Applying Dynamic Load Balancing Techniques to the Parallel *p*-Version FEM using Nested Dissection

Jovana Knežević, Ralf-Peter Mundani

Technische Universität München, Computation in Engineering, 80333 München, Deutschland, E-mail: jovana.knezevic.83@gmail.com, Telefon: +49(0)89/28925057

Abstract

Typically, parallelisation strategies based on hierarchical structures suffer from limited speed-up and parallel efficiency values. Hence, sophisticated load balancing techniques become inevitable in order to exploit the underlying computing power. In this paper, we describe a modified master-slave concept for distributed task queues applied on problems from the field of computational structure dynamics. Therefore, the elements of a *p*-version finite element discretisation are hierarchical organised via an octree structure to be solved using the nested dissection method.

Keywords

Parallelisation, dynamic load balancing, numerical simulation, nested dissection, *p*-FEM

1 Introduction

Numerical simulation of structure dynamics is a challenging task which entails a huge computational effort and, thus, typically arises the necessity of parallelisation. Classical domain decomposition methods are often not sufficient to exploit the full computation power of the underlying hardware. Hence, sophisticated approaches are necessary in order to obtain good speed-up and efficiency values. Otherwise, sophisticated data structures also play a dominant role to organise the computational tasks and to provide easy access to the underlying data. Here, octrees are advantageous due to their spatial substructuring and their hierarchical organisation of the computational domain.

In our approach, we use octrees for the hierarchical organisation of elements stemming from a *p*-version finite element discretisation in order to apply a nested dissection method for the solution of the resulting equation system. As on the one hand, the nested dissection method profits from the octree organisation, on the other hand, tree structures

do not provide the best scalability properties for the parallelisation in this case and, thus, make it difficult to harvest the benefits of distributed computing. Therefore, load balancing techniques are inevitable to achieve proper speed-up and efficiency values. Based on a modified master-slave concept, we will describe a technique that uses distributed task queues to cover the dependencies within nested dissection and that allows to foster a maximum degree of parallelism.

The remainder of this paper is as follows. After a short overview of the numerical treatment of the problem in section 2, we will discuss the load balancing strategy and its communication pattern among all processes in section 3. Section 3 also contains first results of our approach, before in the last section we finalise with a short conclusion.

2 Numerical Treatment

Within our structure simulation we use *p*-version finite elements which are advantageous as higher accuracy can be obtained by increasing the polynomial degree *p* of the Ansatz functions without changing the underlying mesh, too. Nevertheless, the resulting equation systems have very poor condition numbers and, thus, any sophisticated iterative solvers such as Conjugate Gradients (CG) or multigrid methods cannot outperform – and typically even underlie – their direct counterparts such as Gauss and relatives. Unfortunately, Gaussian elimination is quite expensive and also has a limited potential regarding parallelisation. Hence, different approaches for solving the equation system are necessary.

2.1 Nested Dissection

From the field of domain decomposition, non-overlapping substructuring methods based on the Schur complement are known since the early beginnings. J.A. George was the first to apply these methods on finite element problems [1] by recursively splitting a domain Ω in subdomains Ω_i and eliminating all unknowns in the interior of Ω_i. The remaining equation system – the Schur complement system – is then associated with the interfaces of Ω_i and can be solved. This already shows the potential of the nested dissection (ND) method as all Ω_i are independent from each other and, thus, can be easily processed in parallel.

To eliminate the unknowns in the interior of Ω_i, the matrix K of the corresponding equation system $K \cdot u = d$ is therefore divided into parts belonging to the interior (I) and to the interface (O). Written in block form, the equation system looks like

$$\begin{pmatrix} K_{II} & K_{IO} \\ K_{OI} & K_{OO} \end{pmatrix} \cdot \begin{pmatrix} u_I \\ u_O \end{pmatrix} = \begin{pmatrix} d_I \\ d_O \end{pmatrix} . \qquad [1]$$

Expanding this system in two equations and substituting the first one for u_I in the second one leads to the Schur complement system

$$\left(K_{OO} - K_{OI} \cdot K_{II}^{-1} \cdot K_{IO}\right) \cdot u_O = d_O - K_{OI} \cdot K_{II}^{-1} \cdot d_I \qquad [2]$$

where the influence of the unknowns in the interior has been eliminated[1]. Assembling all Schur complements and right-hand sides provides a new equation system that can be solved. In case of more than one recursion step for the substructuring all nested subdomains Ω_j^i of Ω_i are first successively processed in the aforementioned way before ready to be solved.

2.2 Applying Nested Dissection to the *p*-Version

The finite element discretisation already provides a sufficient substructuring, hence, ND can be directly applied on these elements. In order to retrieve a hierarchical organisation necessary to identify interfaces among the subdomains, the single elements are assigned to an octree where each leaf node stores at most one element, i. e. the corresponding element stiffness matrix and right-hand side, or stays empty. Unknowns – referred to as degrees of freedom (DOF) – are stored to those nodes from which all elements that contain a certain DOF in the interface can be reached. It's obvious, that the higher, i. e. the closer to the root node, a DOF is stored the later it will be eliminated and, thus, the more computational effort has to be invested [2]. A nice property of this approach can be observed when the input data (material parameters, geometry …) is modified as only those parts have to be re-computed. From all other parts, the pre-computed Schur complements can be re-used which reduces the total run time on the average at least by an

[1] This can be achieved without directly computing K_{II}^{-1} by a partial Gaussian elimination.

order of magnitude [3]. Even the computations are typically still far beyond real time, ND applied to the *p*-version provides an important ingredient for interactive applications such as computational steering [4].

3 Parallelisation

As one node of the tree always depends on the results, i. e. the Schur complements, of its child nodes, there exists a "natural" order ≤ among the nodes[2] which defines the order of processing. Nevertheless, nodes of different branches do not depend on each other, thus, they can be processed in parallel. Due to the octree structure, the amount of nodes on each level decreases by a factor of 8 and so do speed-up and parallel efficiency. A similar problem stated by Minsky et al. [5] on the parallel summation of $2N$ numbers on N processors shows that speed-up and efficiency of parallel programs based on hierarchical structures are limited. Hence, load balancing strategies are inevitable to exploit the computing power of any underlying architecture such as clusters or supercomputers. Therefore, a modified master-slave strategy is used, where several masters serve the requests of the worker processes. To prevent the typical bottleneck in serving requests when using only one master process, a concept with one main master and several sub masters – so-called traders – has been developed. Here, the main master is the direct interface to the workers while the traders handle all the necessary data exchange.

3.1 Initialisation

In the beginning, the main master analyses the tree structure according to the size of the stored element stiffness matrices and, thus, calculates the total number of operations needed for a Gaussian elimination to be performed for each node and sub-tree, resp. This number is used as weight, i. e. as rough estimation of the amount of work to be done, in order to achieve a fair distribution of tasks among all traders instead of just horizontally cutting the tree at some level. The main master immediately stores the root node in its own queue of tasks. Afterwards, it visits the nodes with the forward edge pointing to the root and compares the calculated number of operations for each of them to the limit. Here, the limit is set as the average number of operations that should be performed by a single trader. If the maximum number of operations for all nodes visited so far is not larger than this limit, the main master is only responsible for the root node. Otherwise, it

[2] For a set T of tasks, a partial order ≤ can be declared by $t_1 \leq t_2$ for $t_1, t_2 \in T$ and ≤ representing a reflexive, antisymmetric, transitive relation.

stores the node's identifier which corresponds to the one with the maximum weight in its own task queue and examines the next hierarchy level for the sub-tree having that particular node as the local root. In the latter case, all nodes from the previous step (except the one referred to the main master) as well as the additional ones, i. e. the sons of the mentioned node, will be considered for the next step.

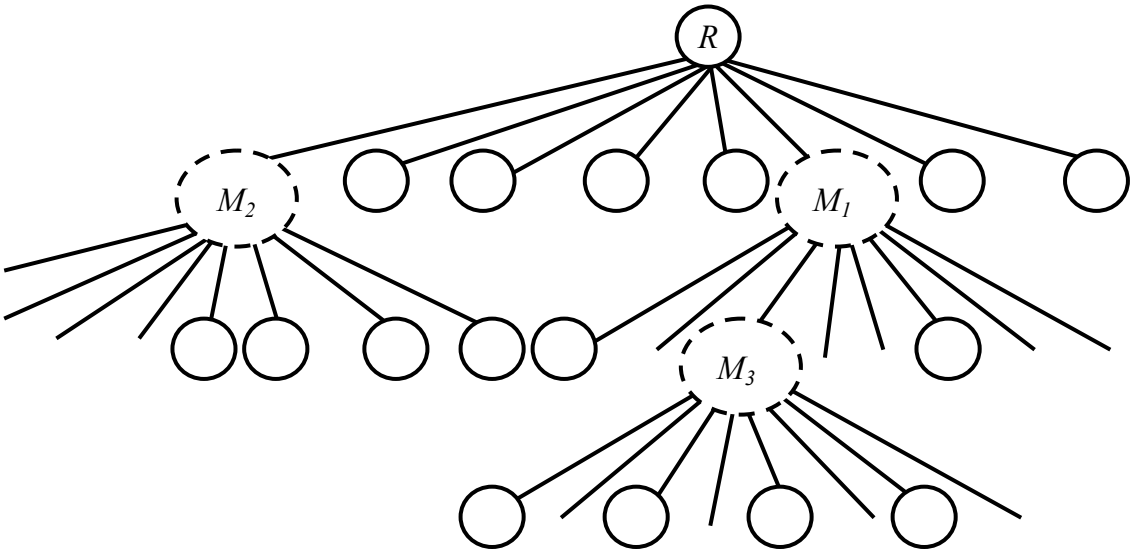

Fig. 1: Starting from the tree root (*R*) several nodes (M_1, M_2, and M_3) with weights exceeding the limit are detected in that order and further processed until all nodes and sub-trees, resp., are below the limit and, thus, ready to be assigned to the different traders

This step is repeated – visiting nodes in breadth-first fashion, assigning the one with a maximum weight to the main master and taking its sons instead – until all node weights are estimated not to exceed the proposed limit. At this point, all nodes exceeding the limit have been assigned to the main master and all remaining nodes (i. e. local roots of sub-trees not exceeding the limit) are ready for being assigned to the traders. Figure 1 shows a sample tree which has been processed by the main master until all nodes and corresponding sub-trees have weights below the limit.

Once finished with processing the tree according to the algorithm stated above, the main master sorts the remaining nodes in descending order regarding their weights, passing through them from left to right and assigning the corresponding tasks to individual traders, taking care that none of them gets more than the limit allows. If in the end some nodes, due to their size, cannot be assigned to any trader without exceeding the

limit, they will be assigned to the traders with minimum work load. Figure 2 shows the achieved load distribution for 4171 tasks assigned to 4, 8, and 16 traders, resp. Even the single tasks were of both small and large sizes, the final distribution is quite fair and heavy imbalances are not to be observed.

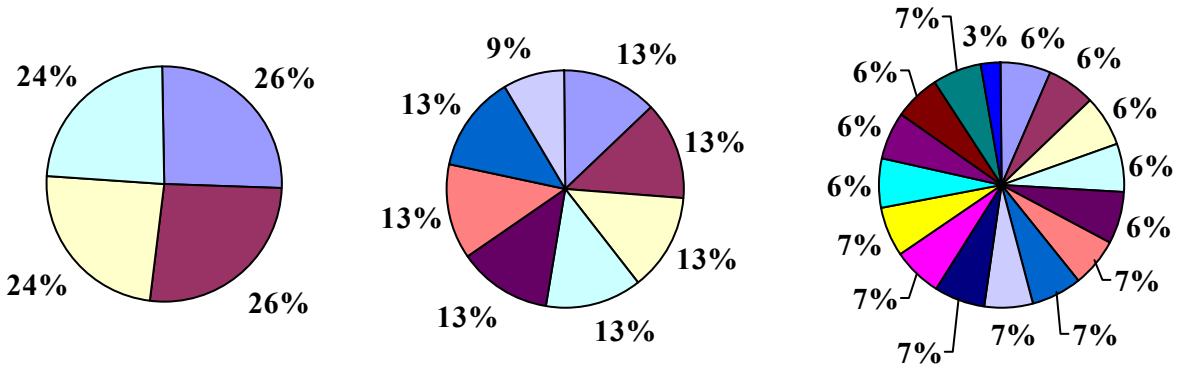

Fig. 2: Load distribution for 4171 tasks, i. e. elements, of different sizes assigned to 4, 8, and 16 traders (from left to right) by the main master

Up to now, balancing based purely on processor loading has been discussed. Since in parallel systems it is also desired to take into consideration the locality of the data which might be required by the different processes, any load balancing strategy depending on the workload only and, thus, neglecting all kind of neighbourhood information of the data obviously seems to incorporate disadvantages. This seems even more obvious, if we take a simple example of four jobs j_1, j_2, j_3, j_4 with corresponding weights w_1, w_2 much smaller than w_3, w_4 to be run on two processors P_1 and P_2. The first two jobs j_1, j_2 should share common input blocks of the data as do the third and the fourth j_3, j_4. If initially j_1 is located on P_1 and j_3 is located on P_2, it may be optimal if j_2 is run on P_1 and j_4 is run on P_2, even if j_1 and j_2 are not time-consuming computations relative to j_3 and j_4. Our load balancing scheme, for instance, would allocate j_1 and j_3 to P_1 and j_2 and j_4 to P_2 in order to achieve a fair load distribution. However, due to the shared input blocks it may be far more advantageous to assign j_1 and j_2 to P_1 and j_3 and j_4 to P_2. Even though P_1 will be idle while P_2 is still busy, by allocating the tasks with similar data requirements at the same processor, we reduce the amount of necessary communication (in a message passing system) and, thus, lower the total execution time.

On the other hand, this is, evidently, dependent on the size of the data blocks and the time needed to move these blocks, so having conflicts of this kind in mind, one should rather make a decision based on his own, specific, data and application requirements. In our case, by visiting nodes in breadth-first manner and traversing more into depth only for those nodes whose weight is over the limit, we at least assure to minimise the amount of sub-trees and, thus, to disperse the data more than necessary. Nevertheless, a possible improvement of the algorithm could be to check the locality of the data and to make decisions dependent on both cost of the communication and operations required.

Finally, it is the main master's responsibility to distribute the identifiers of tasks (sub-trees) to the traders. Each trader then analyses all edges of its sub-trees in order to identify dependencies, i. e. the order of processing among its tree nodes. All identifiers of tasks are stored in a queue, together with the information about the dependencies between the single tasks. It is obvious that tasks related to leaf nodes are free of dependencies as the nested dissection approach proceeds bottom-up and, hence, parent nodes always depend on the elimination results of their children. When all sub masters generated their local queue of tasks the initialisation stage has been finished.

3.2 Communication

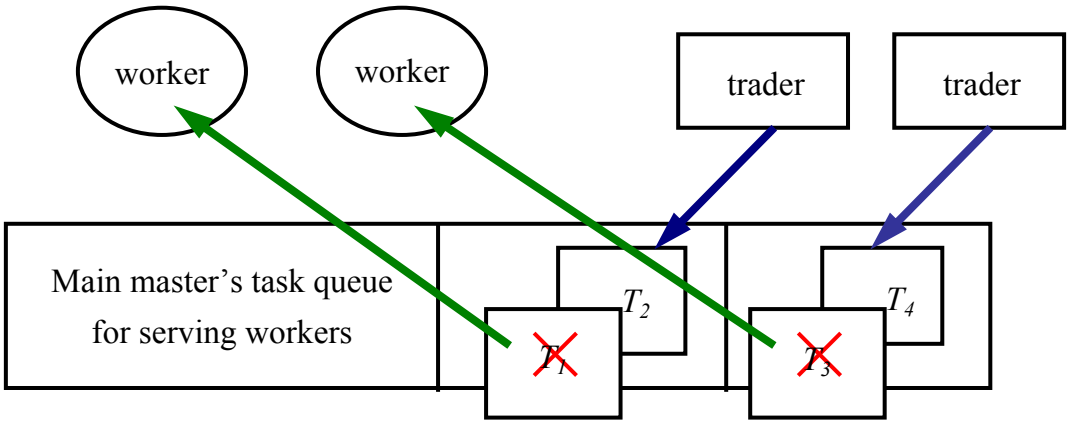

Fig. 3: Main master advertising different tasks T_i from the traders and picking randomly some of these tasks to serve request issued by the workers

Now, workers can request tasks from the main master. In the same time, traders give away one arbitrary task (free of dependencies) still to be processed from their queue, thus

these tasks become available to the main master for being advertised to the workers. Once a worker requests a task, the main master selects randomly one task and *virtually* couples it to a worker (Figure 3). To avoid the main master becoming a bottleneck due to severe data exchange, a worker process is immediately advised to the respective trader (which owns the task) in order to retrieve all subsequent data necessary for the computation. As soon as one task has been assigned to a worker, the trader gives a new task – if available – to the main master for advertising.

When finished with the computation, the worker sends back the results to the trader it received the task from, thus, the trader can check if any of its dependencies can be solved and new tasks are ready to be offered. Furthermore, the trader forwards this information about resolved dependencies to the main master, thus, it can check its own dependencies as well.

Due to the updates of dependencies in the main master's queue, at some point, there are tasks available from the main master itself. Since the main master is not likely to have all the data needed for this calculation – moreover, the data is stored in the memory of the traders – it is obliged to find out to which trader(s) the data belongs and to advise the worker process to request the data from the appropriate one(s). When the computation is finished, a worker sends the corresponding task identifier to the main master, hence, it can check for updates within its own queue of tasks.

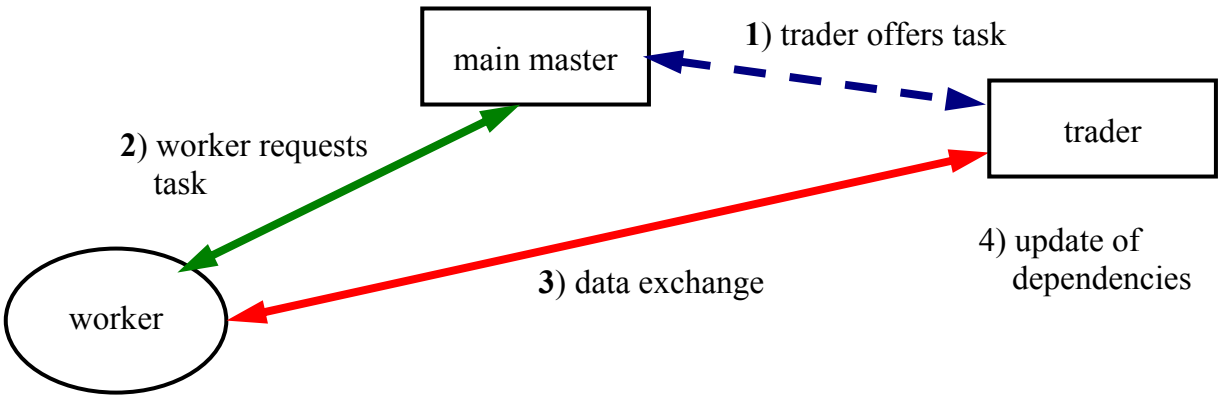

Fig. 4: Communication pattern

Figure 4 shows the main part of the communication pattern. Workers are always initiating the communication stage by sending a message to the main master or the traders. This message can either be a request for a task to be processed, a request for a

task's subsequent data (stiffness matrix, loading vector, Schur complements …) or the result of a computation. On the other side, traders are always catching up the communication stage by receiving a message from the main master or some worker process. The main master is the crucial coordinator of all other processes – its initial operation is also receiving and it only switches to "active" sending in case the entire problem has been solved and both traders and workers have to shut down.

4 Conclusion

In this paper, we have shown a modified master-slave concept for load balancing that tackles the problems related to tree parallelisation which on the other hand is necessary in order to apply a nested dissection method to solve structural problems stemming from a *p*-version finite element discretisation. As this is work in progress, first results sound very promising. Nevertheless, further research is necessary to study the benefits of our approach concerning speed-up and efficiency values.

Acknowledgements

This work has been financially supported by the International Graduate School of Science and Engineering (IGSSE) at Technische Universität München.

References

[1] J.A. George. *Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis,* ***10****:345—363. 1973.*

[2] R.-P. Mundani, H.-J. Bungartz, E. Rank, A. Niggl, and R. Romberg. *Extending the p-version of finite elements by an octree-based hierarchy. Proc. of the 16th Int. Conf. on Domain Decomposition Methods, LNCSE Vol. 55, Springer, 699—706. 2006.*

[3] R.-P. Mundani. *Hierarchische Geometriemodelle zur Einbettung verteilter Simulationsaufgaben. Shaker Verlag. 2006.*

[4] T. Trummer. *Implementierung eines hochperformanten Lösungskonzepts einer Simulations- und Steering-Umgebung im Bereich der Medizintechnik. Bachelor's Thesis, Institut für Informatik, Technische Universität München. 2008.*

[5] M. Minsky, S. Papert. *On some associative, parallel and analog computations. Associative Information Technologies, Elsevier North Holland. 1971.*