Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1454 – 1463

Procedia Computer Science

Graph Grammar-Based Multi-Frontal Parallel Direct Solver for Two-Dimensional Isogeometric Analysis

Krzysztof Kuźnik[a], Maciej Paszyński[a], Victor Calo[b]

[a]*AGH University of Science and Technology, Krakow, Poland*
[b]*King Abdullah University of Science and Technology, Thuwal, Saudi Arabia*

Abstract

This paper introduces the graph grammar based model for developing multi-thread multi-frontal parallel direct solver for two dimensional isogeometric finite element method. Execution of the solver algorithm has been expressed as the sequence of graph grammar productions. At the beginning productions construct the elimination tree with leaves corresponding to finite elements. Following sequence of graph grammar productions generates element frontal matrices at leaf nodes, merges matrices at parent nodes and eliminates rows corresponding to fully assembled degrees of freedom. Finally, there are graph grammar productions responsible for root problem solution and recursive backward substitutions. Expressing the solver algorithm by graph grammar productions allows us to explore the concurrency of the algorithm. The graph grammar productions are grouped into sets of independent tasks that can be executed concurrently. The resulting concurrent multi-frontal solver algorithm is implemented and tested on NVIDIA GPU, providing $O(NlogN)$ execution time complexity where N is the number of degrees of freedom. We have confirmed this complexity by solving up to 1 million of degrees of freedom with 448 cores GPU.

Keywords: parallel computing, isogeometric analysis, models of concurrency, graph grammar

1. Introduction

This paper presents the parallel multi-frontal direct solver for higher order finite element method (FEM). The higher order FEM can be implemented with *hp* finite elements introduced by Demkowicz [3, 4] or with B-spline based finite elements [1]. The advantage of the B-spline based FEM is that it provides higher global continuity of order C^k while Demkowicz finite elements provide only C^0 continuity.

The multi-frontal direct solver is the state of the art algorithm for solving sparse linear systems of equations arising from finite element method computations. The sequential version of the algorithm was first proposed by [5, 6]. The input for the algorithm is the elimination tree, that in the leaves contains the frontal matrices corresponding to particular finite elements. The solver algorithm browses the elimination tree from leaves up to the root. During this operation it performs partial forward eliminations over the frontal matrices, to eliminate the fully assembled degrees of freedom. Resulting Schur complement sub-matrices are merged at parent nodes of the elimination tree. The process

Email address: paszynsk@agh.edu.pl (Maciej Paszyński)
URL: http://home.agh.edu.pl/~paszynsk (Maciej Paszyński)

doi:10.1016/j.procs.2012.04.160

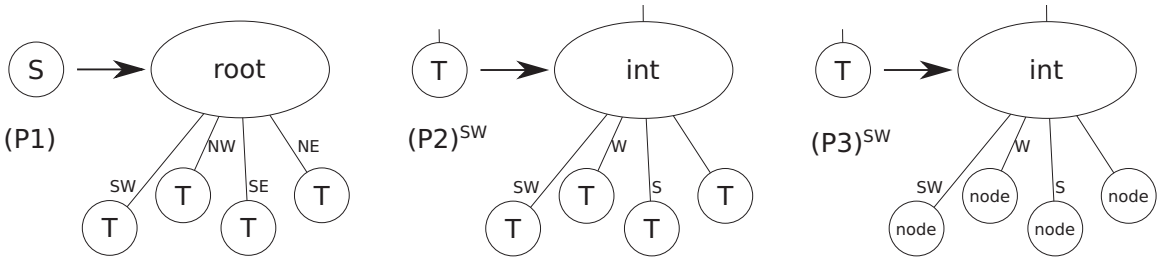

Figure 1: Graph grammar for 2D mesh generation. The productions are cloned with different direction attributes

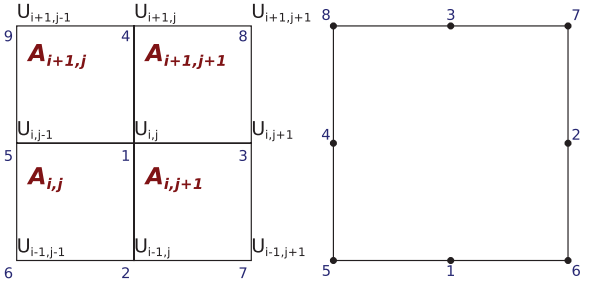

Figure 2: The ordering of 9 nodes in the system created by production **(Mi,j)**.

is repeated up to the root of the elimination tree. The root problem is fully assembled, so it can be solved and followed by recursive backward substitutions going down to the leaves. The parallel multi-frontal solver has been implemented for C^0 two dimensional *hp*-FEM [12].

The multi-frontal solver algorithm together with the process of generation of the elimination tree can be expressed by graph grammar. The graph grammar generating the structure of the computational mesh was derived for two dimensional rectangular, triangular and mixed finite elements mesh [10, 11, 9]. The graph grammar based solver algorithm for two and three dimensional *hp*-FEM computations on Linux clusters was developed [14, 15, 13]. In this paper we present the derivation of the graph grammar based multi-frontal direct solver for two dimensional isogeometric finite element method for shared memory machines.

The algorithm of the multi-frontal solver has been expressed by graph grammar productions. This includes the generation of the elimination tree, creation of the element matrices, partial forward eliminations and merging of the local systems when browsing the elimination tree, solution of the root problem as well as recursive backward substitutions. Expressing the solver algorithm by graph grammar productions allows us to identify basic undividable tasks that must be executed in sequence. The partial order of execution between tasks (graph grammar productions) is identified, and the tasks are grouped into sets of independent tasks that can be executed concurrently. In other words the solver algorithm is transformed into a sequence of sets with basic undividable tasks called graph grammar productions. These sets are executed concurrently, set by set. Thus the graph grammar approach allows us to transform the sequential multi-frontal solver algorithm into highly parallelizable concurrent algorithm intended for the shared memory machines. We claim the graph grammar can be effectively utilized as a model of concurrency.

The multi-frontal solver algorithm implemented on a single processor delivers $O(N^{1.5})$ execution time [7]. It is possible to improve its efficiency to the linear $O(N)$ execution time using the concept of h-matrices [7]. However, the price to pay is that hyper-matrices provide only approximate solution, and the cost of compressing the original matrix into the h-matrix is large.

In this paper we provided the concurrent multi-frontal solver algorithm expressed by the graph grammar formalism. The solver has been tested on NVIDIA GPU. The implementation provides $O(NlogN)$ execution time complexity, where N is the number of degrees of freedom. We have confirmed this complexity of the algorithm experimentally with 448 cores GPU, up to 1 million degrees of freedom. We claim that this is the best possible time complexity one can obtain when using the parallel direct solver algorithm [2].

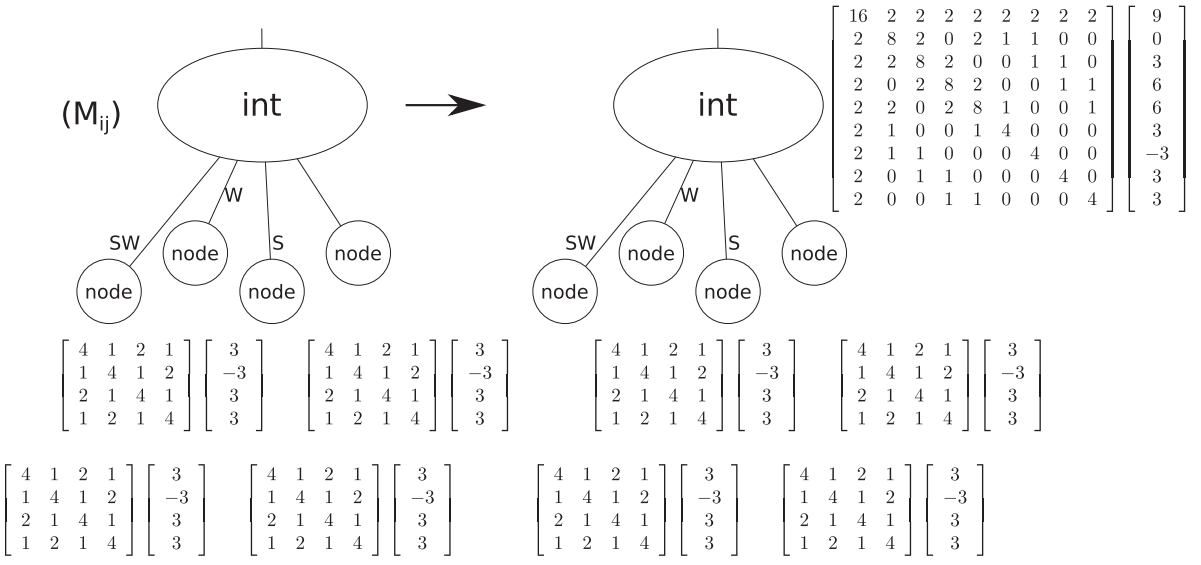

Figure 3: Graph grammar production for merging of four leaves matrices.

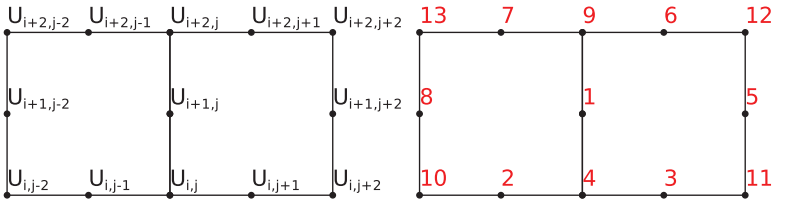

Figure 4: Ordering for nodes for merging of two horizontal blocks on the second level.

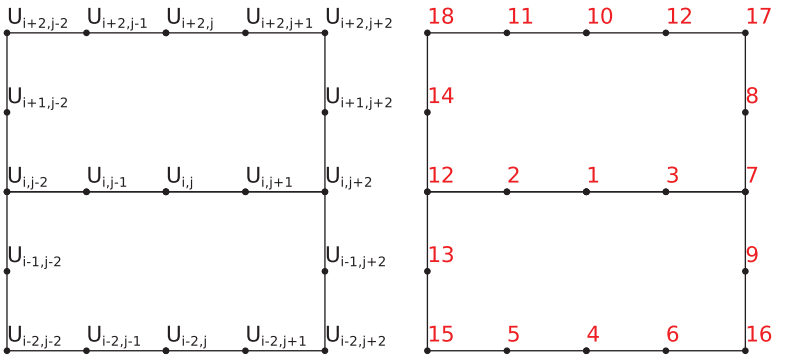

Figure 5: Ordering for nodes for merging of four blocks, and the resulting reordering for the next level.

2. Finite Element Method for Two Dimensional Problems

We focus on two dimensional Laplace equation

$$\Delta u = 0 \tag{1}$$

with boundary conditions

$$u(x_1, 0) = 0 \qquad u(x_1, 1) = 1 \qquad \nabla u(0, x_2) \cdot n = \nabla u(1, x_2) \cdot n = 0 \tag{2}$$

for $x_1, x_2 \in [0, 1]$

The weak variational formulation is obtained by taking L^2 scalar product with functions $v \in H^1(\Omega)$ and performing the shift $u \in \tilde{u} + H^1(\Omega)$ where $\tilde{u} = x_2$.

$$\text{Find } u \in V = \{u \in H^1(\Omega) : tru = 0 \text{ for } x_1 \in [0,1], x_2 \in \{0,1\}\} \text{ such that } b(v,u) = l(v), \forall v \in V$$
$$b(u,v) = \int_\Omega \nabla u \cdot \nabla v dx \qquad l(v) = -\int_\Omega \frac{\partial v}{\partial x_2} dx \tag{3}$$

The computational mesh is partitioned into finite elements. We utilize the B-spline functions for approximation of the solution

$$u(\xi, \eta) \approx \sum_{i,j} B_{i,j;p}(\xi, \eta) U_{i,j} = \sum_{i,j} N_{i,p}(\xi) N_{j,p}(\eta) U_{i,j} \tag{4}$$

where

$$N_{i,0}(\xi) = I_{[\xi_i, \xi_{i+1}]}$$
$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{x_{i+p} - \xi_i} N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{x_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi) \tag{5}$$

We end up with the following discrete form of the weak equation

$$\text{Find } u \in V = \sum_{i,j} B_{i,j;p} U_{i,j} = \sum_{i,j} N_{i,p} N_{j,p} U_{i,j} \text{ such that } \sum_{i,j} b\left(B_{i,j}, B_{k,l}\right) = l(B_{k,l}) \; \forall k, l \tag{6}$$

3. Graph grammar for generation of the elimination tree

In this section we introduce the graph grammar productions responsible for generation of the elimination tree. In Figure 1 we introduce graph grammar productions responsible for generation of the two dimensional mesh. The exemplary sequence of graph grammar productions generating a 4x4 elements mesh is

$$(P1) - (P2)^{SW} - (P2)^{NW} - (P2)^{SE} - (P2)^{NE} - (P3)^{SW} - (P3)^{S} - (P3)^{W} - (P3)_{11} -$$
$$(P3)^{SE} - (P3)^{S} - (P3)^{E} - (P3)_{12} - (P3)^{NE} - (P3)^{E} - (P3)^{N} - (P3)_{22} - (P3)^{NW} - (P3)^{W} - (P3)^{N} - (P3)_{21} \tag{7}$$

4. Graph grammar for generation of local matrices

The computational problem has been partitioned into finite elements. The matrix contributions are partitioned into finite elements K and replaced by Gaussian quadrature with weights $w_{m,n}$ and Gaussian quadrature points (x_m, x_n):

$$b\left(B_{i,j}, B_{k,l}\right) = \int_\Omega \nabla B_{i,j} \cdot \nabla B_{k,l} dx = \sum_K \int_K \nabla B_{i,j} \cdot \nabla B_{k,l} dx = \sum_K \sum_{m,n} w_{m,n} \nabla B_{i,j}(x_m, x_n) \cdot \nabla B_{k,l}(x_m, x_n) |K|$$
$$l(B_{k,l}) = -\int_\Omega \frac{\partial B_{k,l}}{\partial x_2} = -\sum_K \int_K \frac{\partial B_{k,l}}{\partial x_2} dx = -\sum_{m,n} w_{m,n} \frac{\partial B_{k,l}(x_m, x_n)}{\partial x_2} |K| \tag{8}$$

To illustrate the multi-frontal solver algorithm, we will perform some computations manually on a simple 4×4 finite element method mesh, for linear B-splines $p = 1$. We name graph grammar production for generation of the element local frontal matrices. We assume the following ordering of unknowns for each element: bottom left, bottom right, top left, top right.

Production **(Ai,j)** generates an element frontal matrix. As an argument it takes a leaf node graph vertex with label **node**. Here **i, j** denotes two dimensional index of element in our domain.

$$\begin{bmatrix} 4 & 1 & 2 & 1 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 4 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \begin{bmatrix} u_{i-1,j-1} \\ u_{i-1,j} \\ u_{i,j-1} \\ u_{i,j} \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ 3 \\ 3 \end{bmatrix} \tag{9}$$

4.1. Graph grammar for merging local matrices and eliminating fully assembled rows

Let us assume we have four local systems assigned to four adjacent finite elements having common $u_{i,j}$ node. The first system has been generated by production **(Ai,j)**, the second system has been generated by production **(Ai+1,j)**

$$\textbf{(Ai,j)} \begin{bmatrix} 4 & 1 & 2 & 1 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 4 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \begin{bmatrix} u_{i-1,j-1} \\ u_{i-1,j} \\ u_{i,j-1} \\ u_{i,j} \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ 3 \\ 3 \end{bmatrix} \quad \textbf{(Ai+1,j)} \begin{bmatrix} 4 & 1 & 2 & 1 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 4 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \begin{bmatrix} u_{i,j-1} \\ u_{i,j+1} \\ u_{i+1,j-1} \\ u_{i+1,j} \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ 3 \\ 3 \end{bmatrix} \tag{10}$$

The third system has been generated by production **(Ai,j+1)**, the fourth system has been generated by production **(Ai+1,j+1)**

$$\textbf{(Ai,j+1)} \begin{bmatrix} 4 & 1 & 2 & 1 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 4 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \begin{bmatrix} u_{i-1,j} \\ u_{i-1,j+1} \\ u_{i,j} \\ u_{i,j+1} \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ 3 \\ 3 \end{bmatrix} \quad \textbf{(Ai+1,j+1)} \begin{bmatrix} 4 & 1 & 2 & 1 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 4 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \begin{bmatrix} u_{i,j} \\ u_{i,j+1} \\ u_{i+1,j} \\ u_{i+1,j+1} \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ 3 \\ 3 \end{bmatrix} \tag{11}$$

All rows in all these four matrices are not fully assembled yet. We must merge these four matrices into new single matrix, to get one row fully assembled. The row will be associated with the central variable $U_{i,j}$. The ordering in the new system is illustrated in Figure 2. The merging results in the following system of linear equations expressed by production **(Mi,j)**

$$\begin{bmatrix} 16 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 2 & 8 & 2 & 0 & 2 & 1 & 1 & 0 & 0 \\ 2 & 2 & 8 & 2 & 0 & 0 & 1 & 1 & 0 \\ 2 & 0 & 2 & 8 & 2 & 0 & 0 & 1 & 1 \\ 2 & 2 & 0 & 2 & 8 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 & 4 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 & 0 & 4 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 0 & 0 & 4 & 0 \\ 2 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} u_{i,j} \\ u_{i-1,j} \\ u_{i,j+1} \\ u_{i+1,j} \\ u_{i,j-1} \\ u_{i-1,j-1} \\ u_{i-1,j+1} \\ u_{i+1,j+1} \\ u_{i+1,j-1} \end{bmatrix} = \begin{bmatrix} 9 \\ 0 \\ 3 \\ 6 \\ 6 \\ 3 \\ -3 \\ 3 \\ 3 \end{bmatrix} \tag{12}$$

The Figure 3 presents the graph grammar production over the elimination tree.

The next step is to perform the elimination of the first fully assembled row. The first row must be subtracted from all 8 following rows. Notice that we can distinguish particular subtractions so in the parallel version of the solver algorithm they can be performed in concurrent. This is expressed in productions **(S1-k)i,j** for $k = 2, ..., 8$. The first row is fully assembled, since it corresponds to the node $U_{i,j}$ that has support over four elements neighboring the node $U_{i,j}$. We have just assembled all four frontal matrices, so the contribution corresponding to the node are fully assembled. We just subtract the first row from the following rows. Notice that the sparse matrix becomes rather dense matrix after elimination of the first row.

In the next step, having two matrices at son level we merge them into a new matrix at parent level. The ordering for the merging process is depicted in Figure 4. This is done by graph grammar production **(M2i,j)**

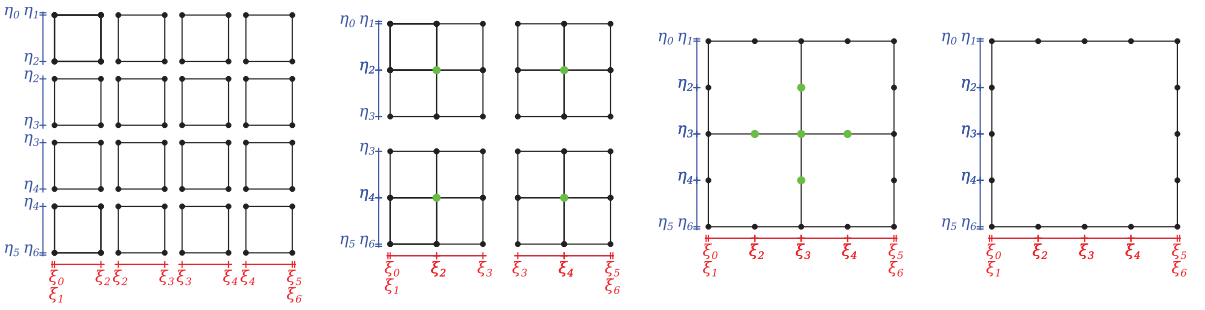

Figure 6: Summary of the multi-frontal solver algorithm for linear B-splines.

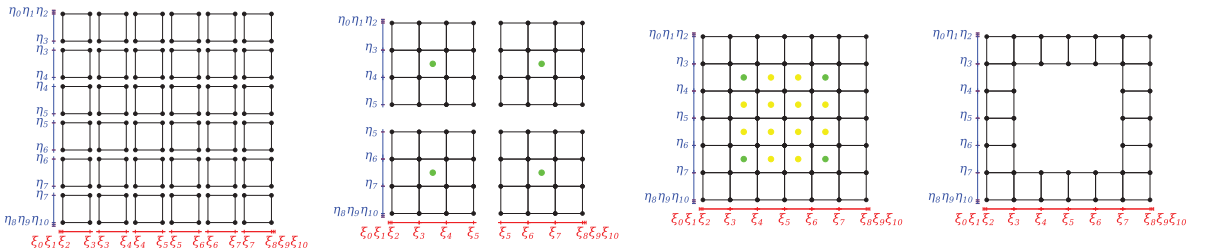

Figure 7: Summary of the multi-frontal solver algorithm for quadratic B-splines.

The next step is to perform the elimination of the first fully assembled row. The first row must be subtracted from all 12 following rows, and in the parallel version of the solver algorithm the subtractions can be performed concurrently. This is expressed in productions **(S1-k)i,j** for $k = 2, ..., 12$.

The first row is fully assembled, since it corresponds to a node $U_{i+1,j}$ that has support over four elements neighboring the node $U_{i+1,j}$. We have just assembled two frontal matrices with contributions from all four elements neighboring the node.

In the next step we merge two resulting blocks into a new block. This is done by production **(M4i,j)**. The ordering for this operation is depicted in top panels in Figure 5. The first three rows must be subtracted from all 16 following rows, and the subtractions can be performed concurrently. This is expressed in productions **(Sl-k)i,j** for $l = 1, 2, 3, k = l + 1, ..., 16$.

The three rows are fully assembled since we have just merged two matrices with all contributions coming from all elements neighboring the nodes $U_{i,j-1}$, $U_{i,j}$ and $U_{i,j+1}$.

Finally, the boundary problem is constructed. This is done by production **(Root)**. The size of the root problem is 16×16, and we perform full forward elimination followed by backward substitution. For the root problem we distinguish productions responsible for particular subtractions of rows. First, we define productions representing subtractions of the first row from all the following rows: **(S1-k)** for $k = 2, ..., 16$. Then, we define tasks for subtractions of the second row from all following rows, and so on. In general we have the following productions **(Sl-k)** for $l = 1, ..., 15, k = l + 1, ..., 16$.

The root problem solution is followed by a sequence of backward substitutions, traveling from the root of the elimination tree down to the leaves. These operations are denoted by **(BS*)** graph grammar productions.

5. Sequence of graph grammar productions for the solver algorithm

The entire algorithm is summarized in Figure 6 for linear B-splines and in Figure 7 for quadratic B-splines. For quadratic and higher order B-splines the algorithm is similar. However, the support of p order B-splines spreads over $(p + 1)^2$ elements. In other words, in the first step we need to merge $(p + 1)^2$ element frontal matrices, and we can eliminate one fully assembled central B-spline denoted in Figure 7 by green dot. In the following steps we merge four frontal matrices and we can eliminate central B-splines denoted in Figure 7 by green dots.

At this point we have expressed the entire multi-frontal solver algorithm by the sequence of graph grammar productions. There are graph grammar productions responsible for generation of the elimination tree, productions responsible for constructing element frontal matrices, productions for concurrent row subtractions during partial forward eliminations, productions for merging of the resulting Schur complement matrices, productions for the following concurrent partial forward eliminations, merging, and finally the root concurrent problem solution followed by recursive backward substitutions. We assume that graph grammar productions are basic undividable tasks that must be executed sequentially. We can find the partial order for executions of these tasks (graph grammar productions), and group them into sets that can be executed concurrently.

The graph grammar based solver has been implemented and tested on NVIDIA GPU. The sets of identical tasks that can be executed concurrently have been scheduled on GPU cores. The necessary data have been transferred from global memory into nodes shared memory. The following chapter introduces the implementation details.

6. NVIDIA CUDA implementation

Recent rapid development of GPUs had lead to the situation where not only are they used for purpose of graphics but they can act as a multi core computer for extensive calculations. Almost every modern GPU supports thousands of active threads which can process data much more efficiently than ordinary CPU. This is why we decided to use NVIDIA CUDA (Compute Unified Device Architecture) and its SDK to implement graph grammar-based solver. This architecture is perfectly suitable for our task as we are able to process grammar production nodes independently and parallelly on different GPU multiprocessors.

Applications using GPU are heterogeneous - they consist of host (CPU) code executed sequentially and device (GPU) code executed parallelly. In our case host code is mostly responsible for memory management and launching GPU actions, while GPU code is responsible for all calculations. While GPU architecture allows to speed up the code it also puts some serious restrictions on resources that can be used i.e. amount of shared memory or number of threads run by multiprocessor. Those restrictions caused our time results to scale up slightly different than theory suggests.

Here is quick outline of CUDA architecture and programming model for reader to better understand the algorithm implementation. GPU consists of several multiprocessors (usually 8-14). Each of those multiprocessors has tens of CUDA cores (eg. 32 for NVIDIA Tesla C2070). There are 4 basic kinds of memory that program can use:

- *global memory* (up to 6GB) - this can be accessed by each thread on device but is pretty slow
- *shared memory* (up to 48KB per multiprocessor) - this is low latency memory which is shared among all threads run by multiprocessor
- *constant memory* and *texture memory* which are not relevant for this paper

For an algorithm to be fast it is highly desirable to use shared memory as much as possible. However, it is often not possible to fit data into 48 kilobytes. This forces use of global memory which is much slower. Moreover, access to global memory should be coalesced because it is transactional. It means that data layout in global memory also has great influence on algorithm performance.

Programming model assumes that software is independent of device to allow high scalability and compatibility with different hardware. Hence the notion of a *block*. Block is a set of threads which run on a single multiprocessor and which use common shared memory. Several blocks might be run on one multiprocessor but one block is never split between multiprocessors. To make programming easier blocks as well as threads inside block can be organized into 2D or 3D grids which simplifies processing of matrices or volumes.

Below we will describe briefly how graph grammar productions were implemented. Let us assume that n is the number of elements in one dimension of a domain. Total number of elements in domain is $n^2 = N$.

6.1. Generation of local matrices

Generation of local matrices is basically evaluation of function $b(B_{ij}, B_{kl})$ over each element. We run a grid of $n \times n$ blocks with $(p+1)^2 \times (p+1)^2$ threads each. Then we have to calculate right hand side of our equation - function $l(B_{kl})$. To do it, we run a grid of $n \times n$ blocks with $(p+1)^2$ threads each. All necessary integration is implemented as weighted summation over Gaussian quadrature points.

6.2. Initial merge and eliminating fully assembled rows

First step of merging is slightly different from next ones. In initial merge we put together $(p + 1)^2$ matrices to fully assemble just one row. We run 2D grid of blocks where each block is responsible for merging one matrix from $(p + 1)^2$ matrices.

6.3. Merging matrices and eliminating fully assembled rows

This is the most time consuming part of the algorithm. Merging was divided into horizontal and vertical merge to limit number of rows eliminated in one step and make better use of fast shared memory. At each level we run as many blocks as many merged matrices we receive. Number of threads in a block depends on stage of the step. At first there is as many threads as length of a row in contributing matrix. Then number of threads is equal to the number of rows eliminated in current step. Then it is equal to the number of rows that are not eliminated. We merge matrices until we reach the top of the elimination tree where we are left with wire frame problem.

6.4. Last merge and solution

Last step is a moment when we are left with a big dense system of linear equations. We decided to use external library (MAGMA) [8] to solve it to achieve high performance.

6.5. Backward substitution

Having last system of equations solved we start a backward substitution using dependencies from rows that we eliminated on each level. Process is analogically reversed to merging. For each block we run as many threads as there were eliminated rows.

6.6. Result calculation

At the end we have coefficients for every basis function. With those we are easily able to calculate values of function u in Gaussian quadrature points. We run grid of $n \times n$ blocks with number of threads equal to number of quadrature points in one element.

7. Numerical results

We tested our implementation on NVIDIA Tesla C2070 device which has 14 multiprocessors with 32 CUDA cores per multiprocessor which gives us 448 CUDA cores. Total amount of global memory is 5375 megabytes. We used CUDA 4.0 version.

Included time results are for $p = 1$ (Figure 8), $p = 2$ (Figure 9) and $p = 3$ (Figure 10). The plots list all parts of the concurrent solver algorithm, including the generation of matrices at leaf nodes, merging of son nodes matrices into a parent node matrices (sum of the time over all levels of the tree), root problem solution (performed by the call to the MAGMA solver [8], since it is a dense problem), backward substitutions and retrieving the solution. The most expensive part is the merging of matrices. This is related to the fact that this process requires expensive transfers from global to nodes shared memory. In these experiments we were limited to 1048576 elements for linear, 589824 for quadratic and 262144 for cubic B-splines because of a global memory size.

We do not compare our solver with MAGMA [8] library designed for GPU since MAGMA provides dense solver, while our problem is sparse and way too large for the MAGMA call. On the other hand we utilize MAGMA for the root dense problem solution in two dimensions.

8. Conclusion

We introduced the graph grammar methodology for concurrent solution of linear systems encountered while working with B-spline based finite element method delivering higher order global continuity of the solution. The implementation provides $O(NlogN)$ execution time complexity where N is the number of degrees of freedom. We have confirmed the time complexity of the algorithm experimentally with 448 cores GPU, up to 1 million degrees of freedom for linear B-splines, up to 589824 for quadratic B-splines and up to 262144 for cubic B-splines. The developed model was implemented and tested on NVIDIA GPU. The future work will include the extension of the methodology for three dimensional problems where we predict to obtain $O(N^{1.33})$ execution time complexity, where N is the number of degrees of freedom.

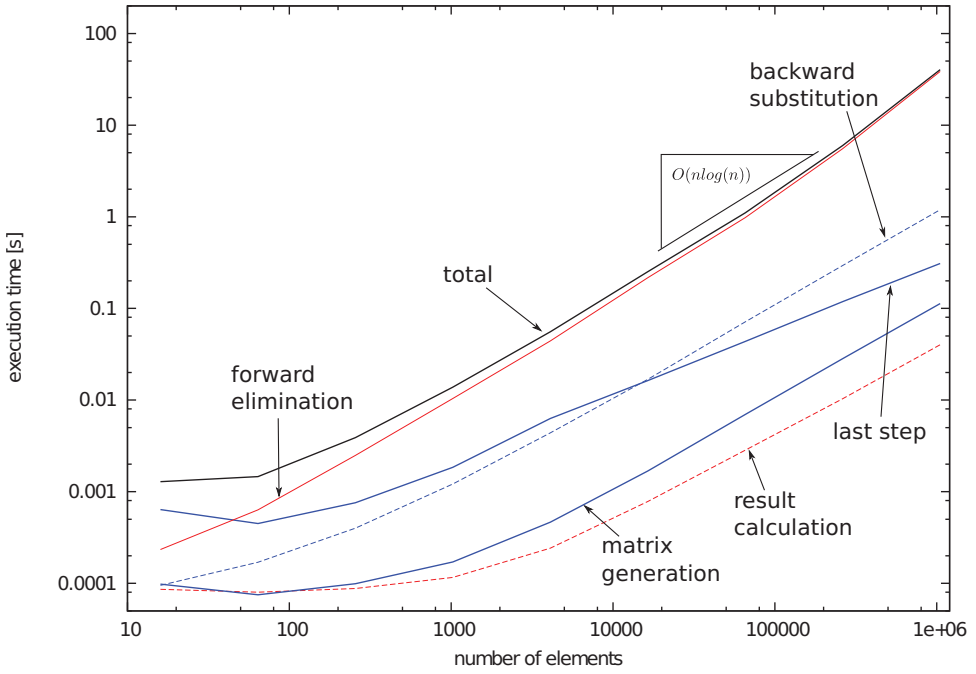

Figure 8: Results for $p = 1$

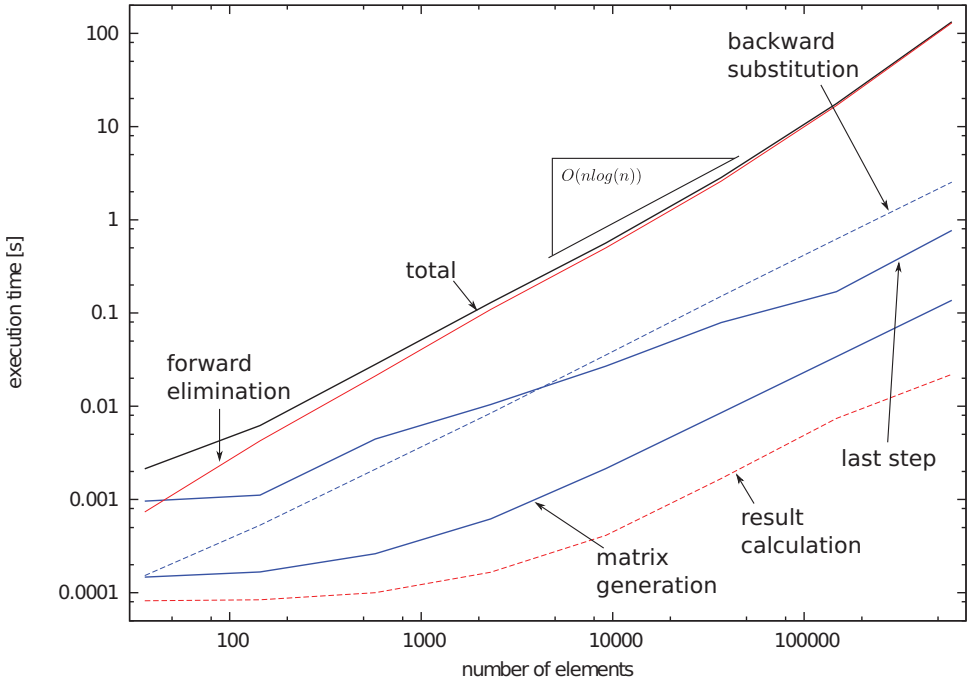

Figure 9: Results for $p = 2$

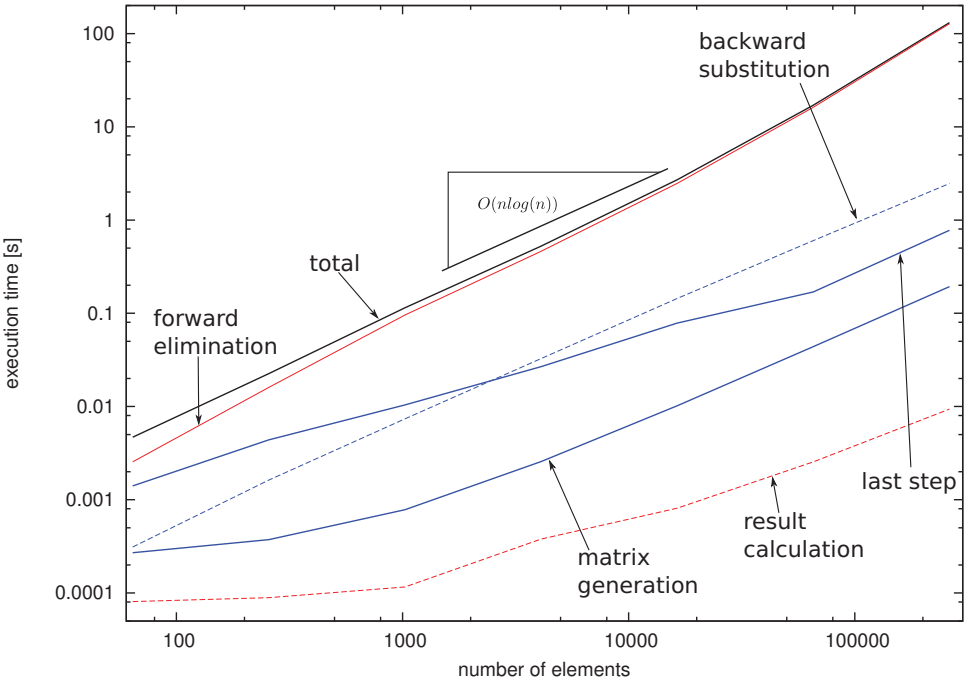

Figure 10: Results for $p = 3$

Acknowledgment

The work was partially supported by the Polish MNiSW grant no. NN 519 447739. The support of this work by the Polish National Science Center under grant no DEC-2011/01/B/ST6/00674 is gratefully acknowledged.

[1] J. A.Cottrel, T. J. R. Hughes, Y. Bazilevs *Isogeometric Analysis. Toward Integration of CAD and FEA*, Wiley, 2009.
[2] N. Collier, D. Pardo, L. Dalcin, M. Paszyński, V. M. Calo *The cost of continuity: a study of the performance of isogeometric finite elements using direct solvers*. Computer Methods in Applied Mechanics and Engineering, in press, doi:10.1016/j.cma.2011.11.002, 2011.
[3] L. Demkowicz, *Computing with hp-Adaptive Finite Element Method. Vol. I.One and Two Dimensional Elliptic and Maxwell Problems*. Chapmann & Hall / CRC Applied Mathematics & Nonlinear Science, 2006.
[4] L. Demkowicz, J. Kurtz, D. Pardo, M. Paszyński, W. Rachowicz, A. Zdunek *Computing with hp-Adaptive Finite Element Method. Vol. II. Frontiers: Three Dimensional Elliptic and Maxwell Problems*. Chapmann & Hall / CRC Applied Mathematics & Nonlinear Science, 2007.
[5] I. S. Duff, J. K. Reid *The multifrontal solution of unsymmetric sets of linear systems*, SIAM Journal on Scientific and Statistical Computing, vol. 5, 1984; p.633–641.
[6] I.S. Duff, J.K. Reid; *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1973) p. 302-325.
[7] Phillip G. Schmitz, Lexing Ying; *A fast direct solver for elliptic problems on general meshes in 2D*, J. Comput. Physics 231(4): p. 1314-1338 (2012)
[8] MAGMA Matrix Algebra on GPU and Multicore Architecture, http://icl.cs.utk.edu/magma/
[9] A. Paszyńska, M. Paszyński, E. Grabska *Graph transformations for modeling* hp*-adaptive Finite Element Method with mixed triangular and rectangular elements*. Lecture Notes in Computer Science vol. 5545, 2009; 875-884.
[10] A. Paszyńska, M. Paszyński, E. Grabska *Graph transformations for modeling* hp*-adaptive Finite Element Method with triangular elements*. Lecture Notes in Computer Science vol. 5103, 2008; 604–613.
[11] M. Paszyński, A. Paszyńska, *Graph transformations for modeling parallel* hp*-adaptive Finite Element Method* . Lecture Notes in Computer Science vol. 4967, 2008; 1313–1322.
[12] M. Paszyński, D. Pardo, C. Torres-Verdin, L. Demkowicz, V. Calo *A Parallel Direct Solver for Self-Adaptive hp Finite Element Method*. Journal of Parall and Distributed Computing vol.70, 2010; p. 270–281.
[13] M. Paszyński, D. Pardo, A. Paszyńska *Parallel multi-frontal solver for p adaptive finite element modeling of multi-physics computational problems*. Journal of Computational Science vol.1, iss.1, 2010; p. 48–54.
[14] M. Paszyński, R. Schaefer *Graph grammar driven partial differential eqautions solver*. Concurrency and Computations: Practise and Experience vol.22 iss.9. 2010; p. 1063–1097.
[15] A. Szymczak, M. Paszyński *Graph grammar based Petri net controlled direct sovler algorithm*. Computer Science vol.11, 2010; p.65–79.