22nd International Conference
ENGINEERING MECHANICS 2016
Svratka, Czech Republic, 9 – 12 May 2016

ON PARALLELIZATION OF ASSEMBLY OPERATIONS IN FINITE ELEMENT SOFTWARE

M. Bošanský*, B. Patzák**

Abstract: *Current development in computer hardware brings in new opportunities in numerical modelling. Computers with a single processing unit, where only one instruction can be processed at any moment in time, allow us to run simulation codes only sequentially. The performance of single processing units is reaching the physical limits, given by transmission delays and heat build-up on the silicon chips. The future of scientific computing seems to be in parallel computing, that allows to overcome the limitations of traditional sequential processing units. Parallel computing is based on simultaneous use of multiple processing units. The fundamental paradigm in parallel computing is based on work decomposition into pieces of work that can be processed simultaneously. This contribution focuses on parallelization of sparse matrix and global vector assembly operations, which are typical to any finite element code. The aim of presented work is to propose an alternative approach to assembly operation based on decomposition of the work into independent element groups, members of which can be processed concurrently without blocking operation. The individual groups contain elements contributing to distinct entries in sparse matrix or global vector. Such decomposition is done using colouring algorithm. As the elements in group contribute to distinct locations, there is no need to prevent the race condition, that can occur when the same location is updated simultaneously. It is only necessary to enforce synchronization before processing each element group. Efficiency of implemented approach is compared to approach based on decomposition of assembly loop using OpenMP and POSIX threads directives and explicit locking of updated locations in sparse matrix or global vector, which was published by the authors in (Bosansky & Patzak, 2016a) and (Bosansky & Patzak, 2016b).*

Keywords: Finite element method, shared memory, colouring algorithm.

1. Introduction

Parallelization can significantly reduce computational time by more efficient use of available hardware resources (Patzak, 2010). The existing parallel computers can be divided by type of memory architecture. The main types include shared, distributed, and hybrid memory systems (Hughes C. & Hughes T., 2003). Shared memory systems are characterized by a common, global memory space, which is accessible to all individual processing units. Various mechanisms are used to control access to the shared memory to prevent race conditions and deadlocks. The distributed systems are characterized only by local memory attached to individual processing units. The tasks exchange data through mutual, explicit communication. Finally, hybrid systems combine the features of the previous two. Efficient and scalable parallel algorithms are more difficult to design than serial ones, because parallelism introduces new sources of complexity. The design of any parallel algorithm requires to split the work into the set of smaller tasks, which are solved concurrently by the simultaneous use of multiple computing resources (Barney, 2010). Additional care must be taken to establish communication and synchronization between the tasks. The scalability of a parallel algorithm, represents the ability to reduce required solution time with increasing number of processing units. However, the ideal linear trend is difficult to obtain due to additional overhead of parallel algorithm (synchronization, locking) and due to the fact that some parts may have to be processed serially.

* 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

In this paper we use Open Multi-Processing (OpenMP) and Portable Operating System Interface (POSIX) Threads programming models to implement and compare different variants of parallel assembly operations. OpenMP is a shared memory programming model that supports multi-platform shared memory multiprocessing programming in C, C++ (Stroustrup, 1997), and Fortran, on most processor architectures and operating systems (Barney, 2014). It consists of a set of compiler directives, library routines, and environment variables that influence run-time behaviour. OpenMP parallel constructs allow us to overcome relatively straightforward parallelization of serial code, by providing constructs enabling to parallelize the for-loops (Chapman et al., 2008). This is achieved by marking such a loop by special compiler directives as a parallel loop. The rest is done almost automatically by a compiler, which splits the loop into parts that are subsequently 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. Shared variables are shared among the individual threads, which means the data are visible and accessible to all threads simultaneously. On the other hand, the local variables are private to each thread and each thread has its own copy. The POSIX Threads (Pthreads) library is standardized thread API for C/C++ (Nicols et al., 1996). Pthreads allows one to spawn a new concurrent process flow. New thread as opposed to a new process can be created with much less operating system overhead. The cost of creating and managing processes is expensive. Major blocks in the Pthreads are Thread Management, Mutex Variables, Condition Variables and Synchorization procedures (Barney, 2015). Thread management allows to allocate, terminate, and pause individual threads. Mutex variables and functions provide services for creating, destroying, locking and unlocking mutexes, which allow to control access to memory by individual threads. A mutex variable acts like a lock protecting access to a shared data. The basic concept of a mutex as used in Pthreads is that only one thread can lock a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. Other threads can‘t own that mutex until the owning thread unlocks that mutex, and therefore have to wait until mutex is released by owning thread. Condition variables based on routines that address communications between threads that share a mutex. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set condition variable attributes are also included. Pthreads is relatively low-level API for working with threads, providing fine-grained control over thread management (create, join, attributes), mutexes, etc. In Pthreads, programmer must define every parallel procedure explicitly.

Finite Element Method (FEM) is the most popular method to solve various problems in engineering. It actually identifies a broad spectrum of methods for finding an approximate solution to boundary value problems for partial differential equations. The differential equations are converted to the algebraic system of equations by using variational methods and by decomposing the problem domain into sub-domains called elements. In structural mechanics, the resulting algebraic equations correspond to the discrete equilibrium equations at nodes of the computational mesh. In matrix notation we can write

$$\mathbf{K}\,\mathbf{r} = \mathbf{f} \qquad (1)$$

where **K** and **f** are global stiffness matrix and load vector, and **r** is vector of unknown displacements.

The vector as well as the stiffness matrix are assembled from individual elements contributions

$$F_i = \sum F_i^e \; ; \; f_i = \sum f_i^e \; ; \; K = \sum K^e \; ; \qquad (2)$$

The assembly process is relying on global numbering of discrete equilibrium equations yielding so called code-numbers assigned to nodes (identifying equilibrium equations assembled in given node) and to elements (identifying equilibrium equation into which the element end-forces contribute). The typical serial algorithm for vector and matrix assembly is following

Algorithm 1: Prototype code - vector of nodal forces assembly

```
001  for elem=1, nelem
002      F^e = compute Element Vector (elem)
003      Loc^e = give Element Code Nubmers (elem)
004      for i = 1, Size (Loc^e)
005          F(Loc^e(i))+ = F^e(i)
```

Algorithm 1: Prototype code for serial global stiffness matrix assemby

```
for elem = 1, nelem
    Ke = computeElementMatrix(elem)
    Loc = geveElementCodeNubmers(elem)
    for i = 1, K.rows
        for j = 1, K.comlums
            K(loc(i), loc(j)) += Ke(i, j)
```

This assembly operation evaluates individual element contributions. This operation in case of nonlinear problem can be repeated several times and can be costly, due to the computational effort needed to evaluate individual element contributions.

The parallelization strategy of the assembly operation is typically based on domain decomposition paradigm, where the elements are divided into disjoint subsets (called sub domains), that are processed by individual threads. The individual element contributions are computed in parallel and then assembled into the corresponding global form a equilibrium equations. However, as elements processed by individual threads may contribute to the same global equilibrium equation, it is necessary to ensure that multiple threads do not update the same value in global force vector or stiffness matrix at the same time, as this will lead to unpredictable and incorrect results.

2. The flow of control of the Assembly Loop and Parallelization of the Assembly Loop

In order to achieve the best scalability, the amount of serial code and mutual synchronization must be reduced. It is therefore natural to decompose the work on the coarsest possible level, in our case on the top level loop over individual elements in assembly operation.

The first approach, based on OpenMP explicit synchronization using critical sections enclosing the section of the code with update of global matrix and vector values. In this case the global matrix and vector are shared by default between threads (all variables declared outside parallel loop are shared by default) and the critical directive is used to restrict the execution of the associated code block just to a single thread at a time. The second approach, based on OpenMP, utilizes explicit simple locks to ensure that only a single thread can update accumulated value of matrix and vector. The localization of element matrix and vector is enclosed using simple locks that lock particular part of code. The third approach utilizes explicit nested locks and locking strategy is similar to simple lock.

The traditional approach, as discussed in previous paragraph, requires explicit locking of updated memory cells to prevent simultaneous update. Such locking and its associated overhead have negative impact on overall parallel efficiency.

As the proposed strategies did not performed as expected, an alternative algorithm has been proposed, based on alternative decomposition of elements into disjoint groups, where contributions of individual elements in a group do not update the global entry updated by another element within the same group. The elements in the same group should not share any common node to satisfy this requirement. Such decomposition can be obtained by a so-called colouring algorithm, described in (West, 2001), for example.

The direct consequence of this decomposition is the possibility to process all elements within a group in parallel without the need of any locking or synchronization. The serial assembly procedure has to be decomposed into outer loop over element groups and inner loop over elements of individual group, that can be processed concurrently. The optimum performance can be expected when the number of groups will be minimal and when the inner loop will be optimally decomposed into available processing nodes, either using static or dynamic load balancing.

The approach with colouring algorithm was evaluated using a benchmark problem of a 3D finite element model of nuclear containment In the first case (further denoted as C1) the mesh consists of 87320 nodes and 959700 tetrahedral elements with linear interpolation. The total number of equations was 260322 and number of colours was 16. In the second case, denoted as C2, the mesh consists of 899780 nodes and 114878457 tetrahedral elements with linear interpolation. The total number of equations was 3013128 and number of colours was 15. The structure has been loaded with self-weight, resulting in nonzero contribution of every element to external load vector. The individual approaches have been tested on Linux workstation (running Ubuntu 14.04 OS) with the two CPU Intel(R) Xeon(R)

CPU E5-2630 v3 @ 2.40GHz and 126GB RAM. The two CPU units consists of eight physical and sixteen logical cores, allowing up to thirty-two threads to run simultaneously. All the tests fit into a system memory.

Executions times [s]	number of threads															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Sequential assembly	**5,22**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OpenMP – Critical	-	3,80	3,25	2,55	2,28	1,88	1,81	1,62	1,62	1,52	1,53	1,47	1,43	1,42	1,42	1,41
OpenMP – Simple lock	-	3,66	3,17	2,52	2,27	1,86	1,80	1,61	1,58	1,49	1,45	1,47	1,41	1,37	**1,34**	1,37
OpenMP – Nested lock	-	3,78	3,14	2,55	2,25	1,89	1,83	1,62	1,54	1,56	1,46	1,46	1,44	1,36	**1,32**	1,38
OpenMP – CP	-	3,75	3,42	2,99	2,67	2,45	2,39	2,25	2,23	2,25	1,95	2,16	2,07	2,23	**1,86**	1,94
POSIX – CP	-	3,80	3,25	2,55	2,28	1,88	1,81	1,62	1,62	1,52	1,53	1,47	1,43	1,42	1,42	1,41
Executions times [s]	**number of threads**															
	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Sequential assembly	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OpenMP – Critical	1,44	**1,39**	1,44	1,46	1,45	1,47	1,47	1,47	1,49	1,52	1,52	1,52	1,53	1,55	1,55	1,57
OpenMP – Simple lock	1,38	1,38	1,43	1,40	1,40	1,45	1,46	1,44	1,47	1,47	1,48	1,52	1,45	1,52	1,51	1,51
OpenMP – Nested lock	1,36	1,49	1,37	1,39	1,40	1,47	1,43	1,45	1,46	1,46	1,49	1,49	1,50	1,48	1,52	1,50
OpenMP – CP	2,12	1,90	2,08	2,06	2,15	2,26	2,25	2,23	2,22	2,23	2,25	2,24	2,21	2,19	2,23	2,25
POSIX – CP	1,44	**1,39**	1,44	1,46	1,45	1,47	1,47	1,47	1,49	1,52	1,52	1,52	1,53	1,55	1,55	1,57

Tab. 1: Execution times for case C1 (smaller benchmark problem).

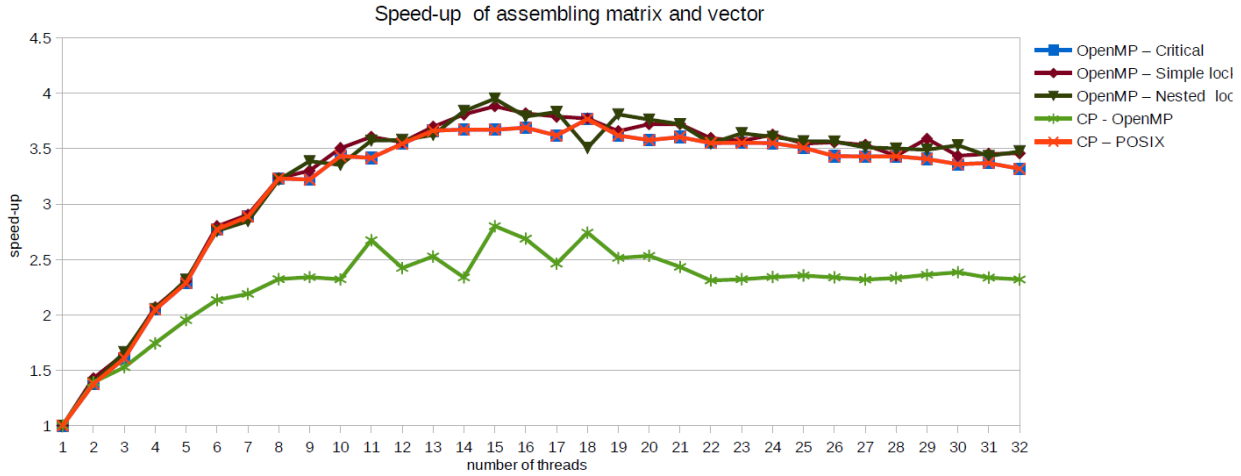

Fig. 1: Speed-ups for case C1 (smaller benchmark problem).

Executions times and speed-ups of assembly operations based on previously mentioned strategies using OpenMP parallel constructs (synchronization using Critical sections or Simple locks) using different numbers of threads were updated from (Bosansky & Patzak, 2016a) and (Bosansky & Patzak, 2016b). Executions times and speed-ups for assembly operations (matrix and vector assembly) obtained using decomposition based on colouring algorithm are compared with previously mentioned strategies in Tab. 1, Fig. 1 and Tab. 2, Fig. 2. The reported execution times have been obtained as an average from five consecutive runs. It can be observed that time needed for calculation is reduced as number from threads gets bigger. Nevertheless, the impact of overhead is dominant with number of threads bigger than twenty and the overall speed-up is clearly not ideal, see Fig. 1 and Fig. 2. This is partially due to the additional overhead associated with parallel implementation (thread creation and management, synchronization) that is not present in the serial version. Compared to traditional loop based decomposition, the colouring decomposition performs better, reducing the parallel overhead. However, the overall trend is similar. Additional performance decrease (see Fig. 1 and Fig. 2) can be attributed to the hyper-threading technology specific to Intel processors, leading to sharing of CPU resources (execution engine, caches, and bus interface) between the hyper-threaded cores, but this is the factor common for both approaches.

Executions times [s]	number of threads															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Sequential assembly	**62,28**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OpenMP – Critical	-	46,10	35,20	27,07	21,64	19,08	17,54	16,66	17,74	17,25	17,49	17,43	18,18	17,83	**16,16**	16,27
OpenMP – Simple lock	-	47,2	34,9	30,4	25	23,9	20,5	18,4	18,3	17,6	18,2	17,2	17,3	16,7	16,8	**16,1**
OpenMP – Nested lock	-	46,1	33,7	28,2	24	22,1	20,5	19,8	18	17,4	17,1	17	16,9	17,1	16,9	**16**
OpenMP – CP	-	52,5	39,6	32,6	28,6	25,8	29,2	23,6	22,5	**18,4**	18,3	18,8	18,6	24,5	25,7	27
POSIX – CP	-	42,4	35,4	25,9	25,4	23,1	23,4	22,4	22,9	22,6	22,4	23,5	23,6	23	24,6	23,5
Executions times [s]	**number of threads**															
	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Sequential assembly	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OpenMP – Critical	17,89	17,49	17,39	19,54	17,48	18,62	18,28	19,91	18,18	21,84	20,20	18,28	18,62	19,49	17,77	19,36
OpenMP – Simple lock	16,8	17,6	17	16,9	18,8	19,8	17,4	18,7	19,8	18,3	17,7	17,4	18,4	17,9	19	23,9
OpenMP – Nested lock	17,3	16,9	17,3	17,2	17,6	19,8	17,7	18,6	17,4	19,9	18,6	18,3	20,4	18,6	18,9	17,8
OpenMP – CP	27,6	29,1	30,3	28,7	29,5	31,4	31,8	32,8	35,3	38,7	39,9	40,4	43,4	50,6	59,4	73,3
POSIX – CP	24	24,4	24,3	24,3	24,2	24,5	24,1	23,8	24,1	23,2	23,1	23,8	23,4	**22,3**	23,4	22,9

Tab. 2: Execution times for case C2 (bigger benchmark problem).

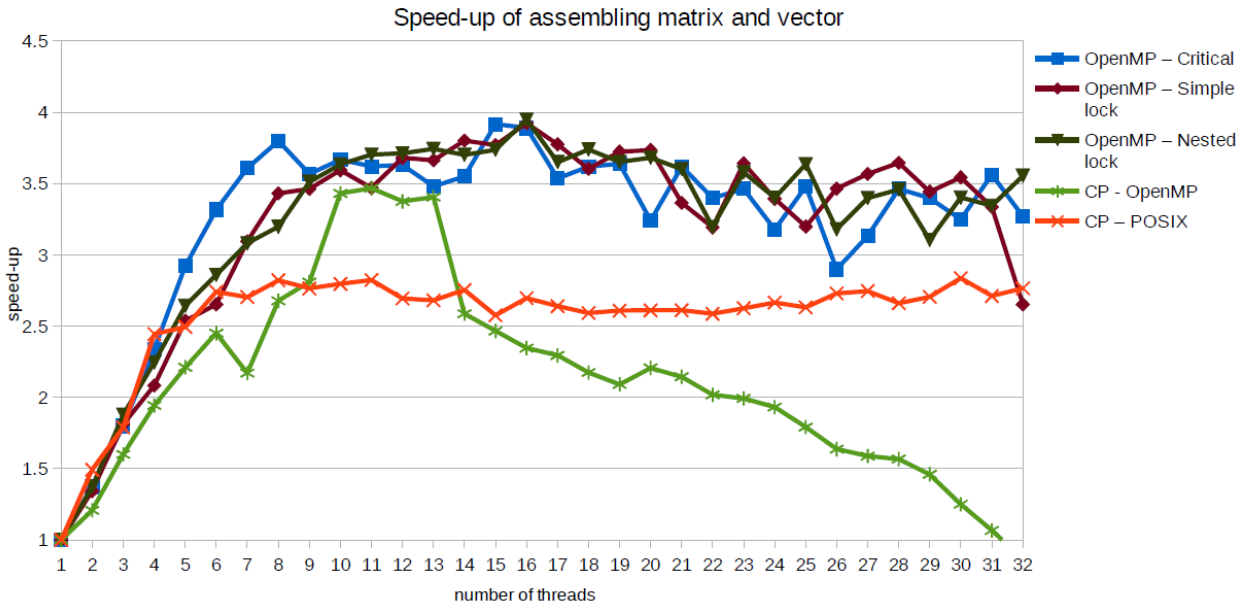

Fig. 2: Speed-ups for case C2 (bigger benchmark problem).

The results indicate that the colouring strategy performs better for lower number of threads involved. In our opinion this is the consequence of avoiding the need for explicit synchronization. However, for higher number of threads, the OpenMP with synchronization performs better. It seems that part of the problem would be in less optimal use of shared L3 cache on the processor (Kathy, 2012) see Fig. 3. The nodes are numbered in a way that allows to minimize sparse profile and reduce data transfer to/from cache memory. However, the parallelization approach using the colouring scheme accesses the individual elements in non-sequential order, which results in less optimal use of cached data with higher ratio of cache misses. However, the reasons for this behaviour are not yet fully understood and further evaluation is required.

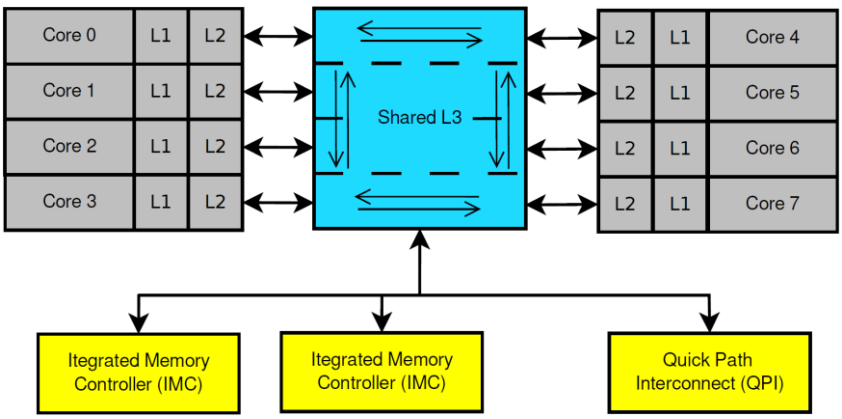

Fig. 3: Schema of CPU Intel Xeon.

3. Conclusions

In this work we compare two approaches for parallelization of assembly operation. The first is based on traditional loop-based decomposition using OpenMP constructs, while the other uses alternative decomposition into element groups based on connectivity considerations, allowing to process elements in individual group in parallel without any extend synchronization or locking. The latter approach performs slightly better, but further optimization can improve it performance.

Acknowledgement

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS15/031/OHK1/1T/11 "Advanced algorithms for numerical modelling in mechanics of structures and materials".

References

Barney B. (2010) Task scheduling for parallel systems. https://computing.llnl.gov/tutorials/parallel comp/.

Barney B., Lawrence Livermore National Laboratory (2014) OpenMP - An Application Program Interface. https://computing.llnl.gov/tutorials/openMP/.

Barney B., Lawrence Livermore National Laboratory (2015) POSIX Threads Programming,

https://computing.llnl.gov/tutorials/pthreads/.

Bosansky M., Patzak B. (2016a) Different Approaches to Parallelization of Vector Assembly. In Applied Mechanics and Materials, Trans Tech Publications, Vol. 821, pp. 341-348.

Bosansky M., Patzak B. (2016b) Different Approaches to Parallelization of Sparse Matrix Operation. Applied Mechanics and Materials, Trans Tech Publications, Vol. 825, pp. 91-98.

Hughes C., Hughes T. (2003) Parallel and Distributed Programming Using C++. ISBN 0-13-101376-9, Pearson Education.

Chapman B., G. Jost, R.V.D. Pas (2008) Using OpenMP - portable shared memory parallel programming. Morgan Kaufmann Publishers, Inc.

Kathy F., (2012) Intel® Xeon® Processor E5-2600/4600 Product Family Technical Overview, https://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview.

Nicols B., Buttlar D., and Farrell J.P. (1996) Pthreads Programming, ISBN 1-56592-115-1, O`Reilly and Associates,

Patzak B. (2010) Parallel computations in structural mechanics. ISBN 978-80-01-04565-7, Czech Technical University in Prague.

Stroustrup B. (1997) The C++ programming language. Murray Hill, AT Labs Murray Hill, New Jersey.

West, D. B. (2001) Introduction to graph theory. Vol. 2, Upper Saddle River: Prentice hall.