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

PARALLEL COMPUTATIONS AND C++ STANDARD LIBRARIES

V. Rek*, I. Němec**

Abstract: *In this paper, the form of modifications of the existing code written in C programming language for the calculation of structures using the explicit form of the Finite Element Method is introduced. It uses the possibilities of multithread running, which is now supported at the level of native C++ programming language using standard libraries. Thanks to the high degree of abstraction that is provided by contemporary C++ programming language, a respective library created in this way can be generalized for other purposes of usage of parallelism in computational mechanics.*

Keywords: Finite Element Method, Explicit Form of the Finite Element Method, Central Difference Method, Parallel Computing, C++ Standard Libraries.

1. Introduction

Due to the significant expansion of multi-core processors in the past decade, many older applications designed to run sequentially have begun to become obsolete, mainly due to the performance of available hardware.

As in the past, and even now, development tools for the development of software applications are slightly lagging behind the choices of available hardware. Alternatives of how to benefit from multi-core processors were technologies such as Intel Threading Building Blocks or Intel Cilk++, or others which also require special installation. Another possibility was to use an application interface provided directly for the respective operating system such as Windows Win32 API (Hart, 2015) or POSIX (Kerrisk , 2010) for Unix-like operating systems, which are often quite cumbersome and limited to the possibilities of C programming language (Prata, 2004).

Since the year 2011, when the new standard of C++ programming language was introduced (version 11), developers have been given the possibility to use threads and all other necessary resources to support thread synchronization (mutex, semaphore, etc.) on the level of the native programming language using its standard libraries (Williams, 2010). Those abilities included programming languages like Java or C#. The main disadvantages are their performance and occasionally their limited portability.

Using the new version of the C++ programming language has been possible for some time in Microsoft Compiler since the Microsoft Visual Studio 2010 IDE (abbrev. for Integrated Development Environment) or in some freely available compilers.

The new standard libraries of C++ programming language already provide an effective interconnection of strong object-oriented programming language and multithread running, which until recently, had been largely limited. Computational software tools that are already written in C or C++ programming languages are now able to enrich the possibility to use parallelism while maintaining the portability of code. In a field of an explicit form of the Finite Element Method (Wu & Gu, 2012) the possibility of usage of parallelism is more than desirable, mainly because of time-consuming calculations, which is caused by the conditional stability of explicit methods used for the direct integration of equations of motion.

* Ing. Václav Rek: Institute of Structural Mechanics, Faculty of Civil Engineering, Brno University of Technology, Veveří 95; 602 00, Brno; CZ, rekv@seznam.cz

** Doc. Ing. Ivan Němec, CSc.: Institute of Structural Mechanics, Faculty of Civil Engineering, Brno University of Technology; Veveří 95; 602 00, Brno; CZ, nemec@fem.cz

2. Explicit Form of the Finite Element Method

This is an explicit algorithm that was originally designed for parallel computing on graphics cards using NVIDIA CUDA (Rek & Němec, 2016).

The algorithm is based on the explicit form of the Finite Element Method, which is based on an artificial transformation of the problem of statics to dynamic problem with damping. At the end we get semi-discrete second-order differential equations of motion, which we use for direct integration by the finite difference method (Har & Tamma, 2012).

$$M\{\ddot{q}\} + C\{\dot{q}\} + K\{q\} = \{F_E\}, \tag{1}$$

where $\{\ddot{q}\}$, $\{\dot{q}\}$ and $\{q\}$ are vectors containing nodal accelerations, velocities and displacements respectively. $K\{q\}$ represents internal forces $(\{F_I\})$ and $\{F_E\}$ the external forces. M is the lumped mass matrix and C is the damping diagonal matrix.

Using the Central Difference Method for direct integration of equations o motion we obtain the following explicit expression for the calculation of the new displacement. For the i_{th} degree of freedom, equation (1) leads to the explicit formula as follows:

$$q_{h,i}^{n+1,df} = \underline{\alpha}^{df}\left(F_{E,i}^{n,df} + F_{I,i}^{n,df}\right) + \underline{\beta}^{df} q_{h,i}^{n,df} - \underline{\gamma}^{df} q_{h,i}^{n-1,df}, \tag{2}$$

where

$$\underline{\alpha}^{df} = \frac{2\Delta t^2}{2m_i^{df} + c_i^{df}\Delta t}, \quad \underline{\beta}^{df} = \frac{4m_i^{df}}{2m_i^{df} + c_i^{df}\Delta t}, \quad \underline{\gamma}^{df} = \frac{2m_i^{df} - c_i^{df}\Delta t}{2m_i^{df} + c_i^{df}\Delta t}.$$

3. Parallel Algorithm

The algorithm is based on a parallel assembly of stiffness matrices for each of the finite elements and the subsequent calculation of the new displacements using explicit formula (2). Assembly of stiffness matrices and external force vector for continuous load is schematically shown in Fig. 1.

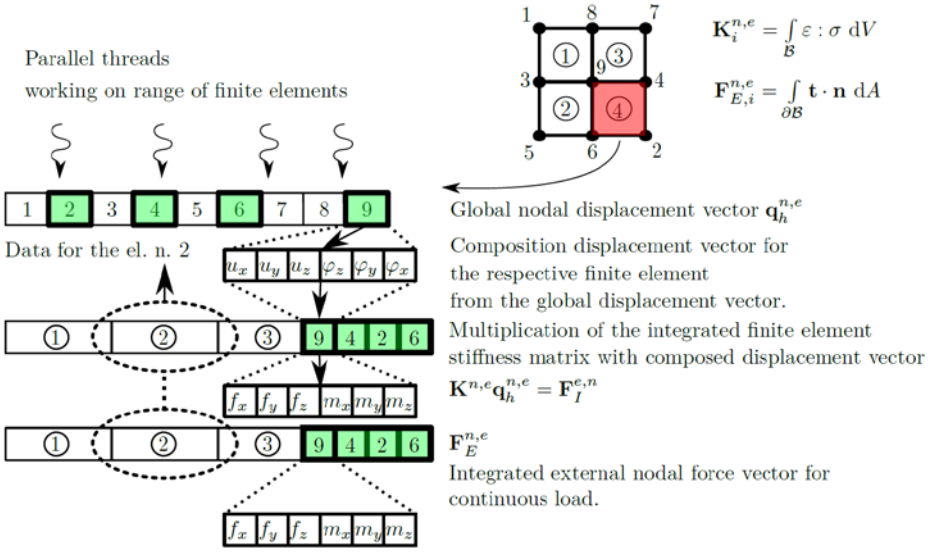

Fig. 1: Parallel composition of external and internal force vectors.

For each finite element node, the sum of the contributions from the neighboring finite elements for the respective node is performed. This procedure is simply shown in Fig. 2.

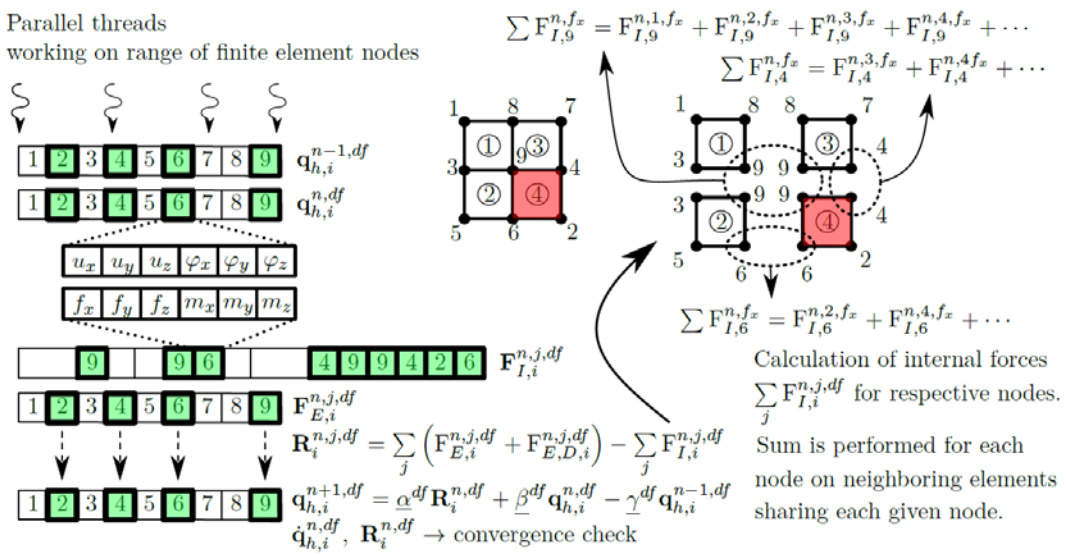

Fig. 2: Parallel composition of new deflections of finite element nodes.

4. C++ Implementation

For the purpose of compiling multithreaded run a general generic class "FEVRThreadFactory" was composed. It is based on the use of generic functions, which are then executed on the cores of a multi-core processor according to the settings which come from the object designed for scheduling of respective problem. The declaration and bindings of classes, which constitute the respective model of the solver, including the declaration of input data for each thread is in Fig. 3. A diagram of the synchronization object and global computing functions is in Fig. 4.

The underlying algorithm originally uses synchronization barrier only. This is frequently used synchronization procedure in GPGPU. Due to the nature of the algorithm used, a different type of synchronization is not needed. Using the C++ native threads and its standard libraries is well described in book C++ Concurrency in Action: Practical Multithreading (Williams, 2010).

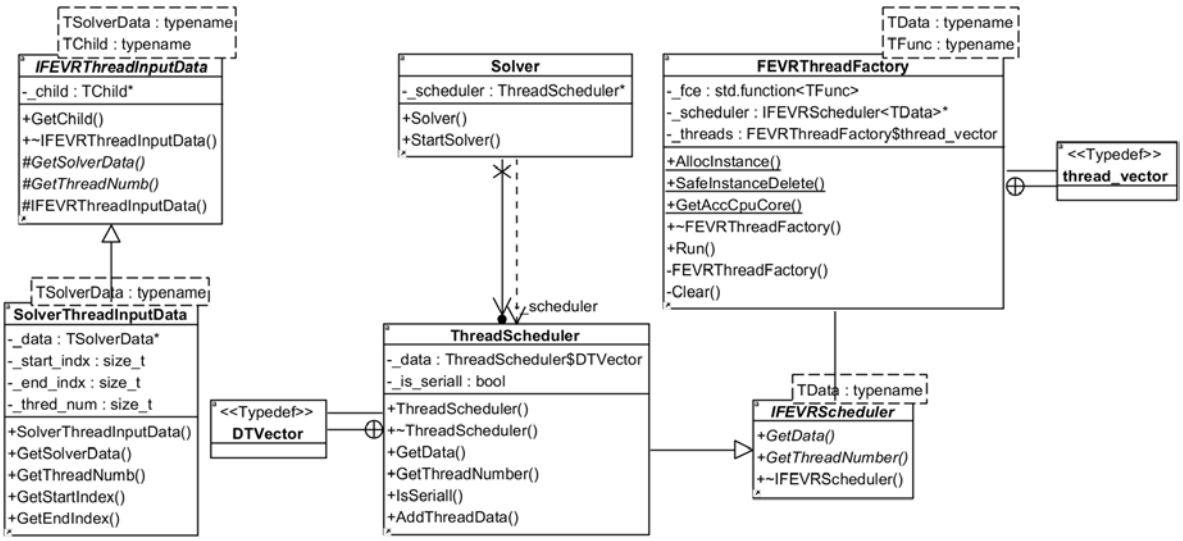

Fig. 3: UML class diagram of parallel solver.

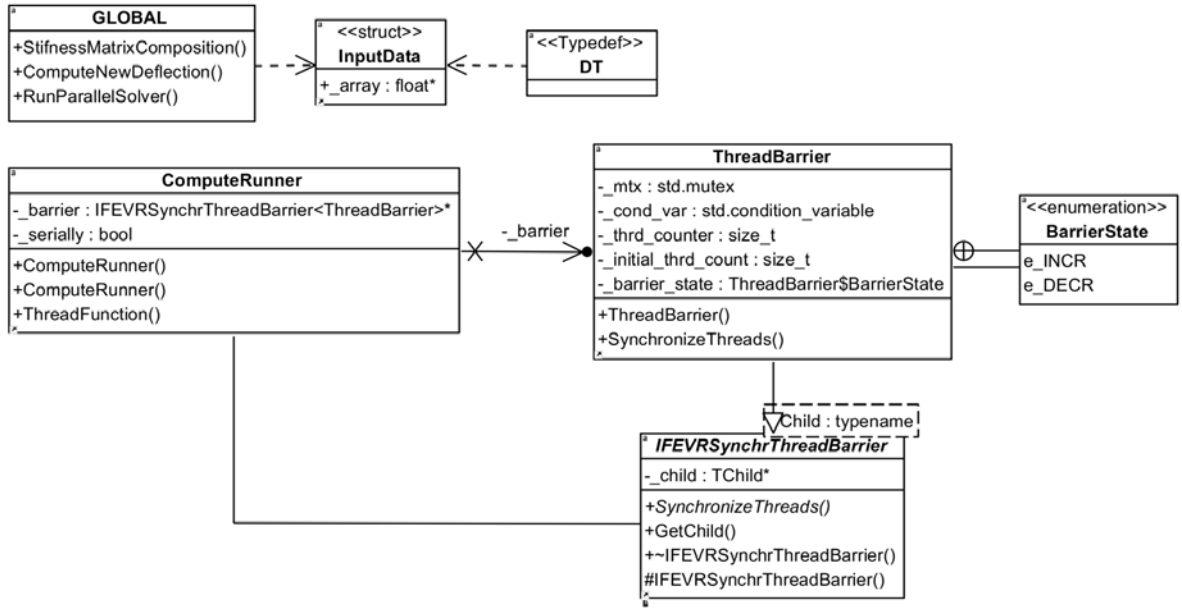

Fig. 4: UML class diagram of synchronization barrier object and encapsulating class for the computational global functions.

5. Performance Test

Test was performed on a model that consists 5 000 finite elements (30 906 DOF) with 100 time steps. The effectiveness of the used algorithm was tested on the three different processors as follows: Intel Core2 Duo SU9400 - 1.40 GHz (2 Cores / 2 Threads), Intel Core i5-3320M Ivy Bridge - 2.6 GHz (2 Cores / 4 Threads) and Intel Core i5-4690 3.5 GHz (4 Cores / 4 Threads). Achieved performance of execution (Sequential time [s] / Parallel time [s]): 1.119, 1.972, 3.826.

6. Conclusions

The introduced approach to the potential usage of the modern form of the C++ programming language and its new standard libraries allows us to make better use of the support of parallel computations on the level of native programming language. Functions from the NVIDIA CUDA library can be easily applied in a C++ code, and thus, the existing computational tool could be adapted to exploit the opportunities of multicore processors. Respective tests on different processors proved the effectiveness of the used approach.

References

Hart, J. M. (2015) Windows System Programming (Addison-Wesley Microsoft Technology), Fourth Edition, Addison-Wesley Professional.

Kerrisk, M. (2010) The Linux Programming Interface: A Linux and UNIX System Programming Handbook, No Starch Press.

Williams, A. (2010) C++ Concurrency in Action: Practical Multithreading, Manning Publications.

Wu, S. R. & Gu, L. (2012) Introduction to the Explicit Finite Element Method for Nonlinear Transient Dynamics, Wiley.

Rek, V. & Němec, I. (2016) Parallel Computing Procedure for Dynamic Relaxation Method on GPU Using NVIDIA's CUDA, Applied Mechanics and Materials, 821, pp. 331-337, Trans Tech Publications.

Prata, S. (2004) C Primer Plus, Fifth Edition, Sams Publishing.

Har, J. & Tamma, K. K. (2012) Advances in Computational Dynamics of Particles, Materials and Structures, Wiley.