Efficient Implementation of Total FETI Solver for Graphic Processing Units using Schur Complement

Lubomír Říha[1], Tomáš Brzobohatý[1], Alexandros Markopoulos[1], Tomáš Kozubek[1], Ondřej Meca[1]
Olaf Schenk[2], and Wim Vanroose[3]

[1] IT4Innovations National Supercomputing Centre,
17. listopadu 15/2172, Ostrava, Czech Republic
{lubomir.riha,tomas.brzobohaty,alexandros.markopoulos,tomas.kozubek,
ondrej.meca@vsb.cz
[2] Institute of Computational Science, Universita della Svizzera italina,
Via Giuseppe Buffi 13, CH-6900 Lugano, Switzerland
olaf.schenk@usi.ch,
[3] University of Antwerp, Department of Mathematics and Computer Science,
Middelheimlaan 1, B-2020 Antwerp, Belgium
wim.vanroose@uantwerpen.be

Abstract. This paper presents a new approach developed for acceleration of FETI solvers by Graphic Processing Units (GPU) using the Schur complement (SC) technique. By using the SCs FETI solvers can avoid working with sparse Cholesky decomposition of the stiffness matrices. Instead a dense structure in form of SC is computed and used by conjugate gradient (CG) solver. In every iteration of CG solver a forward and backward substitution which are sequential are replaced by highly parallel General Matrix Vector Multiplication (GEMV) routine. This results in 4.1 times speedup when the Tesla K20X GPU accelerator is used and its performance is compared to a single 16-core AMD Opteron 6274 (Interlagos) CPU.
The main bottleneck of this method is computation of the Schur complements of the stiffness matrices. This bottleneck is significantly reduced by using new PARDISO-SC sparse direct solver. This paper also presents the performance evaluation of SC computations for three-dimensional elasticity stiffness matrices.
We present the performance evaluation of the proposed approach using our implementation in the ESPRESO solver package.

Keywords: FETI solver, GPGPU, CUDA, Schur complement, ESPRESO

1 Introduction

The goal of this paper is to describe the acceleration of the Finite Element Tearing and Interconnection (FETI) method using Graphic Processing Units (GPU).

The proposed method is based on our variant of FETI type domain decomposition method called Total FETI (TFETI) [2]. The original FETI method, also called the FETI-1, was originally developed for the numerical solution of large linear systems arising in linearized engineering problems by Farhat and Roux [1]. In FETI methods a body is decomposed into several non-overlapping subdomains and the continuity between the subdomains is enforced by Lagrange multipliers. Using the theory of duality, a smaller and relatively well-conditioned dual problem is derived and efficiently solved by a suitable variant of the conjugate gradient (CG) algorithm.

The original FETI algorithm, where only the favorable distribution of the spectrum of the dual Schur complement (SC) matrix [6] was considered, was efficient only for a small number of subdomains. So it was later extended by introducing a natural coarse problem [6, 7], whose solution was implemented by auxiliary projectors so that the resulting algorithm became in a sense optimal [6, 7]. Even if there are several efficient coarse problem parallelization strategies [8], the size limit is always present.

In the TFETI method [2], the Dirichlet boundary conditions are also enforced by Lagrange multipliers. Hence all subdomain stiffness matrices are singular with a priori known kernels which is a great advantage in the numerical solution. With known kernel basis we can effectively regularize the stiffness matrix [3] and use any standard Cholesky-type decomposition for nonsingular matrices. From the implementation point of view TFETI handles each subdomain in the same way. This significantly simplifies the implementation. The nonzero Dirichlet boundary conditions can be also implemented in a less complicated manner. More details regarding the efficiency and other aspects of the TFETI and other can be found in [2].

The stiffness matrices resulting from the three-dimensional elasticity problems are very sparse and they have to be treated in this fashion during the entire runtime of the solver. The operations on stiffness matrices done by FETI solvers therefore are (i) a factorization during the preprocessing stage —single call, and (ii) a backward and forward substitution (a solve routine) —called in every iteration. The second operation is the most time consuming and therefore this paper presents an approach that accelerates this part. For these two operations our ExaScale PaRallel FETI SOlver (ESPRESO) uses the PARDISO sparse direct solver [9–11].

In order to be able to efficiently utilize the potential of the GPU accelerators the solve routine of the sparse direct solver, which is naturally sequential for single right-hand side, has to be replaced by a more parallel operation. To achieve this we have modified the FETI algorithm so that it uses dense SC instead of sparse matrices. The SC is dense and its dimension is given by the size of a subdomain surface rather than its volume. By using the SC three sparse operations, two sparse matrix vector multiplications (SpMV) with gluing matrix $\mathbf{B}^s$ and one solve routine of the PARDISO, can be replaced by a single dense general matrix vector multiplication (GEMV) with the SC. The GEMV operation is well suited

for GPU accelerators, as it offers coalesced memory access pattern and a high degree of parallelism.

1.1 Total FETI Method

The FETI-1 method [2] is based on the decomposition of the spatial domain into non-overlapping subdomains that are glued by Lagrange multipliers $\boldsymbol{\lambda}$, enforcing arising equality constraints by special projectors. The original FETI-1 method assumes that the boundary subdomains inherit the Dirichlet conditions from the original problem. This means physically that subdomains touching the Dirichlet boundary are fixed while the others remain floating; in linear algebra language this means that corresponding subdomain stiffness matrices are nonsingular and singular, respectively. The basic idea of our TFETI [2] is to keep all the subdomains floating and enforce the Dirichlet boundary conditions by means of a constraint matrix and Lagrange multipliers, similarly to the gluing conditions along subdomain interfaces. This simplifies implementation of the generalized inverse. The key point is that kernels of subdomain stiffness matrices are: (i) known a priori; (ii) have the same dimension; and (iii) can be formed without any computation from mesh data. Furthermore, each local stiffness matrix can be regularized cheaply, and the inverse of the resulting nonsingular matrix is at the same time a generalized inverse of the original singular one [3–5].

Let N_p, N_d, N_n, N_c denote the primal dimension, the dual dimension, the null space dimension, and the number of cores available for our computation. Primal dimension means the number of all DOFs including those arising from duplication on the interfaces. Dual dimension is the total number of all equality constraints. Let us consider a partitioning of the global domain Ω into N_S subdomains $\Omega^s, s = 1, \ldots, N_S$ ($N_S \geq N_c$). To each subdomain Ω^s there corresponds the subdomain stiffness matrix $\mathbf{K}^s$ and the subdomain nodal load vector $\mathbf{f}^s$. Matrix $\mathbf{R}^s$ shall be a matrix whose columns span the nullspace (kernel) of $\mathbf{K}^s$. Let $\mathbf{B}^s$ be a signed boolean matrix defining connectivity of the subdomain s with neighbor subdomains. It also enforces Dirichlet boundary conditions when TFETI is used. Using proposed notation the discretized equilibrium equation for s-th subdomain reads

$$\mathbf{K}^s\mathbf{u}^s = \mathbf{f}^s - (\mathbf{B}^s)^T\boldsymbol{\lambda} \tag{1}$$

where additional part in the RHS $(\mathbf{B}^s)^T\boldsymbol{\lambda}$ reflects the influence of neighboring subdomains trough selected LM defined by Boolean matrix $\mathbf{B}^s$. Objects from all subdomains can be collected:

$$\begin{array}{ll} \mathbf{K} = \mathrm{diag}(\mathbf{K}^1, \ldots, \mathbf{K}^{N_S}) & \in \mathbb{R}^{N_p \times N_p}, \\ \mathbf{R} = \mathrm{diag}(\mathbf{R}^1, \ldots, \mathbf{R}^{N_S}) & \in \mathbb{R}^{N_p \times N_n}, \\ \mathbf{B} = [\mathbf{B}^1, \ldots, \mathbf{B}^{N_S}] & \in \mathbb{R}^{N_d \times N_p}, \\ \mathbf{f} = [(\mathbf{f}^1)^T, \ldots, (\mathbf{f}^{N_S})^T]^T & \in \mathbb{R}^{N_p \times 1}, \\ \mathbf{u} = [(\mathbf{u}^1)^T, \ldots, (\mathbf{u}^{N_S})^T]^T & \in \mathbb{R}^{N_p \times 1}, \\ \boldsymbol{\lambda} & \in \mathbb{R}^{N_d \times 1}, \end{array} \tag{2}$$

then global equilibrium equation is written as

$$\mathbf{K}\mathbf{u} = \mathbf{f} - \mathbf{B}^T\boldsymbol{\lambda} \tag{3}$$

which is completed with following compatibility condition

$$\sum_{i=1}^{N_s} \mathbf{B}^s\mathbf{u}^s = \mathbf{B}\mathbf{u} = \mathbf{o}. \tag{4}$$

Previous equations (3) and (4) can be written together as

$$\mathbf{A}\mathbf{x} = \begin{pmatrix} \mathbf{K} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{O} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{o} \end{pmatrix}. \tag{5}$$

The system (5) is reduced by elimination of primal variables $\mathbf{u}$. From equation (3) we can derive

$$\mathbf{u} = \mathbf{K}^+\left(\mathbf{f} - \mathbf{B}^T\boldsymbol{\lambda}\right) + \mathbf{R}\boldsymbol{\alpha} \tag{6}$$

and combining with (4) we get

$$\mathbf{B}\mathbf{u} = \mathbf{B}\mathbf{K}^+\mathbf{f} - \mathbf{B}\mathbf{K}^+\mathbf{B}^T\boldsymbol{\lambda} + \mathbf{B}\mathbf{R}\boldsymbol{\alpha} = \mathbf{o}. \tag{7}$$

Note, that after elimination of $\mathbf{u}$ a new vector of unknowns $\boldsymbol{\alpha}$ appeared (amplitudes of rigid body motions); therefore an additional equation has to be added.

This additional equation follows immediately from the solvability of the first equation in (3) , i.e.,

$$\mathbf{f} - \mathbf{B}^T\boldsymbol{\lambda} \in \mathit{Image}(\mathbf{K}) \tag{8}$$

which can be expressed equivalently using the matrix $\mathbf{R}$ as follows

$$\mathbf{R}^T\mathbf{K}\mathbf{u} = \mathbf{R}^T\mathbf{f} - \mathbf{R}^T\mathbf{B}^T\boldsymbol{\lambda} = \mathbf{o}. \tag{9}$$

Note that columns of $\mathbf{R}$ also span the kernel of $\mathbf{K}$, thus $\mathbf{R}^T\mathbf{K}\mathbf{u} = \mathbf{o}$. Now both (7) and (9) written together,

$$\begin{pmatrix} \mathbf{B}\mathbf{K}^+\mathbf{B}^T & -\mathbf{B}\mathbf{R} \\ -\mathbf{R}^T\mathbf{B}^T & \mathbf{O} \end{pmatrix} \begin{pmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\alpha} \end{pmatrix} = \begin{pmatrix} \mathbf{B}\mathbf{K}^+\mathbf{f} \\ -\mathbf{R}^T\mathbf{f}^T \end{pmatrix}, \tag{10}$$

define the reduced system. Using following notations,

$$\begin{aligned} \mathbf{F} &= \mathbf{B}\mathbf{K}^+\mathbf{B}^T = \sum_{s=1}^{N_s} \mathbf{F}^s = \sum_{s=1}^{N_s} \mathbf{B}^s(\mathbf{K}^s)^+(\mathbf{B}^s)^T, \\ \mathbf{G} &= -\mathbf{R}^T\mathbf{B}^T, \\ \mathbf{d} &= \mathbf{B}\mathbf{K}^+\mathbf{f} \quad = \sum_{s=1}^{N_s} \mathbf{d}^s = \sum_{s=1}^{N_s} \mathbf{B}^s(\mathbf{K}^s)^+\mathbf{f}^s, \\ \mathbf{e} &= -\mathbf{R}^T\mathbf{f}, \end{aligned} \tag{11}$$

it can be rewritten as

$$\begin{pmatrix} \mathbf{F} & \mathbf{G}^T \\ \mathbf{G} & \mathbf{O} \end{pmatrix} \begin{pmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\alpha} \end{pmatrix} = \begin{pmatrix} \mathbf{d} \\ \mathbf{e} \end{pmatrix}. \tag{12}$$

The obtained matrix

$$\mathbf{F} = -\mathbf{S} = \mathbf{B}\mathbf{K}^{+}\mathbf{B}^{T} \tag{13}$$

is actually the negative SC of the system (5). Compared to the original KKT system (5) the new one (12) has smaller dimension and it is also better conditioned.

The equality constraints $\mathbf{G}\boldsymbol{\lambda} = \mathbf{e}$ in (12) can be homogenized to $\mathbf{G}\bar{\boldsymbol{\lambda}} = \mathbf{o}$ by splitting $\boldsymbol{\lambda}$ into $\bar{\boldsymbol{\lambda}} + \widetilde{\boldsymbol{\lambda}}$, where $\widetilde{\boldsymbol{\lambda}}$ satisfies $\mathbf{G}\widetilde{\boldsymbol{\lambda}} = \mathbf{e}$. This implies $\bar{\boldsymbol{\lambda}} \in \operatorname{Ker}\mathbf{G}$. The vector $\widetilde{\boldsymbol{\lambda}}$ can be chosen as the least squares solution of the equality constraints, i.e. $\widetilde{\boldsymbol{\lambda}} = \mathbf{G}^{T}(\mathbf{G}\mathbf{G}^{T})^{-1}\mathbf{e}$. We substitute $\boldsymbol{\lambda} = \bar{\boldsymbol{\lambda}} + \widetilde{\boldsymbol{\lambda}}$, minimize over $\bar{\boldsymbol{\lambda}}$ (terms without $\bar{\boldsymbol{\lambda}}$ can be omitted) and add $\widetilde{\boldsymbol{\lambda}}$ to $\bar{\boldsymbol{\lambda}}$.

Finally, equality constraints $\mathbf{G}\bar{\boldsymbol{\lambda}} = \mathbf{o}$ can be enforced by the orthogonal projector $\mathbf{P} = \mathbf{I} - \mathbf{Q}$ onto the null space of $\mathbf{G}$, where $\mathbf{Q} = \mathbf{G}^{T}(\mathbf{G}\mathbf{G}^{T})^{-1}\mathbf{G}$ is the orthogonal projector onto the image space of $\mathbf{G}^{T}$ (i.e., $\operatorname{Im}\mathbf{Q} = \operatorname{Im}\mathbf{G}^{T}$ and $\operatorname{Im}\mathbf{P} = \operatorname{Ker}\mathbf{G}$). The final problem reads

$$\mathbf{P}\mathbf{F}\bar{\boldsymbol{\lambda}} = \mathbf{P}\widetilde{\mathbf{d}}, \tag{14}$$

where $\widetilde{\mathbf{d}} = \mathbf{d} - \mathbf{F}\widetilde{\boldsymbol{\lambda}}$. Note that we call the action of $(\mathbf{G}\mathbf{G}^{T})^{-1}$ the *coarse problem* of FETI. Problem (14) can be solved with an arbitrary iterative linear system solver. The conjugate gradient method is a good choice thanks to the classical estimate by Farhat, Mandel and Roux [6] of the spectral condition number

$$\kappa(\mathbf{P}\mathbf{F}\mathbf{P}|\mathrm{Im}\mathbf{P}) \leq C\frac{H}{h}. \tag{15}$$

2 Stiffness Matrix Preprocessing and Schur Complement Computation

The FETI solver processing can be divided into two stages: (i) the preprocessing and (ii) the solver runtime. During the preprocessing stage the most time consuming tasks include assembling the distributed inverse matrix of coarse problem $(\mathbf{G}\mathbf{G}^{T})^{-1}$ and factorization of the subdomain stiffness matrices $\mathbf{K}^{s}$. While the coarse problem processing time is mainly given by the number of subdomains, the $\mathbf{K}^{s}$ factorization times are given by the subdomain sizes. The solver runtime stage executes the CG iterative solver and its execution time is therefore given by the number of iterations and the iteration time. If the iteration time is evaluated then the most time consuming part is executing the solve routine (forward and backward substitution) of the sparse direct solver using the Cholesky decomposition of the $\mathbf{K}^{s}$. This operation takes up to 95% of the iteration time. The main focus of this paper is to describe a method to accelerate this operation using GPU accelerators.

The sparse data structures cannot take full advantage of modern processing hardware such as GPU accelerators which is equipped with Single Instruction Multiple Data (SIMD) units. To reach the full potential of a GPU architecture a dense representation of the data that were originally sparse is the key task. In

case of FETI solvers the data that has to be converted to dense structure is the Cholesky decomposition of stiffness matrices $\mathbf{K}^s$.

In the FETI solver prior to calling the solve routine, the vector of Langrange multipliers (the dual variables $\bar{\boldsymbol{\lambda}}$) has to be converted to primal variables using gluing matrix $\mathbf{B}^T$. The result of the solve routine (primal variables) have to be again converted into dual variables using $\mathbf{B}$. This operation in the algorithm of the projected preconditioned conjugate gradient method appears during the initialization stage, and then once in each iteration (in Algorithm 1 the appearances

Data: set $\bar{\boldsymbol{\lambda}} = \mathbf{P}\bar{\boldsymbol{\lambda}}_0,\ \bar{\boldsymbol{\lambda}}_0 \in \mathbb{R}^n,\ \varepsilon > 0,\ i_{max} > 0$
$\mathbf{M} \approx (\mathbf{PFP})^{-1}$
$\mathbf{g} = \mathbf{F}\bar{\boldsymbol{\lambda}} - \tilde{\mathbf{d}}\ \ (\star)$
$\mathbf{w} = \mathbf{PMPg}$
for $i = 0, 1, \cdots, i_{max}$ **do**
  $\mathbf{v} = \mathbf{Fp}\ \ (\star)$
  $\mathbf{w}_{prev} = \mathbf{w}$
  $\mathbf{w} = \mathbf{PMPg}$
  $\rho = -(\mathbf{g}, \mathbf{w})/(\mathbf{p}, \mathbf{v})$
  $\bar{\boldsymbol{\lambda}} = \bar{\boldsymbol{\lambda}} + \mathbf{p}\rho$
  $\mathbf{g}_{prev} = \mathbf{g}$
  $\mathbf{g} = \mathbf{g} + \mathbf{v}\rho$
  if $\sqrt{(\mathbf{g}, \mathbf{Pg})} < \varepsilon$ **then**
    break
  end
  $\beta = (\mathbf{g}, \mathbf{w})/(\mathbf{g}_{prev}, \mathbf{w}_{prev})$
  $\mathbf{p} = \mathbf{w} + \mathbf{p}\beta$
end
$\boldsymbol{\lambda} = \tilde{\boldsymbol{\lambda}} + \bar{\boldsymbol{\lambda}}$

Algorithm 1: The projected preconditioned conjugate gradient method.

marked by the symbol $\star$). This step

$$\mathbf{v} = \sum_{s=1}^{N_s} \mathbf{B}^s (\mathbf{K}^s)^+ (\mathbf{B}^s)^T \mathbf{p}, \tag{16}$$

which in regular CG is equal to matrix-vector multiplication with the system matrix, contains two calls of sparse matrix vector multiplications of sparse BLAS (SpMV with $\mathbf{B}^s$) and one call of PARDISO solve routine (action of generalized inverse $(\mathbf{K})^+$). Instead of executing these three operations on sparse matrices, we can directly assemble the sth contribution $\mathbf{F}^s$ to the global FETI operator $\mathbf{F}$ from

$$\mathbf{A}^s = \begin{pmatrix} \mathbf{K}^s & (\mathbf{B}^s)^T \\ \mathbf{B}^s & \mathbf{0} \end{pmatrix}. \tag{17}$$

We evaluated two approaches how to assemble the matrix $\mathbf{F}^s$. The first approach is by solving the systems $\mathbf{K}^s\mathbf{X}^s = (\mathbf{B}^s)^T$ and then calculating $\mathbf{F}^s = \mathbf{B}^s\mathbf{X}^s$.

The second and more efficient approach is using the incomplete factorization method implemented in PARDISO-SC applied to the system in (17) where $\mathbf{F}^s$ will sit in the block (2,2) if the factorization is stopped after the elimination of the (1,1) block of the system in (17). This method is approximately 3.3 to 4.5 times faster depending on system size; see Table 2. The same table also shows that assembling $\mathbf{F}^s$ is slower than factorization of the matrix $\mathbf{K}^s$ by 13.9 to 18.6 times. It is therefore important to evaluate the trade off between preprocessing time and solver runtime.

3 GPU Acceleration of the FETI Iterative Solver

In general, one of the most limiting factors of GPU acceleration is data transfers between the CPU and GPU main memory over the PCI-Express bus. In our approach the calculation of the Schur complements $\mathbf{F}^s$ is approximately 20 to 60 times slower than the data transfers to GPU if data is transferred only once during the preprocessing stage. However it is not possible to maintain high performance if solver needs to transfer $\mathbf{F}^s$ matrices back and forth between CPU and GPU memory in every iteration. Therefore the size of the GPU global memory is limiting the maximum problem size that can be processed per accelerator.

In case of Tesla K20X with 6 GB of memory, it is able to solve problem of size approximately 0.5 million of DOFs if the $\mathbf{F}^s$ matrices are stored as general dense matrices. This size depends on the surface-to-volume ratio of the subdomains and hence the partitioning. This particular case is for cubic subdomains. For decomposition into different number of cubic subdomains see Table 2. But, since the $\mathbf{K}^s$ is symmetric, the $\mathbf{F}^s$ is also symmetric and it can be stored using the packed format. In this case the problem size solved by one GPU is two times larger.

Since all the $\mathbf{F}^s$ matrices are transferred during the preprocessing only input and output vectors are transferred per iteration. To be able to hide these transfers behind the execution of the GEMV kernel on GPU each subdomain uses its own CUDA streams. Streams are initialized during the preprocessing prior to transfer the $\mathbf{F}^s$ to GPU. Then in every iteration the ith stream transfers the input vector, executes cuBLAS GEMV kernel, and transfers output vector back to CPU memory, where i is the subdomain index. All streams are eventually synchronized to ensure that all output vectors are successfully transferred back to the CPU memory. Then the CG algorithm can continue.

CuBLAS provides three matrix-vector multiplication routines that can be used by our approach: (1) GEMV - general matrix-vector multiplication with storage requirements n^2, (2) SYMV - symmetric matrix-vector multiplication with storage requirements n^2 and (3) SPMV - symmetric packed matrix-vector multiplication with storage requirements $n(n+1)/2$. While the last routine is optimal due to reduced memory usage its performance is significantly lower. We have evaluated the performance of all three routines using cuBLAS version 6.5. The fastest kernel is GEMV. The SYMV kernel performance is lower by 5% (it

is based on implementation in KBLAS). The slowest performance was achieved by SPMV with packed format which is up to 3 times slower then GEMV.

Therefore the optimal choice is to use the SYMV kernel and let two subdomains share the memory allocation of size $n(n+2)$, where the first subdomain uses the upper triangular part of the allocation and the second subdomain uses the lower part. This way the GPU memory usage is optimal, and performance penalty is only 5%. This implies a problem with implementation as two subdomains have to share single memory allocation and the sizes of these two subdomains have to be similar to achieve optimal memory usage.

The proposed approach supports the use of preconditioners in the identical way that the original Total FETI method.

4 Acceleration of Computation of Schur Complement by PARDISO-SC

Let us solve the system introduced in (5) in the itemized form

$$\begin{pmatrix} \mathbf{K}^1 & & & (\mathbf{B}^1)^T \\ & \ddots & & \vdots \\ & & \mathbf{K}^{N_S} & (\mathbf{B}^{N_S})^T \\ \mathbf{B}^1 & \dots & \mathbf{B}^{N_S} & \mathbf{O} \end{pmatrix} \begin{pmatrix} \mathbf{u}^1 \\ \vdots \\ \mathbf{u}^{N_S} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{f}^1 \\ \vdots \\ \mathbf{f}^{N_S} \\ \mathbf{o} \end{pmatrix} \tag{18}$$

in which we assume all diagonal blocks $\mathbf{K}^s$ are nonsingular (to utilize PARDISO also for cases where matrices $\mathbf{K}^s$ are singular, a special regularization technique described in [3] can be used). Solution of such a linear system is given by Schur complement method. First, the SC $\mathbf{S}$ is computed,

$$\mathbf{S} = -\mathbf{F} = \sum_{s=1}^{N_S} \mathbf{S}^s, \tag{19}$$

where

$$\mathbf{S}^s = -\mathbf{B}^s(\mathbf{K}^s)^{-1}(\mathbf{B}^s)^T \tag{20}$$

and then the first-stage part of the solution of the system (5) solved with FETI operator $\mathbf{F}$ instead of $\mathbf{S}$ is obtained by solving

$$\mathbf{F}\boldsymbol{\lambda} = \sum_{s=1}^{N} \mathbf{B}^s(\mathbf{K}^s)^{-1}\mathbf{f}^s. \tag{21}$$

Finally, the second-stage part of the solution can be obtained for all

$$\mathbf{K}^s\mathbf{u}^s = \mathbf{f}^s - (\mathbf{B}^s)^T\boldsymbol{\lambda}. \tag{22}$$

A quick look at (20)–(22) reveals great scope for parallelism. More specifically, the computation of the contributions $\mathbf{B}^s(\mathbf{K}^s)^{-1}(\mathbf{B}^s)^T$ to the SC, the evaluation

of the residual in (21), and the solutions $\mathbf{u}^i$, $s = 1, \ldots, N_S$, can be performed independently.

As we previously mentioned, the data are sparse and unstructured in the case of our application; therefore the computation of $\mathbf{B}^s(\mathbf{K}^s)^{-1}(\mathbf{B}^s)^T$ needs to rely on *sparse* linear algebra kernels. Previously, we used off-the-shelf sparse linear solvers such as PARDISO to first factorize $\mathbf{K}^s$ as $\mathbf{L}^s\mathbf{D}^s(\mathbf{L}^s)^T$, then perform triangular solves with the factors of $\mathbf{K}^s$ for each nonzero column of $(\mathbf{B}^s)^T$, i.e., compute $(\mathbf{K}^s)^{-1}(\mathbf{B}^s)^T$, and, finally, multiply the result from the left with $\mathbf{B}^s$. This approach has two important drawbacks in multicore environments: (i) the triangular solves with $\mathbf{L}^s$ and $(\mathbf{L}^s)^T$ do not scale well with the number of cores [13], being memory bound, and (ii) the sparsity of the columns of $(\mathbf{B}^s)^T$ may not be exploited by some sparse direct linear solvers when solving $\mathbf{L}^s\mathbf{X} = (\mathbf{B}^s)^T$. This feature has been for instance added to the recent version of the open-source MUMPS direct solver.

To address these limitations the computations were revisited and new approach was proposed in [12]. This new approach computes $\mathbf{B}^s(\mathbf{K}^s)^{-1}(\mathbf{B}^s)^T$ from a partial sparse Bunch–Kaufman factorization of the augmented matrix in (17). More specifically, the factorization of $\mathbf{A}^s$ is stopped after pivoting reaches the last diagonal entry of $\mathbf{K}^s$. At this point $-\mathbf{B}^s(\mathbf{K}^s)^{-1}(\mathbf{B}^s)^T$ is computed and resides in the $(2, 2)$ block of $\mathbf{A}^s$.

Traditionally, the factorizing phase has generally required the greatest portion of the total execution time. It typically involves the majority of the floating-point operations, and it is also computation bound. However, in this application, we have to compute $\mathbf{B}^s(\mathbf{K}^s)^{-1}(\mathbf{B}^s)^T$, and memory traffic is the limiting factor. In exploiting the sparsity not only in $\mathbf{K}^s$, but also in $(\mathbf{B}^s)^T$, we (i) significantly reduce the number of floating point operations and (ii) can use in-memory sparse matrix compression technique to reduce the memory traffic on multicore architectures. In the numerical section we refer to this compression techniques as PARDISO-SC, whereas PARDISO is the uncompressed triangular solve based on $\mathbf{K}^s$. As a result, the approach in PARDISO-SC is much better suited for multicore parallelization than the triangular solves and, consequently, the speedup over the previous approach is quite considerable, as we will show in numerical experiments.

4.1 Parallelization Techniques in ESPRESO Solver

The ESPRESO is a FETI-based sparse iterative solver implemented in C++. The solver is developed to support modern heterogeneous accelerators such as Intel Xeon Phi or Nvidia GPU. Due to this fact, the CPU version uses the Intel MKL library and in addition the cuBLAS library is used for GPU acceleration. A significant part of the development effort was devoted to writing a C++ wrapper for (1) the selected sparse and dense BLAS routines of the Intel MKL library and (2) the sparse direct solvers. As of now PARDISO and PARDISO-SC are supported. By simple modification of the wrapper support for additional direct solvers can be added. This is an ongoing work.

The hybrid parallelization inside the ESPRESO is designed to fit two-level decomposition used by the Hybrid FETI Method. This method decomposes the problem into clusters (the first level), then the clusters are further decomposed into subdomains (the second level). In case of the TFETI method the problem is directly decomposed into subdomains and predefined number of subdomains is assigned to a particular compute node. In this paper we focus on the TFETI method only.

In ESPRESO this decomposition is mapped to a parallel hardware in the following way: (level 1) groups of subdomains are mapped to compute nodes, communication between nodes is done using message passing—MPI; and (level 2) subdomains inside nodes are mapped to CPU cores using Cilk++ shared memory model. ESPRESO supports the oversubscription of the CPU cores which means that multiple subdomains are processed by a single CPU core. In a case of the GPU acceleration all subdomains of one node are processed by a single GPU. Then using CUDA streams multiple subdomains can be processed in parallel. Support of multiple GPU accelerators per node is also supported. More about GPU implementation is described in Section 3 .

There are two major parts of the solver that affect its parallel performance and scalability: (i) the communication layer (described in this section) and (ii) the inter node processing routines for shared memory. The second part can be further divided into CPU processing routines and highly computationally intensive routines suitable for offloading to the accelerator.

The first part deals with optimization of the communication overhead caused mainly by multiplication with gluing matrices $\mathbf{B}$, application of the projector (includes multiplication with matrix $\mathbf{G}$ and the application of the coarse problem), and global reduction operation in the CG solver. The distributed memory parallelization is done using MPI 3.0 standard because the communication hiding techniques for an iterative CG solver require the nonblocking MPI collective operations.

The following communication avoiding and hiding techniques for the main CG iterative solver are used: (i) the pipelined conjugate gradient (PipeCG) iterative solver—hides communication cost of the global dot products in CG behind the local matrix vector multiplications; (ii) the coarse problem solver using a distributed inverse matrix—merges two global communication operations (Gather and Scatter) into one (AllGather) and parallelizes the coarse problem processing; and (3) the optimized version of global gluing matrix multiplication ($\mathbf{B}^s$)—implemented as a stencil communication which is fully scalable. The potential of the ESPRESO communication layer is shown in Figure 1.

5 Results

The performance evaluation has been carried out using the Titan machine installed in the Oak Ridge National Lab. Each compute node contains one 16-core 2.2GHz AMD Opteron 6274 (Interlagos) processor, 32 GB of RAM and NVIDIA Tesla K20X accelerator (GPU) with 6 GB of DDR5 memory.

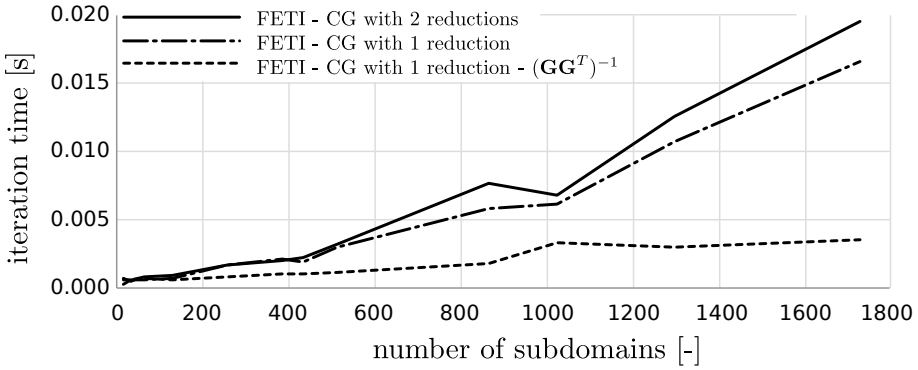

Fig. 1. Performance evaluation of optimization techniques introduced into the communication layer of ESPRESO. (i) CG algorithm with 2 reductions—is the standard version of the CG algorithm. (ii) CG algorithm with 1 reduction —is based on the preconditioned pipelined CG algorithm, where projector is used in place of preconditioner (this algorithm is ready to use nonblocking global reduction for further performance improvements (comes in Intel MPI 5.0)), and (iii) GGTINV—parallelizes the solve of the coarse problem plus merges two Gather and Scatter global operations into single AllGather. Please note that very small subdomains are used (648 DOFs)—small subdomain size is chosen to identify all communication bottlenecks of the solver .

For the evaluation a synthetic 3D cube benchmark is used. The size of the problem is limited by the size of the GPU memory. Tesla K20X has 6 GB of RAM and it can fit problems of size approximately 0.5 million of unknowns if stored as general matrices. Tests were executed for decomposition into 48, 64, 80, 100, 150 and 216 cubical subdomains. For more details regarding the decomposition please see the Table 2.

5.1 Calculation of the Schur Complement using Sparse Direct Solvers

In this section an evaluation of the two methods used to compute the Schur complements $\mathbf{F}^s$ for FETI solver is presented. Two metrics are considered: (i) memory requirements and (ii) processing time to compute the SCs. All results are compared to the original method which uses sparse Cholesky decomposition. This evaluation has been carried out for a cubical subdomains.

In Table 1 the size of the dense $\mathbf{F}^s$ structure is compared to number of nonzero elements in the sparse Cholesky decomposition. This represents the memory requirements of both approaches. It can be observed that if the $\mathbf{F}^s$ is stored in packed format (only upper or lower triangle with the diagonal is kept in memory) its size is lower that number of non zero values in the Cholesky decomposition. If we take into account that the sparse representation of data requires more than 1 integer value (4 bytes) and 1 double value (8 bytes) per element we can see that SC method is more efficient in terms of memory usage.

Table 1. Size comparison of the sparse Cholesky decomposition/factorization using PARDISO sparse direct solver and the dense Schur complement.

Stiffness matrix (**K**) size-sparse format	3993 x 3993	12288 x 12288
SC size: $\mathbf{B}\mathbf{K}^{+}\mathbf{B}^{T}$	1356 x 1356	2931 x 2931
Number of nonzeros in Cholesky factor - using PARDISO sparse direct solver	1,087,101	5,042,903
Number of elements in SC if stored as general dense matrix	1,838,736	8,590,761
Number of elements in SC if stored as symmetric packed dense matrix	920,046	4,296,846

Table 2 compares the stiffness matrix preprocessing times of all three approaches on a problem of size approximately 0.5 million DOFs. The problem is decomposed into various number of subdomains and it can be observed that as the size of the subdomain is getting smaller the calculation of the SC is getting more efficient when compared to the factorization time. See the second values in the PARDISO-SC column which describe how many times calculation of SC is slower than factorization. The advantage of using PARDISO-SC is obvious when processing time is compared to last column of the table which contains computational times of SC using PARDISO. We can see that PARDISO-SC is 3.3 to 4.5 times faster.

Table 2. Performance evaluation of the stiffness matrix preprocessing time on a single node of the Oak Ridge National Lab machine called Titan. The second values in PARDISO-SC column describe how many times it is slower than factorization. The second values in PARDISO column describe how many times it is slower than PARDISO-SC in calculation of the Schur complement.

Problem size [DOFs]	Number of subdomains [-]	Subdomain size [DOFs]	Preprocessing time Factorization	Preprocessing time - Schur complement PARDISO-SC	Preprocessing time - Schur complement PARDISO
513 498	48	12288	4.1 s	76.5 s ; 18.6	256.2 s ; 3.3
419 121	48	10125	3.1 s	56.4 s ; 18.4	233.1 s ; 4.1
446 631	64	8232	2.9 s	49.6 s ; 16.9	210.3 s ; 4.2
439 383	80	6591	2.6 s	42.0 s ; 16.1	190.0 s ; 4.5
423 360	100	5184	2.4 s	34.0 s ; 13.9	144.6 s ; 4.3
475 983	150	3993	2.4 s	34.9 s ; 14.4	134.8 s ; 3.9
499 125	216	3000	2.2 s	30.6 s ; 13.7	112.9 s ; 3.7

5.2 GPU Acceleration of the Conjugate Gradient Solver in Total FETI

The previous section described the additional work that needs to be done in the preprocessing stage. In this section we describe the main performance gain by using the SC in FETI, which is the acceleration of the CG solver.

The results are shown in Table 3. The problem of the same size (0.5 millions of DOFs) and the decomposition into the same number of subdomains as in the previous section is used. It can be seen that GPU acceleration is more efficient for larger subdomains, speedup of 4.1 for subdomain of size 12 288 DOFs and it is getting less efficient for smaller subdomains, speedup of 1.8 for subdomain of size 3000 DOFs. This is an opposite behavior to the preprocessing stage where calculation of the Schur complement is more costly for larger subdomains. The execution time values in Table 3 show the solver runtime for 500 iterations. Based on our measurements this is the sweet spot where GPU acceleration for this hardware configuration becomes more efficient than CPU version.

Table 3. Iterative solver execution time for 500 iterations running on a single Tesla K20X GPU accelerator. Table compares solve routine which includes forward and backward substitution of the PARDISO solver and GEMV (general matrix vector multiplication) routine from the cuBLAS GPU accelerated library.

Problem size [DOFs]	Number of subdomains [-]	Subdomain size [DOFs]	Solver runtime on GPU [s]	Solver runtime on CPU [s]	speedup by GPU [-]
513 498	48	12288	27.76	113.63	4.1
419 121	48	10125	22.99	87.68	3.8
446 631	64	8232	24.46	90.74	3.7
439 383	80	6591	25.33	81.58	3.2
423 360	100	5184	25.94	74.66	2.9
475 983	150	3993	33.03	78.03	2.4
499 125	216	3000	40.29	74.12	1.8

5.3 Overall Performance for Different Classes of Problems in 3D Linear Elasticity

The overall performance which includes preprocessing and solver runtime is shown in Table 4. In the current version the ESPRESO library can generate and solve linear elasticity problems in three dimensions. If we focus on this group of problems, we can evaluate the usability of the approach presented in this paper for three scenarios. The first scenario is a simple linear elasticity problem. In this case the number of iterations is usually less than 100. It is clear that for this types of problems the preprocessing time is dominant and presented approach including GPU acceleration does not bring any advantage. The second scenario is for group of ill-conditioned linear elasticity problems. These problems

approximately converge in several hundreds of iterations. This is the border line situation where assembling Schur complements becomes useful and overall execution time is reduced. The gain for this type of problems is not dramatic. The last group are transient and nonlinear problems with constant tangent stiffness matrix. To solve these problems solver has to perform tens to hundreds of iterations in every time or non-linear solver step. This generates thousands of iterations and the presented approach has a potential to run up to 4.1 times faster.

Table 4. The overall performance evaluation of the GPU acceleration of the Total FETI solver in ESPRESO which includes both preprocessing and solver runtime. Three scenarios with 100, 500 and 2000 iterations are evaluated. The table shows that the sweet spot, where the GPU becomes more efficient is approximately around 500 iterations and more.

Decomposition num. of subdom.; subdom. size [DOFs]	Preproc. time [s] Factorization; Schur compl.	100 iterations CPU[s];GPU[s]; **speedup[-]**	500 iterations CPU[s];GPU[s]; **speedup[-]**	2000 iterations CPU[s];GPU[s]; **speedup[-]**
48; 12288	4.1; 76.5	26.8; 82.1; **0.3**	117.7; 104.3; **1.1**	458.6; 187.6; **2.4**
48; 10125	3.1; 56.4	20.6; 61.0; **0.3**	90.7; 79.3; **1.1**	353.8; 148.3; **2.4**
64; 8232	2.9; 49.6	21.1; 54.5; **0.4**	93.7; 74.1; **1.3**	365.9; 147.5; **2.5**
80; 6591	2.6; 42.0	18.9; 47.1; **0.4**	84.2; 67.3; **1.3**	328.9; 143.3; **2.3**
100; 5184	2.4; 34.0	17.4; 39.2; **0.4**	77.1; 60.0; **1.3**	301.1; 137.8; **2.2**
150; 3993	2.4; 34.9	18.0; 41.5; **0.4**	80.4; 67.9; **1.2**	314.5; 167.0; **1.9**
216; 3000	2.2; 30.6	17.1; 38.7; **0.4**	76.3; 70.9; **1.1**	298.7; 191.8; **1.6**

6 Conclusions

Using the Schur complement opens the possibility to accelerate FETI algorithms by modern highly parallel hardware such as GPU accelerators. However this approach requires more expensive preprocessing in form of calculation of the Schur complement than original approach used in Total FETI. The preprocessing is between 13.7 to 18.6 times slower for the hardware configuration of the Titan supercomputer. This translates into penalty of 28.4 to 72.4 seconds for problems of size approximately 0.5 millions of degrees of freedom. Problems of this size fully utilize the 6 GB of Tesla K20X memory.

This paper presents the proof of concept, that manycore accelerators such as GPU or Intel Xeon Phi are valid hardware technology to make FETI solvers run faster. The GPU implementation is presented here, but the version for Xeon Phi is under active development. The problem evaluated in this paper, 0.5 million of unknowns decomposed into 216 domains, is rather small in order to fit single GPU with 6GB of memory. This means that the coarse problem is not causing any performance bottleneck in this case. The evaluation of the coarse problem processing for Total FETI in ESPRESO is presented in [14]. We are currently

working on extending the ESPRESO solver to support multiple accelerators per node.

Because the computation of the Schur complement for large subdomains becomes very costly it is optimal to decompose the problem into smaller subdomains. The subdomains of sizes 3000 to 12000 DOFs as presented in this paper are reasonable setting for current hardware architectures. Solving extremely large problems decomposed into small subdomains introduces very large coarse problem in case of Total FETI method. This becomes the main bottleneck for large problems. It is also the reason why Total FETI cannot be efficiently used to solve problems with several billions of unknowns.

The solution to this problem is to use multilevel decomposition. This method reduces the coarse problem size by grouping subdomains into clusters. Then the global coarse problem size is defined by a number of clusters and not by a number of subdomains. We are currently working on the implementation of a Hybrid FETI method into ESPRESO together with Schur complement approach.

7 Acknowledgment

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science - LQ1602 and from the Large Infrastructures for Research, Experimental Development and Innovations project IT4Innovations National Supercomputing Center LM2015070; and by the EXA2CT project funded from the EUs Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610741.

References

1. C. Farhat, F-X. Roux: An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems, SIAM J. Sci. Stat. Comput. 13, 379-396, 1992.
2. Z. Dostál, D. Horák, R. Kučera: Total FETI - an easier implementable variant of the FETI method for numerical solution of elliptic PDE, Communications in Numerical Methods in Engineering 22, (12), 1155- 1162, 2006.
3. T. Brzobohatý, Z. Dostál, T. Kozubek, P. Kovář, A. Markopoulos: Cholesky decomposition with fixing nodes to stable computation of a generalized inverse of the stiffness matrix of a floating structure, International Journal for Numerical Methods in Enginering 88, (5), 493- 509, 2011. doi:10.1002/nme.3187
4. Z. Dostál, T. Kozubek, A. Markopoulos, M. Menšík: Cholesky decomposition of a positive semidefinite matrix with known kernel, Applied Mathematics and Computation 217, (13), 6067-6077, 2011. doi:10.1016/j.amc.2010.12.069
5. R. Kučera, T. Kozubek, A. Markopoulos: On large-scale generalized inverses in solving two-by-two block linear systems, Linear Algebra and Its Applications 438 (7), 3011-3029, 2013.
6. C. Farhat, J. Mandel, F-X. Roux, Optimal convergence properties of the FETI domain decomposition method, Comput. Methods Appl. Mech. Eng. 115, 1994; 365–385.

7. F-X. Roux, C. Farhat, Parallel implementation of direct solution strategies for the coarse grid solvers in 2-level FETI method, Contemporary Math. 218, 1998; 158–173.
8. T. Kozubek, V. Vondrák, M. Menšík, D. Horák, Z. Dostál, V. Hapla, P. Kabelikova, M. Cermak, Total FETI domain decomposition method and its massively parallel implementation, Advances of Engeneering Software, 2013; 14–22
9. A. Kuzmin, M. Luisier, O. Schenk, Fast Methods for Computing Selected Elements of the Green's Function in Massively Parallel Nanoelectronic Device Simulations, Euro-Par 2013, LNCS 8097, F. Wolf, B. Mohr, and D. an Ney (Eds.), Springer-Verlag Berlin Heidelberg, pp. 533-544, 2013
10. O. Schenk, M. Bollhöfer, and R. Römer, On large-scale diagonalization techniques for the Anderson model of localization. Featured SIGEST paper in the SIAM Review selected "on the basis of its exceptional interest to the entire SIAM community". SIAM Review 50 (2008), pp. 91-112.
11. O. Schenk, A. Wächter, and M. Hagemann, Matching-based Preprocessing Algorithms to the Solution of Saddle-Point Problems in Large-Scale Nonconvex Interior-Point Optimization. Journal of Computational Optimization and Applications, pp. 321-341, Volume 36, Numbers 2-3 / April, 2007. DOI:10.1007/s10589-006-9003-y
12. C. Petra, O. Schenk, M. Lubin, K. Gänter, An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization, SIAM J. Sci. Comput 36-2, pp. C139-C162, 2014. DOI: 10.1137/130908737
13. J. D. Hogg, J. A. Scott, A note on the solve phase of a multicore solver, SFTC Rutherford Appleton Laboratory, Technical report, Science and Technology Facilities Council, 2010, June
14. L. Říha, T. Brzobohatý, A. Markopoulos, "Highly Scalable FETI Methods in ESPRESO", in P. Ivnyi, B.H.V. Topping, (Editors), "Proceedings of the Fourth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering", Civil-Comp Press, Stirlingshire, UK, Paper 17, 2015. doi:10.4203/ccp.107.17