CORE
Metadata, citation and similar papers at core.ac.uk

Acta Polytechnica **60**(1):25–37, 2020
DOI:10.14311/AP.2020.60.0025
© Czech Technical University in Prague, 2020
available online at https://ojs.cvut.cz/ojs/index.php/ap

PARALLELIZATION OF ASSEMBLY OPERATION IN FINITE ELEMENT METHOD

Michal Bošanský*, Bořek Patzák

Czech Technical University in Prague, Faculty of Civil Engineering, Thákurova 7, 166 29 Prague 6, Czech Republic

* corresponding author: michal.bosansky@fsv.cvut.cz

Abstract.

The efficient codes can take an advantage of multiple threads and/or processing nodes to partition a work that can be processed concurrently. This can reduce the overall run-time or make the solution of a large problem feasible.

This paper deals with evaluation of different parallelization strategies of assembly operations for global vectors and matrices, which are one of the critical operations in any finite element software. Different assembly strategies for systems with a shared memory model are proposed and evaluated, using *Open Multi-Processing* (OpenMP), *Portable Operating System Interface* (POSIX), and *C++11 Threads*. The considered strategies are based on simple synchronization directives, various block locking algorithms and, finally, on smart locking free processing based on a colouring algorithm. The different strategies were implemented in a free finite element code with object-oriented architecture OOFEM [1].

Keywords: Parallel computation, shared memory, finite element method, vector assembly, matrix assembly.

1. Introduction

Current development in computer hardware brings in new opportunities in numerical modelling. The solutions of many engineering problems are extremely demanding, both in terms of time and computational resources. The traditional serial computers permit to run simulation codes only sequentially on a single processing unit, where only one instruction can be processed at any moment in time.

The current trend in technology is parallel processing, relying on the simultaneous use of multiple processing units to solve a given problem. Any parallel algorithm is based on the idea of partitioning the overall work into a set of smaller tasks, which can be solved concurrently by a simultaneous use of multiple computing resources. Parallel processing allows to concurrently perform tasks on a single computer with multiple processing units or even multiple computers with multiple processing units. Parallelization can significantly reduce the computational time by a more efficient use of the available hardware. It can also allow solving large problems that are often not fit for a single machine. The parallel programming requires a development of new techniques and relies on programming language extensions and a use of communication libraries. The principal issue in parallel assembly is to prevent the race conditions, where the same memory location is to be updated by multiple threads.

Parallel computers can be classified, for example, by the type of memory architecture [2]. The shared, distributed, and hybrid memory systems exist. In the shared memory system, the main memory with a global address space is shared between all processing units, which can directly address and access the same global memory. The global memory significantly facilitates the design of the parallel program, as the individual tasks can communicate via the global memory, but the memory bus performance is the limiting factor for the scalability of these systems.

The systems with the distributed memory are built by connecting individual processing units equipped with a local memory. There is no global address space and the individual tasks running on different processing units have to communicate using messages. The design and implementation of parallel algorithms is typically more demanding, however, these systems do not suffer from scalability limitations of shared memory systems. Finally, the hybrid systems try to combine the advantages of shared and distributed systems by combining muti-core processing units with a local memory into powerful systems. The most powerful computers today are build using this paradigm [3].

Finite Element Method (FEM) is one of the most popular methods to solve various problems in engineering. It actually consists of a broad spectrum of methods for finding an approximate solution to boundary value problems described by the system of partial differential equations. The differential equations are converted to the algebraic system of equations using variational methods. Once the integral form is set up, the discretization of the problem domain into non overlapping subdomains called elements is introduced to define an approximation and test functions. In structural mechanics, the resulting algebraic equations correspond to the discrete equilibrium equations at element nodes. In matrix notation, the resulting

equilibrium equations for a linear static problem have the following form

$$K * \mathbf{r} = \mathbf{F}, \quad (1)$$

where **K** and **F** are global stiffness matrix and load vector, respectively, and **r** is the vector of unknown nodal displacements. The global stiffness matrix **K** and global load vector **F** are assembled from individual element and nodal contributions. This process is relying on the global numbering of equations, where the contributions of individual elements are assembled (added) to global matrix/vector values according to global equation numbers of nodal unknowns. The typical sequential vector assembly algorithm is outlined in Algorithm 1.

Algorithm 1: Prototype code - Assembly of load vector

```
001  for elem = 1, nelem
002      F^e = computeElementVector(elem)
003      Loc^e = giveElementCodeNumbers(elem)
004      for i = 1, Size(Loc^e)
005          F(Loc^e(i))+ = F^e(i)
```

The typical sequential algorithm for matrix assembly is similar to the algorithm of vector assembly, see Algorithm 2 for a reference.

Algorithm 2: Prototype code - Assembly of stiffness matrix

```
001  for elem = 1, nelem
002      K^e = computeElementMatrix(elem)
003      Loc^e = giveElementCodeNumbers(elem)
004      for i = 1, Size(Loc^e)
005          for j = 1, Size(Loc^e)
006              K(Loc^e(i), Loc^e(j))+ = K^e(i, j)
```

As already mentioned, the assembly operations are one of the typical steps in the finite element analysis. In order to obtain a scalable algorithm, all the steps have to be parallelized, including the assembly phase, which could be costly. Particularly when solving nonlinear problems, the evaluation of individual element contributions (tangent stiffness matrix, internal force vector) can be computationally demanding and have to be performed for every load increment step and every iteration (the frequency of update of stiffness matrix depends on the actual solution algorithm).

In general, the parallelization of the assembly operation consists in splitting the assembly loop over elements into disjoint subsets, which are processed by individual processing threads. Within each thread, the individual element contribution is evaluated for each element in subset (this part can be evaluated concurrently), followed by the assembly of the local contribution to the target global matrix/vector, see simple demonstration in Figure 1. The key problem is that this step could not be performed concurrently, as the multiple threads may update the same global entry at the same time. Therefore, it is necessary to ensure, that the same global entry is not being updated at the same time by multiple threads. This is known as so called race condition. To prevent the race condition on update, various techniques can be used. They typically consist in using locking primitives to make sure that (i) the code performing the update can be executed only by a single thread, (ii) the specific memory location can be updated only by a single thread, or (iii) the evaluation of element contributions is ordered in a such a way that the conflict cannot occur.

One of the important characteristics of the parallel algorithm is its scalability. Unfortunately, the ideal, linear scalability is difficult to obtain. Almost every parallel algorithm has an overhead cost, when compared to the sequentional version. The individual tasks cannot be executed concurrently without a synchronization and communication with other tasks. Finally, some parts of the algorithm are essentially serial and have to be executed only by a single thread.

The different parallel assembly approaches have been presented, for example, in paper [4]. In [4], an approach based on OpenMP *critical sections* and OpenMP atomic directives, have been proposed.

2. Shared memory frameworks

In this section, different strategies to parallel vector/matrix assembly are presented. They use different techniques to prevent the race condition on data update. These strategies are implemented using different shared memory programming models available, including *Open Multi-Processing* (OpenMP), *Portable Operating System Interface (POSIX) Threads, and* C++ 11 Threads programming interface.

2.1. OpenMP

OpenMP is a shared memory programming model that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran programming languages. It is available for a wide variety of processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence the run-time behaviour, see [5] for details.

The programming in OpenMP consists in using so called parallel constructs (compiler directives), which are inserted in the source code, instructing the compiler to generate a specific code. The OpenMP defines various constructs allowing to parallelize serial code and synchronize the individual threads [6].

To reduce the granularity of the problem and reduce the overhead connected to thread creation and termination, it is usual to parallelize the outermost loops in the algorithm, which, in our particular case, corresponds to the parallelization of the loop over elements in the assembly operation.

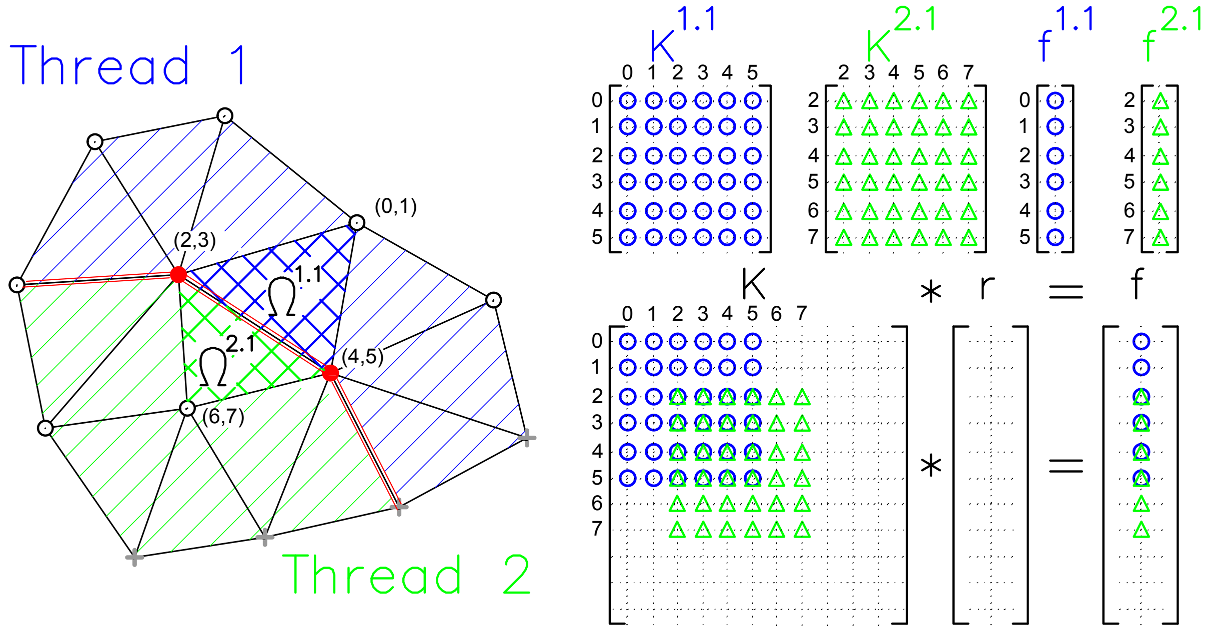

FIGURE 1. Parallel algorithm for Stiffness Matrix assembly schema

2.1.1. SYNCHRONIZATION USING CRITICAL SECTION, ATOMIC UPDATE, SIMPLE LOCK, NESTED LOCK

In this subsection, we start with a simple solution preventing an occurrence of race conditions during data update based on the *critical section*, *atomic update*, *simple lock*, and *nested lock* constructs.

The synchronization using *critical section* is implemented in three variants. In the first variant (marked as A1), the whole assembly operation consisting of loop over element code numbers (rows and columns numbers of global matrix / vector) is enclosed using *critical section*. The second variant considers only enclosing the actual update operation (A2). The last variant (A3) is similar to (A2), but uses the atomic update, see Algorithm 3.

A similar approach is followed to synchronize threads using locks. The lock is used to ensure that either a single thread can process the loop over element code numbers (A1.1 or A1.2 using the nested lock) or an actual update operation (A2.1) is carried out, see Algorithm 4.

Algorithm 3: Prototype code - matrix assembly with explicit synchronization

```
001 # pragma omp parallel for private(K^e, Loc^e)
002 for elem = 1, nelem
003     K^e = computeElementMatrix(elem)
004     Loc^e = giveElementCodeNumber(elem)
005     # pragma omp critical (A1)
006         for i = 1, Size(Loc^e)
007             for j = 1, Size(Loc^e)
008                 # pragma omp critical (A2)
009                 # pragma omp atomic (A3)
010                     K(Loc^e(i), Loc^e(j))+ = K^e(i, j)
```

Algorithm 4: Prototype code - matrix assembly with explicit locks

```
001 omp_init_lock(&my_lock) (A1.1) (A1.2)
002 omp_init_nest_lock(&my_nest_lock) (A2.1)
003 # pragma omp parallel for private(K^e, Loc^e)
004 for elem = 1, nelem
005     K^e = computeElementMatrix(elem)
006     Loc^e = giveElementCodeNumber(elem)
007     omp_set_lock(&my_lock) (A1.1)
008     omp_set_nest_lock(&my_nest_lock) (A2.1)
009         for i = 1, Size(Loc^e)
010             for j = 1, Size(Loc^e)
011                 omp_set_lock(&my_lock) (A1.2)
012                     K(Loc^e(i), Loc^e(j))+ = K^e(i, j)
013                 omp_unset_lock(&my_lock) (A1.2)
014     omp_unset_lock(&my_nest_lock) (A1.1)
015     omp_unset_nest_lock(&my_nest_lock)
    (A2.1)
016 omp_destroy_lock(&my_lock) (A1.1) (A1.2)
017 omp_destroy_nest_lock(&my_nest_lock) (A2.1)
```

The approach followed in this section is rather conservative, ensuring that only a single thread can per-

form any update operation. In reality, the race condition on data update can happen only if two or more threads are attempting to update the same entry in global vector/matrix. This essentially means that these threads are assembling contributions of elements sharing the same node(s). The probability of this can be relatively low, so, in next sections, we try to propose improved algorithms that allow to perform the update operation in parallel, provided that different entries of global vector/matrix are updated.

2.1.2. Synchronization using block locks

The algorithm presented in this section is based on the idea of having an array of locks, each corresponding to consecutive blocks of values in global vector/matrix. Once a specific global value is to be updated by a specific thread, the corresponding lock is acquired, preventing other threads to update values in the same block, but allowing other threads to update values in other blocks. It may seem that the ideal situation is to have unique lock for every global value, but as the global vector/matrices are the dominant data structures (in terms of memory requirements) in a typical FE code, this approach is not feasible. In the presented approach, the individual groups correspond to blocks of rows of global vector/matrix values and the prototype implementation is presented in Algorithm 5.

Algorithm 5: Prototype code - matrix assembly with explicit block locks

```
001  # define NBLOCKS
002  omp_init_t_&my_lock [NBLOCKS]
003  for n = 1, NBLOCKS
004        omp_init_lock(&my_lock [n])
005  blocksize = Size(K.rows)/NBLOCKS
006  # pragma omp parallel for private(K^e, Loc^e)
007  for elem = 1, nelem
008       K^e = computeElementMatrix(elem)
009       Loc^e = giveElementCodeNumber(elem)
010            for i = 1, Size(Loc^e)
011              bI = Loc^e(i)/blocsize //integer division
012                  for j = 1, Size(Loc^e)
013                    omp_set_lock(&my_lock [bI])
014                         K(Loc^e(i), Loc^e(j))+ = K^e(i, j)
015                    omp_unset_lock(&my_lock [bI])
016  for n = 1, NBLOCKS
017    omp_destroy_lock(&my_lock [n])
```

2.2. POSIX Threads

The POSIX Threads (*Pthreads*) libraries are standardized thread programming interfaces for C/C++ language. *Pthreads* allows one to spawn a new concurrent processes. *Pthreads* consists of a set of C/C++ language types and procedure calls, see [7] for further details. The POSIX Threads algorithms are based on distributing the element set into continuous subsets assigned to individual threads.

2.2.1. Synchronization using Simple Mutex and Recursive Mutex

The POSIX Threads synchronization routines allow to protect shared data when multiple threads update the data. The concept is very similar to locks in OpenMP library. In POSIX Threads, only a single thread can lock mutex variable at any given time. In the case where several threads try to lock mutex, only one thread will be successful. No other thread can own mutex until the owning thread unlocks that mutex. The simple mutex can be used only once by a single thread. Attempting to relock the mutex (trying to lock mutex after a previous lock) causes a deadlock. The attempt to unlock a simple mutex that it has not locked leads to an undefined behaviour. The recursive mutex shall maintain the concept of a lock count. When a computing thread successfully acquires recursive mutex for the first time, the lock count shall be set to one. Each time the thread unlocks the recursive mutex, the lock count shall be decremented by one. When the lock count reaches to zero, the recursive mutex shall become available for other threads to acquire. In this section, the prototype algorithms are presented using simple and recursive mutexes to prevent the race condition on a data update. Three variants are considered. The first, marked as B1.1, is using simple mutex to protect loop over code numbers, the second, marked as B2.1, is using recursive mutex again protecting the loop, and finally the one protecting just the update operation using simple mutex, marked as B1.2. All variants are illustrated in Algorithm 6.

Algorithm 6: Prototype code - matrix assembly with explicit synchronization using simple and recursive POSIX mutexes

```
001  void Assembly Element Matrix ( ... )
002    for elem = 1, nelem
003     K^e = compute Element Matrix (elem)
004     Loc^e = give Element Code Numbers (elem)
005     pthread_mutex_lock(&mutex_SIM) (B1.1)
006     pthread_mutex_lock(&mutex_REC) (B2.1)
007      for i = 1, Size(Loc^e)
008        for j = 1, Size(Loc^e)
009          pthread_mutex_lock(&mutex_SIM) (B1.2)
010           K(Loc^e(i), Loc^e(j))+ = K^e(i, j)
011          pthread_mutex_unlock(&mutex_SIM) (B1.2)
012    pthread_mutex_unlock(&mutex_SIM) (B1.1)
013    pthread_mutex_unlock(&mutex_REC) (B2.1)
```

2.3. Synchronization using Colouring Algorithm

As already discussed in previous sections, the conservative strategy on always protecting the update operation may not lead to optimal results. It enforces the serial execution of the update operation for selected values, regardless if there is a real conflict or not. The fact that it prevents a parallel execution can have a significant impact on scalability. Partially, this problem has been addressed in the algorithm using array of locks preventing the update to block a row of values. In this section, we present an alternative approach, which is based on the idea of assigning the individual elements into groups, where elements in

a group should not share any node. This essentially means that elements in a group can be processed concurrently as during the assembly operation only distinct values of a global vector/matrix can be updated. After determining the element distribution into the groups, the algorithm loops over the groups and each group is processed in parallel. The objective is to keep the number of groups minimal. This is known as the so-called colouring algorithm in graph theory [8].

In this paragraph, the introduction to the vertex colouring algorithm will be given. Consider a graph of n mutually connected vertices (representing FE elements). The edges (connections) represent the element connectivity, i.e., the edge between two vertices represents a case, when two elements share the same node. The task is to assign a "colour" to each vertex under the condition that no neighbour has the same colour and keep the number of "colours" minimal. The algorithm for greedy colouring of a graph is the following:

(1.) Loop over the elements $e = 1, n_e$

(a) For the element e, find the colour C assigned to neighbours of element e

- loop over the nodes of element e, $i = 1, n$
- $C = C + fn_{(i)}$

(b) Find the unused available colour not in C and assign it to element e

- loop over the available colours, $c = 1, m$
- if c is not in C then $EC_{(e)} = c$

where EC is the array of colours and $fn_{(i)}$ is the function returning a set of colours assigned to elements sharing the node i

The computational cost of the Greedy colouring algorithm depends heavily on the vertex ordering. In the worst case, the behaviour is poor and solving the of algorithm can take a lot of computation time. However, the graph construction and graph colouring has to be performed only once during the FE code initialization and after that it can be reused in any assembly operation. As already noted, the colouring algorithm splits the elements into groups marked with different colours. The algorithm ensures that a minimum number of colours is used. Once the colouring is available, the assembly algorithm consists of the outer loop over individual colour groups and inner loop over individual elements in a group. Inside the inner loop, the element contributions are evaluated and assembled into global vector/matrix. Now the key is that the inner loop can be parallelized (individual members of a group can be processed by different threads) without the need of synchronization, as the colouring ensures that the race condition on update could not occur. Even though this is appealing, the algorithm has its overhead. This includes the already discussed need to establish the colouring, however, only the inner loop can be parallelized. All threads should finish processing the element's inner loop before processing elements for the next colour. This requires synchronization. Finally, there is also an overhead connected to the creation and termination of threads for each colour. In the colouring algorithm, the individual threads are created and terminated inside the outer loop over individual colours, but this overhead is typically very small, considering typical assembly times for real problems. Eventually, it can have some minor impact on overhead costs.

The prototype code for colouring assembly is presented in Algorithm 7 using OpenMP directives. The additional arguments of the parallel loop directive are used to declare some variables as shared (i.e., each thread accesses the same variable) or private (each thread has its own copy of that variable). The for-loop clause allows accumulating a shared variable without an explicit synchronization.

Algorithm 7: Prototype code - matrix assembly without explicit synchronization using Colouring Algorithm *(OpenMP)*

```
001 for ii = 0, number.of.colours
002 # pragma omp parallel for private(K^e, Loc^e)
003    for ie = 1, Size(colour.group[ii])
004     elem = colour.group[ii][ie]
005     K^e = computeElementMatrix(elem)
006     Loc^e = giveElementCodeNumber(elem)
007        for i = 1, Size(Loc^e)
008           for j = 1, Size(Loc^e)
009              K(Loc^e(i), Loc^e(j))+ = K^e(i, j)
```

The POSIX Threads variant of the colouring based assembly algorithm is presented in Algorithm 8.

Algorithm 8: Prototype code - matrix assembly without explicit synchronization using Colouring Algorithm *(POSIX Threads)*

```
001 threads = new pthread _t[num_threads]
002 for ii = 0, number.of.colours
003  for jj = 0, (num_threads − 1)
004    psize = size(colour.group[ii])/num_threads
005    start.indx = (jj) ∗ psize
006    end.indx = start.indx + psize
007    pthread_create( &threads[jj], NULL, Assembly
   Element Matrix, ii, start.indx, end.indx)
008  for jj = 0, (num_threads − 1)
009    pthread_join( &threads[jj], NULL)

010 void Assembly Element Matrix ( ... )
011  for ie = start.indx, end.indx)
012   elem = colour.group[ii][ie]
013   K^e = compute Element Matrix (elem)
014   Loc^e = give Element Code Numbers (elem)
015      for i = 1, Size(Loc^e)
016        for j = 1, Size(Loc^e)
017          K(Loc^e(i), Loc^e(j))+ = K^e(i, j)
```

2.4. C++11 Threads

Alongside well established OpenMP and Posix Threads, the C++11 standard has introduced a native C++ thread support [9]. The C++11 Thread libraries include utilities for creating and managing threads, which are standardized for C/C++ language.

The C++11 standard library contains classes for a thread manipulation and synchronization, common protected data, and low-level atomic operations.

The parallel program based on C++11 standard library is constructed by defining a new procedure function, which is executed by the thread and start the new thread. The synchronization in the C++11 standard is achieved by classical synchronization mechanisms like mutex object, condition variables, and other mechanisms like locks or controlling features used when threads are transferring computational data.

2.4.1. C++11 Threads - Simple Lock

In this paragraph, the multitasking synchronization using mutex class is presented. The mutex can be used to protect shared data from being simultaneously accessed by multiple threads. The synchronization is enforced in the loop, as illustrated in Algorithm 9.

Algorithm 9: Prototype code - matrix assembly with explicit synchronization using simple locks *(C++11 Threads)*

```
001 std :: mutexMTX
002 threads = new thread _t[Number_Of_Threads]
003 for i = 0, Number_Of_Threads
004     std::thread[Assembly Element Matrix, ... ]
005     for i = 0, Number_Of_Threads
006     std[i].join()

007 void Assembly Element Matrix ( ... )
008   for elem = 1, nelem
009    K^e = compute Element Matrix (elem)
010    Loc^e = give Element Code Numbers (elem)
011   MTX.lock()
012       for i = 1, Size(Loc^e)
013         for j = 1, Size(Loc^e)
014           K(Loc^e(i), Loc^e(j))+ = K^e(i, j)
015   MTX.unlock()
```

3. Performance evaluation

In this section, the performances and efficiencies of the presented approaches are compared both for matrix and vector assembly operations using two benchmark problems. The first benchmark problem is a 3D model of a nuclear containment, marked as *Jete* and shown in Figure 2. The second benchmark problem is a model of a 3D porous micro-structure, marked *Micro* and shown in Figure 3.

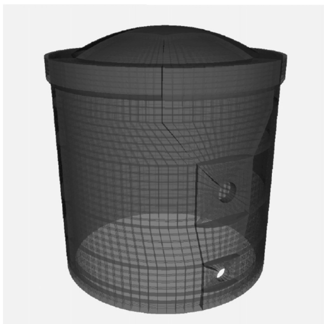

Figure 2. The benchmark problem of 3D finite element model of nuclear containment (*Jete*).

name	nnodes	nelems	neqs
Jete250k	$87k$	$67k$	$260k$
Jete3M	$899k$	$1M$	$3M$
Micro250k	$85k$	$80k$	$256k$
Micro3M	$1M$	$970k$	$3M$

Table 1. Discretizations of the benchmark problems considered where the *nnodes* represents the number of nodes, *nelems* represents number of elements and *neqs* represents number of equations.

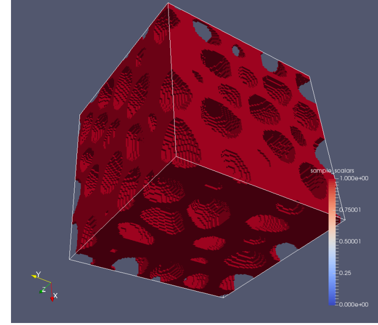

Figure 3. The benchmark problem of 3D finite element model of porous micro-structure cube (*Micro*).

The different discretizations are generated for both benchmark problems with an increasing number of elements, see Table 1. The tetrahedral elements with a linear approximation have been used in the case of the nuclear containment benchmark, while a structured grid of brick elements with linear interpolation was used for the micro-structure benchmark. The porous micro-structure consists of two phases representing the material and voids with a different set of elastic constants.

In both cases, the linear structural analysis has been performed and structures were loaded by self-weight, which implied a nonzero contribution of every element to the external load vector. The benchmark problems are characterized by different sparsity of the system matrix. The model of the porous micro-structure has significantly more nonzero members than the model of the nuclear containment. For example, a number of nonzero members in the stiffness matrix of the problem is $433M$ in the case of *Jete3M*, while the model of the porous micro-structure *Micro3M* has $1528M$ nonzero entries, with the number of unknowns being similar.

All the presented parallelization approaches have been implemented in the OOFEM finite element code. The object oriented C++ OOFEM code has been compiled using *g++ 4.5.3.1* compiler version with optimization flags *-O2*. The tests have been executed on a Linux workstation (running Ubuntu 14.04 OS) with two CPUs *Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz* and *132GB RAM*. Each CPU consists of eight physical and sixteen logical cores, allowing up to thirty-two threads to run simultaneously on the workstation. All the tests fit into the system memory.

For each benchmark problem, the individual strategies have been executed on increasing number of processing cores and speedups (with respect to serial single CPU execution) have been evaluated as an average from three consecutive runs. The performance of individual strategies of the vector assembly (in terms of the achieved speedup versus increasing the number of processors) for *Jete250k* test are presented in Figure 4 and for matrix assembly in Figure 8. Similarly, for the *Jete3M* test, the results are presented in Figures 5, 9, for the *Micro250k* test in Figures 6, 10 and, finally, for the *Micro3M* test in Figures 7, 11. The following notation is used on above mentioned figures to distinguish individual parallelization strategies implemented: OpenMP with *critical sections* (*OMP - CS*), OpenMP with simple locks (*OMP - L*), OpenMP with nested locks (*OMP - NL*), OpenMP with *critical sections* only in update operation of global matrix/vector (*OMP - LCS*), OpenMP with atomic update directive only in update operation of global matrix/vector (*OMP - LATO*), OpenMP with simple locks only in update operation of global matrix/vector (*OMP - LL*), OpenMP using blocks of simple locks (*OMP - B50, OMP - B500, OMP - B10$\hat{5}$*), OpenMP based on colouring (*OMP - CP*), POSIX Threads based on colouring (*PTH - CP*), POSIX Threads with simple mutexes (*PTH - M*), POSIX Threads with recursive mutexes (*PTH - RM*) and finally C++11 Threads with simple mutexes (*THR - M*). The assembly process of the matrix/vector is composed of an evaluation of the local matrix/vector followed by its assembly into global matrix/vector. The execution times of these two operations together with the total execution time are presented in Table 2 for selected benchmark problems.

Note that the complexity of the greedy colouring algorithm is linearly proportional to the number of elements and thus it is the same as the complexity of assembly algorithm. However, as already mentioned, the colouring is to be determined only once and can be reused in all subsequent assembly steps. Theoretically, the algorithms that use colouring strategies should scale linearly up to the limit imposed by the memory bandwidth, but the implementations tested in the paper could not achieve these performance levels. This requires further investigations that are beyond the scope of this paper.

The results for the vector assembly show very similar trend. All the strategies yield approximately the same results in terms of scalability and speedups. From the results, it is evident that the speedups are far from ideal. For example, in the case of *Jete3M* test, the speedups for 16 CPUs are in the range of 2.5 - 4. There are several possible reasons why only sub-optimal scalability has been obtained. The first reason is that the individual tasks are not independent, the update of the global entry is a part that can be processed only by one thread at a time. Second, there is an additional overhead connected with the parallel algorithm (thread creation and management, synchronization) that is not present in the serial version. Third, the individual threads share the common resources (memory bus), which do not appropriately scale in performance. Moreover, from the results, one can observe the performance drop after reaching the limit on 16 threads. This can be attributed to the hyper-threading technology specific to Intel processors [10], which shares some of the CPU resources (execution engine, caches, and bus interface) between the hyper-threaded cores. This trend is more pronounced on larger tests (*Jete3M* and *Micro3M*). The POSIX threads and C++11 Threads implementations show better performance than OpenMP versions, particularly for smaller number of threads.

The results for the sparse matrix assembly show different trends. Some strategies (*OMP-LCS, OMP-LL, OMP-B50, OMP-B500, OMP-B10^5)* were not even able to reach the performance of the serial algorithm and speedups are less than 1. The Colouring based strategies (*OMP-CP, PTH-CP*) have a similar speedup trend for *Jete250k* and *Micro3M* benchmark problems. However, the Colouring based strategy *PTH-CP* clerly shows a better scalability than the Colouring based strategy *OMP-CP* on benchmark *Jete3M*. The opposite trend can be observed for Colouring strategies on benchmark *Micro250k* (*OMP-CP* clearly shows a better scalability than *PTH-CP*). The implementations based on the colouring algorithm do not perform well, which is somehow surprising observation. This could be (partially) explained by so-called false sharing. A typical SMP system has local data cache organized hierarchically in several levels [11]. The system guarantees cache coherence. In the case when one core modifies the memory location that also resides on the different core cache line then the false sharing occurs. This is going to invalidate the cache lines on other cores and forcing to re-read them every time when any other thread has updated the memory in the cache line. The false sharing seems to have a much bigger impact on sparse matrix assembly performance than on the vector assembly performance. The false sharing effect in our case is confirmed using the valgrind tool with memory management and threading bugs monitoring. The benchmark problem *Jete250k* with using *OMP-CS* or *OMP-CP* based on 32 computational threads is used as a representative example for illustrating false sharing effect and the outputs from valgrind are presented in Table 3. Cache accesses for data in table are presented. The parameter *D refs* represents the number of data fetches (summary of reads and writes data) in the system cache memory. The parameter *D1 misses* represents the number of data fetches in cache memory layer L1. The last parameter *DLL misses] represent the instruction and data fetches at the last level cache (L3).* From the reported results, one can see a slight increase of the number of cache misses in the case of colouring algorithm compared to the approach using the synchronization based on

Number of threads	1	2	4	8	12	16	32
Total time [s]	36.7	23.81	13.83	11.70	13.51	12.34	16.59
Evaluation of local matrix [s]	13.83	7.07	3.61	1.73	1.23	0.93	0.465
Localization into global matrix [s]	22.87	16.74	10.22	9.97	12.28	11.41	16.225

Table 2. Matrix assembly total times with dividing to evaluation of local matrix assembly times and localization into global matrix assembly times of the benchmark problem considered (Jete 3M).

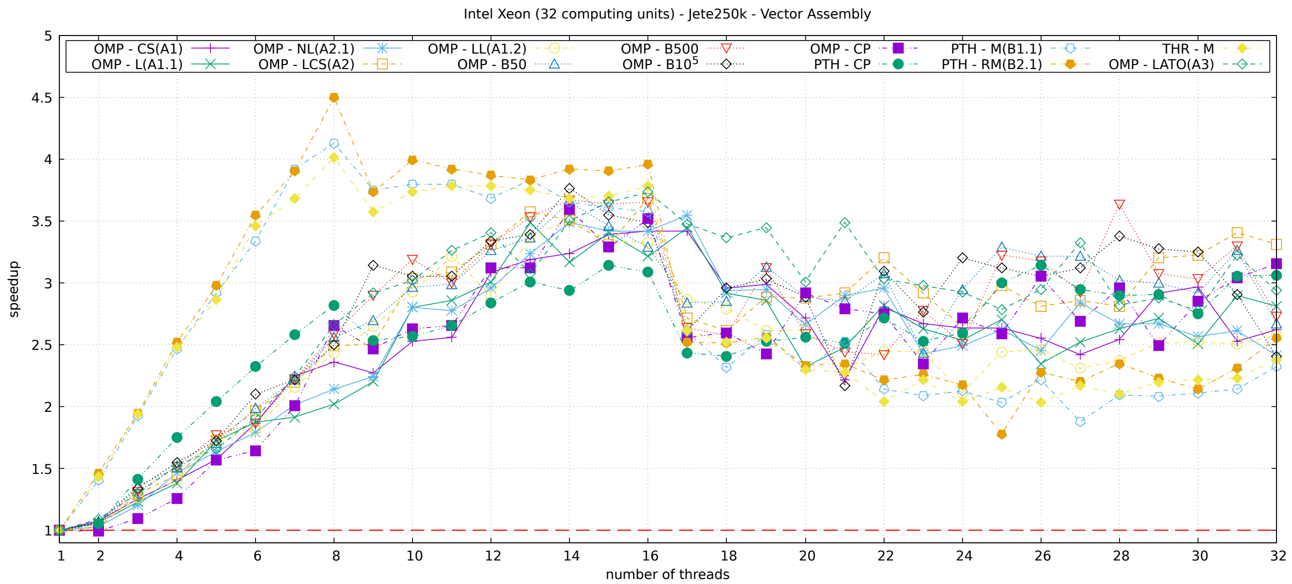

Figure 4. Speedups of the external force vector for benchmark *Jete250k*.

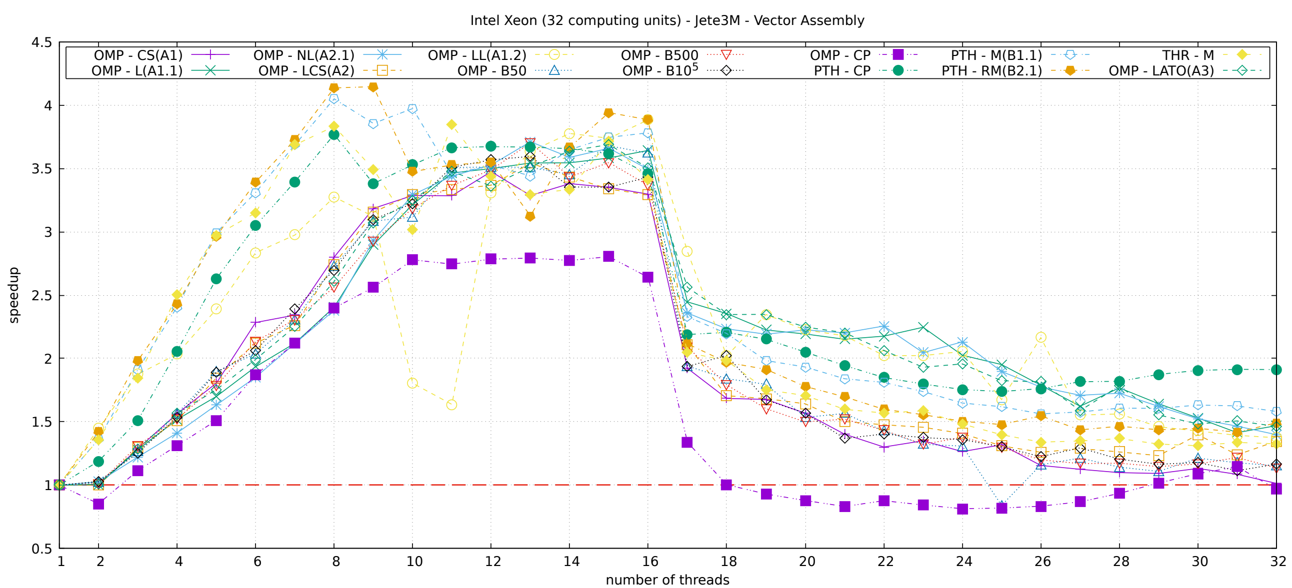

Figure 5. Speedups of the external force vector assembly for benchmark *Jete3M*.

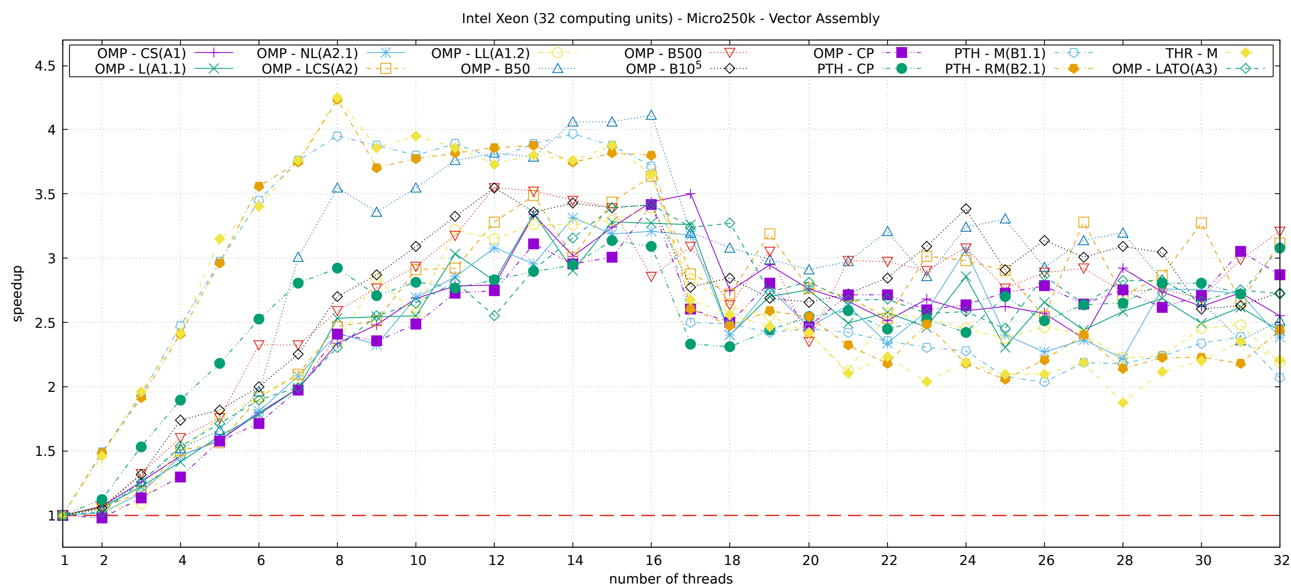

FIGURE 6. Speedups of the external force vector assembly for benchmark *Micro250k*.

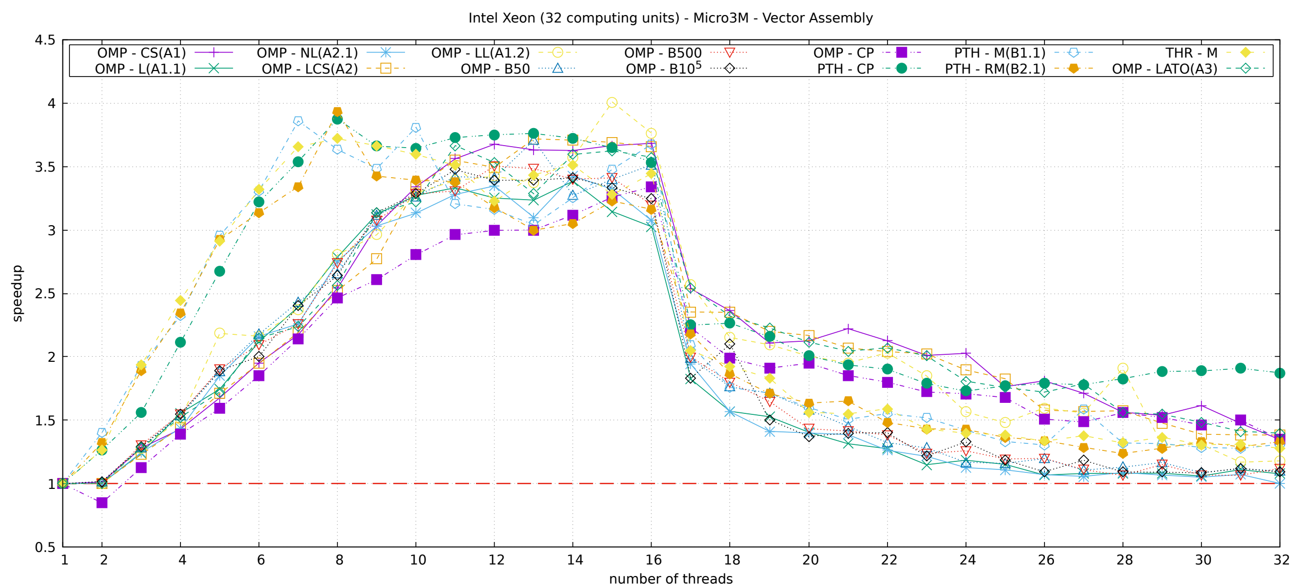

FIGURE 7. Speedups of the external force vector assembly for benchmark *Micro3M*.

OpenMP *critical sections*. To get a further insight into the problem, the Intel VTune Amplifier tool has been used to detect false sharing [12]. The memory access analysis type uses hardware event sampling to collect data for the metric. This monitoring reveals the growing importance of the false sharing effect with increasing number of threads (particularly for 16 and more threads, when Intel hyper-threading technology is active). This is demonstrated by growing memory bound (showing a fraction of cycles spent on waiting due to load or store operations on each cache level) and average memory latency (average load latency in cycles) characteristics.

The individual speedups with and without synchronization for different scheduling options for selected benchmark problem (Jete3M) using critical section synchronization (A1) are presented in Figures 12 and 13. The results are quite interesting. From the Fig. 12 (matrix assembly), one can clearly see that all the executions with synchronization outperform the ones without it. In our opinion, this indicates that the synchronization overhead is more than balanced by the better performance, which is most likely due to single

	OMP-CS	OMP-CP
D refs	547 165 746 467	558 299 751 110
D1 misses	42 487 418 875	43 071 933 527
LLd misses	16 625 383 973	16 650 025 976

TABLE 3. Number of Cache data misses of the benchmark problem *Jete250k*.

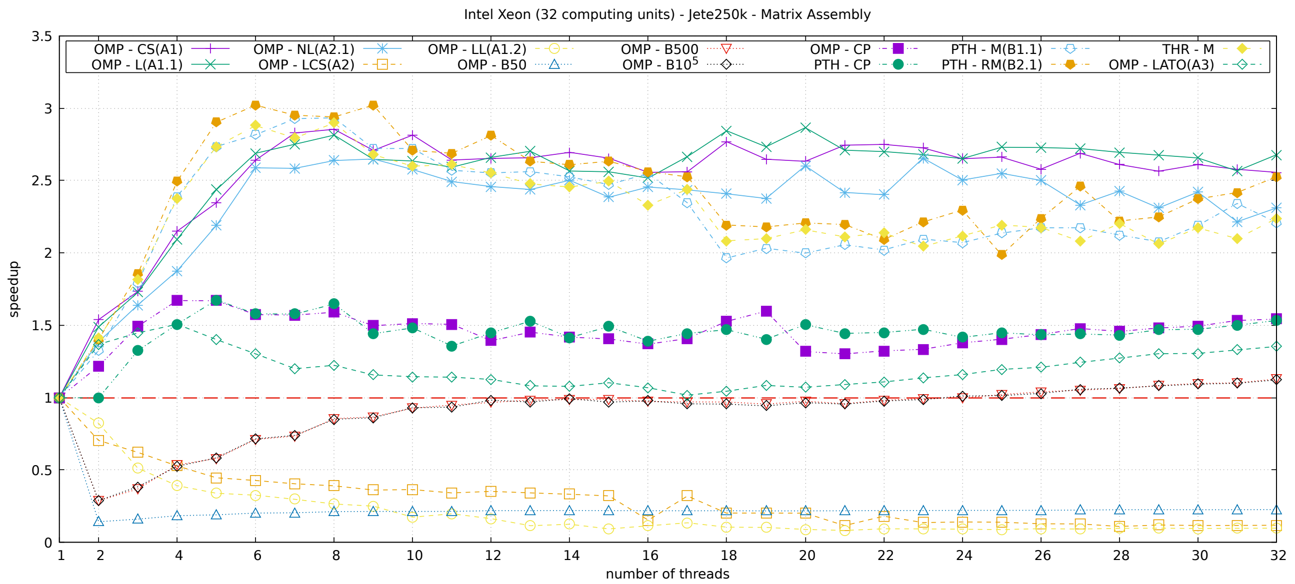

Figure 8. Speedups of the sparse matrix assembly for benchmark *Jete250k*.

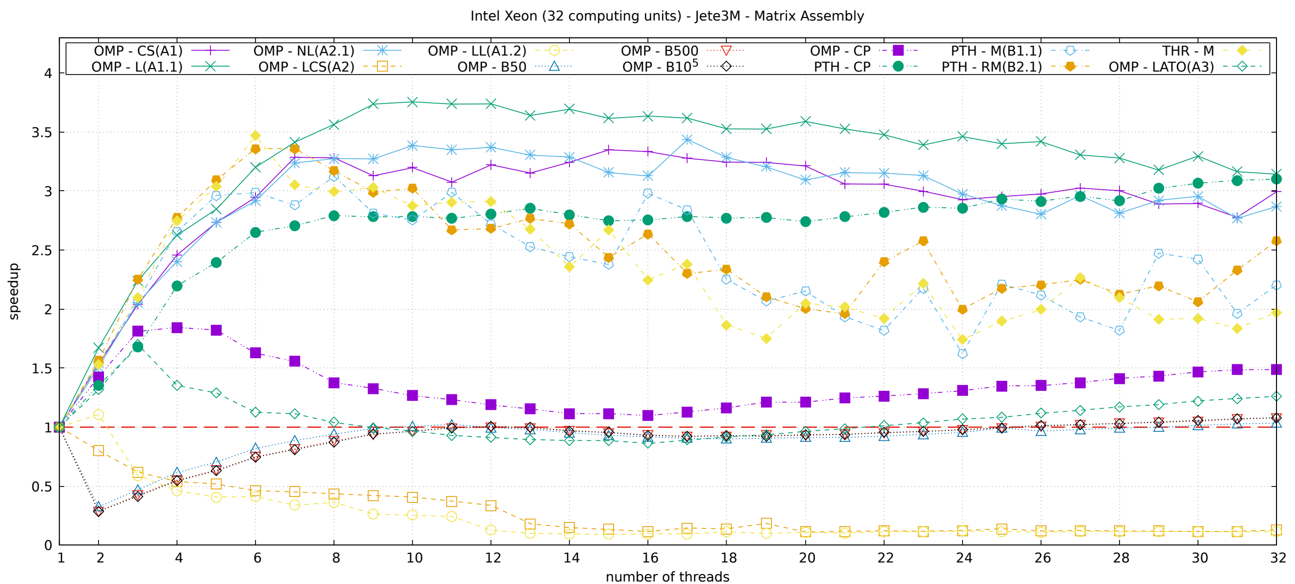

Figure 9. Speedups of the sparse matrix assembly for benchmark *Jete3M*.

memory access at a specific time (ensured by critical section). This can be attributed to false sharing. At the same time, the results demonstrate that the effect of scheduling is negligible for executions without synchronization (symbols without fill) and only partially relevant for executions with synchronization (filled symbols). The effect of scheduling is relatively small. This indicates that limitation in memory bandwidth is the main reason for the sub-optimal scalability obtained.

The results for vector assembly (Fig. 13) are different, as the differences between speedups with and without synchronization are much smaller. This is most likely due to the less complex memory access pattern in the case of vector assembly. Similarly to matrix assembly, the results demonstrate that the effect of scheduling is negligible as well as the effect of different scheduling options. The results again confirm the significant role of the limited memory bandwidth.

POSIX threads and C++11 Threads implementations performed best for lower number of threads, but overall, the OpenMP implementation (*OMP-L, OMP-NL*), and *OMP-CS* performed the best.

In [4] authors reported a speedup 11 for 12 threads. However, these results are achieved on a slightly different architecture, where the computational node consists of two processors Intel Xeon X5650 2,66GHz with 6 cores and results for 12 threads are given. Therefore, the results were not affected by hyperthreading.

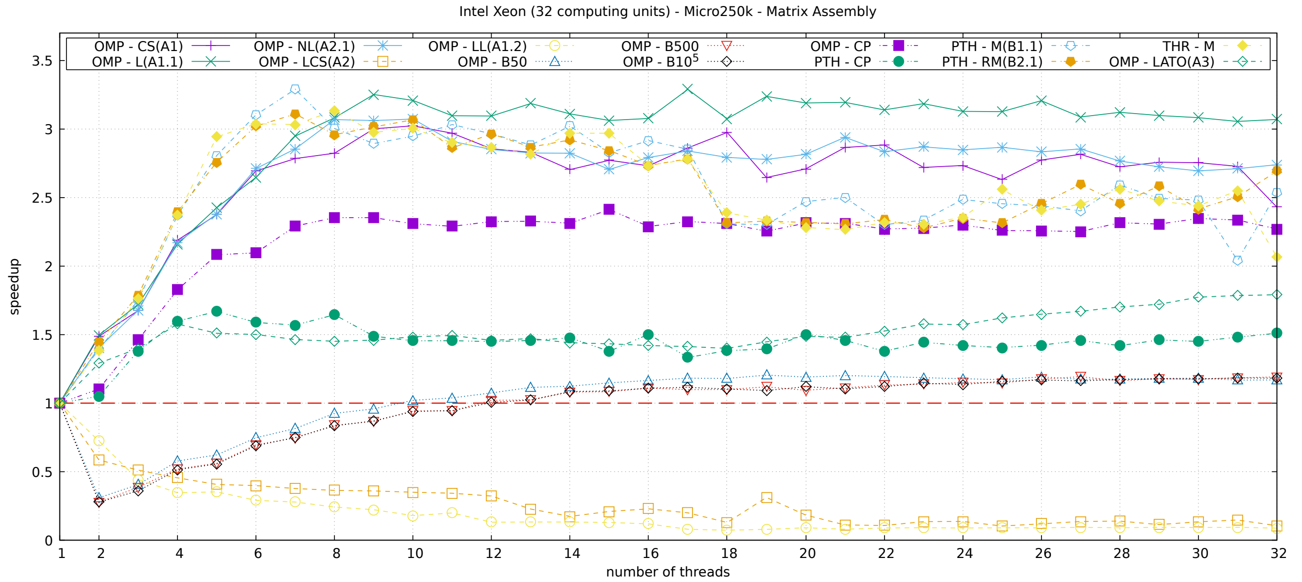

Figure 10. Speedups of the sparse matrix assembly for benchmark *Micro250k*.

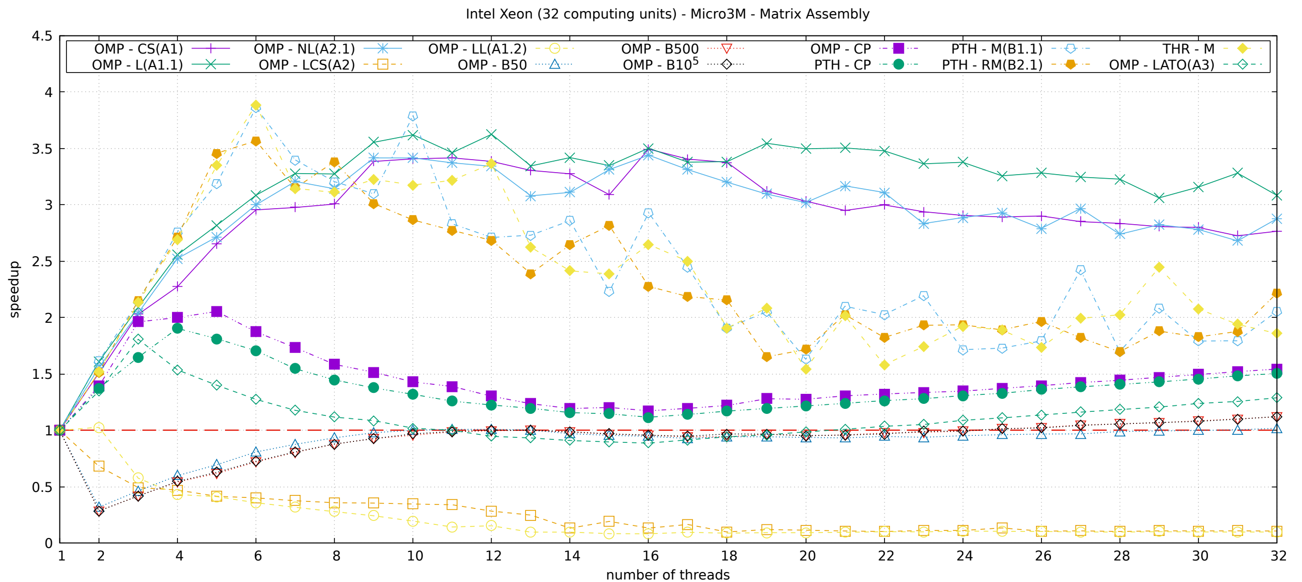

Figure 11. Speedups of the sparse matrix assembly for benchmark *Micro3M*.

4. Conclusions

The paper evaluates different parallelization strategies of right-hand side vector and stiffness matrix assembly operations, which are one of the critical operations in any finite element software. The performance strategies use different techniques to ensure consistency and are implemented using OpenMP, POSIX Threads and C++11 Threads libraries. The performance of individual strategies and libraries is evaluated using two benchmark problems (a 3D structural analysis of a nuclear containment and of a 3D micro-structure) each with two different mesh sizes. For the particular benchmark cases considered, the performance of nearly all strategies has been much better than the performance of serial algorithm with a relatively good scalability, however, in the case of matrix assembly, considerable differences exist and the presented work provides an insight on how to select the optimal strategy.

The achieved results clearly show a performance drop on systems with hyper-threading technology when the number of processes exceeds the number of physical cores. Somehow disappointing are the results of the assembly based on the Colouring approach that did not perform as expected, with the performance being affected most likely with false-sharing phenomena.

The main conclusion of this study is that the performances of individual libraries are comparable, but performances of individual strategies differ, often significantly.

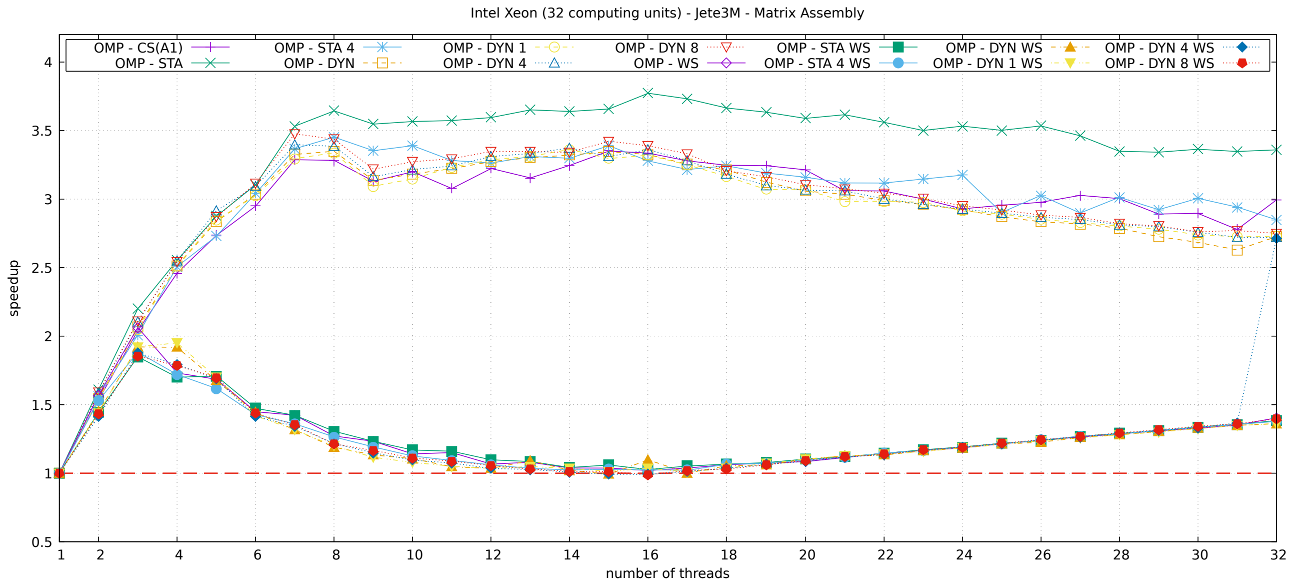

Figure 12. Speedups of the sparse matrix assembly with and without synchronization for different scheduling options for benchmark *Jete3M* using critical section synchronization (A1).

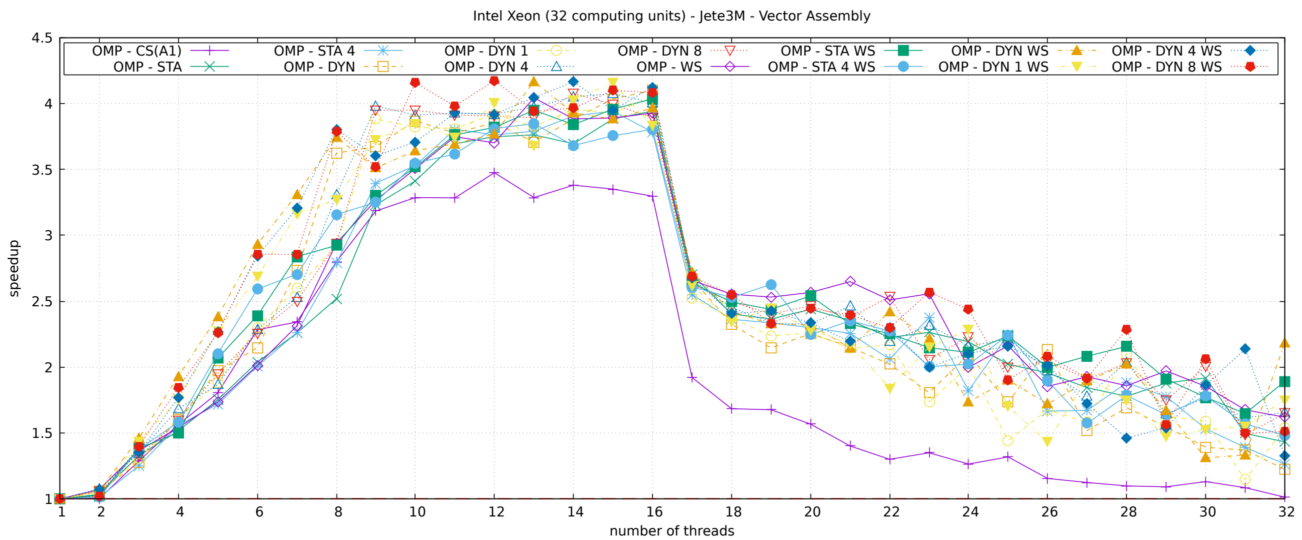

Figure 13. Speedups of the external force vector assembly with and without synchronization for different scheduling options for benchmark *Jete3M* using critical section synchronization (A1).

In general, the presented paper illustrates the potential of parallel assembly operations and importance of benchmarking, allowing to identify an optimal strategy.

Acknowledgements

This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS16/038/OHK1/1T/11 - Advanced algorithms for numerical modeling in mechanics of structures and materials. The second author was supported by OP VVV, Research Center for Informatics, CZ.02.1.01/0.0/0.0/16019/0000765.

References

[1] B. Patzak. OOFEM. `http://www.oofem.org`, 2000.

[2] C. Hughes, T. Hughes. *Parallel and Distributed programming Using C++*. Pearson Education, 2003.

[3] H. Meuer, E. Strohmaier, J. Dongarra, et al. Top 500 the list home page. `https://www.top500.org/`.

[4] P. Jarzebski, K. Wisniewski, R. L. Taylor. On parallelization of the loop over elements in FEAP. *Computational Mechanics* **56**(1):77–86, 2015. DOI:10.1007/s00466-015-1156-z.

[5] B. Barney. OpenMP. `https://computing.llnl.gov/tutorials/openMP/`, 2014.

[6] A. Marowka. Book Review [review of "Using OpenMP: Portable Shared Memory Parallel Programming" (Chapman, B. et al, 2007)]. *IEEE Distributed Systems Online* **9**(1):3–3, 2008. DOI:10.1109/mdso.2008.1.

[7] B. Nicols, D. Buttlar, J. P. Farrell. *Pthreads Programming*. O'Reilly and Associates, 1996.

[8] D. B. West. *Introduction to graph theory.*, vol. 2. Upper Saddle River Prentice hall, 2001.

[9] A. Williams. *C++ Concurrency in Action*. Manning Publications Co, United States of America, 2012.

[10] D. T. Marr, F. Binns, D. L. Hill, et al. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal* **6**(1), 2002.

[11] J. Handy. *The cache memory book*. Academic Press, Inc., 1998.

[12] Intel Corporation. Intel VTune Performance Analyzer. `http://software.intel.com/en-us/intel-vtune/`, 2019.