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

Jan Zapletal[a,b,*], Michal Merta[a], Lukáš Malý[a,b]

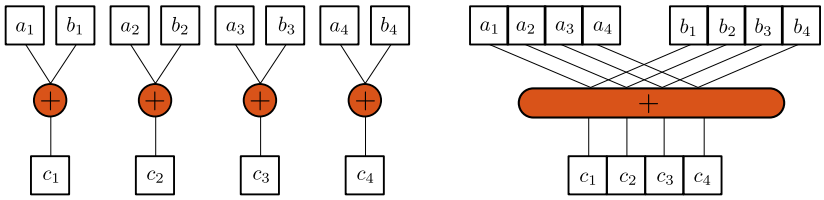
[a] *IT4Innovations, VŠB – Technical University of Ostrava. 17. listopadu 2172/15, 708 33 Ostrava-Poruba, Czech Republic*
[b] *Department of Applied Mathematics, VŠB – Technical University of Ostrava. 17. listopadu 2172/15, 708 33 Ostrava-Poruba, Czech Republic*

Abstract

In the paper we study the performance of the regularized boundary element quadrature routines implemented in the BEM4I library developed by the authors. Apart from the results obtained on the classical multi-core architecture represented by the Intel Xeon processors we concentrate on the portability of the code to the many-core family Intel Xeon Phi. Contrary to the GP-GPU programming accelerating many scientific codes, the standard x86 architecture of the Xeon Phi processors allows to reuse the already existing multi-core implementation. Although in many cases a simple recompilation would lead to an inefficient utilization of the Xeon Phi, the effort invested in the optimization usually leads to a better performance on the multi-core Xeon processors as well. This makes the Xeon Phi an interesting platform for scientists developing a software library aimed at both modern portable PCs and high performance computing environments. Here we focus at the manually vectorized assembly of the local element contributions and the parallel assembly of the global matrices on shared memory systems. Due to the quadratic complexity of the standard assembly we also present an assembly sparsified by the adaptive cross approximation based on the same acceleration techniques. The numerical results performed on the Xeon multi-core processor and two generations of the Xeon Phi many-core platform validate the proposed implementation and highlight the importance of vectorization necessary to exploit the features of modern hardware.

Keywords: boundary element method, quadrature, SIMD, vectorization, Intel Xeon Phi, many-core architecture
2010 MSC: 65N38, 65Y05, 68W10

*Corresponding author
Email address: jan.zapletal@vsb.cz (Jan Zapletal)



(a) Scalar addition. (b) SIMD addition.

Figure 1.1: Scalar and vectorized addition of two vectors $\boldsymbol{c} := \boldsymbol{a} + \boldsymbol{b}$.

1. Introduction

An efficient implementation of the numerical quadrature routines is an inseparable part of any boundary element software. Contrary to the finite element method, where the integrands are often well-behaved polynomial functions and the integrals can sometimes be evaluated analytically, in the boundary element method (BEM) one has to deal with weakly singular integrals. Another price to pay for the dimension reduction is the quadratic complexity of the standard BEM both in terms of the computational time and memory requirements restricting its applicability to moderate problem dimensions. To overcome this issue, several fast BEM methods can be employed to lower the complexity to almost linear. This includes the fast multipole method [1, 2, 3] based on the approximation of the kernel by a truncated series, or the adaptive cross approximation (ACA) [4, 5] building low-rank blocks based on an algebraic point of view. Although these sparsification methods are inevitable for large-scale engineering problems, it is still crucial to efficiently assemble the so-called non-admissible full blocks. Moreover, in the case of ACA, the low-rank approximation requires the evaluation of several rows and columns of every admissible block, which relies on the standard full assembly. The above mentioned approaches can be combined with a domain decomposition method, such as the boundary element tearing and interconnecting method [6, 7, 8]. Although these techniques allow for a massively parallel implementation in distributed memory systems, they still rely on an efficient intra-node implementation with an extra layer of inter-node parallelization.

In the paper we concentrate on the acceleration of the standard BEM assembly in shared memory and its acceleration by ACA. While the parallelization of similar scientific codes at the level of CPU cores has become standard, the feature that is still often overlooked is the in-core vectorization supporting the Single Instruction Multiple Data (SIMD) concept, see Figure 1.1. Since the clock frequency of modern CPUs has not been growing as rapidly as in the previous decades, the increasing advertised theoretical performance limits can only be reached by utilizing all available parallelization techniques.

In 1999 Intel issued the SSE (Streaming SIMD Extensions) instruction set capable of concurrent processing of four 32-bit single-precision floating-point numbers. This feature has since been extended by the sets SSE2-4.2 adding more

instructions and the ability to operate on two 64-bit double-precision operands. The AVX and AVX2 (Advanced Vector Extensions) instruction sets increase the register size from 128 bits to 256 bits. While the first commercially available Intel Many Integrated Core (MIC) architecture Knights Corner (KNC) supports the MIC specific Initial Many Core Instruction (IMCI) set, the plan for future is to (at least partially) unify the instructions across both multi- and many-core platforms – the upcoming Skylake and Knights Landing architectures are to include the new AVX-512 instruction set with 512-bit registers able to accommodate eight 64-bit double-precision operands.

The vectorization can be achieved by various techniques. The direct use of vector instructions can be utilized using the inline assembler, i.e., the assembly code directly inserted into a high-level programming language. Such code is, however, hard to understand and not portable between multiple architectures. A similar possibility is the use of intrinsic functions provided by the vendor of the compiler. These functions produce the corresponding assembly code, leading again to a non-portable code. Another option is to use an external library providing a high-level wrapper of the supported intrinsic functions. This technique leads to a portable code, since the wrapper functions are compiled to the supported vector instruction set. In [9] we describe this approach both for the semi-analytic and fully numerical integration schemes [4] using the Vc library [10]. A more user-friendly way is to use the auto-vectorization capabilities of modern compilers. To help the compiler identify the parts of the code suitable for vectorization the original code usually has to be refactored using techniques such as loop unrolling, tiling, reordering, or collapsing. This has been used by the authors in [11] with additional OpenMP 4.0 [12] pragmas and the accelerators used in the offload mode. Differently from [11], in this paper we concentrate on the native deployment of the code which allows for direct comparison of the capabilities of the available multi- and many-core platforms. Readers interested in multi- and many-core programming are referred to the monographs [13, 14, 15] and a special Knights Landing edition [16] with additional tips on programming on this architecture.

In the context of BEM, the vectorization paradigm has been leveraged in [17], where the authors rely on the automatic vectorization by the Fortran compiler. Although a reasonable speedup is reached, after the loop rearrangement the code is much less readable than the original. In [18] the author explicitly uses the intrinsic functions provided by the compiler to accelerate the generation of BEM matrices. Unfortunately, the treatment of singularities, which represents one of the crucial tasks in BEM and complicates the vectorization, is not discussed. The acceleration of the evaluation of the representation formula on the Intel Xeon Phi coprocessor is described in [19]. For a similar discussion regarding first-order finite element discretization on multi- and many-core architectures including the Xeon Phi coprocessor we also refer to [20].

There are several approaches for integrating the weakly singular kernel functions appearing in the Galerkin boundary element methods. The first approach is based on integral substitutions rendering the integrand analytical via the multiplication with the corresponding Jacobian allowing us to use the standard

tensor product Gaussian quadrature schemes. The advantage of this approach lies in its versatility – the same procedure can be used for a wide class of kernels, including the kernels of the Laplace, Lamé, Stokes, Helmholtz, or Maxwell operators. This method has been studied in, e.g., [21, 22].

The second, a seemingly preferable way, is to compute the inner surface integral analytically, which allows us to treat the limit singular entries. For the outer integral we can use a standard Gaussian quadrature scheme. This approach has been described for the Laplace, Lamé, and Helmholtz equations in [4, 23, 24]. The serious drawback of this method is that tedious work is necessary to evaluate the collocation integral for every combination of the ansatz function and the corresponding kernel. Moreover, it is not clear if the outer Gaussian quadrature converges in an optimal rate due to the non-smooth nature of the collocation integral with discrete ansatz functions.

The paper is organized as follows. In Section 2 we present the model boundary integral equation together with its discretization by means of the boundary element method and briefly describe the regularization approach. In Section 3 the implementation of the four-dimensional quadrature is described together with its optimization to leverage the available vector instructions. The presented approaches build the core of the BEM4I library [25] developed at the IT4Innovations Czech National Supercomputing Center. Numerical experiments validating the proposed techniques are given in Section 4, the concluding remarks follow in Section 5.

2. Boundary element method

In this section we introduce the model problem given by a weakly singular boundary integral equation (BIE). Contrary to the variational formulation leading to the discretization by volume finite elements, the formulation by boundary integral equations reduces the problem to the boundary of the computational domain at the cost of dealing with singular integrals. On the other hand, such formulations are natural for problems in unbounded domains or in the context of shape optimization [26]. For a detailed treatment of BIE and BEM we refer to [22, 27].

2.1. Boundary integral equation

We consider the Dirichlet problem for the Laplace equation,

$$-\Delta u = 0 \text{ in } \Omega, \qquad u = g \text{ on } \partial\Omega, \tag{2.1}$$

with $\Omega \subset \mathbb{R}^3$ denoting a bounded Lipschitz domain and the Dirichlet boundary condition $g \in H^{1/2}(\partial\Omega)$. The solution to (2.1) is given by the representation formula

$$u(\widetilde{\boldsymbol{x}}) = \int_{\partial\Omega} v(\widetilde{\boldsymbol{x}}, \boldsymbol{y}) w(\boldsymbol{y}) \, \mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} - \int_{\partial\Omega} \frac{\partial}{\partial \boldsymbol{n}_{\boldsymbol{y}}} v(\widetilde{\boldsymbol{x}}, \boldsymbol{y}) u(\boldsymbol{y}) \, \mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \quad \text{for } \widetilde{\boldsymbol{x}} \in \Omega \tag{2.2}$$

with $w := \partial u/\partial \boldsymbol{n}$ and the fundamental solution of the Laplacian in three dimensions

$$v(\boldsymbol{x}, \boldsymbol{y}) := \frac{1}{4\pi} \frac{1}{\|\boldsymbol{x} - \boldsymbol{y}\|}.$$

To determine the unknown Neumann data w we take the limit $\Omega \ni \widetilde{\boldsymbol{x}} \to \boldsymbol{x} \in \partial\Omega$ in (2.2) to obtain the boundary integral equation

$$Vw(\boldsymbol{x}) = \frac{1}{2} u(\boldsymbol{x}) + Ku(\boldsymbol{x}) \quad \text{for } \boldsymbol{x} \in \partial\Omega \tag{2.3}$$

with the single- and double-layer boundary integral operators

$$\begin{aligned} V &\colon H^{-1/2}(\partial\Omega) \to H^{1/2}(\partial\Omega), \qquad Vw(\boldsymbol{x}) := \int_{\partial\Omega} v(\boldsymbol{x}, \boldsymbol{y}) w(\boldsymbol{y}) \, \mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}, \\ K &\colon H^{1/2}(\partial\Omega) \to H^{1/2}(\partial\Omega), \qquad Ku(\boldsymbol{x}) := \int_{\partial\Omega} \frac{\partial}{\partial \boldsymbol{n}_{\boldsymbol{y}}} v(\boldsymbol{x}, \boldsymbol{y}) u(\boldsymbol{y}) \, \mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}, \end{aligned}$$

respectively. Both boundary integral operators are linear and bounded. In addition, $H^{-1/2}(\partial\Omega)$-ellipticity of the operator V ensures unique solvability of (2.3). The equivalent Galerkin formulation of the equation (2.3) considered in the following text reads

$$\langle t, Vw \rangle_{\partial\Omega} = \left\langle t, \left(\frac{1}{2} I + K \right) u \right\rangle_{\partial\Omega} \quad \text{for all } t \in H^{-1/2}(\partial\Omega) \tag{2.4}$$

with the $L^2(\partial\Omega)$-based duality pairing $\langle \cdot, \cdot \rangle_{\partial\Omega}$.

2.2. Boundary element method

To discretize the variational problem (2.4) we first decompose the polygonal boundary $\partial\Omega$ into E planar triangular shape-regular boundary elements τ_k with N mesh nodes $\boldsymbol{x}^i$. Since we consider conforming discretization, we introduce two discrete spaces

$$H^{-1/2}(\partial\Omega) \supset S_h^0(\partial\Omega) := \operatorname{span}\{\psi_k\}_{k=1}^E, \quad H^{1/2}(\partial\Omega) \supset S_h^1(\partial\Omega) := \operatorname{span}\{\varphi_i\}_{i=1}^N$$

of piecewise constant and globally continuous piecewise linear functions

$$\psi_k(\boldsymbol{x}) := \begin{cases} 1 & \text{for } \boldsymbol{x} \in \tau_k, \\ 0 & \text{otherwise,} \end{cases} \quad \varphi_i(\boldsymbol{x}) := \begin{cases} 1 & \text{for } \boldsymbol{x} = \boldsymbol{x}^i, \\ 0 & \text{for } \boldsymbol{x} = \boldsymbol{x}^j \neq \boldsymbol{x}^i \\ \text{piecewise linear} & \text{otherwise,} \end{cases} \tag{2.5}$$

respectively. Inserting the approximation of the boundary data

$$w \approx w_h := \sum_{k=1}^E w_k \psi_k, \qquad u \approx u_h := \sum_{i=1}^N u_i \varphi_i$$

into (2.4) and testing with piecewise constant functions ψ_ℓ we obtain the system of linear equations

$$\mathsf{V}_h \boldsymbol{w} = \left(\frac{1}{2}\mathsf{M}_h + \mathsf{K}_h\right) \boldsymbol{u}$$

with the boundary element matrices

$$[\mathsf{V}_h]_{\ell,k} := \frac{1}{4\pi} \int_{\tau_\ell} \int_{\tau_k} \frac{1}{\|\boldsymbol{x}-\boldsymbol{y}\|} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \tag{2.6}$$

$$[\mathsf{K}_h]_{\ell,i} := \frac{1}{4\pi} \int_{\tau_\ell} \int_{\partial\Omega} \frac{\langle \boldsymbol{x}-\boldsymbol{y}, \boldsymbol{n}(\boldsymbol{y})\rangle}{\|\boldsymbol{x}-\boldsymbol{y}\|^3} \varphi_i(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}, \tag{2.7}$$

$$[\mathsf{M}_h]_{\ell,i} = \int_{\tau_\ell} \varphi_i(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}.$$

The matrix M_h, i.e., the Galerkin approximation of the identity operator, is sparse and its action can be performed on the fly without any significant overhead. In the following we thus concentrate on the efficient assembly of the remaining dense matrices V_h, K_h. Let us also mention that for the discretization of the hypersingular boundary integral equation for the treatment of the Neumann (or mixed) problem an additional matrix would have to be assembled, namely

$$[\mathsf{D}_h]_{j,i} := \sum_{\tau_\ell \subset \operatorname{supp}\varphi_j} \sum_{\tau_k \subset \operatorname{supp}\varphi_i} \langle \mathbf{curl}_{\partial\Omega}\, \varphi_i|_{\tau_k}(\boldsymbol{y}), \mathbf{curl}_{\partial\Omega}\, \varphi_j|_{\tau_\ell}(\boldsymbol{x})\rangle [\mathsf{V}_h]_{\ell,k}$$

which in our setting is given by a sparse transformation of the single-layer matrix V_h. The optimized matrix assembly described below thus covers a wide range of boundary value problems encountered in various engineering problems.

2.3. Regularized quadrature scheme

The following exposition is valid for a rather large family of boundary integral operators including the Laplace single- and double-layer kernels, for more details consult [22, Chapter 5].

Both operators V and K can be represented by an abstract operator

$$Au(\boldsymbol{x}) := \int_{\partial\Omega} k(\boldsymbol{y}, \boldsymbol{x}-\boldsymbol{y}) u(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}}, \quad \boldsymbol{x} \in \partial\Omega.$$

The Galerkin formulation and subsequent discretization with ansatz and test functions ϕ_j^{a} and ϕ_i^{t}, respectively, leads to the matrix

$$[\mathsf{A}_h]_{i,j} := \sum_{\tau_k \subset \operatorname{supp}\phi_i^{\mathrm{t}}} \sum_{\tau_\ell \subset \operatorname{supp}\phi_j^{\mathrm{a}}} \int_{\tau_k} \int_{\tau_\ell} k(\boldsymbol{y}, \boldsymbol{x}-\boldsymbol{y}) \phi_i^{\mathrm{t}}(\boldsymbol{x}) \phi_j^{\mathrm{a}}(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}.$$

To assemble A_h we thus have to evaluate the element contributions

$$I := \int_\tau \int_\tau \hat{k}(\boldsymbol{\nu}, \boldsymbol{\mu}) \,\mathrm{d}\boldsymbol{\mu} \,\mathrm{d}\boldsymbol{\nu}$$

transferred to the reference domain τ,

$$\tau := \{\boldsymbol{\mu} \in \mathbb{R}^2 \colon \mu_1 \in (0,1),\, \mu_2 \in (0,\mu_1)\},$$

by the linear mappings defined by the nodes of τ_k, τ_ℓ,

$$\boldsymbol{x} = \boldsymbol{R}^k(\boldsymbol{\nu}) := \boldsymbol{x}^{k,1} + \begin{bmatrix}\boldsymbol{x}^{k,2} - \boldsymbol{x}^{k,1} & \boldsymbol{x}^{k,3} - \boldsymbol{x}^{k,2}\end{bmatrix} \boldsymbol{\nu}, \qquad \boldsymbol{y} = \boldsymbol{R}^\ell(\boldsymbol{\mu}). \tag{2.8}$$

The transported kernel multiplied by the Jacobian and the reference (constant or linear) ansatz and test functions reads

$$\hat{k}(\boldsymbol{\nu},\boldsymbol{\mu}) := 4\Delta_{\tau_k}\Delta_{\tau_\ell} k\big(\boldsymbol{R}^\ell(\boldsymbol{\mu}), \boldsymbol{R}^k(\boldsymbol{\nu}) - \boldsymbol{R}^\ell(\boldsymbol{\mu})\big)\phi_i^{\mathrm{t}}\big(\boldsymbol{R}^k(\boldsymbol{\nu})\big)\phi_j^{\mathrm{a}}\big(\boldsymbol{R}^\ell(\boldsymbol{\mu})\big), \tag{2.9}$$

where Δ_{τ_k} denotes the surface area of the element τ_k.

The idea is to understand the Cartesian product $\tau \times \tau$ as a four-dimensional object, split it into S simplices with singularities moved to the vertices and use a series of transformations to obtain

$$I = \sum_{s=1}^{S} \int_0^1 \int_0^1 \int_0^1 \int_0^1 \hat{k}(\boldsymbol{F}^s(\eta_1,\eta_2,\eta_3,\xi))\boldsymbol{S}^s(\eta_1,\eta_2,\eta_3,\xi)\,\mathrm{d}\eta_1\,\mathrm{d}\eta_2\,\mathrm{d}\eta_3\,\mathrm{d}\xi \tag{2.10}$$

with the substitutions $\boldsymbol{F}^s \colon [0,1]^4 \to S \subset \tau \times \tau$,

$$\begin{aligned} \boldsymbol{F}^s(\eta_1,\eta_2,\eta_3,\xi) &= (\boldsymbol{\nu},\boldsymbol{\mu}) = (\nu_1,\nu_2,\mu_1,\mu_2), \\ \boldsymbol{S}^s(\eta_1,\eta_2,\eta_3,\xi)\,\mathrm{d}\eta_1\,\mathrm{d}\eta_2\,\mathrm{d}\eta_3\,\mathrm{d}\xi &= \mathrm{d}\boldsymbol{\mu}\,\mathrm{d}\boldsymbol{\nu}, \end{aligned} \tag{2.11}$$

and the integrand $(\hat{k} \circ \boldsymbol{F}^s)\boldsymbol{S}^s$ analytically extensible to the neighbourhood of the simplex. The transformations $\boldsymbol{F}^s$ specified in [22, Chapter 5] are different for τ_k, τ_ℓ being identical, sharing exactly one edge, sharing exactly one vertex, or having a positive distance.

To approximate the four-dimensional integral (2.10) we use a tensor product Gaussian scheme

$$I \approx \sum_{s=1}^{S} \sum_{\ell_1=1}^{S_1} \sum_{\ell_2=1}^{S_2} \sum_{\ell_3=1}^{S_3} \sum_{\ell_4=1}^{S_4} w_{\ell_1} w_{\ell_2} w_{\ell_3} w_{\ell_4} \hat{k}(\boldsymbol{F}^s(\eta_{1,\ell_1},\eta_{2,\ell_2},\eta_{3,\ell_3},\xi_{\ell_4}))\boldsymbol{S}^s(\eta_{1,\ell_1},\eta_{2,\ell_2},\eta_{3,\ell_3},\xi_{\ell_4})$$

with appropriately chosen weights $w_\bullet$ and sampling points $\eta_\bullet$, $\xi_\bullet$, see [28].

2.4. Adaptive cross approximation

Clearly, the assembly of the dense matrices V_h, K_h has quadratic computational and memory complexity with respect to the number of surface degrees of freedom. Several approaches can be employed to lower the complexity to the desired almost linear case. Here we concentrate on the adaptive cross approximation (ACA) as presented, e.g., in [4, 5, 29].

Starting from the whole mesh $C_0 := \bigcup_k \tau_k$ we create two disjoint element clusters C_{11}, C_{12} separated by a plane defined by the center of gravity

$$\boldsymbol{c}(C_0) := \frac{\sum_{\tau_k \in C_0} \Delta_{\tau_k} \boldsymbol{c}^k}{\sum_{\tau_k \in C_0} \Delta_{\tau_k}}$$

with the midpoint of τ_k denoted by $\boldsymbol{c}^k$ and the normal vector given by the eigenvector corresponding to the largest eigenvalue of the covariance matrix

$$[\mathsf{C}(C_0)]_{i,j} := \sum_{\tau_k \in C_0} \Delta_{\tau_k} [\boldsymbol{c}^k - \boldsymbol{c}(C_0)]_i [\boldsymbol{c}^k - \boldsymbol{c}(C_0)]_j.$$

Proceeding recursively, the clusters C_{11} and C_{12} are further separated into C_{21}, C_{22}, and C_{23}, C_{24}, respectively, with the final level specified by the maximal number of elements in the leaf clusters. The product of the resulting binary tree of clusters with itself generates a quad-tree of submatrices $\mathsf{A}_h^{C_m \times C_n}$ of the original matrix A_h. The cluster pairs are separated into two groups by the admissibility condition

$$2 \min \big(\mathrm{r}(C_m), \mathrm{r}(C_n) \big) \leq \eta \big(\operatorname{dist}(\boldsymbol{c}(C_m), \boldsymbol{c}(C_n)) - \mathrm{r}(C_m) - \mathrm{r}(C_n) \big)$$

with the radius given by

$$\mathrm{r}(C) := \max_{\tau_k \in C} \max_{\boldsymbol{x} \in \tau_k} \|\boldsymbol{x} - \boldsymbol{c}(C)\|.$$

The blocks failing to satisfy this condition are called non-admissible and have to be assembled in full, the remaining admissible blocks $\mathsf{B} := \mathsf{A}_h^{C_m, C_n} \in \mathbb{R}^{d_m \times d_n}$ are approximated by the product of low-rank matrices

$$\mathsf{B} \approx \mathsf{U}_p \mathsf{V}_p^\top, \quad \mathsf{U}_p \in \mathbb{R}^{d_m \times p}, \quad \mathsf{V}_p \in \mathbb{R}^{d_n \times p}.$$

For an efficient approximation the clusters C_m, C_n should be as large as possible. Thus, after evaluating the admissibility criterion the leaf clusters can be joined together to ensure that the parents of admissible cluster pairs are non-admissible. The ACA method resembles the Lagrange interpolation of wisely chosen matrix entries. We denote $\mathsf{R}_0 := \mathsf{B}$ and start the algorithm by defining the first pivots i_1, j_1, where $i_1 := 1$ and j_1 is chosen such that $[\mathsf{R}_0]_{i_1,j_1}$ is the largest entry of the row $[\mathsf{R}_0]_{i_1,\bullet}$ in modulus. The first approximation is then given by the row and column

$$\mathsf{V}_1 := (1/[\mathsf{R}_0]_{i_1,j_1}) [\mathsf{R}_0]_{i_1,\bullet}^\top, \qquad \mathsf{U}_1 := [\mathsf{R}_0]_{\bullet,j_1}.$$

The algorithm continues iteratively by defining the new residual

$$\mathsf{R}_1 := \mathsf{R}_0 - \mathsf{U}_1 \mathsf{V}_1^\top$$

and choosing the next pivot row i_2 such that $[\mathsf{R}_1]_{i_2,j_1}$ is the largest entry of the column $[\mathsf{R}_1]_{\bullet,j_1}$ in modulus. The index j_2 is chosen from $[\mathsf{R}_1]_{i_2,\bullet}$ in the same manner and the approximating matrices are updated to

$$\mathsf{V}_2 := \begin{bmatrix} \mathsf{V}_1 & (1/[\mathsf{R}_1]_{i_2,j_2}) [\mathsf{R}_1]_{i_2,\bullet}^\top \end{bmatrix}, \qquad \mathsf{U}_2 := \begin{bmatrix} \mathsf{U}_1 & [\mathsf{R}_1]_{\bullet,j_2} \end{bmatrix}.$$

For the error measured in the Frobenius norm it can be shown, see [5], that $\|\mathsf{R}_p\|_{\mathrm{F}} \leq \varepsilon \|\mathsf{B}\|_{\mathrm{F}}$ holds provided that $\|\mathsf{R}_k\|_{\mathrm{F}} \leq \eta \|\mathsf{R}_{k-1}\|_{\mathrm{F}}$ and the stopping criterion

$$\left\|[\mathsf{U}]_{\bullet,p}[\mathsf{V}]_{\bullet,p}^{\top}\right\|_{\mathrm{F}} \leq \frac{1-\eta}{1+\varepsilon}\left\|\mathsf{U}_{p-1}\mathsf{V}_{p-1}^{\top}\right\|_{\mathrm{F}}$$

is satisfied. Although the criterion implies that the admissibility parameter should be chosen such that $\eta < 1$, the literature often suggest $\eta \leq 1.2$ leading to a better compression ratio.

3. Efficient implementation of regularized quadrature schemes

As the procedure for the assembly of V_h, K_h from (2.6) and (2.7), respectively, is almost identical, for brevity we concentrate on the computation of K_h only.

There are two possible alternatives for the assembly. Firstly, one can compute the matrix entry by entry which reduces to two successive loops over the basis of the ansatz and test function spaces. For the ansatz function φ_i and the test function ψ_ℓ this would lead to the evaluation of

$$\mathsf{K}_h[\ell, i] := \frac{1}{4\pi} \sum_{\tau_k \subset \operatorname{supp} \varphi_i} \int_{\tau_\ell} \int_{\tau_k} \frac{\langle \boldsymbol{x}-\boldsymbol{y}, \boldsymbol{n}(\boldsymbol{y})\rangle}{\|\boldsymbol{x}-\boldsymbol{y}\|^3} \varphi_i(\boldsymbol{y}) \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{y}} \,\mathrm{d}\boldsymbol{s}_{\boldsymbol{x}}. \tag{3.1}$$

This, however, means that the element τ_k has to be visited a couple of times, since it is included in the support of three globally continuous piecewise linear functions φ_i. This brings in an extra overhead caused by the recalculation of quadrature points and possibly other quantities relevant to the element. Moreover, the kernel k from (2.9), the computationally most demanding part, would have to be evaluated more than once for the same combination of quadrature points.

```
#pragma omp parallel for
for( int tau_l = 0; tau_l < E; ++tau_l ){
 for( int tau_k = 0; tau_k < E; ++tau_k ){
  getLocalMatrix( tau_l, tau_k, localMatrix );
  FullMatrix.add( tau_l, tau_k, localMatrix );
 }
}
```

Listing 3.1: Element-based assembly.

A better option is to adopt the element-based assembly widely used in finite element codes. Instead of two loops over the degrees of freedom, one loops over the pairs of elements irrespective of the current ansatz and test functions. In case of K_h this leads to the splitting of (3.1) into the individual summands. Thus, every pair of elements is only visited once and contributes to the global matrix with three entries corresponding to every φ_i supported on τ_k. The pseudocode of this approach is displayed in Listing 3.1. The function `getLocalMatrix`

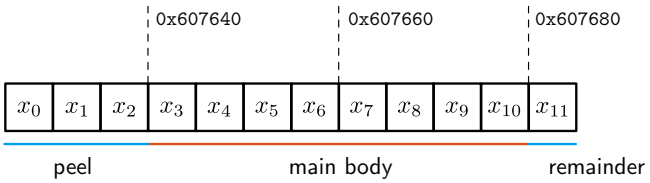


Figure 3.1: Unaligned memory access, loop peeling.

is responsible for the assembly of such local contributions, while `add` maps the element indices to the degrees of freedom, i.e., the rows and columns of the global matrix. The outer loop in this case corresponds exactly to the testing degrees of freedom and is thus a suitable candidate for a shared memory parallelization by OpenMP pragmas (line 1). It should be noted that there is no need for `atomic` operations inside the `add` method since the OpenMP threads distribute whole rows of the matrix. This is especially important for many-core architectures, where the `atomic` operations and `critical` sections could cause suboptimal speedup.

```
1 double * w = new double[ S ];
2 w[ 0 ] = ... // initialization of weights
3 double x [ ] = { ... }; // initialization of quadrature points
4
5 for( int l = 0; l < S; ++l ){
6  result += w[ l ] * f( x[ l ] );
7 }
8 delete [ ] w;
```

Listing 3.2: Simple 1D quadrature.

```
1 double * w = (double *) _mm_malloc( S * sizeof( double ), 64 );
2 w[ 0 ] = ... // fill in the weights and quadrature points
3 double x [ ]  __attribute__( ( aligned( 64 ) ) )  = { ... };
4
5 __assume_aligned( w, 64 ); // make sure compiler sees alignment
6 #pragma omp simd reduction( + : result ) // process the loop concurrently
7 for( int l = 0; l < S; ++l ){
8  result += w[ l ] * f( x[ l ] );
9 }
10 _mm_free( w );
```

Listing 3.3: Vectorized 1D quadrature.

To present the idea of SIMD vectorization for the quadrature scheme, let us first concentrate on a simple quadrature from Listing 3.2. The quadrature weights and nodes are stored in the (statically or dynamically allocated) arrays `w` and `x`, respectively, and the integral is collected in `result`. To vectorize the loop by compiler directives introduced in the OpenMP 4.0 standard [12], we only have to add the clause `#pragma omp simd reduction( + : result )` as shown in Listing 3.3. The `reduction` clause is a data sharing attribute

ensuring that the private copies of `result` corresponding to individual SIMD lanes are correctly summed up at the end of the loop. To load the corresponding vector data from the main memory into the vector registers effectively, the arrays should be aligned at 64-byte boundaries for all considered architectures. In case of static arrays the proper alignment is achieved by using the `__attribute__( ( aligned( 64 ) ) )` clause. To allocate aligned dynamic arrays, one can use the `_mm_malloc` function instead of the classical `malloc` or the `new` operator. The deallocation is achieved by using `_mm_free` instead of `free` or the `delete [ ]` operator. For dynamically allocated data one can use the `__assume_aligned` function to hint to the compiler that the data is properly aligned (especially if the arrays are allocated in another compilation unit).

In case the data is not properly aligned, the compiler divides the loop into more parts. In Figure 3.1 the array `x` is not aligned at the 64-byte boundary and thus the first three elements are processed separately in a peel loop. The following eight elements aligned at `0x607640` can be processed by two successive loops of AVX2 vector operations. The remaining element does not fill the whole vector register and falls into the remainder part, which is processed either by a scalar or a masked vector instruction. While both the peel and remainder loops can be vectorized by the compiler, the efficiency is lower and thus should be avoided if possible. Note that the correct alignment of `x` from Figure 3.1 would eliminate both the peel and remainder loops and `x` could be processed in three iterations of AVX2 instructions. If the vector length is not a multiple of the size of the vector register, the remainder loop can be eliminated by data padding.

Getting back to Listing 3.1, the method `getLocalMatrix` implements the numerical scheme presented in Section 2.3. A naive implementation for the case of identical elements $\tau_k = \tau_\ell$ is presented in Listing 3.4. The current quadrature nodes in the four-dimensional hypercube are stored in the variables `eta1`, `eta2`, `eta3`, `ksi`. The method `cubeToRefIdentical` accepts the current set of quadrature nodes and maps them by $\boldsymbol{F}^s$ from (2.11) to the corresponding nodes `nu`, `mu` in the reference element. The Jacobian $\boldsymbol{S}^s$ is stored in the variable `jac`. The reference nodes are then mapped to the actual quadrature nodes `x`, `y` in the elements τ_ℓ, τ_k given by the nodes `xl1`, ..., `yk3` by the method `refToTri` representing the mappings $\boldsymbol{R}^\ell$, $\boldsymbol{R}^k$ from (2.8). The method `evalDoubleLayerKernel` evaluates the kernel k from (2.9), which is then multiplied by the three ansatz functions φ_i supported on τ_k, the quadrature weights and the Jacobian, and added to `entry1`, `entry2`, `entry3`. These values are mapped to the global matrix by the `add` method from Listing 3.1.

The usual objective of the compiler is to vectorize the innermost loop. The vectorization is effective when the loop is long enough, which is unfortunately not the case of one-dimensional quadrature only using three or four quadrature points. However, if a three- or four-point rule is used for every dimension, the total number of sampling points is $3^4 = 81$ or $4^4 = 256$, respectively. Thus, instead of the four nested loops our aim is to create a single collapsed loop over all combinations of quadrature points.

First of all, the function `cubeToTriRefIdentical` does not have to be called for every pair τ_k, τ_ℓ separately, since the resulting reference nodes `nu`, `mu`

```
for( int l1 = 0; l1 < S1; ++l1 ){
 eta1 = quadNodesEta1[ l1 ]; // read first quadrature point
 for( int l2 = 0; l2 < S2; ++l2 ){
  eta2 = quadNodesEta2[ l2 ]; // read second quadrature point
  for( int l3 = 0; l3 < S3; ++l3 ){
   eta3 = quadNodesEta3[ l3 ]; // read third quadrature point
   for( int l4 = 0; l4 < S4; ++l4 ){
    ksi = quadNodesKsi[ l4 ]; // read fourth quadrature point

    switch( type ){ // decide which regularization should be used

     case identicalElements:
      for( int simplex = 0; simplex < S; ++simplex ){
       // map from hypercube to (tau x tau), get nu, mu, jac
       cubeToRefIdentical( eta1, eta2, eta3, ksi, simplex, nu, mu, jac );
       // map from (tau x tau) to (tau_l x tau_k), get x, y
       refToTri( x1, ..., y3, nu, mu, x, y );
       // multiply kernel with weights and jacobian
       kernel = jac * w1[ l1 ] * w2[ l2 ] * w3[ l3 ] * w4[ l4 ]
         * evalDoubleLayerKernel( x, y, n );
       // multiply by ansatz functions
       entry1 += kernel * phi1;
       entry2 += kernel * phi2;
       entry3 += kernel * phi3;
      }
      break;
     ... // quadrature over other pairs of elements
    } // end switch type
   } // end for ksi
  } // end for eta3
 } // end for eta2
} // end for eta1
```

Listing 3.4: Naive regularized 4D quadrature.

and the Jacobian `jac` are the same for every k, ℓ. The reference quadrature points can thus be precomputed for every combination of `type`, `eta1`, `eta2`, `eta3`, `ksi`, and `simplex`. This is shown in Listing 3.5, with the arrays `nu1Identical`, `nu2Identical`, `mu1Identical`, `mu2Identical` storing the reference nodes. Since this loop is only performed once before the matrix assembly, there is no need for any extensive optimization. However, notice that since we aim to manually collapse the four-dimensional loop from Listing 3.4, in Listing 3.5 we duplicate the quadrature weights so that the arrays `w1V`, `w2V`, `w3V`, `w4V` are all of the length $S_1S_2S_3S_4$. Also, there are two possibilities of storing the two-dimensional reference nodes `nu`, `mu`, see Figure 3.2. The array

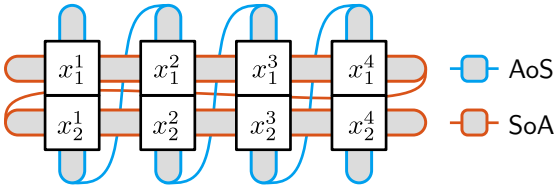


Figure 3.2: Array of structures vs. structure of arrays.

```
for( int l1 = 0; l1 < S1; ++l1 ){
 eta1 = quadNodesEta1[ l1 ]; // read first quadrature point
 for( int l2 = 0; l2 < S2; ++l2 ){
 eta2 = quadNodesEta2[ l2 ]; // read second quadrature point
  for( int l3 = 0; l3 < S3; ++l3 ){
  eta3 = quadNodesEta3[ l3 ]; // read third quadrature point
   for( int l4 = 0; l4 < S4; ++l4 ){
   ksi = quadNodesKsi[ l4 ]; // read fourth quadrature point
    // duplicate weights for loop collapsing
    w1V[ c ] = w1[ l1 ];
    ...
    w4V[ c ] = w4[ l4 ];

    for( int simplex = 0; simplex < S; ++simplex ){
     // map from hypercube to (tau x tau), get nu, mu, jac
     cubeToRefIdentical( eta1, eta2, eta3, ksi, simplex, nu, mu, jac );
     // store nodes componentwise in SoA
     nu1Identical[ simplex ][ c ] = nu[ 0 ];
     nu2Identical[ simplex ][ c ] = nu[ 1 ];
     mu1Identical[ simplex ][ c ] = mu[ 0 ];
     mu2Identical[ simplex ][ c ] = mu[ 1 ];
     jacIdentical[ simplex ][ c ] = jac;
    }
    ... // precompute nodes for other pairs of elements
    ++c;
   } // end for ksi
  } // end for eta3
 } // end for eta2
} // end for eta1
```

Listing 3.5: Computation of reference quadrature nodes.

of structures (AoS) first stores all components of one node before proceeding to another one, while the structure of arrays (SoA) stores the first coordinates of every node before storing the second coordinates. For the purposes of vectorization the SoA format is preferable here and four arrays (two coordinates for each triangle) for each simplex are created in the lines 18–21 in Listing 3.5.

The computation of the quadrature nodes lying in the current triangular elements τ_l, τ_k is implemented in the method `refToTri` called in the line 17 in Listing 3.4. Since this function is called for every pair of triangles, it requires an efficient implementation as shown in Listing 3.6. The four pointers `nu1`, ..., `mu2` point to the reference nodes in the SoA format precomputed in Listing 3.5 (i.e., for identical elements they point to `nu1Identical`, ..., `mu2Identical`), the nodes of the current triangles are stored in the three-dimensional vectors `xl1`, ..., `yk3`. The actual quadrature nodes lying in τ_ℓ, τ_k are written into the six arrays `x1`, ..., `y3`. The structure of the precomputed reference nodes allows us to manually collapse the four `for` loops into a single one. The vectorization of the mapping is achieved by the clause `#pragma omp simd`. A crucial ingredient for efficient vectorization is that the memory storing the reference and actual quadrature nodes is accessed in unit strides, which is accomplished by the SoA structure. The AoS structure would lead to less efficient two-strided loads of reference nodes and three-strided writes to the actual quadrature nodes.

The core of the quadrature is given by the method `evalDoubleLayerKernel` from the line 20 in Listing 3.4. The SoA structure of the quadrature nodes

```
// arrays were allocated by _mm_malloc
__assume_aligned( x1, 64 );
...
__assume_aligned( y3, 64 );
__assume_aligned( nu1, 64 );
...
__assume_aligned( mu2, 64 );

#pragma omp simd
for( int c = 0; c < S1*S2*S3*S4; ++c ){
 x1[ c ] = xl1[ 0 ]
  + ( xl2[ 0 ] - xl1[ 0 ] ) * nu1[ simplex ][ c ]
  + ( xl3[ 0 ] - xl2[ 0 ] ) * nu2[ simplex ][ c ];
  ... // compute x2, x3, y1, y2
 y3[ c ] = yk1[ 2 ]
  + ( yk2[ 2 ] - yk1[ 2 ] ) * mu1[ simplex ][ c ]
  + ( yk3[ 2 ] - yk2[ 2 ] ) * mu2[ simplex ][ c ];
}
```

Listing 3.6: Computation of actual quadrature nodes.

```
#pragma omp declare simd uniform( n )
double evalDoubleLayerKernel(
 double x1, double x2, double x3,
 double y1, double y2, double y3,
 const double * n
) const {

 double d1 = x1 - y1, d2 = x2 - y2, d3 = x3 - y3; // x-y
 double norm = std::sqrt( d1 * d1 + d2 * d2 + d3 * d3 ); // |x-y|
 double dot = d1 * n[ 0 ] + d2 * n[ 1 ] + d3 * n[ 2 ]; // (x-y,n)

 return ( dot / ( norm * norm * norm * 4.0 * M_PI ) );
}
```

Listing 3.7: Evaluation of the double-layer kernel.

arrays requires an adapted signature of the method as shown in Listing 3.7. The normal vector `n` is constant in the inner triangle and can be passed as a three-dimensional array. Since the method is called in a vectorized loop, it should either be inlined in the code (defined in a header file), or the clause `#pragma omp declare simd` should be used, stating that the function can be called with vector arguments. The `uniform( n )` clause tells the compiler that the array `n` is constant in the whole vectorized call.

The final optimized code of the numerical quadrature is presented in Listing 3.8. All relevant arrays are properly aligned, which eliminates the need for loop peeling. For every simplex the updated method `refToTri` returns the actual quadrature nodes computed by the code from Listing 3.6 in the SoA format. The `simd` pragma ensures that the collapsed loop will be vectorized and the kernel thus evaluated efficiently. Due to the definition of the piecewise linear ansatz functions (2.5) the values `phi1`, `phi2`, `phi3` can either be precomputed in the same way as the reference quadrature nodes in Listing 3.5, or evaluated inside

```
__assume_aligned( x1, 64 );
...
__assume_aligned( y3, 64 );
__assume_aligned( w1V, 64 );
...
__assume_aligned( w4V, 64 );
__assume_aligned( jacV, 64 );

switch( type ){
 case( identicalElements ):
  for( int simplex = 0; simplex < S; ++simplex ){

  // map from (tau x tau) to (tau_l x tau_k), get x, y in SoA format
  refToTri( simplex, xl1, ..., yk3, nu1, ..., mu2, x1, ..., y3 );

#pragma omp simd
reduction( + : entry1, entry2, entry3 )
   for ( int c = 0; c < S1*S2*S3*S4; ++c ) { // collapsed loop
    // multiply kernel with weights and jacobian using unit-stride access
    kernel = w1V[ c ] * w2V[ c ] * w3V[ c ] * w4V[ c ] * jacV[ c ] *
     evalDoubleLayerKernel( x1[ c ], x2[ c ],  x3[ c ],
     y1[ c ], y2[ c ], y3[ c ], n );
    // multiply by ansatz functions
    entry1 += kernel * phi1;
    entry2 += kernel * phi2;
    entry3 += kernel * phi3;
   } // end for c
  } // end for simplex
  break;
 ... // quadrature over other pairs of elements
} // end switch type
```

Listing 3.8: Vectorized regularized 4D quadrature.

the numerical quadrature routine using the reference nodes `mu1`, `mu2` causing no significant overhead.

```
#pragma omp parallel for
for( int i = 0; i < n_nonadmissible_blocks; ++i ){
  getNonadmissibleBlock( i, localBlock );
  ACAMatrix.addNonadmissibleBlock( i, localBlock );
}
#pragma omp parallel for
for( int i = 0; i < n_admissible_blocks; ++i ){
  getAdmissibleBlock( i, localBlock );
  ACAMatrix.addAdmissibleBlock( i, localBlock );
}
```

Listing 3.9: ACA assembly.

For completeness, let us comment on the modification of the presented code snippets for the ACA assembly. Firstly, the OpenMP parallelization is performed on the level of individual (non-)admissible matrix blocks with each one assembled by a single thread as shown in Listing 3.9. In the method `getNonadmissibleBlock` the block is constructed in the same manner as in the full matrix assembly following the elementwise construction from Listing 3.1 without the additional OpenMP region. The method `getAdmissibleBlock` im-

plements the approximation described in Section 2.4. Since the ACA algorithm is degree-of-freedom-based, i.e., it directly chooses the rows and columns of the block to be assembled, one first has to translate the degrees of freedom to the supporting elements and then call the standard routine from Listing 3.8. This leads to some entries of the local matrices to be thrown away. However, the computation of these values brings no significant additional cost since this only requires the kernel (which has to be evaluated anyway and dominates the quadrature) to be multiplied by the unused ansatz and test functions.

The matrix-vector multiplication necessary for the iterative solution of the BEM system can be handled efficiently by using an optimized BLAS routine as implemented, e.g., in the Intel MKL library. In the case of ACA one can distribute both admissible and non-admissible blocks by OpenMP and perform the multiplication by submatrices concurrently (note that also the admissible blocks are approximated by a pair of full rectangular matrices).

For larger problems a suitable preconditioner for V_h may be necessary. Several possibilities represented, e.g., by an artificial multilevel preconditioner based on the binary tree already built for the ACA method are presented in [30, 31] and references therein.

4. Numerical experiments

The numerical experiments presented in this section were performed at two different HPC infrastructures. The multi-core Haswell (HSW) and many-core Knights Corner (KNC) architectures were provided by the IT4Innovations National Supercomputing Center featuring the Salomon cluster with 576 nodes equipped with two 12-core Intel Xeon E5-2680v3 (Haswell) processors supporting the AVX2 instruction set and 128 GB of RAM and 432 accelerated nodes additionally equipped with two Intel Xeon Phi 7120P (Knights Corner) coprocessors. Each coprocessor offers 61 cores running at 1.238 GHz, four hardware threads per core and 512-bit SIMD registers supporting the IMCI instruction set. The memory of each KNC card is limited to 16 GB. The theoretical computational power of one coprocessor (1.2 TFLOPS) is approximately 1.26 times higher than the power provided by the pair of the Intel Xeon processors. The comparison is based on the execution of 16 Fused Multiply-Add instructions per cycle on both architectures. The Intel Parallel Studio XE 2016 environment was used for compilation with the flags `-xHost -mkl -qopenmp`. The `-xHost` flag automatically translates to the highest SIMD optimization available, i.e., `-xCORE-AVX2`, `-mmic`, and `-xMIC-AVX512` for the HSW, KNC, and KNL architectures, respectively.

To demonstrate the need for vectorization of scientific codes, further experiments were performed on Intel's Endeavour cluster equipped with a preproduction version of the Intel Xeon Phi 7210 (codenamed Knights Landing, KNL). The new generation of the Intel Xeon Phi devices will be available both in the form of a coprocessor and a standalone self-booting processing unit. The run on the standalone version is similar to running the program natively on the Knights Corner coprocessor, except for the fact that the processor has a direct

threads	2	4	8	16	24
double	2.01	4.04	8.07	16.09	24.07
single	1.97	3.95	7.85	15.71	23.01

Table 4.1: Speedup of OpenMP parallelized full assembly of V_h (2×HSW, AVX2).

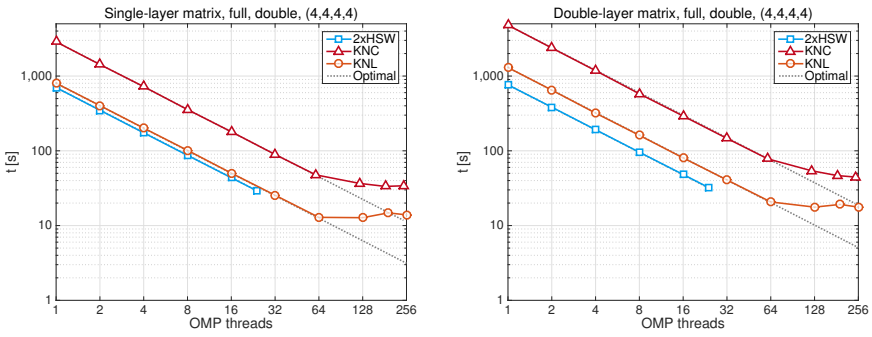
threads	2	4	8	16	24
double	1.99	3.98	7.97	15.83	23.68
single	1.97	3.96	8.02	15.96	23.53

Table 4.2: Speedup of OpenMP parallelized full assembly of K_h (2×HSW, AVX2).

access to up to 384 GB of DDR4 memory. In addition, the processors feature 16 GB of a fast on-package MCDRAM memory, which can be utilized in several modes. The Knights Landing (co)processors have up to 72 cores based on the more modern Airmont architecture. One of the advantages over the Knights Corner technology is the out-of-order handling of instructions and two vector processing units per core, which can decrease the pressure on vector registers and pushes the theoretical performance to 2.66 TFLOPS. The IMCI instruction set is replaced by the AVX-512 set designed for concurrent operations on 8 (16) double (single) precision operands. Moreover, the support of the legacy SSE4.2 and AVX2 instruction sets makes this architecture binary compatible with the server Xeon processors. In this section we provide numerical experiments performed at a single node of the Endeavour cluster equipped with the pre-production self-booting 64-core KNL processor and 96 GB of DDR4 memory.

4.1. Full assembly

For measuring the assembly times of the full matrices V_h, K_h from (2.6), (2.7) using the implementation described above we use a triangular mesh of a five-times refined icosahedron consisting of 20,480 elements and 10,242 nodes. The assembly is repeated five times, the resulting time is an arithmetic average of the last four runs. To fully utilize the processors' vector processing units and the potential of the SIMD instructions we use the tensor Gaussian quadrature with $4^4 = 256$ quadrature nodes per simplex. Such precision is usually necessary to reach the theoretical order of convergence in case of complicated geometries. We compare the scalability performance on Haswell, Knights Corner, and Knights Landing architectures with respect to both the number of OpenMP threads used for distributing the local element contributions and the SIMD register size playing a significant role in the regularized four-dimensional quadrature. Although the boundary element method usually requires higher precision of floating-point operations, as a proof of concept we also provide results for single precision arithmetic virtually doubling the size of the SIMD registers.



(a) V_h (double). (b) K_h (double).

Figure 4.1: OpenMP parallelized full assembly of the BEM matrices, double precision.

threads	4	8	16	32	61	122	183	244
double	3.98	8.06	15.96	32.14	60.23	78.33	85.90	84.29
single	4.01	8.16	16.38	31.97	60.58	75.49	74.19	70.46

Table 4.3: Speedup of OpenMP parallelized full assembly of V_h (KNC, IMCI).

threads	4	8	16	32	61	122	183	244
double	4.06	8.28	16.42	32.72	61.38	88.54	102.71	107.53
single	4.04	8.23	16.32	32.43	61.40	79.60	84.57	84.50

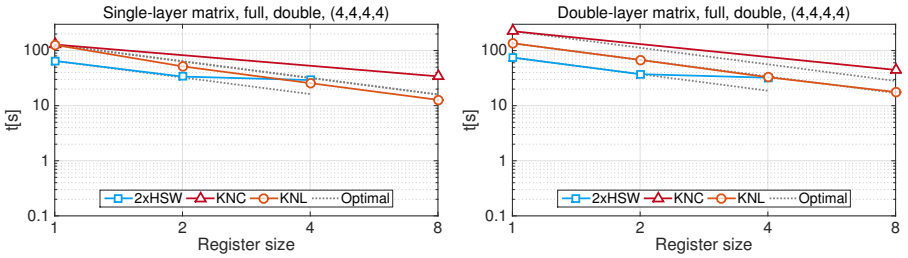
Table 4.4: Speedup of OpenMP parallelized full assembly of K_h (KNC, IMCI).

threads	4	8	16	32	64	128	192	256
double	3.99	7.90	15.93	31.48	62.26	62.89	54.22	57.65
single	4.01	8.02	15.98	31.80	62.74	68.88	54.10	55.06

Table 4.5: Speedup of OpenMP parallelized full assembly of V_h (KNL, AVX-512).

threads	4	8	16	32	64	128	192	256
double	4.07	8.07	16.20	31.89	62.76	73.64	67.76	73.64
single	4.03	8.05	16.01	31.13	61.16	73.40	64.21	64.86

Table 4.6: Speedup of OpenMP parallelized full assembly of K_h (KNL, AVX-512).



(a) V_h (double). (b) K_h (double).

Figure 4.2: Assembly times of full matrices with respect to the width of the SIMD registers employed (1 for scalar instructions, 2 for SSE4.2, 4 for AVX2, and 8 for IMCI/AVX-512).

architecture	threads	precision	SSE4.2	AVX2	IMCI	AVX-512
2xHSW (Xeon E2680v3)	24	double	1.92	2.23	—	—
		single	3.59	6.35	—	—
KNC (Xeon Phi 7120P)	244	double	—	—	3.77	—
		single	—	—	5.67	—
KNL (Xeon Phi 7210)	128	double	2.45	4.95	—	9.94
		single	3.95	7.23	—	13.05

Table 4.7: Speedup of OpenMP vectorized full assembly of V_h.

architecture	threads	precision	SSE4.2	AVX2	IMCI	AVX-512
2xHSW (Xeon E2680v3)	24	double	2.01	2.32	—	—
		single	2.65	4.81	—	—
KNC (Xeon Phi 7120P)	244	double	—	—	5.05	—
		single	—	—	6.13	—
KNL (Xeon Phi 7210)	128	double	2.01	4.14	—	7.69
		single	3.15	6.62	—	11.68

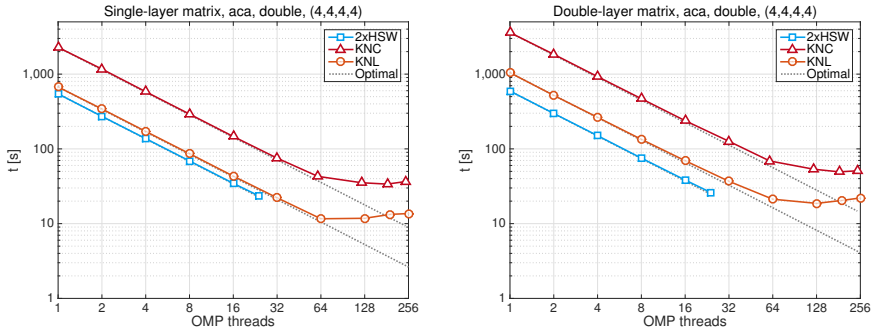
Table 4.8: Speedup of OpenMP vectorized full assembly of K_h.

Firstly, let us concentrate on the scalability with respect to the number of OpenMP threads. The presented results achieved at Knights Landing are compared to those achieved by up to 244 vectorized OpenMP threads on one Knights Corner coprocessor and 24 threads on the Haswell computational node of the Salomon cluster. The reference timings for the vectorized single-threaded assembly of V_h and K_h on KNL in double (single) precision read 801.90 s (436.04 s) and 1,299.73 s (707.60 s), respectively. The computational times reduce to 12.88 s (6.95 s) and 20.71 s (11.57 s) on 64 threads running on all physical cores, which corresponds to the almost optimal speedup of 62.26 (62.74) and 62.76 (61.16), respectively. Best results were achieved on 128 threads, where the speedup reached 62.89 (68.88) and 73.64 (73.40), respectively. The speedup with respect to the best assembly times on Haswell (24 vectorized threads) reaches 2.29 (1.19) and 1.82 (1.26), respectively. The comparison with Knights Corner leads to the speedup of 2.62 (2.99) and 2.53 (2.59) which is reasonable taking into account the theoretical maximum performance of both devices.

In Tables 4.1–4.6 we summarize the speedup achieved by various number of OpenMP threads on all available architectures related to a single-threaded run. It can be seen that although the hyperthreading (using more than 64 threads) on Knights Landing may lead to better computational times, its importance is not as significant as in the case of the former Knights Corner coprocessor. This is mainly due to the more modern core architecture able to handle instructions in an out-of-order manner. In Figure 4.1 we summarize these results graphically for double precision arithmetic.

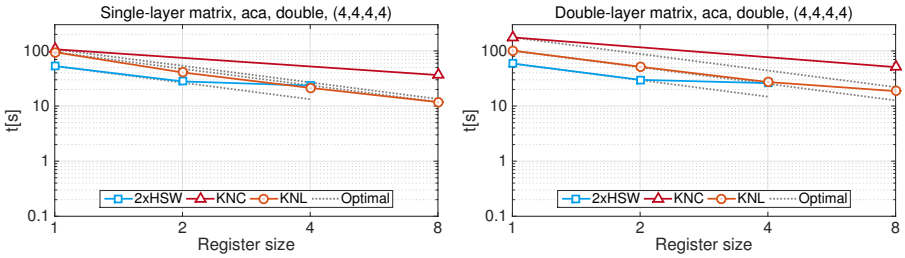
To demonstrate the necessity of proper vectorization we also present numerical experiments with different vector instruction sets allowed. The reference times are derived from the assembly of V_h and K_h on the Knights Landing in double (single) precision on 128 OpenMP threads with no vectorization forced by the `-no-vec -no-simd -qno-opemp-simd` compiler flag instead of `-xHost` and read 126.77 s (82.63 s) and 135.76 s (112.62 s), respectively. With AVX-512 instructions employed by the `-xMIC-AVX512` compiler switch, the times drop to 12.75 s (6.33 s) and 17.65 s (9.64 s) representing the speedup of 9.94 (13.05) and 7.69 (11.68), respectively. Although the optimal speedup gained by the AVX-512 instructions would be 8 (16) for double (single) precision (not taking into account the slightly lower CPU clock frequency for SIMD instructions), the Knights Landing architecture still seems to be more efficient in handling the vector operations than both Knights Corner and Haswell. On Haswell with AVX2 instructions able to concurrently operate on 4 (8) operands, the maximum speedup reached 2.23 (6.35) and 2.32 (4.81), respectively. The IMCI instruction set available for the Knights Corner yielded the speedup of 3.77 (5.67) and 5.05 (6.13), respectively.

In Tables 4.7 and 4.8 we summarize the speedup obtained with various vector instruction sets enabled with respect to the scalar code running on 128, 244, and 24 OpenMP threads on Knights Landing, Knights Corner, and Haswell, respectively. The results prove that the modern hardware architectures can only be fully exploited by codes allowing for proper vectorization. The results for double precision arithmetic are graphically presented in Figure 4.2. The hori-



(a) V_h (double). (b) K_h (double).

Figure 4.3: OpenMP parallelized ACA assembly of the BEM matrices, double precision.



(a) V_h (double). (b) K_h (double).

Figure 4.4: Assembly times of ACA matrices with respect to the width of the SIMD registers employed (1 for scalar instructions, 2 for SSE4.2, 4 for AVX2, and 8 for IMCI/AVX-512).

zontal axis represents various SIMD instruction set extensions employed (none, SSE4.2, AVX2, IMCI/AVX-512) and thus various number of double precision operands that fit into the SIMD registers. It can be clearly seen that the addition of extra vector processing units per core on Knights Landing results in an almost optimal SIMD scalability, while the results on Knights Corner and the difference on Haswell between SSE4.2 and AVX2 are suboptimal, which may be caused by high pressure on the vector registers.

4.2. ACA assembly

Since the ACA sparsification method allows for larger meshes, for the experiments we use the icosahedron geometry subdivided six times to obtain a

threads	2	4	8	16	24
double	2.00	3.97	7.89	15.74	23.13

Table 4.9: Speedup of OpenMP parallelized ACA assembly of V_h (2×HSW, AVX2).

threads	2	4	8	16	24
double	1.99	3.95	7.89	15.49	22.75

Table 4.10: Speedup of OpenMP parallelized ACA assembly of K_h (2×HSW, AVX2).

threads	4	8	16	32	61	122	183	244
double	3.92	7.80	15.57	30.32	53.16	64.69	67.44	62.39

Table 4.11: Speedup of OpenMP parallelized ACA assembly of V_h (KNC, IMCI).

threads	4	8	16	32	61	122	183	244
double	3.88	7.67	15.10	28.49	52.17	67.20	72.94	70.66

Table 4.12: Speedup of OpenMP parallelized ACA assembly of K_h (KNC, IMCI).

threads	4	8	16	32	64	128	192	256
double	3.93	7.80	15.57	30.36	57.92	57.08	50.57	49.34

Table 4.13: Speedup of OpenMP parallelized ACA assembly of V_h (KNL, AVX-512).

threads	4	8	16	32	64	128	192	256
double	3.95	7.78	15.03	28.34	49.17	55.92	51.02	47.30

Table 4.14: Speedup of OpenMP parallelized ACA assembly of K_h (KNL, AVX-512).

architecture	threads	precision	SSE4.2	AVX2	IMCI	AVX-512
2xHSW (Xeon E2680v3)	24	double	1.90	2.26	—	—
KNC (Xeon Phi 7120P)	244	double	—	—	2.94	—
KNL (Xeon Phi 7210)	128	double	2.33	4.45	—	8.06

Table 4.15: Speedup of OpenMP vectorized ACA assembly of V_h.

architecture	threads	precision	SSE4.2	AVX2	IMCI	AVX-512
2xHSW (Xeon E2680v3)	24	double	1.98	2.27	—	—
KNC (Xeon Phi 7120P)	244	double	—	—	3.47	—
KNL (Xeon Phi 7210)	128	double	1.96	3.71	—	5.43

Table 4.16: Speedup of OpenMP vectorized ACA assembly of K_h.

threads	2	4	8	16	24
double	1.67	3.02	5.03	7.08	8.25

Table 4.17: Speedup of OpenMP parallelized ACA application of V_h (2×HSW).

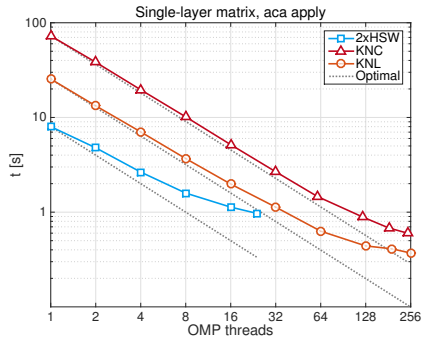


Figure 4.5: Application of V_h (double).

threads	4	8	16	32	61	122	183	244
double	3.71	7.16	14.08	27.20	49.75	81.61	106.81	121.05

Table 4.18: Speedup of OpenMP parallelized ACA application of V_h (KNC).

mesh with 81,920 elements and 40,962 nodes. Again, the assembly times are measured five times in a row and with the last four averaged to give the resulting value. For every dimension in the tensor Gaussian quadrature we use four nodes resulting in SIMD vectors of the length 256. The element clusters are recursively bisected until they hold no more than 50 elements, after checking the admissibility condition with $\eta := 1.2$ they can be joined together up to 500 elements to ensure good load balancing. The stopping criterion constant is chosen as $\varepsilon := 10^{-4}$. Differently from the full assembly experiments we only provide results in double precision arithmetic since the ACA method can prove sensitive to inaccuracies caused by lower precision number comparisons.

Firstly, we compare the scalability properties of the code with respect to the number of OpenMP threads employed. As shown in Section 2.4, individual (non-)admissible blocks are distributed among available threads and each block is assembled in the element-based manner without further threading. The quadrature is accelerated by the SIMD approach as in the full assembly.

The reference times on the Knights Landing obtained by a single-threaded vectorized run read 673.03 s and 1,043.97 s for V_h and K_h, respectively. Running on all physical cores reduces the times to 11.62 s and 21.23 s, which corresponds to the speedup of 57.92 and 49.17, respectively. The best time for K_h was achieved on 128 threads, i.e., with two threads running on each physical core and

threads	4	8	16	32	64	128	192	256
double	3.62	6.92	12.86	22.53	40.41	57.86	62.10	68.81

Table 4.19: Speedup of OpenMP parallelized ACA application of V_h (KNL).

reads 18.67 s further increasing the speedup to 55.92. The ratio of the best time achieved on Haswell to the best Knights Landing performance reaches 2.02 and 1.40, respectively. The speedup with respect to the Knights Corner results reads 2.91 and 2.64, respectively. The results for all architectures are summarized in Tables 4.9–4.14, the graphical comparison with the optimal speedup is depicted in Figure 4.3.

To compare the SIMD capabilities we again compare the times achieved with the various available optimization options enabled to the times reached with the `-no-vec -no-simd -qno-openmp-simd` flag. The number of OpenMP threads is always fixed to 24, 244, and 128 for Haswell, Knights Corner, and Knights Landing, respectively.

The reference sequential times on Knights Landing read 95.00 s and 101.33 s for V_h and K_h, respectively. Enabling the AVX-512 instructions lowers the result to 11.79 s and 18.67 s, which corresponds to the speedup of 8.06 and 5.43, respectively. In case of V_h this is again slightly better than the expected optimum, the relatively lower performance of the K_h assembly might correspond to the additional write operations in the local element contributions and a more complicated mapping between the degrees of freedom and element indices. The highest speedup attained on Haswell with the AVX2 instructions reached 2.26 and 2.27, respectively. For Knights Corner with IMCI enabled the ratio reads 2.94 and 3.47, respectively. Again, the performance on the newest Knights Landing architecture surpasses the other two as expected. In Tables 4.15 and 4.16 we summarize the findings described above. The results can also be seen in Figure 4.4 together with the desired speedup.

An inseparable part of the ACA approximation is the matrix-vector product performed by an iterative solver. Due to suitable preconditioners [30, 31] the number of V_h applications necessary to solve the boundary element system can be reduced significantly. Here we present the scalability results for 20 successive applications with respect to the number of OpenMP threads dynamically distributing both admissible and non-admissible blocks and performing the multiplication of the global vector restricted to the corresponding degrees of freedom. For the multiplication by the full blocks and the approximating full rectangular matrices we use the `dgemv` BLAS method implemented in Intel's MKL library. Since the blocks are relatively small, we do not use nested threading, i.e., each block is processed sequentially. Moreover, no further vectorization is performed outside of the MKL library. Contrary to the assembly results the MCDRAM serving as the last-level cache resulted in better scaling and thus we used it in this mode here.

The single-threaded run on Knights Landing resulted in the time 25.46 s and was reduced to 2.03 s on 64 threads running on all cores. The best time of 0.60 s was achieved by using four threads per core, i.e. 256 threads in total, which translates to the reasonable speedup of 68.81. Similarly, on Knights Corner the speedup on 244 logical threads results in the speedup of 121.05. The ratio of the fastest times observed on KNL and KNC reads 1.62, the speedup of the KNL with respect to the best time achieved on Haswell reads 2.62. The results are again summarized in Tables 4.17–4.19 and graphically in Figure 4.5.

5. Conclusion

The aim of the paper was to present implementation techniques for the boundary element method in 3D utilizing modern multi- and many- core architectures. The SIMD vectorization for the four-dimensional regularized quadrature proved efficient for both full and ACA assembly with the best results achieved on the newest many-core Knights Landing platform. Due to the physical limitations of the manufacture of modern processors we believe that such architectures present the future of HPC together with the GP-GPU programming on graphical units. The great advantage of Intel's Many Integrated Core architecture is the straightforward portability of the standard multi-core CPU implementation. Moreover, additional optimization for many-core usually brings further speedup on the original architecture as well.

The presented method is easily applicable to first-order BEM for other problems including linear elasticity, electromagnetism, or wave scattering. Some additional work has to be done for the Helmholtz equation, where one deals with complex-valued matrices. In addition to the structure-of-arrays format of the coordinates, similar approach should be used for the real and imaginary parts of the related quantities to ensure unit-strided access to the data.

The area of future research includes higher-order elements and the adaptation of the method for *hp*-BEM or the isogeometric approach. We should remark here that globally continuous higher-order polynomial ansatz functions require atomic operations when mapping the local contributions to the global matrix. This presents a serious issue for many-core processing. One possible remedy would be to distribute the elements in such a way that the number of concurrent write operations is minimized. Moreover, the *hp* approach brings in the difficulty of different quadrature schemes for individual elements, and thus not all quadrature data can be computed beforehand. Also, to utilize all available hardware resources in HPC environments, the native run on accelerators has to be substituted by the offload mode, where also the host processor is assigned a certain portion of the computation [11].

Acknowledgements

The authors acknowledge the support provided by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project "IT4Innovations excellence in science - LQ1602" and the grant SP2016/113 provided by VŠB – Technical University of Ostrava. The first two authors were additionally supported by the Large Infrastructures for Research, Experimental Development and Innovations project "IT4Innovations National Supercomputing Center – LM2015070".

References

[1] V. Rokhlin, Rapid solution of integral equations of classical potential theory, Journal of Computational Physics 60 (2) (1985) 187–207. doi:10.1016/0021-9991(85)90002-6.

[2] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, Journal of Computational Physics 73 (2) (1987) 325–348. doi:10.1016/0021-9991(87)90140-9.

[3] G. Of, Fast multipole methods and applications, in: M. Schanz, O. Steinbach (Eds.), Boundary Element Analysis, Vol. 29 of Lecture Notes in Applied and Computational Mechanics, Springer Berlin Heidelberg, 2007, pp. 135–160. doi:10.1007/978-3-540-47533-0_6.

[4] S. Rjasanow, O. Steinbach, The Fast Solution of Boundary Integral Equations, Mathematical and Analytical Techniques with Applications to Engineering, Springer, 2007.

[5] M. Bebendorf, Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems, Lecture Notes in Computational Science and Engineering, Springer, 2008.

[6] G. Of, O. Steinbach, The all-floating boundary element tearing and interconnecting method, Journal of Numerical Mathematics 17 (4) (2009) 277–298.

[7] U. Langer, O. Steinbach, Boundary element tearing and interconnecting methods, Computing 71 (3) (2003) 205–228.

[8] C. Pechstein, Finite and Boundary Element Tearing and Interconnecting Solvers for Multiscale Problems, Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2012.

[9] M. Merta, J. Zapletal, Acceleration of boundary element method by explicit vectorization, Advances in Engineering Software 86 (2015) 70–79. doi:10.1016/j.advengsoft.2015.04.008.

[10] M. Kretz, V. Lindenstruth, Vc: A C++ library for explicit vectorization, Software: Practice and Experience 42 (11) (2012) 1409–1430. doi:10.1002/spe.1149.
URL https://github.com/VcDevel/Vc

[11] M. Merta, J. Zapletal, J. Jaros, Many core acceleration of the boundary element method, in: T. Kozubek, R. Blaheta, J. Šístek, M. Rozložník, M. Čermák (Eds.), 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, 2016, pp. 116–125. doi:10.1007/978-3-319-40361-8_8.

[12] OpenMP Architecture Review Board, OpenMP Application Program Interface (7 2013).
URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[13] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, 1st Edition, Morgan Kaufmann, 2013.

[14] J. Jeffers, J. Reinders, High Performance Parallelism Pearls Volume One: Multicore and Many-core Programming Approaches, Elsevier Science, 2014.

[15] J. Jeffers, J. Reinders, High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, Elsevier Science, 2015.

[16] J. Jeffers, J. Reinders, A. Sodani, Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition, Elsevier Science, 2016.

[17] M. T. F. Cunha, J. C. F. Telles, F. L. B. Ribeiro, Streaming SIMD extensions applied to boundary element codes, Advances in Engineering Software 39 (11) (2008) 888–898.

[18] U. Iemma, On the use of a SIMD vector extension for the fast evaluation of boundary element method coefficients, Advances in Engineering Software 41 (3) (2010) 451–463.

[19] M. López-Portugués, J. A. López-Fernández, N. Díaz-Gracia, R. Ayestarán, J. Ranilla, Aircraft noise scattering prediction using different accelerator architectures, The Journal of Supercomputing 70 (2) (2014) 612–622. doi:10.1007/s11227-014-1107-z.

[20] K. Banaś, F. Krużel, J. Bielański, Finite element numerical integration for first order approximations on multi- and many-core architectures, Computer Methods in Applied Mechanics and Engineering 305 (2016) 827–848. doi:10.1016/j.cma.2016.03.038.

[21] S. Erichsen, S. A. Sauter, Efficient automatic quadrature in 3-d Galerkin BEM, Computer Methods in Applied Mechanics and Engineering 157 (3–4) (1998) 215–224. doi:10.1016/S0045-7825(97)00236-3.

[22] S. Sauter, C. Schwab, Boundary Element Methods, Springer Series in Computational Mathematics, Springer, 2010.

[23] J. Zapletal, J. Bouchala, Effective semi-analytic integration for hypersingular Galerkin boundary integral equations for the Helmholtz equation in 3D, Applications of Mathematics 59 (5) (2014) 527–542. doi:10.1007/s10492-014-0070-6.

[24] J. Zapletal, The boundary element method for the Helmholtz equation in 3D, Master's thesis, VŠB-TU Ostrava (2011). doi:10.13140/RG.2.1.2621.5128.

[25] M. Merta, J. Zapletal, BEM4I, IT4Innovations (2013).
URL http://bem4i.it4i.cz

[26] K. Bandara, F. Cirak, G. Of, O. Steinbach, J. Zapletal, Boundary element based multiresolution shape optimisation in electrostatics, Journal of Computational Physics 297 (2015) 584–598. doi:10.1016/j.jcp.2015.05.017.

[27] O. Steinbach, Numerical Approximation Methods for Elliptic Boundary Value Problems: Finite and Boundary Elements, Texts in applied mathematics, Springer, 2008.

[28] F. Hildebrand, Introduction to Numerical Analysis: Second Edition, Dover Books on Mathematics, Dover Publications, 2013.

[29] D. Lukáš, P. Kovář, T. Kovářová, M. Merta, A parallel fast boundary element method using cyclic graph decompositions, Numerical Algorithms 70 (4) (2015) 807–824.

[30] O. Steinbach, Artificial multilevel boundary element preconditioners, PAMM 3 (1) (2003) 539–542. doi:10.1002/pamm.200310539.

[31] G. Of, An efficient algebraic multigrid preconditioner for a fast multipole boundary element method, Computing 82 (2) (2008) 139–155. doi:10.1007/s00607-008-0002-y.