 CMES, vol.84, no.4, pp.297-331, 2012

Hybrid Parallelism of Multifrontal Linear Solution Algorithm with Out Of Core Capability for Finite Element Analysis

Min Ki Kim[1] **and Seung Jo Kim**[2]

Abstract: Hybrid parallelization of multifrontal solution method and its parallel performances in a multicore distributed parallel computing architecture are represented in this paper. To utilize a state-of-the-art multicore computing architecture, parallelization of the multifrontal method for a symmetric multiprocessor machine is required. Multifrontal method is easier to parallelize than other direct solution methods because the solution procedure implies that the elimination of unknowns can be executed simultaneously. This paper focuses on the multithreaded parallelism and mixing distributed algorithm and multithreaded algorithm together in a unified software. To implement the hybrid parallelized algorithm in a distributed shared memory environment, two innovative ideas are proposed to reduce the required physical memory. The first idea is pairing the factorization matrix. Pairing two factorization matrices in two threads with a square matrix for parallel factorizing reduces approximately half of the required physical memory. The second idea is splitting the factorization matrix, which enhances the solver performance by removing additional memory. The use of an out-of-core storage is necessary for the structural analysis with limited computing resources. To improve the computing efficiency with the out-of-core storage, it is essential to cache the factored data into free physical memory. To maximize the amount of cached data, selective data caching and recovery is used. Parallel performances of the proposed method are shown. The multithreaded parallel multifrontal algorithm is more computationally efficient than the serial algorithm.

Keywords: multithread, hybrid parallelism, structural analysis, multifrontal method, out of core, memory reduction

[1] Seoul National University, Seoul, Mechanical Aerospace Engineering, Korea
[2] Korea Aerospace Research Institute, Daejeon, President, Korea

1 Introduction

After the finite element method appeared, the necessity of structural analysis based on the finite element has grown out of various scientific and industrial fields. Rapid and optimal design of products requires a high-fidelity finite element structural analysis technique. Airplanes, automobiles, ships, buildings and electric machines should be designed using structural analysis. High performance, large scale structural analysis is essential to the fields of industrial business and academic research.

The required computational power of the structural analysis is increasing. Fortunately, the computational cost is decreasing according to Mohr's law. Currently, state-of-the-art computers allows a single machine to have computational capability in the hundreds giga-floating-point operations per a second (GFLOPS), with several tens of CPUs and more than 100 giga-byte (GB) physical memory. To exploit this computing power, finite element software should adopt new computing methodology. Software efficiency is no less important than hardware performance.

Finite element analysis, particularly in structural analysis, requires solving linear equations to acquire the stress or flux distributions, even in nonlinear analysis. Except for explicit time transient analysis, structural analyses depend on solving the solutions of linear equations. In addition to the static analysis of a structure, natural frequency analysis, nonlinear static analysis and implicit dynamic analysis also require the linear solver. These three analyses iterate the loop with the linear solver to converge to a solution; many eigensolvers find the intermediate eigenvectors with shifted eigenvalues by the linear solver, and nonlinear analysis requires computing incremental solution of the tangential stiffness matrix by a linear solver. Generally, a linear solver is one of the essential parts of structural analysis software.

The main computing time of all analyses described above is caused by the procedure of calculating the linear equation. Other parts of the software consume much smaller amounts of time than that of the linear solver. Hence, the optimized implementation of linear solver is the most important point for improving the performance of the software.

Sparse direct solver is most commonly used to analyze a structural problem. The most well-known structural analysis codes, such as NASTRAN [Duff and Reid (1983)], ABAQUS, ANSYS and our software IPSAP[1], use the direct solver to analyze the static, eigenvalue and implicit time integration problems. These employ multifrontal solution algorithm based on their own implementation or third party developer, BCSEXT-LIB[2] for ANSYS.

Several direct methods for sparse matrices have been proposed. A range of sparse

[1] `http://ipsap.snu.ac.kr`

[2] `http://www.aanalytics.com/`, initially developed by Boeing Company

direct solvers have been developed and distributed in public. Not only the symmetric positive definite (SPD) matrix solvers but also symmetric indefinite and unsymmetric solvers have been developed. SuperLU [Demmel, Gilbert and Li (1999); Li and Demmel (2003)], UMFPACK [Davis (2004); Davis (2004)] and MA41 .[ARIOLI, DEMMEL and DUFF (1989)], MA42 [Duff and Scott (1993)] in HSL[3] subroutines are well-known for solving unsymmetric matrices. For unsymmetric or symmetric indefinite cases, partial pivoting strategies are required to avoid the zero division and errors accumulation.

There are many solvers specialized for symmetric positive definite matrix such as CHOLMOD [Chen, Davis, Hager and Rajamanickam (2008); Davis and Hager (2009)], PaStiX [Henon, Ramet and Roman (2002)], PSPASES [Joshi, Karypis, Kumar, Gupta and Gustavson (1999)], TAUCS [Toledo, Chen and Rotkin (2003)] distributed parallel multifrotal algorithm implemented in IPSAP [Kim, Lee and Kim (2002); Kim, Lee and Kim (2005)] and the proposed method in this paper. In contrast to unsymmetric matrices, no pivoting is required for symmetric positive definite matrix. Some solvers have the capabilities to solve both symmetric and unsymmetric matrices, such as PARDISO [Schenk and Gartner (2004); Schenk and Gartner (2006)], MUMPS [Amestoy, Duff, L'Excellent and Koster (2002) ; Amestoy, Guermouche, L'Excellent and Pralet (2006)], WSMP [Gupta, Karypis and Kumar (1997); Gupta (2003)], BCSLIB-EXT and MA77 [Reid and Scott (2008); Reid and Scott (2009)] in HSL.

The sparse direct solvers mentioned above are classified into two categories. The first is based on left looking factoring algorithm. SuperLU, CHOLMOD, PaStiX and PARDISO follow the left/right looking elimination order to solve an equation. Supernodal Cholesky or LU factorization concept, which refers to a group of same sparse pattern of consecutive unknowns, is used to utilize the dense subroutines in BLAS and LAPACK for computing efficiency. The other group solvers exploit the multifrontal method; UMFPACK, HSL subroutines, MUMPS, PSPASES, WSMP, BCSLIB-EXT and IPSAP implement the multifrontal algorithm. Multifrontal method constructs an elimination tree and factors the unknowns via the elimination tree. Each frontal matrix in a same elimination level is independent from the other frontal matrices, so they are factored simultaneously in a multiprocessor machine. Moreover, factored matrices and their indices information are saved into the secondary storage, called out-of-core memory.

All direct solvers pass to analyze the problem prior to numerical operation in order to minimize the fill-in entries. All sparse solvers require a permutation matrix given by user or their own interface connected to 3^{rd} party reordering library. Var-

[3] Harwell Subroutine Library, `http://www.hsl.rl.ac.uk/`

ious kinds of reordering schemes are proposed. Multiple minimum degree (MMD) [Liu (1985)], approximated minimum degree (AMD) and its variant algorithms [Amestoy, Davis and Duff (1996)], SCOTCH [Pellegrini and Roman (1996)] and MeTiS are widely used to scientific computing. MeTiS nested dissection reordering routine (METIS_NodeND) [Karypis and Kumar (1998)] is mostly frequently implemented in this overview.

To exploit the high-performance computing technology, many parallel algorithms for distributed parallel or the shared memory multiprocessor environment are developed. Message Passing Interface (MPI) or Parallel Virtual Machine (PVM) is most frequently used to implement the software for the parallel computation of a distributed multiprocessor, and OpenMP or POSIX thread library is generally employed for use with the shared memory multiprocessor machine. WSMP proposed large scale numerical factorization of multifrontal algorithm with up to nearly one thousand processor for distributed 1, 024 processors [Gupta, Karypis and Kumar (1997)]. PSPASES also showed scalable multifrontal Cholesky factorization with 256 processors [Joshi, Karypis, Kumar, Gupta and Gustavson (1999)], and MUMPS introduced the dynamic [Amestoy, Duff, L'Excellent and Koster (2002)] or hybrid scheduling [Amestoy, Guermouche, L'Excellent and Pralet (2006)] of the distributed parallel multifrontal factorization for both the symmetric and unsymmetric matrices [Amestoy, Duff and L'Excellent (2000)]. SuperLU and PARDISO addressed the parallel implementation of the left/right looking unsymmetric LU factorization both on the symmetric multiprocessing [Demmel, Gilbert and Li (1999); Schenk and Gartner (2004)] and massively parallel processing [Li and Demmel (2003)] environment. The original algorithm in IPSAP [Kim, Lee and Kim (2005)] issued the parallel Cholesky factorization with flexible block sizes for the distributed parallel computing architecture.

Tab. 1 represents the brief information of above mentioned sparse direct solvers in terms of basic algorithm, matrix type, parallelized method and out-of-core capability. They have been applied successfully to a range of numerical analyses including the finite element problem, whereas, the proposed multifrontal algorithm is the most suitable for the structural analysis in practical cases among presented solvers in Tab. 1.

First of all, many unsymmetric solvers, such as SuperLU, cannot take advantages of the properties of the symmetric positive definiteness of the finite element problems. They are developed for the symmetric indefinite or unsymmetric matrices. This nature leads to a requirement for additional computation and storage to get rid of dividing by zero or finding square roots of negative diagonals. In addition, unsymmetric matrix factorization requires two times larger factor spaces for upper/lower part than symmetric cases. But most of the finite element applications

Table 1: Summary of direct solution methods for sparse matrix

Solver	Algorithm	Matrix Type[4]	Parallelizing	OOC	Version[5]
PARDISO	Left/Right Looking	SPD, Sym, Unsym	OpenMP & MPI	O	3.2[6]
CHOLMOD	Left Looking	SPD	-	-	1.7.3
MA77	Multifrontal	SPD, Sym	OpenMP	O	5.7.0
MUMPS	Multifrontal	SPD, Sym, Unsym	MPI	O	4.8.4
MFS[7]	Multifrontal	SPD	MPI	O	1.7
PMFS[8]	Multifrontal	SPD	POSIX & MPI	O	2.1
SuperLU	Left/Right Looking	Unsym	OpenMP & MPI	-	-
WSMP	Multifrontal	SPD, Unsym	POSIX & MPI	-	-
UMFPACK	Multifrontal	Unsym	-	-	-
MA41	Multifrontal	Unsym	-	-	-
TAUCS	Multifrontal	SPD, Unsym	-	O	-
PaStiX	Left/Right Looking	SPD	POSIX & MPI	O	-
PSPASES	Multifrontal	SPD	MPI	-	-

generate the SPD global stiffness matrix, so partial pivoting and saving both upper/lower factored matrices are not necessary.

Next, many solvers have been implemented only with physical memory or else their performances of the out-of-core storage are poor. Structural analysis software requires the out-of-core capability, such as a hard disk, to utilize limited physical memory. All commercial software presented above use the out of core capability. Impossibilities or performance deterioration of the out-of-core storage is a great obstacle to their use for structural analysis. Performance degeneration of the out-of-core storage will be proven in section 5.2.

Finally, the proposed multifrontal method is optimized to not only the distributed parallel machine but also the current multicore architecture used commonly. Several codes in Tab. 1 are tested with the proposed method for eight benchmark problems and their actual performances are worse than those of the proposed method in many cases shown in section 5.1.

The multifrontal solver embedded in our software was already parallelized for the distributed parallel computing environment. It showed better performances than other famous commercial software and it was successfully applied a range of structural analysis problems. However, it was seldom considered for use in the multicore CPU computer. Before this work, the multifrontal solver solved problems by their serial algorithm with parallelized numerical libraries. Embedded numerical

routines in BLAS and LAPACK was optimized for the CPU and was parallelized for the multicore CPUs, so serial multifrontal solver shows the "apparent" parallel performances, which actually came from the external routines.

This paper presents the hybrid parallelism of both shared memory multithreaded and distributed data parallelism of the multifrontal solution method. The distributed parallel multifrontal method was already developed and tested in a massively distributed parallel computer with up to 256 computing nodes as described in [Kim, Lee and Kim (2005)]. Therefore, this paper focuses on the multithreaeded parallelization and combining the multithreaded algorithm with the distributed algorithm to unify those two parallelisms, called hybrid parallelism. Novel handling and data structure of the factorization matrix are proposed for the efficient use of physical memory. Performance comparisons of a range of benchmark problems are carried out by parallel and serial multifrontal method. The proposed parallel algorithm shows the better performances in terms of the numerical factorization and the triangular solution of the equations. How to parallelize the multifrontal solution algorithm and why the method is better for the multicore shared memory architecture will be presented.

2 Overview of Multi-frontal Solution Methods

The basic concept of the multifrontal method was first introduced by Duff *et al*. [Duff and Reid (1983)] as a generalization of the frontal method of Iron [Irons (1970)]. The frontal method was proposed to solve the finite element method with a lesser core storage requirement. The key point of frontal method is to eliminate DOFs (degree of freedom) during the assembly of the stiffness matrices.

After some DOFs are assembled completely, they can be eliminated directly because the matrix coefficients and components of the load vector associated with those DOFs are not changed. The dense matrix assembled in the current step is called the "frontal matrix".

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} \tag{1}$$

Subscript 1 means fully assembled DOFs and subscript 2 means adjacent DOFs, which are not fully assembled. After $\boldsymbol{U}_1$ is assembled, the frontal matrix $\boldsymbol{K}_{11}$ can be factorized and substituted into in terms of $\boldsymbol{U}_2$such as below.

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} = \begin{bmatrix} L_1 & 0 \\ \bar{K}_{12}^T & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & \bar{K}_{22} \end{bmatrix} \begin{bmatrix} L_1^T & \bar{K}_{12} \\ 0 & I \end{bmatrix}$$

$$K_{11} = L_1 L_1^T$$

$$\bar{K}_{12} = L_1^{-1} K_{12} \tag{2}$$

$$\bar{K}_{22} = K_{22} - \bar{K}_{12}^T \bar{K}_{12}$$

The intermediate frontal matrix $\bar{K}_{22}$ in (2) should be saved in the temporary stack and the factored data L_1 and $\bar{K}_{12}$ are saved in the physical memory or the out-of-core storage. Most direct solvers implemented in structural analysis software used widely, including original implementation of the multifrontal method [Kim, Lee and Kim (2005)], save the factored data blocks in the out of core storage only, for a structural engineer to handle the relatively large size problem with limited computing resources. The enhanced algorithm stores the data in the physical memory rather than out-of-core storage. Novel data caching and memory management techniques are developed by optimizing the memory usage of the limited physical memory.

In the general finite element structural analysis problem, the global and the element stiffness matrices are symmetric and positive definite (SPD). Therefore, equation (2) implies Cholesky factorization and substitution. This process employ BLAS and LAPACK library routines optimized to a range of computer architectures. The maximized performance of the CPU is achieved using those libraries embedded in the solver.

All sparse direct solvers pass three phases: analysis of the problem, numerical factorization, and substitution of the solution vector. The analysis of the problem should be conducted before the numerical procedure to reduce the floating point operations and memory requirement. In the multifrontal solver, the analysis phase consists of three sub-phases: building a finite element graph, dividing the graph into many pieces of small subdomains, and symbolic analysis based on the divided mesh to obtain the size of the required memory. Other sparse direct solvers reorder the global stiffness matrix with a range of reordering algorithm to minimize the fill-in entries of the global stiffness matrix. Numerical factorization and substitution is calculated based on the analysis phase, so the analysis of the problem is important. To employ a sparse direct solver for finite element problems, another phase should be added before the beginning of the solution procedure; it is to assemble the global stiffness matrix.

The multifrontal solver is implemented to characterize finite element problems, so that the explicit assembling global stiffness matrix is not required. In contrast, other sparse direct solvers need global stiffness matrix assembled in the sparse matrix structure, such as compressed sparse row (CSR) or other similar format. The algorithm exploits subdomain-base approaches rather than constructing the global

sparse matrix. These subdomains are in bottom of the elimination tree, and the numerical factorization starts from those subdomains. All operations, including merge and extend-add operation, are performed with the many pieces of the element subdomains. Therefore, extra computations and memory space for the global stiffness matrix are not required in the multifrontal solver. The elapsed time for assembling the global stiffness matrix is proportional to the problem size, and the ratio of total elapsed time is approximately 5-10%. Another advantage of the algorithm is that the analysis time of the problems, including building the finite element graph, partitioning the domain and symbolic factorization, is reduced significantly. The finite element graph is smaller in general cases than the sparse structure of the global stiffness matrix, so its analysis is simpler than other direct solvers.

The multifrontal solver factors multiple frontal matrices showing in Fig. 3 for the 4 regularly partitioned subdomain case. Actually, the entire domain of a problem is divided into several subdomains by METIS (METID_NodeND) in the proposed algorithm.

The distributed data parallelism of the multifrontal method is described precisely in [Kim, Lee and Kim (2002); Kim, Lee and Kim (2005)]. Parallel extend-add distributed dense matrix operations are performed by way of [Gupta, Karypis and Kumar (1997)] which shows good parallel efficiency for a large scale problem. In order for efficient implementation and minimization of global data communication to be achieved, flexible matrix block size is adopted in our work. Fig. 1 represents extend-add operation of distributed matrix for a case of four processors with MPI data communication.

Fig. 2 represent the hybrid parallelized numerical factorization and substitution of the 4x 8 square problem regularly partitioned as 2x4 in two computing nodes with four threads each. Computing nodes and their threads are depicted by blue boxes and yellowish brown boxes, respectively. The light black lines in every thread denote internal unknowns of each subdomain eliminated in first stage, and the bold blue and bold yellow lines represent the second and third stage of the numerical operation executed independently in all computing nodes. The red lines are the group of DOFs in final front eliminated by two computing nodes. The numerical factorization associated with internal DOFs of a subdomain depicted by black lines is called the divided working layer, the smallest independent job unit of hybrid parallelized algorithm. The operation of DOFs in bold blue lines and bold yellow lines is a co-working layer, which means elimination levels with co-processing tasks of multiple threads, such as the second and third level in Fig. 4. Divided working layer and co-working layers do not depend on other processes. Finally, the operation of unknowns in red lines is called the distributed computing layer. This is because all data including matrices and indices arrays are distributed in

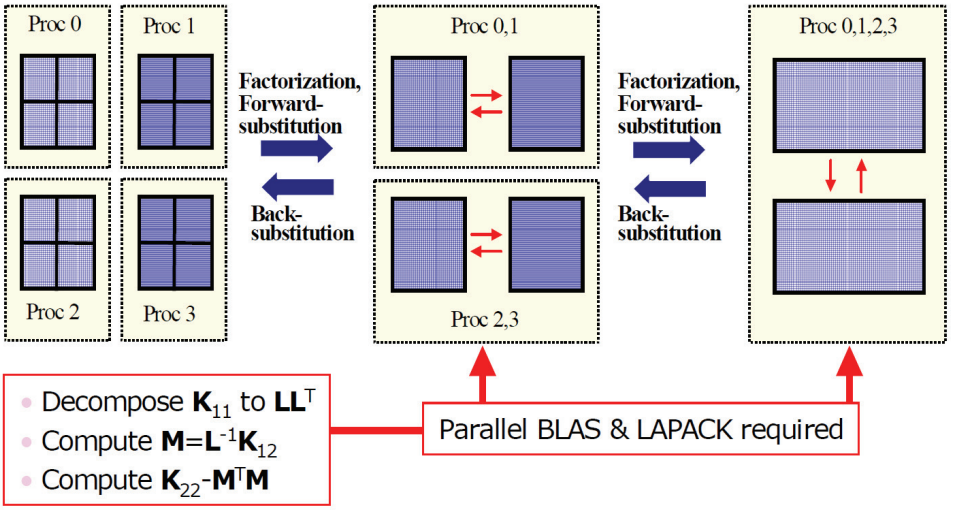

Figure 1: Parallel implementation of the distributed matrix factorization

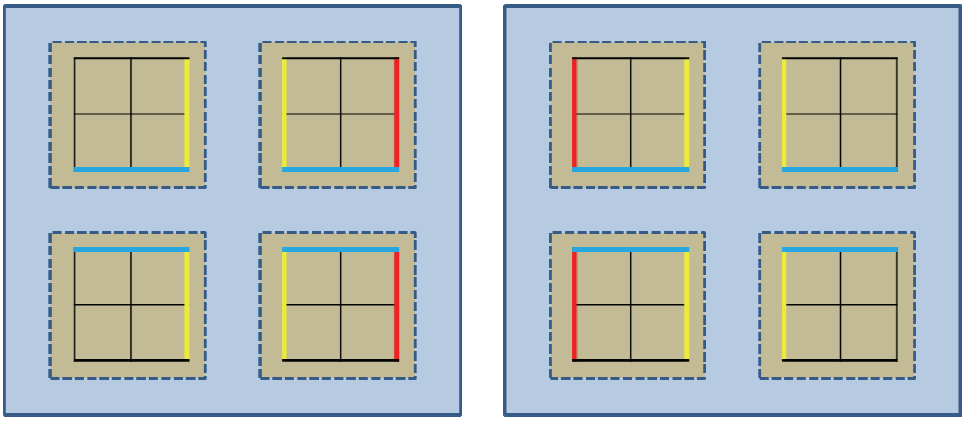

Figure 2: Hybrid parallel numerical factorization in two computing nodes with four threads each

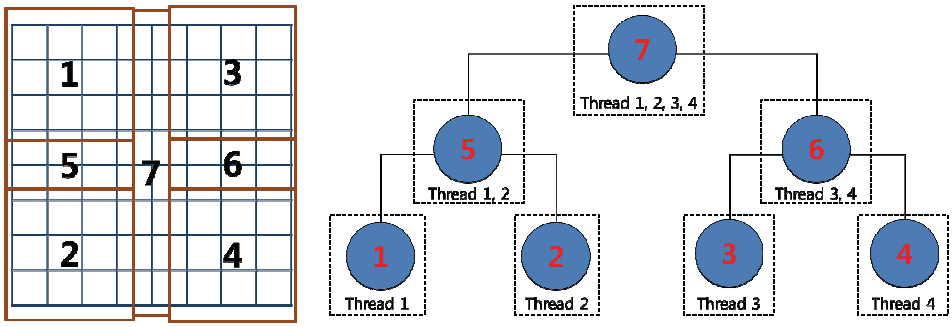

Figure 3: Finite element domain and its elimination tree of the four threads problem

multiple computing nodes.

3 Multithreaded Parallelization of Multifrontal Solution Method

3.1 Multithreaded Parallel Algorithm of MFS

Multifrontal solution method eliminates the unknowns in a subdomain or on an interface independently because each group of unknowns does not have any connections with other groups. Thus, degrees of freedom (DOF) in all subsets of an elimination level can be factored simultaneously without any perturbing the solution procedure.

Fig. 3 shows the elimination tree of the four subdomain problem regularly partitioned into 2x2. All internal DOFs in all subdomains, grouped in 1, 2, 3 and 4 in Fig. 3, are independent from each other's internal subdomain. Unknowns on the subdomain interface, grouped in 5 and 6, also have no adjacency to each other. Each group of DOF in a lower level of the elimination tree is factored in terms of the interfacial DOF in the higher level.

This nature of a multifrontal algorithm of the finite element problem gives some hints for parallelization. The serial algorithm uses the elimination order which follows the post-order of the elimination tree, whereas the parallel algorithm uses a level-based, bottom-up heap order of the elimination tree. For the case of Fig. 3, the serial algorithm should pass seven stages to complete the factorization while the multithreaded parallel algorithm should go through just three stages. DOF groups 1 to 4 at the bottom level in Fig. 3 are factored at the same time by four tasks of the parallel algorithm. After the first step, groups 5 and 6 are factored in a similar way by two tasks consisting of pairs of threads, respectively. The final stage of the

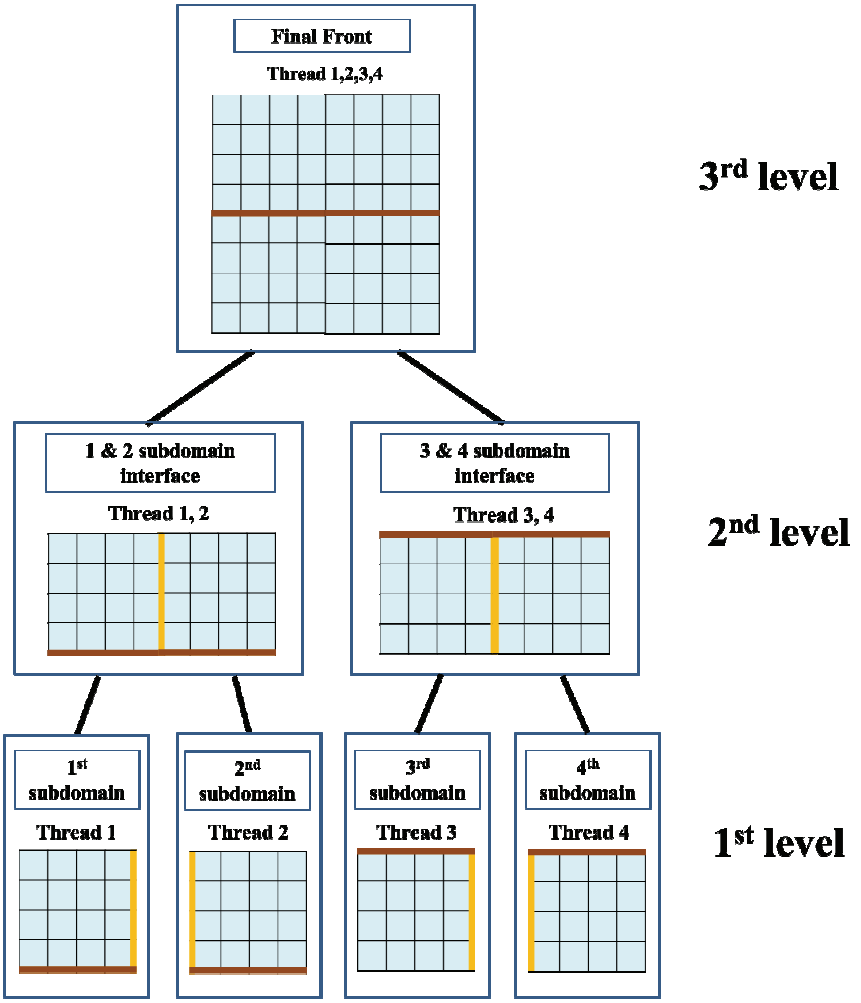

Figure 4: Elimination tree of multithreaded parallel factorization

factorization involves all threads.

The elimination tree with parallel computation of Fig. 3 is represented in Fig. 4. Parallel procedure in a shared memory multiprocessor is carried out by several threads with Win32/POSIX thread application program interface (API). Unlike distributed memory case, parallelizing MFS for a shared memory machine need not communicate the data stream from a thread to adjacent thread. Multithreaded parallelism of MFS accelerates the solver performance higher up to 10-80% than that of original algorithm in a symmetric multiprocessor machine (SMP). Performance

enhancement will be presented a later section.

The main drawback of the parallelism is that it requires a larger amount of physical memory than does the sequential algorithm. All threads invoked by the main procedure need their own local stack and working space. To remedy this large memory consumption, minimization of temporal space is necessary to the solver. Two novel concepts for multithreaded MFS are proposed. One is pairing the factorization matrix, and another is splitting the factorization matrix.

3.2 Numerical Performance Comparison of Factorization Matrix Form

Before introducing the next two concepts, the numerical performance according to the form of factorization matrix should be considered. Many finite element problems, including structural analysis problems, generate a symmetrical positive definite matrix. Therefore, not only the square form of the factorization matrix but also the triangular packed form can be considered.

Obviously, the triangular form of the factorization matrix is more efficient than the square form in terms of the usage of the physical memory. However, the computing performance of the square matrix form of the frontal matrix is faster than that of triangular matrix. Therefore, the square matrix structure is more desirable than the triangular form for computational efficiency. The tested computer has 32GB of physical memory and Intel Xeon X5482 3.20 GHz CPU which combines 8 cores. Tab. 2 lists the specifications of the tested machine and the program. DPOTRF routine is used for a square matrix, and DPPTRF is used for a triangular matrix. Tab. 3 lists the elapsed time comparison of the factorization routines of LAPACK for a various sizes of symmetric positive definite matrices. Tab. 3 shows that the numerical factorization of a square matrix is more efficient than the triangular form in the most common cases arisen from the multifrontal method.

Table 2: Tested machine and program specifications

CPU	Intel Xeon X5482 3.2GHz, 8 cores
Memory	32GB DDR2 399MHz
Library	Intel Math Kernel Library 10.3.2.
Compiler	Intel C/C++ Compiler with highest optimization option
OS	Microsoft Windows Server 2008 HPC Edition, x64 version

From the tests shown in Tab. 3, numerical factorization of the square form of the frontal matrix is more suitable than the triangular form. Consequently, square form should be adapted in the solver to utilize the computing performance. The remaining issue is how to optimize the physical memory with the square factorization

Table 3: Elapsed time comparison of factorization routines in LAPACK for a SPD matrix

Routine \ Size	**100**	**500**	**1000**	**5000**	**10000**
DPPTRF [1]	1.25E-4	3.79E-3	1.83E-2	1.37	8.88
DPOTRF [2]	1.09E-4	2.43E-3	9.36E-3	0.55	3.90
Ratio [1]/[2]	**1.15**	**1.56**	**1.95**	**2.51**	**2.28**

matrices.

3.3 Pairing the Factorization Matrix into a Square Matrix

Because finite element stiffness matrix is symmetric, only half of the square factorization matrix is valid. The other half of the matrix is not accessed by the threads. Not using these halves of the matrices is wasting a huge size of physical memory. To reduce the size of wasted space, it is necessary to merge two factorization matrices used by two threads into a single square matrix. One matrix occupies the upper part of the square matrix, and the other matrix is in lower part of the square space. A square matrix is partitioned into two factor spaces and used by per every two threads.

In other words, two factorization matrices are paired together into the single square matrix to remove unused space of the square form. In many cases, the maximum required memory size of pairing two factorization matrices is approximately half of the separated square factorization matrices. The triangular form uses less memory, but it is not recommended in terms of computational efficiency.

Fig. 5 shows the basic concept of pairing the matrices with four threads case. In this figure, two tasks share a square matrix to factor their assigned job in the bottom and second elimination level. Two square matrices are used at the lowest level, and one square matrix is required at the second and final level. Generally, the size of the matrix is similar to those of other matrices in the same elimination level. Therefore, the total required memory for pairing two matrices is comparable for the case of triangular form, and the numerical performance of pairing is absolutely better than that of the triangular matrix. Pairing two matrices is the most competitive case both in terms of computational speed and memory requirement.

3.4 Splitting the Factorization Matrix

Another idea both to reduce the memory and to improve computing performance is splitting the factorization matrix. The original algorithm [Kim, Lee and Kim (2005)] uses one dense square factorization matrix per a process. After being fac-

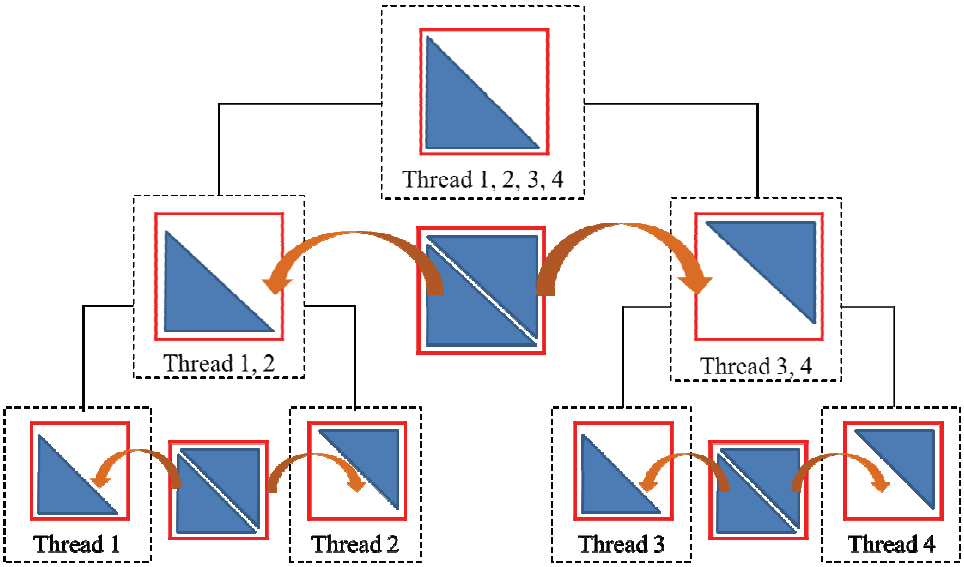

Figure 5: Basic concept of pairing two matrices for four threads case

tored in current step, sub-matrices $\boldsymbol{K}_{11}$, $\boldsymbol{K}_{12}$ should be cached in other place of physical memory or saved in out-of-core storage by the original MFS. The half part of the factorization matrix of the previous MFS is utilized as a frontal stack to save intermediate data including associated DOF indices arrays and matrices $\boldsymbol{K}_{22}$.

The way of the factorization matrix of the original MFS is simple and easy to implement. Data structure of the square matrix is a straightforward extension of the 2-dimensional arrays. Assembling and merging the frontal matrices is easier, and has less computing overheads in the contiguous space than in the non-contiguous space. However, additional memory is required to cache the factored data $\boldsymbol{K}_{11}$ and $\boldsymbol{K}_{12}$ and to save intermediate frontal stack data including $\boldsymbol{K}_{22}$. Moreover, the copy operations, memory to memory or memory to secondary storage of $\boldsymbol{K}_{11}$, $\boldsymbol{K}_{12}$ and $\boldsymbol{K}_{22}$, are also required because those sub-matrices in the square factorization matrix are on different position from the factored storage and the stack space.

Splitting the factorization matrix can remedy these demerits. Its basic idea is that a factorization matrix is torn into three sub-matrices. Sub-matrices $\boldsymbol{K}_{11}$, $\boldsymbol{K}_{12}$ and $\boldsymbol{K}_{22}$ are allocated to their own discontinuous position in the available physical memory. The sub-matrices $\boldsymbol{K}_{12}$ and $\boldsymbol{K}_{11}$ can be allocated in the cache space for factored (CSF) and $\boldsymbol{K}_{22}$ can be allocated in the frontal stack to avoid the copy operation of memory to memory. Further additional work space for factorization matrix associated with those three sub-matrices is not needed. The sub-matrices $\boldsymbol{K}_{12}$ and $\boldsymbol{K}_{11}$ are

allocated dynamically in CSF and $\boldsymbol{K}_{22}$ is allocated in frontal stack space during the numerical factorization. The memory space occupied by sub-matrices $\boldsymbol{K}_{11}$ and $\boldsymbol{K}_{12}$ can be not only in the factorization work space but also in cache space for them. The sub-matrix $\boldsymbol{K}_{22}$, which is intermediate frontal matrix, can be reused in the next step of the factorization procedure. In the original MFS, the work space for the numerical factorization differs from both the cache space and the stack space. So the copy operation and additional memory cannot be avoidable in this case.

Splitting the factorization matrix utilizes the separated memory associated with the $\boldsymbol{K}_{11}$, $\boldsymbol{K}_{12}$ and $\boldsymbol{K}_{22}$ matrices. Because of the non-contiguity of the sub-matrices, the implementation of this concept is more complex than the square factorization matrix. However, the advantages of this idea make the solution performance better than the square factorization matrix form. The complexity and the overheads of splitting the factorization matrix can be ignorable. A range of numerical experiments, which will be shown in the next section, prove this statement.

A novel data structure called "triplet factorization matrix" has been invented to apply "logically continuous" on actually discontinuous memory space. $\boldsymbol{K}_{11}$, $\boldsymbol{K}_{12}$, and $\boldsymbol{K}_{22}$ are on discontinuous location in the physical memory, but those sub-matrices should constitute a "logical" factorization matrix from the caller application. These three elements are logically composed of a data structure, and it has been named "triplet matrix" to represent three components bound together.

Fig. 6 represents the splitting of the factorization matrix with the pairing two factor

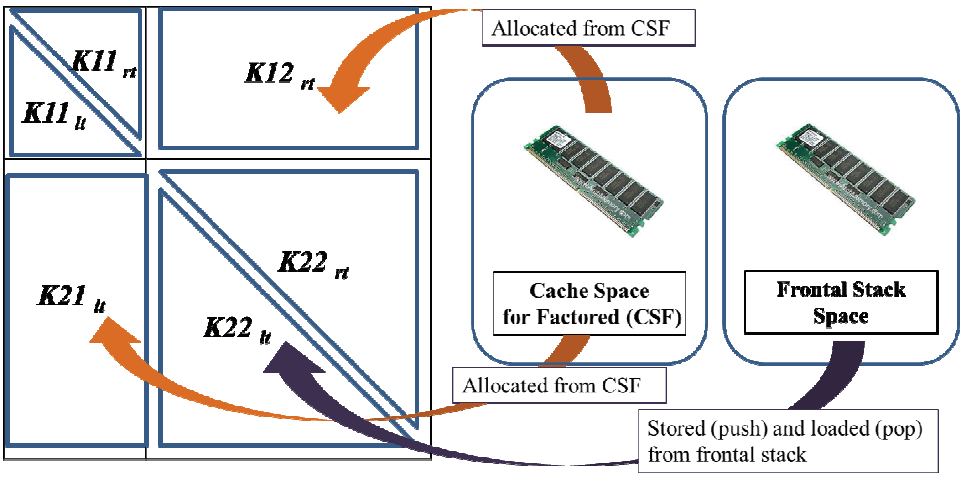

Figure 6: Splitting the Factorization Matrix - Triplet Factorization Matrix with pairing the sub-matrices

spaces. In this case, a total of six sub-matrices are combined into a logical data structure. Two tasks employ the paired $\boldsymbol{K}_{11}$ and $\boldsymbol{K}_{22}$matrices, and use their own $\boldsymbol{K}_{12}$ spaces in the split paired factorization matrix. But the concept of splitting the factorization matrix is independent from pairing the square matrix. So these two ideas of treating the factorization matrix are applied individually in the application. The numerical performance and required memory of the composition of two ideas will be presented in the next section.

3.5 Memory Requirements of the Ways of the Factorization Matrix

The amount of memory requirements of what kind of the factorization matrix is employed is shown in this section. Two structural analysis problems, "Cubic" and "Wing" described in Tab. 4, are tested to check the required memory size and total factored space size. First one of the tested problems is a 3-dimensional cubic figure modeled in 50x50x50 8-node hexahedron elements; another is 2-dimensional airplane wing model made up of 4-node quadratic shell. These numerical experiments are conducted by eight processors.

Four types of tests of the factorization matrix are described below:

- A general separated square matrix per a task.
- A pair of square matrices per two tasks.
- A split square matrix with separated sub-matrices per a task.
- Split square matrices with paired sub-matrices per two tasks.

With these options, all tests measure three sections of used memory; required memory size for the solver, the size of cache space for factored (CSF) and total peak memory size. Tab. 4 and Fig. 7 show the results in megabyte (MB) of the benchmark problems and scaled sizes in terms of paired split case, respectively. The column at the right end in Tab. 4 represents the ratio of required solver memory compared to 4^{th} option. Pairing two matrices drops the required solver memory approximately in half of the unpaired matrix. Splitting the matrix also reduces the solver memory about 30-50% because the extra working spaces for the submatrices $\boldsymbol{K}_{11}$, $\boldsymbol{K}_{12}$ and $\boldsymbol{K}_{22}$ are not required. The split square matrix with paired sub-matrices option requires the least physical memory of both two test problems.

4 Factored Data Caching and Recovery

The factored front data should be saved in either physical or out-of-core storage. To overcome the limitation of the physical memory, utilizing out-of-core storage

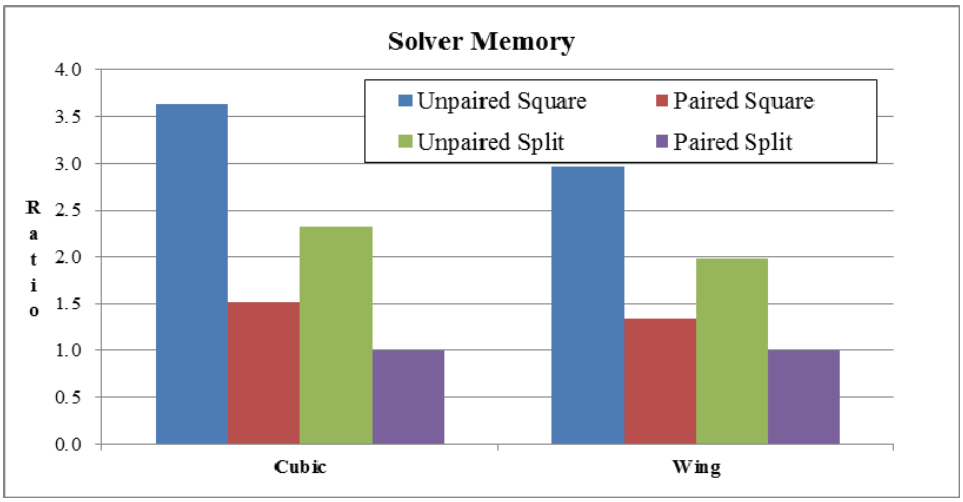

Figure 7: The graph of the required solver memory scaled in terms of the size of paired split factorization matrices

Table 4: Memory size of multifrontal solver with a range of the factorization matrix options

Problem	Case	Required	Size of CSF	Peak	Ratio
Cubic	1	5266.36	4606.87	9873.23	3.63
	2	2214.88		6821.75	1.53
	3	3372.92		7979.79	2.32
	4	1451.91		6058.78	1.00
Wing	1	1355.5	2452.52	3808.02	2.97
	2	608.42		3060.94	1.33
	3	906.80		3359.32	1.98
	4	456.98		2909.50	1.00

should be considered. Many algorithms evaluating the size of out-of-core data [Liu (1986)] and efficient usage of the secondary storage for serial/shared memory platform [Duff and Scott (1993); Dobrian and Pothen (2000); Rotkin and Toledo (2004); Reid and Scott (2008); Reid and Scott (2009)] and distributed parallel [Agullo, Guermouche and L'Excellent (2008)] have been designed. The original multifrontal algorithm [Kim, Lee and Kim (2005)] was implemented to utilize only the out-of-core storage for factored data so that the solver could solve relatively large-scale problems within the limited memory space, even in a poor machine. This is similar to other commercial finite element analysis software.

To accelerate the overall performance of the solution algorithm with the out-of-core storage, smart strategies of the data storing and fetching are required. Data caching, which means to cache the factored data into the physical memory, is a powerful to utilize available physical memory. Cacheable data should be chosen carefully to maximize the space used by data recovery. Data recovery is an algorithm to fetch and to allocate the data from the out-of-core storage into the physical memory.

Before parallelizing multifrontal solution algorithm for a symmetric multiprocessor machine, the original solver was improved to employ the physical memory to store the factored data into CSF. Data caching in a sufficient physical memory is not difficult to implement. The solver calculates the required memory size in a symbolic analysis phase and allocates the physical memory space prior to numerical factorization. The remaining issue of data caching is due to the lack of physical memory.

Storing the factored data blocks in out-of-core storage deteriorates the computing performance of both the numerical factorization phase and substitution phase. Performance degeneration is clear, particularly in the substitution phase, so eigenvalue analysis or implicit time transient analysis with the multifrontal algorithm will be extremely slow. Therefore, the solution method requires optimization of the data caching technique to remedy this drawback.

Data recovery is desired to enhance the substitution performance of the solver. After the numerical factorization, the square matrix is released freely. The memory block occupied by the factorization matrix can be used to recover the data in out-of-core storage. The goal is to increase the size of data allowed to recover within a free memory block. Selective data caching is proposed to maximize the amount of recovered data from the out-of-core storage to the physical memory during the substitution phase.

In the proposed algorithm, selective data caching and selective data recovery are adapted to enhance the input and output (IO) performance. Selective data caching aims to save the data in the physical memory according to the top-down or bottom-up heap order of the elimination tree. In contrast, sequential data caching accumulates the factored data by following the post-order of the elimination tree. The cacheable data block is chosen carefully to avoid the size of CSF exceeding the allowable physical memory size.

Selective data caching makes the solver IO performance better during the substitution phase by increasing recovered data space. Selective data caching provides a larger amount of recovered data space than sequential caching. The large sized data block has higher cache priority than the small sized block by selective caching. Therefore, the temporary space required by the substitution phase of se-

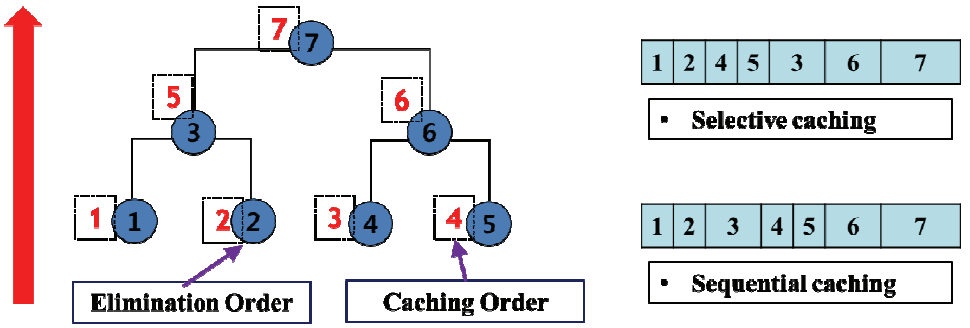

Figure 8: Data caching diagrams of the selective caching and sequential caching for a serial MFS

lective caching is smaller than that of sequential caching. Generally, the size of the data block follows the level from the elimination tree: the size of the data blocks increases with increasing level of the DOF groups on the elimination tree. Hence, a larger amount of cache space subtracted temporary space from the free memory is available by selective caching. A similar process can be applied to data recovery conducted in the substitution phase. Fig. 8 represents the conceptual diagrams and the memory structures of CSF with the selective caching and sequential caching for four subdomain case.

The selective data caching is applied both the serial algorithm and the parallel one. The only different thing between serial data caching and parallel data caching is that cache priorities are determined according to the top-down order of the elimination tree for the parallel algorithm, while those of the serial algorithm are determined according to bottom-up order.

Factored data blocks must be allocated in the CSF, which is the remaining part that solver subtracts the factorization matrix and the frontal stack space from available physical memory. For the serial algorithm case, top-down caching order of the elimination level is more desirable to reduce the reserved space for the frontal stack. For the parallel algorithm, in contrast, reserved space including both the factorization matrix and the frontal stack at a lower elimination level is larger than those of the higher levels. Therefore, unlike the serial algorithm, the bottom-up order, indicated by the red arrow shown in Fig. 8, should be preferable.

The data recovery phase is conducted during the substitution phase. The principle of data recovery is identical to the selective caching. If a data block in the higher level is recovered in advance, the smaller temporal substitution matrix should be allocated for loading the out-of-core data blocks.

5 Performance Measurement of Multithreaded Parallel MFS

5.1 Performances of the serial, parallel MFS and other sparse direct solvers with in-core memory

To compare the numerical performance of the multithreaded parallel MFS with the serial MFS, PARDISO, MUMPS, CHOLMOD and MA77 presented in Tab. 1, eight benchmark problems are tested in a multicore shared memory computer. The brief information of the tested problems is shown in Tab. 5. All tests are performed by eight processors with their own parallel algorithms and parallelized BLAS and LAPACK. Serial MFS and parallel MFS is named "MFS" "PMFS, " respectively, for all tests presented in this paper.

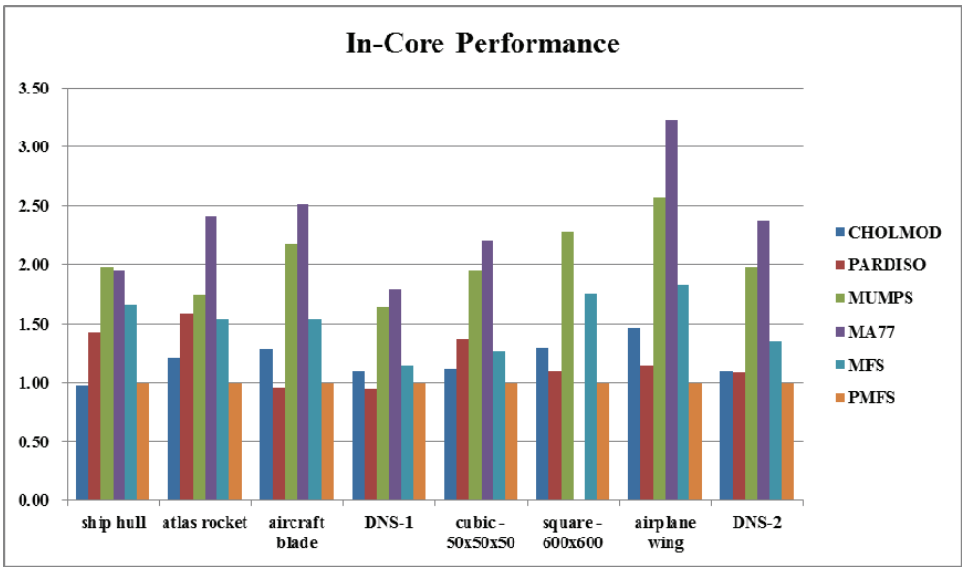

Figure 9: Elapsed time of the direct solvers of tested problems scaled in terms of that of PMFS

In this section, all tests are conducted only with the in-core memory. The performance with the out-of-core storage in the limited physical memory will be shown in the next section. Fig. 9 is the elapsed time comparisons of a range of direct solution methods scaled in terms of that of PMFS. For all tests in this paper, elapsed times are measured by summing up three phase of the solution of the direct solvers; analysis steps including reordering the unknowns (element graph partitioning for MFS and PMFS) and symbolic factorization, numerical factorization and solution of the equations. Assembling global stiffness matrix as the sparse matrix form is

not added to this result. The "Square" problem in Tab. 6 cannot be solved by MA77 only with physical memory.

Table 5: Benchmark problems of numerical performances of the solver

Problem	No. of DOFs	No. of Elements	No. of nonzeros	Description
Cubic	390, 150	125, 000	3.037E+7	Hexa8 50x50x50
Square	2, 163, 600	360, 000	5.937E+7	Quad4 600x600
Ship Hull	272, 897	98, 828	7.910E+6	Bending of ship hull
Wing	1, 022, 142	170, 356	2.808E+7	Airplane wing model
ATLAS	1, 201, 551	255, 550	3.755E+7	Eigenvalue analysis of ATLAS rocket
Blade	461, 739	121, 536	1.629E+7	A blade of a VTOL
DNS-1	199, 107	94, 916	7.491E+6	Direct numerical simulation of composite materials
DNS-2	448, 243	138, 240	1.737E+7	

Table 6: Elapsed time of the direct solution algorithms scaled in terms of that of PMFS

	Ship Hull	ATLAS	Blade	DNS-1	Cubic	Square	Wing	DNS-2
CHOLMOD	0.98	1.21	1.29	1.10	1.12	1.29	1.46	1.10
PARDISO	1.43	1.58	0.96	0.96	1.38	1.10	1.15	1.09
MUMPS	1.98	1.74	2.18	1.64	1.95	2.28	2.56	1.98
MA77	1.96	2.41	2.52	1.80	2.21	N/A	3.22	2.37
MFS	1.66	1.54	1.54	1.14	1.27	1.75	1.83	1.34
PMFS	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

By this test, PMFS shows the best performances for five benchmark problems, and shows second best performance for the other three problems. Because assemblage time of the global stiffness matrix is not shown in this result, actual performances of PMFS and MFS are more efficient than the other solvers for those finite element problems. The performance of the proposed PMFS is 30∼100% higher than serial algorithm. PARDISO or CHOLMOD sometimes shows the best performance of the numerical factorizations for some cases, but other solution phases of those two solvers generally take more time than MFS and PMFS. As shown in Fig. 9, the proposed PMFS is one of the best solvers for the case of in-core memory performance.

5.2 Performance of the PMFS, MFS and other solvers with out-of-core storage

In this section, multithreaded parallel performances are measured in a limited allowable physical memory. In this case, solvers including PMFS should utilize the out-of-core storage to store the factored data into a secondary storage to handle a large-size problem with a relatively small size of available physical memory. To evaluate the precise effects of the performance degradation by the out-of-core storage, all tests were conducted without IO buffering provided by the operating systems (OS). The modern OS, including Microsoft Windows, provide IO data buffering capability to enhance the IO performance of various applications. IO buffering can cache the data that should be saved in the out-of-core storage into the free space of physical memory not used by any software. This functionality generally shows higher IO performance than of the un-buffered (direct) action, but it is also an obstacle to measuring the accurate influence of IO. Therefore, all tests in this section are conducted in the virtual machines with small sizes of allowed physical memory on Hyper-V to remove the IO caching operations by OS. Tab. 7 shows the tested problems, their core memory options and total size of the memory of their virtual machines. The memory sizes of the virtual machines in Tab. 7 are set to be sufficient to solve the problems with allocated core memory but not to exceed total required memory of the problems.

Tab. 7 and Fig. 10 represent the performances of solvers with the out-of-core storage for five large scale problems among eight problems in Tab. 5. The out-of-core performance of PMFS and MFS are absolutely higher than any other solvers for all tests problems. Different from in-core performances, PARDISO shows poor out-of-core performance. MA77 also has poor out-of-core performance, as well as a poor in-core benchmark. MUMPS is middle ranking for all cases of out-of-core options. As shown in Fig. 10, performance gap between PMFS or MFS and other direct solvers of out-of-core cases are more significant than the differences of in-core cases.

Table 7: Elapsed time of out-of-core solver performances scaled in terms of that of PMFS

	Blade	**Cubic**	**Square**	**Wing**	**DNS-2**
Core Memory	1500MB	3000MB	4000MB	1500MB	3500MB
VM memory	2.5GB	5.0GB	7.0GB	3.5GB	5.0GB
PARDISO	5.64	2.90	5.30	4.23	6.36
MUMPS	2.65	1.84	2.50	2.02	2.95
MA77	5.28	3.49	5.98	2.98	4.58
MFS	1.17	1.11	1.13	1.06	1.13
PMFS	1.00	1.00	1.00	1.00	1.00

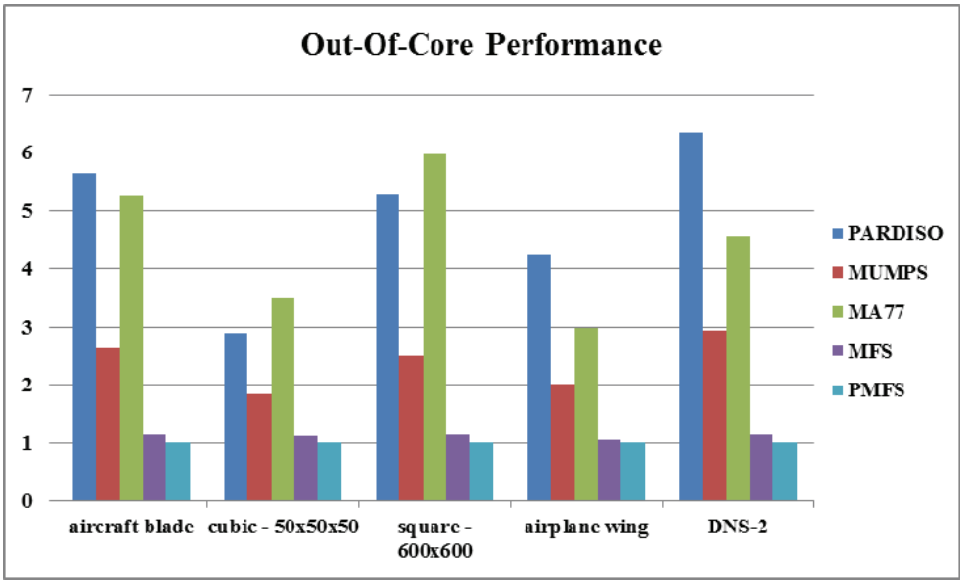

Figure 10: Elapsed time of the out-of-core solver performances scaled in terms of that of PMFS

PMFS shows slightly better out-of-core performances than MFS, while the size of out-of-core data blocks of PMFS is larger than that of MFS. To consider the precise evaluation of PMFS and MFS with out-of-core storage, two problems, "Cubic" and "Wing" in Tab. 5, are analyzed by PMFS and MFS with two kinds of allocated solver memory shown in Tab. 8. The sum of the cached data and recovered data is approximately the same as the total available physical memory size. Selective data caching and data recovery methodology maximize recoverable data space by minimizing the size of substitution matrix, which is negligible compared to allow-

able memory. The sizes of recovered data blocks of all available memory cases are nearly equal to each other for all tested problems.

The recovered data space is recycled from the residual space, which is subtracted the CSF from the available physical memory. For the serial algorithm, the size of recovered data approximately equals the size of memory for the factorization matrix. For the parallel algorithm, it is more complicated to represent the recovered data size. The reserved space of the lower layer is bigger than the higher layers, so it is reused for the CSF and reserved space of the higher layers. The accumulated residue, subtracted reused space of the higher layers from reserved space of the lower layer, cannot be usable for the numerical factorization. The sum of residue at each layer is eventually for the recovered space in the case of the parallel algorithm.

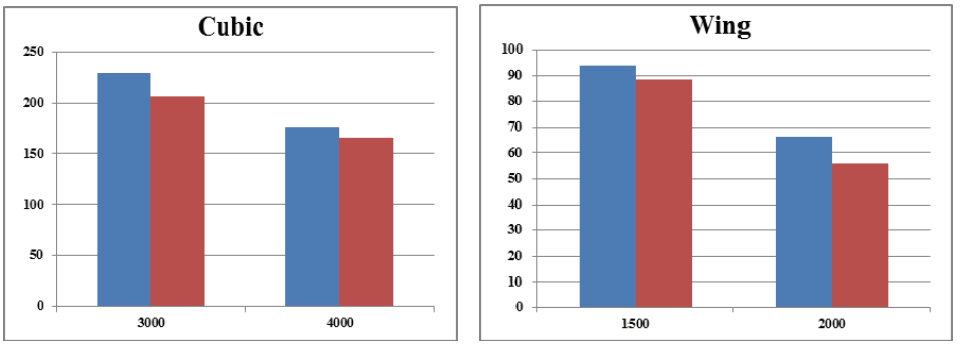

Figure 11: Charts of the total elapsed time of the benchmark problems with a range of allowable physical memory by PMFS and MFS

Tab. 8 and Fig. 11 represent the elapsed time of both the numerical factorization and the substitution of the PMFS and MFS. The blue bar and red bar in Fig. 11 represents the total elapsed time of MFS and PMFS, respectively. In this result, the parallel algorithm shows the better performance with the out-of-core storage as well as it is only with the physical memory. The parallel numerical factorization is much faster than the serial one, but the substitution of the parallel solver is worse than serial solver. Total elapsed time of PMFS is smaller than MFS.

The reason why the substitution of PMFS is slower than MFS is that the amount of recovered data of PMFS is larger than MFS. In most cases, the required memory of the parallel algorithm is greater than the serial algorithm. Therefore, the size of cached data of the parallel solver is smaller than the serial one, while the size of the recovered data of the parallel one should be larger than the serial one.

Table 8: The elapsed time of the benchmark problems by PMFS and MFS with the out-of-core storage

Problem	Allowable Memory	Factorization		Substitution		Total	
		PMFS	MFS	PMFS	MFS	PMFS	MFS
Cubic	3000	147.33	167.22	54.02	54.64	206.78	229.51
	4000	126.10	136.71	36.92	31.44	168.52	176.74
Wing	1500	53.56	58.97	28.98	27.94	88.18	93.84
	2000	34.55	42.15	15.94	16.53	56.13	66.24

In our work, all IO procedures were conducted sequentially, even in parallel solvers. Storing the data in the out-of-core storage and also loading the data from the out-of-core storage were executed with the concept of mutual exclusion to remove the bottleneck caused by multiple IO accesses. Further, loading or recovering multiple data blocks requires the same number of the substitution matrices. This reduces the recovered memory. By these reasons, locking IO operations against the multiple requests is applied in our algorithm. IO performance of PMFS is same as the MFS. IO waiting time is only proportional to the amount of IO in both parallel and serial algorithm; there is no relationship with the number of invoked threads.

The data assigned to be recovered is loaded from the out-of-core storage to the physical memory when the forward elimination phase is carried out for the first time. The elapsed time of the substitution in Tab. 8 includes the IO time of the out-of-core data blocks. After the data is recovered, the total amounts of cached data blocks are similar to each other. Thus, numerical performance of the parallel substitution at the next iteration, such as eigenvalue and linear time transient analysis, is the same or better than the serial version.

Consequently, the numerical performance of the parallel algorithm is better than the serial MFS and other sparse direct solvers even with the same limited physical memory.

5.3 Parallel Performance of MFS in Single SMP and a SMP cluster machine

The parallel performance of PMFS and MFS was measured both in a symmetric multiprocessing machine (SMP) and in an SMP cluster to comprehend the computational influence of the multithreaded parallel algorithm and the serial one. PMFS is parallelized for both the SMP and massively parallel processing (MPP) environment via Win32/POSIX and MPI, while MFS is specialized for only distributed multiprocessor cluster architecture. In this paper, hybrid parallel performance comparisons between PMFS and MFS in eight SMP cluster machines up to totally 64

CPUs, and multithreaded parallel performances in a 16-way SMP are presented. The tested cluster is eight SMP computing nodes which have 8 CPUs and 64GB physical memory in every node. Another tested machine has dual eight-core Intel Xeon processors with 512GB main memory. The brief information of the SMP cluster and 16-way SMP are described in Tab. 9 and 10, respectively. All tests in this section were computed with only physical memory to measure the accurate parallel performances of those two algorithms. IO operation from the out-of-core storage is conducted sequentially even in the parallel algorithm, so a solution procedure with the out-of-core storage deteriorates the parallel performance.

Table 9: The specification of the tested SMP cluster

No. of Nodes	8 computing nodes, 1 login node
CPU	Intel Xeon X5570 2.93GHz X 2, 8cores/node, 64cores/cluster
Memory	DDR3 400MHz, 64GB/node, 512GB/cluster
Network / MPI	Mellanox Infiniband 10Gbps / MS-MPI in HPC Pack 2008 R2
OS	Microsoft Windows Server 2008 R2 HPC Edition, x64 version

Table 10: The specification of the 16-way SMP machine

CPU	Intel Xeon E7-2830 2.13GHz X 2, 16 cores
Memory	512GB
OS	Microsoft Windows Server 2008 R2 x64 Enterprise Edition

Table 11: Problem information for measuring the parallel efficiency

	No. of DOF	**No. of Processors**	**Numerical Operation**
1600x1600	15, 379, 206	$1 \sim 64$	2.55E+13
1270x1270	9, 692, 646	32	1.28E+13
1060x1060	6, 754, 326	$1 \sim 16$	6.01E+12
860x860	4, 447, 926	8	2.84E+12
700x700	2, 948, 406	4	1.56E+12
530x530	1, 691, 766	2	6.82E+11
430x430	1, 114, 566	1	3.60E+11

Two types of parallel performances were measured. The first test was the parallel speedup with constant problem size defined as:

$$S_p = \frac{V_p}{V_1} \tag{3}$$

where p is the number of processors, S_p is the parallel speedup of p processors and V_p is the computing speed of the algorithm by p processors, respectively. The computational speed was normalized in terms of multithreaded algorithm. The factorization phase and substitution phase of the solver was considered; the other part of computing time was so relatively small that they are ignored in this test.

Second parallel speedup is scalability test defined as:

$$S_p = \frac{V_{p,n(p)}}{V_{1,n(1)}} \tag{4}$$

where $n(p)$ is the function of the problems size in terms of p. The scalability means that the program can solve the problem with the processor proportional amount of the target work. For example, if an ideal scalable program can handle a job of size 1 within 1 processing unit, the software can treat a job of size 3 within 3 processing units. The ideal parallel scalable algorithm shows the same computing time if the group of processors proportional to the computations is engaged. For a precise evaluation, the ratios of the computations over the processors of the problems summarized in Tab. 11 were similar.

First, multithreaded parallel performances are presented in Tab. 12. Tested problems are from 430x430 to 1060x1060 in Tab. 11.

Table 12: Parallel speedup and parallel scalability of PMFS and MFS in 16-way SMP

	No. of threads	**1**	**2**	**4**	**8**	**16**
Parallel Speedup	**PMFS**	1.00	1.97	3.60	6.40	9.67
	MFS	1.04	1.93	3.07	4.30	5.30
Parallel Scalability	**PMFS**	1.00	2.00	3.84	7.27	11.2
	MFS	1.04	1.88	3.08	4.09	6.12

As shown in Tab. 12 and Fig. 12, two kinds of parallel speedup of PMFS are more similar to the ideal parallel performance than MFS in a 16-way SMP. Differing from MFS, PMFS is all parts of the multifrontal algorithm including assembling element stiffness matrices, merging matrices, saving intermediate frontal data blocks and caching the factored data blocks. Numerical operations in BLAS and LAPACK are parallelized already by their vender, so MFS conducts the parallel operations only in these routines.

Another reason for these differences of parallel speedup is in the numerical routines in BLAS and LAPACK. Those numerical routines are optimized by the maker to achieve the maximum performance of their computer architecture. However, even

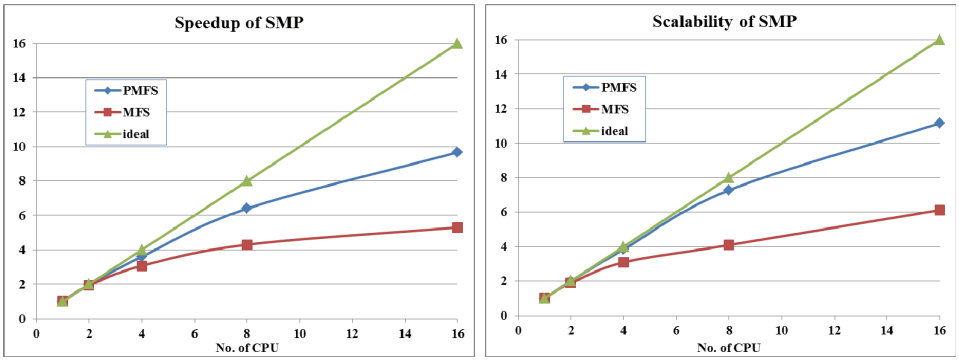

Figure 12: Parallel speedup and parallel scalability of PMFS and MFS in 16-way SMP

with the effort of the manufacturer, the numerical performances of the small size matrices or vectors are generally worse than larger one. MFS sequentially factors smaller size of factorization matrix in the lower elimination levels. Although the small matrices also are computed by the parallel routines, their parallel efficiencies are poor. In contrast, PMFS simultaneously calculates several small sizes of factorization matrices at once in the divided working layer. Because Intel MKL does not support the nested parallelism, numerical routines, called in parallel from the divided working layer of PMFS, execute serially their job. Numerical performance of serial execution of the small jobs in a parallelized loop is better than parallel execution of the small jobs in a serial loop. Simultaneous calculations of the serial jobs enhance the whole numerical performance

Second, hybrid parallel performances both with multithreads and MPI are measured as shown in Tab. 13 and Fig. 13. Tested problems are from 860x860 to 1600x1600 in Tab. 11.

Table 13: Parallel speedup and parallel scalability of PMFS and MFS with MPI

	No. of Nodes	**1**	**2**	**4**	**8**
Parallel Speedup	**PMFS**	1.00	1.70	2.71	3.46
	MFS	0.90	1.67	2.38	3.18
Parallel Scalability	**PMFS**	1.00	1.74	3.17	3.66
	MFS	0.65	1.31	2.66	3.36

Similar to multithreaded parallel performances, multithreaded solution algorithm

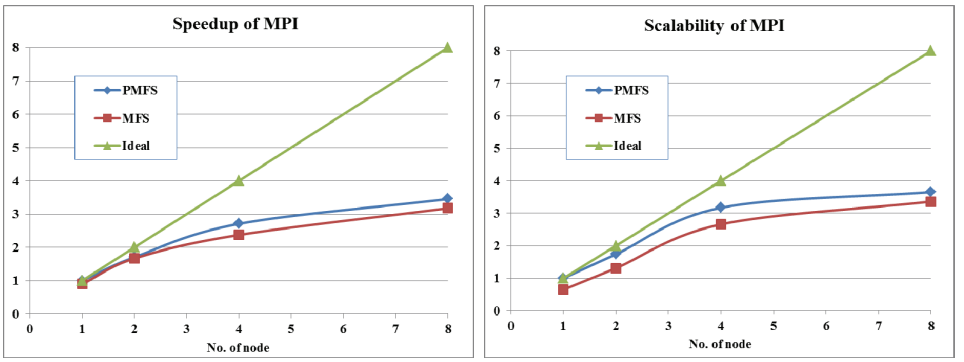

Figure 13: Parallel speedup and parallel scalability of PMFS and MFS with MPI

is better than single threaded algorithm for the distributed parallel computing environment. However, the performance gap between PMFS and MFS is not significant compared to the multithreaded performances. Moreover, the speedup is not as good as the multithreaded speedup. This is because the parallel factorization stages on the distributed computing nodes are conducted only with MPI communication and does not adopt multithreaded parallelism. By this test, current hybrid parallelized algorithm should be upgraded for a distributed SMP cluster. Future work should focus on the optimized implementation of distributed parallel factorization on a SMP clusters.

6 Applications

To apply the improved MFS, several problems were carried out. The first is the direct numerical simulation (DNS) of the composite material, and the second is natural frequency analysis of a whole body structure of airplane model.

6.1 Direct Numerical Simulations of Composite Materials

Composite materials are used widely in the aerospace industry. Currently, the considerable requirements of mass reductions of aerospace vehicles are essential to reduce the fuel cost and CO2 emission for the flight. Composite materials are commonly used not only in airplanes but also in satellites to reduce their mass but still have the desired structural stiffness. AIRBUS A380 and BOEING 787 Dreamliner exploit the state-of-the-art of the composite materials.

DNS of the composite material simulates 3-dimensional microscopic behaviors without any assumption of a structure, such as a beam or shell. DNS is conducted

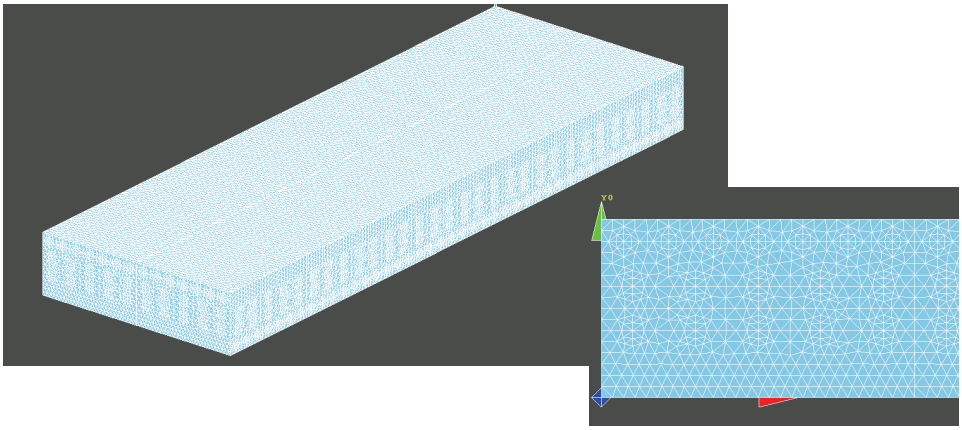

Figure 14: Finite element mesh of the specimen of the woven composite plate

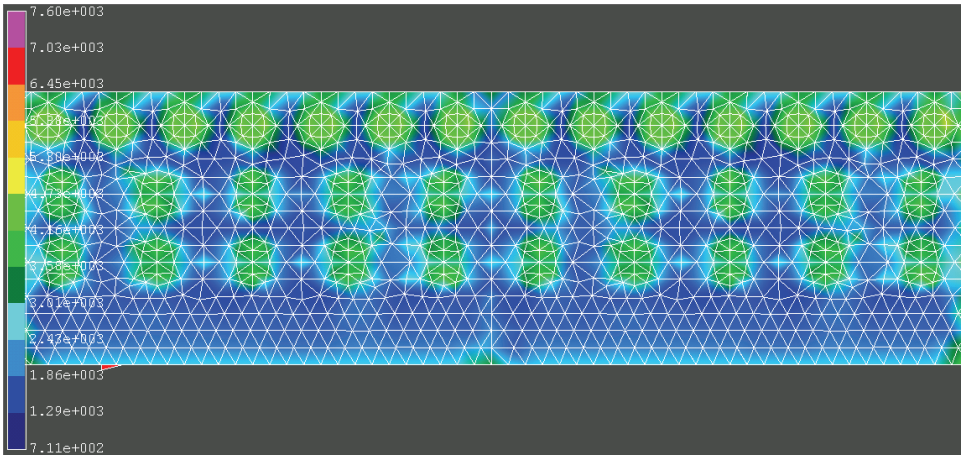

Figure 15: Normal stress distribution of the tensile direction (σ_{xx}) of the specimen

in 3-D scale, so it consumes considerable computing resources and man hours to model it as a finite element problem. DNS analysis is not widely used in practical work because of these drawbacks. The proposed PMFS can solve these large scale problems in a single or multiple computers within a reasonable time.

The application problem is a finite element model of a specimen of a woven composite plate, which has 1, 152, 000 8 nodes hexahedral elements and 3, 713, 328 unknowns, as shown in Fig. 14. It is composed woven style stencils of 9 orthotropic materials. The specimen is enforced to stretch 0.1 in X-direction at the left side of the plate, and the other side is constrained to fix the specimen. Fig 15 shows the

normal stress distribution of the tensile direction (σ_{xx}) of DNS test. This model requires nearly 50GB of total physical memory. The elapsed time of this problem, solved by eight computing nodes described in Tab. 9, is approximately 150 seconds including the numerical factorization and the substitution.

6.2 Natural Frequency Analysis of an Whole Aircraft Structure with Hybrid Parallel PMFS

Natural frequency analysis is widely used to analyze a structure in order to identify its dynamic characteristics and to avoid the resonance of the structure. Natural frequency is known from the eigenvalues of the structure. The most commonly used method of eigenvalue analysis is block Lanczos algorithm [Grimes, Lewis and Simon (1994)], which is the improvement of the original Lanczos method [Simon (1984)] to extract multiple eigensolutions. The intermediate equations of the system are bound together into a block, and are solved by the proper equation solver in the Lanczos iteration. In our work, the equation solver is multifrontal method.

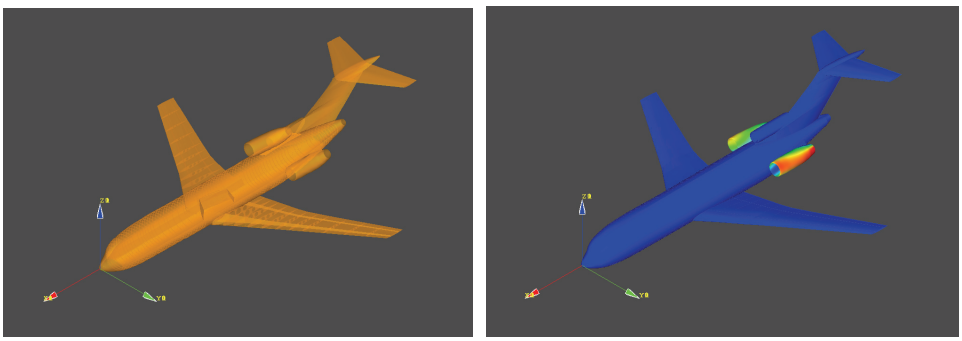

Figure 16: Finite element model of B727 (left) and its 9th eigenmode (right)

Second application of hybrid PMFS is the natural frequency analysis of a whole body structure of B727 developed by Boeing in 1963. The tested model has 1, 465, 612 four- node quadrilateral elements and 28, 046 three-node triangular elements, and the total number of DOF is 8, 714, 406, as shown in left picture of Fig. 16. The main purpose of this problem is to extract 10 eigenvalues and their eigenmodes with rigid body motions of the structure. This model has neither boundary condition nor constraint, so its eigensolutions must have six rigid body modes with zero eigenvalue. To avoid the singularity due to the rigid body modes, a small negative real number is applied as the shift value in block Lanczos algorithm. Right one of Fig. 16 represents 9th deformed mode shape of the problem.

The intermediate shift-inverted eigensolution is solved by the hybrid parallel multifrontal solver in SMP cluster as described in Tab. 9. The total number of Lanczos iteration is 10. The amount of physical memory consumed by the solver is approximately 45GB and the elapsed time of the numerical factorization and that of 10 times repetitive substitution are 107.2 seconds and 97.5 seconds, respectively.

7 Concluding Remarks

Hybrid parallelization of the multifrontal solution method for distributed symmetric multiprocessor machines has been presented. Numerical factorization of the parallelized solution method consists of divided working layer, co-working layer and distributed computing layer. At the divided working layer, each thread factors their internal unknowns in terms of adjacent interface unknowns. All threads cooperate in their tasks at the co-working layers. In a distributed computing layer, the matrix is stored in block cyclic manner and flexible matrix block size is used to implement efficient data communication among computing nodes. Multithreaded parallelization makes the multifrontal solver performance better in a symmetric multiprocessor machine.

The main drawback of the parallelized method is that parallelization requires more physical memory than the serial algorithm. Pairing and splitting the factorization matrix are proposed to reduce the size of required physical memory. Pairing the factorization matrix is to combine two factorization matrices into a square matrix. Two paired matrices are located in the lower and upper half part of the square matrix. Splitting the factorization matrix is to partition the factorization matrix into three kinds of submatrices, and to allocate them in their own separated space. Splitting the matrix removes the redundant space for the sub-matrices by allocating submatrices in a cache space for factored (CSF). The triplet matrix is designed to implement the split factorization matrix. Pairing and splitting the factorization matrix significantly reduce the total physical memory consumed by the solver.

Data caching and recovery is adapted to enhance the solution performance. Selective data caching and selective data recovery is used to increase the cacheable and recoverable data blocks, respectively. The original motivation of selective caching and recovery was proposed for serial algorithm of multifrontal solution method to utilize the physical memory rather than the out-of-core storage. The main idea of selective caching and recovery can be applied to the paralleled method by the nature of parallelism of the numerical factorization and the substitution.

The parallel algorithm was tested with a range of examples by the linear static analysis and eigenvalue analysis. All tests reported the efficiency of the parallelized algorithm. Parallel speedup and parallel scalability benchmark also verified that par-

allelization enhances the solution performance. The parallel algorithm is efficient for not only the linear solution of the system but also the analysis of eigenvalue.

Acknowledgement: This study was supported by the National Space Lab program of National Research Foundation of Korea. (Contract No. 2009-0092052)

References

.**Agullo, E.;A. Guermouche and J. Y. L'Excellent** (2008): A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory. *Parallel Computing*. vol. 34, no. 6, pp. 296-317.

Amestoy, P. R.;T. A. Davis and I. S. Duff (1996): An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*. vol. 17, no. 4, pp. 886-905.

Amestoy, P. R.;I. S. Duff and J. Y. L'Excellent (2000): Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering*. vol. 184, no. 2-4, pp. 501-520.

Amestoy, P. R.;I. S. Duff;J. Y. L'Excellent and J. Koster (2002): A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*. vol. 23, no. 1, pp. 15-41.

Amestoy, P. R.;A. Guermouche;J. Y. L'Excellent and S. Pralet (2006): Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*. vol. 32, no. 2, pp. 136-156.

ARIOLI, M.;J. DEMMEL and I. DUFF (1989): Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*. vol. 10, no. 2, pp. 165-190.

Chen, Y.;T. A. Davis;W. W. Hager and S. Rajamanickam (2008): Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*. vol. 35, no. 3, pp. 22.

Davis, T. A. (2004): Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*. vol. 30, no. 2, pp. 196-199.

Davis, T. A. (2004): A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*. vol. 30, no. 2, pp. 165-195.

Davis, T. A. and W. W. Hager (2009): Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Transactions on Mathematical Software (TOMS)*. vol. 35, no. 4, pp. 1-23.

Demmel, J. W.;J. R. Gilbert and X. S. Li (1999): An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*. vol. 20, no., pp. 915-952.

Dobrian, F. and A. Pothen (2000): Oblio: a sparse direct solver library for serial and parallel computations, Citeseer.

Duff, I. S. and J. K. Reid (1983): The Multifrontal Solution of Indefinite Sparse Symmetric Linear-Equations. *ACM Transactions on Mathematical Software (TOMS)*. vol. 9, no. 3, pp. 302-325.

Duff, I. S. and J. A. Scott (1993): MA42 - a new frontal code for solving sparse unsymmetric systems. *Rutherford Appleton Laboratory Report RAL-93-064*. no.

Grimes, R. G.;J. G. Lewis and H. D. Simon (1994): A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems. *SIAM Journal on Matrix Analysis and Applications*. vol. 15, no. 1, pp. 228-272.

Gupta, A. (2003): Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM Journal on Matrix Analysis and Applications*. vol. 24, no. 2, pp. 529-552.

Gupta, A.;G. Karypis and V. Kumar (1997): Highly scalable parallel algorithms for sparse matrix factorization. *Parallel and Distributed Systems, IEEE Transactions on*. vol. 8, no. 5, pp. 502-520.

Henon, P.;P. Ramet and J. Roman (2002): PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Computing*. vol. 28, no. 2, pp. 301-321.

Irons, B. M. (1970): A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*. vol. 2, no. 1, pp. 5-32.

Joshi, M.;G. Karypis;V. Kumar;A. Gupta and F. Gustavson (1999): PSPASES: An efficient and scalable parallel sparse direct solver, Citeseer.

Karypis, G. and V. Kumar (1998): A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*. vol. 20, no. 1, pp. 359-392.

Kim, J. H.;C. S. Lee and S. J. Kim (2005): High-performance domainwise parallel direct solver for large-scale structural analysis. *AIAA journal*. vol. 43, no. 3, pp. 662-670.

Kim, S. J.;C. S. Lee and J. H. Kim (2002): Large-scale structural analysis by parallel multifrontal solver through Internet-based personal computers. *AIAA journal*. vol. 40, no. 2, pp. 359-367.

Li, X. S. and J. W. Demmel (2003): SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions*

on Mathematical Software (TOMS). vol. 29, no. 2, pp. 110-140.

Liu, J. W. H. (1985): Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software (TOMS)*. vol. 11, no. 2, pp. 141-153.

Liu, J. W. H. (1986): On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software (TOMS)*. vol. 12, no. 3, pp. 249-264.

Pellegrini, F. and J. Roman (1996): Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, Springer.

Reid, J. and J. Scott (2008): An efficient out-of-core sparse symmetric indefinite direct solver, Technical Report RAL-TR-2008-024, Rutherford Appleton Laboratory.

Reid, J. K. and J. A. Scott (2009): An out-of-core sparse Cholesky solver. *ACM Transactions on Mathematical Software (TOMS)*. vol. 36, no. 2, pp. 9.

Rotkin, V. and S. Toledo (2004): The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software (TOMS)*. vol. 30, no. 1, pp. 19-46.

Schenk, O. and K. Gartner (2004): Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*. vol. 20, no. 3, pp. 475-487.

Schenk, O. and K. Gartner (2006): On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*. vol. 23, no., pp. 158-179.

Simon, H. D. (1984): The Lanczos-Algorithm with Partial Reorthogonalization. *Mathematics of Computation*. vol. 42, no. 165, pp. 115-142.

Toledo, S.;D. Chen and V. Rotkin (2003): Taucs: A library of sparse linear solvers.