Acta Polytechnica CTU Proceedings **26**:7–12, 2020
doi:10.14311/APP.2020.26.0007

available online at http://ojs.cvut.cz/ojs/index.php/app

ON TUNING THE DYNAMIC LOAD BALANCING FEM FRAMEWORK

Michal BOŠANSKÝ*, Bořek PATZÁK

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

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

Abstract. Developments in computer hardware are currently bringing new opportunities for numerical modelling. The current trend in technology is parallel processing making use of multiple processing units simultaneously to solve a given problem. This paper deals with exploring the parallel dynamic load balancing framework implemented in the finite element software. This framework is based on a domain decomposition paradigm for distributed memory model. The paper describes the improved technique to determine the actual processor weights related to performance of individual processing units. The load recovery consisting in mesh (re)partitioning is based on actual processor weights. The (re)partitioning process has to be performed during the simulation and whenever the load imbalance is significant. The performance of the proposed technique is tested on the benchmark problem and discussed.

Keywords: Distributed memory, domain decomposition, load balancing, message passing, processor.

1. Introduction

The main goal of parallel algorithms is partitioning the problem into a set of smaller tasks that can be solved simultaneously. The problem can be partitioned only once before the solution process, which represents static load balancing. When the problem is partitioned during the solution, it represents dynamic load balancing, see [1].

Ideal scalability is difficult to obtain because of the overhead cost of the parallel algorithm (synchronization and communication) and because some parts of the problem are essentially sequential. An effective load balancing process that (re)distributes the work to individual computing units is needed in many cases to obtain reasonable scalability.

One of the important characteristics of the parallel algorithm is its computational scalability, which is the most important goal in parallel computing. The scalable parallel algorithm achieves a reduction in execution time by using more processing units, ideally in a linear trend. Ideal scalability is difficult to obtain because of the overhead costs of the parallel algorithm (synchronization and communication). Almost every parallel algorithm has an overhead cost compared to a sequential version. Individual tasks cannot be executed concurrently without synchronizing and communicating with other tasks. Some parts of the algorithm are also essentially serial and can only be executed by a single thread. In addition to speedup, parallel computing allows large and complex problems to be solved that could not be solved on a single, well-equipped machine.

The Finite Element Method (FEM) has become a widely used tool for solving problems described by partial differential equations and been widely adopted by engineering and scientific communities as a reliable numerical tool. In mathematics, the FEM is a numerical method for finding a solution to boundary value problems for an ordinary differential equation (ODE) or a partial differential equation (PDE). In the FEM, differential equations are converted into an algebraic system of equations by using variational methods aided by decomposition of the problem domain into sub-domains called elements with an appropriate choice of interpolation functions. In many cases the resulting system of equations is nonlinear. In structural mechanics, the non-linearity can originate from nonlinear geometrical relationships (large deformations), from constitutive relationships, and from non-linear boundary condition (e.g. follower type of loading). Nonlinear problems are solved incrementally, typically using the $Newton - Raphson$ algorithm. This makes the nonlinear problem solution more demanding than linear problems. The problem is solved in a series of load or displacement increments in which the equilibrium state is iteratively searched.

The paper examines the solutions of large scale engineering problems by using parallel computation based on a parallel load-balancing framework. This framework can significantly reduce computational time by using available hardware more efficiently. This paper presents improved algorithm for determining processor weights in load-balancing FEM framework. When solving the real problem on the parallel computer, the workload of individual processors can change during the solution. The first source of imbalance originates from the character of the problem. For example, in nonlinear problems, the transition from initial elastic material response to nonlinear regime is often associated with increased computational cost and this

transition is often associated only to certain regions of overall domain. The second source of load imbalance includes external factors, which can change performance of individual processing nodes or communication network. This typically happens in non-dedicated cluster environments, where processing nodes and communication infrastructure is shared between users. In both cases, the gradual grow of imbalance can have significant effect on performance and on scalability.

The load-balancing process is driven not only by emerging applications but also by emerging parallel architectures. These parallel architectures span many scales. For example, clusters have become viable alternatives to tightly coupled parallel computers in small scale systems. On the medium scale, supercomputers are constructed as networks of shared memory multiprocessors (SMPs) and have complex and non-homogeneous interconnection topologies [2]. Finally, on the largest scale, grid technologies have enabled computations on widely distributed systems, combining distributed clusters and supercomputers into a single computational resource. These grids are a source of extreme computational power and high network heterogeneity. In order to effectively distribute data from any program on such systems, partitioning must account for heterogeneity in the solution environment. Load balancing requires evaluation of computing environment parameters (e.g. computing speed, memory and network availability) and determining how to apply this information in the load-balancing process (e.g. adjusting computational domain sizes, selecting partitioning algorithms).

The capabilities and performance of the load-balancing framework depends on many aspects, including problem at hand and hardware configuration and are illustrated on the solution of selected engineering problem. The advantages of implementing this approach are discussed in this paper.

2. Introduction and overall design of Load Balancing Framework

The models are often solved on complex domains leading to many degrees of freedom and often nonlinear effects have to be taken into account. Parallel algorithms should account for load imbalance between particular sub-domains. The imbalance can occur due to several reasons: (i) first load imbalance factor is part of the character of the solution process, as if switching from a linear to nonlinear response in some regions, (ii) second load imbalance factor is an external factor concerning resource relocation, typical in cases when a solution process is run in cluster environments where the available individual computing units are shared by other processes owned by the system or users and lead to changes in allocating processing power. The imbalance is detected in the solution process by monitoring processes during the run time of the process. The decision depends on the amount of load imbalance and the cost of load redistribution. Redistributing the work leads to serializing the problem data (elements, nodes, boundary conditions) into messages sent over the network and a receiving process based on unpacking, followed by a topology update reflecting the new work distribution.

The idea of parallelization strategy is based on the domain decomposition paradigm. In general, two dual partitioning techniques for the parallel distribution of finite element codes exist, node-cut and element-cut strategy [3]. In this paper, the node-cut strategy, dividing cut dividing the problem mesh into partitions is given in the node-cut technique. The node-cut strategy is assumed when cut dividing the problem domain runs through the nodes. Nodes on mutual partition boundaries are called shared nodes, and nodes inside individual partitions are called local nodes. The node-cut partitioning scheme can be interpreted as mesh decomposition using cuts passing through shared nodes of the mesh without crossing any element. The cut strategy ensures duplication of the shared nodes. Each processing node is responsible for assembling its contribution to the global system matrix by summing the contributions from individual elements. In order to minimize communication, it is natural that each processing node will assemble and maintain in its local memory the block of global stiffness matrix rows corresponding to unknowns on its partition. This is uniquely defined for local nodes, which are exclusively shared by local elements on individual partitions. For shared nodes, which are shared by elements from multiple partitions, ownership must be defined. In this work, the partition with the lowest rank sharing the shared node is the owner of the shared node and thus responsible for maintaining the corresponding row entry of the global stiffness matrix. It is clear that for shared nodes, the contributions to the corresponding row have to be received from partitions sharing a particular node. The assembly process requires global, unique numbering of equations to be established across the computing nodes in addition to local numbering on individual partitions [4].

Partitioning based on the load balancing process can be affected by many factors. Ideally, no processor during the parallel solving process should be waiting for another processor to complete the solution. In the typical FEM code, the most of the work is proportional to individual elements. The total computational work is determined as the sum of the contribution of individual elements. The computational work of individual element is expressed relatively to computational work of reference element. The load balancing process is based on distributing the overall computational work to a group of available processors with respect to their relative performance. Another significant factor concerns minimizing and reducing communication between computational partitions. The assembly of equilibrium equations for shared nodes requires to

sum up contributions from local as well as remote partitions. The cost of accessing remote memory is much higher than the cost of accessing the local memory. It is therefore important to reduce communication between computational partitions.

The present contribution examines the design of an object-oriented framework for dynamic load balancing implemented in OOFEM code [5].

3. Design of micro benchmarks tests

Overall processor performance depends on many factors, notably on its frequency, the performance of memory subsystem, and type of code executed. The performance of two CPUs can be different for integer and floating point operations, on some SMPs, some resources are shared between processing cores, etc. The adopted approach to evaluate the individual processor performances is based on a set of so-called micro benchmark tests, that evaluate processor performance for different typical tasks and computing overall performance as the weighted average of performances of individual micro-benchmarks. The processor weights are updated each time, when rebalancing should occur. It is therefore important, that the individual benchmarks have very low time demands and on the other hand provide reliable estimation of individual processors.

The first micro benchmark test proposed is a function to compute a numerical integral. In the test, the integral of Cosine function is evaluated using numerical integration using Newton-Cortes formula. The parameter n_i defines the number of integration points used to divide the area. Therefore, with increasing parameter n_i (increasing number of integration points), we have a solution process closely approximating the integral value. This test measures performance in floating point operations.

The next type of micro benchmark problem was based on solving a linear system of equations. A FEM model of the cantilever beam, divided into n_{eq} elements is used to define a linear system. The right hand side of the force vector in our case is a simple vector with its first member set to 1, while all the other elements of the vector are set to zero. The number of equations of the linear system directly depends on parameter n_{eq} , which also represents the dimension of the stiffness matrix and right hand side of the force vector.

Another micro benchmark problem called Whetstones is a test that attempts to measure the speed and efficiency of a computer performing floating-point operations. The Whetstone benchmark was the first designed for benchmarking [6]. This benchmark is very simple, comprising several sub-tests with active loops executed via procedure calls. This module represents a mix of operations typically performed in scientific applications. The test involves integer arithmetic, floating point arithmetic, "if" statements, calls and so on. At the end of this benchmark, a statement with the results is printed. Weights were attached to the different modules (realized as loop bounds for loops around the individual modules statements). The weight distribution of Whetstone instructions for the benchmark matched the distribution seen in the program sample. The execution frequency of each module was proportional to the input value of such that the scaling factors for $n_l = 10$ gave the modules a total weight corresponding to one million Whetstone instructions for more details see [7].

The final load balancing processor weighs are obtained as a weighted average of individual weights from micro bench-mark tests. The final weights are assembled in the ratio of 15% micro benchmark problem based on a function to compute numerical integral, 80% benchmark problem based on the solution of a linear system of equations and 5% benchmark problem based on measuring the speed and efficiency of a computer performing floating-point operations. The time requirements of individual micro benchmark problems were controlled using parameter n as follows: benchmark to compute numerical integral with parameter $n_i = 10^6$, solution of a linear system of equations with parameter $n_{eq} = 10^3$ and computer performing floating-point operations with parameter $n_l = 6 * 10^4$.

4. Dynamic load balancing framework

The software design process is an important part of dynamic load-balancing research. The design of a dynamic load balancing framework based on a problem partitioned during the solution process is discussed in this section. Dynamic load balancing is typically used for problems involving multiple solution or time steps, where work redistribution is performed to reflect potential work imbalance. Partitioning the problem during solution is based on monitoring computation time of individual processors during the solution step. Once the solution step is finished, the load balance process is activated by evaluating the measured imbalance. If the imbalance is larger than the user defined tolerance, the (re)partitioning is performed, taking into account the actual performance of processing nodes and processing cost of individual elements. As already noted, when solving a real problem on a parallel computer, the load balance can change during the solution.

The only way of reflecting the growing imbalance is to adaptively redistribute work between processing units in order to restore load balance and thereby ensure optimal use of resources. The monitoring and evaluation of imbalance is the task of *Load Balance Monitor*. This monitor can detect load imbalance by monitoring the time required to perform allocated work on individual processing units. The differences in processing time indicate an imbalance. After imbalance is detected, the decision of whether to restore the load balance or continue is made. This can be

a complex task, as load redistribution may in fact be a very complex problem with non-negligible time requirements. The cost of load (re)balancing may be higher than the cost of continuing with a slight load-imbalance. All these aspects have to be considered and are, unfortunately, specific to the problem and implementation.

When the imbalance is detected, the work (expressed in terms of individual elements) has to be redistributed to reflect the performance of individual processing nodes. Individual elements have assigned weights that reflect the relative computational costs. These depend on element type, material state (elastic, plastic, etc.). These weights are typically obtained in advance from benchmark measurements. The processing power of individual processing units also has to be determined. The load (re)balancing process in this chapter is based on (re)distributing the computational work proportionally to the performance of individual processing units. In principle, additional factors, including available communication bandwidth between individual processing units or available memory, can be taken into account. Any load balancing should not only distribute the work according to processing powers but also attempt to minimize the communication cost between individual processing units. In the FEM, this means the cuts between partitions (number of shared nodes) should be minimized. Failing to meet the secondary criteria can significantly impact overall performance, as the cost of communication (in terms of the time required) is much higher than the cost of computation. Load (re)balancing should also attempt to minimize the reallocation of elements as much as possible in order to minimize communication costs.

The ParMETIS library [8] was used in this work, however, the ParMETIS implementation is not restricted to one specific partitioning library. A parallel partitioner can take advantage of the increased memory capacity of parallel machines (distributed memory model) and improve overall performance. A general load balancing algorithm is responsible for dynamic repartitioning using the processor weights provided by the load balance monitor during the micro benchmark problem and should provide the new partitions with numbers for all local elements on each partition. After the updated element partition assignment is determined, the distribution and classification of nodes also have to be determined. Each node is classified as either a local node that remains local on an existing partition or a shared node that is assigned to a remote partition. Node classification can be determined from element partitioning. The dynamic load balancing framework implemented in OOFEM is using the weights based on measuring of individual time involved in the computation. These times represent how long each CPU work on the predefined number of solution steps. The repartitioning is based on default weights set as the ratio of total solution time to individual computational thread time.

5. Example using a dynamic load balancing framework

The above-mentioned strategy is illustrated on nonlinear 3D finite element analysis of an anchor pullout test. The FE model involved nonlinear nonlocal anisotropic damage model to describe fracture process and consists of 1456 nodes and 16772 tetrahedral elements, which was subsequently refined in 20 steps into a final mesh with 125400 elements and 22441 nodes. The number of solution steps was set to 80 steps. Due to the character of the solution, the number of required equilibrium iterations was increasing with progressing solution steps. The number of required equlibrium iterations gradually increased form 9 (first solution step) to 263 (solution step number 80). The existing and proposed processor weight evaluation methods were tested on a workstation (running Ubuntu 16.04 OS) with an Intel ® Core ™ i7-4790 CPU @ 3.60 GHz with four cores consisting of two logical processors per core connected to a workstation with an Intel® Core™ i3-2370M CPU @ 2.40 GHz with two cores consisting of two logical cores. Each workstation could simultaneously run a maximum of eight threads and four threads on their CPUs and had 15 GB and 8 GB of system memory, respectively.

The iterative linear equation solver from the PETSc library was used with a block Jacobi preconditioner. The dynamic load balancing framework parameters were set up. Load (re)balancing was activated after each fifth solution step. The relative wall clock imbalance parameter represents the relative imbalance between wall clock solution time of individual computational threads. When greater than the provided threshold the rebalancing procedure is activated. The additional parameter called absolute wall clock imbalance parameter allows to triggered rebalancing procedure when an achieved absolute imbalance between solution times of individual processing threads is greater than the threshold. The absolute wall clock imbalance parameter was set to 10.0 $[s]$. The minimum absolute wall clock imbalance parameter allows setting minimum absolute wall clock imbalance threshold for performing rebalancing and was set to 0.5 $[s]$.

The obtained results are presented in Tables 1, 2 and 3. In the presented tables, the number of processing units used in the parallel computations is marked as "NP". The results obtained using the existing dynamic load balancing framework without the micro benchmark tests to set up processor weights were compared to the results of the computation process without load balancing. These load balancing strategies were finally compared to the presented method based on evaluating the microbenchmark tests. Tables 1 and 2 show examples of computation done by processors with hyper-threading [9] technology disabled (Intel i3 max. 2 threads, Intel i7 max. 4 threads). The computational process with hyper-threading technology enabled on the i3 processor is shown in Table 3.

NP	No DLB [s]	DLB [s]	DLB W. [s]
2	1659	1473	1411
3	1408	1399	1230
4	1372	1215	1181
5	1277	1179	1131

TABLE 1. Execution times comparing solutions with estimated processing weights (DLB W.), uniform processing weights (DLB) and without a load balancing (No DLB) framework (Intel i3 - 1 computing unit + Intel i7 - 4 computing units).

NP	No DLB [s]	DLB [s]	DLB W. [s]
4	1524	1479	1373
5	1365	1314	1154
6	1330	1302	1140

TABLE 2. Execution times comparing solutions with estimated processing weights (DLB W.), uniform processing weights (DLB) and without a load balancing (No DLB) framework (Intel i3 - 2 computing units + Intel i7 - 4 computing units).

NP	No DLB [s]	DLB [s]	DLB W. [s]
2	1847	1662	1373
3	1606	1339	1303
4	1610	1340	1435
5	1853	1642	1578

TABLE 3. Execution times comparing solutions with estimated processing weights (DLB W.), uniform processing weights (DLB) and without a load balancing (No DLB) framework (Intel i3 - 4 computing units (HT) + Intel i7 - 4 computing units).

The results confirm that better performance is obtained when appropriate weights are used. In the case with hyper-threading technology enabled, the scalability trend is not ideal in the solution based on six and eight computational threads and is worse, for example, than the solution based on two and four computational threads. The significant decrease in performance can be attributed to the hyper-threading technology of Intel processors, which assembles and shares some CPU resources between hyper-threaded cores and only takes place with 4 threads (note that the workstation with Intel® Core™ i3 had two physical hyper-threaded cores). However, by using dynamic load balancing and appropriate weights (processor performance parameter), we can achieve better performance than in the solving process without load balancing or with the existing dynamic load balancing framework not using micro benchmark tests.

6. Conclusions

The contribution based on an improved methodology to determine processing weights parameters as a part of the load balancing framework was presented in this paper. The parallelization strategy was based on the dynamic load balancing process using weights that represented the performance of computational units. The performance of the upgraded dynamic load balancing process (processor weights parameters) was compared to the previously implemented dynamic load balancing process. In this work a nonlinear benchmark problem was considered to evaluate the performance of dynamic load balancing framework. The results showed differences in the performance of the upgraded and previously implemented dynamic load balancing process for the considered benchmark. The upgraded dynamic load balancing process had better performance than the previously implemented dynamic load balancing process. Future studies will investigate other testing examples (simple linear equation system solutions) to more precisely assess variables as a representation of computational unit performance. It is clearly a benefit to using the appropriate measures of performance of individual computational units performance of computational units in the load balancing process as a necessary part of any Finite Element parallel code.

The performance of the described load balancing framework could potentially be significantly improved by considering communication speeds between processing nodes by using parameters that represent communication speeds through different, heterogeneous networks. The load balancing framework is based on parallel computation using a distributed memory model. This model uses a message passing interface. For example, many distributed systems are now being constructed using a variety of different communication networks, such as Ethernet and Asynchronous Transfer Mode (ATM). In addition to this hardware heterogeneity, there is a heterogeneity in the types of messages produced by parallel programs. Short synchronization messages require low latency. Conversely, large data messages require high-bandwidth, though they can tolerate high start-up latency. The different types of networks have different performance characteristics, while the different types of communication messages may have different communication requirements. The performance of parallel computations based on a distributed memory model is typically limited by communication overhead. High-performance networks and the use of multiple heterogeneous networks can help reduce this overhead. Further research could focus on designing the load balancing framework's input parameters, which represent different types of networks providing a data path between the same pair of network nodes. These load balancing parameters (network weight parameters) may allow a solution process to maximize efficient use of different types of networks rather than passively accept the given features of a single network.

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 modelling in mechanics of structures and materials. The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics".

References

[1] K. Schloegel, G. Karypis, V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience* **14**(3):219–240, 2002. doi:10.1002/cpe.605.

[2] K. D. Devine, E. G. Boman, R. T. Heaphy, et al. New challenges in dynamic load balancing. *Applied Numerical Mathematics* **52**(2-3):133–152, 2005. doi:10.1016/j.apnum.2004.08.028.

[3] P. Krysl, Z. Bittnar. Parallel explicit finite element solid dynamics with domain decomposition and message passing; deal programming scalability. *Comput Struct* **79**(3):45–60, 2001.

[4] M. Bošanský, B. Patzák. Parallel Approach To Solve Of The Direct Solution Of Large Sparse Systems Of Linear Equations. *Acta Polytechnica CTU Proceedings* **13**:16, 2017. doi:10.14311/app.2017.13.0016.

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

[6] H. J. Curnow. *A synthetic benchmark*, vol. 19. Oxford University Press (OUP), 1976. doi:10.1093/comjnl/19.1.43.

[7] S. Harbaugh, J. A. Forakis. Timing studies using a synthetic whetstone benchmark. *ACM SIGAda Ada Letters* **IV**(2):23–34, 1984. doi:10.1145/998395.998396.

[8] V. K. G. Karypis. *ParMETIS: Parallel graph partitioning and sparse matrix ordering library.* Department of Computer Science, University of Minnesota,, 1997.

[9] D. H. e. a. D. Marr, F. Binns. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal* **6**(1), 2002.