20th International Conference
ENGINEERING MECHANICS 2014
Svratka, Czech Republic, 12 – 15 May 2014

ON PARALLELIZATION OF STIFFNES MATRIX ASSEMBLY

M. Bošanský[*], B. Patzák[**]

Abstract: *The aim of this paper is to evaluate efficiency of different approaches to parallelization of sparse matrix assembly using OpenMP. The OpenMP platform is an Application Program Interface (API) for multi-platform shared-memory parallel programming in C/C++ and Fortran. The paper shows that parallelization can efficiently use modern available hardware, significantly reducing the needed computation time.*

Keywords: Assembly operation, Parallel efficiency.

1. Introduction

The development in computer hardware in last decades shows enormous progress, enabled by the introduction of parallel computers. Nowadays, the parallel technology is available even on desktop PC due to widespread use of multicore chips. The rapid development in hardware is, however, not accompanied by corresponding development in software technology. To fully utilize the potential of modern hardware, new algorithms and techniques have to be developed.

The aim of the paper is to study different approaches to parallelization of sparse matrix assemble operation using shared memory programming model using OpenMP library. The study has been done in OOFEM, which is a free finite element code with object oriented architecture for solving mechanical, transport and fluid mechanics problems (Patzak, 2002).

We focused our attention on sparse matrix assembly operation. During this phase, the individual characteristic matrices of individual elements are evaluated and assembled into a global problem matrix using element code numbers. This operation can be relatively demanding, especially, when nonlinear problems are taken into account. The parallelization strategy is based on decomposition of the total work into parts assigned to individual computing nodes. In the present context, the serial loop over all elements is split into disjoint parts assigned to individual computing nodes. The individual element matrices are computed in parallel on individual nodes and then assembled into global stiffness matrix, which is shared between individual nodes (shared memory). It is necessary to ensure, that multiple threads do not update the same value in the sparse matrix, as this will lead to unpredictable and incorrect results. The OpenMP provides different mechanisms that can be used to ensure consistency when accessing and updating shared memory. In this paper, we discuss the individual possibilities and compare their efficiency.

The paper is organized as follows. In Section 2 we briefly explain the assembly of the stiffness matrix and OpenMP API constructs which we use in Section 2. Results are shown and discussed in Section 3. Finally, we conclude the paper in Section 4.

2. Forming Stiffness Matrix with the Use of Parallel Directives

The typical (serial) implementation of sparse matrix assembly consists of loop over elements. Inside this loop, the individual element matrices are evaluated and assembled into a global matrix. The mapping between element local degrees of freedom and corresponding global degrees of freedom is described by code numbers. The individual values in element matrices are added to the corresponding global matrix entry, which row and column indices are determined using the element code numbers, see Tab. 1.

[*] Ing. Michal Bošanský, Department of Mechanics, Faculty of Civil Engineering, CTU in Prague, Thákurova 7/2077; 166 29, Prague; CZ, michal.bosansky@fsv.cvut.cz

[**] Prof. Bořek Patzák, Department of Mechanics, Faculty of Civil Engineering, CTU in Prague, Thákurova 7/2077; 166 29, Prague; CZ, Borek.Patzak@fsv.cvut.cz

Tab. 1: Prototype code for serial sparse matrix assembly.

```
For elem=1, nelem
    Ke = computeElementMatrix(elem)
    Loc = geveElementCodeNumbers(elem)
    For i=1, K.rows
      For j=1, K.columns
         K(loc(i), loc(j)) += Ke (i, j)
       End
    End
End
```

2.1. Application of OpenMP platform

The OpenMP parallel constructs allow relatively straightforward parallelization of serial code, by providing construct enabling to parallelize the for-loops. This is achieved by marking such loop by a special compiler pragmas as parallel loop. The rest is done almost automatically by a compiler, which splits the loop into parts that are then assigned to individual threads of execution. The user can adjust certain parameters that affect the work assignment (static or dynamic), and the scope (local or shared) of variables inside the loop. In order to achieve the best scalability, the amount of serial code must be reduced to profit from parallel execution. It is therefore natural to parallelize the top level loop over individual elements in assembly operation. The parallel OpenMP loop construct allows to mark variables inside the loop as shared or private variables. Shared variables are shared among the individual computing nodes, which means that the data are visible and accessible by all threads simultaneously. On the other hand, the private variables are private to each thread and therefore each thread will have its own copy and use it as a temporary variable. The elements matrices and code number vectors were marked as private.

The consistency in accessing and updating shared data can be achieved by a number of different ways in OpenMP. The following approaches have been considered:

- **A1.** The localization of element stiffness matrix is enclosed in critical section. The critical section ensures that the section can be processed by only a single thread in time.
- **A2.** Only the update operation of global stiffness matrix entry enclosed in critical section.
- **A3.** The update operation of global matrix entry marked as atomic. Atomic directive ensures that the memory update is atomic. A compiler might use special hardware instructions for better performance than when using *critical section*.
- **A4.** The consistency ensured using lock or semaphore that lock particular part of code. When the lock is activated the code block which is locked can be executed by only one thread at a time that owns the lock. Single lock has been used.
- **A5.** The consistency ensured using multiple locks. In this approach, the global matrix is divided in blocks, corresponding to a range of matrix rows (the number of blocks is a parameter) with dedicated lock. This approach does not locks the whole matrix for a single thread, it locks only specific block, so that the others threads can perform update in other rows.

Tab. 2: Prototype code for A1, A2 and A3 strategies.

```
#pragma opm parallel for
For elem=1, nelem
  Ke = computeElementMatrix(elem)
  Loc = geveElementCodeNumbers(elem)
  #pragma omp critical (A1)
  For i=1, K.rows
    For j=1, K.columns
      #pragma omp critical (A2)
      K(loc(i), loc(j)) += Ke (i, j)
    End
  End

End
```

A1/A2: Sparse assembly with critical section

```
#pragma opm parallel for
For elem=1, nelem
    Ke = computeElementMatrix(elem)
    Loc= geveElementCodeNumbers(elem)
    For i=1, K.rows
      For j=1, K.columns
        #pragma omp atomic
        K(loc(i), loc(j)) += Ke (i,j)
       End
    End
End
```

A3: Sparse assembly with atomic section

Tab. 3: Prototype code for A4 and A5 strategies.

```
omp_lock_t writelock;
OMP_INIT_LOCK(&writelock);

#pragma opm parallel for
For elem=1, nelem
 Ke = computeElementMatrix(elem)
 Loc = geveElementCodeNumbers(elem)
 For i=1, K.rows
  For j=1, K.columns
    OMP_SET_LOCK(writelock);
    K(loc(i), loc(j)) += Ke (i, j)
    OMP_UNSET_LOCK(writelock);
  End
 End

   End
OMP_DESTROY_LOCK(&writelock);
```

A4: Sparse assembly with single lock

```
#define NBLOCKS 10
omp_lock_t writelock[NBLOCKS]
For i=1, NBLOCKS
  OMP_INIT_LOCK(&writelock[i])
End
// block size; note integer division
blockSize = K.rows/NBLOCKS
#pragma opm parallel for
For elem=1, nelem
 Ke = computeElementMatrix(elem)
 Loc = geveElementCodeNumbers(elem)
 For i=1, K.rows
   For j=1, K.columns
     Bi = loc(i)/blockSize
     OMP_SET_LOCK(writelock[bi])
     K(loc(i), loc(j)) += Ke (i, j)
     OMP_UNSET_LOCK(writelock[bi]);
   End
  End
End
```

A5: Sparse assembly with block locks

3. Testing Example, Results and Discussion

The individual approaches have been evaluated using a 3D finite element model of nuclear containment reactor. The mesh consists of 87320 nodes and 959700 tetrahedral elements with linear interpolation. The total number of equations was 260322. The stiffness matrix has been stored in compressed row sparse format. In this storage format, on floating-point array is needed to store actual nonzero values as they are traversed in a row-wise fashion. The other two integers arrays store the column indexes of the elements in the vector of values and the second stores the locations of those values that start a row. Instead of storing $\mathbf{n}^2$ elements (where n is the number of equations), we need only 2**nnz + n + 1** storage locations, where **nnz** is number of nonzero entries.

The four described approaches (A1-A5) have been implemented and their efficiency compared on test example. The evaluations have been done on a desktop PC with 8 cores. The runtime, in terms of wall clock time, has been recorded for each strategy for varying number of processors. In the strategy A5, with multiple locks, each one for dedicated row blocks, the total number of blocks has been also changed.

In Table 4, the execution times for strategies A1-A4 are given. It can be clearly observed, that time needed for the execution is reduced with increasing number of threads. However, the gain is non-proportional, due to the overhead associated with resource sharing (data bus), data consistency (locking, critical sections), and thread management (creation, termination). The more threads are used, the less is the gained speed up. The best speedups and also absolute times are observed for A1 strategy, where the whole local matrix assembly block was contained within a critical section. Comparing to the A2, where only the update of global entry was in the critical section, it is clear, that the overhead associated with critical section is higher than possible gain, when the loops over local entries can be done in parallel. The strategy A5 with multiple locks for dedicated row blocks is efficient than the strategies A3 and A4 relying on simple lock strategy. The results show that there are significant differences between individual strategies. In general, the optimal strategy is problem dependent and must be evaluated on several examples. However, with each strategy one can achieve better performance compared to serial code.

4. Conclusions

The presented work evaluates the different strategies for parallel assembly of sparse matrix using OpenMP programming model. The individual strategies are based on critical sections, atomic constructs, and locks used to lock the whole matrix or its section. The strategies are compared on forming sparse stiffness matrix of a large scale engineering problem. The results show that using parallelization the computation time can be significantly reduced. The overhead costs of individual approaches can have a significant impact on overall computation time.

Tab. 4: Execution times for strategies A1 – A5.

execution times [s]	number of threads				
	1	2	4	6	8
A1	4.531	2.750	1.789	1.640	1.608
A2	5.019	3.124	5.553	7.487	8.488
A3	5.109	3.131	5.643	7.501	8.528
A4	5.002	3.120	5.553	7.417	8.345
A5 – NBLOCKS 50	5,134	3.175	2.539	2.482	2.451
A5 – NBLOCKS 100	5.152	3.159	2.429	2.383	2.359
A5 – NBLOCKS 500	5.155	3.151	2.306	2.175	2.131
A5 – NBLOCKS 1000	5.150	3.1722	2.299	2.172	2.126

Acknowledgement

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant **No. SGS OHK1-089/14** “Advanced algorithms for numerical modeling in mechanics of structures and materials”.

References

Patzak, B. (2002) OOFEM project home page, In: http://www.oofem.org [online], 2002-05-01 [cited 2014-26-1]. Available from: http://www.oofem.org/en/documentation/manual.html.

Chapman, B., Jost, G., Pas, R. V. D. foreword Kuck, D. J. (2008) Using OpenMP – portable shared memory parallel programming The MIT press, Cambridge Massachusetts, London, England: ISBN-13: 978-0-262-53302-7.

Stroustrup, B. (1997) The C++ programming language Third edition. Murray Hill, New Jersey: 1997 by AT T. ISBN 0-201 885954-4.