Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Parallel Computing 29 (2003) 135–159

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Task scheduling using a block dependency DAG for block-oriented sparse Cholesky factorization ☆

Heejo Lee [a,*], Jong Kim [b], Sung Je Hong [b], Sunggu Lee [c]

[a] *Ahnlab, Inc., 8F V-Valley Bldg., 724 Suseo-dong, Gangnam-gu Seoul 135-744, South Korea*
[b] *Department of Computer Science and Engineering, Pohang University of Science and Technology, Pohang 790-784, South Korea*
[c] *Department of Electrical Engineering, Pohang University of Science and Technology, Pohang 790-784, South Korea*

Received 11 April 2002; accepted 17 September 2002

Abstract

Block-oriented sparse Cholesky factorization decomposes a sparse matrix into rectangular subblocks; 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 and reduces the overall communication volume 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 represents the execution behavior of block sparse Cholesky factorization in a distributed-memory system. Since the characteristics of tasks for 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 (ACM). The early-start clustering stage is used to cluster tasks while preserving the earliest start time of a task without limiting parallelism. After task clustering, the ACM stage allocates clusters to processors considering both communication cost and load balance. Experimental results on

☆ This research was supported in part by the Ministry of Education of Korea through its BK21 program toward the Electrical and Computer Engineering Division at POSTECH.

* Corresponding author. Address: Ahnlab, Inc., 8F V-Valley Bldg., 724 Suseo-dong, Gangnam-gu Seoul 135-744, South Korea.

E-mail addresses: heejo@ahnlab.com (H. Lee), jkim@postech.ac.kr (J. Kim), sjhong@postech.ac.kr (S.J. Hong), slee@postech.ac.kr (S. Lee).

0167-8191/02/$ - see front matter
PII: S0167-8191(02)00220-X

a Myrinet cluster system show that the proposed task scheduling approach outperforms other processor mapping methods.

Keywords: Task scheduling; Parallel sparse matrix factorization; Block-oriented Cholesky factorization; Directed acyclic graph

1. Introduction

Sparse Cholesky factorization is a computationally 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 factorization methods 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 [8,17]. In the supernodal Cholesky, a supernode, which is a group of consecutive columns with the same row structure, is mapped to a single processor [5,26]. The amalgamated supernodal Cholesky uses the supernode amalgamation technique in which several small supernodes are merged into a larger supernode, and an amalgamated supernode is then mapped to a single processor [4,30]. In the 2-D block Cholesky, a matrix is decomposed into rectangular blocks, and a block is mapped to a single processor [10,31].

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) [7,8]. Such a 2-D decomposition is more scalable than a 1-D decomposition and has an increased degree of concurrency [34,35]. Also, the 2-D decomposition allows one to use efficient computation kernels such as Level 3 BLAS so that caching performance is improved [30]. Even in a single processor system, block factorizations are performed efficiently [25].

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 [31]. Similarly, Dumitrescu et al. introduced the block fan-in algorithm [10]. Gupta, Karypis, and Kumar [16] also used 2-D mapping for implementing a multifrontal method. In [29], 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 concurrency and reduces overall communication volume, the performance achieved is not satisfactory due to load imbalance among the processors. Therefore, several load balancing heuristics have been proposed in [32].

However, the load balance is not the sole key parameter for improving the performance of parallel block sparse Cholesky factorization. Mapping for load balance 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 exact computation and communication costs. 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 adversely affect performance [37]. But we do not consider the memory space limitations in this paper, 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 have been recent efforts for scheduling irregular computations on parallel systems [11,18,19], 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 those of 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 a Myrinet cluster 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 [3,20,29,31]. 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 [4,5]. 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 cause non-zero blocks to become as dense as possible and easy to handle due to shared common boundaries [31].

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 [1,4,31]. Supernode amalgamation is a process of identifying the locations of zero elements that would produce larger supernodes if they were treated as non-zeros. On the other hand, large supernodes are splitted by the maximum allowable supernode size in order to exploit concurrency within dense block computations [10,29]. In the following, such an amalgamated supernode will be assumed by default, and will be referred to simply as a supernode.

In a given $n \times n$ sparse matrix with N supernodes, the supernodes divide the columns of the matrix $(1,\ldots,n)$ into contiguous subsets $(\{1,\ldots,p_2-1\},\{p_2,\ldots,p_3-1\},\ldots,\{p_N,\ldots,n\})$. The size of the ith 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 submatrix decomposed by supernode i and supernode j. Then, the row numbers of elements in $L_{i,j}$ are in $\{p_i,\ldots,p_{i+1}-1\}$, and the column numbers of elements in $L_{i,j}$ are in $\{p_j,\ldots,p_{j+1}-1\}$.

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 $N(N-1)/2$ off-diagonal 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. A non-zero off-diagonal rectangular block is referred to as a rectangular block.

After block decomposition using supernodes, the resulting structure is quite regular [31]. 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.

2.2. Block Cholesky factorization

The sequential algorithm for block Cholesky factorization, as described in [31], is shown in Fig. 1. The algorithm works with the blocks decomposed by supernodes to

```
1. for k = 1 to N do
2.     L_kk = Factor(L_kk)
3.     for i = k + 1 to N with L_ik ≠ 0 do
4.         L_ik = L_ik L_kk^{-1}
5.     for j = k + 1 to N with L_jk ≠ 0
6.         for i = j to N with L_ik ≠ 0 do
7.             L_ij = L_ij − L_ik L_jk^T
```

Fig. 1. Sequential block Cholesky factorization.

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 [6]. Such block computations require no indirect addressing, which leads to enhanced caching performance and close to peak performance on modern computer architectures [7].

2.3. Block operations

Let us denote the dense Cholesky factorization of a diagonal block L_{kk} (Step 2 in Fig. 1) as $\mathrm{bfact}(k)$. Similarly, let us denote the operation of Step 4 as $\mathrm{bdiv}(i,k)$, and the operation of Step 7 as $\mathrm{bmod}(i,j,k)$. 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 is rarely required [29]. Therefore, we assume that all block operations are handled with dense matrix operations.

For $\mathrm{bfact}(k)$, an efficient dense Cholesky factorization, such as _POTRF( ) in LAPACK, can be used. Also, $\mathrm{bdiv}(i,k)$ and $\mathrm{bmod}(i,j,k)$ are supported by 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 [7].

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

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

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

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

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

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 one processor [31], 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 block factorization. The task graph, referred to as a block dependency DAG, reflects 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., $\mathrm{nmod}(L_{j,j}) = m$, its corresponding task has $m+1$ subtasks including a $\mathrm{bfact}(j)$ operation. Then, the task has m parent tasks as shown in Fig. 2. Let us denote the m blocks of parent tasks as $L_{j,k_0}, \ldots, L_{j,k_{m-1}}$, $1 \leqslant k_i \leqslant j-1$ for $0 \leqslant i \leqslant m-1$. If there is no block update required for $L_{j,j}$, i.e., $\mathrm{nmod}(L_{j,j}) = 0$, then the task has no parent task and only one subtask for $\mathrm{bfact}(j)$.

If a rectangular block $L_{i,j}$ requires m block updates, i.e., $\mathrm{nmod}(L_{i,j}) = m$, then the corresponding task consists of $m+1$ subtasks including the subtask for the $\mathrm{bdiv}(i,j)$ operation. The task has $2m+1$ parents as shown in Fig. 3. Let us refer to the $2m+1$ parent blocks as $L_{i,k_0}, L_{j,k_0}, \ldots, L_{i,k_{m-1}}, L_{j,k_{m-1}}, L_{j,j}$. A subtask for a block update $\mathrm{bmod}(i,j,k)$ 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 $\mathrm{bmod}(j,j,k)$ operation.

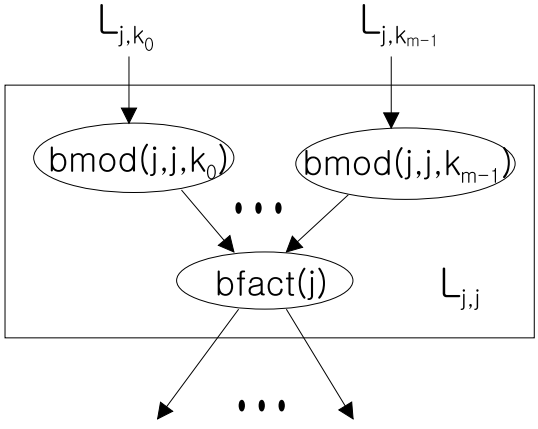

Fig. 2. Task for a diagonal block $L_{j,j}$.

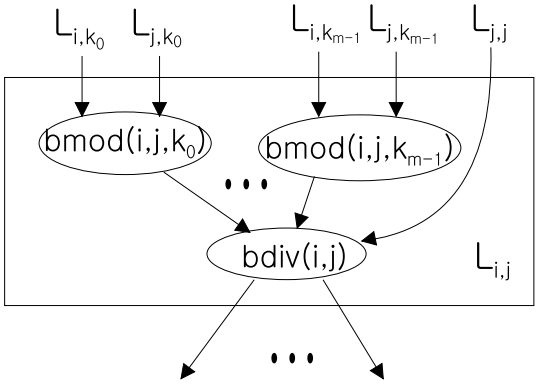

Fig. 3. Task for a rectangular block $L_{i,j}$.

3.2. Task graph

We now present a task graph for the block fan-out (right-looking) Cholesky factorization. The task graph, referred to as the block dependency DAG, reflects the precedence relationships among the blocks and the computation and 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 $T_1, \ldots, T_v$ and

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

We assign 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 enumerated 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 $\{T_1, \ldots, T_v\}$ 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

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

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

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

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 [7].

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.

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

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 resolving this situation. One is designing a task model which includes 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. Thus, this task model is difficult to handle using a scheduling algorithm. The latter uses a simple, precise task graph as a block dependency DAG. Nevertheless, we can 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.

3.3. Task execution behavior on previous block mapping methods

Using the task model, we can estimate the performance of the previous block mapping methods. The methods are 2-D block cyclic mapping [31] and load balance mapping [32]. As an example, we use a 70×70 sparse matrix decomposed into five supernodes as shown in Fig. 4. The shaded blocks are non-zero blocks, and their task numbers are counted lexicographically such that tasks of $L_{11}, L_{31}, L_{51}, \ldots, L_{55}$ are $T_1, T_2, T_3, \ldots, T_{10}$, respectively. The corresponding task graph of the given sparse matrix is shown in Fig. 5. We use $t_s = 0, t_c = 10$ for the communication costs.

3.3.1. 2-D cyclic mapping

2-D cyclic mapping uses the index of $L_{i,j}$ for mapping the block to a processor. On a logical $(P_r \times P_c)$ mesh, the mapping function is as follows:

$$\text{map}(L_{i,j}) = (i \bmod P_r, j \bmod P_c).$$

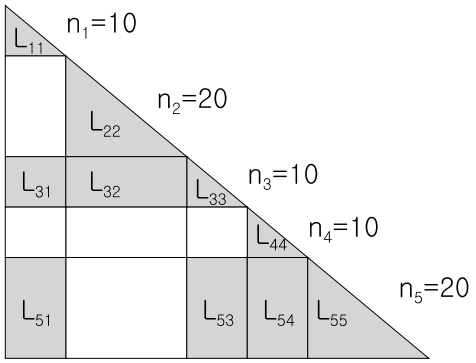

Fig. 4. Sparse matrix decomposed using five supernodes.

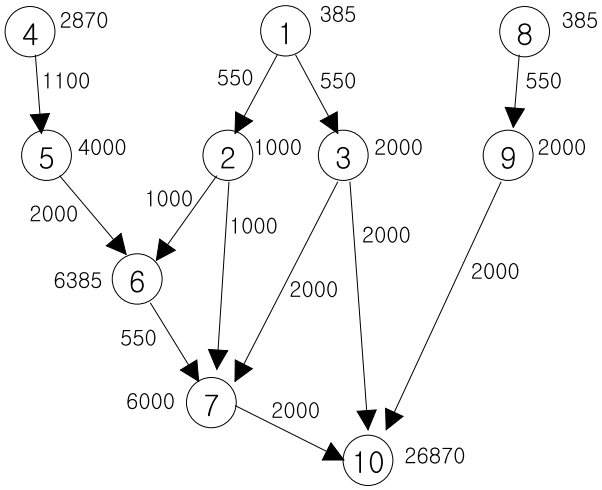

Fig. 5. Block dependency DAG for the example sparse matrix.

Fig. 6 shows the mapping result of cyclic mapping on 2×1 processors. P_0 stands for processor $(0,0)$ and P_1 for processor $(1,0)$. The mapping is quite unbalanced between the two processors. P_1 has 8 blocks, but P_0 has only 2 blocks. The total completion time of the cyclic mapping is 49 225, as shown in Fig. 7.

3.3.2. Load balanced mapping

To improve the load imbalance, block mapping heuristics have been proposed [32]. The block mapping heuristics are based on the *Cartesian product* that maps a column of blocks to a column of processors and a row of blocks to a row of processors. Then the communication patterns become quite restricted, in that a block needs

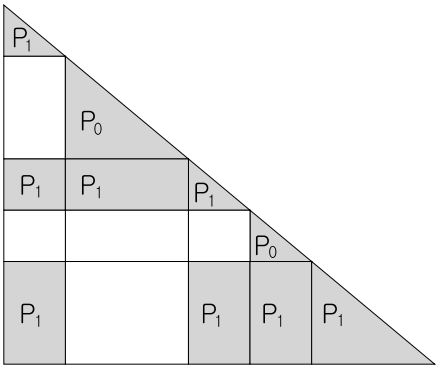

Fig. 6. 2-D cyclic block mapping.

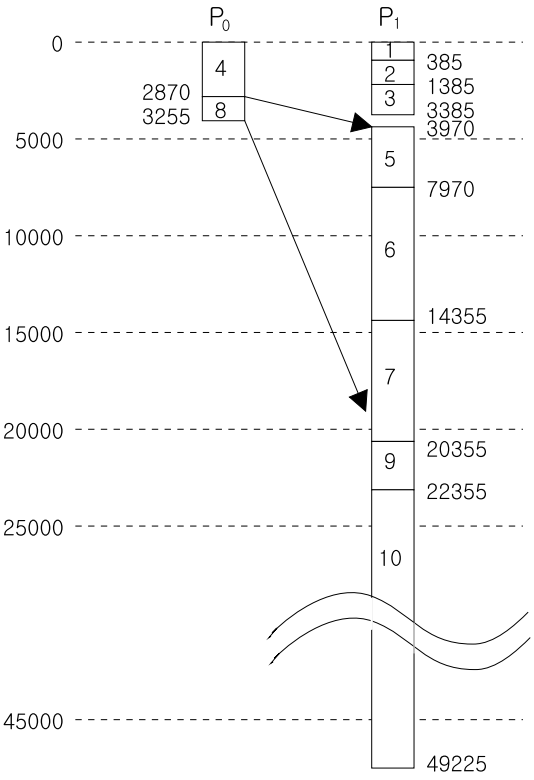

Fig. 7. The completion time by 2-D cyclic mapping.

to be sent to $O(\sqrt{P})$ processors on a P processor system ($P = P_r \times P_c$). The mapping function based on the Cartesian product is as follows:

$$\mathrm{map}(L_{ij}) = (\mathrm{map}I(i), \mathrm{map}J(j)),$$

where

$$\mathrm{map}I : \{1 \ldots N\} \to \{0 \ldots P_r - 1\} \quad \text{and} \quad \mathrm{map}J : \{1 \ldots N\} \to \{0 \ldots P_c - 1\}.$$

The mapping function, map$I$ or map$J$, for blocks can be determined by several heuristics: decreasing work, increasing number, decreasing number, and increasing depth. Based on such a strategy, each heuristic algorithm maps a block to the processor with the minimum amount of work. Among such algorithms, algorithms based on the decreasing number heuristic and the increasing depth heuristic usually balance better than the other two methods [32].

When load balanced mapping with the decreasing number heuristic is applied to the sparse matrix in Fig. 4, the mapping result shown in Fig. 8 is obtained. In this figure, Work$I[i]$ represents the total work of blocks in row i, and Work$J[j]$ represents the total work of blocks in column j. Mapping by other heuristics are all the same for this example. Although the mapping greatly improves the load balance, the completion time is not reduced by very much. The completion time by load balanced mapping is 46 060 as shown in Fig. 9. Since there is a lot of waiting time, the completion time is reduced by only 6.5% from the cyclic mapping result.

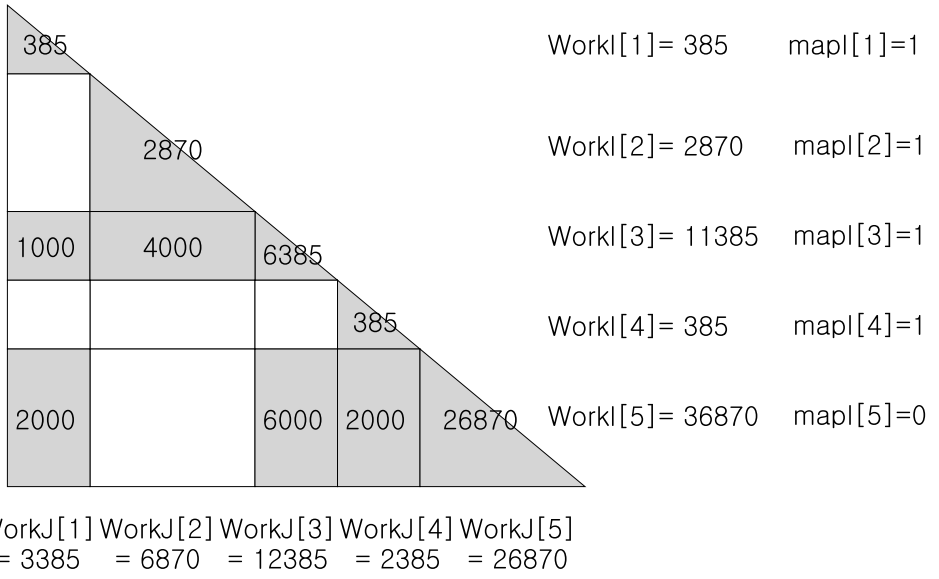

Fig. 8. Illustration of load balance mapping. (For column mapping, map$J[j] = 0$ for all $j = 1, \ldots, 5$. For row mapping, map$I[5] = 0$, and 1 for others.)

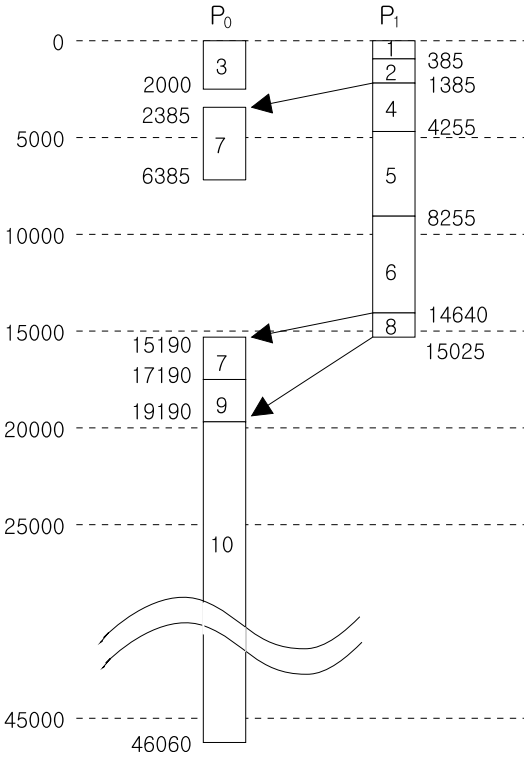

Fig. 9. The completion time by load balance mapping.

4. Task scheduling using a block dependency DAG

The problem of finding the optimal solution for a weighted DAG is known to be NP-hard in the strong sense [33,36]. When each task in a block dependency graph consists of only one or two subtasks, the scheduling problem using the block dependency DAG reduces to the NP-hard scheduling problem. Thus, finding an optimal scheduling of tasks in a block dependency DAG is an NP-hard problem. Therefore, 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 [14,15,21,28,33,39]. 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 method of mapping clusters to processors, called affined cluster mapping (ACM), attempts 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 for a given block dependency DAG, include the work required for each subtask in a task, the parent information for subtasks, the earliest start time of a task, the earliest completion time of a task, and the level of a task.

4.1.1. Work and parents of subtasks

A task T_i requiring m_i block updates consists of $m_i + 1$ subtasks. We refer to the work for the $m_i + 1$ subtasks as $W_{i,0}, W_{i,1}, \ldots, W_{i,m_i}$. 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 $T_{k_0}, T_{k_1}, \ldots, T_{k_{m_i-1}}$. If T_i is the task for a rectangular block, there are $2m_i + 1$ parents, which are referred to as $T_{k_0}, T_{k_1}, \ldots, T_{k_{2m_i}}$.

4.1.2. 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, $\mathrm{est}(i)$, is defined recursively using the earliest completion time of the parent tasks. The earliest completion time of T_k is denoted by $\mathrm{ect}(k)$—$\mathrm{ect}(k)$ will be formally defined later in this section. If T_i is the task for a diagonal block, then

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

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

$$\mathrm{est}(i) = \begin{cases} \mathrm{ect}(k_0) + c_{k_0,i} & \text{if } \mathrm{parent}(i) = \{k_0\}, \\ \min_{0 \leqslant j \leqslant m-1}(\max(\mathrm{ect}(k_{2j}) + c_{k_{2j},i}, \mathrm{ect}(k_{2j+1}) + c_{k_{2j+1},i})) & \text{otherwise.} \end{cases}$$

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.

4.1.3. Earliest completion time of a task

The earliest completion time of T_i, referred to as $\mathrm{ect}(i)$, is the earliest possible completion time of all subtasks in T_i. To define $\mathrm{ect}(i)$, we use $\mathrm{pest}(i, j)$, which represents the earliest start time of a particular jth subtask in task i ($0 \leqslant j \leqslant m_i$). If T_i is the task for a diagonal block, then

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

If T_i is the task for a rectangular block,

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

4.1.4. 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:

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

4.2. Early-start clustering

The proposed early-start clustering (ESC) algorithm attempts to reduce 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.

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 Fig. 10.

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

Proof. Given a block dependency graph G, let k be the attainable maximum degree of concurrency. This means that, at most, k tasks can run independently and concurrently. Let us denote these k tasks as $v_1, \ldots, v_k$. Consider the case when a node v_i ($1 \leqslant i \leqslant k$) is clustered with any one of its parents. If the parent is already clustered with v_j ($1 \leqslant j \leqslant k, i \neq j$), then the degree of concurrency becomes $k - 1$. Otherwise,

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 $UEG \neq \emptyset$ do
6. $\quad T_i = head(FL)$.
7. $\quad$ find the parent T_{k_j} that satisfies $ect(k_j) = \min_{k \in parent(i),\ k \notin CL}(ect(k))$.
8. $\quad$ if k_j is found, then
9. $\quad\quad CLUST(k_j) = CLUST(k_j) \cup \{T_i\}$.
10. $\quad\quad CLUST(i) = \emptyset$.
11. $\quad\quad CL = CL \cup \{T_{k_j}\}$.
12. $\quad\quad c_{k_j,i} = 0$.
13. $\quad$ else
14. $\quad\quad T_i$ remains in a unit cluster.
15. $\quad$ endif
16. $\quad$ sort m_i subtasks of T_i, and calculate $ect(i)$.
17. $\quad EG = EG \cup \{T_i\},\ UEG = UEG - \{T_i\}$.
18. $\quad$ for each $T_k \in child(i)$
19. $\quad\quad$ if $\forall_{j \in parent(k)} T_j \in EG$, then $FL = FL \cup \{T_k\}$.
20. endwhile

Fig. 10. The early-start clustering algorithm.

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.

Next, 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 with the ESC algorithm. Thus, the ESC algorithm guarantees the maximum degree of concurrency. □

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 eventually 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* [12,27,38], 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.

In this subsection, we introduce the 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 *heaviest clusters*.

- *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 the 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 following search sequence: the cluster with the highest affinity among affined clusters to that processor, the heaviest cluster among the free clusters, and then 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 amount of work has some clusters already allocated to it, then ACM checks whether there are affined clusters for that processor. If not, ACM checks the free clusters. If there are no affined or free clusters, then the heaviest cluster among the unmapped clusters is allocated to that processor.

ACM uses the lists CLUST, MC, UMC, and FC. CLUST is the entire set of clusters, and $\mathrm{CLUST}(T_i)$ 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 $\mathrm{MC} \cup \mathrm{UMC} = \mathrm{CLUST}$. 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 Fig. 11.

The time complexity of ACM is analyzed as follows. Let P denote the total number of processors. Since the number of clusters is less than the number of tasks, the number of clusters is bounded by $\mathrm{O}(v)$. In Steps 3–5, $\mathrm{O}(vP)$ time is required for initialization. In the while loop, Step 7 takes $\mathrm{O}(P)$ time and Step 8 takes $\mathrm{O}(v)$ time; since the loop iterates for each cluster, these two steps take $\mathrm{O}(v^2 + vP)$ time. Steps 14–20 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

```
1.  MC = ∅, UMC = CLUST, FC = ∅.
2.  add free clusters in UMC to FC.
3.  for each C_i ∈ UMC do
4.      for p = 0 to P − 1 do
5.          set C_i's affinity[p] = 0.
6.  while UMC ≠ ∅ 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 ≠ ∅ 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 ∪ {C_i}, UMC = UMC − {C_i}
14.     for each T_j ∈ C_i do
15.         for each T_k ∈ child(j) do
16.             if CLUST(T_k) ∈ UMC then
17.                 CLUST(T_k)'s affinity[p] = affinity[p] + c_{j,k}.
18.         for each T_k ∈ parent(j) do
19.             if CLUST(T_k) ∈ UMC then
20.                 CLUST(T_k)'s affinity[p] = affinity[p] + c_{k,j}.
21. endwhile
```

Fig. 11. The affined cluster mapping algorithm.

loop of the ACM algorithm, once for checking the parents and once for checking the children. Therefore, the complexity for updating affinities is $O(2e)$. The overall complexity of the ACM algorithm is thus $O(vP + v^2 + e)$.

4.4. Running trace of the proposed scheduling algorithm

To illustrate the operation of the proposed scheduling algorithm, we use the example matrix in Fig. 4. The result of clustering by the ESC algorithm is shown in Fig. 12. After mapping by the ACM algorithm, the completion time is estimated and shown in Fig. 13. The proposed scheduling algorithm takes only 35 510 units of time. This is 1.39 times faster than 2-D cyclic mapping and 1.30 times faster than load balanced mapping.

Fig. 14 shows a larger DAG that maps blocks onto processors using the proposed scheduling method for the 100×100 R2D100 matrix. The sparse matrix R2D100 is generated by first randomly triangulating the unit square with 100 grid points, which is taken from the SPOOLES library [2]. Then, 38 blocks are decomposed using 15 supernodes. In this example, the load balanced mapping does not allocate any block to P_2. since the mapping allocates rows and columns to processors independently. The proposed scheduling method merges 38 blocks into 18 clusters, which results in the maximum degree of concurrency. Thus, the proposed scheduling method

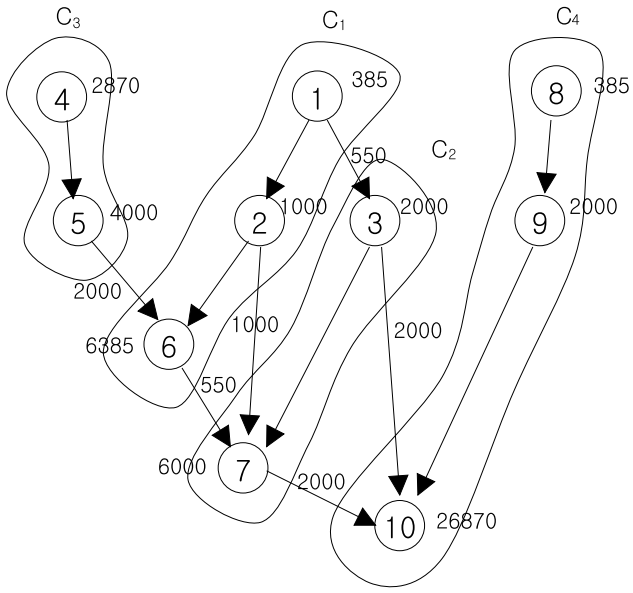

Fig. 12. Clustering by the ESC algorithm.

results in a better load balance with a higher degree of concurrency than load balanced mapping.

5. Performance comparison

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

- *Wrap*: 1-D wrap mapping [13] simply allocates all blocks in column j, i.e., $L_{*,j}$, to the processor $(j \bmod P)$.
- *Cyclic*: 2-D cyclic mapping [31] allocates $L_{i,j}$ to the processor $(i \bmod P_r, j \bmod P_c)$.
- *Balance*: Balance mapping [32] attempts to balance the workload among processors. The decreasing number heuristic is used for ordering within a row or column.
- *Schedule*: The task scheduling method proposed in this paper.

The test sparse matrices are mainly taken from the Harwell-Boeing matrix collection [9], which is widely used for evaluating sparse matrix algorithms. In addition, three sparse matrices are taken from the independent matrix sets in Matrix Market. The characteristics of the test sparse matrices are shown in Table 1. [1] All matrices are

[1] The number of operations (W) is measured for the non-zero block operation so that it is somewhat larger than the number of required operations only for non-zero elements. This number of operations is considered as a workload in scheduling.

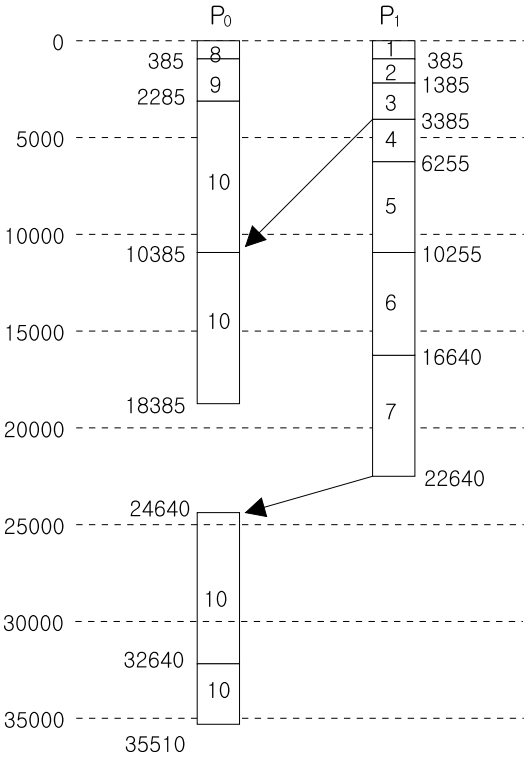

Fig. 13. Completion time by the proposed scheduling algorithm.

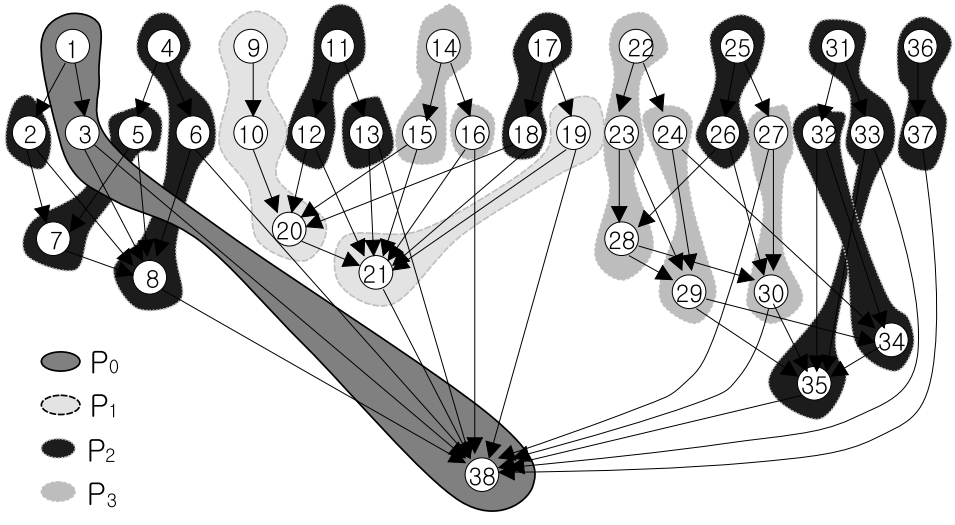

Fig. 14. Proposed clustering and four-processor mapping result for R2D100. The number of entry tasks is 10. The maximum degree of concurrency is 18. The number of clusters is 18.

Table 1
Benchmark sparse matrices

Matrix	n	$\|L\|$	N	v	W
662_bus	662	1568	65	191	2.07M
1138_bus	1138	2596	106	288	3.18M
gr_30_30	900	4322	67	212	4.00M
bcsstk14	1806	32 630	167	627	2.01M
bcsstk27	1224	28 675	73	308	9.68M
bcsstk15	3948	60 882	307	2311	207.81M
bcsstk16	4884	147 631	388	2682	297.82M
bcsstk25	15 439	133 840	1578	12 465	814.79M
s1rmt3m1	5489	112 505	343	1989	142.42M
s2rmq4m1	5489	143 300	351	2216	159.62M

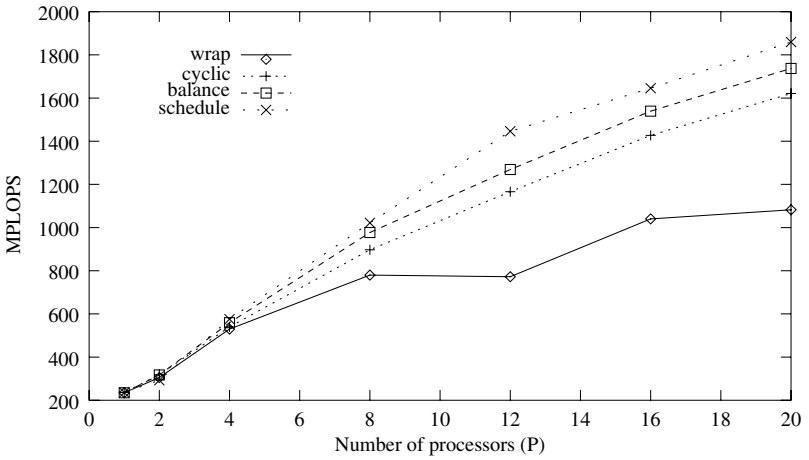

Fig. 15. MFLOPS performance over P for the "bcsstk15" matrix.

ordered using a generalized nested dissection ordering [24]. Supernode amalgamation [4] is applied to the ordered matrices to improve the efficiency of block operations. The amalgamation parameter used is $\max\_zero = 256$ and $\max\_size = 32$ using the SPOOLES library [2].

The parallel block fan-out method [31] is implemented with MPI on a Myrinet cluster system. The cluster system consists of 20 733 MHz Pentium-III PCs interconnected by high performance Myrinet switches and LANai 7 network interface cards. The unidirectional one-to-one bandwidth using the Myrinet network was measured to be 655 Mbps based on benchmark tests with MPI running on GM version 1.4. GM is a low-level message passing system for Myrinet systems developed by Myricom, the creator of Myrinet. For the block operations, we used a Pentium-III optimized BLAS library version 1.3e. [2] The peak performances of the Level 3 BLAS

[2] http://www.cs.utk.edu/~ghenry/distrib/index.html.

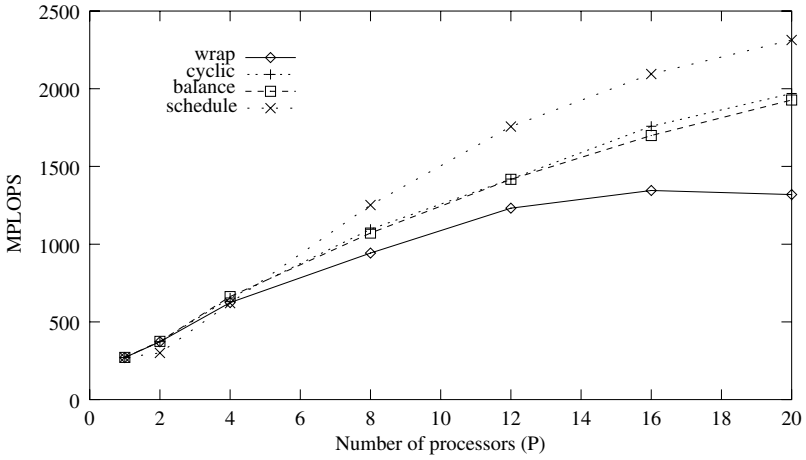

Fig. 16. MFLOPS performance over P for the "bcsstk16" matrix.

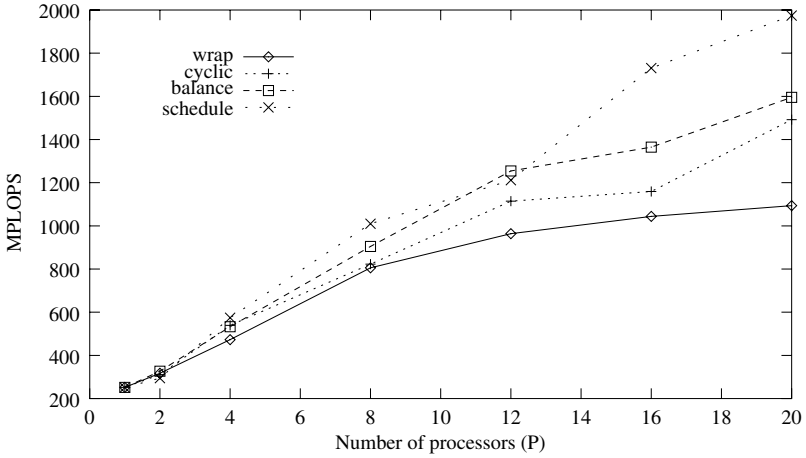

Fig. 17. MFLOPS performance over P for the "s1rmt3m1" matrix.

operators were measured to be 564 MFLOPS for DGEMM( ) and 228 MFLOPS for DTRSM( ) on a single processor machine of the Myrinet cluster. In order to model communication costs, we use $t_s = 10$ and $t_c = 1$, which were estimated from the benchmark tests.

The performance results when factorizing the four matrices, "bcsstk15", "bcsstk16", "s1rmt3m1", and "s2rmq4m1", are shown in Figs. 15–18, respectively. The MFLOPS performances of "wrap", "cyclic", "balance" and "schedule" as a function of P are shown in order from the lowest line in each figure. As the number of processors (P) increases, the proposed schedule outperforms the other methods and shows the best scalability with respect to the number of processors.

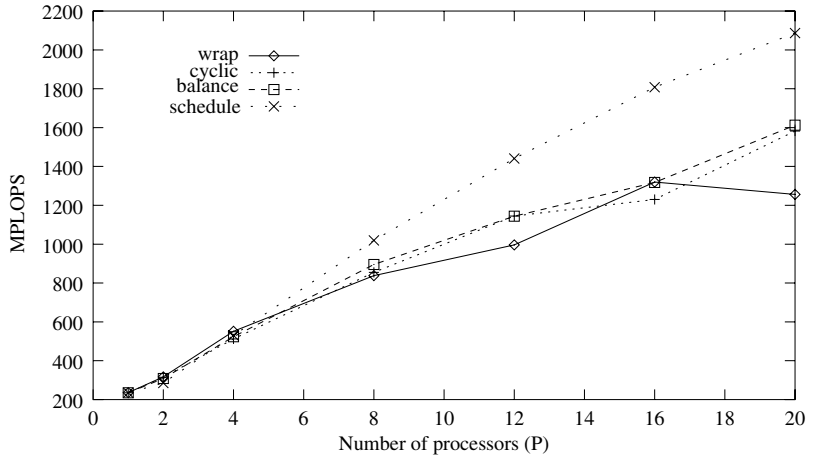

Fig. 18. MFLOPS performance over P for the "s2rmq4m1" matrix.

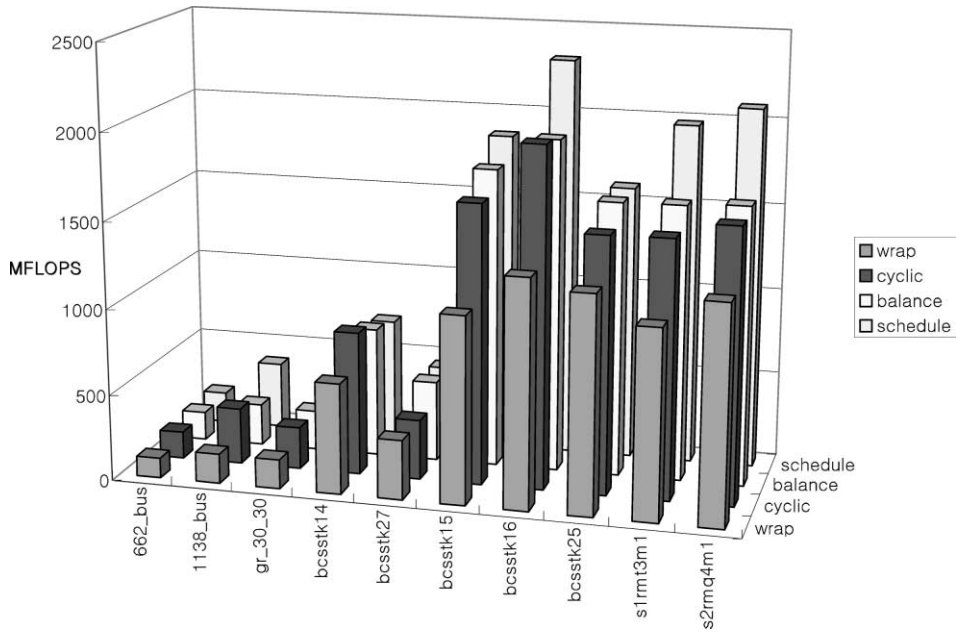

Fig. 19. Performance comparison.

The performance comparison of the four mapping methods on 20 processors is shown in Fig. 19. "schedule" outperforms three other mapping methods, especially with large matrices rather than small matrices. For instance, "schedule" reaches 2.3 GFLOPS with "bcsstk16", which performs 20% better than 'balance'. In the average case, 'schedule' performs 15% better than 'balance'.

From the overall comparison, we can deduce that the proposed scheduling method achieves the best performance on the Myrinet cluster. However, can we confirm that the proposed scheduling method also performs better in other computing environments? To partially answer this question, we also experimented with a lower-bandwidth Ethernet interface [22] and with the Fujitsu AP1000+ multicomputer [23]. Similar results were obtained on these platforms. As the ratio of communication cost to computation cost increases, and as the number of processors increases, the proposed scheduling method increasingly outperforms other mapping methods.

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 ACM. Based on experiments using a Myrinet cluster system, we have shown that the proposed scheduling algorithm outperforms previous processor mapping methods. Also, 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 factorization. We are currently working on applying this scheduling algorithm to sparse LU factorization. Since numerical factorization is the most computation intensive part of solving linear systems, we have focused on parallelizing numerical factorization. However, we also plan to study the use of our scheduling approach for solving triangular systems after numerical factorization.

Acknowledgements

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.

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, SPOOLES: An object-oriented sparse matrix library, In Proc. of 1999 SIAM Conference on Parallel Processing for Scientific Computing, March 1999.

[3] C.C. Ashcraft, S. Eisenstat, J. Liu, A. Sherman, A comparison of three column-based distributed sparse factorization schemes, Technical report, Department of Computer Science, Yale University, New Haven, CT, 1990, YALEU/DCS/RR-810.

[4] C.C. Ashcraft, R.G. Grimes, The influence of relaxed supernode partitions on the multifrontal method, ACM Trans. Math. Software 15 (4) (1989) 291–309.
[5] C.C. Ashcraft, R.G. Grimes, J.G. Lewis, B.W. Peyton, H.D. Simon, Progress in sparse matrix methods for large linear systems on vector supercomputers, Int. J. Supercomput. Appl. 1 (4) (1987) 10–30.
[6] J.J. Dongarra, J.D. Croz, I.S. Duff, S. Hammarling, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Software 16 (1) (1990) 1–17.
[7] J.J. Dongarra, I.S. Duff, D.C. Sorensen, H.A. van der Vorst, Numerical Linear Algebra for High Performance Computers, SIAM (1998).
[8] I.S. Duff, Sparse numerical linear algebra: Direct methods and preconditioning, Technical report, CERFACS, Toulouse Cedex, France, 1996. TR/PA/96/22.
[9] I.S. Duff, R.G. Grimes, J.G. Lewis, Sparse matrix test problems, ACM Trans. Math. Software 15 (1989) 1–14.
[10] B. Dumitrescu, M. Doreille, J.-L. Roch, D. Trystram, Two-dimensional block partitionings for the parallel sparse cholesky factorization, Numer. Algorithms 16 (1) (1997) 17–38.
[11] C. Fu, T. Yang, Run-time techniques for exploiting irregular task parallelism on distributed memory architectures, J. Parallel Distrib. Comput. 42 (1997) 143–156.
[12] A. George, M. Heath, J. Liu, Parallel cholesky factorization on a shared memory processor, Lin. Algebra Appl. 77 (1986) 165–187.
[13] A. George, M. Heath, J. Liu, E.G. Ng, Sparse cholesky factorization on a local memory multiprocessor, SIAM J. Sci. Stat. Comput. 9 (1988) 327–340.
[14] A. Gerasoulis, T. Yang, Comparison of clustering heuristics for scheduling DAGs on multiprocessors, J. Parallel Distrib. Comput. (1992) 276–291.
[15] A. Gerasoulis, T. Yang, On the granularity and clustering directed acyclic task graphs, IEEE Trans. Parallel Distrib. Syst. 4 (6) (1993) 686–701.
[16] A. Gupta, G. Karypis, V. Kumar, Highly scalable parallel algorithms for sparse matrix factorization, IEEE Trans. Parallel Distrib. Syst. 8 (5) (1997) 502–520.
[17] M.T. Heath, E.G.Y. Ng, B.W. Peyton, Parallel algorithms for sparse linear systems, SIAM Rev. (1991) 420–460.
[18] P. Henon, P. Ramet, J. Roman, A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization, in: EuroPAR'99, 1999, pp. 1059–1067.
[19] P. Henon, P. Ramet, J. Roman, PaStiX: A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions, in: Irregular'2000, 2000.
[20] L. Hulbert, E. Zmijewski, Limiting communication in parallel sparse cholesky factorization, SIAM J. Sci. Stat. Comput. 12 (1991) 1184–1197.
[21] Y.-K. Kwok, I. Ahmad, Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors, IEEE Trans. Parallel Distrib. Syst. 7 (5) (1996) 506–521.
[22] H. Lee, Scheduling and Processor Allocation of Matrix Computations on Parallel Systems, PhD thesis, Pohang University of Science and Technology, January 2000.
[23] H. Lee, J. Kim, S.J. Hong, S. Lee, Task scheduling using a block dependency dag for block-oriented sparse cholesky factorization, in: Proceedings of 14-th ACM Symposium on Applied Computing, March 2000, pp. 641–648.
[24] R.J. Lipton, D.J. Rose, R.E. Tarjan, Generalized nested dissection, SIAM J. Numer. Anal. 16 (2) (1979) 346–358.
[25] E.G. Ng, B.W. Peyton, Block sparse cholesky algorithms on advanced uniprocessor computers, SIAM J. Sci. Comput. 14 (5) (1993) 1034–1056.
[26] E.G. Ng, B.W. Peyton, A supernodal cholesky factorization algorithm for shared-memory multiprocessors, SIAM J. Sci. Comput. 14 (4) (1993) 761–769.
[27] J.M. Ortega, Introduction to Parallel and Vector Solution of Linear Systems, Plenum, New York, 1988.
[28] M.A. Palis, J.-C. Liou, D.S. Wei, Task clustering and scheduling for distributed memory parallel architectures, IEEE Trans. Parallel Distrib. Syst. 7 (1) (1996) 46–55.

[29] 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) (1996) 699–713.
[30] E. Rothberg, A. Gupta, The performance impact of data reuse in parallel dense cholesky factorization, Technical report, Stanford University, 1992.
[31] E. Rothberg, A. Gupta, An efficient block-oriented approach to parallel sparse cholesky factorization, SIAM J. Sci. Comput. 15 (6) (1994) 1413–1439.
[32] E. Rothberg, R. Schreiber, Improved load distribution in parallel sparse cholesky factorization, in: Proceedings of Supercomputing'94, 1994, pp. 783–792.
[33] V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors, The MIT Press, Cambridge, MA, 1989.
[34] R. Schreiber, Scalability of sparse direct solvers, in: The IMA Volumes in Mathematics and its Applications, vol. 56, Springer-Verlag, New York, 1993, pp. 191–209.
[35] K. Shen, X. Jiao, T. Yang, Elimination forest guided 2D sparse LU factorization, in: Proceedings of ACM Symposium on Parallel Algorithm and Architecture, 1998, pp. 5–15.
[36] B. Veltman, B. Lageweg, J. Lenstra, Multiprocessor scheduling with communication delays, Parallel Comput. 16 (1990) 173–182.
[37] T. Yang, C. Fu, Space/time-efficient scheduling and execution of parallel irregular computations, ACM Trans. Prog. Lang. Syst. 20 (6) (1998) 1195–1222.
[38] T. Yang, A. Gerasoulis, PYRROS: Static task scheduling and code generation for message passing multiprocessors, in: Proceedings of 6th ACM International Conference on Supercomputing, 1992, pp. 428–437.
[39] T. Yang, A. Gerasoulis, DSC: scheduling parallel tasks on an unbounded number of processors, IEEE Trans. Parallel Distrib. Syst. 5 (9) (1994) 951–967.