Acceleration of the BEM4I library using the Intel Xeon Phi coprocessors

Michal Merta, Jan Zapletal

IT4Innovations National Supercomputing Center

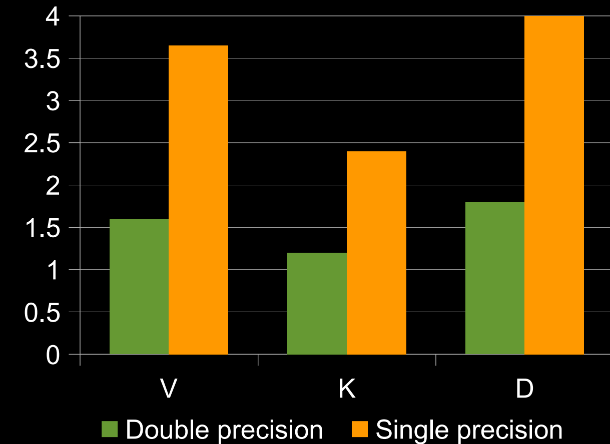
13. Workshop on fast boundary element methods in industrial applications
October 23 – 26, Soellerhaus, Austria

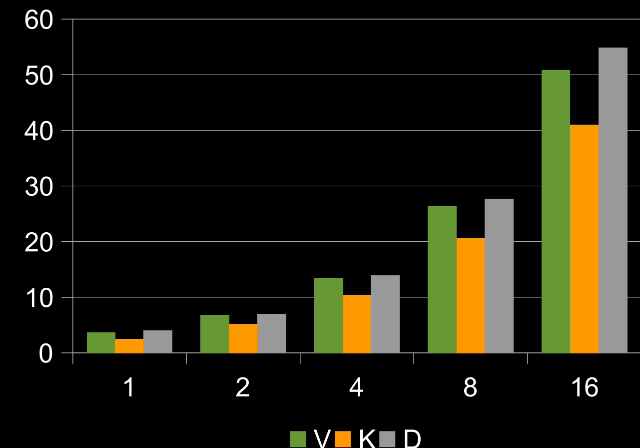
The BEM4I library

- Developed at IT4Innovations National Supercomputing Center, Ostrava, Czech Republic
- C++
- OpenMP & MPI
- SIMD vectorization
- Acceleration using Intel Xeon Phi
- ACA
- 3D Laplace, Helmholtz, Lamé, wave equation

Current results overview

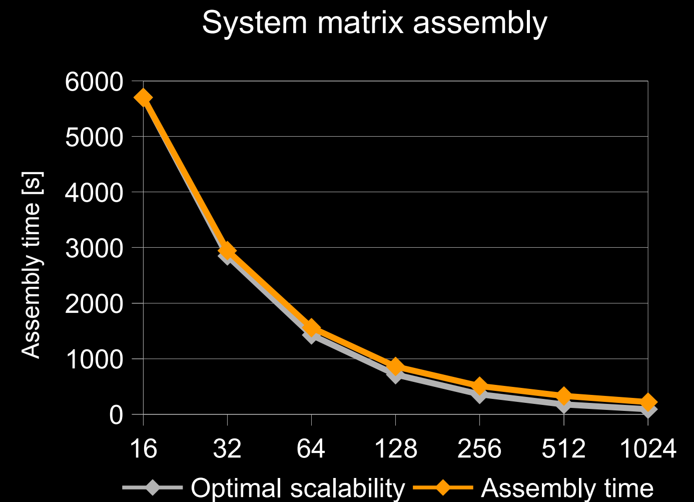
- Code vectorization
  - Using the Vc library and Intel's pragmas
  - Vectorization of both semi-analytic & fully numerical quadrature
  - Merta, M.; Zapletal, J. *Acceleration of boundary element method by explicit vectorization*. Advances in Engineering Software 86, Elsevier, 2015, pp. 70-79.

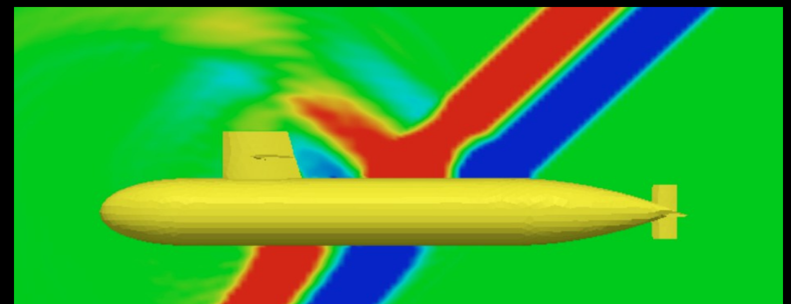




Current results overview

- Time-dependent BEM
  - Parallel solver for wave equation in 3D
  - In cooperation with A. Veit
  - Veit, A.; Merta, M.; Zapletal, J.; Lukáš, D. *Efficient solution of time-domain boundary integral equations arising in sound-hard scattering*. Int. J. Num. Methods Engng. Submitted, under revision.

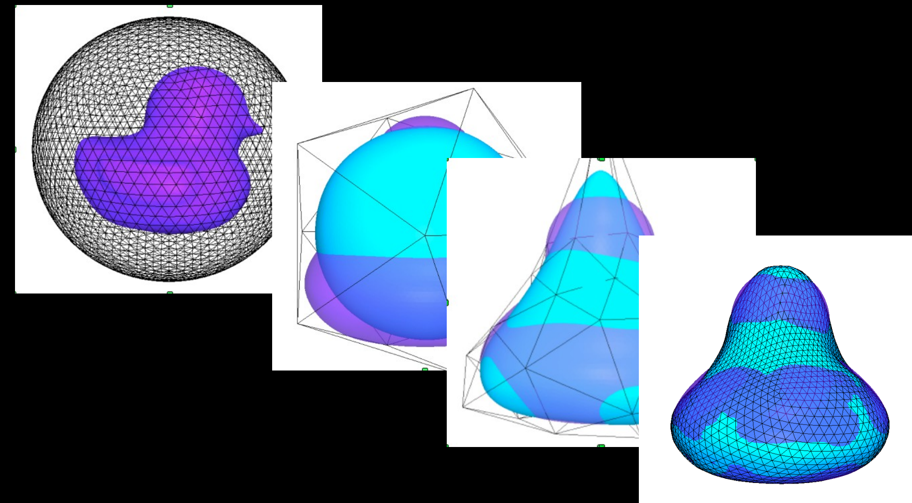


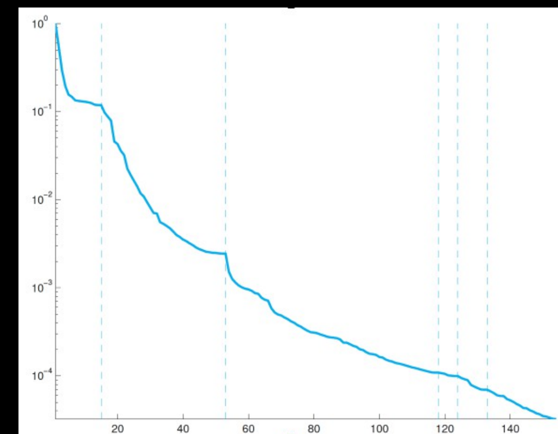


Current results overview

- Shape optimization
  - Using BEM & subdivision surfaces
- Homogenization
  - Cooperation with University of West Bohemia, Dept. of Mechanics
  - Comparison of FEM & BEM for periodic structured domains
- BEM parallelization
  - Lukáš, Kovář, Kovářová, Merta. *A Parallel Fast Boundary Element Method Using Cyclic Graph Decompositions.* Numerical algorithms, 2015.

Heat source reconstruction

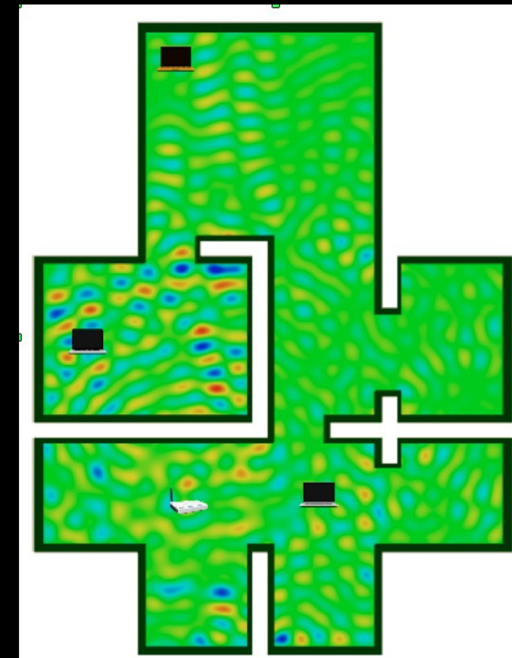




Current results overview

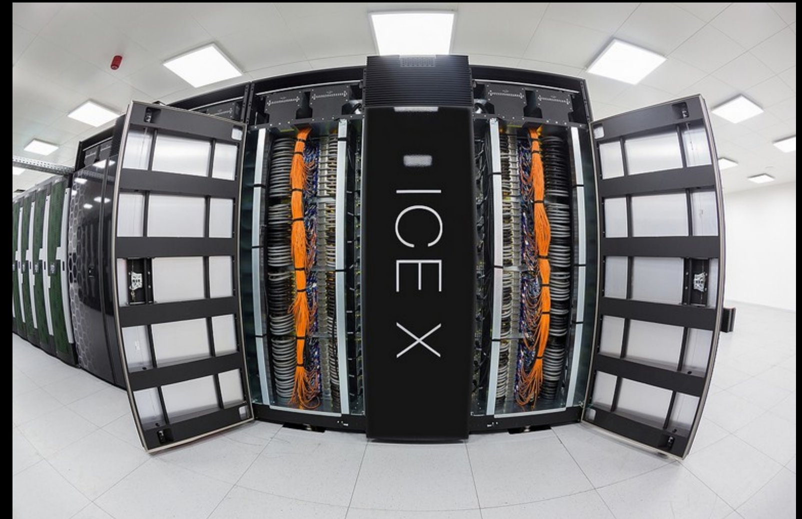
- Summer of HPC
  - PRACE organized internship program
  - BEM4I used for simplified wifi signal propagation modelling
  - Modelled using 1152 cores
- Cooperation with ESPRESO solver
  - Domain decomposition solver developed at IT4Innovations
  - Primarily FETI method, currently implementing BETI

Signal propagation in an apartment



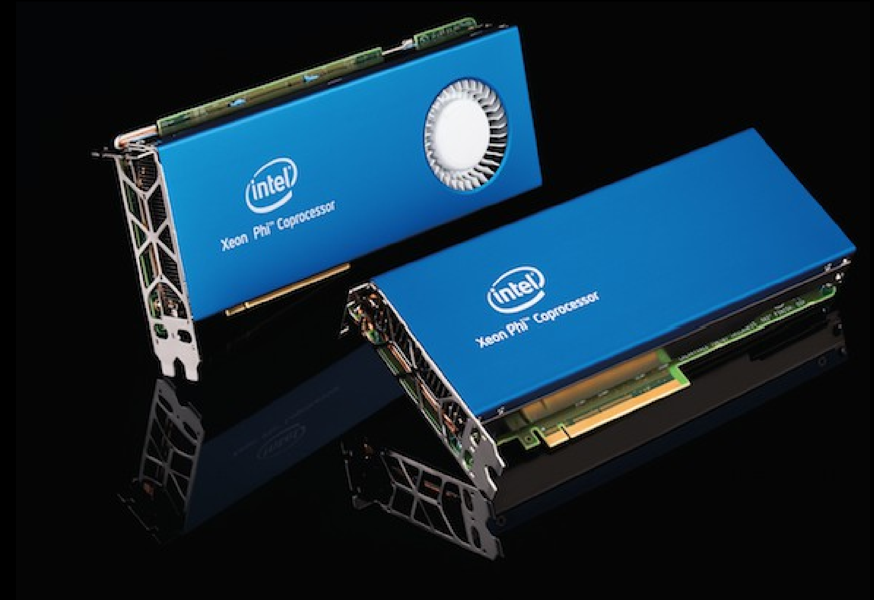
Current project: porting to Xeon Phi

- Salomon supercomputer at IT4Innovations operational since August
  - Rpeak = 2.011 Pflops (40 in Top 500)
  - 1008 compute nodes
  - 2x Intel Xeon E5-2680v3 (2.5 GHz, 12 cores, Haswell), 128 GB RAM per node
  - 432 nodes equipped with 2 Intel Xeon Phi 7210P coprocessors
  - 76896 cores in total
  - IT4I became Intel Parallel Computing Center



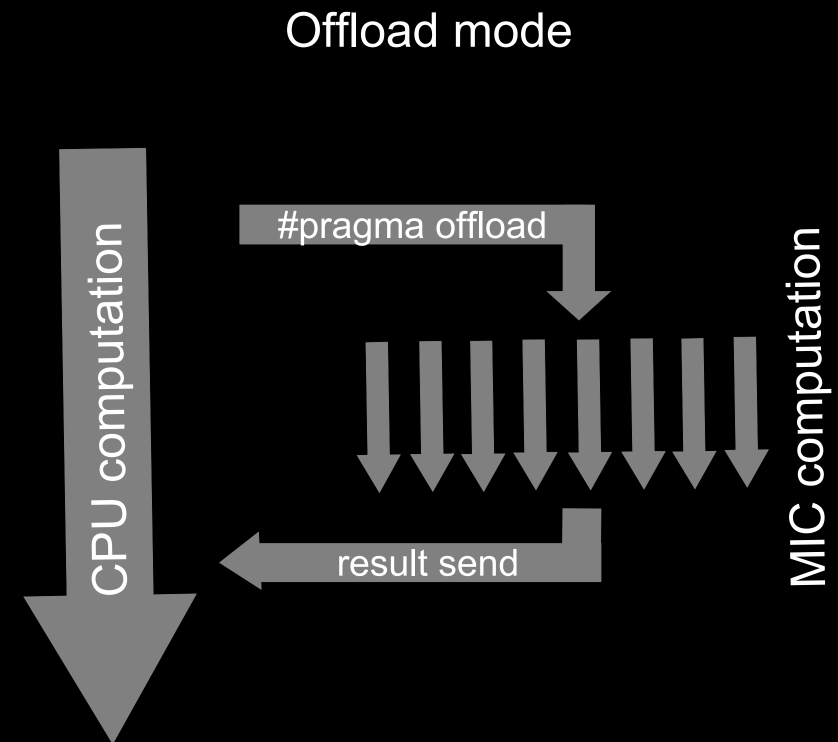
Current project: porting to Xeon Phi

- Xeon Phi 7120P
  - Intel's Many Integrated Cores architecture (MIC)
  - Rpeak = 1.208 Tflops
  - 61 cores, 4 hw. threads per core, 1.238 Ghz
  - 512 bit wide vector registers
  - 16 GB memory
  - 350 GB/s max. memory bandwidth
  - 1 Xeon Phi ≈ 2 Xeon (Haswell)



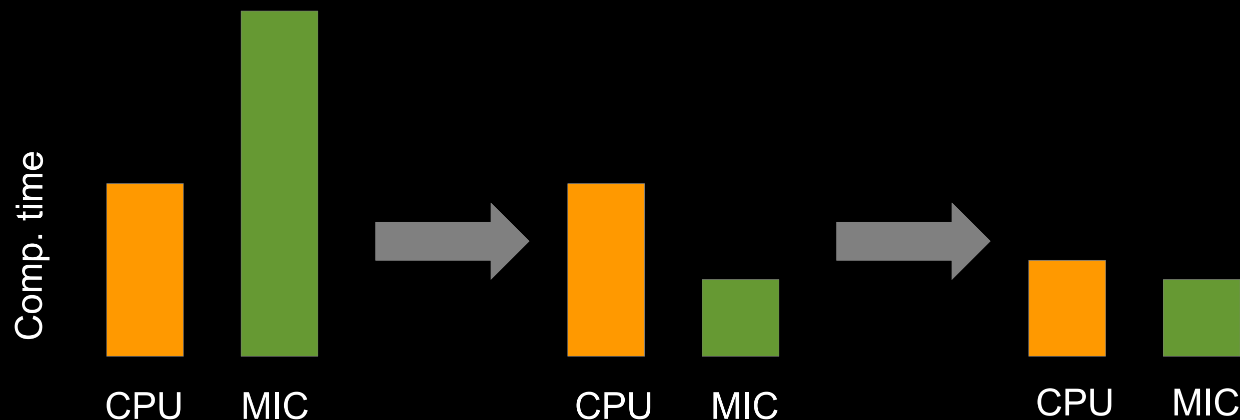
Current project: porting to Xeon Phi

- Computation modes
  - Native mode
    - Whole application compiled for MIC
    - Easier, limited memory
  - Offload mode
    - Only parts of the code offloaded to MIC
    - Requires more complex changes to the code
    - Possible bottleneck – data transfer to/from host memory via PCIe
- To achieve good performance: vectorize and scale!



Current project: porting to Xeon Phi

- Why?
  - Accelerate computation on a single node up to 3 times
  - Prepare for the upcoming Intel technologies (KNL, KNH)
  - Original CPU code hugely benefits from optimization for MIC



Library structure

Boundary element spaces approximations

- BESpace
- FastBESpace

Bilinear forms approximations (matrix assemblers)

- BEBilinearForm
- BEBilinearFormLaplace
- BEBilinearFormLamé
- BEBilinearFormHelmholtz
- BEBilinearFormWave

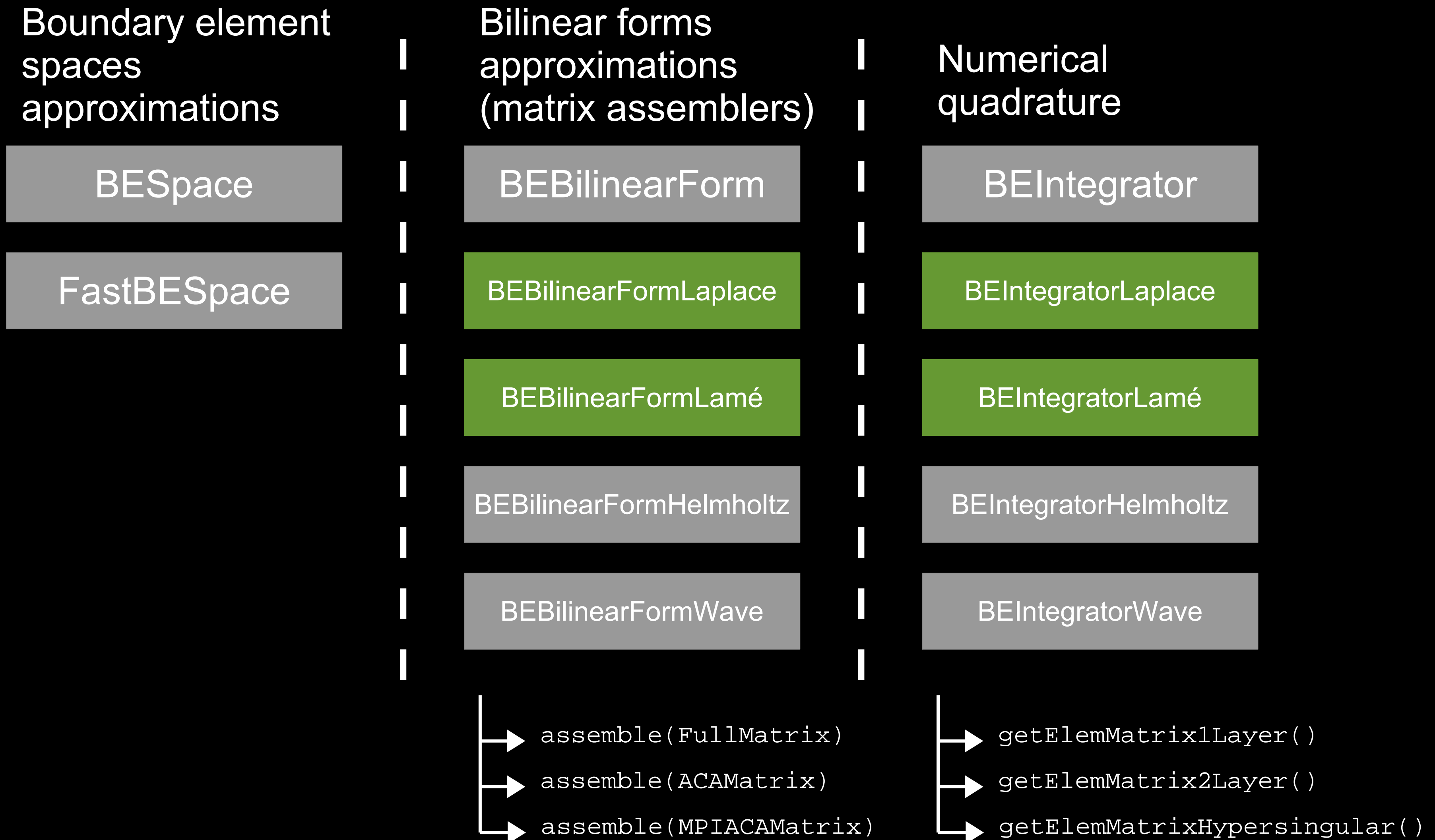
```
assemble(FullMatrix)
assemble(ACAMatrix)
assemble(MPIACAMatrix)
```

Numerical quadrature

- BEIntegrator
- BEIntegratorLaplace
- BEIntegratorLamé
- BEIntegratorHelmholtz
- BEIntegratorWave

```
getElemMatrix1Layer()
getElemMatrix2Layer()
getElemMatrixHypersingular()
```

Library structure



Offload to MIC

```
class BEBilinearFormLaplace1Layer {

  void assemble(FullMatrix & V) {

    ...

    #pragma omp parallel
    FullMatrix V(iMax, jMax);
    BEIntegrator( mesh, quadratureOrders );

    #pragma omp for collapse(2)
    for ( int i = 0; i < iMax; i++ ) {
      for ( int j = 0; j < jMax; j++ ) {

        integrator.getElemMatrix1Layer( i, j, localMatrix );

        // atomic update
        {
          V.addToPositions( i , j, localMatrix );
        }
      }
    }
  }
}
```

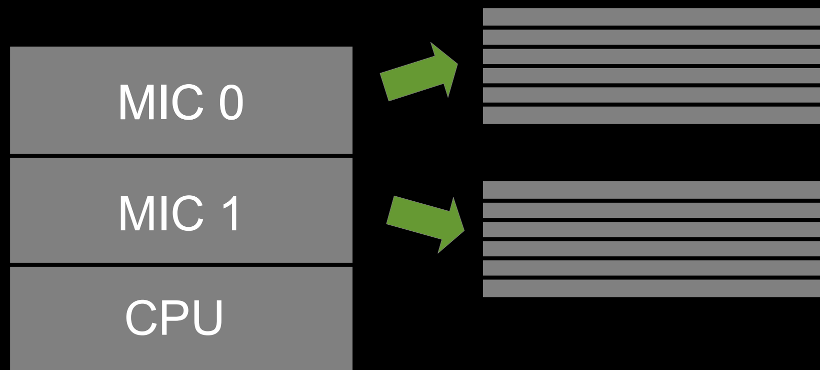
Original code

Offload to MIC

1.

```
int * elements = mesh.getElements();
double * nodes = mesh.getNodes();
int nElems     = mesh.getNElems();
int nNodes     = mesh.getNNodes();
```

2.



1. Extract raw data from C++ objects (int * elements, double * nodes, ...)
2. Split the matrix among MICs and CPU
3. Allocate data on CPU (using `_mm_malloc( ..., 64)`)
4. Initial offload to MIC (send and preallocate all necessary data)
5. Concurrent computation on MICs and CPU
6. Send result from MIC to CPU
7. Combine results

Offload to MIC

3.

```
// allocate matrix buffers
matrixBuffers[ 0 ] =
    (double *) _mm_malloc( bufferLength *
    sizeof ( double ), 64 );

matrixBuffers[ 1 ] =
    (double *) _mm_malloc( bufferLength *
    sizeof ( double ), 64 );


// help to manually unroll some loops
double * outWeights = (double *) _mm_malloc(
    nOutP * nInP * sizeof ( double ), 64 );
double * inWeights  = (double *) _mm_malloc(
    nOutP * nInP * sizeof ( double ), 64 );

for ( int i = 0; i < nOutP; ++i ) {
  for ( int j = 0; j < nInP; ++j ) {
    outWeights[ counter ] = quadWeights[ i ];
    inWeights[ counter ]  = quadWeights[ j ];
  }
}

...
```

1. Extract raw data from C++ objects (int * elements, double * nodes, ...)
2. Split the matrix among MICs and CPU
3. Allocate data on CPU (using `_mm_malloc( ..., 64 )`)
4. Initial offload to MIC (send and preallocate all necessary data)
5. Concurrent computation on MICs and CPU
6. Send result from MIC to CPU
7. Combine results

Offload to MIC

4.

```
// send data to MIC 0
#pragma offload_transfer target( mic : 0 )
in ( matrixBuffer[ 0 ] : length( bufferLength )
alloc_if( 1 ) free_if( 0 ) )
in ( matrixBuffer[ 1 ] : length( bufferLength )
alloc_if( 1 ) free_if( 0 ) )
in( nodes : length( 3 * nNodes )
alloc_if( 1 ) free_if( 0 ) )
in( elements : length( 3 * nElems )
alloc_if( 1 ) free_if( 0 ) )
...
```

1. Extract raw data from C++ objects (int * elements, double * nodes, ...)
2. Split the matrix among MICs and CPU
3. Allocate data on CPU (using `_mm_malloc( ..., 64 )`)
4. Initial offload to MIC (send and preallocate all necessary data)
5. Concurrent computation on MICs and CPU
6. Send result from MIC to CPU
7. Combine results

Offload to MIC

5. & 6. & 7.

```
// initiate parallel region on CPU
#pragma omp parallel
{
  // thread num. 0 communicates with MIC
  if ( omp_get_thread_num() < 1 ) {
    for (int i = 0; i < nSubmatrices; ++i) {
      #pragma offload target( mic : 0 )
      signal( i )
      ...
      {
        // offloaded region, we have 240 threads on MIC
        #pragma omp parallel num_threads( 240 )
        {
          ...
          for ( int i = myStart; i < myEnd; ++i ) {
            for ( int j = 0; j < nElems; ++j ) {
              getElemMatrixOnMIC( i, j, matrix);
            }
          }
          ...
        }
      }
    }
    // send result from the previous iteration
    // and add them to the global matrix
    #pragma offload_transfer target( mic : 0 )
    wait( i - 1 ) out( matrixBuffer[ (i - 1) % 2 ] )
    ...
  } else {
     // other CPU threads do some work
  }
}
```

1. Extract raw data from C++ objects (int * elements, double * nodes, ...)
2. Split the matrix among MICs and CPU
3. Allocate data on CPU (using `_mm_malloc( ..., 64 )`)
4. Initial offload to MIC (send and preallocate all necessary data)
5. Concurrent computation on MICs and CPU
6. Send result from MIC to CPU
7. Combine results

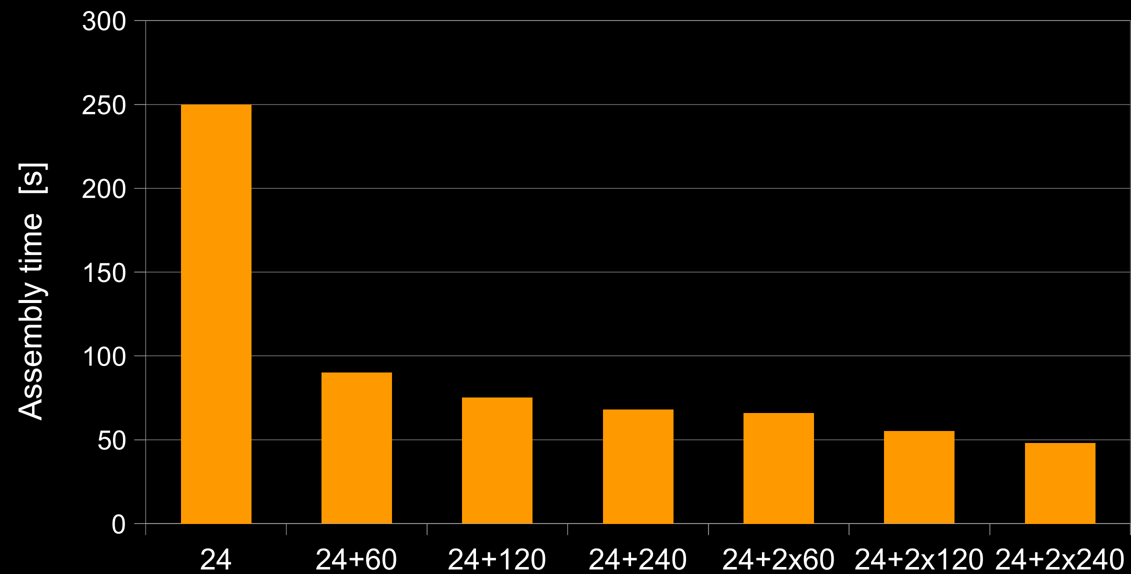
Offload to MIC

GetElemMatrix1LayerOnMIC() function

```
// Gaussian quadrature over disjoint elements

__assume_aligned( outWeights, 64 );
__assume_aligned( inWeights, 64 );
...

// quadrature points in Structure of Arrays form
__assume_aligned( outX1, 64 );
__assume_aligned( outX2, 64 );
__assume_aligned( outX3, 64 );
...

// assist vectorization using Intel's pragma
#pragma simd linear( i : 1 ) reduction( + : entry )
for ( i = 0; i < nOutP * nInP; ++i ) {
  norm = std::sqrt(
    ( outX1[ i ] - inX1[ i ] ) * ( outX1[ i ] - inX1[ i ] ) +
    ( outX2[ i ] - inX2[ i ] ) * ( outX2[ i ] - inX2[ i ] ) +
    ( outX3[ i ] - inX3[ i ] ) * ( outX3[ i ] - inX3[ i ] ) );

    entry += outWeights[ i ] * inWeights[ i ] * ( PI_FACT / norm );
  }

  entry *= areas[ outerElem ] * areas[ innerElem ];
  *elemMatrix = entry;
}
```

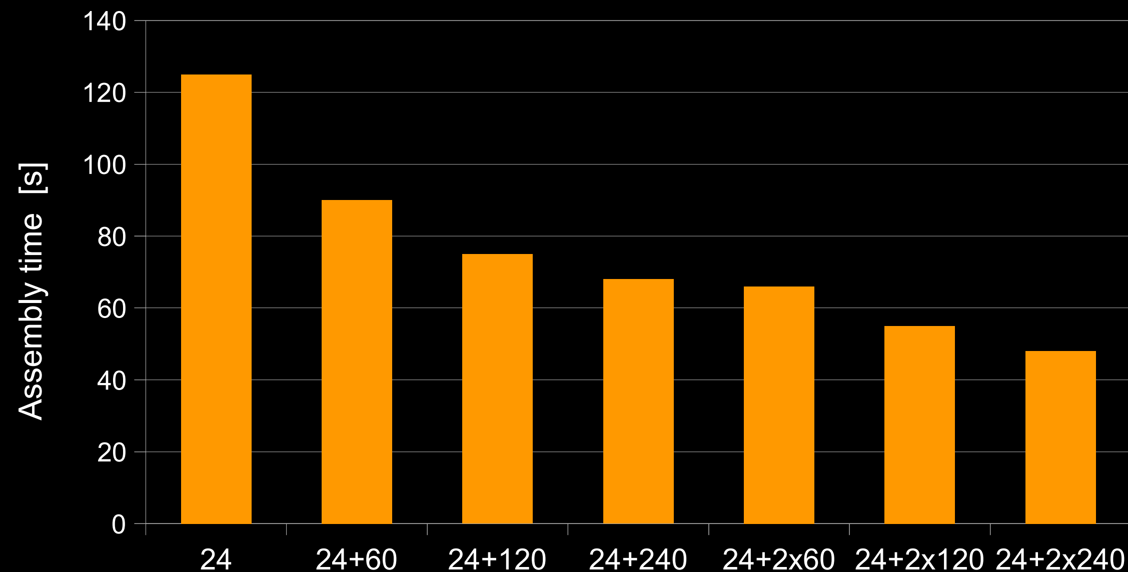
- Performance benefit from using Structure of Arrays (SoA) instead of Array of Structures (AoS)
  - [x1, y1, z1, x2, y2, z2, x3, y3, z3 ...] → [x1, x2, x3, ..., y1, y2, y3, ..., z1, z2, z3, ...]

Numerical benchmarks



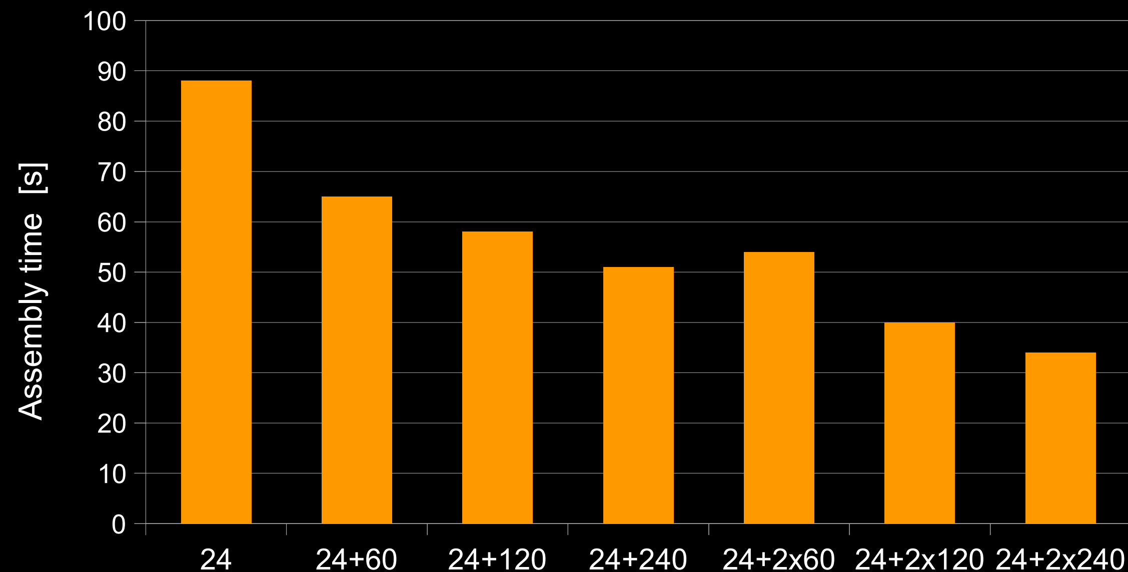
- Single layer
- 81,920 surface elements

Numerical benchmarks



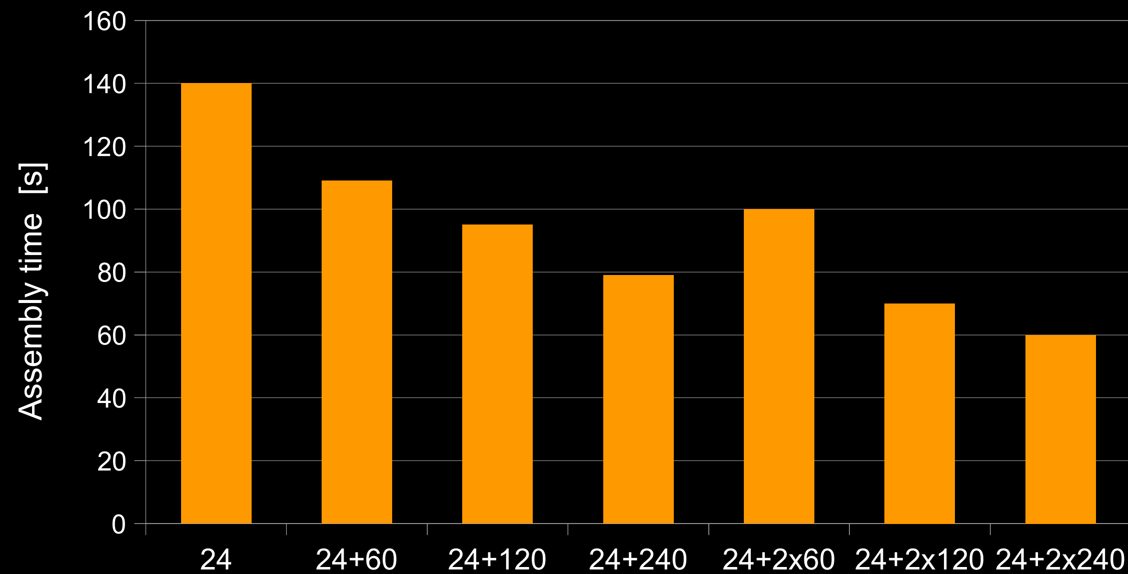
- Single layer
- 81,920 surface elements
- Double precision arithmetic
- Max. speedup: 2.65

Numerical benchmarks



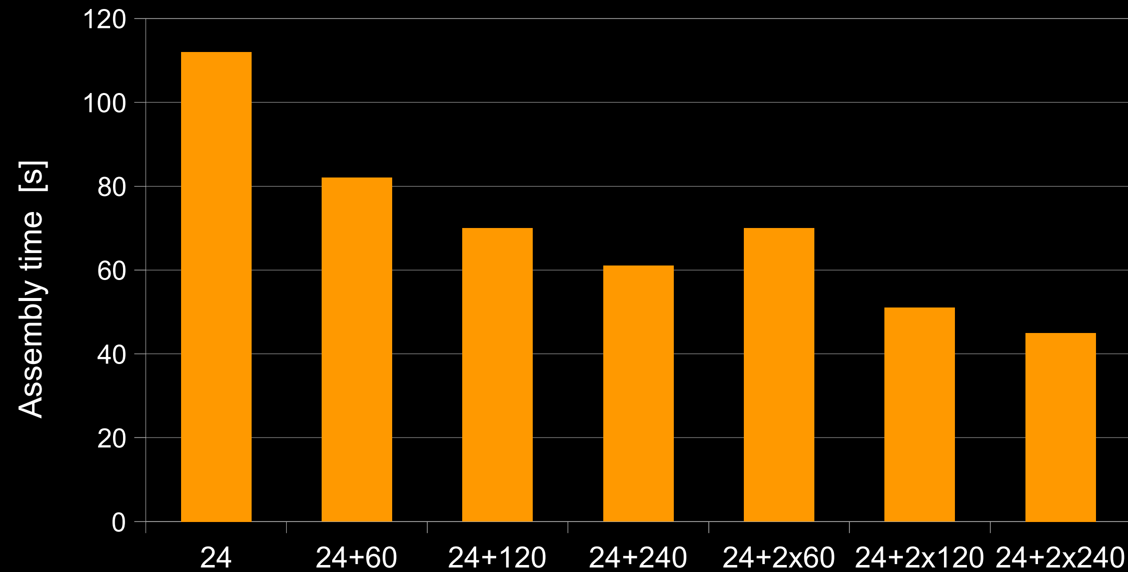
- Single layer
- 81,920 surface elements
- Single precision arithmetic
- Max. speedup: 2.45

Numerical benchmarks



- Double layer
- 81,920 surface elements
- Double precision arithmetic
- Max. speedup: 2.43

Numerical benchmarks



- Double layer
- 81,920 surface elements
- Single precision arithmetic
- Max. speedup: 2.43

Acceleration of the Lamé solver

- Similar principles
- Additionally
  - all matrix blocks assembled at once
  - `getElemMatrix()` returns 10 local matrices (+ local K)
- Preliminary results show speed-up up to 4 (double prec.)

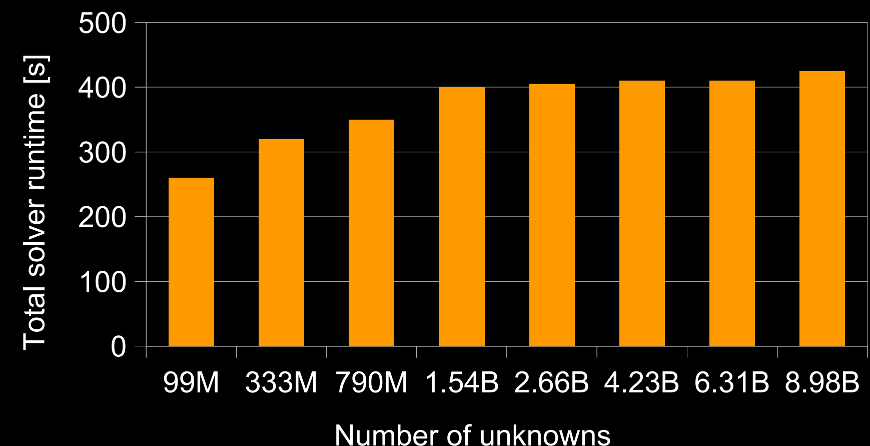
$$\mathsf{V}_{m,h}^{\text{Lamé}} = \frac{1+\nu}{2E(1-\nu)} \cdot$$

$$\left((3-4\nu) \begin{pmatrix} \mathsf{V}_{m,h} & \mathsf{O} & \mathsf{O} \\ \mathsf{O} & \mathsf{V}_{m,h} & \mathsf{O} \\ \mathsf{O} & \mathsf{O} & \mathsf{V}_{m,h} \end{pmatrix} + \begin{pmatrix} \mathsf{V}_{11,m,h} & \mathsf{V}_{12,m,h} & \mathsf{V}_{13,m,h} \\ \mathsf{V}_{12,m,h} & \mathsf{V}_{22,m,h} & \mathsf{V}_{23,m,h} \\ \mathsf{V}_{13,m,h} & \mathsf{V}_{23,m,h} & \mathsf{V}_{33,m,h} \end{pmatrix} \right)$$

$$\mathsf{V}_{m,h}[k,l] := \frac{1}{4\pi} \int_{\tau_k^m} \int_{\tau_l^m} \frac{1}{\|x-y\|} \, \mathrm{d}s_y \, \mathrm{d}s_x \qquad \mathsf{V}_{ij,m,h}[k,l] := \frac{1}{4\pi} \int_{\tau_k^m} \int_{\tau_l^m} \frac{(x_i-y_i)(x_j-y_j)}{\|x-y\|^3} \, \mathrm{d}s_y \, \mathrm{d}s_x$$

BETI in ESPRESO

- ESPRESO
  - FETI solver for elasticity developed at IT4I
  - C++, MPI, Pardiso
  - Scalability demonstrated up to 17,496 cores and 9 billion unknowns



Our goal: MIC-accelerated BETI in ESPRESO

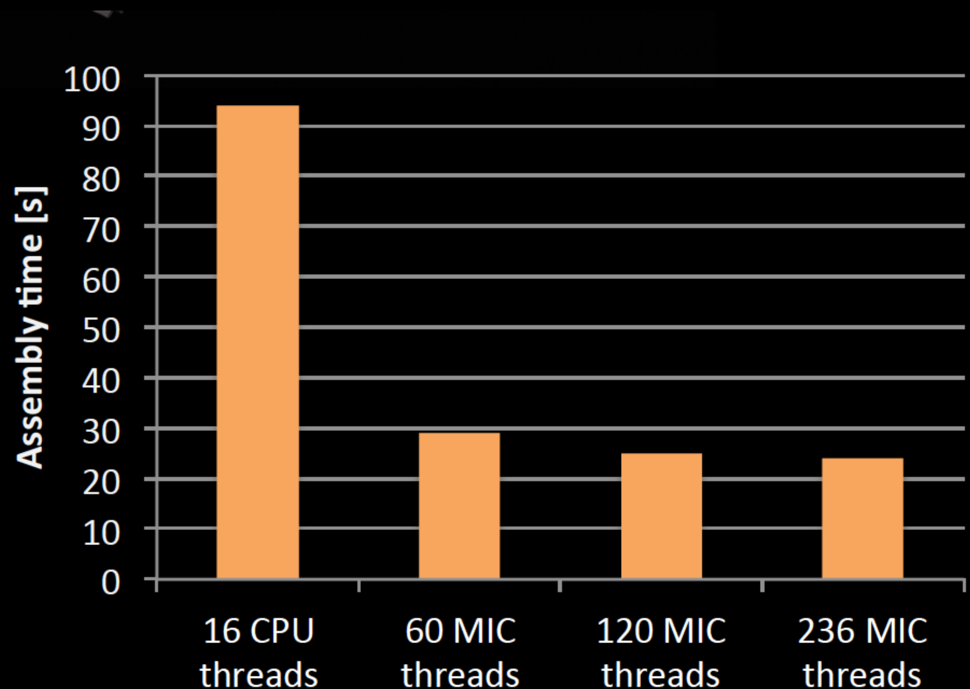
BETI in ESPRESO

Sorry, no numerical results for MIC today...

Older result, without Xeon Phi.

Future: ACA acceleration

- Offloading block approximations to the coprocessor
- Preliminary results from the Anselm cluster



Conclusion

- Acceleration of the boundary element computation in BEM4I using the Intel Xeon Phi coprocessors
  - Currently Laplace & Lamé
- Speedup up to 2.5 using 2 coprocessors + CPU (Laplace)
- Focusing on MIC optimization can improve the original CPU code
- Currently working on BETI in ESPRESO library
- Future plans ACA