On Mapping Data and Computation for Parallel Sparse Cholesky Factorization

Kalluri Eswar C.-H. Huang P. Sadayappan

Department of Computer and Information Science

The Ohio State University

Columbus Ohio 43210

Abstract

When performing the Cholesky factorization of a sparse matrix on a distributed-memory multiprocessor, the methods used for mapping the elements of the matrix and the operations constituting the factorization to the processors can have a significant impact on the communication overhead incurred. This paper explores how two techniques, one used when mapping dense Cholesky factorization and the other used when mapping sparse Cholesky factorization, can be integrated to achieve a communication-efficient parallel sparse Cholesky factorization. Two localizing techniques to further reduce the communication overhead are also described. The mapping strategies proposed here, as well as other previously proposed strategies fit into the unifying framework developed in this paper. Communication statistics for sample sparse matrices are included.

1 Introduction

The Cholesky factorization of a large unstructured symmetric sparse matrix is an important computation in several scientific and engineering applications. The Cholesky factor of a symmetric matrix A is a lower triangular matrix L such that $A = LL^T$. The factorization is usually performed in-place, *i.e.*, the elements of L overwrite those of A. For a sparse matrix, the extra nonzero positions present in L but not in A are determined in advance, and explicit zero-valued entries are introduced in A at those positions.

```
for k := 1 to n do
    L_kk := sqrt(L_kk)
    (* Normalize column k *)
    for i > k such that L_ik ≠ 0 do
        L_ik := L_ik / L_kk
    end
    (* For each column j updated by column k *)
    for j > k such that L_jk ≠ 0 do
        (* Update column j using column k *)
        for i ≥ j such that L_ik ≠ 0 do
            L_ij := L_ij − L_jk * L_ik
        end
    end
end
```

Figure 1: Sparse Cholesky factorization

Figure 1 shows one form of the sequential sparse Cholesky factorization algorithm. The outer loop index k represents a column of the matrix, which is *normalized* and then used to *update* other matrix columns (indexed by j). There are four kinds of operations that constitute the factorization computation:

- *Diagonal formation*: These are the square root operations ($\sqrt{L_{kk}}$) used to determine the final values of diagonal elements.

- *Normalization*: These are the division operations (L_{ik}/L_{kk}) used to scale a matrix element by the diagonal element in its column, producing the final value for the element.

- *Update formation*: These are the multiplication operations ($L_{jk} * L_{ik}$) between two elements in the same column (the *source* column) to form an update contribution to an element (L_{ij}) in another column (the *target* column). The element L_{jk} is called the *factor* element, the element L_{ik} is called the *source* element, and the element L_{ij} is called the *target* element.

- *Update application*: These are the subtraction operations used to apply an update contribution to a target element. In some cases, several update contributions for the same target element may be added together before the combined contribution is subtracted from the target element. Such addition operations are also considered to be update application operations. NOTE: In some implementations, the negative of an update contribution ($-L_{jk} * L_{ik}$) may be formed and *added* to the target element. Similarly, a combined contribution may be formed using subtraction operations, and then added to the target element.

The update formation and update application operations usually form the dominant component of the total set of operations required for the factorization computation.

When a distributed-memory multiprocessor is to be used for carrying out the sparse Cholesky factorization, there are three important issues to be considered:

- Mapping of the data, *i.e.*, the matrix elements, to processors.

- Mapping of the computation, *i.e.*, the operations listed above, to processors.

- Ordering of the operations within each processor.

This paper focuses on the two mapping issues since the goal here is to study methods for reducing the communication overhead incurred, as reflected by static measures such as the total number of messages and the total message volume. The ordering of operations on each processor can affect the dynamic load balance of the parallel factorization process. It can also affect the local efficiency of each processor in executing its operations because of variations in the use of the memory hierarchy components, usually a data cache and primary memory.

The natural approach to the mapping problems is to first specify a mapping of the data, and then to use some rule to derive the mapping of the computation based on the data mapping. All the mapping strategies considered in this paper and all the methods known to the authors use this approach.

For dense Cholesky factorization, *index-based* mappings of data (also called *torus wrap* mappings), based on two- and three-dimensional logical views of the processor system, have been shown to lower the communication overhead significantly. Section 2 reviews the index-based approach in a systematic manner. Ground work for a unifying mapping framework is laid by identifying the different components of the communication overhead incurred during the parallel factorization.

For sparse Cholesky factorization, *nested* mappings of data result in much lower communication overhead than nonnested mappings. These mappings localize data to smaller subsets of processors, thereby localizing the communication. Section 3 reviews the nested mapping approach for parallel sparse Cholesky factorization. It is shown that the effect of using this approach is different for different kinds of communication overhead.

Section 4 looks at how the ideas of using index-based mappings and using nested mappings can be integrated to derive a mapping strategy for parallel sparse Cholesky factorization with good asymptotic communication complexity. This section provides some experimental results that demonstrate that certain components of the communication overhead are reduced more by the integrated approach than certain other components.

A technique for localizing data further, thus lowering the communication overhead, is presented in Section 5. Two variants of this approach corresponding to two different rules for the computation mapping are considered. Section 6 describes a mapping strategy that can be derived from the strategies of Section 5 by modifying the computation mapping rule. This strategy attempts to localize the computation of update contributions from a column to a small subset of the processors. A mapping strategy based on a parallel multifrontal algorithm is shown to be a variant of this strategy. This provides a clean integration of the multifrontal strategy along with the others into a unifying framework.

2 Index-based mapping for dense factorization

When performing dense Cholesky factorization in parallel, it is natural to use each matrix element's row and/or column indices for determining the processor it is mapped to. In this section, such index-based approaches are classified based on the number of dimensions in the logical view taken of the processor system. The communication overhead incurred in each case is classified into different components. It will be seen that the number of different communication components is related to the number of dimensions in the logical view of the processor system.

2.1 One-dimensional logical view

A one-dimensional logical view of the processor system does not mean that the processors are assumed to be arranged in the topology of a ring or a chain. It is simply the normal view in which the identity of each processor is specified using just one number. If there are P processors, they may be numbered $0, 1, \ldots, P-1$.

2.1.1 Data mapping

A simple index-based scheme for the data mapping when performing parallel dense Cholesky factorization is

$$\mathsf{map}_D(L_{ij}) = j \ \mathbf{mod}\ P.$$

Since $\mathsf{map}_D(L_{ij})$ (the D is for "data") is independent of i, it means that columns of the matrix are being mapped onto the processors. This data mapping function is known as a *column-wrap* mapping. The processor to which a matrix element is mapped is said to *own* the element.

A slightly more complicated scheme for the data mapping is to consider the matrix columns in groups of consecutively numbered columns, which may be called *block-columns*. A block size b is chosen to limit the number of columns in a block-column. The data mapping function is then constructed so that all the columns in a block-column are mapped onto the same processor. For example,

$$\mathsf{map}_D(L_{ij}) = (j \ \mathbf{div}\ b) \ \mathbf{mod}\ P.$$

2.1.2 Computation mapping

The diagonal formation operations (see Section 1) are always mapped onto the processor to which the diagonal element (L_{kk}) is mapped by the data mapping function. Similarly, the normalization operations are always mapped onto the processor to which the element being normalized (L_{ik}) is mapped by the data mapping function. In the rest of this paper, all discussion regarding the computation mapping will therefore be restricted to the update formation and the update application operations.

One possible rule for specifying the mapping of the update operations is a *target computes* rule. This means that the update formation and update application operations are mapped onto the processor owning the target element. This may be expressed as

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = \mathsf{map}_C(application(L_{jk} * L_{ik})) = \mathsf{map}_D(L_{ij}).$$

The C in map_C is for "computation".

Consider the communication requirements of a parallel factorization with the data and computation mappings specified as above. Since all the elements in a matrix column will be mapped onto a single processor, no communication is required for performing the normalization operations. The factor element (L_{jk}) and the source element (L_{ik}) used in forming an update contribution are both in the same matrix column, and will hence be located on the same processor. The factorization may be organized at a column level as follows. After normalizing the elements of a column that a processor owns, the entire column (minus the diagonal element) could be sent to any processor owning a target of this column. Once a normalized column is received by a processor, it can be used to form and apply update contributions onto all the appropriate target elements owned by that processor. Since each factor element L_{jk} is anyway needed as a source element when updating L_{jj}, the only kind of communication needed is what will be called *source communication*. The one dimension in the logical view of the processor system corresponds to this one component of the communication. The mapping scheme just considered will be labeled the **S** scheme, where **S** stands for source.

Another possible rule for specifying map_C is a *source computes* rule. This means that the update formation operations are mapped onto the processor owning the source element. Formally,

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = \mathsf{map}_D(L_{ik}).$$

The update application operations for a target element are distributed between the processors forming the update contributions and the processor owning the target element as follows. Each processor combines (*i.e.*, adds) all update contributions that it has formed for the same target element. The combined contribution is then sent to the processor owning the target element, where it is applied to the target element. Combined contributions are also called *aggregates*. All combined contributions destined for target elements belonging to the same column will be sent together, so that aggregate columns are communicated.

Since the factor element to be multiplied with a source element will be present on the same processor, the only kind of communication needed with this mapping scheme is what will be called *aggregate communication*. The one dimension in the logical view of the processor system corresponds to this one component of the communication. The mapping scheme just considered will be labeled the **A** scheme, where **A** stands for aggregate.

2.2 Two-dimensional logical view

In the **S** scheme, each matrix column might have to be sent to every other processor. In the **A** scheme, each matrix column might have to receive an aggregate column from every other processor. Therefore, if n is the number of rows or columns in the matrix, the number of words that might have to be communicated in both the **S** and the **A** schemes would be $O(n^2P)$. Three mapping schemes that each have a communication complexity that is $O(n^2\sqrt{P})$ are considered in this section. When P is large, this would be a substantial reduction in the communication overhead incurred.

In all three schemes to be described, the P processors in the processor system are considered to be logically organized as a $P_1 \times P_2$ grid of processors, where $P = P_1P_2$. Each processor is identified by two numbers forming a tuple $<c_1, c_2>$, where c_1 is its coordinate in the P_1 direction ($0 \leq c_1 \leq P_1 - 1$), and c_2 is its coordinate in the P_2 direction ($0 \leq c_2 \leq P_2 - 1$).

The data mapping function, map_D, for the first two schemes is a two-dimensional analog of the index-based mapping used in the one-dimensional view case. It is given by

$$\mathsf{map}_D(L_{ij}) = <i \ \mathbf{mod}\ P_1, j \ \mathbf{mod}\ P_2>.$$

When this function is used, all matrix elements belonging to the same matrix column will be distributed among a processor "column" of P_2 processors. Similarly, all matrix elements belonging to the same matrix row will be distributed among a processor "row" of P_1 processors. As in the one-dimensional case, variations on this function are possible, for example, by specifying two block sizes b_1 and b_2, and using $i \ \mathbf{div}\ b_1$ and $j \ \mathbf{div}\ b_2$ instead of i and j, respectively, in the above formula.

The first of the two schemes sharing the above data mapping function uses a target computes rule for specifying the computation mapping. That is,

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = \mathsf{map}_C(application(L_{jk} * L_{ik})) = \mathsf{map}_D(L_{ij}).$$

Consider the communication requirements of this scheme. $\mathsf{map}_D(L_{kk})$ and $\mathsf{map}_D(L_{ik})$ could differ in their first component. Therefore, L_{kk} would have to be communicated to other processors in the same processor column as its owner so that the normalization operations may be performed. This kind of communication is called *diagonal communication*. Since there are only n diagonal elements, this component of the communication is dominated by the communication required for performing the update operations. Consequently, when discussing the communication requirements of all the mapping schemes considered in this paper, the diagonal communication component, if it exists, will be ignored. $\mathsf{map}_D(L_{ik})$ and $\mathsf{map}_D(L_{ij})$ differ only in their second component. Therefore, the communication of the source element L_{ik} happens along the second dimension (of length P_2) of the logical processor grid. Each matrix element will have to be sent as a source element to a certain number of processors (at most $P_2 - 1$) in the processor row of the processor owning the element. This kind of communication is called source communication, as it was in the one-dimensional case. The number of words in this component of the communication overhead will be $O(n^2 P_2)$. The element L_{jk} is needed as a factor element for certain elements in matrix column j. All these target elements are owned by processors belonging to the same processor column, which could be different from that of the processor owning L_{jk}. Therefore, L_{jk} needs to be communicated as a factor element to at most P_1 processors in that processor column. Since L_{jk} is also needed as a source element for updating L_{jj}, it is anyway going to get to a processor belonging to the processor column containing elements of matrix column j. That processor can then send L_{jk} as a factor element, along the processor column, to the appropriate processors (at most $P_1 - 1$). This kind of communication will be called *factor communication*. The number of words in this component of the communication overhead will be $O(n^2 P_1)$. The overall communication complexity of this scheme will therefore be $O(n^2 \max(P_1, P_2))$, which will be $O(n^2 \sqrt{P})$, if $P_1 = P_2 = \sqrt{P}$.

The two dimensions in the logical processor grid correspond to the two components of the communication overhead. The mapping scheme just considered will be labeled the $\mathbf{F} \times \mathbf{S}$ scheme, where **F** stands for factor, and **S** stands for source. With this correspondence between the dimensions of the processor grid and the communication components established, the processor system may be viewed as a $P_{\mathbf{F}} \times P_{\mathbf{S}}$ grid, instead of as a $P_1 \times P_2$ grid. Figure 2 illustrates the communication requirements of the $\mathbf{F} \times \mathbf{S}$ mapping scheme.

The second mapping scheme considered in this section uses the same data mapping function as the first, but uses a source computes rule for specifying the computation mapping. That is,

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = \mathsf{map}_D(L_{ik}).$$

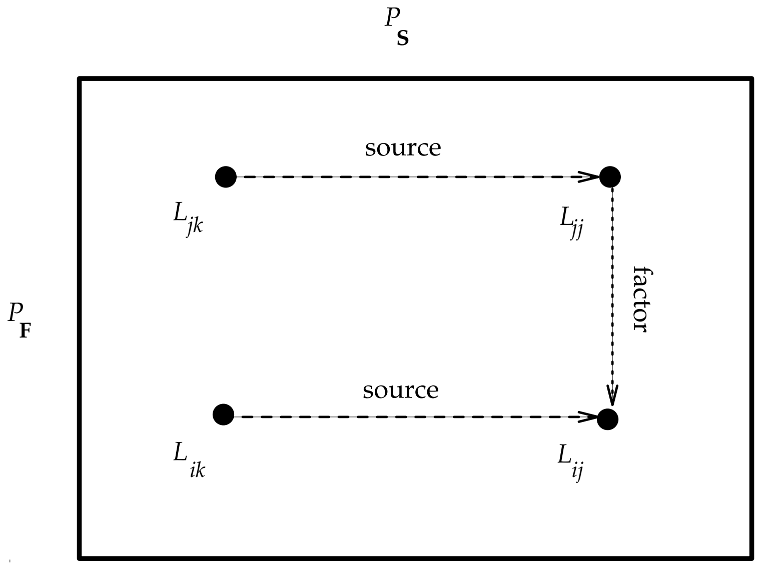

Figure 2: Communication in the $\mathbf{F} \times \mathbf{S}$ mapping scheme

Just as it is with the **A** scheme of Section 2.1.2, the update contributions for the same target element are combined or aggregated at the processors forming the update contributions. The update application operations are therefore distributed between these processors and the processor owning the target element.

Two kinds of communication take place in this scheme. The first is the communication of matrix elements as factor elements to the processors owning the source elements that the factor elements have to be multiplied with. Since the factor element and the source element belong to the same matrix column, the data mapping function ensures that the factor communication occurs only along the first dimension of the logical processor grid. The other component of the communication overhead is the aggregate communication towards target elements. A processor will receive aggregates only from other processors in the same processor row, because the source elements for a target element belong to the same matrix row as the target element. As with the $\mathbf{F} \times \mathbf{S}$ scheme, the communication complexity of this scheme can be seen to be $O(n^2\sqrt{P})$, when $P_1 = P_2 = \sqrt{P}$.

The two dimensions of the logical processor grid correspond to the factor communication and the aggregate communication. This second mapping scheme will therefore be labeled $\mathbf{F} \times \mathbf{A}$, using the same convention as above. The processor system may be viewed as a $P_{\mathbf{F}} \times P_{\mathbf{A}}$ grid in this case. Figure 3 illustrates the communication requirements of the $\mathbf{F} \times \mathbf{A}$ mapping scheme.

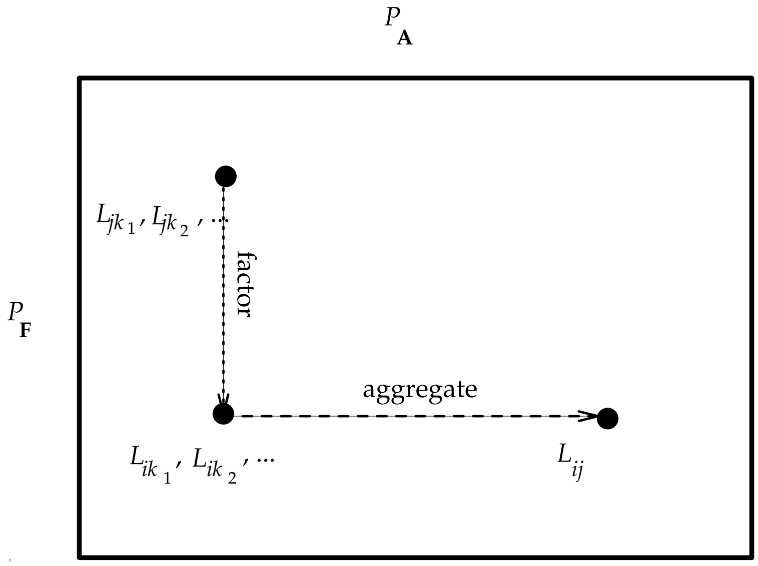

Figure 3: Communication in the $\mathbf{F} \times \mathbf{A}$ mapping scheme

Both the $\mathbf{F} \times \mathbf{S}$ scheme and the $\mathbf{F} \times \mathbf{A}$ scheme can be considered to take a two-dimensional view of the matrix, since elements belonging to the same matrix column may be owned by different processors. The **S** scheme and the **A** scheme of Section 2.1.2 take a one-dimensional view of the matrix, since elements in the same matrix column reside on the same processor. The third scheme to be considered in this section also takes a one-dimensional view of the matrix, but uses a two-dimensional logical view of the processor system.

The data mapping function is an index-based function like the one used for the preceding two mapping schemes. The difference is that a matrix element's row index is not used. A possible data mapping function is

$$\mathsf{map}_D(L_{ij}) = <j \ \mathbf{mod}\ P_1, j \ \mathbf{mod}\ P_2>.$$

As usual, variants are possible, for example, by grouping columns into block-columns.

The rule used for specifying the computation mapping is an example of an *intermediate computes* rule. This means that it is possible that neither the processor owning the source element nor the processor owning the target element might be responsible for the formation of an update contribution from the source element to the target element. The specific rule used for the scheme being considered here may be expressed as

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = <[\mathsf{map}_D(L_{ij})]_1, [\mathsf{map}_D(L_{ik})]_2>.$$

The notation $[\mathsf{x}]_c$ is used to indicate the extraction of the c^{th} component of the tuple x. With the data mapping given above, the computation mapping may also be expressed as

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = <j \ \mathbf{mod}\ P_1, k \ \mathbf{mod}\ P_2>.$$

This computation mapping specifies that the update contribution be formed at the processor that is at the intersection of the processor row containing the target element and the processor column containing the source element. The update contributions formed on a processor for the same target element are combined or aggregated, as is the case with a source computes rule.

Consider the communication overhead incurred with this scheme. Source elements and factor elements need to be communicated from their owning processors to the processors where the update contributions are formed. Since the elements of each matrix column are all owned by the same processor, the factor and the source elements may be sent together. In addition, the update contributions from source elements belonging to the same matrix column for all the target elements belonging to the same matrix column are formed on the same processor. Therefore, a processor owning a matrix column can send the entire column (minus the diagonal element) to other processors belonging to the same processor column. This component of the communication will be called the source communication. These are the only processors that might form update contributions using factor elements and source elements from this matrix column. Since all the target elements belonging to the same matrix column also reside on the same processor, the combined contributions for all the target elements belonging to the same matrix column may be communicated together as an aggregate column. A column will receive aggregates only from other processors belonging to the same processor row. This component of the communication is called the aggregate communication. It can be seen that the number of words in the source communication is $O(n^2P_1)$ and that the number of words in the aggregate communication is $O(n^2P_2)$. As in the $\mathbf{F}\times\mathbf{S}$ scheme and the $\mathbf{F}\times\mathbf{A}$ scheme, if $P_1 = P_2 = \sqrt{P}$, the overall communication complexity is $O(n^2\sqrt{P})$.

The two dimensions of the logical processor grid correspond to the source communication and the aggregate communication. This mapping scheme will therefore be labeled $\mathbf{S}\times\mathbf{A}$, using the same convention as before. The processor system may be viewed as a $P_{\mathbf{S}} \times P_{\mathbf{A}}$ grid in this case. Figure 4 illustrates the communication requirements of the $\mathbf{S}\times\mathbf{A}$ mapping scheme.

2.3 Three-dimensional logical view

A two-dimensional view of the matrix for the data mapping combined with an intermediate computes rule for the computation mapping results in the mapping scheme described in this section. The P processors are viewed logically to form a three-dimensional $P_1 \times P_2 \times P_3$ grid, with $P = P_1P_2P_3$. Each processor is identified by three numbers forming a tuple $< c_1, c_2, c_3 >$, where, for $d = 1, 2, 3$, c_d is its coordinate in the P_d direction ($0 \leq c_d \leq P_d - 1$).

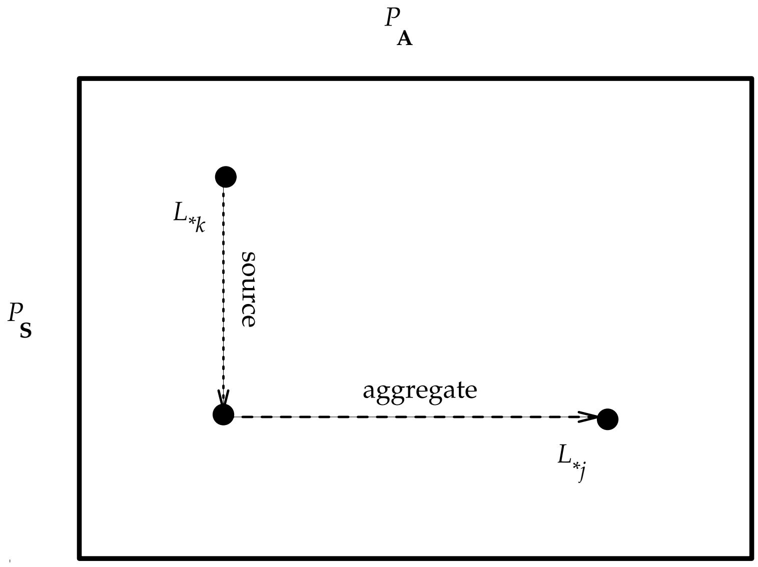

Figure 4: Communication in the $\mathbf{S} \times \mathbf{A}$ mapping scheme

The data mapping function is given by

$$\mathsf{map}_D(L_{ij}) = <i \ \mathbf{mod}\ P_1, j \ \mathbf{mod}\ P_2, j \ \mathbf{mod}\ P_3>.$$

It can be seen that all the matrix elements belonging to the same matrix column will be mapped onto a set of processors forming a line parallel to the first dimension in the logical processor grid. Similarly, all the matrix elements belonging to the same matrix row will be mapped onto a set of processors forming a plane in the logical processor grid perpendicular to the first dimension in the logical processor grid.

The computation mapping is determined using an intermediate computes rule similar to that used for the $\mathbf{S} \times \mathbf{A}$ scheme. It is expressed as

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = <[\mathsf{map}_D(L_{ik})]_1, [\mathsf{map}_D(L_{ij})]_2, [\mathsf{map}_D(L_{ik})]_3>,$$

which can be rewritten using the map_D function given above, so that

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = <i \ \mathbf{mod}\ P_1, j \ \mathbf{mod}\ P_2, k \ \mathbf{mod}\ P_3>.$$

All the update contributions for a particular target element L_{ij} will be formed at a set of processors forming a line in the logical processor grid parallel to the third dimension. Aggregation of these update contributions takes place at the processors forming them.

The communication overhead incurred with this mapping scheme has three components: source communication, aggregate communication, and factor communication. Matrix elements have to be sent as source elements to other processors on the line parallel to the second dimension of the logical processor grid that passes through the owning processor. Matrix elements receive aggregates from other processors on the line parallel to the third dimension of the logical processor grid that passes through the owning processor. Each matrix element L_{jk} has to be sent as a factor element to processors on a line of processors parallel to the first dimension of the logical processor grid. This is because it is needed as a factor element for target elements in matrix column j with the source elements coming from matrix column k. From the data mapping function and the computation mapping function, it can be seen that the processors where the update contributions from elements in matrix column k for elements in matrix column j are formed differ only in their first coordinate. L_{jk} would anyway have to be sent as a source to the processor forming the update contribution $L_{jk} * L_{jk}$ for the target element L_{jj}. This processor can then send the value of L_{jk} to other processors on the line parallel to the first dimension of the logical processor grid passing through it. If $P_1 = P_2 = P_3 = \sqrt[3]{P}$, the number of words in each of the communication overhead components will be $O(n^2 \sqrt[3]{P})$.

The three dimensions of the logical processor grid correspond to the three communication components. This mapping scheme will therefore be labeled $\mathbf{F} \times \mathbf{S} \times \mathbf{A}$, using the same conventions as before. The processor system may be viewed as a $P_{\mathbf{F}} \times P_{\mathbf{S}} \times P_{\mathbf{A}}$ grid in this case. Figure 5 illustrates the communication requirements of the $\mathbf{F} \times \mathbf{S} \times \mathbf{A}$ mapping scheme.

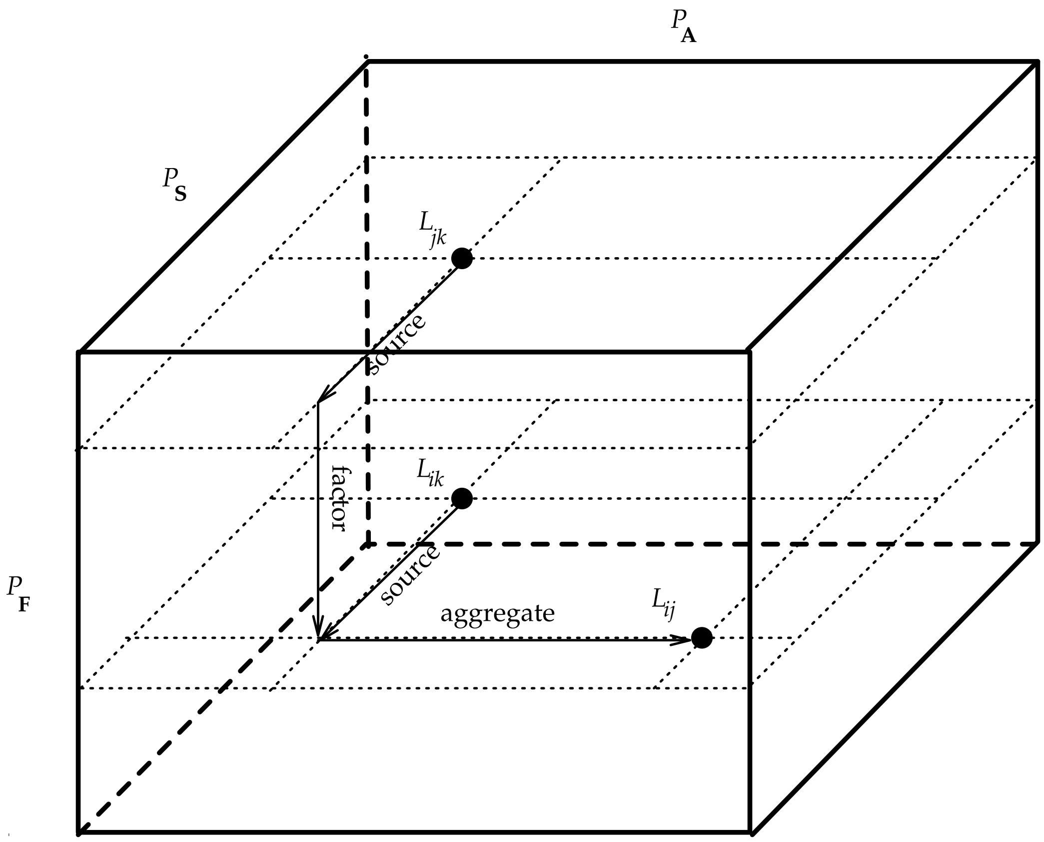

Figure 5: Communication in the $\mathbf{F} \times \mathbf{S} \times \mathbf{A}$ mapping scheme

3 Nested mapping for sparse factorization

During sparse Cholesky factorization, each matrix column updates only a subset of its succeeding column, depending on the nonzero structure of the source column. A consequence of this is that there can be quite a lot of parallelism in the k and j loops in Figure 1. This motivates more effective mapping strategies for sparse Cholesky factorization as compared to those used for dense

Cholesky factorization. *Nested mapping* is an example of such techniques. It is able to lower the communication overhead by reducing the number of processors among which related data are distributed.

The nested mapping approach is significantly superior to an index-based approach to the data mapping, presented in Section 2.1.1. The combination of a column-level data mapping and the target computes rule for the computation mapping results in a parallel factorization scheme called the *column fan-out* algorithm [11]. Similarly, the use of the source computes rule for the computation mapping results in the *column fan-in* algorithm [3]. This section reviews the nested mapping approach for data mapping and compares the effect it has on these two algorithms.

3.1 Nested mapping basics

The column-level dependence structure of a sparse Cholesky factorization can be captured succinctly by a tree called the *elimination tree* [18]. Each matrix column has a node in this tree corresponding to it. The terms column and node will be used interchangeably in the following. The mother of each node is defined by

$$\mathsf{mother}(j) = \min \{\, i \mid i > j \text{ and } L_{ij} \neq 0 \,\},$$

which is just the lowest-numbered column that column j updates. It can be shown that the set of columns updating any column has to be a subset of the set of nodes in the subtree of the elimination tree rooted at that node. This means that computations related to nodes in subtrees of the elimination tree that do not have any common nodes may be performed independently. Nodes in such subtrees could be mapped onto separate processors. Since load balance between the processors is desirable, these subtrees will not extend to levels very far up from the leaves of the elimination tree. Once these nodes have been removed from consideration, the number of independent subtrees becomes smaller. The processors may now be considered to be grouped into small sets to which these subtrees are mapped. The nodes in each subtree will then be distributed in a balanced manner among the processors in the processor group that the subtree is mapped to. This step will remove some more nodes from consideration. This process can be continued, finding independent subtrees at higher and higher levels of the elimination tree and assigning them to larger and larger groups of processors, until, at the top level, a subtree rooted at the root gets assigned to the group of all processors. The higher level processor groups are formed by merging two or more processor groups at a lower level. The assignment of subtrees to processor groups is done so that the processor group that a subtree gets assigned to is a superset of the processor groups that any of its "descendant" subtrees get assigned to.

3.2 Nested mapping for the fan-out and fan-in algorithms

Consider the effect of such a nested mapping approach on the communication overhead incurred by the column fan-out algorithm. In an index-based mapping approach, each column may, in all likelihood, have to be sent as a source column to all (or almost all) the other processors. In the nested mapping approach, the number of processors needing a column as a source would generally depend on the highest level in the elimination tree where a target of the column can be found. The closer this level is to the root, the higher the probability of more processors needing the source. On the average, though, a reduction in the number of processors to which a column needs to be sent as a source can be expected, thus lowering the communication overhead, which in this case is entirely source communication.

In the case of the column fan-in algorithm, the index-based mapping approach would most likely result in the processor owning each column having to receive aggregates from every other processor. With the nested mapping approach, the number of processors from which aggregates have to be received for a target column is limited by the size of the group of processors to which the target column was assigned. This is because all its source columns have to be in the subtree rooted at that node, and the group of processors to which a node's descendant is assigned to is a subset of the group of processors assigned to the node. Since a large number of nodes in the elimination tree will be assigned to groups of processors much smaller than the group of all processors, using the nested mapping approach can, in general, be expected to result in much lower communication overhead, which in this case is entirely aggregate communication. The effect may be expected to be much more substantial than for the column fan-out algorithm. These expectations are borne out by experimental results [8].

3.3 Grouping of processors

The manner in which processor groups are formed may be specified by means of a tree called the *composite processor tree*. The root of this tree is a node associated with the set of all processors. The number of nodes in and the structure of the composite processor tree may be arbitrary. Each node in the composite processor tree is identified by a node number. The only requirement is on the sets of processors associated with the nodes in the composite processor tree. If a node m in the composite processor tree has daughters $d_1, d_2, \ldots$, and if the function procset specifies the processor set associated with each node in the composite processor tree, it is required that

$$\mathsf{procset}(m) = \mathsf{procset}(d_1) \cup \mathsf{procset}(d_2) \cup \ldots,$$

and that

$$\mathsf{procset}(d_1) \cap \mathsf{procset}(d_2) \cap \ldots = \phi.$$

The nested mapping algorithm then needs to specify a mapping of elimination tree nodes to composite processor nodes. The notation $\mathtt{cpmap}(j)$ will be used to denote the composite processor to which matrix column j is mapped. The actual data mapping to an individual may be determined by using an index-based mapping among the processors in the composite processor to which a node is mapped.

3.4 Finding a load balanced nested map

A nested map that is not only communication-reducing but that also results in a load balanced factorization is desirable. This can be derived by associating numerical weights with the nodes in the elimination tree. In general, the weight associated with a node would be related to the number of operations of the factorization involving in some way the column corresponding to the node. It is also desirable to count each operation towards the weight of exactly one node.

For both the fan-out algorithm and the fan-in algorithm, the normalization operations are included in the weights of the node corresponding to the column being normalized. For the fan-out algorithm, the update formation and update application operations are included in the weights of the target column being updated. This is because the computation mapping is specified using a target computes rule. Similarly, for the fan-in algorithm, the update operations are included in the weights of the source column of the updates. Here, however, the choice is not completely obvious. This is because some of the update application operations will be mapped onto the processor owning the target column. However, it is not possible to determine how the update application operations are distributed unless the data mapping is fixed, and that is what the nested mapping algorithm is trying to do! Since the number of update application operations mapped onto the target column's owner is limited by the number of processors, it is expected that the number of update application operations at the source processors would dominate the total. Hence the approach of counting all update operations towards the weights of the source columns.

The nested mapping algorithm uses the weights as follows. A breadth-first traversal of the elimination tree is begun, starting at the root. Suppose that the root of the composite processor tree has t daughters. Nodes visited in this traversal are assigned to the root of the composite processor tree. This process continues until t sets of subtrees are found with total weights that are approximately in the same proportion as the sizes of the t daughter composite processors. When this occurs, the algorithm is recursively applied to each of the t sets of subtrees using the corresponding subtree in the composite processor tree.

The *recursive partitioning* mapping algorithm [7] is a special case of the above nested mapping algorithm in which the composite processor tree is a complete binary tree, and the the group of processors associated with each node is equally split between its two daughters.

This section is concluded with some more comments on the appropriateness of the weight association strategies described above for the fan-out algorithm and the fan-in algorithm. In the case of the fan-out algorithm, the approach of associating the update operations with the target column would be suitable when only static balance, in the form of an even distribution of the operations among the processors, is being sought, it does not account for the fact that the update operations actually happen soon after the source column is normalized. This means that the operations contributing to the weight of a node do not usually happen together over a contiguous period of time. This means that this weight association strategy may not result in as good a dynamic load balance as the static load balance. In the case of the fan-in algorithm, the strategy of associating the update operations with the weights of the source column does not correspond to the dynamics of the factorization, since the original fan-in algorithm ordered the operations on each processor so that the update operations for each target column were performed together. The *kji-agg-sup* algorithm uses a different local operation ordering strategy in which all the update operations that use a source column are performed together. Therefore, for this algorithm, associating the update operations with the source column's weight is appropriate both for the static load balance and the dynamic load balance goals.

4 Integration of index-based and nested mappings

The $\mathbf{F} \times \mathbf{S} \times \mathbf{A}$ mapping scheme of Section 2.3 may be used for sparse Cholesky factorization also. If η is the number of nonzeros in L, the communication complexity of this scheme would be $O(\eta \sqrt[3]{P})$. This would be a purely index-based mapping strategy, with no use being made of the information available in the elimination tree. This mapping scheme will be called the **GI** scheme, for **G**lobal **I**ndex-based. The reason for the qualification "Global" will become clear when "Local" mapping schemes are described in subsequent sections.

In this section, a method of combining the nested mapping idea described in Section 3 with the index-based mapping idea of Section 2.3 (three-dimensional logical processor system view) is presented. This mapping scheme will be called the **GN** scheme, for **G**lobal **N**ested.

4.1 GN mapping scheme

The **GN** mapping scheme consists of a nested data mapping function to be discussed in this section, and a computation mapping function derived from the data mapping using the same intermediate computes rule that was used in Section 2.3 for the $\mathbf{F} \times \mathbf{S} \times \mathbf{A}$ scheme.

The first step in determining the data mapping function for the **GN** mapping scheme is the specification of the composite processor tree's structure. Assuming that the number of processors P is a power of 2, the simplest choice is a complete binary tree with $\log_2 P + 1$ levels and $2P - 1$ nodes, the leaves of this tree corresponding to individual processors, the next higher level corresponding to groups of two processors, the next higher level having groups of four processors, and so on.

The next step is specifying the groups of processors associated with the composite processor nodes. Recall that the three-dimensional logical view of the processors is that of a $P_{\mathbf{F}} \times P_{\mathbf{S}} \times P_{\mathbf{A}}$ grid. The root composite processor is associated with the group of all processors, which corresponds to the entire logical grid. The groups of processors associated with the two daughters of the root composite processor should each have $\frac{P}{2}$ processors. Of the many possible ways in which these groups may be defined, three possibilities stand out as the most natural ones. Each of these corresponds to evenly slicing the logical processor grid using a plane perpendicular to one of the three dimensions (**F**, **S**, or **A**) of the logical processor grid. If any of $P_{\mathbf{F}}$, $P_{\mathbf{S}}$, and $P_{\mathbf{A}}$ is 1, the corresponding slicing is not possible. After this first slicing, the two subgrids thus created become associated with the two daughters of the root composite processor. The groups of processors associated with the other composite processor tree nodes are determined in a similar manner, *i.e.*, by deciding along which direction to slice the mother composite processor.

As a result of the above process, each composite processor will correspond to a contiguous subgrid of the original logical grid. Each subgrid may be identified by its extent (lower and upper coordinates) in each of the three dimensions. If c is a composite processor, the notations $\mathsf{low}_{\mathbf{F}}(c)$, $\mathsf{upp}_{\mathbf{F}}(c)$, $\mathsf{low}_{\mathbf{S}}(c)$, $\mathsf{upp}_{\mathbf{S}}(c)$, $\mathsf{low}_{\mathbf{A}}(c)$, and $\mathsf{upp}_{\mathbf{A}}(c)$, will be used for the six numbers identifying the subgrid associated with c. The notation $\mathsf{subgrid}(c)$ will also be used to refer to these six quantities collectively.

The nested mapping procedure described in Section 3 is used to map nodes of the elimination tree onto composite processors. The same name used there, *i.e.*, cpmap, is used for this mapping function. The nodes of the elimination tree are not considered to correspond to columns in the matrix, but just to indices x, where $1 \leq x \leq n$.

The data mapping function for the integrated mapping approach **GN** is then specified. The notation $\mathsf{rmod}(x, l, u)$ is used to represent the expression $l+x \ \mathbf{mod}\ (u-l+1)$. This is just a modulus function whose value is between l and u. Consider a matrix element L_{ij}. Let ci denote $\mathsf{cpmap}(i)$ and let cj denote $\mathsf{cpmap}(j)$. Then the data mapping function is given by

$$\mathsf{map}_D(L_{ij}) = <\mathsf{rmod}(i, \mathsf{low}_{\mathbf{F}}(ci), \mathsf{upp}_{\mathbf{F}}(ci)), \mathsf{rmod}(j, \mathsf{low}_{\mathbf{S}}(cj), \mathsf{upp}_{\mathbf{S}}(cj)), \mathsf{rmod}(j, \mathsf{low}_{\mathbf{A}}(cj), \mathsf{upp}_{\mathbf{A}}(cj))>.$$

There are some similarities and some differences between the index-based data mapping function of Section 2.3 and the one given here. Both functions are constructed so that the first component of the map is dependent only on the value of i, and the other two components are dependent only on the value of j. This ensures that all matrix elements in the same matrix row are owned by processors falling within a plane perpendicular to the **F** dimension, and that all matrix elements in the same matrix column are owned by processors falling within a line parallel to the **F** dimension. Consequently, the factor communication, source communication, and the aggregate communication will be constrained to occur parallel to their corresponding dimensions, and the communication complexity can be guaranteed to be $O(\eta\sqrt[3]{P})$.

The difference between the two mapping functions is in the range of values allowed for the three components of the map. Consider a matrix column j. With an index-based data mapping function, the elements in this column could be anywhere in the processor grid, because the modulus operations for the second and third components of the map are performed using P_2 and P_3. On the other hand, if cj, the composite processor to which the index j gets mapped by the nested mapping, is associated with a subgrid which is smaller than the entire logical grid, the nested mapping data function will ensure that the possible choices for the second and third components of the map for the elements in column j are limited to a smaller range than $0..P_2 - 1$ and $0..P_3 - 1$. Because of the way the nested mapping is determined, matrix elements in all columns that update column j will also have their second and third map components restricted to within these smaller ranges and to possibly even smaller ranges.

The nested mapping function therefore achieves some amount of localization of related data, and can be expected to reduce the source communication and the aggregate communication as compared to the index-based mapping function. The reasoning behind this expectation is similar to that given in Section 3.2 when discussing the effect of a nested mapping strategy on the fan-out and the fan-in algorithms. There it was shown that the fan-in algorithm's aggregate communication benefitted more from the use of a nested mapping than the fan-out algorithm's source communication. It might be expected that a similar behavior is expected when going from **GI** to **GN**. Experimental results, to be presented in Section 4.3, bear out this expectation.

The nested mapping function also reduces the range of possible values for the first component of the map. This has the effect of lowering the factor communication. The reason why this is so is as follows. Consider a matrix element L_{ij}. It is needed as a factor element for L_{ii}, $L_{i_1 i}$, $L_{i_2 i}$, ..., where i_1, i_2, ..., are row indices of nonzeros in column j whose value is greater than i. In the elimination tree, it will be the case that i_1 will be an ancestor of i, i_2 will be an ancestor of i_1, i_3 will be an ancestor of i_2, and so on. Let i_t be the last index in this sequence. The subgrid associated with $\mathsf{cpmap}(i_t)$ may have a smaller range than $0..P_1 - 1$ in the first dimension. This means that the first component of $L_{i_t i}$'s map will be within this smaller range. Also, all the other elements mentioned above will have their map's first component within this smaller range, since their row indices all correspond to descendants of i_t in the elimination tree. This means that, the number of processors over which these elements are distributed will, in general, be smaller when a nested mapping is used as compared to when an index-based mapping is used. The amount of factor communication can therefore be expected to be less in **GN** than in **GI**.

Before proceeding to the experimental results of comparing the **GI** and the **GN** schemes, some information on the sample problem that will be used throughout this paper is provided.

4.2 Sample problem details

The sample problem used in the rest of this paper as an example for showing experimentally obtained communication statistics when various mapping schemes are used is that of factorizing a sparse matrix derived by applying a 9-point stencil on a $63x63$ grid, with the number of processors being 64. The processors are viewed logically to be organized as a $4 \times 4 \times 4$ grid. The matrix has about 4000 columns, and about 100000 nonzeros in L. The number of operations required to factorize it is about 4 million.

It is usually important to consider partitioning a sparse matrix into blocks for a number of reasons, including the desire to reduce the number of messages in the communication overhead and to increase the number of dense operations. One blocking strategy involves partitioning the index space $(1..n)$ into contiguous ranges so that each partition has a maximum size of b, where b is a an appropriately chosen integer, and so that the columns corrresponding to each partition have a common nonzero structure. The latter property can be ensured by identifying groups of columns with the same nonzero structure (called *supernodes*) and numbering the columns so that nodes in a supernode are consecutively numbered. The index space partitioning derived in this manner is used to split both the column index space and the row index space. The resulting “grid” structure defines the block structure of the sparse matrix. With this blocking scheme, blocks will be dense in the column direction but may be sparse in the row direction. All the results for the sample matrix presented in this paper are derived based on a block size of 6.

4.3 Experimental comparison of GI and GN schemes

Table 1 shows some of the communication statistics for the sample problem situation. The slicing order that was used in determining the shapes of the subgrids associated with composite processors was **AASFSF**. The message counts were determined by assuming that blocks that belonged to the same block-column would be sent together.

Table 1: Comparison between **GI** and **GN**

Comm. type	Mesg. count		Mesg. volume	
	GI	**GN**	**GI**	**GN**
Factor	16562	9854	203200	182092
Source	14169	4820	212635	132329
Aggregate	9324	162	222302	10586
Total	40055	14386	638137	325007

It can be seen that, when going from **GI** to **GN**, the aggregate communication is reduced the most, the source communication is next, and the factor communication is affected the least. In general, a communication balance between the three dimensions of the logical processor grid is desirable. The next section looks at how localizing related data even further can be used to reduce the factor communication further at the expense of a little increase in one of the other components of the communication overhead.

5 Localizing data

Consider a matrix element L_{ij}. Clearly, $i \geq j$, and i will be an ancestor of j in the elimination tree (a node is considered to be ancestor of itself). Let $ci = \mathsf{cpmap}(i)$ and let $cj = \mathsf{cpmap}(j)$. Then ci will be an ancestor of cj in the composite processor tree. Therefore, the extent in each dimension of the subgrid associated with the composite processor cj must be smaller than or equal to that of the subgrid associated with the composite processor ci. In particular, it must be true that

$$\mathsf{upp}_{\mathbf{F}}(cj) - \mathsf{low}_{\mathbf{F}}(cj) \leq \mathsf{upp}_{\mathbf{F}}(ci) - \mathsf{low}_{\mathbf{F}}(ci).$$

This motivates the use of cj in determining the first component of the data mapping for L_{ij}, since the set of processors owning elements belonging to a particular matrix column will, in general, become smaller than that with the **GN** scheme, which will lead to a reduction in the factor communication, based on the reasoning given in Section 4.1.

5.1 The new data mapping

The new data mapping function is

$$\mathsf{map}_D(L_{ij}) = < \mathsf{rmod}(i, \mathsf{low}_{\mathbf{F}}(cj), \mathsf{upp}_{\mathbf{F}}(cj)), \mathsf{rmod}(j, \mathsf{low}_{\mathbf{S}}(cj), \mathsf{upp}_{\mathbf{S}}(cj)), \mathsf{rmod}(j, \mathsf{low}_{\mathbf{A}}(cj), \mathsf{upp}_{\mathbf{A}}(cj)) > .$$

Notice that only the first component of the map is different from that in the data mapping function for the **GN** scheme.

5.2 Retaining the computation map: the LDSC scheme

An important consequence of the above data mapping function is that the matrix elements belonging to the same matrix row are no longer guaranteed to have the same value for the first component of their map. This is because their respective column indices are being used now. This means that a source element L_{ik} and a target element L_{ij} may be at different "levels" in the **F** dimension of the logical processor grid. The computation mapping rule given in Section 2.3, which is what the **GN** scheme uses, has to be reexamined. That rule is reproduced here for convenience:

$$\mathsf{map}_C(formation(L_{jk} * L_{ik})) = <[\mathsf{map}_D(L_{ik})]_1, [\mathsf{map}_D(L_{ij})]_2, [\mathsf{map}_D(L_{ik})]_3>,$$

Since the second and third components of the data maps of matrix elements have not been affected, it appears reasonable to leave the second and third components of the computation mapping unchanged.

The computation map given above uses $[\mathsf{map}_D(L_{ik})]_1$ for the first component. This is the "level" in the **F** direction of the source element, which may now be different from that of the target element. If this is retained as it stands, the resulting mapping scheme will be called the **LDSC** scheme, for **L**ocal **D**ata **S**ource-level **C**omputes.

Consider the communication requirements of the **LDSC** scheme. Since the processors where the update formation operations take place in relation to the locations of the source elements have not changed from what they were in the **GN** scheme (their absolute locations may have changed in the first component because the data mapping has changed in that component), the amount of source communication required will remain unchanged relative to the **GN** scheme, and will be restricted to occur only along the **S** dimension. The factor communication is expected to be reduced as compared to the **GN** scheme, as discussed above. This component of the communication overhead continues to be restricted to occur only along the **F** dimension. The situation with the aggregate communication is different. While it was the case in the **GN** scheme that the locations where the update contributions for a target element were formed would lie on a line parallel to the **A** dimension passing through the processor owning the target element, such is not necessarily the case in the **LDSC** scheme because the update contributions are formed at processors whose first component would be that of the corresponding source element, which could be different from that of the target element. The second component of the processors where such update contributions are formed would, however, be the same since it depends only on the data mapping of the target element. Therefore, the combined contributions for a target element would have to be sent from a subset of the processors in a plane perpendicular to the **S** dimension, passing through the processor owning the target element. This means that the aggregate communication is $O(\eta P_{\mathbf{F}} P_{\mathbf{A}})$, and that the overall communication complexity in the case when $P_{\mathbf{F}} = P_{\mathbf{S}} = P_{\mathbf{A}} = \sqrt[3]{P}$ would become $O(\eta \sqrt[3]{P^2})$.

5.3 Modifying the computation map: the LDTC scheme

Another possibility for the computation mapping function is to use $[\mathsf{map}_D(L_{ij})]_1$ as its first component. This means that the update formation takes place at the same "level" in the **F** dimension as the location of the target element. The resulting mapping scheme is labeled the **LDTC** scheme, for **L**ocal **D**ata **T**arget-level **C**omputes.

The communication requirements of the **LDTC** scheme are quite the opposite of that of the **LDSC** scheme. The aggregate communication will not be affected compared to what it is for the **GN** scheme, because the processor locations where the update contributions are formed and aggregated do not change in relation to the target elements (their absolute locations may have changed to a processor with a different first component). The factor communication is expected to be reduced as compared to the **GN** scheme, and retains its property that it is restricted to occur only along the **F** dimension. The source communication, however, is changed. Since the target elements of a source element may be at "levels" in the first dimension different from that itself, the element may have to be sent as a source element to processors having a different first component in their identifying tuple. The restriction that the source communication does not occur along the **A** dimension still remains because the third component of the computation mapping function depends only on the location of the source element. A source element may, therefore, have to be sent to a subset of the processors in a plane perpendicular to the **A** dimension, passing through the processor owning the source element. By a reasoning similar to that used for the **LDSC** scheme, the overall communication complexity for the **LDTC** scheme would also be $O(\eta \sqrt[3]{P^2})$, when $P_{\mathbf{F}} = P_{\mathbf{S}} = P_{\mathbf{A}} = \sqrt[3]{P}$.

5.4 Communication statistics for LDSC and LDTC

The sample problem situation described in Section 4.2 was again used to obtain communication statistics for the two new mapping schemes just described. The results are summarized in Table 2.

Table 2: Communication statistics for **LDSC** and **LDTC**

Comm. type	Mesg. count		Mesg. volume	
	LDSC	**LDTC**	**LDSC**	**LDTC**
Factor	2015	4402	102992	133927
Source	3295	6529	132329	159464
Aggregate	820	162	40750	10586
Total	6130	11093	276071	303977

It can be seen that, with respect to the factor communication, there is a greater reduction in the **LDSC** scheme than in the **LDTC** scheme. The increase in the source communication when going from **GN** to **LDTC** is greater than the increase in the aggregate communication when going from **GN** to **LDSC**.

6 Localizing computation

The **LDSC** mapping scheme incorporates two techniques for data localization: the use of a nested mapping and bringing elements of the same matrix column "closer". Using the nested mapping helps in reducing the aggregate communication the most. The data localization technique of section 5 helps in reducing the factor communication only. Data localization causes some computation localization and hence communication localization. This section considers a technique for localizing computation in a direct manner. The data mapping function used by the **LDSC** scheme is retainedc. The resulting mapping scheme will reduce the source communication further at the expense of more aggregate communication. This mapping scheme will be seen to have some common features with a mapping strategy for a parallel multifrontal algorithm.

6.1 The new computation map

In the **LDSC** scheme, the second component of the computation mapping function is an important determinant of the amount of source communication required. The second component of the **LDSC**'s computation mapping function is $[\mathsf{map}_D(L_{ij})]_2$, which is actually $\mathsf{rmod}(j, \mathsf{low}_{\mathbf{S}}(cj), \mathsf{upp}_{\mathbf{S}}(cj))$, where $cj = \mathsf{cpmap}(j)$. The range of this component therefore extends over the second dimension of the subgrid associated with cj. Consider the first and third components of the computation map. They are, respectively, $[\mathsf{map}_D(L_{ik})]_1$ and $[\mathsf{map}_D(L_{ik})]_3$, which are actually $\mathsf{rmod}(i, \mathsf{low}_{\mathbf{F}}(ck), \mathsf{upp}_{\mathbf{F}}(ck))$ and $\mathsf{rmod}(k, \mathsf{low}_{\mathbf{F}}(ck), \mathsf{upp}_{\mathbf{F}}(ck))$, where $ck = \mathsf{cpmap}(k)$. This means that their ranges extend over the corresponding dimensions of the subgrid associated with ck. Since k is a descendant of j in the elimination tree, the subgrid associated with ck should be contained within the subgrid associated with cj. The processor where the update contribution of L_{ik} onto L_{ij} is formed will, therefore, be somewhere within the subgrid associated with cj.

The computation localization idea is to restrict these update contributions to take place at processors within the subgrid associated with ck, since that will, in general, be smaller. This can be achieved by changing the second component of the computation mapping function to

$$\mathsf{rmod}(j, \mathsf{low}_{\mathbf{S}}(ck), \mathsf{upp}_{\mathbf{S}}(ck)),$$

while keeping the other two components unchanged. This ensures that the update formation operations that use a matrix element are restricted to take place within the subgrid associated with the composite processor to which that element's column index is mapped to. The number of processors to which a matrix element has to be sent as a source element is expected to be smaller, thus lowering the source communication.

With this computation mapping, the update contributions for a target element may be formed and aggregated at processors within the source column's subgrid. These aggregates may then be sent to the processor owning the target element. There can be no guarantee that aggregates for a target element will come only from processors within a line or within a plane. In the worst case, aggregates may come from every other processor in the system. However, it can be expected that due to the data localization techniques used, such situations would be very rare. The increase in the aggregate communication is therefore not expected to outweigh the decrease in the source communication.

Instead of having the aggregates sent directly from the processors where the update contributions are formed, a two-stage aggregation process may be employed. In the first stage, the aggregates for a target element from all source elements whose column indices have been mapped to the same subgrid are accumulated at a designated processor within the subgrid. Such a processor will be called a *temporary accumulator* for the target element. In the second stage, all the temporary accumulators for a target element then send their accumulated aggregates to the processor owning the target processor. The first stage may be called *local aggregation* and the second stage may be called *global aggregation*.

The temporary accumulator processor locations are chosen so that the local aggregation communication occurs only along the **A** dimension. This means that the temporary accumulator's third component is the only component that should be different from where the update contribution is formed. The temporary accumulator for a target element L_{ij} for update contributions coming from source elements whose column indices are mapped onto the composite processor ck can therefore be given by

$$< \mathsf{rmod}(i, \mathsf{low}_F(ck), \mathsf{upp}_{\mathbf{F}}(ck)), \mathsf{rmod}(j, \mathsf{low}_{\mathbf{S}}(ck), \mathsf{upp}_{\mathbf{S}}(ck)), \mathsf{rmod}(j, \mathsf{low}_{\mathbf{A}}(ck), \mathsf{upp}_{\mathbf{A}}(ck)) > .$$

The new mapping scheme that uses the above ideas will be labeled the **LDLC** scheme, for **L**ocal **D**ata **L**ocal **C**omputation.

6.2 Communication statistics for LDLC

Table 3 gives the communication statistics for the **LDLC** scheme. The aggregate communication has been split into its local and global components.

Table 3: Communication statistics for **LDLC**

Comm. type	Mesg. count	Mesg. volume
Factor	1721	102992
Source	890	75276
Local aggr.	146	9158
Global aggr.	3083	77815
Total	5840	265241

7 Related work

Various previous works have addressed the use of index-based mapping strategies for dense matrix factorization [14, 9, 1]. Several researchers have also proposed nested mapping strategies for sparse matrix factorization [12, 10, 16, 7].

The combination of index-based mapping and nested mapping strategies has been studied independently by other researchers in specific contexts. In [13], a mapping strategy for a multifrontal algorithm [5, 15] is presented that combines a two-dimensional partitioning of the matrix with a subtree-to-subcube mapping [12]. In [4], a mapping strategy is presented that integrates the torus wrap mapping and a nested mapping based on the elimination tree. However, the present work is the first attempt at a comprehensive characterization of the communication implications of integrating index-based mappings with nested mappings. Two new strategies for performing this integration so that there is further reduction in communication have also been included.

The mapping framework introduced in this paper allows for several existing mapping strategies to be included in it. A mapping strategy for a parallel multifrontal algorithm, which uses a three-dimensional logical view of the processor system is closely related to the **LDLC** mapping scheme. This provides for a clean integration of mapping strategies for the multifrontal algorithm into the unifying framework as described below. The block fan-out algorithm presented in [17] corresponds to an index-based mapping scheme where $P_{\mathbf{A}}$ is unity, *i.e.*, it is a $\mathbf{F} \times \mathbf{S}$ scheme, using the notation of this paper. The fan-both algorithm presented in [2] corresponds to a mapping scheme where $P_{\mathbf{F}}$ is unity. Nested mapping is not used in [2], but can easily be used as shown here.

In a multifrontal algorithm, each node in the elimination tree is associated with a dense lower triangular matrix called its *front*. The front consists of the column itself and one column for each of its target columns. The latter columns are said to form an *update matrix*. Consider a leaf node k^d in the elimination tree. The elements in its update matrix correspond to the update contributions from column k^d to all its targets. Let k^m be the mother of k^d in the elimination tree. It can be shown that every element (except those in column k^m) updated by elements in column k^d is updated by an element in column k^m. The multifrontal approach consists of applying the update contributions from column k^d to the corresponding locations in column k^m's front instead of directly onto the target elements. This process is known as *merging* the daughter node's update matrix into the mother node's front. After all daughters of k^m have performed the merging operation, column k^m may be normalized and its update contributions may be applied into its front. The update matrix of k^m may then be merged into the front of its mother node, and the process continued until the root of the elimination tree.

For a parallel multifrontal algorithm, it has been shown that parallelism needs to be exploited both within a front (forming the update matrix) and between fronts (merging update matrices into fronts). Each node's front is therefore distributed among several processors, which cooperatively perform the computation associated with that front. A nested mapping, such as recursive partitioning, is especially appropriate for a parallel multifrontal algorithm. The processors comprising the composite processor to which a node is mapped will be the processors cooperatively processing the node's front.

Multifrontal algorithms are usually implemented with a supernodal view of the matrix. As mentioned earlier, a supernode is a group of columns with identical nonzero structures. A supernodal multifrontal algorithm has one front associated with each supernode. The front consists of the columns constituting the supernode, and the update matrix containing one column for each of the targets of these columns. The intrafrontal computation of a supernodal front consists of performing the update operations among the supernodal columns, their normalization operations, and the update formation operations for their contributions to their targets. An update contribution thus formed is applied to the appropriate location in the front, where the update contributions to the same target element from any descendant nodes in the elimination tree would have already been aggregated.

The intrafrontal computation in such a multifrontal algorithm and the formation of aggregated updates at the temporary accumulator processors in the **LDLC** scheme described earlier are essentially the same computation because both involve the formation of updates from sources to their targets with the updates being aggregated at processors that are within the subgrid associated with the sources. The difference between the two algorithms lies in how these aggregated updates are further aggregated and applied to their ultimate targets. In the **LDLC** scheme, a global aggregation phase occurs in which the local aggregates are directly sent from the temporary accumulators to the processor owning the target element. In the multifrontal algorithm, the local aggregates are combined in a hierarchical manner driven by the elimination tree structure. A local aggregate is sent to the processor where the corresponding element in the mother node's front is mapped, where it is merged into that element.

8 Conclusion

The paper has considered how two techniques, the use of index-based mappings and the use of nested mappings, can be systematically integrated to achieve efficient communication during parallel sparse Cholesky factorization. The different components in the communication overhead of parallel sparse Cholesky factorization were identified. The concepts of data localization and computation localization were used to motivate two new communication-reducing strategies. Many existing mapping strategies and the strategies presented here were shown to fit into a unifying framework. Experimentally obtained communication statistics were used to demonstrate the efficacy of mapping strategies and to confirm expectations on the effect of various mapping strategies on different components of the communication overhead.

References

[1] C. Ashcraft, "A taxonomy of distributed dense LU factorization methods," Boeing Computer Services, Technical Report No. ECA-TR-161, 1991.

[2] C. Ashcraft, "The fan-both family of column-based distributed Cholesky factorization algorithms," Boeing Computer Services, Technical Report No. MEA-TR-208, 1992.

[3] C. Ashcraft, S. Eisenstat, and J. W.-H. Liu, "A fan-in algorithm for distributed sparse numerical factorization," *SIAM J. Sci. Statist. Comput.*, Vol. 11, pp. 593-599, 1990.

[4] A. J. Cleary, "A new torus-like mapping for parallel sparse matrix factorization," Advanced Computation Group, Australian National University, Technical Report No. ACTR_13_09_92, 1992.

[5] I. Duff and J. Reid, "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Trans. Math. Software*, Vol. 9, pp. 302-325, 1983.

[6] K. Eswar, C.-H. Huang, and P. Sadayappan, "On mapping data and computation for parallel sparse Cholesky factorization," Department of Computer and Information Science, The Ohio State University, Technical Report No. OSU-CISRC-7/94-TR40, 1994.

[7] K. Eswar, P. Sadayappan, and V. Visvanathan, "Multifrontal factorization of sparse matrices on shared-memory multiprocessors," *Proceedings of the Twentieth International Conference on Parallel Processing*, St. Charles, IL, Vol. III, pp. 159-166, 1991.

[8] K. Eswar, P. Sadayappan, and V. Visvanathan, "Parallel direct solution of sparse linear systems," in *Parallel Computing on Distributed Memory Multiprocessors*, F. Özgüner and F. Erçal (eds.), Springer Verlag, Berlin Heidelberg New York, 1993.

[9] G. Fox, *et al.*, *Solving problems on concurrent processors: Volume 1*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[10] G. Geist and E. G.-Y. Ng, "Task scheduling for parallel sparse Cholesky factorization," *Internat. J. Parallel Programming*, Vol. 18, pp. 291-314, 1989.

[11] A. George, M. Heath, J. W.-H. Liu, and E. G.-Y. Ng, "Sparse Cholesky factorization on a local-memory multiprocessor," *SIAM J. Sci. Statist. Comput.*, Vol. 9, pp. 327-340, 1988.

[12] A. George, J. W.-H. Liu, and E. G.-Y. Ng, "Communication results for parallel sparse Cholesky factorization on a hypercube," *Parallel Comput.*, Vol. 10, pp. 287-298, 1989.

[13] A. Gupta and V. Kumar, "A scalable parallel algorithm for sparse matrix factorization," Department of Computer Science, University of Minnesota, Technical Report No. TR 94-19, 1994.

[14] B. Hendrickson and D. Womble, "The torus-wrap mapping for dense matrix calculations on massively parallel computers," Sandia National Laboratories, SAND Report No. SAND 92-0792, 1992.

[15] J. W.-H. Liu, "The multifrontal method for sparse matrix solution: theory and practice," Dept. of Computer Science, York University, Tech. Report CS-90-04, 1990.

[16] A. Pothen and C. Sun, "A distributed multifrontal algorithm using clique trees," Computer Science, Pennsylvania State University, Technical Report No. 91-24, 1991.

[17] E. Rothberg and A. Gupta, "An efficient block-oriented approach to parallel sparse Cholesky factorization," *Supercomputing '92 Proceedings*, 1992.

[18] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Trans. Math. Software*, Vol. 8, pp. 256-276, 1982.