Acta Polytechnica CTU Proceedings **13**:16–19, 2017
doi:10.14311/APP.2017.13.0016

PARALLEL APPROACH TO SOLVE OF THE DIRECT SOLUTION OF LARGE SPARSE SYSTEMS OF LINEAR EQUATIONS

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 paper deals with parallel approach for the numerical solution of large, sparse, non-symmetric systems of linear equations, that can be part of any finite element software. In this contribution, the differences between the sequential and parallel solution are highlighted and the approach to efficiently interface with distributed memory version of SuperLU solver is described.

Keywords: linear equation solver, domain, parallelization, threads, distributed memory, communication.

1. Introduction

Many engineering problems lead to large-scale problems. The solution of these problems is being limited by available resources. In many cases, the parallel and distributed computing paradigms can be successfully used to enable solution of large and complex problems, which are not possible to solve on single machine with one processing unit [1]. The traditional serial computers permit to run simulation codes only sequentially and often have limited memory. Parallelization can significantly reduce computational time by more efficient use of modern hardware. Any parallel algorithm is based on idea of the partitioning the work into the set of smaller task which can be solved concurrently by the simultaneous use of multiple computing resources.

Parallel computers can be classified, for example, by type of memory architecture. The shared, distributed, and hybrid memory systems [2] 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 logical memory. The global memory significantly facilitates the design of parallel program, but the memory bus performance is limiting factor for scalability with increasing number of processing units. In distributed memory systems, the memory is physically distributed to individual processing units and there is no global address space. When a processor needs to access data on another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. The cost of communication compared to local memory access can be very high, on the other hand, the advantage is that the overall memory is scalable with increasing number of processors. Hybrid systems combine the features of shared and distributed memory system, providing global, shared memory for reasonably small number of processing units that are combined into the distributed memory system.

The design of parallel algorithms requires the partitioning of the problem into a set of tasks, that can be processed concurrently. The partitioning of the problem can be either fixed (static load balancing) or can change during the solution (dynamic load balancing) [3]. The latter option is often necessary to achieve a good load balancing of individual processors and consequently also reasonable scalability. The scalability of computation is often regarded as one of the most important goals of parallel computing. The parallel algorithm is considered as scalable, when by using increasing number of processing threads, the execution time is monotonically decreasing, ideally in a linear trend. However, when individual computing threads are mutually dependent, the ideal scalability is difficult to obtain due to the overhead of parallel algorithm (necessary synchronization and communication between threads that is not present in serial code) and due to the fact that some parts of the overall algorithm are essentially serial. In addition to the achieved speedup, the parallel computing allows to solve large, complex problems that do not fit into a single, even well equipped machine.

The SuperLU library provides parallel implementation of the direct solver [4]. It provides serial, multithreaded (shared-memory), and distributed memory version. The implementation and performance of the multithreaded version has been reported by the authors in [5, 6]. In this papers, the focus is on distributed version of SuperLU based on message passing interface (MPI) [7].

Numerical modeling using high performance computers brings in new opportunities. Many engineering problems lead to solution of sparse linear system of equations. This is also the case of the Finite Element Method (FEM) which is used here as an representative example. In FEM the differential equations are converted to the algebraic system of equations by using variational methods with the help of decomposition of the problem domain into sub-domains called elements

$$K = \begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline & 8 & 6 & 0 & 2 & 11 & 1 \\ & 0 & 0 & 0 & 3 & 5 & 2 \\ & 9 & 0 & 2 & 6 & 0 & 3 \\ & 4 & 0 & 0 & 0 & 13 & 4 \\ & 7 & 1 & 0 & 0 & 3 & 5 \end{array}$$

K_nzval	8	6	2	11	3	5	9	2	6	4	13	7	1	3
K_colind	1	2	4	5	4	5	1	3	4	1	5	1	2	3

K_row_ptr	1	5	7	10	12

Figure 1. The compressed row format of matrix schema.

and smart choice of interpolation functions. For linear elasticity problem, considered here, the resulting algebraic equations, represent the discrete equilibrium equations at nodes of the computational mesh

$$\mathbf{K} * \mathbf{r} = \mathbf{f}, \tag{1}$$

where $\mathbf{K}$ is the global stiffness matrix, $\mathbf{f}$ is global vector of external forces, and $\mathbf{r}$ is vector of unknown displacements. One of the key feature of the FEM is that the stiffness matrix is typically positive definite, symmetric matrix with sparse structure, which is a consequence of using interpolation and test functions with limited nonzero support. The efficient algorithm for solving linear system must exploit the symmetry and sparse structure, by saving considerable memory and CPU resources. There exist number of different storage schemes for sparse matrices. The most widely used are skyline, compressed row, and compress column formats. In this contribution, the compressed row format will be used. This choice comes from SuperLU library, which requires the sparse matrix in this format on input.

In the compressed row format, only nonzero entries of the sparse matrix are stored in one dimensional array. Additional integer array is needed to store column indices of the stored values. In practical implementation, the one dimensional arrays storing nonzero values are merged together into a single array containing all nonzero entries, row by row. The same applies for arrays of column indices. An additional array with size equal to number of rows is needed to point to the beginning of each row record, see Fig. 1.

2. Implementation SuperLU Interface

As noted already in the previous section, the distributed memory programming model does not provide global address space and global memory accessible by all processing nodes, but the memory is distributed among the processing units. For large problems, it is therefore essential to establish a distributed representation of the sparse matrix. This feature is also supported by SuperLU library, allowing the user to split the global sparse matrix stored in comprow format into blocks, representing compressed row storage for consequent, non-overlapping groups of rows, which are distributed across computing nodes.

We assume, that the decomposition of the discretized problem domain has been established and individual, non-overlapping sub domains (partitions) are assigned to and stored locally on individual computing nodes. The so called node-cut strategy is assumed, where the cut dividing the problem domain runs through the nodes. Nodes on mutual partition boundaries are called shared nodes, the nodes inside individual partitions are so-called local nodes. Each processing node is responsible for assembling its contribution to the global system matrix by summing up the contributions from individual elements. To minimize the 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, the ownership has to be defined. In this work, the partition with lowest rank sharing the shared node is the owner of shared node and thus responsible of 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 particular node. The assembly process requires to establish global, unique numbering of equations across computing nodes in addition to the local numbering on individual partitions.

The assembly process of the global, distributed system matrix requires basically two steps: set up of data structures required to store the block of compressed row records on individual partitions and the assembly process itself based on localization of individual element contributions into global equilibrium equations according to their connectivity.

2.1. Initialization of data structures for distributed matrix

This phase should determine the required size of arrays used to store all nonzero entries of the global matrix. The nonzero contributions of each element can be identified from element code numbers. The code numbers represent simultaneously the numbering of equilibrium equations and numbering of corresponding unknowns. The individual values in the stiffness matrix represent nodal forces caused by corresponding unit displacement. The entry in element stiffness matrix representing the nodal force contributing to k-th equilibrium equation and being multiplied by displacement located in global displacement vector at l-th position, should be added to position (k,l) in the global matrix. By assuming that element contributions are full matrices, one can initialize the nonzero structure of the global stiffness matrix using element

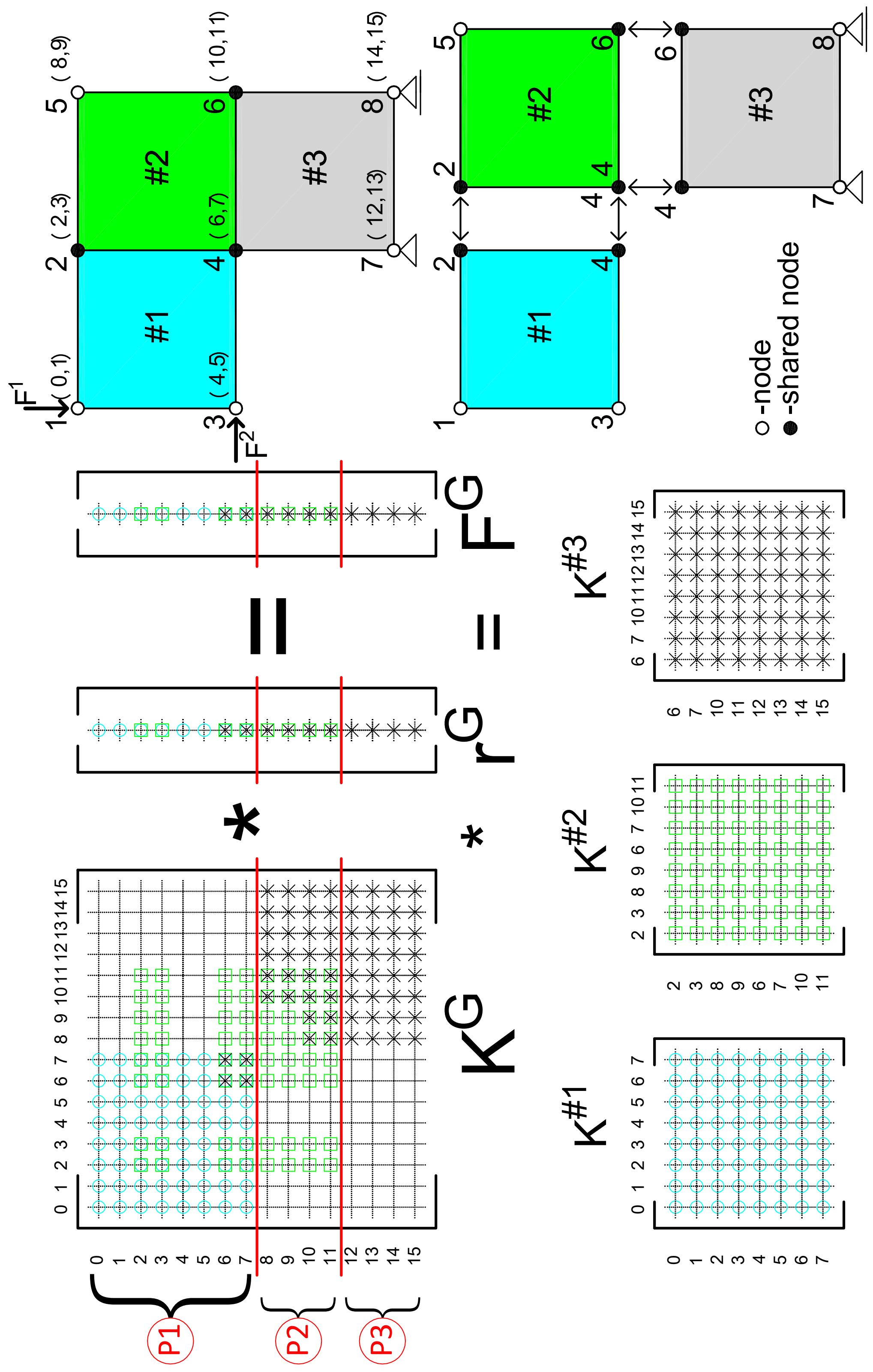

FIGURE 2. Example of simple problem consisting of three elements distributed into different computing nodes, illustrating the global sparse matrix structure and its distribution.

code numbers. In distributed case, each partition is responsible for initializing and prealocating its block structure of compressed rows records. In case of the compressed row storage, the number of nonzero entries in each row has to be determined together with corresponding nonzero column positions of the individual entries.

Each processing node is responsible for initialization and assembly of the assigned block of rows. This block is composed from local contributions, as well as from shared node contributions from neighboring partitions. Therefore, the overall initialization can be divided into (i) local stage, where the local contributions to nonzero pattern are identified (from the partition elements), and (ii) communication stage, where contributions from neighboring partitions are received for locally maintained rows, corresponding to equilibrium equations for shared nodes.

Any partition has to therefore keep its locally maintained block of compressed rows and also block compressed rows corresponding to shared nodes not being locally maintained. These two blocks can be stored separately. In the initial stage, the block structure is initialized on every partition using only the contributions from local elements. The row entries, that correspond to the shared nodes which are not locally maintained are then communicated (send) to the corresponding partition maintaining corresponding row. After the sending the data, the contributions from neighboring partitions are received containing the contributions to the locally maintained shared equations. Non-blocking communication [8] is used to send/receive the contributions. Non-blocking communication allows to utilize the potential overlap of communication and computation, which is utilized in the actual implementation. After finishing the communication stage, each partition has fully initialized data structure for locally maintained block of rows.

2.2. Assembly of the global distributed matrix

After finishing the initialization step, in which memory has been allocated for every non zero entry of the global distributed stiffness matrix, one can proceed with the actual assembly of the matrix from the element contributions. This is done in a similar fashion to the previous step. On each partition the contributions from local elements are assembled. Then, the rows corresponding to shared nodes not being locally maintained have to be sent to the partition maintaining corresponding row. Finally, each partition has to receive remote contribution to the locally maintained rows of shared nodes. This process is illustrated on Fig. 2. In this example, the problem is composed of three elements (corresponding to sub domains as well) and eight nodes, from which five are local on individual partitions and three nodes are shared.

2.3. Assembly of the global right hand side

The process of assembling the global right hand side vector is very similar to the process of assembling the stiffness matrix, but in many aspects is simpler, do to the assembly of local vector contributions. The global vector is again distributed among partitions, the distribution corresponds to the distribution of matrix rows. The contributions from local elements are assembled first, then those corresponding to rows maintained on remote partitions are communicated and contributions from neighbors are received and entries updated.

By finishing the steps described above, the distributed sparse matrix, represented by locally maintained blocks of row entries on individual partitions, can be directly passed into SuperLU routines.

3. Conclusions

In this work the algorithm of assembling the distributed sparse matrix in row-column format is described, in order to develop the interface to the distributed SuperLU solver. The evaluation of the performance and scalability is the subject of ongoing work. The authors also aim to compare the efficiency of the distributed solver to the efficiency of the multithreaded SuperLU version.

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.

References

[1] B. Patzak. *Parallel computations in structural mechanics.* České vysoké učení technické v Praze, 2010.

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

[3] B. Patzak, D. Rypl. Parallel adaptive finite element computations with dynamic load balancing. *Proceedings of the First International Conference on Parallel, Distributed and Grid Computing for Engineering* 2009.

[4] X. S. Li, J. W. Demmel, J. R. Gilbert, et al. Superlu users‘ guide program interface. http://crd.lbl.gov/xiaoye/SuperLU/, September 1999.

[5] M. Bosansky, B. Patzak. Evaluation of different approaches to solution of the direct solution of large, sparse systems of linear equations. *Advanced Materials Research* **1144**(1144):97–101, 2016.

[6] M. Bosansky, B. Patzak. On parallelization of linear system equation solver in finite element software. *Engineering Mechanics 2017* **1**(1):206–209, 2017.

[7] P. S. Pacheco. *A User‘s Guide to MPI.* Univ. of San Francisco, 1997.

[8] T. Saif, M. Parashar. *Unferstanding the Behavior and Performance of Non-blocking Communications in MPI.* Department of Electrical and Computer Engineering, Rutgers University, 2002.