Task Scheduling using a Block Dependency DAG for Block-Oriented Sparse Cholesky Factorization

Heejo Lee*, Jong Kim*, Sung Je Hong*, Sunggu Lee**
*Dept. of Computer Science and Engineering
**Dept. of Electrical Engineering
Pohang University of Science and Technology
San 31 Hyoja Dong, Pohang 790-784, KOREA
{heejo,jkim}@postech.ac.kr

Keywords

Task scheduling, parallel sparse matrix factorization, block-oriented Cholesky factorization, directed acyclic graph.

ABSTRACT

Block-oriented sparse Cholesky factorization decomposes a sparse matrix into rectangular sub-blocks; each block can then be handled as a computational unit in order to increase data reuse in a hierarchical memory system. Also, the factorization method increases the degree of concurrency with the reduction of communication volumes so that it performs more efficiently on a distributed-memory multiprocessor system than the customary column-oriented factorization method. But until now, mapping of blocks to processors has been designed for load balance with restricted communication patterns. In this paper, we represent tasks using a block dependency DAG that shows the execution behavior of block sparse Cholesky factorization in a distributed-memory system. Since the characteristics of tasks for the block Cholesky factorization are different from those of the conventional parallel task model, we propose a new task scheduling algorithm using a block dependency DAG. The proposed algorithm consists of two stages: *early-start clustering*, and *affined cluster mapping*. The *early-start clustering* stage is used to cluster tasks with preserving the earliest start time of a task without limiting parallelism. After task clustering, the *affined cluster mapping* stage allocates clusters to processors considering both communication cost and load balance. Experimental results on the Fujitsu parallel system show that the proposed task scheduling approach outperforms other processor mapping methods.

1. INTRODUCTION

Sparse Cholesky factorization is a computation intensive operation commonly encountered in scientific and engineering applications including structural analysis, linear programming, and circuit simulation. Much work has been done on parallelizing sparse Cholesky factorization, which is used for solving large sparse systems of linear equations. The performance of parallel Cholesky factorization is greatly influenced by the method used to map a sparse matrix onto the processors of a parallel system. Based on the mapping method, parallel sparse Cholesky factorizations are classified into the column-oriented Cholesky, the supernodal Cholesky, the amalgamated supernodal Cholesky, and the 2-D block Cholesky. The earliest work is based on the column-oriented Cholesky in which a single column is mapped to a single processor [6; 13]. In the supernodal Cholesky, a supernode, which is a group of consecutive columns with the same row structure, is mapped to a single processor [3; 16]. The amalgamated supernodal Cholesky uses the supernode amalgamation technique in which several small supernodes are merged into a greater supernode, and an amalgamated supernode is then mapped to a single processor [2; 19]. In the 2-D block Cholesky, a matrix is decomposed into rectangular blocks, and a block is mapped to a single processor [8; 20].

The recent advanced methods for sparse Cholesky factorization are based on the use of the 2-D block Cholesky to process non-zero blocks using Level 3 Basic Linear Algebra Subprograms (BLAS) [5; 6]. Such a 2-D decomposition is more scalable than a 1-D decomposition and has an increased degree of concurrency [23; 24]. Also, the 2-D decomposition allows us to use efficient computation kernels such as Level 3 BLAS so that caching performance is improved [19]. Even in a single processor system, block factorizations are performed efficiently [15].

There are few works reported for the 2-D block Cholesky in a distributed-memory system. Rothberg and Gupta introduced the block fan-out algorithm [20]. Similarly, Dumitrescu *et al.* introduced the block fan-in algorithm [8]. Gupta, Karypis, and Kumar [12] also used 2-D mapping for implementing a multifrontal method. In [18], Rothberg has shown that a block fan-out algorithm using the 2-D decomposition outperforms a panel multifrontal method using 1-D decomposition. Even though the block fan-out algorithm increases the concurrency with reduced communication volumes, the performance achieved is not satisfactory due to load imbalance among the processors. Therefore, several load balance heuristics have been proposed in [21].

However, the load balance is not the sole key parameter for improving the performance of parallel block sparse Cholesky factorization. The load balancing mapping only guarantees that the computation is well distributed among processors; it does not guarantee that the computation is well scheduled when considering the communication requirements. Thus, communication dependencies among blocks may cause some processors to wait even with balanced loads.

In this paper, we introduce a task scheduling method using a DAG-based task graph, which represents the behavior of block sparse Cholesky factorization with the exact amount of computation and communication cost. As we will show in Section 3, a task graph for sparse Cholesky factorization is different from a conventional parallel task graph. Hence we propose a new heuristic algorithm which attempts to minimize the completion time while preserving the earliest start time of each task in a graph. It has been reported that a limitation on memory space can affect the performance [26]. But we do not consider the memory space limitations, since we assume that the factorization is done on a distributed-memory system with sufficient memory to handle the work assigned to each processor. Even though there has been an effort to use DAG-based scheduling for irregular computations on a parallel system with a low-overhead communication mechanism [9], this paper presents the first work that deals with the entire framework of applying a scheduling approach for block-oriented sparse Cholesky factorization in a distributed system.

The next section describes the block fan-out method for parallel sparse Cholesky factorization. In Section 3, the sparse Cholesky factorization is modeled as a DAG-based task graph, and the characteristics of a task for this problem are summarized. Since the characteristics of this type of task are different from the conventional precedence-constrained parallel task, a new task scheduling algorithm is proposed in Section 4. The performance of the proposed scheduling algorithm is compared with the previous processor mapping methods using experiments on the Fujitsu AP1000+ parallel system in Section 5. Finally, in Section 6, we summarize and conclude the paper.

2. BLOCK-ORIENTED SPARSE CHOLESKY FACTORIZATION

This section describes the block fan-out method for sparse Cholesky factorization, which is an efficient method for distributed memory systems. The block Cholesky factorization method decomposes a sparse matrix into rectangular blocks, and then factorizes it with dense matrix operations.

2.1 Block Decomposition

The most important feature in sparse matrix factorizations is the use of supernodes [2; 3]. A supernode is a set of adjacent columns in the sparse matrix, which consists of a dense triangular block on the diagonal, and identical non-zero structures in each column below the diagonal. Since supernodes represent the sparsity structure of a sparse matrix, block decomposition with supernodes makes non-zero blocks as dense as possible, and easy to handle due to shared common boundaries [20].

The performance of the factorization is improved by supernode amalgamation, in which small supernodes are amalgamated into bigger ones in order to reduce the overhead for managing small supernodes and to improve caching performance [1; 2; 20]. Supernode amalgamation is a process of identifying locations of zero elements that would produce larger supernodes if they were treated as non-zeros. In the following, amalgamated supernodes will be assumed by default, and will be referred to simply as supernodes.

In a given $n \times n$ sparse matrix with N supernodes, the supernodes divide the columns of the matrix $(1,..,n)$ into contiguous subsets $(\{1,..,p_2-1\}, \{p_2,..,p_3-1\},.., \{p_N,..,n\})$. The size of the i-th supernode is n_i, i.e., $n_i = p_{i+1} - p_i$ and $\sum_{i=1}^{N} n_i = n$. A partitioning of rows and columns using supernodes produces blocks such that a block $L_{i,j}$ is the sub-matrix decomposed by supernode i and supernode j. Then, the row numbers of elements in $L_{i,j}$ are in $\{p_i,..,p_{i+1}-1\}$, and the column numbers of elements in $L_{i,j}$ are in $\{p_j,..,p_{j+1}-1\}$.

After the block decomposition of the sparse factor matrix, the total number of blocks is $N(N+1)/2$. The number of diagonal blocks is N, and all diagonal blocks are non-zero blocks. Each of the $N(N-1)/2$ rectangular blocks is either a zero block or a non-zero block. A zero block refers to a block whose elements are all zeros, and a non-zero block refers to a block that has at least one non-zero element.

After block decomposition using supernodes, the resulting structure is quite regular [20]. Each block has a very simple non-zero structure in which all rows in a non-zero block are dense and blocks share common boundaries. Therefore, the factorization can be represented in a simple form.

```
1.  for $k = 1$ to $N$ do
2.      $L_{kk} = \mathsf{Factor}(L_{kk})$
3.      for $i = k+1$ to $N$ with $L_{ik} \neq 0$ do
4.          $L_{ik} = L_{ik} L_{kk}^{-1}$
5.      for $j = k+1$ to $N$ with $L_{jk} \neq 0$
6.          for $i = j$ to $N$ with $L_{ik} \neq 0$ do
7.              $L_{ij} = L_{ij} - L_{ik} L_{jk}^{T}$
```

Figure 1: Sequential block Cholesky factorization.

2.2 Block Cholesky Factorization

The sequential algorithm for block Cholesky factorization, as described in [20], is shown in Figure 1. The algorithm works with the blocks decomposed by supernodes to retain as much efficiency as possible in block computation. The block computations can be done using efficient matrix-matrix operation packages such as Level 3 BLAS [4]. Such block computations require no indirect addressing, which leads to enhanced caching performance and close to peak performance on modern computer architectures [5].

2.3 Block Operations

Let us denote the dense Cholesky factorization of a diagonal block L_{kk} (Step 2 in Figure 1) as $bfact(k)$. Similarly, let us denote the operation of Step 4 as $bdiv(i,k)$, and the operation of Step 7 as $bmod(i,j,k)$. These three block operations are the primitive operations used in block factorization.

Even though a non-zero block has a sparse structure, we handle it as a dense structure. Since the blocks decomposed by supernodes are well-organized, such a sparse operation for blocks are rarely required [18]. Therefore, we assume that all block operations are handled with dense matrix operations.

For $bfact(k)$, an efficient dense Cholesky factorization can be used, and $bdiv(i,k)$ and $bmod(i,j,k)$ are supported by the level 3 BLAS routines such as $\_TRSM()$ and $\_GEMM()$. Therefore, we can measure the total number of operations required for each block operation as follows [5].

$$\begin{aligned} W_{bfact(k)} &= n_k(n_k+1)(2n_k+1)/6 \\ W_{bdiv(i,k)} &= n_i n_k^2 \\ W_{bmod(i,j,k)} &= 2n_i n_j n_k \end{aligned}$$

2.4 Required Number of Block Update Operations

The most computation intensive parts of block factorization are the block update operations. The block update operations, $bmod(i,j,k)$, are performed using a doubly nested loop, and thus take most of the time required for block factorization.

The number of required block updates for block $L_{i,j}$ can be measured. We use the notation $\alpha_{i,j}$ to denote whether $L_{i,j}$ is a non-zero block or not.

$$\alpha_{i,j} = \begin{cases} 0 & \text{if } L_{ij} = \emptyset \\ 1 & \text{otherwise} \end{cases}$$

Let $nmod(L_{i,j})$ denote the number of required $bmod()$ updates for $L_{i,j}$. When $L_{i,j}$ is a rectangular block,

$$nmod(L_{i,j}) = \sum_{k=1}^{j-1} \alpha_{i,k} \times \alpha_{j,k}.$$

For a diagonal block $L_{j,j}$,

$$nmod(L_{j,j}) = \sum_{k=1}^{j-1} \alpha_{j,k}.$$

Thus, the maximum number of updates for $L_{i,j}$ is $j-1$.

3. TASK MODEL WITH COMMUNICATION COSTS

Since a non-zero block $L_{i,j}$ is assigned to a processor [20], all block operations for a block can be treated as one task. This means that a task is executed in one processor, and a task consists of several subtasks for block operations. This section describes the characteristics of tasks, and proposes a task graph that represents the execution sequence of the block factorization. The task graph, referred to as a *block dependency DAG*, contains the costs of computations and communications, and the precedence relationships among tasks.

3.1 Task Characteristics

A task consists of multiple subtasks depending on the required block updates, and is represented using a tree of at most 2 levels of subtasks. If a diagonal block $L_{j,j}$ requires m block updates, i.e., $nmod(L_{j,j}) = m$, its corresponding task has $m+1$ subtasks including a $bfact(j)$ operation. Then, the task has m parent tasks as shown in Figure 2. Let us denote the m blocks of parent tasks as $L_{j,k_0}, .., L_{j,k_{m-1}}$, $1 \le k_i \le j-1$ for $0 \le i \le m-1$. If there is no block update required for $L_{j,j}$, i.e., $nmod(L_{j,j}) = 0$, then the task has no parent task and only one subtask for $bfact(j)$.

If a rectangular block $L_{i,j}$ requires m block updates, i.e., $nmod(L_{i,j}) = m$, then the corresponding task consists of

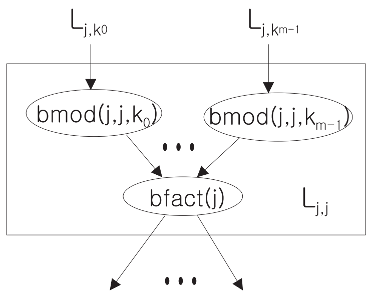

Figure 2: Task for a diagonal block $L_{j,j}$.

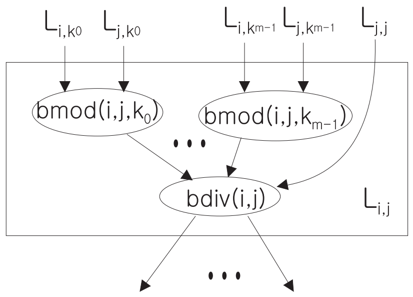

Figure 3: Task for a rectangular block $L_{i,j}$.

$m+1$ subtasks including the subtask for the $bdiv(i,j)$ operation. The task has $2m+1$ parents as shown in Figure 3. Let us refer to the $2m+1$ parent blocks as $L_{i,k_0}, L_{j,k_0}, .., L_{i,k_{m-1}}$, $L_{j,k_{m-1}}$, $L_{j,j}$. A subtask for a block update $bmod(i,j,k)$ executes after the two parent tasks for $L_{i,k}$ and $L_{j,k}$ have completed and sent their blocks to $L_{i,j}$. Also, for a diagonal block $L_{j,j}$, only one parent for $L_{j,k}$ needs to be completed before executing the $bmod(j,j,k)$ operation.

3.2 Task Graph

We now present a task graph for block Cholesky factorization. The task graph, referred to as the *block dependency DAG*, contains the precedence relations of blocks and the computation and the communication costs required for each block.

Let us assume that there are v non-zero blocks in the decomposed factor matrix. Each block is represented as a single task, so that there are v tasks $T_1, .., T_v$ and

$$\sum_{j=1}^{N} \sum_{i=j}^{N} \alpha_{i,j} = v.$$

We give a number to each block starting from the blocks in column 1 and ending at column N. For blocks in the same column, the block in the smallest row number is counted first. Such a numbering method implies that the task with the smallest number should be executed first among the precedence-constrained tasks in a processor.

Block Cholesky factorization is represented as a DAG, $G = (V, E, W, C)$. V is a set of tasks $\{T_1, .., T_v\}$ and $|V| = v$. E is a set of communication edges among tasks, and $|E| = e$. W_x, where $W_x \in W$, represents the computation cost of T_x. If T_x is a task corresponding to a diagonal block $L_{j,j}$, then

$$W_x = W_{bfact(j)} + \sum_{k=1}^{j-1} \alpha_{j,k} W_{bmod(j,j,k)}.$$

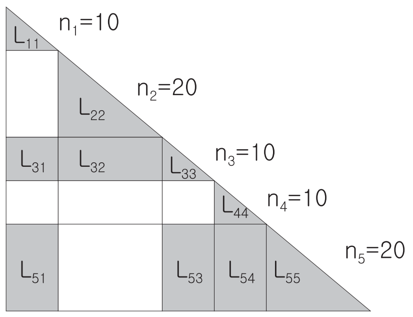

Figure 4: Sparse matrix decomposed using 5 supernodes.

Also, if T_x corresponds to a rectangular block $L_{i,j}$,

$$W_x = W_{bdiv(i,j)} + \sum_{k=1}^{j-1} \alpha_{i,k} \times \alpha_{j,k} W_{bmod(i,j,k)}.$$

C is a set of communication costs, and $c_{x,y}$ denotes the communication cost incurred along the edge $e_{x,y} = (T_x, T_y) \in E$. If T_x is the task for $L_{i,k}$ and T_y is the task for $L_{i,j}$, then T_x needs to send the block $L_{i,k}$ to T_y. Therefore, we can estimate the communication cost $c_{x,y}$ in a message-passing distributed system as follows:

$$c_{x,y} = t_s + t_c n_i n_k.$$

In the above equation, t_s is the startup cost for sending a message and t_c is the transfer cost for sending one floating point number. For most current message-passing systems, per-hop delay caused by the distance between two processors is negligible due to the use of "wormhole" routing techniques and the small diameter of the communication network [5].

We let $parent(x)$ denote the set of immediate predecessors of T_x, and $child(x)$ denote the set of immediate successors of T_x.

$$parent(x) = \{y | e_{y,x} \in E\}$$

$$child(x) = \{y | e_{x,y} \in E\}$$

Generally, the task graph G has multiple entry nodes and a single exit node. When a task consists of multiple subtasks, some of them can be executed as soon as the data is ready from the parents of the task. Thus, the time from start to finish for a task T_x is not a fixed value, e.g., W_x, but rather depends on the time when the required blocks for subtasks are ready from their parents. Therefore, scheduling a task as a run-to-completion task would result in an inefficient schedule. Most previous DAG-based task scheduling algorithms assume that a task is started after all parent tasks have been finished.

There are two approaches to resolve this situation. One is designing a task model including subtasks. The other is using a block dependency DAG. The former is a complicated approach because the task model may have many subtasks and some of them may already be clustered. This task model is also difficult to handle by a scheduling algorithm. The latter uses a simple, precise task graph as a block dependency DAG. But we can also extract relevant information on all subtasks from the relations in a block dependency graph. Therefore, we devise a task scheduling algorithm using a block dependency DAG.

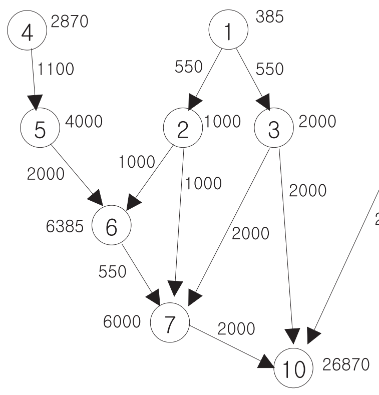

Figure 5: Block dependency DAG for the example sparse matrix.

4. TASK SCHEDULING USING A BLOCK DEPENDENCY DAG

Finding the optimal solution for a weighted DAG is known to be an NP-hard problem in the strong sense [22; 25]. When a task of a block dependency graph consists of only one or two subtasks, the scheduling problem using the block dependency DAG is reduced to the NP-hard scheduling problem. Thus, finding an optimal scheduling of a block dependency DAG is an NP-hard problem, so that a heuristic algorithm is presented in this section.

The proposed scheduling algorithm consists of two stages: task clustering without considering the number of available processors and cluster-to-processor mapping on a given number of processors. Most of the existing algorithms for a weighted DAG also use such a framework [28]. The goal of the proposed clustering algorithm, called *early-start clustering*, is to preserve the earliest possible start time of a task without reducing the degree of concurrency in the block dependency DAG. The proposed cluster mapping to processors, called *affined cluster mapping*, tries to reduce the communication overhead and balance loads among processors.

4.1 Task Scheduling Parameters

Several parameters are used in our scheduling method. The parameters, which are measured from a given block dependency DAG, include the work required for subtasks, and their parents, the earliest start time of a task, the earliest completion time of a task, and the level of a task.

Work and Parents of Subtasks:
A task T_i requiring m_i block updates consists of $m_i + 1$ subtasks. We refer the work for the $m_i + 1$ subtasks as $W_{i,0}, W_{i,1}, .., W_{i,m_i}$. Then the following equation is satisfied:

$$W_i = \sum_{j=0}^{m_i} W_{i,j}.$$

If T_i is the task for a diagonal block, then there are m_i parents. The parent tasks are referred to as $T_{k_0}, T_{k_1}, .., T_{k_{m_i-1}}$. If T_i is the task for a rectangular block, there are $2m_i + 1$ parents, which are referred to as $T_{k_0}, T_{k_1}, .., T_{k_{2m_i}}$.

Earliest Start Time of a Task:
The earliest start time of a task is defined as the earliest time when one of its subtasks is ready to run. Note that the earliest start time of a task is not the time when all required blocks for the task are received from the parent

tasks, although this time has been used by general DAG-based task scheduling algorithms. The earliest start time of T_i, $est(i)$, is defined recursively using the earliest completion time of the parent tasks. If T_i is the task for a diagonal block, then

$$est(i) = \begin{cases} 0 & \text{if } parent(i) = \emptyset, \\ \min\limits_{k \in parent(i)} (ect(k) + c_{k,i}) & \text{otherwise.} \end{cases}$$

Also, if T_i is the task for a rectangular block, then

$$est(i) =$$

$$\begin{cases} ect(k_0) + c_{k_0,i} & \text{if } parent(i) = \{k_0\}, \\ \min\limits_{0 \le j \le m-1} \begin{pmatrix} \max(ect(k_{2j}) + c_{k_{2j},i}, \\ ect(k_{2j+1}) + c_{k_{2j+1},i}) \end{pmatrix} & \text{otherwise.} \end{cases}$$

When T_i is clustered with a parent T_k, then we can omit the communication cost from T_k to T_i by setting $c_{k,i} = 0$. Thus, the above equations can be used in all of the clustering steps.

Earliest Completion Time of a Task: The earliest completion time of T_i, referred to as $ect(i)$, is the earliest possible completion time of all subtasks in T_i. To define $ect(i)$, we use $pest(i,j)$, which represents the earliest start time of j-th subtask ($0 \le j \le m_i$). If T_i is the task for a diagonal block, then

$$\begin{aligned} ect(i) &= pest(i, m_i) + W_{i,m_i}, \\ pest(i,0) &= est(i), \\ pest(i,j) &= \max\left(pest(i,j-1) + W_{i,j-1},\right. \\ &\qquad \left. ect(k_{j-1}) + c_{k_{j-1},i}\right), \\ pest(i,m_i) &= pest(i, m_i - 1) + W_{i,m_i-1}. \end{aligned}$$

If T_i is the task for a rectangular block,

$$\begin{aligned} ect(i) &= pest(i, m_i) + W_{i,m_i}, \\ pest(i,0) &= est(i), \\ pest(i,j) &= \max\left(pest(i,j-1) + W_{i,j-1},\right. \\ &\qquad \max\left(ect(k_{2j}) + c_{k_{2j},i},\right. \\ &\qquad\qquad \left.\left. ect(k_{2j+1}) + c_{k_{2j+1},i}\right)\right), \\ pest(i,m_i) &= \max(pest(i, m_i - 1) + W_{i,m_i-1}, \\ &\qquad ect(k_{2m_i}) + c_{k_{2m_i},i}). \end{aligned}$$

Level of a Task: The level of T_i is the length of the longest path from T_i to the exit task, including the communication costs along that path. The level of T_i corresponds to the worst-case remaining time of T_i. The level is used for the priority of T_i in task clustering. Level is defined as follows.

$$level(i) = \begin{cases} W_i & \text{if } child(i) = \emptyset, \\ \max\limits_{k \in child(i)} (level(k) + c_{i,k}) + W_i & \text{otherwise.} \end{cases}$$

4.2 Early-Start Clustering

The proposed early-start clustering (ESC) algorithm reduces the total completion time of all tasks by preserving the earliest start time of each task. ESC uses the level of a task as its priority so that a task on the critical path of a block dependency DAG can be examined earlier than other tasks. Each task is allowed to be clustered with only one of its children to preserve maximum parallelism.

```
1.  $EG = \emptyset,\ UEG = V,\ CL = \emptyset$.
2.  compute $level$ for each task.
3.  add all free entry tasks to $FL$.
4.  set $est(i) = 0$ for all $T_i \in FL$.
5.  while $UEG \ne \emptyset$ do
6.      $T_i = head(FL)$.
7.      find the parent $T_{k_j}$ that satisfies
            $ect(k_j) = \min_{k \in parent(i), k \notin CL} (ect(k))$.
8.      if $k_j$ is found, then
9.          $CLUST(k_j) = CLUST(k_j) \cup \{T_i\}$.
10.         $CLUST(i) = \emptyset$.
11.         $CL = CL \cup \{T_i\}$.
12.         $c_{k_j,i} = 0$.
13.     else
14.         $T_i$ remains in a unit cluster.
15.     endif
16.     sort $m_i$ subtasks of $T_i$, and calculate $ect(i)$.
17.     $EG = EG \cup \{T_i\}$, $UEG = UEG - \{T_i\}$.
18.     for each $T_k \in child(i)$
19.         if $\forall_{j \in parent(k)} T_j \in EG$, then $FL = FL \cup \{T_k\}$.
20. endwhile
```

Figure 6: The early-start clustering algorithm.

The ESC algorithm uses the lists EG, UEG, CL, and FL. EG contains the examined tasks. UEG is for unexamined tasks. CL is a list of the tasks clustered with one of its children so that a task in CL cannot be clustered with other children. FL is a list of all free tasks maintained by a priority queue in UEG so that the highest level task can be examined earlier than others. CLUST(T_i) refers to the cluster for task T_i. The complete ESC algorithm is described in Figure 6.

PROPERTY 1. *The ESC algorithm guarantees the maximum degree of concurrency.*

PROOF. Given a block dependency graph G, let k be the maximum degree of concurrency. This means that, at most, k tasks can run independently and concurrently. Let us denote these k tasks as $v_1, .., v_k$. Consider the case that a node v_i ($1 \le i \le k$) is clustered with any one of its parents. If the parent is already clustered with v_j ($1 \le j \le k, i \ne j$), then the degree of concurrency becomes $k - 1$. Otherwise, the degree of concurrency does not change. Since the ESC algorithm does not allow two tasks to be clustered with the same parent, the maximum degree of concurrency does not decrease while all k tasks to be examined are clustered with one of their parents.

Let us consider the situation when ESC examines each child of the k tasks. Any one child can be clustered with one of the k tasks, and none of the k tasks are allowed to be clustered with two children. Therefore, the degree of concurrency does not change after examining all children by the ESC algorithm. Thus, the ESC algorithm guarantees the maximum degree of concurrency. $\square$

The time complexity of ESC is as follows. To calculate the levels of tasks in Step 2, tasks are visited in post-order, and all edges are examined. Hence, the complexity of Step 2 is $O(e)$. In the while loop, the most time consuming part is the calculation of $ect(i)$ in Step 16, which sorts all subtasks with respect to the ready time of each subtask. Since the number of subtasks in a task is not larger than N, sorting of subtasks takes $O(N\log(N))$ time. The complexity of Step 16 is $O(vN\log(N))$ for v iterations. Thus, the overall complexity of the ESC algorithm is $O(e + vN\log(N))$.

4.3 Affined Cluster Mapping

The most common cluster mapping method is based on a *work profiling method* [10; 17; 27], which is designed only for load balancing based on the work required for each cluster. However, by considering the required communication costs and preserving parallelism whenever possible, we can further improve the performance of block Cholesky factorization.

We introduce the affined cluster mapping (ACM) algorithm, which uses the affinity as well as the work required for a cluster. The affinity of a cluster to a processor is defined as the sum of the communication costs required when the cluster is mapped to other processors. We classify the types of unmapped clusters into *affined clusters*, *free clusters*, and the *heaviest cluster.*

- **Affined cluster:** If any task in a cluster needs to communicate with the tasks allocated to processor p, then the cluster is an affined cluster of processor p. The affinity of the cluster is the sum of communication costs between the tasks in the cluster and the tasks allocated to processor p.
- **Free cluster:** If a cluster has an entry task, the cluster is a free cluster. Such a free cluster allows a processor to execute the entry task without waiting. Note that an affined cluster can be a free cluster.
- **Heaviest cluster:** If the sum of the computation costs of tasks in a cluster is the largest among all unmapped clusters, then that cluster is the heaviest cluster.

ACM finds the processor with the minimum amount of work, and allocates a cluster found by the search sequence. (The cluster with the highest affinity among affined clusters to the processor, or the heaviest cluster among free clusters, or the heaviest cluster among all unmapped clusters.) Initially, all processors have no tasks. Therefore, the heaviest free cluster is allocated first. If the processor with the minimum work has some clusters already allocated, then ACM checks whether there are affined clusters with the processor. If not, ACM checks free clusters. If there are no affined or free clusters, then the heaviest cluster among unmapped clusters is allocated to the processor.

The ACM algorithm uses the lists $CLUST$, MC, UMC, and FC. $CLUST$ means the entire set of clusters, and $CLUST(T_i)$ represents the cluster containing task T_i. MC is a list of the mapped clusters, and UMC is a list of the unmapped clusters. Thus $MC \cup UMC = CLUST$. FC is a list of the free clusters. UMC and FC are maintained as priority queues with the heaviest cluster first. Each cluster has a list of affinity values for all processors. The complete ACM algorithm is described in Figure 7.

The time complexity of ACM is analyzed as follows. Let P denote the number of processors. Since the number of cluster is less than the number of tasks, the number of cluster is bounded by $O(v)$. In Steps 3$\sim$5, $O(vP)$ time is required for initialization. In the while loop, Step 7 takes $O(P)$ time and Step 8 takes $O(v)$ time; since the loop iterates for each cluster, these two steps take $O(v^2 + vP)$ time. Steps 14 $\sim$ 20 are for updating the affinity values for each unmapped cluster communicating with the current cluster. In the worst case, all edges are traced twice during the whole loop, for checking parents and for checking children. Therefore, the complexity for updating affinities is $O(2e)$. The overall complexity of the ACM algorithm is thus $O(vP + v^2 + e)$.

```
1.  $MC = \emptyset$, $UMC = CLUST$, $FC = \emptyset$.
2.  add free clusters in $UMC$ to $FC$.
3.  for each $C_i \in UMC$ do
4.      for $p = 0$ to $P - 1$ do
5.          set $C_i$'s affinity[$p$] $= 0$.
6.  while $UMC \neq \emptyset$ do
7.      find the processor $p$ with the smallest work.
8.      find the cluster $C_i$ that has the highest affinity[$p$].
9.      if $C_i = NULL$ and $FC \neq \emptyset$ then
10.         $C_i = head(FC)$, $FC = FC - \{C_i\}$.
11.     else$C_i = head(UMC)$.
12.     allocate $C_i$ to processor $p$.
13.     $MC = MC \cup \{C_i\}$, $UMC = UMC - \{C_i\}$
14.     for each $T_j \in C_i$ do
15.         for each $T_k \in child(j)$ do
16.             if $CLUST(T_k) \in UMC$ then
17.                 $CLUST(T_k)$'s affinity[$p$] = affinity[$p$] $+ c_{j,k}$.
18.         for each $T_k \in parent(j)$ do
19.             if $CLUST(T_k) \in UMC$ then
20.                 $CLUST(T_k)$'s affinity[$p$] = affinity[$p$] $+ c_{k,j}$.
21. endwhile
```

Figure 7: The affined cluster mapping algorithm.

5. PERFORMANCE COMPARISON

The performance of the proposed scheduling method is compared with various mapping methods. The methods used for comparison are as follows.

- **wrap**: 1-D wrap mapping [11] simply allocates all blocks in column j, i.e., $L_{*,j}$, to the processor (j *mod* P).
- **cyclic**: 2-D cyclic mapping [20] allocates $L_{i,j}$ to the processor (i *mod* P_r, j *mod* P_c).
- **balance**: Balance mapping [21] attempts to balance the workload among processors. For ordering in row and column mapping, we use the decreasing number heuristic.
- **dag**: A general DAG-based task scheduling method is applied to block dependency graphs. The scheduling method uses the *dominant sequence clustering* algorithm [28] for task clustering, and the *work profiling method* [10; 17; 27] is used for load-balanced cluster mapping.
- **schedule**: The task scheduling method proposed in this paper.

The test sparse matrices are taken from the Harwell-Boeing matrix collection [7], which is widely used for evaluating sparse matrix algorithms. All matrices are ordered using the multiple minimum degree ordering [14], which is the best

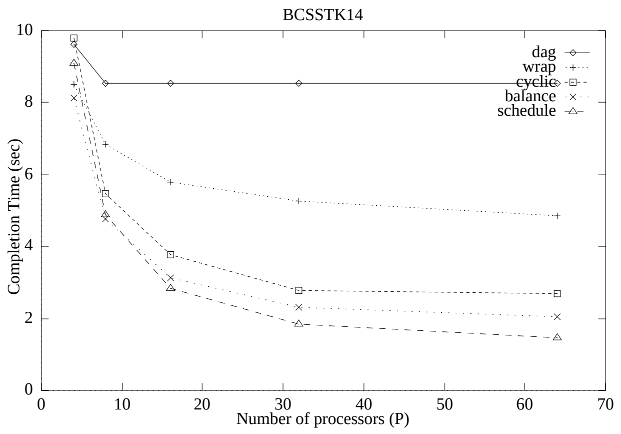

Figure 8: Completion time comparison for BCSSTK14 ($n = 1806, N = 159$).

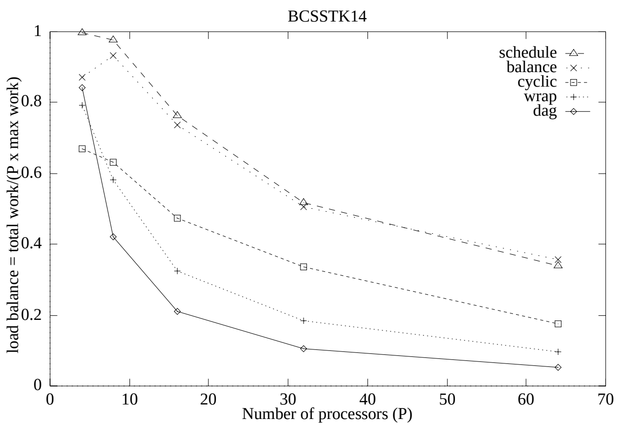

Figure 9: Load balance comparison for BCSSTK14.

ordering method for general or unstructured problems. Supernode amalgamation [2] is applied to the ordered matrices to improve the efficiency of block operations.

The parallel block fan-out method [20] is implemented on the Fujitsu AP1000+ multicomputer. The AP1000+ system is a distributed-memory MIMD machine with 64 cells. Each cell is a SuperSPARC processor with 64MB of main memory. The processors are inter-connected by a torus network with 25MB/sec/port. For the block operations, we used the single processor implementation of BLAS primitives taken from NETLIB. In order to measure communication costs, we use $t_s = 500$ and $t_c = 20$, which are estimated from the theoretical peak performance and benchmark tests.

The completion times of BCSSTK14 using various methods are shown in Figure 8. As P increases, the proposed method, *schedule*, performs well compared with the other methods. The completion time of *dag* does not decrease even though P increases. This is mainly due to the fact that one cluster includes all the tasks in the critical path of a block dependency DAG, and the huge cluster limits the completion time.

Figure 9 shows the load balance for all methods. The metric of load balance is

$$\textit{load balance} = \frac{\textit{total work}}{P \cdot \textit{max work}},$$

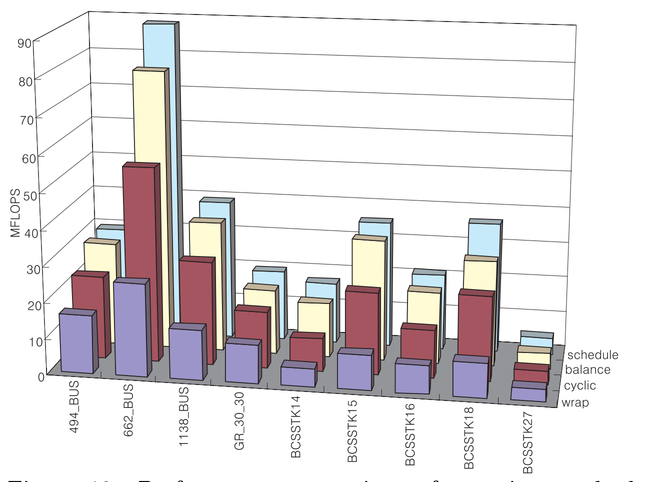

Figure 10: Performance comparison of mapping methods (MFLOPS).

where *total work* is the total work of all tasks and *max work* is the maximum amount of work assigned to any processor. As P increases, *load balance* decreases in all methods. Among them, *balance* and *schedule* keep the load well balanced. However, as shown in Figure 8, the completion time of *schedule* is shorter than that of *balance*. This shows that the best load balance does not guarantee the highest performance.

The performance comparison of *wrap, cyclic, balance, schedule* are shown in Figure 10. In the average case, *cyclic* is 1.9 times faster than *wrap*. Also, *balance* and *schedule* are 2.5 and 2.8 times faster than *wrap*, respectively. This means that *balance* takes 31.6% less time than *cyclic*, and *schedule* takes 12.0% less time than *balance*. In the best case, *schedule* takes 30% less time than *balance*. Since the experiments have been conducted on a 2-D mesh topology, *cyclic* and *balance* can take the full advantage of their algorithmic properties. If experiments were conducted on other topologies, the gap between the performance of the proposed schedule algorithm and that of *cyclic* and *balance* would be larger. From the overall comparison, it can be seen that the proposed scheduling algorithm has the best performance.

6. CONCLUSION

We introduced a task scheduling approach for block-oriented sparse Cholesky factorization on a distributed-memory system. The block Cholesky factorization problem is modeled as a block dependency DAG, which represents the execution behavior of 2-D decomposed blocks. Using a block dependency DAG, we proposed a task scheduling algorithm consisting of *early-start clustering* and *affined cluster mapping*. Based on experimental results using the Fujitsu AP1000+ multicomputer, we have shown that the proposed scheduling algorithm outperforms previous processor mapping methods. Also, when we applied a previous DAG-based task scheduling algorithm to a block dependency DAG, all tasks in a critical path were clustered into a huge cluster. Thus, the completion time is always bounded by the huge cluster even when the number of processors is increased. However, the proposed scheduling algorithm has good scalability, so that its performance improves progressively as the number of processors increases. The main contribution of this work is the introduction of a task scheduling approach with a refined task graph for sparse block Cholesky factorizations.

Acknowledgment

We would like to thank Cleve Ashcraft for his valuable comments on this work. In addition, he provided us with his technical reports and the SPOOLES library, which is used for ordering and supernode amalgamation. We also give special thanks to Fujitsu Laboratories Ltd. for providing access to their facilities, and especially to Takao Saito for supporting immediate help with many urgent requests.

7. REFERENCES

[1] C. C. Ashcraft. The domain/segment partition for the factorization of sparse symmetric positive definite matrices. Technical report, Boeing Computer Services, Seattle, Washington, 1990. ECA-TR-148.

[2] C. C. Ashcraft and R. G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, 15(4):291–309, Dec. 1989.

[3] C. C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Int'l J. Supercomputer Appl.*, 1(4):10–30, 1987.

[4] J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16(1):1–17, 1990.

[5] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, 1998.

[6] I. S. Duff. Sparse numerical linear algebra: Direct methods and preconditioning. Technical report, CERFACS, Toulouse Cedex, France, 1996. TR/PA/96/22.

[7] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[8] B. Dumitrescu, M. Doreille, J.-L. Roch, and D. Trystram. Two-dimensional block partitionings for the parallel sparse cholesky factorization. *Numer. Algorithms*, 16(1):17–38, 1997.

[9] C. Fu and T. Yang. Run-time techniques for exploiting irregular task parallelism on distributed memory architectures. *J. of Parallel and Distributed Computing*, 42:143–156, 1997.

[10] A. George, M. Heath, and J. Liu. Parallel cholesky factorization on a shared memory processor. *Lin. Algebra Appl.*, 77:165–187, 1986.

[11] A. George, M. Heath, J. Liu, and E. G. Ng. Sparse cholesky factorization on a local memory multiprocessor. *SIAM J. Sci. Stat. Comput.*, 9:327–340, 1988.

[12] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. on Parallel and Distrib. Systems*, 8(5):502–520, May 1997.

[13] M. T. Heath, E. G. Y. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, pages 420–460, 1991.

[14] J. W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. on Math. Software*, 11:141–153, 1985.

[15] E. G. Ng and B. W. Peyton. Block sparse cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14(5):1034–1056, 1993.

[16] E. G. Ng and B. W. Peyton. A supernodal cholesky factorization algorithm for shared-memory multiprocessors. *SIAM J. Sci. Comput.*, 14(4):761–769, 1993.

[17] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. New York: Plenum, 1988.

[18] E. Rothberg. Performance of panel and block approaches to sparse cholesky factorization on the iPSC/860 and paragon multicomputers. *SIAM J. Sci. Comput.*, 17(3):699–713, May 1996.

[19] E. Rothberg and A. Gupta. The performance impact of data reuse in parallel dense cholesky factorization. Technical report, Stanford University, 1992.

[20] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. *SIAM J. Sci. Comput.*, 15(6):1413–1439, Nov. 1994.

[21] E. Rothberg and R. Schreiber. Improved load distribution in parallel sparse cholesky factorization. In *Proc. of Supercomputing'94*, pages 783–792, 1994.

[22] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, Cambridge (MA), 1989.

[23] R. Schreiber. *Scalability of sparse direct solvers*, volume 56 of *The IMA Volumes in Mathematics and its applications*, pages 191–209. Springer-Verlag, New York, 1993.

[24] K. Shen, X. Jiao, and T. Yang. Elimination forest guided 2D sparse LU factorization. In *Proc. of ACM Symp. on Parallel Algorithm and Architecture*, pages 5–15, 1998.

[25] B. Veltman, B. Lageweg, and J. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.

[26] T. Yang and C. Fu. Space/time-efficient scheduling and execution of parallel irregular computations. *ACM Trans. Prog. Lang. Sys.*, 20(6):1195–1222, Nov. 1998.

[27] T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message passing multiprocessors. In *Proc. of 6th ACM Int'l Conf. Supercomputing*, pages 428–437, 1992.

[28] T. Yang and A. Gerasoulis. DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distrib. Systems*, 5(9):951–967, 1994.