Factorization of the Nucleus

The parallel factorization of the nucleus relies on the concept of compatible pivot elements and is outlined in figure **??**. In every (parallel) iteration, a set of e compatible pivot elements is selected and the corresponding pivot rows are broadcast to all processors. Then, every processor computes the L-loop and update-loop asynchronously in the order it receives the pivot rows. This is iterated until the active submatrix becomes "too dense" or the matrix is completely factorized.

```
    Select set of $e$ compatible pivot elements.
    Broadcast local pivot rows.
    For all pivots $i$:
        Do local part of L-loop for pivot $i$.
        Do local part of update-loop for pivot $i$.
    Gossip: Update column counts.
    Set $s := s + e$.
Until ($density >$ MAX_DENSITY or $s \geq n$).
```

Figure 7: *Outline of parallel sparse factorization of the nucleus.*

For setting up the performance model in section **??**, we determine the time complexities for the various steps in one parallel iteration. Broadcasting the pivot rows is $O(z_e e)$, where z_e denotes the average number of nonzeros in the e chosen pivot rows or columns. The L-loop for one pivot element requires $O(z_e)$ and the update-loop $O(z_e^2)$ time. Updating the column counts is $O(z_e e)$. Adding $O(1)$ for communication latencies and code that must be processed independent of the pivot element or candidate number yields a total time complexity of $O(z_e e + z_e^2 e +$

1) for all steps of one iteration, except the selection of pivot elements, which will be discussed in the following subsection.

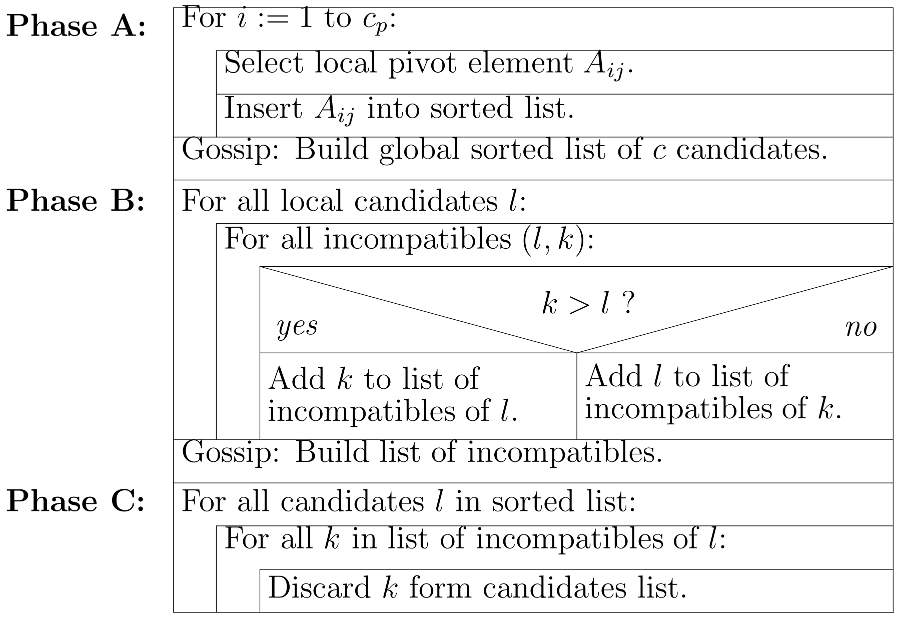
3.6.1 Parallel Selection of Compatible Pivot Elements

Three goals must be pursued by the selection of pivot elements:

1. Numerical stability (by means of threshold pivoting).
2. Fill reduction (by means of low Markowitz numbers).
3. Large set of compatibles for maximal exploitable parallelism.

Fortunately, objective 2 favours 3, since matrix elements with low Markowitz numbers have few nonzeros in their row and column and are, hence, likely to be compatible to many other matrix elements. Therefore, matrix elements with low Markowitz number should be good candidates for compatible pivot elements.

This is the idea of our algorithm for selecting compatible pivot elements, which is sketched in figure **??**: It is divided into three phases, denoted A, B and C. In phase A a sorted list of pivot candidates is set up. Phase B detects all incompatibilities, while in phase C, incompatible pivot elements are discarded in a *greedy* fashion.



```
Phase A:  For $i := 1$ to $c_p$:
              Select local pivot element $A_{ij}$.
              Insert $A_{ij}$ into sorted list.
          Gossip: Build global sorted list of $c$ candidates.
Phase B:  For all local candidates $l$:
              For all incompatibles $(l, k)$:
                  $k > l$ ?
                  yes: Add $k$ to list of incompatibles of $l$.
                  no:  Add $l$ to list of incompatibles of $k$.
          Gossip: Build list of incompatibles.
Phase C:  For all candidates $l$ in sorted list:
              For all $k$ in list of incompatibles of $l$:
                  Discard $k$ form candidates list.
```

Figure 8: *Parallel selection of compatible pivot elements.*

Phase A: Selection of Pivot Candidates. In the first phase of the algorithm, a list of pivot candidates is generated and sorted according to increasing Markowitz numbers, in order to give preference to pivot candidates with low Markowitz number in phase C. More specifically, we use the following sorting criterion, which avoids ties:

1. Minimal Markowitz number,
2. Minimal column number,
3. Minimal processor number.

Initially, every processor p selects and sorts c_p candidates from its local part of the active submatrix. The selection algorithm for a single local pivot candidate is displayed in figure **??** using $p = 1$. The local lists are then merged to a global sorted list with a gossiping algorithm as described in figure **??**.

The average time complexity for locally selecting one pivot candidate is $O(z_c)$, where z_c denotes the average number of nonzeros in the candidate rows or columns (cf. figure **??**). Inserting a candidate into the list, requires $O(c)$ time. Adding $O(c+1)$ for the gossip and the latencies involved yields a total of $O(c^2+z_cc+c+1)$ for phase A.

Phase B: Detection of Incompatibilities. In phase B every processor detects all pairs of incompatible pivot candidates, with one candidate being local to the processor itself. When two incompatible candidates l and k, $l > k$, are found, l is added to the list of incompatibles of candidate k, where "$>$" is defined by the sorting criterion for pivot candidates. This allows for fast execution of phase C. Finally, all local incompatible lists are merged to a global list on every processor using the gossiping algorithm.

Every candidate is tested against every other candidate. Hence, including the gossiping, the time complexity of phase B is $O(c^2 + 1)$.

Phase C: Selection of Compatible Elements. At the beginning of phase C, every processor has exactly the same sorted list of pivot candidates and the entire set of incompatible candidate pairs. Hence, applying the same selection algorithm on every processor will yield the same set of compatible pivot elements.

The algorithm should extract a "good" subset of compatible pivot elements, where "good" means, "many pivot elements with low Markowitz numbers". This may be modeled as a *stable set problem*. It is not neccessary to really solve this stable set problem to optimality, since setting up the candidate set was done heuristically in the first place. We chose the greedy heuristic, which gives preference to pivot candidates with small Markowitz number. Starting with the first candidate in the sorted list, all elements, incompatible to this, are discarded. Then, the next (not yet discarded) candidate is selected and its incompatibles are discarded. This is iterated until no more candidates are in the list. The time complexity of the greedy algorithm is $O(c)$. From our experiments we expect the solutions from the greedy algorithm to be very close to optimality.

Discussion. This pivot selection scheme yields a different sequence of pivot elements than the sequential code. In sequential code, the pivot selection always relies on the actual Markowitz numbers. When selecting several pivot elements at a time, the selection of all but the first element relies on Markowitz numbers from some eliminations before. However, since the selected pivot elements are compatible, their Markowitz numbers remain unchanged during the iteration.

On the other hand, searching more pivot candidates at a time, yields a larger part of the matrix to be inspected. Hence, chances are good, that elements are found with lower Markowitz number, than in the sequential case. Pivot elements with high Markowitz numbers tend to be discarded by the greedy algorithm. Which of the two effects dominates, depends on the number of pivot candidates and on the matrix. We will see in section **??**, that even for one processor, best performance is achieved when selecting more than one pivot element at a time. This might be exploited for further improvements of sequential algorithms.

Finally, we sum up all time complexities for one parallel factorization iteration in figure **??**. This yields the time

$$T(c) = \alpha c^2 + \beta_1 c + \beta_2 z_c c + \gamma_1 z_e e + \gamma_2 z_e^2 e + \lambda, \tag{6}$$

for c candidates, where the parameters α, β_1, β_2, γ_1, γ_2 and λ depend on the target machine. Note, that the number of pivot elements $e = e(c)$ depends on the number of candidates c and the matrix to be factorized.

3.6.2 Load Balancing

Load balancing is an important issue for efficient execution of parallel algorithms. In general, there are two types of load balancing [**?**]. Static load balancing may be used, when there is enough a priori information to design a distributed algorithm, that keeps all processors equally loaded. Otherwise, load must be balanced dynamically, relying on information that arises at runtime. This requires a *measure* for processor load (and its imbalance) and a *method* for balancing the load.

In sparse LU decomposition static load balancing is not applicable: Pivot elements might always be selected from the same processor, such that this processor will have its part of the matrix almost eliminated, while other processors still have a part of the matrix of the initial size. Hence, dynamic load balancing should be applied.

A cheap estimate for the load of processor p at a given iteration s is its number of rows or nonzeros in the active submatrix. A better estimate would have to consider the sparsity pattern, since the amount of work to be done in one iteration significantly depends on the nonzero distribution of the pivot columns. For simplicity, we chose the number of rows r_p^s as load estimate, since these numbers need not be computed.

One may distinguish "active" and "passive" methods for balancing the load. The active technique would send matrix rows from loaded processors to unloaded ones, at the beginning of each iteration. This increases the communication volume. Instead, we adopted a passive approach. Every processor p selects a number of pivot candidates c_p proportional to its number of local rows r_p^s

$$c_p = \lfloor \frac{r_p^s}{n^s} c \rfloor, \tag{7}$$

where $n^s = n - s$ denotes the dimension of the active submatrix at stage s. This increases the probability of finding pivot elements located on processing elements with high row number r_p^s. Therefore, the number of rows will be reduced more on such processors than on others, leading to a better balance in the next iteration.

Altough, this balancing strategy remains suboptimal in terms of the load estimate, this needs not be a problem, if the load estimate is inaccurate already and the imbalance is small enough (cf. section **??**). Further, this approach does not introduce any additional overhead, neither in computation nor in communication. This makes it favorable in our context.

3.7 Semidense Factorization

As the active submatrix becomes denser due to fill-ins and elimination of the sparse parts of the matrix, less compatible pivot elements will be found. Hence, at some point it is advisable to select one pivot element at a time, for saving most of the communication required for determining compatible pivot elements. We found a density of 80% to be a reasonable value for switching.

Our semidense factorization algorithm is given in figure **??**. It requires only one broadcast per iteration, whereas phase III uses four gossips per parallel iteration. We refer to it as semidense, since we use algorithm **??** for pivot selection. Note, that we do not update the column nonzero numbers to avoid the latencies involved with gossiping. Hence, the nonzero counts get more and more out of date, but empirically, it turned out to have negligible impact on the amount of fill.

Load balancing is easily achieved in this phase, since only one processor selects a pivot element per iteration. We simply have the processors with the most rows of the active submatrix select and broadcast a pivot row, where ties are broken by the processor number. By that, after a startup period, the number of rows per processor differ no more than by one.

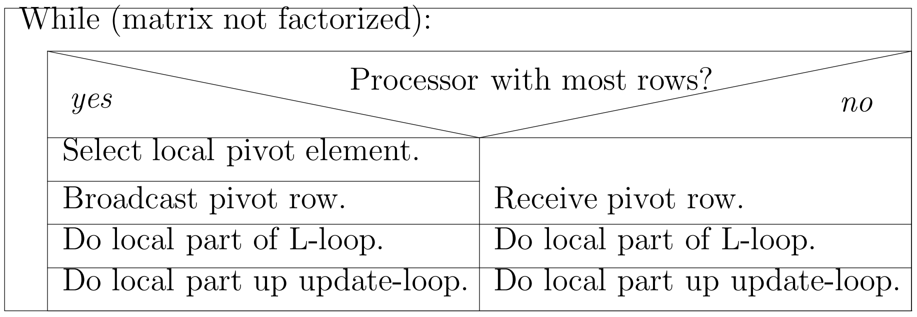

While (matrix not factorized):
Processor with most rows?
yes
no
Select local pivot element.
Broadcast pivot row.
Receive pivot row.
Do local part of L-loop.
Do local part of L-loop.
Do local part up update-loop.
Do local part up update-loop.

Figure 9: *Semidense factorization.*

4 The Impact of Latencies and the Number of Pivot Candidates

Phase III of the factorization algorithm offers a tuning parameter, namely the number of candidates c. Clearly, selecting more candidates will probably yield more compatible pivot elements for parallel elimination. On the other hand, more incompatible candidates are likely to be discarded as well, i.e., work will be done without direct gain for the factorization. Hence, the aim should be to find the optimal tradeoff between these two effects by selecting the appropriate number of candidates. This could even be done dynamically during the factorization.

Let $e(c)$ denote the average number of compatible pivot elements, when selecting c candidates, and $T(c)$ the time required for one parallel iteration. Then, we define the *factorization speed function* for one parallel iteration as

$$S(c) = \frac{e(c)}{T(c)}. \tag{8}$$

Optimal performance is achieved when selecting c such as to maximize the speed function, which is determined by the zeros of the derivative

$$0 = \frac{d\,S(c)}{d\,c} = \frac{e'(c)T(c) - T'(c)e(c)}{T^2(c)}. \tag{9}$$

For solving this equation, both functions $e(c)$ and $T(c)$ are required. The latter is given by equation (**??**). For the numbers of nonzeros per row z_c and z_e a first approximation is

$$z_c = z_e = \frac{z^s}{n^s}, \tag{10}$$

i.e., the average number of nonzeros per row or column for the active submatrix at elimination step s. Due to the pivot selection algorithm given in figure **??**, one expects both, z_c and z_e, to have smaler values, though.

Inserting the time function $T(c) = \alpha c^2 + \beta c + \gamma e(c) + \lambda$, with $\beta = \beta_1 + \frac{z^s}{n^s}\beta_2$ and $\gamma = \frac{z^s}{n^s}\gamma_1 + \left(\frac{z^s}{n^s}\right)^2 \gamma_2$, into equation (**??**) yields an optimal candidate number c as a zero of the function

$$\Phi(c) = e'(c)(\alpha c^2 + \beta c + \lambda) - e(c)(2\alpha c + \beta). \tag{11}$$

Since $\Phi(c)$ does not depend on the parameters γ_1 and γ_2, the optimal candidate number does not, as well. Note, that this is a consequence of approximation (**??**) and does not apply to the general case, where the optimal number of candidates depends on the matrix through β and $e(c)$.

The function $e(c)$ will generally depend on the input matrix. In the sequel, we will discuss its first order approximation. Clearly, there is no constant term in the expansion.

4.1 Linear approximation of $e(c)$

Consider the linear approximation for the function $e(c)$

$$e(c) = \epsilon c. \tag{12}$$

We will see in section **??**, that this approximation is quite accurate for a large range of candidate numbers c. However, for $c \approx n$ this will no longer hold. The factor ϵ may be interpreted as yield of compatible pivot elements.

Factorization Time. The time required for eliminating a single pivot element in the linear approximation (**??**) is

$$S^{-1}(c) = \frac{\alpha}{\epsilon}c + \frac{\lambda}{\epsilon c} + \frac{\beta}{\epsilon} + \gamma = \frac{\alpha}{\epsilon}c + \frac{\lambda}{e(c)} + \frac{\beta}{\epsilon} + \gamma. \tag{13}$$

This should be proportional to the total factorization time. Equation (**??**) consists of a proportional, an antiproportional and constant terms.

The parameter λ describes constant time occurring in one parallel iteration. For more than one processor, this time is dominated by communication latencies and so is the antiproportional term (see section **??**). Hence, equation (**??**) quantifies, that the problem of latencies may, indeed, be reduced by eliminating multiple pivot elements concurrently. This is only true, if the latencies are independent on the number of pivot elements, which we achieve through message fusion, by broadcasting all pivot rows with only one message per processor.

The proportional term has a factor $\frac{\alpha}{\epsilon}$. Since α is related to compatibility determination, it describes the penalty for selecting multiple pivot candidates. This penalty increases with decreasing yield ϵ of compatible pivot elements.

There are two constant terms. γ is related to the L-loop and update-loop, i.e., the computation that we actually wanted to be done. This applies to β as well, which is related to candidate selection. Since some candidates will be discarded due to incompatibility, $\frac{1}{\epsilon}$ times more candidates must be selected than in the sequential case.

Optimal Candidate Number. Inserting equation (**??**) to (**??**) yields the optimal candidate number

$$c = \sqrt{\frac{\lambda}{\alpha}}. \tag{14}$$

This does not depend on the input matrix. Instead, c is solely determined by hardware parameters. The parameter α is due to the sorting of pivot candidates and the determination of incompatibles. It is not related to any communication and, hence, reflects the computation performance of the processors. The parameter λ describes constant time in one iteration, which is dominated by communication latencies for more than one processor. Hence, in this approximation, the optimal number of candidates is determined by the granularity of the parallel computer.

Dependence on Latencies. Inserting the optimal candidate number (**??**) into the factorization time function (**??**) quantifies the impact of latencies on the factorization performance:

$$S^{-1}\left(\sqrt{\frac{\lambda}{\alpha}}\right) = \frac{2}{\epsilon}\sqrt{\alpha\lambda} + \frac{\beta}{\epsilon} + \gamma. \tag{15}$$

This states, that $\frac{\beta}{\epsilon}+\gamma$ is a lower bound for the factorization time of our algorithm, even when run on a parallel processor with no latencies. Note, that latencies $\lambda = 0$ are possible, since latencies may be hidden through asynchronous communication.

5 Experimental Results

5.1 Target Architecture: Intel iPSC/860

For testing the algorithm, we had an Intel iPSC/860 hypercube to our disposal. We measured 85.5μs for sending one double precision floating point value (8 bytes) from one processor to its neighbour for an unloaded interconection network and 120.5μs for 12 values. This corresponds to a communication bandwidth of 2.51 Bytes/μs and a latency of 82.5μs. Due to different protocols for "short" and "long" messages the latencies for message sizes of more than 100 bytes are increased to 179.5μs, whereas the bandwidth remains unchanged.

These values must be viewed in relation to the computing performance of an i860 processor. We measured 0.5μs for one iteration of the following loop:

```
for(i = 0; i < MAX_ITERATION; ++i)
    result[i] = factor1[i] * factor2[i] ;
```

Hence, the processor is able to do 165 double precision floating point operations in such a loop during the latency for "short" messages and 359 for "long" messages. As a comparision, 1.9μs for one double precision floating point operation and 4.8μs for the communicating latency are reported for a T800 transputer mesh network [?, ?].

5.2 Test Matrices

For our test, we used a set of matrices from the optimal simplex bases of various linear programming problems. Some of the problems come from the Netlib test set[1]. Table **??** gives some statistics on a subset of five test matrices, varying in size n and nonzero density $\rho = \frac{z}{n^2}$, where z denotes the number of nonzeros. The last column contains the factorization time for the sequential code run on a single processors of an Intel iPSC/860.

Nr	n	z	ρ	t_{seq}
1	4248	54239	0.0030	2.66
2	2349	32655	0.0059	2.28
3	16675	51147	0.00018	1.27
4	2243	10641	0.0021	0.73
5	1503	17530	0.0078	0.50
$\sum$				7.44

Table 1: Statistics and sequential factorization time for five test matrices.

5.3 Load Balancing

Figure **??** shows the impact of the passive load balancing scheme when factorizing test matrix 1 on 8 processors. Both, the maximum and minimum number of rows per processor are displayed as function of the dimension d of the active submatrix.

Entering phase 3 at $d \approx 3700$, there is a slight imbalance due to the distribution of singletons. While without load balancing, this imbalance increases up to the point, where one processor runs out of rows (at about $d = 500$), passive load balancing keeps the imbalance within acceptable bounds.

5.4 Factorization Times

Figure **??** displays the sum of the factorization times for the five test matrices as a function of the number of candidates for several numbers of processors. Only

[1]The Netlib LP list has been compiled by David Gay and is available through anonymous ftp at research.att.com (192.20.225.2).

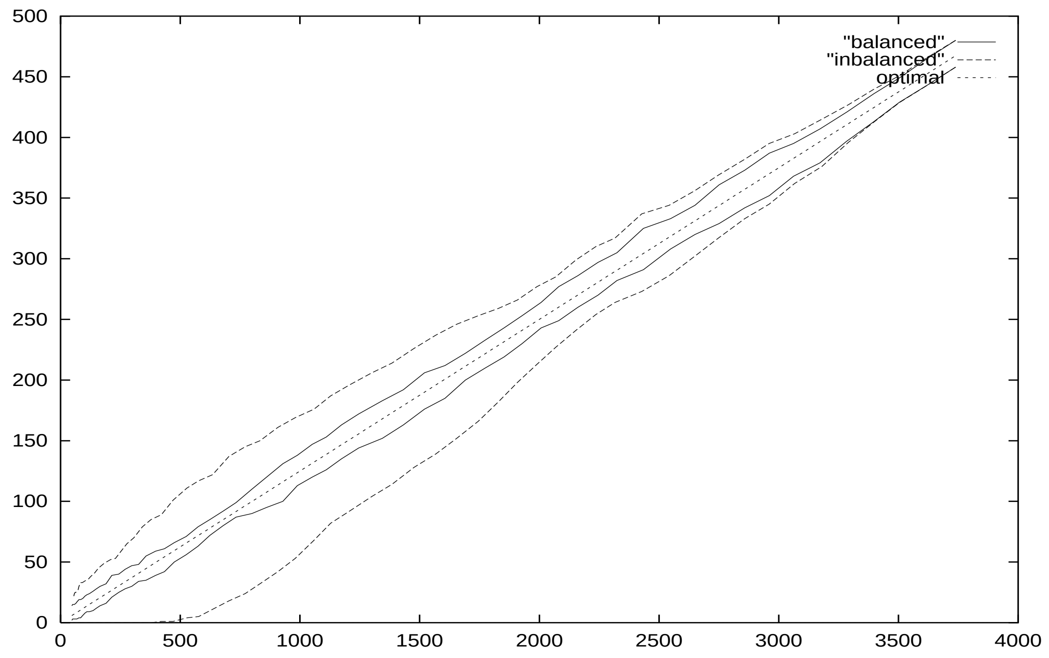


Figure 10: *Maximum and minumum number of rows per processor with and without passive load balancing compared to optimal balance (for 8 processors).*

a speedup of 1.6 is achieved with 16 processors in comparison to the parallel code run on one processor. Comparing to the sequential code, the speedup is only ≈1.3. The different times for the sequential code and the parallel algorithm when run on one processor, are mainly due to message buffers the latter handles.

The plots are very well described by equation (**??**). This can be seen table **??**, which gives the parameters of equation (**??**) fitted to the measured times. Column Δ showes the average error of the fitted curve.

P	α/ϵ	λ/ϵ	$\beta/\epsilon+\gamma$	Δ	c_{opt}^{theo}	c_{opt}^{exp}
1	0.0166	3.35	8.93	0.18	14	16
2	0.0083	19.74	6.56	0.18	49	64
4	0.0044	37.67	5.56	0.29	93	96
8	0.0024	56.36	5.22	0.33	153	192
16	0.0015	90.56	5.16	0.19	246	224

Table 2: Fitted values of the parameters of equation (12).

From the fitted values of parameters $\frac{\alpha}{\epsilon}$ and $\frac{\lambda}{\epsilon}$ the optimal candidate number can be computed using equation (**??**). The resulting values are given in column c_{opt}^{theo}. They compare very well to the experimental data in the last column. Note, that factorization times have only been measured for candidate numbers 8, 16

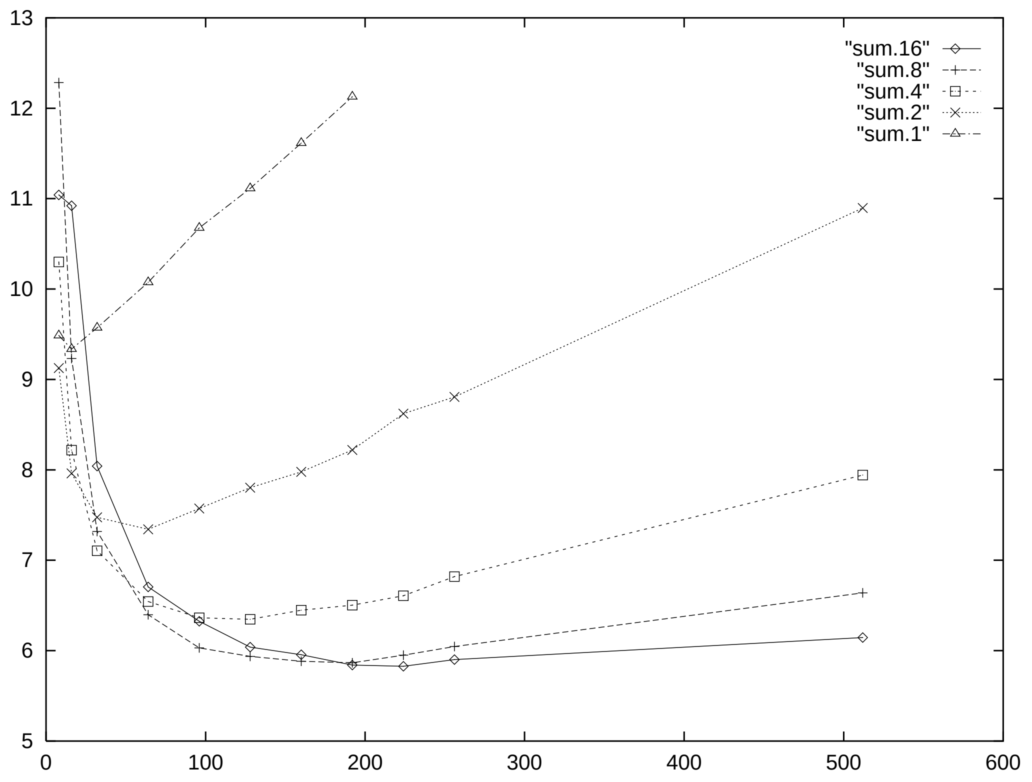


Figure 11: *Factorization time as a function of the number of candidates displayed for different numbers of processors (1, 2, 4, 8 and 16).*

and multiples of 32.

The decrease of parameter α indicates, that the related computation, i.e., the determination of compatibility, parallelizes very well. Parameter $\frac{\lambda}{\epsilon}$ shows a logarithmic increase with the number of processors P. This logarithmic behaviour corresponds to the communication structure of gossiping and the broadcasting. Doubling the number of processors yields an increase of more than 16. These numbers show, how essential it is to avoid communication whenever possible.

The last parameter $\frac{\beta}{\epsilon} + \gamma$ plays the most important role, since it is a lower bound for the computation time of the algorithm, when run on a parallel computer with no communication latencies. Hence, on such an architecture, the algorithm would yield a speedup of 1.73 for 16 processors, or 1.44 compared to the sequential algorithm. It should be noted, though, that linear programming bases generally tend to be comparably bad suited for parallel processing [**?**].

6 Conclusion

Current parallel computer architectures have a ratio of communication latency to time per floating point operation that is typically magnitudes larger than one.

We designed, implemented and analyzed a distributed LU factorization algorithm for unsymmetric sparse matrices. It achieves some speedup even for such parallel architectures and sparse matrices from linear programming.

In the algorithm design, communication latencies are taken into account by *hiding latency* through asynchronous communication, and *avoiding latencies* by reducing the *number* of required communications. The latter is achieved through

- exploition of parallelism of coarser granularity,
- the design of algorithms that require as few communications as possible and
- a thorough implementation using *message fusion*.

Our algorithm uses a *passive* load balancing technique, that achieves an approximate load balance at almost no expense. This method controls local program parameters, i.e., the number of local pivot candidates to be selected, according to the actual processor load in order to achieve a better balance for the *next iteration*. This technique may usefully be applied to other parallel alorithms as well.

The algorithm has been tested on an Intel iPSC/860. It achieves speedups, even for linear programming bases. However, the efficiency for such matrices is rather low, because of the reduced exploitation of parallelism, that is necessary to cope with the communication latencies.

Further, we developed a quantitative performance model for the algorithm. It shows, that the consideration of latencies in the program design, leads to an algorithm with poor scaling properties, even when run on a parallel computer with low latencies.

Generally, we showed that avoiding latencies may indeed help in parallelizing algorithms for parallel computers with rather high latency/computation ratio. However, for the particular case of parallel sparse LU decomposition we showed the limits of this technique, as it reduces the amount of concurrency in the program. This is another example for the importance of low communication latencies and strong latency hiding techniques for future parallel architectures.

It remains an open question up to which latency/computation ratio an algorithm, that exploits more parallelism, runs more efficiently than ours. We plan to investigate this using a lower latency architecture.

7 Acknowledgements

The authors are grateful to Prof. Martin Grötschel for his helpful comments and to Prof. Robert E. Bixby for providing test matrices. Further, the authors would like to thank the “Zentralinstitut für Angewandte Mathematik” of the research center Jülich for providing computation time on the Intel iPSC/860 and

Dipl. Inf. Renate Knecht for her assistence in using it. This research is supported in parts by Cray Research, Inc.

References

[1] R. H. Bisseling and L. D. J. C. Loyens, Towards peak parallel linpack performance on 400 transputers, *SUPERComputer* 45 (1991) 20–27.

[2] ,. E. Bixby, private communications, 1993.

[3] T. A. Davis and P. Yew, A nondeterministic parallel algorithm for gerneral unsymmetric sparse LU factorization, *SIAM J. Matrix Anal. Appl.* 11 (1990) 383–402.

[4] I. S. Duff, Parallel implementation of multifrontal schemes, *Parallel Computing* 3 (1986) 193–204.

[5] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, (Clarendon Press, 1986).

[6] I. S. Duff and J. K. Reid, The mulitfrontal solution of unsymmetric sets of linear equations, *SIAM J. Sci. Stat. Comput.* 5 (1984) 633–641.

[7] A. George, M. T. Heath, J. Liu, and E. NG, Sparse cholesky factorization on a local-memory multiprocessor, *SIAM J. Sci. Stat. Comput.* 9 (1988) 327–340.

[8] A. Goscinski, *Distributed Operating Systems: The Logical Design*, (Addison-Wesley, 1991).

[9] M. Heath, E. NG, and B. Peyton, Parallel algorithms for sparse linear systems, *SIAM Review* 33 (1991) 420–460.

[10] D. W. Krumme, G. Cybenko, and K. N. Venkataraman, Gossiping in minimal time, *SIAM J. Comput.* 21 (1992) 111–139.

[11] P. Sadayappan and S. K. Rao, Communication reduction for distributed sparse matrix factorization on a processor mesh, In *Supercomputing '89* (1989) 371–379.

[12] U. H. Suhl and L. M. Suhl, Computing sparse LU factorizations for large-scale linear programming bases, *ORSA Journal on Computing* 2 (1990) 325–335.

[13] A. F. v. d. Stappen, R. H. Bisseling, and J. G. G. v. d. Vorst, Parallel sparse LU decomposition on a mesh network of transputers, *SIAM J. Matrix Anal. Appl* 14 (1993) 853–879.

[14] M. Yannakakis, Computing the minimum fill-in is NP-complete, *SIAM J. Algebraic Discrete Methods* 1 (1981) 77–79.