

Boundary element quadrature schemes for multi- and many-core architectures

Jan Zapletal, Michal Merta, Lukáš Malý

IT4Innovations, Dept. of Applied Mathematics
VŠB-TU Ostrava
jan.zapletal@vsb.cz

Intel MIC programming workshop, February 7, 2017

IT4Innovations
national01\$#&0
supercomputing
center@#01%101



EUROPEAN UNION
EUROPEAN REGIONAL DEVELOPMENT FUND
INVESTING IN YOUR FUTURE



**OP Research and
Development for Innovation**

1 BEM4I

- Boundary element method
- OpenMP threading
- OpenMP vectorization
- Adaptive cross approximation

2 Numerical experiments

- Full assembly
- ACA assembly

3 Conclusion

- 1 BEM4I
 - Boundary element method
 - OpenMP threading
 - OpenMP vectorization
 - Adaptive cross approximation
- 2 Numerical experiments
 - Full assembly
 - ACA assembly
- 3 Conclusion

BEM4I

- BEM is an alternative to FEM for the solution of PDEs,
- BEM4I - 3D boundary element library for
 - Laplace equation (heat transfer),
 - Helmholtz equation (wave scattering),
 - Lamé equation (linear elasticity),
 - time-dependent wave equation.
- Specifications
 - C++, interface to MKL or other BLAS, LAPACK,
 - ACA for matrix sparsification.
- Strategies
 - SIMD vectorization for surface integrals (**OpenMP**, Vc library),
 - OpenMP for local element contributions / individual ACA blocks,
 - MPI for distributed matrices,
 - BETI with the ESPRESO library,
 - Intel MIC offload/**native**.

BEM4I

- BEM is an alternative to FEM for the solution of PDEs,
- BEM4I - 3D boundary element library for
 - Laplace equation (heat transfer),
 - Helmholtz equation (wave scattering),
 - Lamé equation (linear elasticity),
 - time-dependent wave equation.
- Specifications
 - C++, interface to MKL or other BLAS, LAPACK,
 - ACA for matrix sparsification.
- Strategies
 - SIMD vectorization for surface integrals (**OpenMP**, Vc library),
 - OpenMP for local element contributions / individual ACA blocks,
 - MPI for distributed matrices,
 - BETI with the ESPRESO library,
 - Intel MIC offload/**native**.

BEM4I

- BEM is an alternative to FEM for the solution of PDEs,
- BEM4I - 3D boundary element library for
 - Laplace equation (heat transfer),
 - Helmholtz equation (wave scattering),
 - Lamé equation (linear elasticity),
 - time-dependent wave equation.
- Specifications
 - C++, interface to MKL or other BLAS, LAPACK,
 - ACA for matrix sparsification.
- Strategies
 - SIMD vectorization for surface integrals (**OpenMP**, Vc library),
 - OpenMP for local element contributions / individual ACA blocks,
 - MPI for distributed matrices,
 - BETI with the ESPRESO library,
 - Intel MIC offload/**native**.

BEM4I

- BEM is an alternative to FEM for the solution of PDEs,
- BEM4I - 3D boundary element library for
 - Laplace equation (heat transfer),
 - Helmholtz equation (wave scattering),
 - Lamé equation (linear elasticity),
 - time-dependent wave equation.
- Specifications
 - C++, interface to MKL or other BLAS, LAPACK,
 - ACA for matrix sparsification.
- Strategies
 - SIMD vectorization for surface integrals (**OpenMP**, Vc library),
 - OpenMP for local element contributions / individual ACA blocks,
 - MPI for distributed matrices,
 - BETI with the ESPRESO library,
 - Intel MIC offload/**native**.

Boundary value problem for the Laplace equation

$$\begin{cases} -\Delta u = 0 & \text{in } \Omega, \\ u = f & \text{on } \Gamma_D, \\ \frac{\partial u}{\partial \mathbf{n}} = g & \text{on } \Gamma_N. \end{cases}$$

Representation formula for $\mathbf{x} \in \Omega$

$$u(\mathbf{x}) = \frac{1}{4\pi} \int_{\partial\Omega} \frac{1}{\|\mathbf{x} - \mathbf{y}\|} \frac{\partial u}{\partial \mathbf{n}}(\mathbf{y}) \, d\mathbf{s}_y - \frac{1}{4\pi} \int_{\partial\Omega} \frac{\langle \mathbf{x} - \mathbf{y}, \mathbf{n}(\mathbf{y}) \rangle}{\|\mathbf{x} - \mathbf{y}\|^3} u(\mathbf{y}) \, d\mathbf{s}_y.$$

Boundary integral equations for $\mathbf{x} \in \partial\Omega$

$$\begin{aligned} \frac{1}{4\pi} \int_{\partial\Omega} \frac{1}{\|\mathbf{x} - \mathbf{y}\|} \frac{\partial u}{\partial \mathbf{n}}(\mathbf{y}) \, d\mathbf{s}_y &= \frac{1}{2} u(\mathbf{x}) + \frac{1}{4\pi} \int_{\partial\Omega} \frac{\langle \mathbf{x} - \mathbf{y}, \mathbf{n}(\mathbf{y}) \rangle}{\|\mathbf{x} - \mathbf{y}\|^3} u(\mathbf{y}) \, d\mathbf{s}_y, \\ -\frac{\partial}{\partial \mathbf{n}_x} \frac{1}{4\pi} \int_{\partial\Omega} \frac{\langle \mathbf{x} - \mathbf{y}, \mathbf{n}(\mathbf{y}) \rangle}{\|\mathbf{x} - \mathbf{y}\|^3} u(\mathbf{y}) \, d\mathbf{s}_y &= \frac{1}{2} \frac{\partial u}{\partial \mathbf{n}}(\mathbf{x}) - \frac{1}{4\pi} \int_{\partial\Omega} \frac{\langle \mathbf{y} - \mathbf{x}, \mathbf{n}(\mathbf{x}) \rangle}{\|\mathbf{x} - \mathbf{y}\|^3} \frac{\partial u}{\partial \mathbf{n}}(\mathbf{y}) \, d\mathbf{s}_y. \end{aligned}$$

Discretization leads to the systems

$$\mathbf{V}_h \mathbf{g} = \left(\frac{1}{2} \mathbf{M}_h + \mathbf{K}_h \right) \mathbf{f}, \quad \mathbf{D}_h \mathbf{f} = \left(\frac{1}{2} \mathbf{M}_h - \mathbf{K}_h \right)^\top \mathbf{g}$$

with the matrices

$$\mathbf{V}_h[\ell, k] := \frac{1}{4\pi} \int_{\tau_\ell} \int_{\tau_k} \frac{1}{\|\mathbf{x} - \mathbf{y}\|} d\mathbf{s}_y d\mathbf{s}_x$$

$$\mathbf{K}_h[\ell, i] := \frac{1}{4\pi} \int_{\tau_\ell} \int_{\partial\Omega} \varphi_i(\mathbf{y}) \frac{\langle \mathbf{x} - \mathbf{y}, \mathbf{n}(\mathbf{y}) \rangle}{\|\mathbf{x} - \mathbf{y}\|^3} d\mathbf{s}_y d\mathbf{s}_x$$

$$\mathbf{D}_h[j, i] := \frac{1}{4\pi} \int_{\partial\Omega} \int_{\partial\Omega} \frac{\langle \mathbf{curl} \varphi_i(\mathbf{y}), \mathbf{curl} \varphi_j(\mathbf{x}) \rangle}{\|\mathbf{x} - \mathbf{y}\|} d\mathbf{s}_y d\mathbf{s}_x = \mathbf{T}_h^\top \text{diag}(\mathbf{V}_h, \mathbf{V}_h, \mathbf{V}_h) \mathbf{T}_h,$$

$$\mathbf{M}_h[\ell, i] := \int_{\tau_\ell} \varphi_i(\mathbf{x}) d\mathbf{s}_x.$$

■ OpenMP threading for V_h

```
1 #pragma omp parallel for
2 for( int tau_k = 0; tau_k < E; ++tau_k ){ // columns
3   for( int tau_l = 0; tau_l < E; ++tau_l ){ // rows
4     SLIntegrator.getLocalMatrix( *tau_l, *tau_k, Vloc );
5     V.set( *tau_l, *tau_k, Vloc.get( 0, 0 ) );
6   } }
```

■ OpenMP threading for K_h

■ OpenMP threading for V_h

```

1 #pragma omp parallel for
2 for( int tau_k = 0; tau_k < E; ++tau_k ){ // columns
3   for( int tau_l = 0; tau_l < E; ++tau_l ){ // rows
4     SLIntegrator.getLocalMatrix( *tau_l, *tau_k, Vloc );
5     V.set( *tau_l, *tau_k, Vloc.get( 0, 0 ) );
6   } }

```

■ OpenMP threading for K_h

```

1 #pragma omp parallel for
2 for( int tau_k = 0; tau_k < E; ++tau_k ){ // columns
3   for( int tau_l = 0; tau_l < E; ++tau_l ){ // rows
4     DLIntegrator.getLocalMatrix( *tau_l, *tau_k, Kloc );
5     #pragma omp atomic // (inside of add_atomic)
6     K.add_atomic( *tau_l, tau_k->node[ 0 ], Kloc.get( 0, 0 ) );
7     #pragma omp atomic // (inside of add_atomic)
8     K.add_atomic( *tau_l, tau_k->node[ 1 ], Kloc.get( 0, 1 ) );
9     #pragma omp atomic // (inside of add_atomic)
10    K.add_atomic( *tau_l, tau_k->node[ 2 ], Kloc.get( 0, 2 ) );
11  } }

```

- OpenMP threading for V_h

```

1 #pragma omp parallel for
2 for( int tau_k = 0; tau_k < E; ++tau_k ){ // columns
3   for( int tau_l = 0; tau_l < E; ++tau_l ){ // rows
4     SLIntegrator.getLocalMatrix( *tau_l, *tau_k, Vloc );
5     V.set( *tau_l, *tau_k, Vloc.get( 0, 0 ) );
6   } }

```

- OpenMP threading for K_h

```

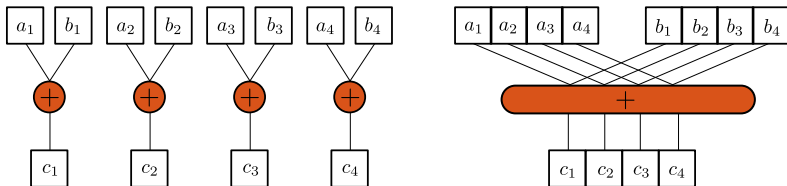
1 #pragma omp parallel for
2 for( int tau_l = 0; tau_l < E; ++tau_l ){ // rows
3   for( int tau_k = 0; tau_k < E; ++tau_k ){ // columns
4     DLIntegrator.getLocalMatrix( *tau_l, *tau_k, Kloc );
5
6     K.add( *tau_l, tau_k->node[ 0 ], Kloc.get( 0, 0 ) );
7
8     K.add( *tau_l, tau_k->node[ 1 ], Kloc.get( 0, 1 ) );
9
10    K.add( *tau_l, tau_k->node[ 2 ], Kloc.get( 0, 2 ) );
11  } }

```

- Avoid `#pragma omp critical`, use `#pragma omp atomic` if applicable.

■ Single Instruction Multiple Data (SIMD)

- processing vector with a single operation,
- provides data level parallelism,
- elements are of the same type.



■ Vector length

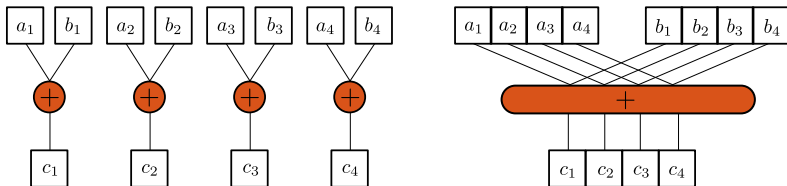
- 256 bits for AVX-2 (Haswell), 4 DP operands,
- 512 bits for IMCI (KNC), AVX512 (KNL), 8 DP operands.

■ Vectorization achieved by

- compiler auto-vectorization (code refactoring can help),
- OpenMP 4.0 pragmas (`#pragma omp simd`),
- intrinsic functions,
- wrapper library (Vc),
- assembly.

■ Single Instruction Multiple Data (SIMD)

- processing vector with a single operation,
- provides data level parallelism,
- elements are of the same type.



■ Vector length

- 256 bits for AVX-2 (Haswell), 4 DP operands,
- 512 bits for IMCI (KNC), AVX512 (KNL), 8 DP operands.

■ Vectorization achieved by

- compiler auto-vectorization (code refactoring can help),
- **OpenMP 4.0 pragmas** (`#pragma omp simd`),
- intrinsic functions,
- wrapper library (Vc),
- assembly.

■ Tuned vectorized integration over a square

```
1 double * w = new double[ S ];
2 w[ 0 ] = ... // init. weights
3 // x^1_1, x^1_2, ..., x^S_1, x^S_2
4 double x [ ] = { ... };
5
6
7
8
9
10
11 for( int l = 0; l < S; ++l ){
12     result += w[ l ] * f( x[ 2 * l ], x[ 2 * l + 1 ] );
13 }
14
15 delete w;
```

■ Loop peeling & strided access to memory

■ Tuned vectorized integration over a square

```

1 double * w = new double[ S ];
2 w[ 0 ] = ... // init. weights
3 // x^1_1, x^1_2, ..., x^S_1, x^S_2
4 double x [ ] = { ... };
5
6
7
8
9
10 #pragma omp simd reduction( + : result )
11 for( int l = 0; l < S; ++l ){
12     result += w[ l ] * f( x[ 2 * l ], x[ 2 * l + 1 ] );
13 }
14
15 delete w;

```

■ Loop peeling & strided access to memory

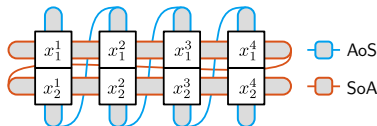
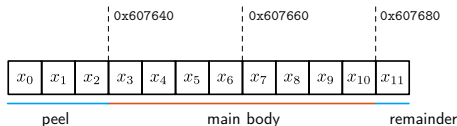
■ Tuned vectorized integration over a square

```

1 double * w = new double[ S ];
2 w[ 0 ] = ... // init. weights
3 // x^1_1, x^1_2, ..., x^S_1, x^S_2
4 double x [ ] = { ... };
5
6
7
8
9
10 #pragma omp simd reduction( + : result )
11 for( int l = 0; l < S; ++l ){
12     result += w[ l ] * f( x[ 2 * l ], x[ 2 * l + 1 ] );
13 }
14
15 delete w;

```

■ Loop peeling & strided access to memory



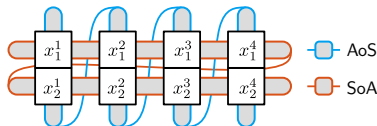
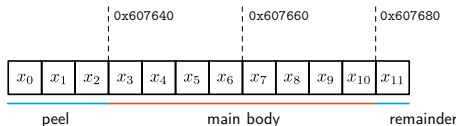
■ Tuned vectorized integration over a square

```

1 double * w = (double *) _mm_malloc( S * sizeof(double), 64 );
2 w[ 0 ] = ... // init. weights
3 // x^1_1, ..., x^S_1
4 double x1 [ ] __attribute__(( aligned( 64 ) )) = { ... };
5 // x^1_2, ..., x^S_2
6 double x2 [ ] __attribute__(( aligned( 64 ) )) = { ... };
7
8 __assume_aligned( w, 64 ); // tell compiler about alignment
9
10 #pragma omp simd reduction( + : result )
11 for( int l = 0; l < S; ++l ){
12     result += w[ l ] * f( x1[ l ], x2[ l ] );
13 }
14
15 _mm_free( w );

```

■ Loop peeling & strided access to memory

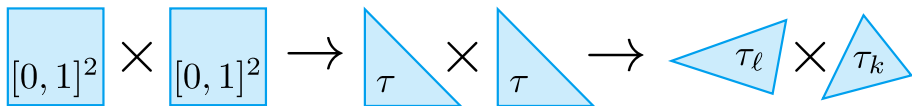


Duffy substitution for $\tau_\ell \times \tau_k$,

$$V_h[l, k] = \sum_s \int_0^1 \int_0^1 \int_0^1 \int_0^1 k(\mathbf{F}^s(\eta_1, \eta_2, \eta_3, \xi)) \mathbf{S}^s(\eta_1, \eta_2, \eta_3, \xi) d\eta_1 d\eta_2 d\eta_3 d\xi$$

with $\mathbf{F}^s: [0, 1]^4 \rightarrow S \subset \tau_\ell \times \tau_k$,

$$\mathbf{F}^s(\eta_1, \eta_2, \eta_3, \xi) = (\mathbf{x}, \mathbf{y}), \quad \mathbf{S}^s(\eta_1, \eta_2, \eta_3, \xi) d\eta_1 d\eta_2 d\eta_3 d\xi = d\mathbf{s}_x d\mathbf{s}_y.$$



Approximated by tensor Gauss quadrature

$$V_h[l, k] \approx \sum_s \sum_m w_m \sum_n w_n \sum_o w_o \sum_p w_p k(\mathbf{F}^s(x_m, x_n, x_o, x_p)) \mathbf{S}^s(x_m, x_n, x_o, x_p).$$

- Collapsed integration loop in getLocalMatrix.

```

1  __assume_aligned( x1ss, 64 ); // all data aligned
2  ...
3
4  switch( type ){
5      case( identicalElements ):
6          for( int simplex = 0; simplex < 6; ++simplex ){
7
8              refToTri( simplex, x1, ..., y3, x1ref, ..., y2ref, x1ss,
9                  ..., y3ss );
10
11 #pragma omp simd reduction( + : entry )
12     for ( c = 0; c < S1*S2*S3*S4; ++c ) { // collapsed
13         kernel = weights_jacV[ c ]
14             * evalSingleLayerKernel( x1ss[ c ], x2ss[ c ],
15                 x3ss[ c ], y1ss[ c ], y2ss[ c ], y3ss[ c ] );
16         entry += kernel;
17     } }
18     break;
19     ... // quadrature over other pairs of elements
20 }

```

■ SIMD evaluation of quadrature points in refToTri.

```

1  __assume_aligned( x1ss, 64 ); // all data aligned
2  ...
3
4  #pragma omp simd
5  for( int c = 0; c < S1*S2*S3*S4; ++c ){
6    x1ss[ c ] = x1[ 0 ]
7      + ( x2[ 0 ] - x1[ 0 ] ) * x1ref[ simplex ][ c ]
8      + ( x3[ 0 ] - x1[ 0 ] ) * x2ref[ simplex ][ c ];
9      ... // compute x2ss, x3ss, y1ss, y2ss, y3ss
10 }

```

■ SIMD evaluation of the kernel in evalSingleLayerKernel.

```

1  #pragma omp declare simd
2  double evalSingleLayerKernel(
3    double x1, double x2, double x3,
4    double y1, double y2, double y3
5  ) const {
6
7    double d1 = x1 - y1, d2 = x2 - y2, d3 = x3 - y3;
8    double norm = sqrt( d1 * d1 + d2 * d2 + d3 * d3 );
9
10   return ( 1 / ( norm * 4.0 * M_PI ) );
11 }

```

- SIMD evaluation of quadrature points in refToTri.

```

1  __assume_aligned( x1ss, 64 ); // all data aligned
2  ...
3
4  #pragma omp simd
5  for( int c = 0; c < S1*S2*S3*S4; ++c ){
6    x1ss[ c ] = x1[ 0 ]
7      + ( x2[ 0 ] - x1[ 0 ] ) * x1ref[ simplex ][ c ]
8      + ( x3[ 0 ] - x1[ 0 ] ) * x2ref[ simplex ][ c ];
9      ... // compute x2ss, x3ss, y1ss, y2ss, y3ss
10 }

```

- SIMD evaluation of the kernel in evalSingleLayerKernel.

```

1  #pragma omp declare simd
2  double evalSingleLayerKernel(
3    double x1, double x2, double x3,
4    double y1, double y2, double y3
5  ) const {
6
7    double d1 = x1 - y1, d2 = x2 - y2, d3 = x3 - y3;
8    double norm = sqrt( d1 * d1 + d2 * d2 + d3 * d3 );
9
10   return ( 1 / ( norm * 4.0 * M_PI ) );
11 }

```

Intel Advisor

Intel Advisor 2017 interface showing vectorization analysis for a BEM4 project.

File View Help
Welcome e000 x

Vectorization Workflow Threading Workflow
Elapsed time: 96,74s Vectorized Not Vectorized FILTER: All Modules All Sources Loops All Threads OFF Smart Mode

Summary Survey Report Refinement Reports

Function Call Sites and Loops

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops	Efficiency	Gain	VL	Traits
[loop in bem4i::BEIntegrator<int, double, b...		42,548s	48,415s	Vectorized (Bo...	1 pragma supersede...	AVX	100%	5,51x	4	Divisions;
[loop in bem4i::BEIntegrator<int, double, bem...	1 Vector regis...	38,379s	45,862s	Vectorized (Body)	1 pragma supersedes...	AVX	100%	4,67x	4	Divisions;
[loop in bem4i::BEIntegrator<int, double, bem...		17,229s	17,229s	Vectorized Versi...	1 pragma supersedes...	AVX	100%	7,20x	4	
[loop in bem4i::BEIntegrator<int, double, bem...		14,836s	14,836s	Vectorized Versi...	1 pragma supersedes...	AVX	100%	7,20x	4	
[loop in bem4i::BEIntegrator<int, double, bem...		0,676s	0,676s	Scalar	loop with multiple e...					
[loop in bem4i::BEIntegrator<int, double, bem...		0,567s	62,500s	Scalar	compile time constr...					Extracts; Ir
[loop in bem4i::BEIntegrator<int, double, bem...		0,522s	67,885s	Scalar	compile time constr...					Inserts; Un
[loop in bem4i::BEIntegrator<int, double, bem...		0,501s	0,501s	Scalar	loop with multiple e...					
[loop in bem4i::BEIntegrator<int, double, bem...		0,240s	0,916s	Scalar	loop control flow is t...					
[loop in bem4i::BEIntegrator<int, double, bem...		0,153s	0,153s	Scalar	loop control flow is t...					
[loop in bem4i::BEIntegrator<int, double, bem...		0,150s	0,651s	Scalar	loop control flow is t...					

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

File: BEIntegratorScalar.cpp:304 bem4i::BEIntegrator<int, double>::computeElemMatrix1LayerSautersSchwabP0P0

Line	Source	Total Time	%	Loop Time	%	Traits
298						
299	#if defined(__INTEL_COMPILER) __INTEL_COMPILER >= 1600					
300	#pragma omp simd linear(i : 1) reduction(+ : entry)					
301	#elif defined(__INTEL_COMPILER) && __INTEL_COMPILER >= 1500					
302	#pragma simd linear(i : 1) reduction(+ : entry)					
303	#endif					
304	for (i = 0; i < totalSize; ++i) {	7,001s		48,414s		
305	entry += thisIntegrator->evalSingleLayerKernel(x1ss[i], x2ss[i],	6,006s				Division...
306	x3ss[i], y1ss[i], y2ss[i], y3ss[i]) * w0[i] * w1[i] *	23,467s				
307	w2[i] * w3[i] * jacobian[i];	12,146s				
308	}					
309	}					
310						
311	SCVT innerArea =					
312	this->getSpace()->getRightMesh()->getElemArea(innerElem);	0,053s				
313	SCVT outerArea =					
	Selected (Total Time):	23,467s				

2.1 Check Dependencies
2.2 Check Memory Access

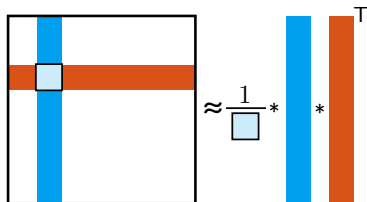
Intel compiler reports

- `-qopt-report=5 -qopt-report-phase=vec`

```
1 LOOP BEGIN at BEIntegratorScalar.cpp(304,5)
2   remark 15340: pragma supersedes previous setting [
3     BEIntegratorScalar.cpp(300,1) ]
4   remark 15388: vectorization support: reference x1ss[i] has
5     aligned access [ BEIntegratorScalar.cpp(305,55) ]
6   ... // all data reported as aligned
7   remark 15305: vectorization support: vector length 4
8   remark 15309: vectorization support: normalized
9     vectorization overhead 0.443
10  remark 15301: OpenMP SIMD LOOP WAS VECTORIZED
11  remark 15448: unmasked aligned unit stride loads: 16
12  remark 15475: --- begin vector cost summary ---
13  remark 15476: scalar cost: 97
14  remark 15477: vector cost: 17.500
15  remark 15478: estimated potential speedup: 5.510
16  remark 15488: --- end vector cost summary ---
17 LOOP END
```


Adaptive cross approximation

- Complexity $\mathcal{O}(n^2) \rightarrow \mathcal{O}(n \log n)$,
- mesh divided into clusters,
- *non-admissible* clusters assembled in full,
- *admissible* clusters approximated as $C \approx UV^T$,
- assembly of clusters distributed by OpenMP.



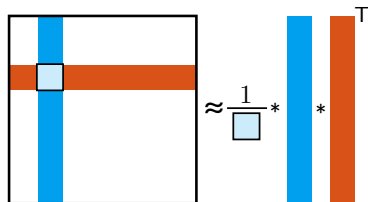
```

1 #pragma omp parallel for
2 for( int i = 0; i < n_nonadmissible_blocks; ++i ){
3     getNonadmissibleBlock( i, localBlock );
4     ACAMatrix.addNonadmissibleBlock( i, localBlock );
5 }
6
7 #pragma omp parallel for
8 for( int i = 0; i < n_admissible_blocks; ++i ){
9     getAdmissibleBlock( i, localBlock );
10    ACAMatrix.addAdmissibleBlock( i, localBlock );
11 }

```

Adaptive cross approximation

- Complexity $\mathcal{O}(n^2) \rightarrow \mathcal{O}(n \log n)$,
- mesh divided into clusters,
- non-admissible* clusters assembled in full,
- admissible* clusters approximated as $C \approx UV^T$,
- assembly of clusters distributed by OpenMP.



```

1 #pragma omp parallel for
2 for( int i = 0; i < n_nonadmissible_blocks; ++i ){
3     getNonadmissibleBlock( i, localBlock );
4     ACAMatrix.addNonadmissibleBlock( i, localBlock );
5 }
6
7 #pragma omp parallel for
8 for( int i = 0; i < n_admissible_blocks; ++i ){
9     getAdmissibleBlock( i, localBlock );
10    ACAMatrix.addAdmissibleBlock( i, localBlock );
11 }

```

- 1 BEM4I
 - Boundary element method
 - OpenMP threading
 - OpenMP vectorization
 - Adaptive cross approximation

- 2 Numerical experiments
 - Full assembly
 - ACA assembly

- 3 Conclusion

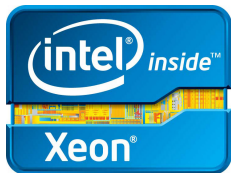
Experiment setting

- Full (ACA) assembly tested on a mesh with 20.480 (81.920) surface elements.
- 4 quadrature points in each dimension (256 per simplex) to utilize SIMD registers.
- ACA settings
 - maximal number of elements in clusters: 500,
 - preallocation: 10 %.

- Assembly performed on single nodes
 - Salomon - 2 × Xeon 2680v3, AVX2, 2x12 cores, 2.5 GHz, 128 GB RAM,
 - Salomon - Xeon Phi 7120P, IMCI, 61 cores, 1.238 GHz, 16 GB RAM,
 - Endeavor - Xeon Phi 7210, AVX-512, 64 cores, 1.3 GHz, 16 + 96 GB RAM.

Experiment setting

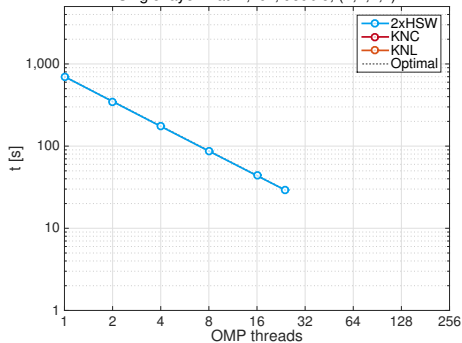
- Full (ACA) assembly tested on a mesh with 20.480 (81.920) surface elements.
- 4 quadrature points in each dimension (256 per simplex) to utilize SIMD registers.
- ACA settings
 - maximal number of elements in clusters: 500,
 - preallocation: 10 %.



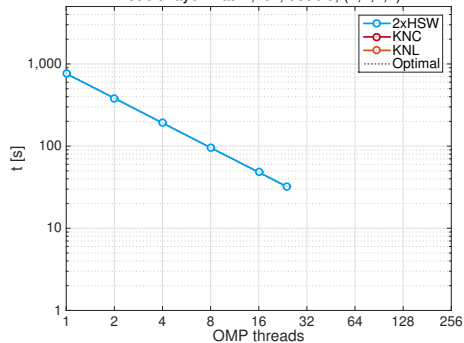
- Assembly performed on single nodes
 - Salomon - 2 × **Xeon 2680v3**, AVX2, 2x12 cores, 2.5 GHz, 128 GB RAM,
 - Salomon - **Xeon Phi 7120P**, IMCI, 61 cores, 1.238 GHz, 16 GB RAM,
 - Endeavor - **Xeon Phi 7210**, AVX-512, 64 cores, 1.3 GHz, 16 + 96 GB RAM.

OMP scalability, full assembly, 256 points per simplex

Single-layer matrix, full, double, (4,4,4,4)



Double-layer matrix, full, double, (4,4,4,4)

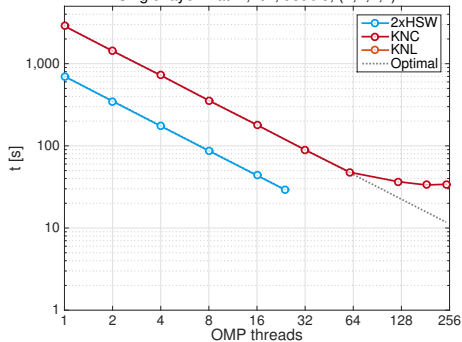


Xeon 2680v3

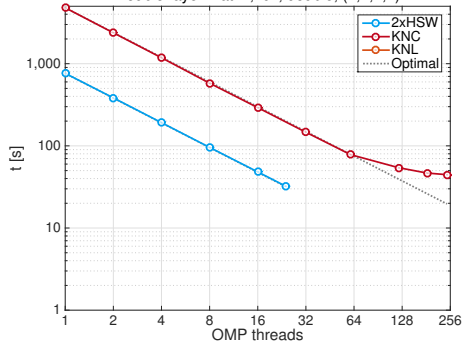
matrix	2	4	8	16	24
V_h	2.01	4.04	8.07	16.09	24.07
K_h	1.99	3.98	7.97	15.83	23.68

OMP scalability, full assembly, 256 points per simplex

Single-layer matrix, full, double, (4,4,4,4)



Double-layer matrix, full, double, (4,4,4,4)

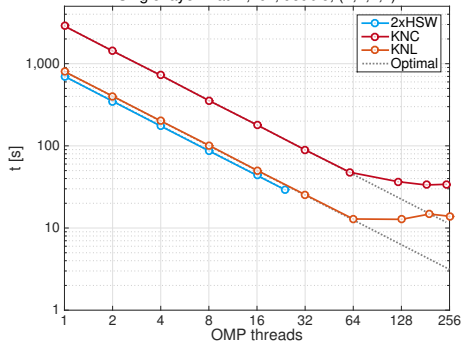


Xeon Phi 7120P

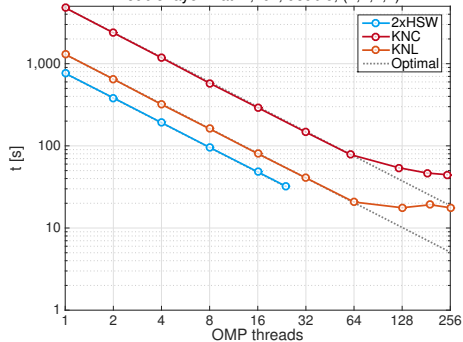
matrix	2	4	8	16	32	61	122	183	244
V_h	2.01	3.98	8.06	15.96	32.14	60.23	78.33	85.90	84.29
K_h	2.02	4.06	8.28	16.42	32.72	61.38	88.54	102.71	107.53

OMP scalability, full assembly, 256 points per simplex

Single-layer matrix, full, double, (4,4,4,4)



Double-layer matrix, full, double, (4,4,4,4)

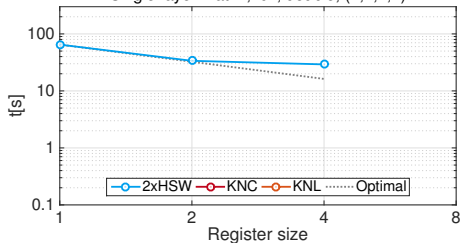


Xeon Phi 7210

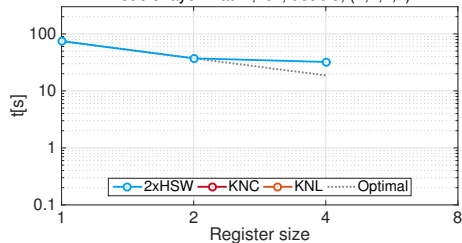
matrix	2	4	8	16	32	64	128	192	256
V_h	1.99	3.99	7.90	15.93	31.48	62.26	62.89	54.22	57.65
K_h	2.00	4.07	8.07	16.20	31.89	62.76	73.64	67.76	73.64

SIMD scalability, full assembly, 256 points per simplex

Single-layer matrix, full, double, (4,4,4,4)



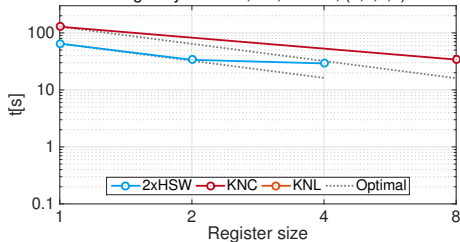
Double-layer matrix, full, double, (4,4,4,4)



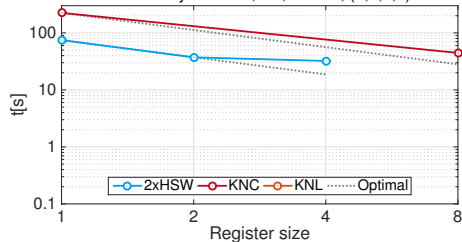
architecture	threads	matrix	SSE4.2	AVX2	IMCI	AVX-512
Xeon E2680v3	24	V_h	1.92	2.23	—	—
		K_h	2.01	2.32	—	—
Xeon Phi 7120P	244	V_h	—	—	3.77	—
		K_h	—	—	5.05	—
Xeon Phi 7210	128	V_h	2.45	4.95	—	9.94
		K_h	2.01	4.14	—	7.69

SIMD scalability, full assembly, 256 points per simplex

Single-layer matrix, full, double, (4,4,4,4)



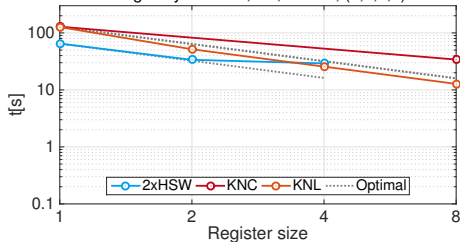
Double-layer matrix, full, double, (4,4,4,4)



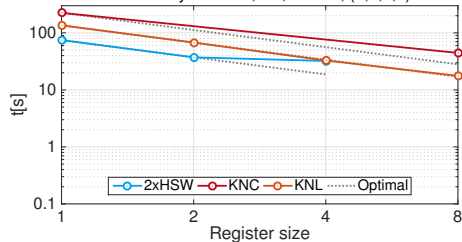
architecture	threads	matrix	SSE4.2	AVX2	IMCI	AVX-512
Xeon E2680v3	24	V_h	1.92	2.23	—	—
		K_h	2.01	2.32	—	—
Xeon Phi 7120P	244	V_h	—	—	3.77	—
		K_h	—	—	5.05	—
Xeon Phi 7210	128	V_h	2.45	4.95	—	9.94
		K_h	2.01	4.14	—	7.69

SIMD scalability, full assembly, 256 points per simplex

Single-layer matrix, full, double, (4,4,4,4)



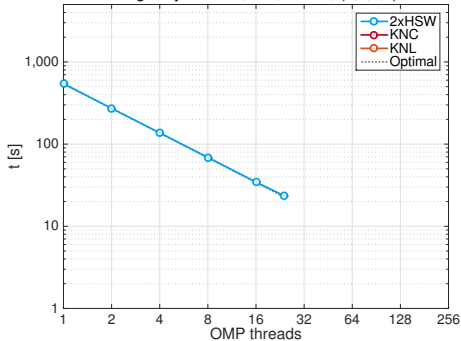
Double-layer matrix, full, double, (4,4,4,4)



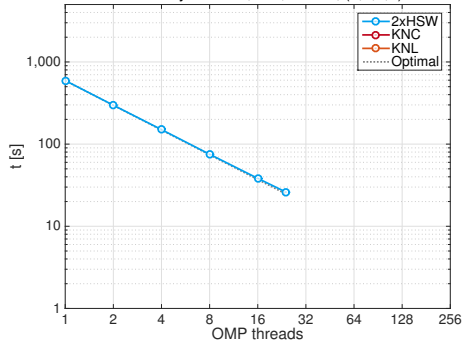
architecture	threads	matrix	SSE4.2	AVX2	IMCI	AVX-512
Xeon E2680v3	24	V_h	1.92	2.23	—	—
		K_h	2.01	2.32	—	—
Xeon Phi 7120P	244	V_h	—	—	3.77	—
		K_h	—	—	5.05	—
Xeon Phi 7210	128	V_h	2.45	4.95	—	9.94
		K_h	2.01	4.14	—	7.69

OMP scalability, ACA assembly, 256 points per simplex

Single-layer matrix, aca, double, (4,4,4,4)



Double-layer matrix, aca, double, (4,4,4,4)

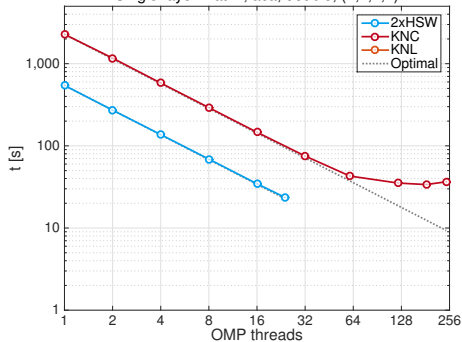


Xeon 2680v3

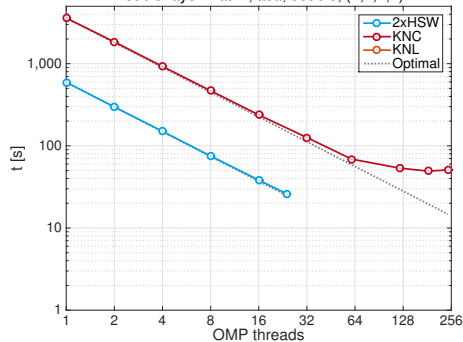
matrix	2	4	8	16	24
V_h	2.00	3.97	7.89	15.74	23.13
K_h	1.99	3.95	7.89	15.49	22.75

OMP scalability, ACA assembly, 256 points per simplex

Single-layer matrix, aca, double, (4,4,4,4)



Double-layer matrix, aca, double, (4,4,4,4)

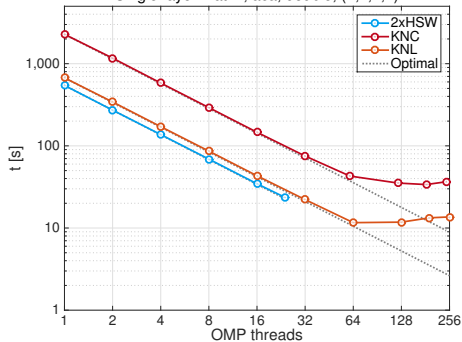


Xeon Phi 7120P

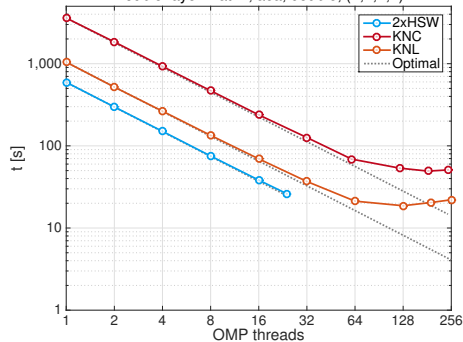
matrix	2	4	8	16	32	61	122	183	244
V_h	1.95	3.92	7.80	15.57	30.32	53.16	64.69	67.44	62.39
K_h	1.95	3.88	7.67	15.10	28.49	52.17	67.20	72.94	70.66

OMP scalability, ACA assembly, 256 points per simplex

Single-layer matrix, aca, double, (4,4,4,4)



Double-layer matrix, aca, double, (4,4,4,4)

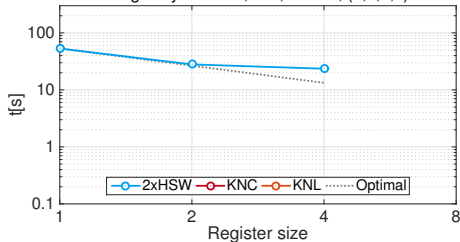


Xeon Phi 7210

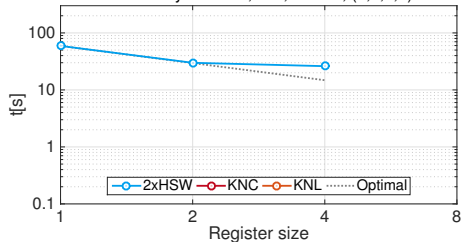
matrix	2	4	8	16	32	64	128	192	256
V_h	1.97	3.93	7.80	15.57	30.36	57.92	57.08	50.57	49.34
K_h	2.01	3.95	7.78	15.03	28.34	49.17	55.92	51.02	47.30

SIMD scalability, ACA assembly, 256 points per simplex

Single-layer matrix, aca, double, (4,4,4,4)



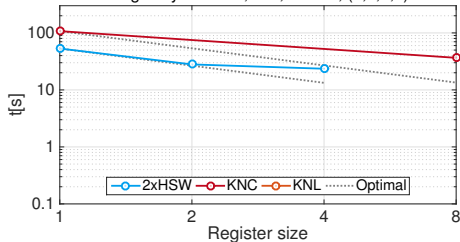
Double-layer matrix, aca, double, (4,4,4,4)



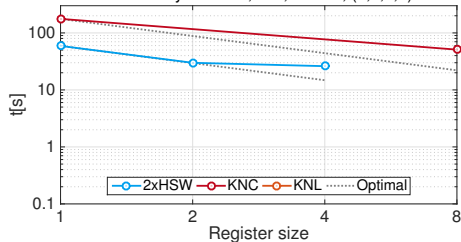
architecture	threads	matrix	SSE4.2	AVX2	IMCI	AVX-512
Xeon E2680v3	24	V_h	1.90	2.26	—	—
		K_h	1.98	2.27	—	—
Xeon Phi 7120P	244	V_h	—	—	2.94	—
		K_h	—	—	3.47	—
Xeon Phi 7210	128	V_h	2.33	4.45	—	8.06
		K_h	1.96	3.71	—	5.43

SIMD scalability, ACA assembly, 256 points per simplex

Single-layer matrix, aca, double, (4,4,4,4)



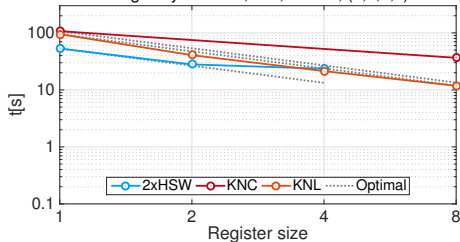
Double-layer matrix, aca, double, (4,4,4,4)



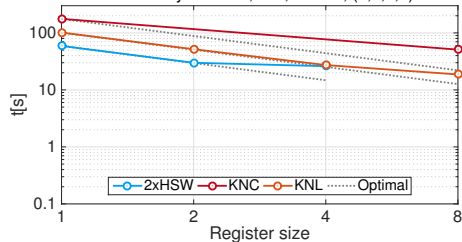
architecture	threads	matrix	SSE4.2	AVX2	IMCI	AVX-512
Xeon E2680v3	24	V_h	1.90	2.26	—	—
		K_h	1.98	2.27	—	—
Xeon Phi 7120P	244	V_h	—	—	2.94	—
		K_h	—	—	3.47	—
Xeon Phi 7210	128	V_h	2.33	4.45	—	8.06
		K_h	1.96	3.71	—	5.43

SIMD scalability, ACA assembly, 256 points per simplex

Single-layer matrix, aca, double, (4,4,4,4)



Double-layer matrix, aca, double, (4,4,4,4)



architecture	threads	matrix	SSE4.2	AVX2	IMCI	AVX-512
Xeon E2680v3	24	V_h	1.90	2.26	—	—
		K_h	1.98	2.27	—	—
Xeon Phi 7120P	244	V_h	—	—	2.94	—
		K_h	—	—	3.47	—
Xeon Phi 7210	128	V_h	2.33	4.45	—	8.06
		K_h	1.96	3.71	—	5.43

- 1 BEM4I
 - Boundary element method
 - OpenMP threading
 - OpenMP vectorization
 - Adaptive cross approximation

- 2 Numerical experiments
 - Full assembly
 - ACA assembly

- 3 Conclusion

Conclusion

■ KNL vs. HSW speedup

- Full – V_h 2.29, K_h 1.82,
- ACA – V_h 2.64, K_h 1.40.

■ What we learned

- SIMD processing becoming more efficient with KNL,
- multi-core code benefits from many-core optimizations.

■ Work in progress

- object-oriented offload to Xeon Phi (full, ACA),
- load balancing for offload mode (full, ACA),
- massively parallel BETI with ESPRESO library (MPI, OpenMP, SIMD, offload).

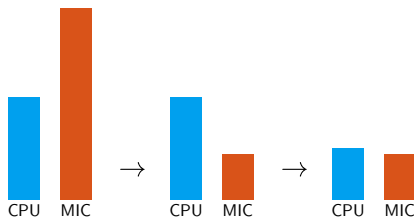
Conclusion

■ KNL vs. HSW speedup

- Full – V_h 2.29, K_h 1.82,
- ACA – V_h 2.64, K_h 1.40.

■ What we learned

- SIMD processing becoming more efficient with KNL,
- multi-core code benefits from many-core optimizations.



■ Work in progress

- object-oriented offload to Xeon Phi (full, ACA),
- load balancing for offload mode (full, ACA),
- massively parallel BETI with ESPRESSO library (MPI, OpenMP, SIMD, offload).

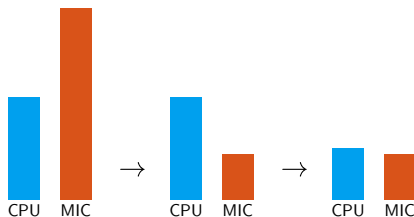
Conclusion

■ KNL vs. HSW speedup

- Full – V_h 2.29, K_h 1.82,
- ACA – V_h 2.64, K_h 1.40.

■ What we learned

- SIMD processing becoming more efficient with KNL,
- multi-core code benefits from many-core optimizations.



■ Work in progress

- object-oriented offload to Xeon Phi (full, ACA),
- load balancing for offload mode (full, ACA),
- massively parallel BETI with ESPRESSO library (MPI, OpenMP, SIMD, offload).

References



Zapletal, J.; Merta, M.; Malý, L.

Boundary element quadrature schemes for multi- and many-core architectures.
Computers and Mathematics with Applications (accepted).



Merta, M.; Zapletal, J.; Jaroš, J.

Many core acceleration of the boundary element method.

High Performance Computing in Science and Engineering: Second International Conference, HPCSE 2015, Soláň, Czech Republic, May 25-28, 2015, Revised Selected Papers, Springer International Publishing 14:116–125, 2016.

Preprints available at **researchgate.com**.

Thank you

References



Zapletal, J.; Merta, M.; Malý, L.

Boundary element quadrature schemes for multi- and many-core architectures.
Computers and Mathematics with Applications (accepted).



Merta, M.; Zapletal, J.; Jaroš, J.

Many core acceleration of the boundary element method.

High Performance Computing in Science and Engineering: Second International Conference, HPCSE 2015, Soláň, Czech Republic, May 25-28, 2015, Revised Selected Papers, Springer International Publishing 14:116–125, 2016.

Preprints available at **researchgate.com**.

Thank you