Application of parallel computing in finite element analysis of constructions

S I Homenyk and V Yu Kozub

Zaporizhzhia national university, Zaporizhzhia, 69600, Ukraine

Abstract To optimize finite elements computational processes of the "MIRELA +" computing complex, parallel programming algorithms have been developed for creating stiffness matrices of finite elements and calculating the stress-strain. The parallel data processing is implemented using OpenMP library. On the basis of computational methods of experiments, the use of parallel computing algorithms at the beginning of solving problems with traditional finite element algorithms for large-dimensional grids has been established.

1. Introduction

One of the most common methods of calculating structures is the finite element method. Depending on the type of problem in finite-element modeling, the calculated grids reach large sizes, in particular, for composite multilayer structures, structures with stress concentrators, composite structures. In this case, the time of calculation of the stiffness matrix is a significant part of the total time to solve the problem.

The need to solve complex three-dimensional problems with minimal costs leads to the need to optimize computational processes. This is especially true of the application of the method of superelements, in which the construction of solving equations for individual superelements can be performed in parallel, which significantly reduces the calculation time [1].

The use of several computing units to solve a computational problem simultaneously is a parallel computation. Provided that the computational problem can be divided into separate, autonomous from each other, parts that can be processed simultaneously, thereby ensuring the solution of problems in less time with several computing units than with a single computing unit.

Usage of several computing units to solve a computational problem simultaneously is defined as parallel computation. Provided that the computational problem can be divided into separate, autonomous from each other, parts that can be processed simultaneously, thereby ensuring the solution of problems in less time with several computing units than with a single computing unit.

Computing units can be a single computer with multiple processors/cores, such as laptops with dual-core or quad-core processors or a supercomputer created from a large number of such computers (or nodes) connected by a network that creates a cluster. Each node consists of several cores that are connected in a socket. For example, the supercomputer Stampede2 is a cluster of 4200 nodes, each of which contains 68 cores (a total of 285,600 cores).

2. Advantages and disadvantages

Parallel programs have the following features:

- management of work of set of processes is carried out;
- data exchange between processes is organized;
- determinism of behavior is lost due to asynchrony of data access;
- non-local and dynamic errors predominate;
- there is a possibility of deadlocks;
- there are problems with the scalability of the program and balancing the load of computer nodes.

When using multiprocessor systems, it is expected that the speed of calculations will increase n times while using n processors instead of one. In fact, the acceleration is much less than expected. The decrease
in productivity growth in parallel computing is due to two factors: the first is related to the properties of the program algorithm (these properties were studied in 1967 by supercomputer developer Gene Amdal) [2], and the second - to the technical properties of the computer system.

The essence of the problem is that each parallel program contains some part of non-parallel code. On systems with shared memory, this is the part of the code that is executed only by the main thread, and on systems with distributed memory, it is the part of the code that is executed by all processors. Assume that the ratio of the execution time of the non-parallel part of the program to the total execution time is \( f \) \((0 < f < 1)\). If the execution time on one processor is equal to \( t_s \), then the execution time of the parallel part is \((1-f)t_s\), and the execution time of the serial part is \(-ft_s\). Ideally, when using \( n \) processors, the execution time of the parallel part will be \((1-ft_s)/n\), and the execution time of the serial part will remain \( ft_s\).

Thus, the acceleration of program execution on \( n \) processors will be:

\[
S(n) = \frac{t_s}{ft_s + (1-f)t_s/n} = \frac{n}{1+(n-1)f}.
\]

Expression (1) is a formalized definition of Amdahl's law. At \( n \), which goes to infinity, the acceleration will be \( 1/f \). It follows from this law that if part of the hardware in the program is 5 it is impossible to get more than 20 times the acceleration program, no matter how many processors are used.

When partitioning a large task into smaller subtasks that need to run in parallel on different processors, there is often a situation where there’s a need to wait for some processors to complete some execution stage. In this case, the total execution time will be determined by the slowest subtask. Therefore, when developing a parallel program, it is extremely important to distribute computing as evenly as possible between processors. Achieving a load-balanced state is a non-trivial task.

In shared-memory systems, where information is exchanged between processors using variables stored in shared memory, process synchronization is especially important. If several processes simultaneously try to modify the same variable, then it’s required a special mechanism for synchronizing the execution of processes to ensure the determinism of program execution.

The efficiency of parallel programs on systems with distributed memory significantly depends on the communication environment. The communication environment is characterized by two parameters: bandwidth, which determines the number of bytes transmitted per unit time, and latency, which determines the time spent preparing for the transmission of the message. For example, for a Gigabit Ethernet network bandwidth in MPI programs is about 70 Mbps, and the latency is about 40 μsec [3]. For a large number of tasks in which data exchange is infrequent and small in size, these parameters are quite satisfactory, however, in cases where the program transmits many small messages, such parameters become unacceptable, and the scalability of the program is extremely low. In any case, communication operations are much slower than accessing local memory, so the most effective will be those parallel programs in which exchanges are minimized.

3. Approaches

Approaches to parallelization in FEM can be divided into two categories - approaches with distributed memory and shared memory, which shown in Figure 1 and Figure 2 respectively. When working with the first category, a high-performance model of parallel programming Message Passing Interface (MPI) is usually used. MPI is an easily scalable means of parallel programming, but on the condition of workload distribution done, which usually leads to large-scale changes in the program [4,5]. An MPI program is a collection of independent processes, each of which runs its own program (not necessarily the same), written in C/C++ or Fortran. The processes of an MPI program interact with each other by calling communication procedures. Despite significant advances in the development of programming technology using the message transfer mechanism, the complexity of programming using this technology is still too high in the context of getting quick results.
Parallelization in the second category is mostly done using special compiler directives. The OpenMP standard [5, 6] was developed to provide basic concurrency constructs in Fortran and C / C ++ programs. The use of OpenMP gives limited control over threads compared to the more fundamental Pthreads standard [7]. However, OpenMP provides greater development speed due to ease of use.

The ability to quickly create effective parallel programs is a strong argument when choosing programming tools. The technology was originally designed so that the user could work with a single text for parallel and sequential programs. The usual compiler on the serial machine simply "does not notice" the OpenMP directives as they are located in comments (except for environment variables and special functions). An additional feature of OpenMP is the possibility of progressive, "incremental" parallelization of the program. Based on sequential code, it is possible to step by step add directives describing parallel constructions. With this approach, there is no need to immediately write a complete parallel program - its development is consistent. This simplifies both programming process and program debugging.

It is worth noting that programs on MPI can be executed on systems with shared memory no less efficiently than on OpenMP. From this point of view, MPI technology is more versatile. However, programming in OpenMP, firstly, is much more convenient than using MPI technology, and secondly, OpenMP technology is much more economical to spend RAM on systems with shared memory. For tasks that require a large amount of RAM, it often turns out that it is impossible to run so many processes on a computer system to load all the computing cores.

4. Parallelization in "MIRELA+"
The "MIRELA +" computing system is designed to solve problems in the deformed solid mechanics. To eliminate the effect of "false shift" in the complex used a moment scheme of finite elements, the essence of which is to approximate fields of displacement and deformation in the form of decomposition in a series of power functions of a special kind, that allows a fairly simple relationship between decomposition components and avoid unnecessary decomposition components of deformations

\[
u_i = \sum_{p,q,r=0}^{m,n,l} w_{pqr}^{(pqr)} (x_1)^p (x_2)^q (x_3)^r,
\]

![Figure 1. Distributed memory model.](image1)

![Figure 2. Shared memory model.](image2)
\[ \varepsilon = \sum_{s,t,g} \varepsilon_{ij}^{(stg)} \varphi^{(stg)}, \]
\[ \varepsilon = F \psi' w = F \psi' Au, \] (2)

where \( A \) is the transformation matrix that establishes the relationship between functions of the form of a finite element and the power basis functions.

Similarly for the function of changing the volume by decomposition by power functions allows considering the low compressibility of materials (such as elastomers) \[8\]
\[ \theta = \sum_{a,b,c,d=0} \varepsilon_{ij}^{(ab)} g^{ab} \]
\[ \theta = F \psi' w = F \psi' Au. \] (3)

For problems of structure statics, the variational equation of equilibrium of a linearly elastic body has the form
\[ \delta \int \left( \mu g^{ij} g^{ij} \varepsilon_{ij} + \frac{\lambda}{2} \theta^2 \right) dv + \int \rho P \delta u dv + \int F \delta u ds, \] (4)

where \( \mu, \lambda, \rho \) - mechanical characteristics of the material, \( P^i, F^i \) - body and surface forces respectively.

Given the approximation methods used the variation of the deformation energy has the form
\[ \delta W = \int \left( \mu g^{ij} g^{ij} \varepsilon_{ij} + \frac{\lambda}{2} \theta^2 \right) dv + \int \left( \varepsilon_{ij} \right) \lambda \delta \varepsilon_{ij} dv + \int \left( \xi \right) \lambda \delta \xi dv, \]
\[ H^{ij \theta} = \int \int \int 2 \mu g^{ij} g^{ij} \left( \varepsilon_{ij} \right) \lambda \delta \varepsilon_{ij} dv + \int \int \int \lambda \delta \varepsilon_{ij} dv, \]
\[ \delta W = \delta u^T K_i^r, u^r + \delta u^T K_o^{\theta r}, u_r, \] (5)

where \( K^{ir} = A^T F^T H^T F A \), \( K_o^{ir} = A^T F^T H^T F_{ij} A \) - stiffness matrices.

The procedure for forming the stiffness matrix of a finite element consists of the following steps:
- calculation of displacement coefficients and coefficients of matrix \( A \);
- numerical integration according to the Gaussian scheme.

The following procedures are performed for integration points:
- calculation of transition matrices from the global coordinate system to the local coordinate system and the inverse transformation matrix at integration points;
- calculation of partial derivatives for the formation of a matrix \( F \), which connects the displacement and deformation in a finite element.

The implementation of parallel processing in the study was performed using the OpenMP library. Joint processing of large data sets requires autonomy of operations on them for each cycle of formation of the stiffness matrix and calculation of the parameters of the stress-strain state.

Parallelism is achieved by dividing the iterations of the computational cycle into blocks, which are evenly and parallel distributed over the workflows. In OpenMP, the DO pre-loop directive is used to implement such an execution scheme \[9\]. The SCHEDULE (STATIC) option specifies the block-cyclic distribution of iterations in blocks of size determined by dividing the number of iterations by the number of threads. The next lines show the example of parallelized loop:

```c
!$OMP PARALLEL PRIVATE(S, SN)
!$OMP DO SCHEDULE(STATIC) ORDERED
```
DO 1 I=1,N
  <Parallel loop code>
  !$OMP ORDERED
  WRITE (*) RESULT
  !$OMP END ORDERED
  1 CONTINUE
  !$OMP END DO
  !$OMP END PARALLEL

The PRIVATE directive allows defining variables (arrays) that must be local to each thread. While entering the parallel region, a separate instance is created for each thread that has nothing to do with the original variable outside the parallel region. By default, the compiler defines variables that were initialized to the parallel section as shared (SHARED) for all threads.

Before the end of the loop, the procedure of outputting the results of the calculation of the finite element is performed, which must be performed in the same order as in the sequential version of the loop. The sequential access behavior is provided by the ORDERED directive.

To support parallelism the program uses a fork-join model which shown in Figure 3. It means execution forks parallel at designated points in the program to "join" (merge) at the next point and resume sequential execution. Sequential region of the OpenMP program is executed only by the main thread (master, №0). When entering a parallel region ($OMP PARALLEL), the master thread generates additional threads (Fork operation is performed). After generation, each thread receives its own unique number. All generated threads execute the same code corresponding to the parallel region. When exiting ($OMP END PARALLEL) from the parallel region, the main thread waits for the completion of other threads, and further execution of the program continues only master (Join operation is performed).

![Figure 3. Process of parallel program execution.](image-url)

As an example, for verification and testing, the calculation of a square plate clamped along the contour under the action of the bending load is presented. The computing complex "MIRELA+" with an application of the moment scheme of finite elements is used for calculation. Testing was performed on a device equipped with an Intel i7-9750H processor (6 cores and 12 threads) and 16 GB of RAM. Figure 4 shows speed comparison of execution for meshes with different count of finite elements.

![Figure 4. Speed comparison of execution for meshes with different count of finite elements.](image-url)
5. Conclusion
With the advent of multi-core processors, shared memory systems have become the most popular type of multiprocessor system, and OpenMP, in turn, provides one of the most convenient software development interfaces for such systems. The OpenMP version of the program can also be compiled like a sequential program using the compiler parameter "non-parallel" (OpenMP directives will be ignored by the compiler), which provides portability of code to different platforms. With the growing popularity of SMP computers, the importance of having efficient and cross-platform parallel codes is growing.

The paper analyzes the existing technical means of parallelization, strengths and weaknesses of the implementation of parallel solutions, using the OpenMP directives developed a parallel version of the calculation procedure to perform target calculations based on the "MIRELA+" computing system. Performance tests in the presented example showed satisfactory acceleration using parallel sections.

References
[1] Jarzebski P, Wisniewski K and Taylor R 2015 On parallelization of the loop over elements in FEAP Comput. Mech. 56 pp 77–86
[2] Amdahl G M 1967 Validity of the single-processor approach to achieving large scale computing capabilities In AFIPS Conference Proceedings 30 pp 483-485
[3] Datsyuk V N, Datsyuk O V, Bukatov A A and Vinogradova S A 2017 HPC Programming Guide http://rsuib.cc.rsu.ru/tutor/high_performance_computing/index.html [Online]
[4] Akhmetova D, Iakymchuk R, Ekeberg O and Laure E 2017 Performance Study of Multithreaded MPI and OpenMP Tasking in a Large Scientific Code Performance Study of Multithreaded MPI and OpenMP Tasking in a Large Scientific Code pp 756-765
[5] Pantalé O 2005 Parallelization of an object-oriented FEM dynamics code: influence of the strategies on the Speedup Adv. Eng. Softw. 36 pp 361–373
[6] OpenMP Architecture Review Board. Openmp 3.0 api specifications. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf 2021 [Online]
[7] Jia R and Sundèn B 2004 Parallelization of a multi-blocked CFD code via three strategies for fluid flow and heat transfer analysis Comput Fluids 33 pp 57–80
[8] Kirichevskiy V V, Doknyak B M, Kozub Yu G etc. 2005 Finite element method in obtained complex “MIRELA+” (Kyiv: Naukova dumka) 402
[9] Kopysov S, Kuzmin I, Nedozhogin, N, Novikov A, Rychkov V, Sagdeeva Yu and Tonkov L 2014 Parallel implementation of finite element algorithms on graphics accelerators in the software complex FEStudio Computer research and modeling 6 pp 79-97