Maciej Woźniak
Anna Bukowska

COMPARISON OF MULTI-FRONTAL AND ALTERNATING DIRECTION PARALLEL HYBRID MEMORY iGRM DIRECT SOLVER FOR NON-STATIONARY SIMULATIONS

Abstract *Three-dimensional isogeometric analysis (IGA-FEM) is a modern method for simulation. The idea is to utilize B-splines or NURBS basis functions for both computational domain descriptions and engineering computations. Refined isogeometric analysis (rIGA) employs a mixture of patches of elements with B-spline basis functions and C^0 separators between them. This enables a reduction in the computational cost of direct solvers. Both IGA and rIGA come with challenging sparse matrix structures that are expensive to generate. In this paper, we show a hybrid parallelization method using hybrid-memory parallel machines. The two-level parallelization includes the partitioning of the computational mesh into sub-domains on the first level (MPI) and loop parallelization on the second level (OpenMP). We show that the hybrid parallelization of the integration reduces the contribution of this phase significantly. We compare the multi-frontal solver and alternating direction solver, including the integration and the factorization phases.*

Keywords isogeometric finite element method, integration, shared memory, parallel

Citation Computer Science 21(4) 2020: 419–439

Copyright © 2020 Author(s). This is an open access publication, which can be used, distributed and reproduced in any medium according to the Creative Commons CC-BY 4.0 License.

1. Introduction

The isogeometric analysis (IGA-FEM) introduced by Cottrel et. al. [16] is a new modern technique for the integration of geometrical modeling within CAD systems with engineering computations performed in computer-aided engineering (CAE) systems. The IGA method utilizes B-splines or their rationalized version (NURBS [31]) for both the descriptions of the problem geometry and the engineering simulations. Isogeometric analysis has multiple applications in shear deformable shell theory [10], phase field modeling [18], phase-separation simulations [21], wind turbine aerodynamics [23], incompressible hyper-elasticity [19], turbulent flow simulations [14], and biomechanics [13, 22]. An alternative approach is to utilize the T-spline basis functions [8, 9]; however, we focus on B-splines and NURBS in this paper.

Refined isogeometric analysis (rIGA) [20] is a new method for solving numerical problems. It enriches the standard IGA basis functions. Namely, it adds C^0 separators between selected patches of elements. Experiments show that this results in the best sparsity pattern of a matrix structure, which in turn speeds up the direct solvers.

In this paper, we present a hybrid parallelization of an algorithm for the generation of element matrices for the rIGA method [20]. This work is an extension of the distributed memory rIGA described in [30, 33]. We have parallelized the integration routines of the three-dimensional rIGA code. We are aware of other parallel FEM packages – some of which supporting adaptive computations for IGA (including PETIGA [17], a part of PETSc) [4–6]. PETIGA supports the MPI-enabled version of the quadratures algorithm – namely, the Gauss Legendre and Gauss Lobatto rules. The hybrid parallelization can be applied to speed up sequential IGA solvers [15], distributed memory IGA solvers [37], or shared-memory IGA solvers [35]. The hybrid parallelization can be applied for both IGA and rIGA codes. Additionally, when performing the stabilization of isogeometric finite element method codes by using the residual minimization method (iGRM) [24, 25], we deal with a saddle point problem formulation (Equation 1), with the Gram matrix G constrained by the problem matrix B. Both matrices have to be factorized; the presented methodology can also be applied there. Thus, reducing the factorization costs for iGRM matrices is our motivation here.

$$\begin{bmatrix} G & B \\ B^T & 0 \end{bmatrix} \begin{vmatrix} r \\ u \end{vmatrix} = \begin{vmatrix} l \\ 0 \end{vmatrix} \tag{1}$$

There are some alternative sequential algorithms for the fast integration of IGA--FEM [7, 12]. However, all of them employ integration with some number of quadrature points per the elements at some point; thus, the methodology presented here can also be applied for these alternative integration schemes. Moreover, these alternative schemes require some regular structures of the integrated functions on the left-hand and right-hand sides. Our methodology can be applied for arbitrary forcing functions, even such for which the alternative fast integrations are not possible to perform.

2. Model problem

In Figure 1, we present the basis functions that are utilized in the exemplary one-dimensional IGA-FEM and rIGA-FEM setup. The idea of the rIGA method [20] is the following: it introduces C^0 separators between patches of elements. These increase the sparsity of the global matrix by reducing the overlap between the element matrices. The two borderline cases are the following: 1) if the C^0 separators are present between each pair of elements, we end up with Lagrange polynomials (standard FEM) where the matrix is largest but sparsest; and 2) if the C^0 separators are removed, the matrix is the smallest yet the densest. The rIGA matrices are a compromise between the sparsity and dimension of the global matrix, which, in conjunction with the direct solvers, provide the optimal computational cost [20]. In our computations in Section 4, we use 3D rIGA with the optimal placement of C^0 separators, which we learned from [20].

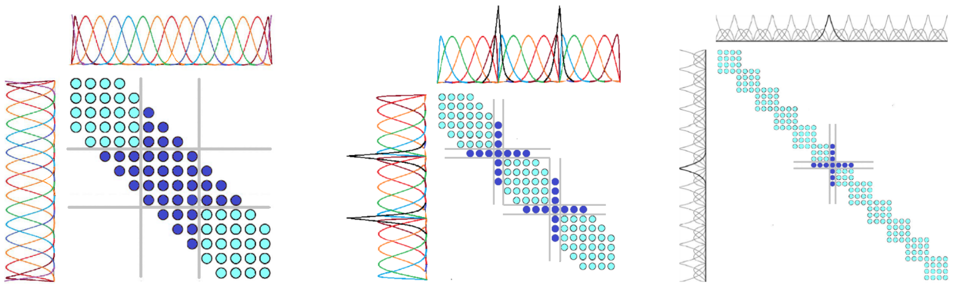

Figure 1. Comparison of basis functions and global matrices utilized in exemplary one-dimensional IGA, rIGA, and FEM setup

Let us focus on three-dimensional computation over the regular three-dimensional patch of elements. The basis functions are defined as the tensor products of one-dimensional basis functions.

Let us consider a stationary elliptic problem in Sobolev space:

$$H_0^1(\Omega) = \{u \in L^2(\Omega) : D^\alpha u \in L^2\Omega,\ |\alpha| \leq 1,\ \operatorname{tr} u = 0 \text{ on } \partial\Omega\} \tag{2}$$

we introduce the classical weak formulation of the Poisson problem in $H_0^1(\Omega)$. We seek $u \in H_0^1(\Omega)$:

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x, \quad \forall v \in H_0^1(\Omega) \tag{3}$$

We may also express the above problem with abstract notation:

$$b(u,v) = l(v): \qquad b(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \qquad l(v) = \int_\Omega f v \,\mathrm{d}x \tag{4}$$

We project the weak problem into finite dimensional subspace $V_h \subset H_0^1(\Omega)$:

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, \mathrm{d}x = \int_\Omega f v_h \, \mathrm{d}x, \quad \forall v_h \in V_h \subset H_0^1(\Omega) \tag{5}$$

We utilize B-spline basis functions with C^0 separators to discretize the problem above.

When using a multi-frontal solver for IGA-FEM or rIGA-FEM, we must construct a relatively large sparse linear system. A frontal solver is a variant of Gaussian elimination that can avoid most operations that involve zero terms. A multi-frontal solver is an improvement on a frontal solver that enables parallel computing. An alternating direction solver works with an IGA-FEM or rIGA-FEM setup that possesses a Kronecker product structure. Instead of a large sparse linear system, we can solve three subsequent 1D small sparse linear systems (in 3D).

3. Parallel OpenMP implementation

The standard algorithm for integration and aggregation in all of the three mentioned cases (FEM, IGA-FEM, and rIGA-FEM) is identical. In general, the generation of the matrices for finite element method computations involves nested loops (starting from the elements) and Gauss integration points through the test basis functions and to the trial basis functions. Our parallelization of the integration process is based on the decomposition of loops concerning the local basis functions and Gaussian quadrature points. Below, we present the OpenMP pseudo-code algorithm that is responsible for the integration of the element matrices in all of the mentioned FEM, IGA, and rIGA setups for both the multi-frontal and alternating direction solvers. For optimal parallel performance, we aggregated multiple loops into a single one.

3.1. Multi-frontal

For use with the multi-frontal solver, we compute the global mass matrix and the global right-hand-side vector in the following subroutine. All of the computations are performed in the hybrid memory model with MPI domain decomposition and OpenMP loop parallelization. We do compute the local element matrices in a fully independent parallel mode. We aggregate the dimensional variables into small matrices. After the parallel part of the computations, all values are moved to a shared global sparse matrix in a critical synchronized part where only one thread can enter.

```
subroutine integrate
  use omp_lib

! element arrays
  real (kind = 8), dimension(:,:,:,:,:,:), allocatable :: elarr
  real (kind = 8), dimension(:,:,:,:,:,:), allocatable :: elrhs
  allocate(elarr(0:px,0:py,0:pz,0:px,0:py,0:pz))
  allocate(elrhs(0:px,0:py,0:pz))

! lnelem[x,y,z], mine[x,y,z], maxe[x,y,z] - range of elements &
! associated with basis functions assigned to this MPI process
  total_size = lnelemx * lnelemy * lnelemz

! common parallel loop over all elements
!$OMP PARALLEL DO &
!$OMP DEFAULT (SHARED) &
!$OMP PRIVATE(ex,ey,ez,ix,e,elarr,elrhs,kx,ky,kz,k,J) &
!$OMP PRIVATE(W,ax,ay,az,a,ax1,ay1,az1,a1,resvalue)
  do all = 1,total_size
! map all to element coefficients - ex, ey, ez
    ez = modulo(all-1,lnelemz)
    ix = (all-ez)/lnelemz+1
    ey = modulo(ix-1,lnelemy)
    ex = (ix-ey)/lnelemy+1
! fix distributed part
! mine - range of elements associated with basis functions &
! assigned to this process
    ex = ex + minex
    ey = ey + miney
    ez = ez + minez
    e = (/ex,ey,ez/)
! reset local element arrays
    elarr = 0.d0
    elrhs = 0.d0
! Jacobian
    J = Jx(ex)*Jy(ey)*Jz(ez)
! ng[x,y,z] - number of quadrature points
! loop over quadrature points
    do kx = 1,ngx
      do ky = 1,ngy
        do kz = 1,ngz
          k = (/kx,ky,kz/)
! weigths
          W = Wx(kx)*Wy(ky)*Wz(kz)*J
! loop over degrees of freedom
! loop over test functions over element
          do ax = 0,px
            do ay = 0,py
              do az = 0,pz
                a = (/ax,ay,az/)
```

```
! compute value for RHS
! NN[x,y,z] - values of (p+1) nonzero basis functions &
! at points of Gauss quadrature
                resvalue = J * W * NNx(ax,kx,ex)* &
                  NNy(ay,ky,ey)* &
                  NNz(az,kz,ez)* &
                  RHS_fun(e,a,k)
! enter computed value to local array
                elrhs(ax,ay,az) = elrhs(ax,ay,az) + resvalue
! loop over trial functions over element
                do ax1 = 0,px
                  do ay1 = 0,py
                    do az1 = 0,pz
                      a1 = (/ax1,ay1,az1/)
! compute value for Mass Matrix
! NN[x,y,z] - values of (p+1) nonzero basis functions &
! at points of Gauss quadrature
                      resvalue = W * NNx(ax,kx,ex)* &
                        NNy(ay,ky,ey)* &
                        NNz(az,kz,ez)* &
                        NNx(ax1,kx,ex)* &
                        NNy(ay1,ky,ey)* &
                        NNz(az1,kz,ez)
! enter computed value to local array
                      elarr(ax,ay,az,ax1,ay1,az1) = &
                        elarr(ax,ay,az,ax1,ay1,az1) + resvalue
                    enddo
                  enddo
                enddo
              enddo
            enddo
          enddo
        enddo
      enddo
    enddo
!$OMP CRITICAL
! enter local element matrix into global sparse matrix here
! only one thread can enter - we are using shared global matrix
    do ax = 0,px
      do ay = 0,py
        do az = 0,pz
          a = (/ax,ay,az/)
          call enter_local_rhs_2_global(ellrhs ,a,e,rhs)
          do ax1 = 0,px
            do ay1 = 0,py
              do az1 = 0,pz
                a1 = (/ax1,ay1,az1/)
                call enter_local_mtrx_2_global(elarr,a,a1,e,
                                                   Mass_mtrx)
              enddo
            enddo
          enddo
        enddo
      enddo
    enddo
  enddo
!$OMP END PARALLEL DO

end subroutine integrate
```

3.2. Alternating directions

For use with the alternating direction solver, we first compute a set of three different mass matrices along p_x, p_y, and p_z. Since the communication costs would become dominant, we use only OpenMP loop parallelization and repeat the computations on each MPI process. We utilize a thread safe sparse matrix data structure.

```
subroutine integrate_Mass_Matrix
  use omp_lib
! lnelem[x,y,z], mine[x,y,z], maxe[x,y,z] - range of elements &
! associated with basis functions assigned to this MPI process
! ng[x,y,z] - number of quadrature points
  total_size = (nelem) * (ng) * (p + 1)*(p + 1)

! common parallel loop over elements, &
! shape functions, and quadrature points
!$OMP PARALLEL DO &
!$OMP DEFAULT(SHARED) &
!$OMP PRIVATE(c,d,e,i,tmp,val)
  do all = 1,total_size
! loop over shape functions over elements (p1+1 functions)
    d = modulo(all - 1, p + 1)
    tmp = (all - d) / (p + 1)
! loop over shape functions over elements (p1+1 functions)
    c = modulo(tmp, p + 1)
    tmp = (tmp - c) / (p + 1)
! loop over Gauss points
```

```
    i = modulo(tmp, ng) + 1
! loop over elements
    e = (tmp - i + 1) / (ng) + 1
! NN[x,y,z] - values of (p+1) nonzero basis functions &
! at points of Gauss quadrature
! W(i) weight for Gauss point i
! J(e) jacobian for element e
! compute value
    val = NN(c, i, e) * NN(d, i, e) * J(e) * W(i)
    call add_to_sparse(sprsmtrx,c,d,e,val)
  enddo
!$OMP END PARALLEL DO

end subroutine integrate_Mass_Matrix
```

We compute the vector of the right-hand-side vectors in the following subroutine. All of the computations are performed in the hybrid memory model with MPI domain decomposition and OpenMP loop parallelization. We do compute the local element matrices in a fully independent parallel mode. We aggregate the dimensional variables into small matrices. After the parallel part of the computations, all of the values are moved to the shared global sparse matrix in a critical synchronized part where only one thread can enter.

```
subroutine integrate_RHS
  use omp_lib
! element
  real (kind = 8), dimension(:,:,:,:,:,:), allocatable :: elrhs
  allocate(elrhs(0:px,0:py,0:pz))

! lnelem[x,y,z], mine[x,y,z], maxe[x,y,z] - range of elements &
! associated with basis functions assigned to this MPI process
  total_size = lnelemx * lnelemy * lnelemz

! common parallel loop over all elements
!$OMP PARALLEL DO &
!$OMP DEFAULT(SHARED) &
!$OMP PRIVATE(tmp,ex,ey,ez,e,kx,ky,kz,k,J}&
!$OMP PRIVATE(W,ax,ay,az,a,resvalue,elarr)
  do all=1,total_size
! map all to element coefficients - ex, ey, ez
    ez=modulo(all-1,lnelemz)
    ix=(all-ez)/lnelemz+1
    ey=modulo(ix-1,lnelemy)
    ex=(ix-ey)/lnelemy+1
```

```
! fix distributed part
! mine - range of elements associated with basis functions &
! assigned to this process
    ex = ex + minex
    ey = ey + miney
    ez = ez + minez
    e = (/ex,ey,ez/)
! reset local element array
    elrhs = 0.d0
! Jacobian
    J = Jx(ex)*Jy(ey)*Jz(ez)
! ng[x,y,z] - number of quadrature points
! loop over quadrature points
    do kx = 1,ngx
      do ky = 1,ngy
        do kz = 1,ngz
          k = (/kx,ky,kz/)
! weigths
          W = Wx(kx)*Wy(ky)*Wz(kz)*J
! loop over trial functions over element
          do ax = 0,px
            do ay = 0,py
              do az = 0,pz
                a = (/ ax, ay, az /)
! compute value for RHS
! NN[x,y,z] - values of (p+1) nonzero basis functions &
! at points of Gauss quadrature
                resvalue = J * W * NNx(ax,kx,ex)* &
                  NNy(ay,ky,ey)* &
                  NNz(az,kz,ez)* &
                  RHS_fun(e,a,k)
! enter computed value to local array
                elrhs(ax,ay,az) = elrhs(ax,ay,az) + resvalue
              enddo
            enddo
          enddo
        enddo
      enddo
    enddo
! enter local element matrix into global matrix here
! only one thread can enter - we are using shared global matrix
!$OMP CRITICAL
    do ax = 0,px
```

```
      do ay = 0,py
        do az = 0,pz
          a = (/ax,ay,az/)
          call enter_local_rhs_2_global(ellrhs,a,e,rhs)
        enddo
      enddo
    enddo
!$OMP END CRITICAL
  enddo
!$OMP END PARALLEL DO

end subroutine integrate_RHS
```

4. Scalability of parallel integration

In this section, we present the scalability of the parallel integration using a single Linux cluster node. Namely, the numerical experiments were performed on the shared-memory node with four Intel R XeonR CPU E7-4860 processors, each possessing 10 physical cores (for a total of 40 cores). We utilize quadratic, cubic, and quartic B-splines ($p = \{2, 3, 4\}$) over a patch of $40 \times 40 \times 40$ finite elements. In Figures 2–6, we present strong scalability – with a fixed problem size and a variable number of processors.

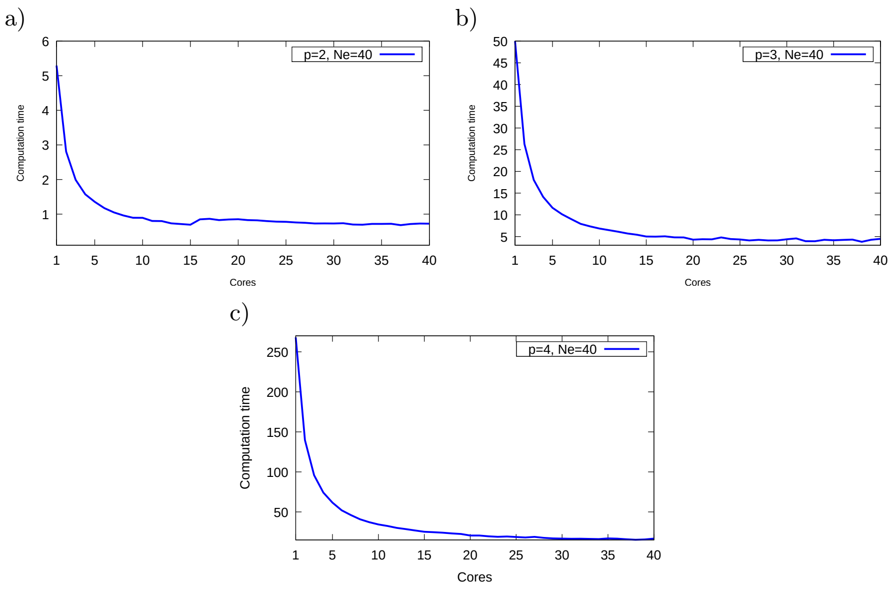

Figure 2. Execution time in seconds of parallel integration algorithm according to increasing number of cores. 3D hexahedral element: a) $p = 2$; b) $p = 3$; c) $p = 4$

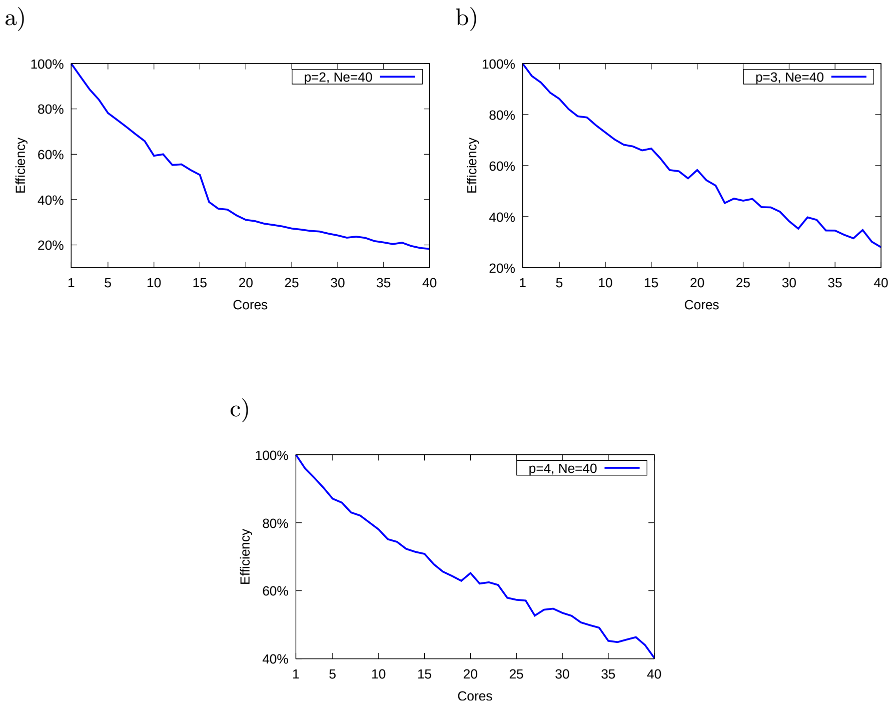

Figure 3. Parallel efficiency of parallel integration algorithm. 3D hexahedral element. Multi-frontal solver: a) $p = 2$; b) $p = 3$; c) $p = 4$

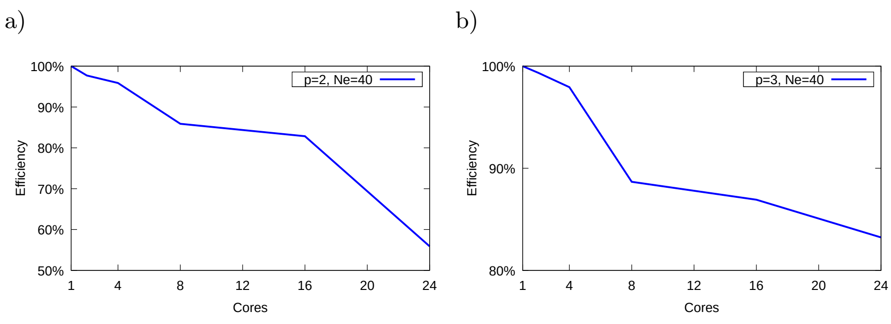

Figure 4. Parallel efficiency of parallel integration algorithm. 3D hexahedral element. Alternating direction solver: a) $p = 2$; b) $p = 3$

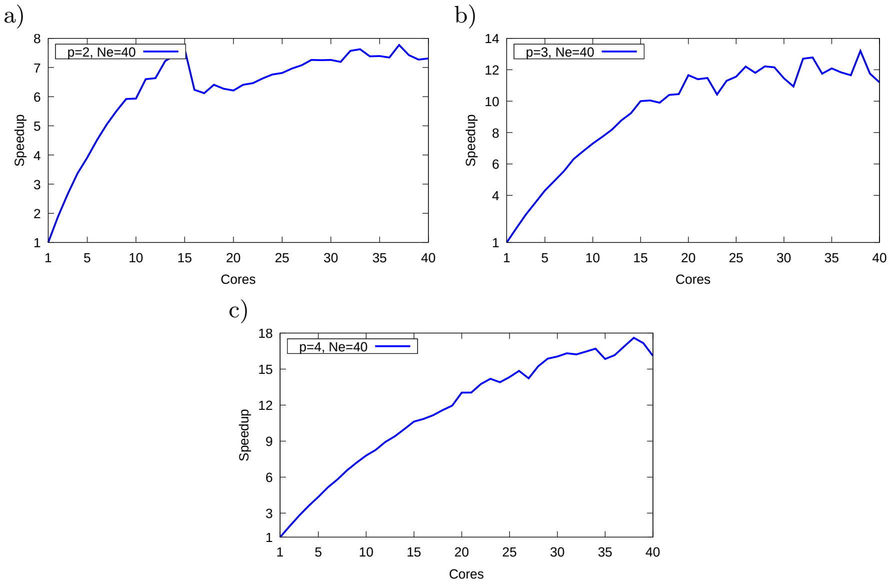

Figure 5. Parallel speedup of parallel integration algorithm. 3D hexahedral element. Multi-frontal solver: a) $p = 2$; b) $p = 3$; c) $p = 4$

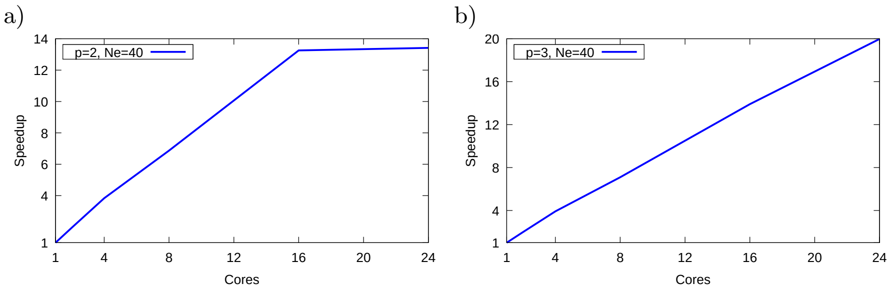

Figure 6. Parallel speedup of parallel integration algorithm. 3D hexahedral element. Alternating direction solver: a) $p = 2$; b) $p = 3$

In Figure 2, we show the measured execution time. In Figures 5 and 6, we present speedup $s = \frac{T_{single}}{T_{nproc}}$. Figures 3 and 4 depict efficiency $e = \frac{speedup}{nproc} \cdot 100\%$. From the presented experiments, it is implied that our OpenMP integration scales well for up to 15 cores for the quadratic B-splines and 20 cores for the cubics; for the quartics, the efficiency grows to 25 cores. A higher number of cores results in an efficiency that is below 60%. Both the MUMPS and ADS cases deal with the same computational problems. Distributed memory MPI-based integration presents linear scalability – the same as that which is presented in [36].

5. Comparison of integration and solution phases for hybrid computations

The generation of a system of linear equations is followed by the factorization phase. Depending on the structure of the matrix and the employed time-integration scheme, we may end up with a matrix that possesses the Kronecker product structure [24, 25]; then, the parallel alternating direction solver can be used (like the one described in [26]). Alternatively, if the matrix possesses a more complicated sparsity structure, a frontal or multi-frontal direct solver is required (like MUMPS). In this section, we compare the parallel integration time with the solution performed by the MUMPS parallel direct solver [1–3] for both the IGA and rIGA phases. For the other (the parallel alternating direction solver), the factorization time is negligible with the integration time [26]. We have executed our experiments on the Prometheus [32] Linux cluster from ACK Cyfronet [11]. In Figure 9, we report the execution time of the parallel MUMPS executed for both the IGA and rIGA cases for a patch of $64 \times 64 \times 64$ elements with quadratic and cubic B-splines as well as for a patch of $40 \times 40 \times 40$ elements for quartic B-splines.

Using PAPI [34], we measured both the time and FLOPS (floating point operation count) for the different parts of solving IGA-FEM problems with different mesh sizes and polynomial orders (namely, for the integration and direct solver parts). These computations were performed by using serial versions of the algorithms. A comparison of the proportions for the different mesh sizes and polynomial orders in cost for the multi-frontal and alternating direction approaches are presented in Figures 7 (FLOPS) and 8 (time). In Figure 7, we present the proportion of the integration of FLOPS to the total computational cost with the MUMPS and ADS solvers, respectively (namely, to determine what part of the total FLOPS. In Figure 8, we present the proportion of the integration time to the total computational cost with the MUMPS and ADS solvers, respectively (namely, to determine part of the total time the integration takes).

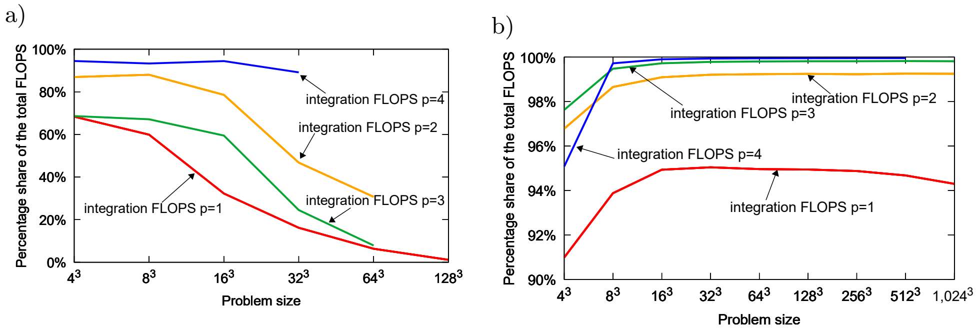

Figure 7. Proportion of integration of FLOPS to total computational cost with: a) MUMPS; b) ADS solvers, respectively

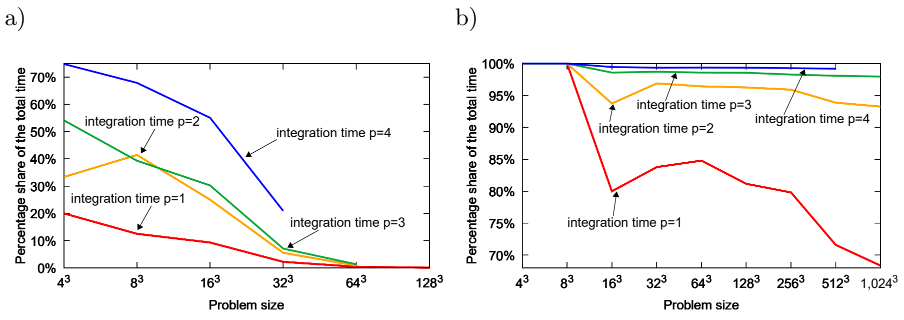

Figure 8. Proportion of integration time to total computational cost with: a) MUMPS; b) ADS solvers, respectively

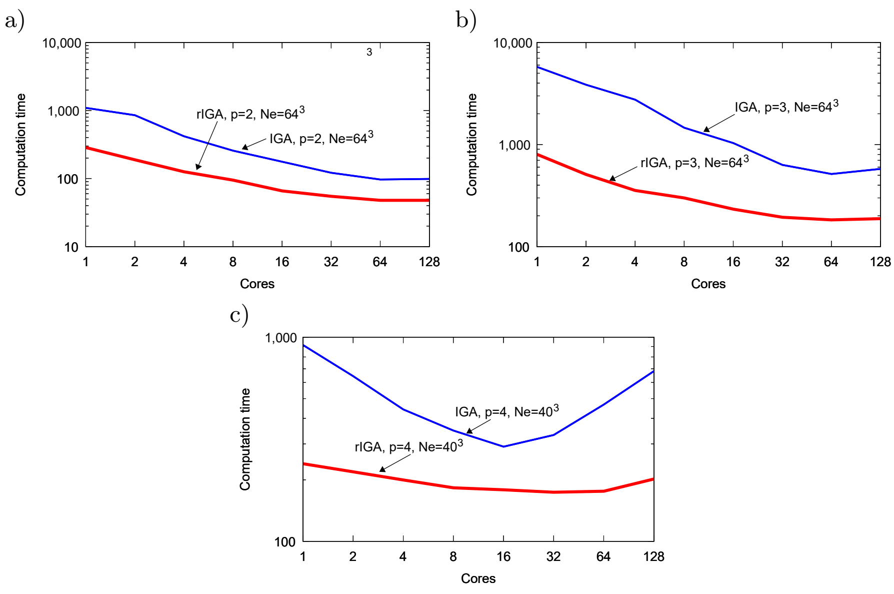

Figure 9. Scalability of parallel MUMPS solver for cubic B-splines for IGA and rIGA computations: a) $p = 2$; b) $p = 3$; c) $p = 4$

First, let us consider the sequential integration and sequential MUMPS solver. The sequential integration takes around 5 seconds for the quadratic B-splines, 50 seconds for the cubic B-splines, and 250 seconds for the quartic B-splines for a patch of $40 \times 40 \times 40$. The solution phase with the MUMPS solver takes 1,000 seconds for the quadratic B-splines over a patch of $64 \times 64 \times 64$ elements (so, the sequential integration would take $5 \cdot 4 = 20$ seconds on the same-sized patch). For the cubics, this takes 6,000 seconds over the same-sized patch (so, the sequential integration would take $50 \cdot 4 = 200$ seconds on the same-sized patch), and for the quartics, it takes

1,000 seconds on the smaller $40 \times 40 \times 40$ patch (and the sequential integration takes 250 seconds here). The sequential integration phase is 2% of the total execution time for the quadratic B-splines, 3% of the total execution time for the cubics, and 25% of the total execution time for the quartics. Thus, for higher-order B-splines, the integration is a significant part of the solution even when we use expensive direct solvers.

Next, we investigate the use of a parallel MUMPS solver with centralized input. In this case, the integration is performed on a single Linux cluster node, and the matrix is distributed internally by the MUMPS solver. The parallel MUMPS solver takes 50 seconds for the quadratic B-splines, 200 seconds for the cubics, and 200 seconds for the quartics when using 16 processors (nodes) – see Figure 9. Performed on the host processor when submitting the matrix to MUMPS, the parallel integration reduces this time down to 1 second for the quadratic B-splines (2% of the total execution time), 5% for the cubics (10% of the total execution time), and 20 seconds for the quartics (30% of the total execution time) using 12 cores. The OpenMP parallelization does not suffice to significantly reduce the integration cost.

Finally, we assume the use of the parallel MUMPS solver with distributed entries. When we apply the domain decomposition paradigm [27, 29], the parallel integration is affected by both the reduction of the number of elements per single processor and the utilization of multiple cores. The resulting hybrid scalability (with increasing numbers of compute nodes and cores per node) of the integration is presented in Figure 10.

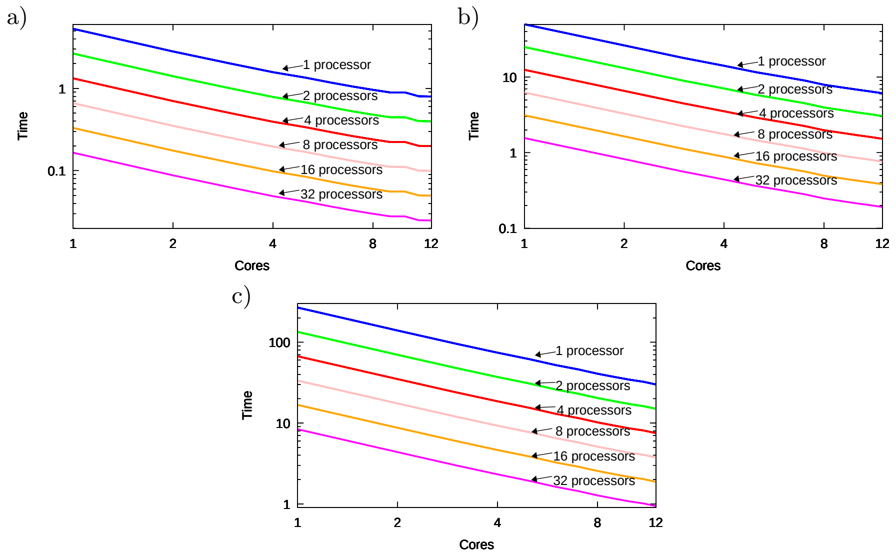

Figure 10. Scalability of hybrid integration for rIGA solver: a) $p = 2$; b) $p = 3$; c) $p = 4$

We conclude that the hybrid parallelization of the rIGA computations with the MUMPS solver reduces the integration below 1% of the solver time. The execution time goes down to 40 seconds for the solver call and to less than 0.1 seconds for the integration phase for the quadratic B-splines. It also goes down to 150 seconds for the solver and to less than 1 second for the integration for the cubics as well as down to 200 seconds for the solver and to less than 1 second for the integration for the quartics.

6. Trace theory description of solver algorithm

In this section, we present a dependency graph that results from a trace theory analysis for the alternating direction solver, while for the multi-frontal solver, the trace theory decomposition is presented in Section 6.3 of [28]. As mentioned before, one of the versions of the hybrid memory parallel direct solver that can be used for rIGA is the alternating direction solver (ADS) [26,36]. This will happen if the matrix that results from the time integration scheme has a Kronecker product structure like the one in [24,25]. In this chapter, we provide an algorithm and its parallel model for a hybrid memory cluster – namely, a cube of processors. In Figure 11, we present a trace theory-based graph of the tasks as well as the dependencies between them.

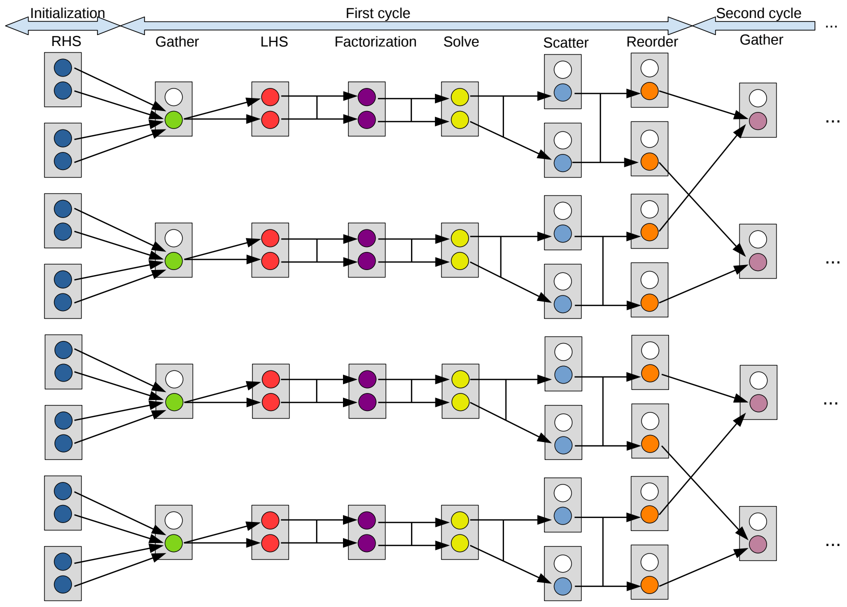

Figure 11. Activity and communication for cluster of $8 = 2 \times 2 \times 2$ nodes with two processors each. Inscriptions above arrows indicate stages, and inscriptions below describe phases. Nodes are marked with rectangles, while circles represent processors. Active and passive processors are represented with colors and white, respectively

The algorithm can be presented in the following steps. Each substep is described in detail below.

1. Initialization
2. **First cycle.** Gather each row of processors into OYZ face of the processors. Solve N_yN_z 1D problems with right-hand side of size N_x. Scatter results and reorder right-hand side.
3. **Second cycle.** Gather each row of processors into OXZ face of the processors. Solve N_xN_z 1D problem with right-hand side of size N_y. Scatter results and reorder right-hand side.
4. **Third cycle.** Gather each row of processors into OXY face of the processors. Solve N_xN_y 1D problem with right-hand side of size N_z. Scatter results and reorder right-hand side.

6.1. Initialization

The initialization consists of dividing a grid into pieces and mapping them onto processors that will integrate them. Due to the lack of dependence between the individual right-hand sides of the equations, hybrid memory parallel integration that utilizes all of the available processors could be used. This uses both MPI (distributed memory) and OpenMP (shared memory).

6.2. Each of three cycles

Each of the three cycles corresponds to a division along one of the directions and consists of six phases:

1. Gather – collecting the results from each row of processors on one of the cube walls. This stage can be imagined as placing the processor cube with one of the walls in front of the observer. In this way, the observer sees only one processor from each row (the one to which the values are transferred). At this stage, only the visible wall of processors will work. This is performed using MPI to increase the amount of data available on one node and to enable left-hand-site calculations.
2. LHS – generation of left-hand side. Each wall processor performs the same calculations, as it is faster than performing calculations on different nodes in parallel and sending out the results. The parallelization of the work on one node is performed by using OpenMP. All of the processors of a given node must exit before the factorization stage begins.
3. Factorization – LU or Cholesky decomposition. Again, redundant calculations are better than using parallelism (this is mainly due to the large amount of time needed to send the results). The parallelization of the work on one node is also performed by using OpenMP. Similar to the previous phase, each processor within the node must complete its work before it starts to solve the system of equations with multiple right-hand-side vectors.

4. Solve – repeatedly applying the factorized left-hand-side vectors to the subsequent right-hand-side vectors (within one node). Again, the parallelism of the work on one node here is performed by using OpenMP. It is very important for all processors to finish before sending the data.
5. Scatter – the distribution of the results to all cube processors. We use MPI to send the data and enable the reorder.
6. Reorder – changing the splitting direction.

The gather and scatter phases were implemented by using MPI, while the LHS, solve, and factorization phases were based on OpenMP.

7. Conclusions

In the case of sequential rIGA computations with a direct solver, the integration phase becomes a significant factor of the solution time; namely, around 25% on moderately sized grids. In this paper, we presented the scalability of parallel rIGA integration as compared to the scalability of the the parallel MUMPS direct solver. The obtained results in the shared memory show good strong scalability for up to 15 cores for quadratic B-splines, up to 20 cores for cubics, and up to 20 cores for quartics. The parallel integrator has been obtained through OpenMP parallelization of the sequential 3D rIGA code. The application of the domain decomposition paradigm and OpenMP parallel implementation reduces the integration cost below 1% of the total execution time. This method can be applied for iGRM simulations – separately for the creation of Gramm and problem matrices.

Acknowledgements

The research presented in this paper was partially supported by National Science Center, Poland – Grant No. 2017/26/M/ST1/00281.

References

[1] Amestoy P.R., Duff I.S., L'Excellent J.Y.: Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, vol. 184, pp. 501–520, 2000.

[2] Amestoy P.R., Duff I.S., L'Excellent J.Y., Koster J.: A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal of Matrix Analysis and Applications*, vol. 23(1), pp. 15–41, 2001.

[3] Amestoy P.R., Guermouche A., L'Excellent J.Y., Pralet S.: Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, vol. 32, pp. 136–156, 2006.

[4] Balay S., Abhyankar S., Adams M.F., Brown J., Brune P., Buschelman K., Dalcin L., Dener A., Eijkhout V., Gropp W.D., Karpeyev D., Kaushik D., Knepley M.G., May D.A., McInnes L.C., Mills R.T., Munson T., Rupp K., Sanan P., Smith B.F., Zampini S., Zhang H., Zhang H.: PETSc Web page, 2019. https://www.mcs.anl.gov/petsc.

[5] Balay S., Abhyankar S., Adams M.F., Brown J., Brune P., Buschelman K., Dalcin L., Dener A., Eijkhout V., Gropp W.D., Karpeyev D., Kaushik D., Knepley M.G., May D.A., McInnes L.C., Mills R.T., Munson T., Rupp K., Sanan P., Smith B., Zampini S., Zhang H., Zhang H.: *PETSc Users Manual*, 2020. https://www.mcs.anl.gov/petsc.

[6] Balay S., Gropp W.D., McInnes L.C., Smith B.F.: Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In: Arge E., Bruaset A.M., Langtangen H.P. (eds.), *Modern Software Tools for Scientific Computing*, pp. 163–202, Springer Science+Business Media, New York, 1997.

[7] Bartoň M., Calo V.: Optimal quadrature rules for isogeometric analysis. 2015. ArXiv:1511.03882.

[8] Bazilevs Y., Calo V.M., Cottrell J.A., Evans J.A., Hughes T.J.R., Lipton S., Scott M.A., Sederberg T.W.: Isogeometric analysis using T-splines, *Computer Methods in Applied Mechanics and Engineering*, vol. 199, pp. 229–263, 2010.

[9] Beirão da Veiga L., Buffa A., Sangalli G., Vázquez R.: Analysis-suitable T-splines of arbitrary degree: definition, linear independence and approximation properties, *Mathematical Models and Methods in Applied Sciences*, vol. 23(11), pp. 1979–2003, 2013.

[10] Benson D.J., Bazilevs Y., Hsu M.C., Hughes T.J.R.: A large-deformation, rotation-free isogeometric shell, *Computer Methods in Applied Mechanics and Engineering*, vol. 200, pp. 1367–1378, 2011.

[11] Bubak M., Kitowski J., Wiatr K. (eds.): *eScience on Distributed Computing Infrastructure: Achievements of PLGrid Plus Domain-Specific Services and Tools*, vol. 8500, Springer, 2014.

[12] Calabrò F., Sangalli G., Tani M.: Fast formation of isogeometric Galerkin matrices by weighted quadrature, *Computer Methods in Applied Mechanics and Engineering*, vol. 316, pp. 606–622, 2017.

[13] Calo V.M., Brasher N.F., Bazilevs Y., Hughes T.J.R.: Multiphysics model for blood flow and drug transport with application to patient-specific coronary artery flow, *Computational Mechanics*, vol. 43, pp. 161–177, 2008.

[14] Chang K., Hughes T.J.R., Calo V.M.: Isogeometric Variational Multiscale Large--Eddy Simulation of Fully-Developed Turbulent Flow Over a Wavy Wall, *Computers and Fluids*, vol. 68, pp. 94–104, 2012.

[15] Collier N., Pardo D., Dalcin L., Paszyński M., Calo V.M.: The cost of continuity: A study of the performance of isogeometric finite elements using direct solvers, *Computer Methods in Applied Mechanics and Engineering*, vol. 213–216, pp. 353–361, 2012.

[16] Cottrell J.A., Hughes T.J.R., Bazilevs Y.: *Isogeometric Analysis: Toward Integration of CAD and FEA*, 2009.

[17] Dalcin L., Collier N., Vignal P., Côrtes A.M.A., Calo V.M.: PetIGA: A framework for high-performance isogeometric analysis, *Computer Methods in Applied Mechanics and Engineering*, vol. 308, pp. 151–181, 2016.

[18] Dedè L., Borden M.J., Hughes T.J.R.: Isogeometric Analysis for Topology Optimization with a Phase Field Model, *Archives of Computational Methods in Engineering*, vol. 19, pp. 427–465, 2012.

[19] Duddu R., Lavier L.L., Hughes T.J.R., Calo V.M.: A finite strain Eulerian formulation for compressible and nearly incompressible hyperelasticity using high-order B-spline finite elements, *International Journal of Numerical Methods in Engineering*, vol. 89, pp. 762–785, 2012.

[20] Garcia D., Pardo D., Dalcin L., Paszyński M., Collier N., Calo V.M.: The Value of Continuity: Refined Isogeometric Analysis and Fast Direct Solvers, *Computer Methods in Applied Mechanics and Engineering*, vol. 316, pp. 586–605, 2017, https://doi.org/10.1016/j.cma.2016.08.017.

[21] Gómez H., Hughes T.J.R., Nogueira X., Calo V.M.: Isogeometric analysis of the isothermal Navier–Stokes–Korteweg equations, *Computer Methods in Applied Mechanics and Engineering*, vol. 199, pp. 1828–1840, 2010.

[22] Hossain S.S., Hossainy S.F.A., Bazilevs Y., Calo V.M., Hughes T.J.R.: Mathematical modeling of coupled drug and drug-encapsulated nanoparticle transport in patient-specific coronary artery walls, *Computational Mechanics*, vol. 49, 2012. https://doi.org/10.1007/s00466-011-0633-2.

[23] Hsu M.-C., Akkerman I., Bazilevs Y.: High-performance computing of wind turbine aerodynamics using isogeometric analysis, *Computers and Fluids*, vol. 49, pp. 93–100, 2011.

[24] Łoś M., Muñoz-Matute J., Muga I., Paszyński M.: Isogeometric Residual Minimization Method (iGRM) for Stokes and Time-Dependent Stokes Problems. ArXiv:2001.00178 [math.NA].

[25] Łoś M., Muñoz-Matute J., Muga I., Paszyński M.: Isogeometric Residual Minimization Method (iGRM) with direction splitting for non-stationary advection-diffusion problems, *Computers & Mathematics with Applications*, vol. 79, pp. 213–229, 2020.

[26] Łoś M., Woźniak M., Paszyński M., Lenharth A., Hassaan M.A., Pingali K.: IGA-ADS: Isogeometric analysis FEM using ADS solver, *Computer Physics Communications*, vol. 217, pp. 99–116, 2017.

[27] Paszyński M.: On the Parallelization of Self-Adaptive hp-Finite Element Methods Part I. Composite Programmable Graph Grammar Model, *Fundamenta Informaticae*, vol. 93(4), pp. 411–434, 2009.

[28] Paszyński M.: *Fast Solvers for Mesh-Based Computations*, CRC Press, Taylor & Francis, 2016.

[29] Paszyński M., Paszyńska A.: Graph Transformations for Modeling Parallel *hp*-Adaptive Finite Element Method. *Lecture Notes in Computer Science*, vol. 4967, pp. 1313–1322, 2008.

[30] Paszyński M., Siwik L., Woźniak M.: Concurrency of three-dimensional refined isogeometric analysis. *Parallel Computing*, vol. 80, pp. 1–22, 2018.

[31] Piegl L., Tiller W.: *The NURBS Book (Second Edition)*. Springer-Verlag New York, Inc., 1997.

[32] Prometheus. http://www.cyfronet.krakow.pl/computers/15226,artykul,prometheus.html.

[33] Siwik L., Woźniak M., Trujillo V., Pardo D., Calo V.M., Paszyński M.: Parallel Refined Isogeometric Analysis in 3D, *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 1134–1142, 2019.

[34] Terpstra D., Jagode H., You H., Dongarra J.: Collecting Performance Data with PAPI-C. In: Müller M., Resch M., Schulz A., Nagel W. (eds.), *Tools for High Performance Computing 2009*, Springer, Berlin–Heidelberg, pp. 57–173, 2010.

[35] Woźniak M., Kuźnik K., Paszyński M., Calo V.M., Pardo D.: Computational cost estimates for parallel shared memory isogeometric multi-frontal solvers, *Computers & Mathematics with Applications*, vol. 67, pp. 1864–1883, 2014.

[36] Woźniak M., Łoś M., Paszyński M., Dalcin L., Calo V.M.: Parallel Fast Isogeometric Solvers for Explicit Dynamic, *Computing and Informatics*, vol. 36(2), pp. 423–448, 2017.

[37] Woźniak M., Paszyński M., Pardo D., Dalcin L., Calo V.M.: Computational cost of isogeometric multi-frontal solvers on parallel distributed memory machines, *Computers Methods in Applied Mechanics and Engineering*, vol. 284, pp. 971–987, 2015.

Affiliations

Maciej Woźniak
AGH University of Science and Technology, Krakow, Department of Computer Science, Poland, macwozni@agh.edu.pl, ORCID ID: https://orcid.org/0000-0002-5576-5671

Anna Bukowska
AGH University of Science and Technology, Krakow, Department of Computer Science, Poland, bukowska@student.agh.edu.pl, ORCID ID: https://orcid.org/0000-0002-3179-7863

Received: 29.05.2020
Revised: 23.07.2020
Accepted: 24.07.2020