J. Comput. Struct. Eng. Inst. Korea, 30(2)
pp.127~135, April, 2017
https://doi.org/10.7734/COSEIK.2017.30.2.127

pISSN 1229-3059 eISSN 2287-2302
Computational Structural
Engineering Institute of Korea

멀티코어 CPU를 갖는 공유 메모리 구조의 대규모 병렬 유한요소 코드에 대한 설계 고려 사항

조 정 래[1] · 조 근 희[1†]

[1]한국건설기술연구원 구조융합연구소

Design Considerations on Large-scale Parallel Finite Element Code in Shared Memory Architecture with Multi-Core CPU

Jeong-Rae Cho[1] and Keunhee Cho[1†]

[1]*Structural Engineering Research Institute, Korea Institute of Civil Engineering and Building Technology, Goyang, 10223, Korea*

Abstract

The computing environment has changed rapidly to enable large-scale finite element models to be analyzed at the PC or workstation level, such as multi-core CPU, optimal math kernel library implementing BLAS and LAPACK, and popularization of direct sparse solvers. In this paper, the design considerations on a parallel finite element code for shared memory based multi-core CPU system are proposed; (1) the use of optimized numerical libraries, (2) the use of latest direct sparse solvers, (3) parallelism using OpenMP for computing element stiffness matrices, and (4) assembly techniques using triplets, which is a type of sparse matrix storage. In addition, the parallelization effect is examined on the time-consuming works through a large scale finite element model.

Keywords : *finite element code, multi-core CPU, OpenMP, sparse solver*

1. 서 론

컴퓨터 하드웨어 및 소프트웨어의 급속한 발전이 이루어짐에 따라 이를 유한요소 해석에 적용하려는 노력이 지속적으로 이루어지고 있다. 특히 멀티코어 CPU가 일반화됨에 따라 개인용 컴퓨터 또는 워크스테이션에서도 대규모 고성능 연산이 가능한 하드웨어 기반이 마련되었으며, 그 성능을 극대화할 수 있는 여러 소프트웨어적인 노력이 추진되고 있다. 이 논문에서는 멀티코어 CPU를 갖는 공유 메모리 구조에 대한 병렬 유한요소 프로그램 설계시 고려사항을 제시하고, 대규모 수치모델을 통해 그 효과를 검토하였다.

유한요소 프로그램은 (1) 요소 행렬 및 개별 하중 벡터의 계산, (2) 전체 행렬과 전체 하중 벡터의 계산, (3) 선형 방정식 또는 고유치 문제를 통해 해의 계산(변위 계산 등), (4) 계산된 해를 이용한 후처리(하중 계산, 응력 계산 등) 등으로 구성된다. (3)에 해당하는 $Ax=b$ 형태의 선형방정식(linear system of equations)이나 $Ax=\lambda x$ 또는 $Ax=\lambda Mx$ 형태의 고유치 문제(eigenvalue problems)를 해석하는데 가장 많은 시간과 메모리가 필요하다. 따라서 멀티코어 CPU를 갖는 현대적인 컴퓨터 환경을 반영한 최적화된 수치라이브러리 및 최신 희소 솔버를 적용할 필요가 있다. (2)는 어셈블 단계로 불리는데 기존 방식은 구속조건 등을 고려한 전체 행렬에 대한 희소행렬 패턴을 미리 계산한 후 요소 행렬의 각 항을 희소행렬 패턴내의 위치를 검색하여 추가하는 형태로 구성된다. 이와 같은 방식은 유한요소 프로그램 설계의 복잡도를 대폭 증가시키는 가장 큰 원인이 된다(Cho, 2009).

본 연구에서는 유한요소법의 수치해석 관점에서 최신 컴퓨팅 환경 변화를 조사하였다. 이러한 환경 변화를 고려하여 멀티코어 CPU를 갖는 공유 메모리 구조에 대한 병렬 유한요소 프로그램 설계시 중요 고려 사항으로 (1) 최적화된 수치라이브러리의

† Corresponding author:
Tel: +82-31-910-0132; E-mail: kcho@kict.re.kr
Received December 29 2016; Revised January 3 2017;
Accepted January 10 2017

사용, (2) 최신 직접 희소 솔버의 사용, (3) OpenMP를 이용한 병렬 요소 강성 행렬의 계산, (4) 희소행렬 저장방식의 일종인 triplet을 이용한 어셈블 기법 등을 제시하였다. 이중 (1)과 (2)는 적절한 라이브러리를 선택함으로써 달성가능하며, (3)과 (4)는 병렬 유한요소 코드의 설계 관점에서 보다 코드의 복잡도를 낮추는 새로운 방법을 제시하였다. 또한 대규모 유한요소 모델을 통해 많은 시간이 소요되는 작업을 기준으로 병렬화 효과를 검토하였다.

2. 최적화된 수치라이브러리

수치해석과 관련된 컴퓨터 하드웨어의 변화 중 가장 큰 변화는 캐쉬 메모리를 장착한 CPU와 멀티코어 CPU의 등장이다. 초창기 데스크탑용 CPU는 CPU의 동작속도와 메인메모리의 동작속도가 큰 차이가 없었다. 이후 기술의 발전에 따라 CPU의 동작속도가 메인메모리에 비해 상대적으로 빨라졌으며, CPU와 메모리의 속도 차이에서 오는 성능저하를 극복하기 위해 CPU 내에 캐쉬 메모리라 불리는 고속의 메모리를 장착하게 된다. 보다 최근에는 하나의 CPU내에 여러 개의 코어(core)가 존재하는 CPU가 등장하고 있다. Fig. 1은 최근의 Intel CPU를 나타내고 있으며, Fig. 2는 비교적 저렴한 비용으로 구성할 수 있는 2개의 CPU를 장착한 워크스테이션의 구조를 나타낸 것이다. Fig. 1과 Fig. 2와 같이 여러 코어가 동일한 공유 메모리에 억세스하는 시스템을 대상으로 병렬 프로그램을 수행하는 프로그래밍 모델을 공유 메모리 구조 병렬화(parallelism for share memory architecture)라고 하며 일반적으로 병렬 프로그래밍 API인 OpenMP가 사용된다(Cho, 2009).

캐쉬 메모리가 장착된 CPU나 multi-core CPU의 성능을 최대한 이끌어 내기 위해서는 최근 대중화된 최적 수치라이브러리(math kernel library)를 적극적으로 활용해야 한다. 이들 라이브러리는 벡터과 행렬 간의 기본 연산을 정의한 BLAS (basic linear algebra subprogram), 꽉찬 행렬(dense

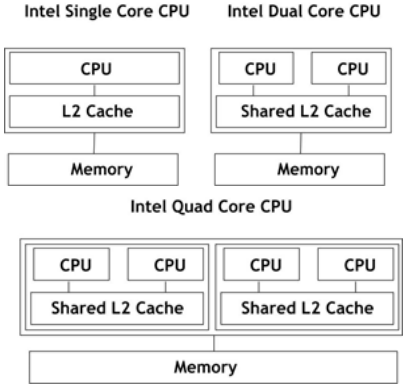

Fig. 1 Recent CPUs with memory hierarchy

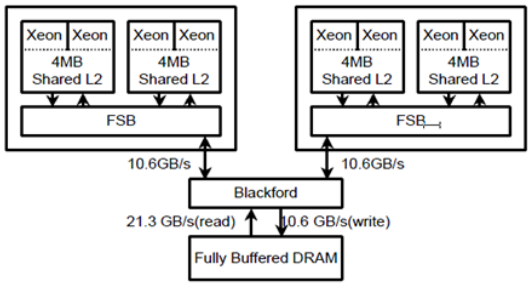

Fig. 2 SMP machine(2 Xeon: total 8 processors)

matrix)에 대한 선형방정식과 고유치문제의 해를 풀 수 있는 함수 집합인 LAPACK(Linear Algebra PACKage) 등의 표준 수치 라이브러리 인터페이스를 캐쉬 메모리와 멀티코어 CPU의 성능을 극대화할 수 있도록 CPU 차원에서 최적화한 라이브러리를 의미한다. 다음은 대표적인 최적 수치라이브러리를 정리한 것이다.

- Intel MKL : Intel 제공. BLAS, LAPACK, PBLAS, ScaLAPACK 등. 최근 무료화
- AMD ACML : AMD 제공. BLAS, LAPACK 등. 개발종료
- ATLAS : 오픈 소스 수치라이브러리, 일부 LAPACK 포함. 개별 머쉰에서 인스톨시 최적화 수행
- GotoBlas2 : 오픈 소스 수치라이브러리
- OpenBlas : GotoBlas를 수정하여 작성한 오픈 소스 수치라이브러리

현재 가장 흔히 사용되는 최적 수치라이브러리는 Intel MKL과 OpenBlas이다. 최적 수치라이브러리들은 모두 1개 프로세서를 사용하는 시퀀셜 버전(sequential version)과 멀티코어 CPU 등에서 여러 개의 프로세서를 사용하는 멀티스레드 버전(multi-threaded version)을 제공하고 있다. 시퀀셜 버전의 경우에도 캐쉬 메모리의 특성을 사용하기 때문에 간단한 행렬 연산이라 하더라도 직접 코딩한 것보다 계산 속도가 빠르며, 멀티스레드 버전은 추가적으로 OpenMP를 기반으로 멀티코어의 성능을 활용하게 된다. 본 연구에서는 비교적 성능이 우수하고 최근 무료화된 Intel MKL을 사용하였다.

3. 직접 희소 솔버

유한요소법에서는 $Ax=b$ 형태의 선형 방정식이나 $Ax=\lambda x$ 또는 $Ax=\lambda Mx$ 등과 같은 고유치 문제를 풀어야 한다. 이때 강성행렬 A와 질량행렬 M 등의 시스템 행렬은 희소성을 갖는 특징을 가지고 있다(Fig. 3 참조). 따라서 대규모 문제를

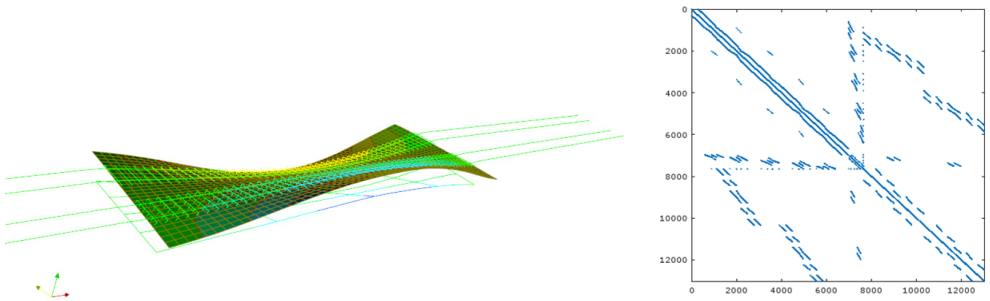

Fig. 3 Finite element model of plate girder railway bridge and its stiffness matrix

해결하기 위해서는 강성 행렬의 희소성을 고려하여 경제적으로 해를 계산하는 알고리즘을 구비한 희소 솔버(sparse solver)를 사용해야 한다. 선형 방정식 $Ax = b$의 해를 구하는 선형 방정식 희소 솔버(sparse solver for linear systems of equations)는 적용되는 알고리즘에 따라 직접 희소 솔버(direct sparse solver)와 반복 희소 솔버(iterative sparse solver)가 있다. 최근까지 대규모 희소 행렬에 대한 직접 및 반복 희소 솔버에 대해 많은 연구가 진행되었으며(Sadd, 2003; Davis, 2006), PARDISO, MUMPS, CHOLMOD, UMPFACK 등 여러 효율적인 희소 솔버 소프트웨어 패키지가 직접 희소 솔버를 중심으로 개발되었다. 한편 2000년대 들어 대부분의 범용 유한요소 프로그램이 희소 솔버를 도입하였으며, 전통적인 스카이라인 솔버(skyline solver)에 비해 상당히 계산속도가 향상되었다(ADINA, 2005; ABAQUS, 2007; ANSYS, 2007). Lim 등(2016)은 지반-구조물 상호작용 해석 코드인 KIESSI-3D에 p-version 동적무한요소와 직접 희소 솔버인 PARDISO를 적용하는 등의 개선을 통해 기존 대비 3.4~25.6배까지 계산속도가 향상되었다고 보고하고 있다.

직접 희소 솔버는 한번 행렬 분해(factorization)을 수행하고 나면 여러 번 우변 벡터(right-hand side vector)에 적용 가능하며, 행렬의 상태(condition)가 나쁘더라고 동일한 시간이 소요되는 등의 장점을 가지고 있다. 하지만 필인(fill-in, 원래 0인 항이나 행렬 분해 도중에 0이 아닌 항으로 바뀌는 항)의 존재로 인해 메모리가 많이 소요되는 단점이 있다. Multifrontal, supernodal methods 등의 알고리즘을 사용하는 직접 희소 솔버는 그래프 이론(graph theory)에 따라 연산회수와 필인을 최소화하며, 연산도중 생성되는 작은 크기의 꽉찬 행렬(dense submatrices)을 최적화된 BLAS와 LAPACK을 이용함으로써 빠르게 해를 구할 수 있다(Davis, 2006). 직접 희소 솔버는 Fig. 4와 같이 전통적인 직접 솔버와 달리 필인을 최소화하는 분석 단계(analysis phase)를 포함하며 3단계로 구성된다. 이 분석 단계는 전통적인 밴드 솔버(band solver)나 스카이라인 솔버에서 행렬의 대각성분으로부터 0이 아닌 항의 분산폭으로 정의되는 밴드폭을 최소화하도록(band-width minimization) 절점 번호 또는 수식 번호를 최적화하는 것과 동등한 역할을 한다. 따라서 최신 직접 희소 행렬 솔버를 적용할 경우 별도의 절점 번호 또는 수식 번호에 대한 최적화 과정이 불필요하다.

반복 희소 솔버는 Jacobi, Gauss-Seidal, Symmetric Successive Over-Relaxation(SSOR) 등 고전적인 반복법과 Krylov 반복법을 적용한 솔버로 구분할 수 있다. 유한요소법에서는 positive definite 대칭 행렬을 위한 Conjugate Gradient Method(CG), 일반 행렬을 위한 Generalized Minimum Residual Method(GMRES) 등을 주로 적용하며 둘 다 Krylov 반복법에 근거한다. Krylov 반복법에서는 반복 횟수를 줄이기 위한 선형방정식에 preconditioner라 불리는 행렬의 역행렬을 곱한 선형방정식을 대상으로 해를 구하게 되는데, preconditioner는 고전적인 반복법에 근거하거나 부정확한 LU 분해한 결과(Incomplete LU) 등을 사용한다(Saad, 2003). 반복 희소 솔버는 해를 구할 때 추가적인 메모리가 필요없으며, 특히 대규모 3차원 솔리드 모델에서 직접 희소 솔버보다 빠른 계산이 가능하다. 하지만, 셸 구조와 같은 행렬 상태가 나쁜(ill-conditioned) 문제에서 수렴을 하지 않거나 반복회수가 늘어나는 단점이 있다(ADINA, 2005; Benzi, *et al.*, 1998; Benzi, *et al.*, 2000; Cho, 2009). 이러한 견고성(robustness)의 부족으로 인해 보통 범용 유한

```
Given Ax = y
1. Analysis phase       : Compute P from A's sparsity for minimizing fill-in
2. Factorization phase  : Compute LU = PA
3. Solution phase       : LUx = Py
```

Fig. 4 Solver phases of direct sparse solver

요소 프로그램에서는 직접 희소 솔버를 디폴트로, 반복 희소 솔버를 옵션으로 제공한다(ABAQUS, 2007). 반복 희소 솔버 역시 수식 번호 최적화를 수행할 때 효율이 높아지는 것으로 보고되고 있다(Benzi, 2002)

본 연구에서는 견고성이 우수하고 수식번호 최적화 기능이 내장되어 간단한 유한요소 코드 설계가 가능한 직접 희소 솔버인 PARDISO 솔버를 채용하였다. PARDISO는 University of Basel에서 개발하였으며, Intel MKL에 도입되어 무료로 사용 가능하며 성능이 우수한 것으로 알려져 있다.

4. OpenMP를 이용한 병렬 요소 강성 행렬 계산

본 연구에서는 공유 메모리 구조를 위한 병렬 프로그래밍 API인 OpenMP를 이용하여 모든 요소를 순회하여 요소 강성 행렬을 계산하는 과정을 병렬화하였다. 개별 요소의 계산시간이 다를 경우 개별 코어로 요소 강성 행렬의 계산 작업을 할당하는 스케줄링(scheduling)이 필요하며 이는 병렬화 작업의 효율성에 큰 영향을 미치게 된다.

본 연구에서는 동일한 요소 타입에 대한 병렬화를 순차적으로 진행하는 간단한 방식을 적용하였다. 예를 들어 유한요소 모델 내에 8절점 솔리드 요소와 4절점 쉘 요소가 혼재되어 있는 경우 8절점 솔리드 요소 집합에 대한 병렬 루프(parallel loop)를 실행하고, 이후 4절점 쉘 요소 집합에 대한 병렬 루프를 실행하는 방식으로 설계하였다.

이 방식은 동일한 정식화 과정을 거친 유한요소의 경우 요소 강성 행렬의 계산 시간이 동일하다는 가정을 통해 고안된 것이다. 만약 비선형 변형을 보일 경우 이 방식은 효율이 떨어질 수도 있으나 요소 강성 행렬의 병렬 계산을 간단하게 구현할 수 있는 장점이 있다.

5. Triplet을 이용한 어셈블 과정

5.1 희소 행렬 저장 방식

희소 행렬 솔버는 CSR(compressed sparse row), CSC (compressed sparse column), triplet(또는 COO, Coordinate) 등 표준화된 희소 행렬 저장방식을 사용한다. Fig. 5는 시작 인덱스로 1을 채용한 희소 행렬 저장방식을 예시한 것이다. C 언어로 희소 행렬을 다룰 때는 0을 시작 인덱스로 채용하기도 한다. CSC는 열을 기준으로 0이 아닌 항에 순차적으로 인덱스를 부여한 후 각 열이 시작되는 점을 저장하는 정수 배열 $Aj(n+1)$와 행 번호를 저장하는 정수배열 $Ai(nnz)$, 값을 저장하는 실수 배열 $Ax(nnz)$로 구성된다. 이때 n은 열의 개수이고, nnz는 0이 아닌 항의 개수이다. CSR은 행을 기준으로 인덱스를 부여하는 방식으로 CSC와 유사하며 $Ai(m+1)$, $Aj(nnz)$, $Ax(nnz)$ 배열로 저장하며 이때 m은 행의 개수이다. Triplet은 0이 아닌 항의 i, j 인덱스와 값 x의 쌍인 (i, j, x)를 $Ai(nnz)$, $Aj(nnz)$, $Ax(nnz)$의 배열로 표현하는 방식이다(Davis, 2006)

희소 행렬 솔버는 빠른 위치 계산 등 계산의 효율성을 위해서 중복된 항이 없고 인덱스가 정렬된 CSC 또는 CSR 형식을 적용한다. 예를 들어 Fig. 6과 같이 중복항이 있거나

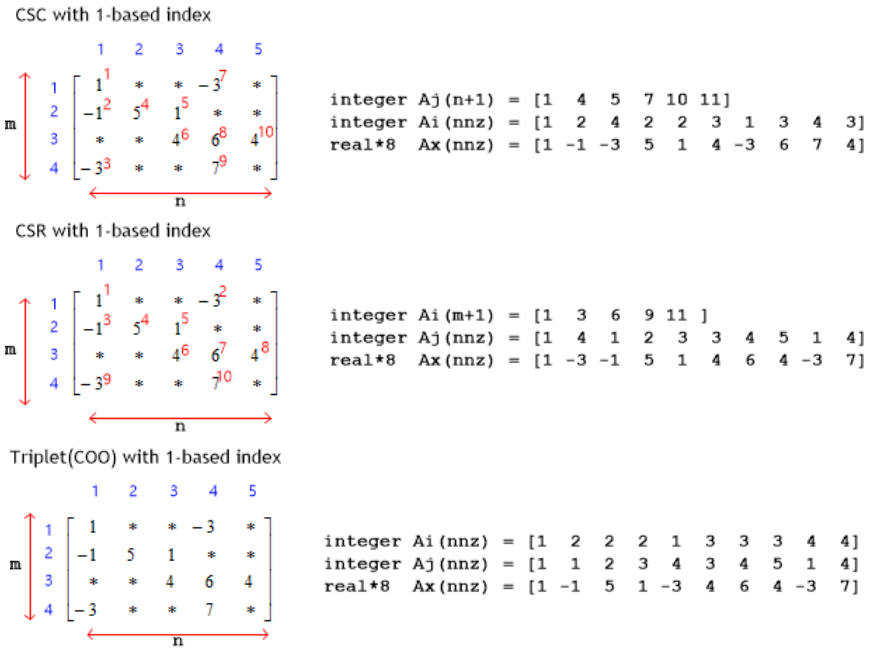

Fig. 5 Sparse matrix storage format(fortran style 1-based index)

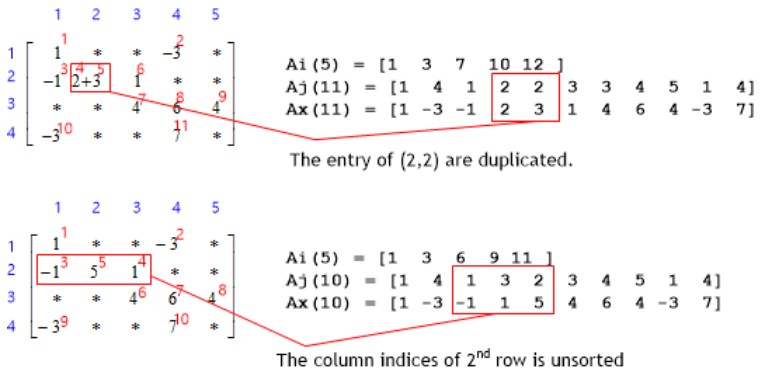

Fig. 6 CSC with duplicate entry and unsorted column indices

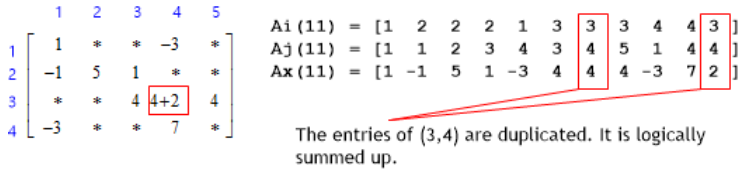

Fig. 7 Triplet with duplicate entries

열 번호가 정렬되지 않는 경우 사용할 수 없다. 반면에 triplet은 (i, j, x) 쌍을 사용하기 때문에 정렬의 개념이 없으며 중복항을 허용하는데 희소행렬의 교환(exchange)이나 입출력 등에 사용된다. 예를 들어 Fig. 7은 중복항을 갖는 triplet을 나타내고 있다. 중복항을 갖는 triplet을 CSC나 CSR로 변환할 경우 중복항이 합산되어 변환되게 된다.

5.2 Triplet을 이용한 어셈블 과정

CSC 또는 CSR 저장형식을 사용하는 희소 솔버를 사용하는 유한요소 코드에서 요소 행렬을 전체 행렬로 어셈블하는 방법은 다음과 같이 세가지 경우를 생각해볼 수 있다.

(1) Model preprocessing 단계에서 절점 번호에 대한 최적화(renumbering)를 수행하고 CSC 또는 CSR 형식의 0이 아닌 항의 위치를 미리 계산한 후, 요소 행렬을 어셈블할 때 각 항의 위치를 검색해 더해 나가는 방법

(2) Model preprocessing 단계에서 CSC 또는 CSR 형식의 0이 아닌 항의 위치를 미리 계산한 후, 요소 행렬을 어셈블할 때 각 항의 위치를 검색해 더해 나가는 방법. 자유도 번호의 최적화는 어셈블된 CSC 또는 CSR 형식을 대상으로 희소 솔버 수준에서 수행

(3) 요소행렬을 Triplet에 중복항을 허용하여 어셈블하고, 어셈블이 끝난 후 CSC나 CSR 형식으로 일괄 변환하는 방법. 자유도 번호의 최적화는 어셈블된 CSC 또는 CSR 형식을 대상으로 희소 솔버 수준에서 수행

방법 (1)은 유한요소 교과서(Bathe, 1995)에서 밴드 솔버나 스카이라인 솔버 등을 사용할 때 주로 사용했던 방법이다. 강성행렬의 밴드폭이나 스카이라인의 크기를 줄이기 위해서 미리 절점 번호의 최적화를 수행하는 특징을 가지고 있다. 방법 (2)는 방법 (1)과 유사하나 사전에 절점 번호의 최적화를 수행하는 대신 어셈블된 전체 행렬에 대해 자유도 번호를 대상으로 최적화를 수행하는 방식이다. CSC 또는 CSR를 사용하는 경우 사전 절점 번호 최적화에 따라 0이 아닌 항의 개수가 변경되지 않는다. 또한 직접 희소 솔버의 경우 분석 단계(analysis phase)에서 자유도 번호의 최적화를 자체적으로 수행하며, 반복 희소 솔버에서도 다양한 자유도 번호의 최적화 알고리즘이 제안되어 있다. 따라서 방법 (1) 보다는 방법 (2)가 더 유용한 방법이다. 방법 (3)은 model preprocessing 단계 없이 단순히 중복항을 고려한 triplet을 이용하고 CSC 또는 CSR로 일괄 변환하는 방식이다.

본 연구에서는 유한요소 모델 어셈블 방법으로 triplet을 이용한 방법 (3)을 채택하였다. 이 방법에서는 방법 (2)와 달리 CSC 또는 CSR 저장형식을 미리 계산하고 어셈블시 요소행렬의 각 항의 위치를 계산하는 번거러움이 없기 때문에 유한요소 프로그램의 설계 구조를 간략화할 수 있다. 또한 해석도중에 요소의 연결성(connectivity) 변경에 쉽게 대응이 가능하다. 방법 (2)와 비교할 경우 어셈블 속도는 이론적으로 동일하나 어셈블 도중 중복된 항이 존재하기 때문에 더 많은 메모리가 요구되는 단점이 있다. 메모리 문제는 어셈블 도중 Triplet을 CSC 또는 CSR로 변환하여 중복항을 제거하고 다시 triplet을 진행하는 방식으로 해결이 가능하다.

6. 수치예제 결과 및 고찰

본 연구에서 제안한 방법에 대한 효과를 검토하기 위해 Fig. 8과 같은 캔틸레버 모델을 사용하였다. 해석 모델에서는 8절점 솔리드 요소를 x, y, z 방향으로 n_x, n_y, n_z 개로 분할하여 해석한다. 방향별 길이는 3000, 30, 30으로 설정하였다. 수치실험은 Table 1에 표시한 것과 같은 메쉬의 크기를 변경한 4가지 모델에 대해 수행하였다. 가장 큰 해석 모델은 최대 자유도수가 200만개인 대규모 유한요소 모델이다. 표는 각 모델에 대한 요소수(No. of Elements), 절점수(No. of Nodes), 총 자유도수(No. of DOFs), 구속된 자유도를 제외한 수식 수(No. of Equations), 강성행렬에서 0이 아닌 항의 개수(No. of Nonzeros) 등이 표시되어 있다. M1은 강성행렬을 중복항이 없는 CSC나 CSR로 저장할 때 필요한 메모리, M2는 본 연구에서 제안한 triplet을 이용한 어셈블 방법을 사용할 때 소요되는 순간 최대 메모리, M3는 본 연구에서 사용한 직접 희소 솔버인 Intel MKL PARDISO에서 해를 구할 때 소요되는 순간 최대 메모리이다. Triplet을 이용하여 어셈블할 때 순간 최대 메모리가 많이 소요되지만 직접 희소 솔버를 사용할 때 소요되는 순간 최대 메모리 보다는 상당히 작음을 알 수 있다. 따라서 본 연구에서 제안한 어셈블 방법에서 메모리는 문제되지 않음을 알 수 있다.

Table 1에서 제시된 모델에 대한 수치실험은 Intel Xeon(R) E5-2687W v4@3.00GHz CPU 2기와 256 GB 메모리를 장착한 시스템에서 수행되었다. 시스템의 전체 코어수는 24개이다. Table 2와 Table 3은 코어 1개와 코어 24개를 사용할 때의 해석소요시간을 나타낸 것이다. State Determination Time은 전체 강성행렬을 계산하는 시간을 의미하며 Direct Sparse Solver Time은 Intel MKL PARDISO 솔버에서 해를 구하는데 소요되는 시간이다. State Determination Time은

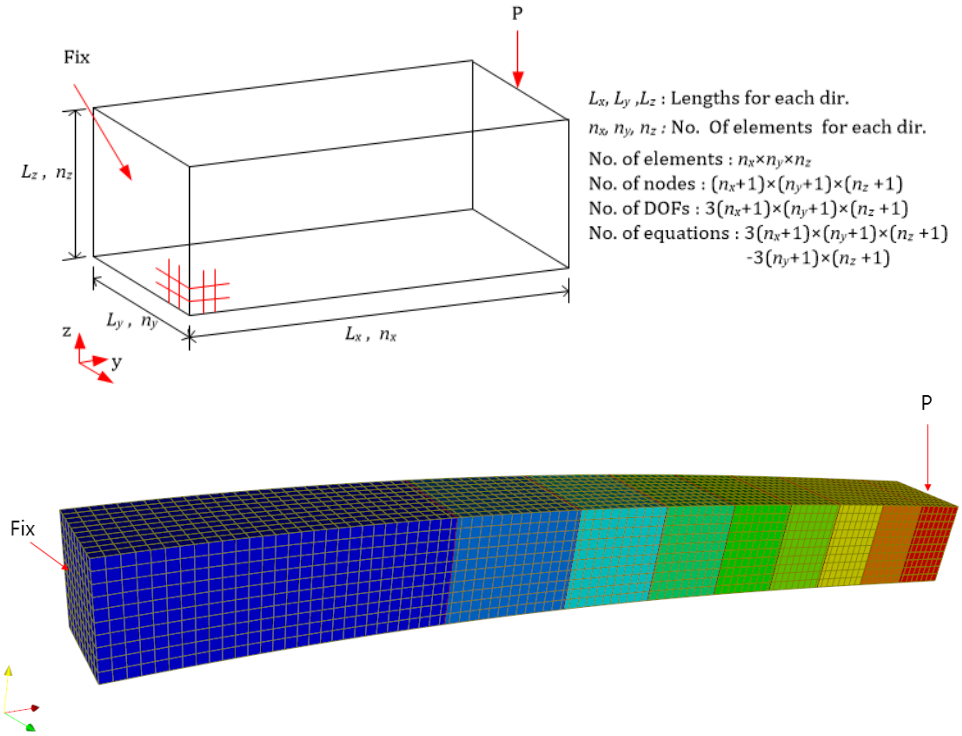

Fig. 8 Cantilever analysis model and C100x10x10 model

Table 1 Cantilever analysis model cases

Model	No. of Elements	No. of Nodes	No. of DOFs	No. of Equations	No. of non-zeros	M1 (MB)	M2 (MB)	M3 (MB)
C100x10x10	10,000	12,221	36,663	36,300	1,306,852	15	88	229
C200x20x20	80,000	88,641	265,923	264,600	10,145,512	117	703	2,519
C300x30x30	270,000	289,261	867,783	864,900	33,895,972	391	2,373	11,318
C400x40x40	640,000	674,081	2,022,243	2,017,200	79,938,232	923	5,625	36,417

*M1: Memory storing stiffness matrix in CSC/CSR storage
*M2: Peak memory assembling by triplet
*M3: Peak memory in PARDISO

Table 2 Timing of cantilever models - 1 core used

Model	State determination time(sec)			Direct sparse solver time(sec)			
	Element	Assembling	Total	Analysis	Factorization	Solution	Total
C100x10x10	0.355667	0.017641	0.373308	0.332523	1.3907	0.113749	1.836972
C200x20x20	1.96513	1.10086	3.06599	3.47487	32.665	1.43983	37.5797
C300x30x30	6.58962	4.21628	10.8059	14.8158	252.147	7.0591	274.0219
C400x40x40	15.6457	12.3295	27.9752	48.4076	1128.41	22.3869	1199.205

Table 3 Timing of cantilever models - 24 core used

Model	State determination time(sec)			Direct sparse solver time(sec)			
	Element	Assembling	Total	Analysis	Factorization	Solution	Total
C100x10x10	0.0444918	0.2290222	0.273514	0.313844	0.180964	0.0252515	0.52006
C200x20x20	0.125481	2.163509	2.28899	1.9719	2.81713	0.291406	5.080436
C300x30x30	0.36754	6.0274	6.39494	7.14851	20.8204	1.32534	29.29425
C400x40x40	0.795834	18.012766	18.8086	24.1359	113.54	3.9931	141.669

모든 요소를 돌며 요소 강성행렬 계산하는 시간(element로 표기)과 이를 강성행렬로 어셈블하는 시간(assembling으로 표기)으로 구성된다. Direct Sparse Solver Time은 Analysis, Factorization, Solution 등과 같이 Fig. 4에서 설명한 3단계로 구성된다. 표에서 Direct Sparse Solver Time이 State Determination Time보다 많은 시간이 소요되는 것을 알 수 있다. State Determination Time에서는 코어를 1개 사용할 때는 Element Time과 Assembling Time이 비슷하나 코어를 24개를 사용할 때는 Element Time은 병렬화가 잘되어 계산 시간이 감소되나 Assembling Time은 공유메모리를

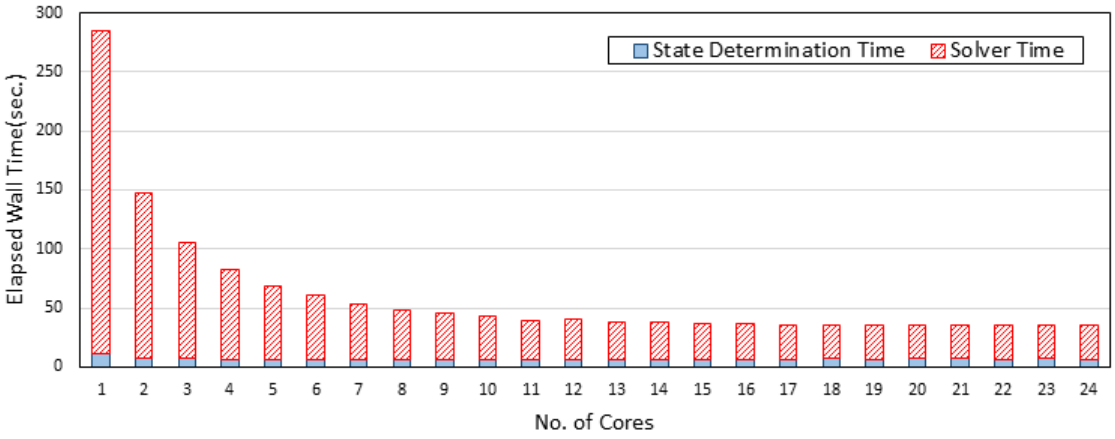

Fig. 9 Analysistime of C300x30x30

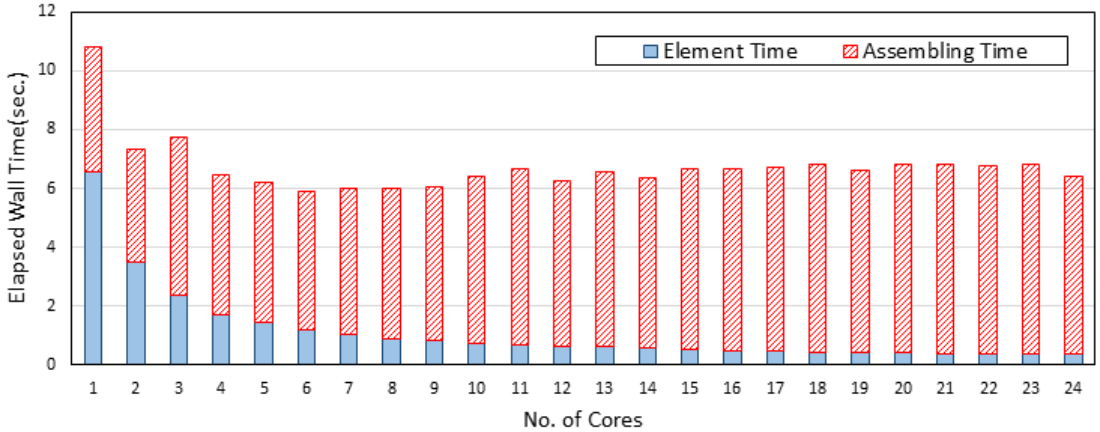

Fig. 10 State determination time of C300x30x30

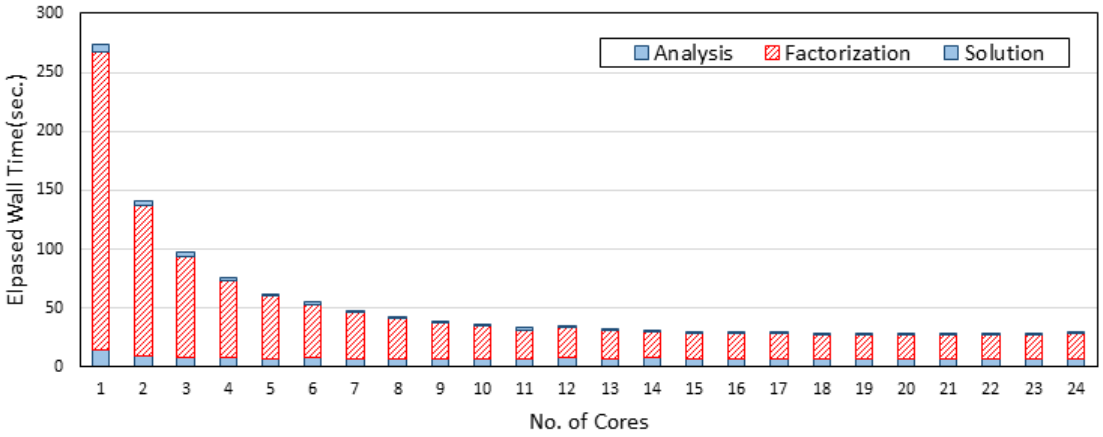

Fig. 11 Solver time of C300x30x30 in solver

대상으로 하는 대입 작업 등의 영향으로 병렬화가 불가능한 부분이 많아 코어수가 증가하더라도 계산시간이 감소되지 않는 것을 알 수 있다. Direct Sparse Solver Time내에서는 Numerical Factorization에 가장 많은 시간을 필요한 것을 알 수 있으며, 전반적으로 코어수가 많을 때 계산시간이 감소됨을 알 수 있다. 한편 200만개 이상의 자유도를 갖는 대규모 모델 C400×40×40에서의 계산시간이 약 160초 정도로 현대적인 컴퓨터 환경을 도입한 유한요소 코드의 성능을 확인할 수 있게 해준다.

Fig. 9~11은 약 86만개의 자유도를 갖는 C300×30×30 모델에 대한 코어별 계산 시간을 나타낸 것이다. State Determination Time 중 Element Time은 본 연구에서 제안한 방법에 따라 병렬화가 잘 이루어짐을 나타내고 있다. 반면에 이전 단락에서 언급한 것과 같이 Assembling Time에 대한 병렬화 효과는 거의 없다. Direct Sparse Solver Time의 경우 약 10개 정도의 코어까지는 병렬화가 비교적 잘 이루어지나 그 이후는 병렬화 효과가 떨어지는 것을 알 수 있다. 이는 병렬화가 불가능한 영역이 지배적이기 때문이다.

7. 결 론

이 논문에서는 멀티코어 CPU를 갖는 공유 메모리 구조에 대한 병렬 유한요소 프로그램 설계시 고려사항으로 (1) 최적화된 수치라이브러리의 사용, (2) 최신 직접 희소 솔버의 사용, (3) OpenMP를 이용한 병렬 요소 강성 행렬의 계산, (4) 희소 행렬 저장방식의 일종인 triplet을 이용한 어셈블 기법 등을 제시하였다. 이들 설계 사항은 비교적 코드 반영이 용이한 장점이 있다. 또한 약 200만 자유도를 갖는 대규모 수치모델을 통해 많은 시간이 소요되는 작업을 기준으로 병렬화 효과를 검토하였으며, PC나 워크스테이션에서 병렬화를 통해 충분히 대규모 모델을 계산할 수 있음을 확인하였다.

향후 연구에서는 반복 희소 솔버의 대규모 유한요소모델에 대한 적용성을 검토할 예정이며, Message Passing Interface (MPI)를 활용한 분산 메모리 구조(distributed memory architecture)에 대한 유한요소 코드 설계와 관련된 주제를 수행할 예정이다.

감사의 글

본 연구는 산업통상자원부(MOTIE)와 한국에너지기술평가원(KETEP)의 지원을 받아 수행한 연구 과제입니다(No. 2016 1520101130).

References

ABQUS Inc. (2007) ABAQUS Analysis User's Manual (Version 6.7).

ACML-AMD Core Math Library http://developer.amd.com/tools-and-sdks/archive/compute/amd-core-math-library-acml.

ADINA R&D Inc. (2005) Theory and Modeling Guide, ADINA System 8.3.

ANSYS Inc. (2007) Release 11.0 Documentation for ANSYS-Mutitibody Analysis Guide.

ATLAS-Automatically Tuned Linear Algebra Software, http://math-atlas.sourceforge.net

Bathe, K.J. (1995) Finite Element Procedures, Prentice Hall.

Benzi, M., Kouhia, R., Tuma, M. (1998) An Assessment of Some Preconditioning Techniques in Shell Problems, *Comm. Numer. Methods Eng.*, 14, pp.897~906.

Benzi, M., Cullum, J.K., Tuma, M. (2000) Robust

Approximate Inverse Preconditioning for the Conjugate Gradient Method, *J. Sci. Comput.*, 22(4), pp.1318~1332.

Benzi, M. (2002) Preconditioning Techniques for Large Linear Systems: A Survey. *J. Comput. Phys.*, 182(2), pp.418~477.

BLAS (Basic Linear Algebra Subprograms), www.netlib.org/blas/

Cho, J.-R. (2009) Object-oriented Finite Element Framework Using Hybrid Programming, PhD. Dissertation, Seoul National University.

Cho, M., Parmerter, R.R. (1992) An Efficient Higher Order Plate Theory for Laminated Composites, *Compos. Struct.*, 20, pp.113~123.

CHOLMOD in SUITESPARSE, http://faculty.cse.tamu.edu/davis/suitesparse.html

Davis, T.A. (2006) Direct Methods for Sparse Linear Systems, Siam.

GOTOBLAS2 https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2

Intel Math Kernel Library, http://software.intel.com/en-us/intel-mkl.

LAPACK-Linear Algebra PACKage, http://www.netlib.org/lapack/

Lim, J.-S., Son I.-M., Kim J.-M., Seo C.-G. (2016) A Speed-Up in Computing Time for SSI Analysis by p-version Infinite Elements, *J. Comput. Struct. Eng. Inst. Korea*, 29(5), pp.471~482.

MUMPS, http://mumps.enseeiht.fr/

OpenBLAS, http://www.openblas.net/

PARDISO, http://www.pardiso-project.org

Saad, Y. (2003) Iterative Methods for Sparse Linear Systems. Siam.

UMPFACK in SUITESPARSE, http://faculty.cse.tamu.edu/davis/suitesparse.html

요 지

멀티코어 CPU와 BLAS, LAPACK을 구현한 최적 수치라이브러리, 직접 희소 솔버의 대중화 등 PC나 워크스테이션 수준에서도 대규모 유한요소 모델을 해석할 수 있도록 컴퓨팅 환경이 급속도로 변화되었다. 이 논문에서는 멀티코어 CPU를 갖는 공유 메모리 구조에 대한 병렬 유한요소 프로그램 설계시 고려사항으로 (1) 최적화된 수치라이브러리의 사용, (2) 최신 직접 희소 솔버의 사용, (3) OpenMP를 이용한 병렬 요소 강성 행렬의 계산, (4) 희소행렬 저장방식의 일종인 triplet을 이용한 어셈블 기법 등을 제시하였다. 또한 대규모 수치모델을 통해 많은 시간이 소요되는 작업을 기준으로 병렬화 효과를 검토하였다.

핵심용어 : 유한요소코드, 멀키코어 CPU, OpenMP, 희소 솔버