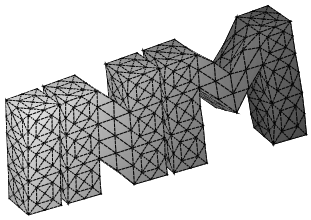
Technische Universität Graz

Parallel and vectorized implementation of analytic evaluation of boundary integral operators

J. Zapletal, G. Of, M. Merta



Berichte aus dem Institut für Numerische Mathematik

Bericht 2017/3

Technische Universität Graz

Parallel and vectorized implementation of analytic evaluation of boundary integral operators

J. Zapletal, G. Of, M. Merta

Berichte aus dem Institut für Numerische Mathematik

Bericht 2017/3

Technische Universität Graz
Institut für Numerische Mathematik
Steyrergasse 30
A 8010 Graz

WWW: `http://www.numerik.math.tu-graz.at`

Parallel and vectorized implementation of analytic evaluation of boundary integral operators

Jan Zapletal[1,2], Günther Of[3], Michal Merta[1,2]

[1] IT4Innovations, VŠB – Technical University of Ostrava, 17. listopadu 2172/15, 708 33 Ostrava-Poruba, Czech Republic

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

[3] Institute of Computational Mathematics, Graz University of Technology, Steyrergasse 30, A-8010 Graz, Austria

In the paper we describe an efficient analytic evaluation of boundary integral operators. Firstly, we concentrate on a novel approach based on a simultaneous evaluation for all three linear shape functions defined on a boundary triangle. This results in the speedup of 2.35–3.15 compared to the old approach of separate evaluations. In the second part we comment on the OpenMP parallelized and vectorized implementation of the suggested formulae. The employed code optimizations include techniques such as data alignment and padding, array-of-structures to structure-of-arrays data transformation, or unit-strided memory accesses. The presented scalability results with respect both to the number of threads employed and the width of the SIMD register performed on an Intel® Xeon™ processor and two generations of the Intel® Xeon Phi™ family (co)processors validate the performed optimizations and show that vectorization is an inherent part of modern scientific codes.

Keywords: boundary element method, analytic integration, SIMD, vectorization, Intel Xeon Phi, many-core

1 Introduction

The computation of matrix entries and the evaluation of the representation formula are of major importance in boundary element methods (BEM). On one hand, the often singular integrals have to be computed with sufficient accuracy to preserve important matrix properties and the optimal order of convergence. On the other hand, the computation has to be fast as this is a major part of the total computational time even for fast boundary element methods. A popular approach is to use explicit analytical formulae for the evaluation of boundary integral operators. The related formulae have been topic of research for decades, recent publications discussing the topic include, e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9]. In most cases, the formulae are provided for plane triangles, the kernel $|x-y|^{-1}$, and its derivatives. As the formulae are exact, they are obviously related. However, the knowledge of a formula is just part of the story as certain geometric settings lead to

special cases in its evaluation which have to be handled with extra care in the implementation. Thus, a pure comparison of the formulae is not sufficient to rate the quality of the approaches.

In this paper, we use a carefully developed and excessively tested implementation based on the formulae in [4, 9]. We focus on the evaluation of single- and double-layer potentials of the 3D Laplace kernel and linear shape functions. The formulae in [4, 9] suggest to choose a local coordinate system in the plane triangle related to the considered linear shape function. Then, three independent computations are required to compute the integrals for the three linear shape functions of a single triangle. In this paper, we compute these three integrals at once which reduces the computational effort to almost one third. To do so, we present additional analytic formulae which are related to the setting chosen in [4]. The formulae (2.13) and (2.14) for the double-layer potential with constant basis function were known but unpublished. The formulae (2.15) and (2.16) for the double-layer potential and some other linear basis function, as well as the corresponding formulae (2.26) and (2.28) for the single-layer potential are new in this setting. As we observed that all formulae of these three cases have major parts in common, we were able to elaborate the simultaneous computation of the potentials for all three linear shape functions of a triangle. These results are presented in Section 2.2.4 for the double-layer potential and in Section 2.3.4 for the single-layer potential. The results of Section 4.1 show good speedups of the related computational times ranging from 2.35 to 3.15 by the new simultaneous computation.

The second part of the paper is devoted to the efficient implementation of the suggested evaluation routines for modern multi- and many-core (co)processors with wide SIMD registers. It has become more or less standard in scientific codes to utilize shared- and distributed-memory parallelism achieved by OpenMP and MPI and thus to use the computational power of all available cores. However, in recent years the theoretical peak performance of CPUs has also been rising due to the capabilities of vector processing units able to perform simultaneous computations on vectors of data. This concept, known as Single Instruction Multiple Data (SIMD), becomes increasingly important. Indeed, the newest AVX512 (Advanced Vector Extensions) instruction set is able to operate on 512 bits of floating-point data which translates to 8 double-precision operands. Neglecting in-core vectorization can thus reduce the performance by the factor of 8 (or even 16 in single-precision arithmetic). Recently, several papers have been published dealing with many-core vectorized implementation of numerical methods, see [10] for 2D BEM for the Laplace equation, [11] for efficient quadrature routines in the context of the finite element method (FEM), [12, 13] for stencil-based simulations of geophysical flows, [14, 15, 16] for the performance of CFD codes, or [17] for the acceleration of the finite element tearing and interconnecting (FETI) solver.

Vectorization can be achieved via different strategies. One option is the inline assembly code or compiler-specific intrinsic functions for compute-intensive kernels. Although these can achieve optimal speedup, the code is not portable between multiple architectures. A second option is to use wrapper libraries providing vector implementation of common mathematical functions in several vector instructions sets (including, e.g., SSE4.2, AVX2, or AVX512) resulting in a portable implementation. In [18] we describe the application of the Vc library [19] to both the semi-analytic and numerical BEM assembly. The VCL library [20] can be used in a similar fashion. In [21] we used OpenMP SIMD pragmas described by the OpenMP standard [22] for the vectorization of the regularized numerical assembly of BEM matrices. Differently from [21], where we showed that the efficiency of this approach can get very close to the optimal values, in this paper we use OpenMP SIMD to accelerate the presented analytic evaluations. Although this approach is a bit less explicit than the methods mentioned above, the compiler is able to perform additional optimizations and can contribute to better performance.

This part of the paper is structured as follows. Code optimizations employed for the efficient parallelization and vectorization of the semi-analytic assembly and the exact evaluation of the representation formula are presented in Section 3. In Sections 4.2, 4.3 we provide results obtained

on multi- and many-core architectures represented by Intel Xeon and Xeon Phi families. The suggested rather simple threading approach leads to optimal speedup on all tested architectures, see Tables 4.2-4.4 for detailed results. For the performance of the vectorized code we refer to Tables 4.5-4.7, where one can see that changing the width of a SIMD vector processed simultaneously by vector processing units leads to significant speedups ranging from 4.95 to 7.75 for the matrix assembly and the evaluation of the representation formula, respectively.

2 Analytic evaluation of singular integrals

In the following we consider the Dirichlet boundary value problem for the Laplace equation in three spatial dimensions. We discuss analytical formulae to compute the single- and double-layer potentials for plane triangles and linear shape functions. In particular, we present some analytical formulae which are new in the setting of [4, Section C.2]. The presented complete set of formulae allows the simultaneous computation of the integral operators for the three linear shape functions of a triangle at once to reduce the computational times significantly.

2.1 Model problem

In particular, we solve

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

where $\Omega \subset \mathbb{R}^3$ denotes a bounded Lipschitz domain and $g \in H^{1/2}(\partial\Omega)$ is the given Dirichlet datum. An explicit formula for the solution to (2.1) is given by, see, e.g., [23],

$$u(\widetilde{x}) = \int_{\partial\Omega} v(\widetilde{x}, y) w(y) \,\mathrm{d}s_y - \int_{\partial\Omega} \frac{\partial}{\partial n_y} v(\widetilde{x}, y) g(y) \,\mathrm{d}s_y \quad \text{for } \widetilde{x} \in \Omega \tag{2.2}$$

with $w := \partial u/\partial n$ and $v\colon \mathbb{R}^3 \to \mathbb{R}^3$ denoting the fundamental solution to the Laplace equation in 3D, i.e.,

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

The unknown Neumann datum $w \in H^{-1/2}(\partial\Omega)$ can be determined by solving the weakly singular boundary integral equation obtained from (2.2) by taking the limit $\Omega \ni \widetilde{x} \to x \in \partial\Omega$,

$$Vw(x) = \frac{1}{2} g(x) + Kg(x) \quad \text{for almost all } x \in \partial\Omega \tag{2.3}$$

with the single- and double-layer boundary integral operators

$$V\colon H^{-1/2}(\partial\Omega) \to H^{1/2}(\partial\Omega), \qquad Vw(x) := \int_{\partial\Omega} v(x, y) w(y) \,\mathrm{d}s_y,$$

$$K\colon H^{1/2}(\partial\Omega) \to H^{1/2}(\partial\Omega), \qquad Kg(x) := \int_{\partial\Omega} \frac{\partial}{\partial n_y} v(x, y) g(y) \,\mathrm{d}s_y,$$

respectively. Both boundary integral operators are linear and bounded, and the $H^{-1/2}(\partial\Omega)$-ellipticity of V ensures unique solvability of (2.3), see, e.g., [23]. The variational formulation equivalent to the equation (2.3) used for the discretization by the boundary element method reads

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

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

To derive a boundary element discretization of (2.4) we decompose the polyhedral boundary $\partial\Omega$ into E planar triangular boundary elements τ_k with N mesh nodes x^i in total. We introduce two conforming discrete approximation spaces, namely

$$S_h^{1,0}(\partial\Omega) := \operatorname{span}\{\varphi_i\}_{i=1}^N \subset H^{1/2}(\partial\Omega),$$
$$S_h^{1,-1}(\partial\Omega) := \operatorname{span}\left\{\varphi^{\mathrm{pw}}_{3(k-1)+1}, \varphi^{\mathrm{pw}}_{3(k-1)+2}, \varphi^{\mathrm{pw}}_{3(k-1)+3}\right\}_{k=1}^E \subset H^{-1/2}(\partial\Omega).$$

Here, $S_h^{1,0}(\partial\Omega)$ denotes the space of piecewise linear affine and globally continuous functions, while $S_h^{1,-1}(\partial\Omega)$ is the space of piecewise linear affine functions which can be discontinuous between elements. We denote the related basis functions by

$$\varphi_i(x) := \begin{cases} 1 & \text{for } x = x^i, \\ 0 & \text{for } x = x^j \neq x^i, \\ \text{piecewise linear} & \text{otherwise,} \end{cases} \qquad \varphi^{\mathrm{pw}}_{3(k-1)+m} := \begin{cases} \psi_{\tau_k,m} & \text{in } \tau_k, \\ 0 & \text{otherwise,} \end{cases}$$

where $\psi_{\tau_k,m}$ $(m = 1, 2, 3)$ are the linear shape functions related to the m–th node of triangle τ_k. Using the discrete spaces to approximate the boundary data

$$w \approx w_h := \sum_{j=1}^{3E} w_j \varphi_j^{\mathrm{pw}}, \qquad g \approx g_h := \sum_{i=1}^{N} g_i \varphi_i \tag{2.5}$$

in (2.4) and testing with $\varphi_\ell^{\mathrm{pw}}$ leads to the system of linear equations

$$\mathsf{V}_h \underline{w} = \left(\frac{1}{2}\mathsf{M}_h + \mathsf{K}_h\right) \mathsf{P}_h \underline{g} \tag{2.6}$$

with $\underline{w}$, $\underline{g}$ containing the coefficients w_j, g_i from (2.5), the boundary element matrices

$$[\mathsf{V}_h]_{\ell,j} := \frac{1}{4\pi} \int_{\tau_{\lfloor \ell/3 \rfloor}} \varphi_\ell^{\mathrm{pw}}(x) \int_{\tau_{\lfloor j/3 \rfloor}} \varphi_j^{\mathrm{pw}}(y) \frac{1}{|x-y|} \,\mathrm{d}s_y \,\mathrm{d}s_x, \tag{2.7}$$

$$[\mathsf{K}_h]_{\ell,j} := \frac{1}{4\pi} \int_{\tau_{\lfloor \ell/3 \rfloor}} \varphi_\ell^{\mathrm{pw}}(x) \int_{\tau_{\lfloor j/3 \rfloor}} \varphi_j^{\mathrm{pw}}(y) \frac{n_y \cdot (x-y)}{|x-y|^3} \,\mathrm{d}s_y \,\mathrm{d}s_x, \tag{2.8}$$

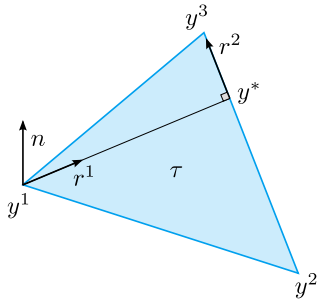
$$[\mathsf{M}_h]_{\ell,j} = \int_{\tau_{\lfloor \ell/3 \rfloor} \cap \tau_{\lfloor j/3 \rfloor}} \varphi_\ell^{\mathrm{pw}}(x) \varphi_j^{\mathrm{pw}}(y) \,\mathrm{d}s_x,$$

and P_h realizing the natural translation from $S_h^{1,0}(\partial\Omega)$ to $S_h^{1,-1}(\partial\Omega)$ as the continuous basis functions φ_i can be represented by a local shape function $\psi_{\tau,j}$ in each triangle τ of the support of φ_i. The numerical solution can then be plugged into the discretized representation formula

$$u_h(\widetilde{x}) := \int_{\partial\Omega} v(\widetilde{x}, y) w_h(y) \,\mathrm{d}s_y - \int_{\partial\Omega} \frac{\partial}{\partial n_y} v(\widetilde{x}, y) g_h(y) \,\mathrm{d}s_y \quad \text{for } \widetilde{x} \in \Omega \tag{2.9}$$

to evaluate the solution in the domain.

Both $\mathsf{M}_h, \mathsf{P}_h$ from (2.6) are sparse and not interesting from the perspective of assembly. In [21] we discussed the assembly of the remaining matrices V_h, K_h by a regularized numerical scheme described, e.g., in [24]. An alternative way is to compute the inner integral analytically and perform numerical quadrature only for the remaining integral. In the following we concentrate on the latter approach and provide closed-form analytic formulae for the relevant integrals. In particular, we focus on the simultaneous computations of the integrals related to all linear affine shape functions of a boundary element.



$$s_\tau = |y^* - y^1| \qquad s_x = (x - y^1) \cdot r^1$$
$$t_\tau = |y^3 - y^2| \qquad t_x = (x - y^1) \cdot r^2$$
$$t_* = (y^1 - y^2) \cdot r^2 \qquad u_x = (x - y^1) \cdot n$$
$$\alpha_1 = -t_*/s_\tau$$
$$\alpha_2 = (t_\tau - t_*)/s_\tau$$

Figure 2.1: Local coordinate system and basic quantities used in the analytic evaluation.

2.2 Double-layer potential

We consider the analytic computation of the collocation integrals of the double layer potential

$$D_j(x) = \frac{1}{4\pi} \int_\tau \frac{n_y \cdot (x - y)}{|x - y|^3} \psi_{\tau,j}(y) \, \mathrm{d}s_y, \qquad j \in \{1, 2, 3\}, \tag{2.10}$$

where τ is a plane triangle, $x \in \mathbb{R}^3$ is a collocation point, and $\psi_{\tau,j}$, $j \in \{1, 2, 3\}$, are the local linear shape functions of the triangle τ associated with its three nodes $y^j \in \mathbb{R}^3$.

For the analytic evaluation we reuse the notation as in [4, Section C.2] with the exception that for better readability we use y^j to denote the nodes of the triangle instead of x^j. Each triangle defines a local coordinate system with one of its nodes serving as the origin, by default we choose y^1. The orthonormal basis is spanned by the normal vector associated to the triangle, the direction of the edge opposite to the origin, and the perpendicular direction in the plane of the triangle. Angle tangents α_1, α_2 and the length s_τ describe the size and the shape of the triangle. The quantities s and t denote the local coordinates of a point $y \in \tau$, while s_x, t_x, and u_x are the local coordinates of the collocation point x. For an illustration and corresponding quantities see Figure 2.1.

2.2.1 Double-layer potential for a shape function linear in s

The analytic representation of the integral D_1 for the linear form function $\psi_{\tau,1}$ is given in [4, Section C.2.2] by

$$\begin{aligned} D_1(x) = D_s(x) &= \frac{1}{4\pi} \int_\tau \frac{n_y \cdot (x - y)}{|x - y|^3} \psi_{\tau,1}(y) \, \mathrm{d}s_y \\ &= \frac{1}{4\pi} \int_0^{s_\tau} \frac{s_\tau - s}{s_\tau} \int_{\alpha_1 s}^{\alpha_2 s} \frac{u_x}{((t - t_x)^2 + (s - s_x)^2 + u_x^2)^{3/2}} \, \mathrm{d}t \, \mathrm{d}s \\ &= \frac{1}{4\pi s_\tau} \left(F_s^D(s_\tau, \alpha_2) - F_s^D(0, \alpha_2) - F_s^D(s_\tau, \alpha_1) + F_s^D(0, \alpha_1) \right) \end{aligned} \tag{2.11}$$

where

$$\begin{aligned} F_s^D(s, \alpha) = &- \frac{1}{2} u_x \log\left(v^2 + A_1 v + B_1\right) + (s_\tau - s_x) \frac{u_x}{|u_x|} \arctan \frac{2v + A_1}{2G_1} \\ &+ \frac{1}{2} u_x \log\left(v^2 + A_2 v + B_2\right) - (s_\tau - s_x) \frac{u_x}{|u_x|} \arctan \frac{2v + A_2}{2G_2} \\ &- u_x \frac{\alpha}{\sqrt{1 + \alpha^2}} \log\left(\sqrt{1 + \alpha^2}(s - p) + \sqrt{(1 + \alpha^2)(s - p)^2 + q^2}\right) \end{aligned} \tag{2.12}$$

with the coefficients A_i, B_i, G_i as used in [4, Section C.2]. Here, the local coordinate system is set up with the origin y^1. To compute the integrals for the other two shape functions, i.e, D_2 and D_3, local coordinate systems can be set up with the origin y^2 and y^3, respectively. Thus, the total effort to compute all three integrals of (2.10) corresponds to three evaluations of (2.11).

Our aim is to reduce the computational time of evaluating D_1, D_2, and D_3 to about one third, i.e., the time necessary to evaluate one of the integrals.

2.2.2 Double-layer potential for a constant shape function

In addition to (2.11), the formula for the constant shape function is known as

$$\begin{aligned} D_0(x) &= \frac{1}{4\pi}\int_\tau \frac{n_y\cdot(x-y)}{|x-y|^3}\,\mathrm{d}s_y = \frac{1}{4\pi}\int_0^{s_\tau}\int_{\alpha_1 s}^{\alpha_2 s} \frac{u_x}{\left((t-t_x)^2+(s-s_x)^2+u_x^2\right)^{3/2}}\,\mathrm{d}t\,\mathrm{d}s \\ &= \frac{1}{4\pi}\left(F_0^D(s_\tau,\alpha_2)-F_0^D(0,\alpha_2)-F_0^D(s_\tau,\alpha_1)+F_0^D(0,\alpha_1)\right) \end{aligned} \tag{2.13}$$

where

$$F_0^D(s,\alpha) = \frac{u_x}{|u_x|}\arctan\frac{2v+A_1}{2G_1} - \frac{u_x}{|u_x|}\arctan\frac{2v+A_2}{2G_2}. \tag{2.14}$$

The calculations follow the lines of [4, Section C.2.2] by canceling the term $\frac{s_\tau - s}{s_\tau}$. Therefore, the logarithmic term of $F_s^D(s,\tau)$ is canceled and the numerator of the remaining integral changes to $(\alpha s - t_x)$. Thus, we apply the same partial fraction decomposition as in [4, Section C.2.2] where only the coefficients

$$D_1 = D_2 = 0, \qquad E_{1/2} = \pm\frac{\sqrt{1+\alpha^2}}{2}\left(q \pm \frac{t_x-\alpha s_x}{1+\alpha^2}\right)$$

change. Note that the symbols D_1, D_2 from the previous formula are local variables introduced in [4, Section C.2.2] and do not denote the integrals (2.10). In the two auxiliary integrals $I_{1/2}$ the logarithmic term drops out and it remains to compute

$$I_{1/2} = \pm\frac{u_x}{|u_x|}\arctan\frac{2v+A_{1/2}}{2G_{1/2}}.$$

2.2.3 Double-layer potential for a shape function linear in t

The third case, which is rather simple to compute, is

$$D_t(x) = \frac{1}{4\pi}\int_0^{s_\tau}\int_{\alpha_1 s}^{\alpha_2 s} \frac{u_x(t-t_x)}{\left((t-t_x)^2+(s-s_x)^2+u_x^2\right)^{3/2}}\,\mathrm{d}t\,\mathrm{d}s.$$

Using the formula [25, p. 970, eq. 207]

$$\int \frac{z}{(z^2+a^2)^{3/2}}\,\mathrm{d}z = -\frac{1}{\sqrt{z^2+a^2}}$$

we get

$$D_t(x) = -\frac{u_x}{4\pi}\int_0^{s_\tau}\left[\frac{1}{\sqrt{(t-t_x)^2+(s-s_x)^2+u_x^2}}\right]_{\alpha_1 s}^{\alpha_2 s}\mathrm{d}s.$$

This term is known from the calculation of D_s. In particular, it is related to

$$-\alpha u_x \int \frac{1}{\sqrt{(1+\alpha^2)(s-p)^2+q^2}}\,\mathrm{d}s$$

in the calculation of F_s^D in [4, p. 249] as

$$(1+\alpha^2)(s-p)^2+q^2 = (1+\alpha^2)\left(s-\frac{\alpha t_x+s_x}{1+\alpha^2}\right)^2+\frac{(t_x-\alpha s_x)^2}{1+\alpha^2} = (\alpha s-t_x)^2+(s-s_x)^2+u_x^2.$$

In case of F_s^D, this integral results in the last term of (2.12). Similarly, we can state

$$D_t(x) = \frac{1}{4\pi}\left(F_t^D(s_\tau,\alpha_2)-F_t^D(0,\alpha_2)-F_t^D(s_\tau,\alpha_1)+F_t^D(0,\alpha_1)\right) \tag{2.15}$$

where

$$F_t^D(s,\alpha) = -u_x\frac{1}{\sqrt{1+\alpha^2}}\log\left(\sqrt{1+\alpha^2}(s-p)+\sqrt{(1+\alpha^2)(s-p)^2+q^2}\right). \tag{2.16}$$

Note that the antiderivatives (2.12), (2.14), and (2.16) involve the same major terms and $F_s^D(s,\alpha)$, $F_0^D(s,\alpha)$, and $F_t^D(s,\alpha)$ can thus be computed simultaneously. This is much more efficient than three separate evaluations of (2.11) by switching the origin among y^1, y^2, and y^3. To compute the integrals D_2 and D_3 related to the other two shape functions $\psi_{\tau,2}$ and $\psi_{\tau,3}$ we build linear combinations of D_s, D_0, and D_t.

2.2.4 Simultaneous computation of double-layer potentials for linear shape functions

The linear shape functions $\psi_{\tau,j}$, $j\in\{1,2,3\}$, which are equal to 1 in the node y^j and 0 in the other nodes of the triangle τ, can be represented in the local basis 1, $\frac{s-s_\tau}{s_\tau}$, and $t-t_x$ with respect to the origin y^1 by

$$\begin{aligned}
\psi_{\tau,1}(s,t) &= 0\cdot 1+1\cdot\frac{s_\tau-s}{s_\tau}+0\cdot(t-t_x),\\
\psi_{\tau,2}(s,t) &= \frac{1}{\alpha_1-\alpha_2}\left[\left(\frac{t_x}{s_\tau}-\alpha_2\right)1+\alpha_2\frac{s_\tau-s}{s_\tau}+\frac{1}{s_\tau}(t-t_x)\right],\\
\psi_{\tau,3}(s,t) &= \frac{1}{\alpha_2-\alpha_1}\left[\left(\frac{t_x}{s_\tau}-\alpha_1\right)1+\alpha_1\frac{s_\tau-s}{s_\tau}+\frac{1}{s_\tau}(t-t_x)\right].
\end{aligned} \tag{2.17}$$

Thus, we can compute D_2 and D_3 from D_s, D_0, and D_t. Using the notation

$$\begin{aligned}
L_1(s,\alpha) &:= \frac{u_x}{2s_\tau(\alpha_2-\alpha_1)}\log\left(v^2+A_1v+B_1\right),\\
T_1(s,\alpha) &:= \frac{1}{s_\tau(\alpha_2-\alpha_1)}\frac{u_x}{|u_x|}\arctan\frac{2v+A_1}{2G_1},\\
L_2(s,\alpha) &:= \frac{u_x}{2s_\tau(\alpha_2-\alpha_1)}\log\left(v^2+A_2v+B_2\right),\\
T_2(s,\alpha) &:= \frac{1}{s_\tau(\alpha_2-\alpha_1)}\frac{u_x}{|u_x|}\arctan\frac{2v+A_2}{2G_2},\\
L_3(s,\alpha) &:= \frac{-u_x}{s_\tau(\alpha_2-\alpha_1)}\frac{1}{\sqrt{1+\alpha^2}}\log\left(\sqrt{1+\alpha^2}(s-p)+\sqrt{(1+\alpha^2)(s-p)^2+q^2}\right),
\end{aligned} \tag{2.18}$$

taking into account some terms of the coefficients of (2.17), for the main terms of the integrals D_s, D_0, and D_t, we can write

$$D_j(x) = \frac{1}{4\pi}\left(F_j^D(s_\tau,\alpha_2) - F_j^D(0,\alpha_2) - F_j^D(s_\tau,\alpha_1) + F_j^D(0,\alpha_1)\right), \tag{2.19}$$

where

$$F_j^D(s,\alpha) = \delta_{j,1}L_1(s,\alpha) + \delta_{j,2}T_1(s,\alpha) + \delta_{j,3}L_2(s,\alpha) + \delta_{j,4}T_2(s,\alpha) + \delta_{j,5}L_3(s,\alpha) \tag{2.20}$$

with the coefficients of Table 2.1. Moreover, we compute the term $F_j^D(s_\tau,\alpha) - F_j^D(0,\alpha)$ at once. Now we are able to compute D_1, D_2, and D_3 simultaneously based on the computation of the major terms (2.18).

Table 2.1: Coefficients of the antiderivatives $F_j^D(s,t)$ in (2.20) of the integrals D_j in (2.19).

j	$\delta_{j,1}$	$\delta_{j,2}$	$\delta_{j,3}$	$\delta_{j,4}$	$\delta_{j,5}$
1	$-(\alpha_2-\alpha_1)$	$(s_\tau-s_x)(\alpha_2-\alpha_1)$	$(\alpha_2-\alpha_1)$	$-(s_\tau-s_x)(\alpha_2-\alpha_1)$	$\alpha(\alpha_2-\alpha_1)$
2	α_2	$-(t_x-\alpha_2 s_x)$	$-\alpha_2$	$t_x-\alpha_2 s_x$	$-(1+\alpha\alpha_2)$
3	$-\alpha_1$	$t_x-\alpha_1 s_x$	α_1	$-(t_x-\alpha_1 s_x)$	$1+\alpha\alpha_1$

Remark 1. While we compared the results of the old formula and the new combined formulae, we observed small, but noticeable differences in some specific cases. These differences are related to finite precision arithmetic and cancellation of a few digits in computing the sums of the antiderivatives. This effect is observed in both versions and is related to dominant arctangent terms. This occurs for cases when $t_x^2 \gg (s_x^2 + u_x^2)$ where the evaluation point x is far from the triangle τ. To avoid this situation, we use some precomputations to find an appropriate single origin from y^1, y^2, and y^3 in the actual implementation . The related precomputations can be done without any square root (otherwise necessary to set up the local coordinate system) or any other expensive computations and only lead to a little additional effort.

2.3 Single-layer potential

For the single-layer potential we consider the analytic computation of the collocation integrals

$$S_j(x) = \frac{1}{4\pi}\int_\tau \frac{1}{|x-y|}\psi_{\tau,j}(y)\,\mathrm{d}s_y, \qquad j\in\{1,2,3\}, \tag{2.21}$$

where τ is a plane triangle, $x\in\mathbb{R}^3$ is the collocation point, and $\psi_{\tau,j}$, $j\in\{1,2,3\}$, are the local linear shape functions of the triangle τ associated with its three nodes y^1, y^2, $y^3\in\mathbb{R}^3$.

2.3.1 Single-layer potential for a shape function linear in s

The analytic representation of the single-layer potential $S_1(x)$ for the linear shape function $\psi_{\tau,1}$ is given in [26, p. 10] and [9, eq. (2.6)] by

$$\begin{aligned} S_1(x) = S_s(x) &= \frac{1}{4\pi}\int_\tau \frac{1}{|x-y|}\psi_{\tau,1}(y)\,\mathrm{d}s_y = \frac{1}{4\pi s_\tau}\int_0^{s_\tau}\int_{\alpha_1 s}^{\alpha_2 s} \frac{s_\tau - s}{\sqrt{(s-s_x)^2+(t-t_x)^2+u_x^2}}\,\mathrm{d}t\,\mathrm{d}s \\ &= \frac{1}{4\pi s_\tau}\left(F_s^S(s_\tau,\alpha_2) - F_s^S(0,\alpha_2) - F_s^S(s_\tau,\alpha_1) + F_s^S(0,\alpha_1)\right) \end{aligned} \tag{2.22}$$

with

$$
\begin{aligned}
F_s^S(s,\alpha) = {} & \frac{1}{2}(s_x - s)(s + s_x - 2s_\tau) \log\left[\alpha s - t_x + \sqrt{\beta^2(s-p)^2 + q^2}\right] \\
& + \frac{1}{4}s^2 + \frac{1}{2}s(s_x - 2s_\tau) - \frac{1}{2}u_x^2 \log\left[q + \sqrt{\beta^2(s-p)^2 + q^2}\right] \\
& + \frac{1}{2\beta^2}(t_x - \alpha s_x)\left[q + \sqrt{\beta^2(s-p)^2 + q^2}\right] \\
& + \frac{1}{2\beta}[\alpha q^2 + 2(t_x - \alpha s_x)(s_x - s_\tau)] \log\left[\beta(s-p) + \sqrt{\beta^2(s-p)^2 + q^2}\right] \\
& - \frac{1}{2}u_x^2 \log\left[(\beta^2 q - (\alpha s_x - t_x))\frac{\beta^2(s-p)^2}{(q + \sqrt{\beta^2(s-p)^2 + q^2})^2}\right. \\
& \qquad \left. + 2\alpha q \frac{\beta^2(s-p)}{q + \sqrt{\beta^2(s-p)^2 + q^2}} + \beta^2 q + (\alpha s_x - t_x)\right] \\
& - 2u_x(s_x - s_\tau) \arctan\frac{1}{u_x}\left[\frac{[\beta^2 q - (\alpha s_x - t_x)](s-p)}{q + \sqrt{\beta^2(s-p)^2 + q^2}} + \alpha q\right],
\end{aligned}
\tag{2.23}
$$

and $\beta = \sqrt{1+\alpha^2}$. Note that the constant terms have been dropped compared to [26], as these terms cancel in the summation (2.22).

Analogously as in the case of the double-layer potential discussed in Section 2.2, we will reduce the total effort of computing the contributions S_1, S_2, and S_3 to about one third when compared to the old approach of switching the origin of the local coordinate system among y^1, y^2, and y^3.

2.3.2 Single-layer potential for a constant shape function

In addition to (2.22), the formula for the constant form function is given in [4, Section C.2.1]

$$
\begin{aligned}
S_0(x) &= \frac{1}{4\pi}\int_\tau \frac{1}{|x-y|}\,\mathrm{d}s_y = \frac{1}{4\pi}\int_0^{s_\tau}\int_{\alpha_1 s}^{\alpha_2 s} \frac{1}{\sqrt{(s-s_x)^2 + (t-t_x)^2 + u_x^2}}\,\mathrm{d}t\,\mathrm{d}s \\
&= \frac{1}{4\pi}\left(F_0^S(s_\tau,\alpha_2) - F_0^S(0,\alpha_2) - F_0^S(s_\tau,\alpha_1) + F_0^S(0,\alpha_1)\right)
\end{aligned}
\tag{2.24}
$$

where

$$
\begin{aligned}
F_0^S(s,\alpha) = {} & (s - s_x)\log\left[\alpha s - t_x + \sqrt{\beta^2(s-p)^2 + q^2}\right] - s \\
& + \frac{\alpha s_x - t_x}{\beta}\log\left[\beta(s-p) + \sqrt{\beta^2(s-p)^2 + q^2}\right] \\
& + 2u_x \arctan\frac{1}{u_x}\left[\frac{[\beta^2 q - (\alpha s_x - t_x)](s-p)}{q + \sqrt{\beta^2(s-p)^2 + q^2}} + \alpha q\right].
\end{aligned}
\tag{2.25}
$$

2.3.3 Single-layer potential for a shape function linear in t

The third case for the shape function linear in t reads

$$
\begin{aligned}
S_t(x) &= \frac{1}{4\pi}\int_0^{s_\tau}\int_{\alpha_1 s}^{\alpha_2 s} \frac{t - t_x}{\sqrt{(s-s_x)^2 + (t-t_x)^2 + u_x^2}}\,\mathrm{d}t\,\mathrm{d}s \\
&= \frac{1}{4\pi}\int_0^{s_\tau}\left[\sqrt{(s-s_x)^2 + (t-t_x)^2 + u_x^2}\right]_{t=\alpha_1 s}^{\alpha_2 s}\,\mathrm{d}s
\end{aligned}
$$

$$S_t(x) = \frac{1}{4\pi}\left(F_t^S(s_\tau,\alpha_2) - F_t^S(0,\alpha_2) - F_t^S(s_\tau,\alpha_1) + F_t^S(0,\alpha_1)\right) \tag{2.26}$$

where

$$F_t^S(s,\alpha) = \int \sqrt{(s-s_x)^2 + (\alpha s - t_x)^2 + u_x^2}\,\mathrm{d}s. \tag{2.27}$$

Using the same transformations and substitutions as in [4, Section C.2.1], we can calculate

$$F_t^S(s,\alpha) = \frac{q^2}{2\beta}\log\left(\beta(s-p) + \sqrt{\beta^2(s-p)^2+q^2}\right) + \frac{1}{2}(s-p)\sqrt{\beta^2(s-p)^2+q^2} \tag{2.28}$$

where $\beta = \sqrt{1+\alpha^2}$. In detail, we rewrite (2.27) as

$$F_t^S(s,\alpha) = \int \sqrt{\beta^2(s-p)^2+q^2}\,\mathrm{d}s$$

and apply the transformation [4, page 247]

$$s = p + \frac{q}{\beta}\sinh u, \quad \frac{\mathrm{d}s}{\mathrm{d}u} = \frac{q}{\beta}\cosh u \tag{2.29}$$

to obtain

$$F_t^S(s,\alpha) = \frac{q^2}{\beta}\int \cosh^2 u\,\mathrm{d}u = \frac{q^2}{2\beta}\left(u + \frac{1}{2}\sinh(2u)\right).$$

For the inverse transformation of (2.29) we use

$$\sinh u = \frac{\beta}{q}(s-p), \qquad \frac{1}{2}\sinh(2u) = \sinh u\cosh u = \sinh u\sqrt{1+\sinh^2 u}$$

to obtain

$$F_t^S(s,\alpha) = \frac{q^2}{2\beta}u + \frac{s-p}{2}\sqrt{q^2+\beta^2)(s-p)^2}.$$

If we now use, see [4, page 248],

$$u = \log\left(\beta(s-p) + \sqrt{\beta^2(s-p)^2+q^2}\right) - \log q$$

and drop the constant term $\log q$ in the antiderivative, we obtain (2.28).

2.3.4 Simultaneous computation of single-layer potentials for linear shape functions

We can reuse the linear combinations (2.17) to compute the integrals (2.21) as

$$\begin{aligned}
S_1(x) &= S_s(x),\\
S_2(x) &= \frac{1}{\alpha_1-\alpha_2}\left[\left(\frac{t_x}{s_\tau} - \alpha_2\right)S_0(x) + \alpha_2 S_s(x) + \frac{1}{s_\tau}S_t(x)\right],\\
S_3(x) &= \frac{1}{\alpha_2-\alpha_1}\left[\left(\frac{t_x}{s_\tau} - \alpha_1\right)S_0(x) + \alpha_1 S_s(x) + \frac{1}{s_\tau}S_t(x)\right].
\end{aligned} \tag{2.30}$$

In the computation of the antiderivatives of $S_s(x)$, $S_0(x)$, and $S_t(x)$ the same major terms show up. The terms

$$\log\left[\beta(s-p)+\sqrt{\beta^2(s-p)^2+q^2}\right], \qquad \sqrt{\beta^2(s-p)^2+q^2}$$

are present in all three antiderivatives, and the terms

$$\log\left[\alpha s - t_x + \sqrt{\beta^2(s-p)^2+q^2}\right], \qquad \arctan\frac{1}{u_x}\left[\frac{[\beta^2 q-(\alpha s_x - t_x)](s-p)}{q+\sqrt{\beta^2(s-p)^2+q^2}}+\alpha q\right]$$

occur in $F_s^S(x)$ and $F_0^S(x)$. Thus, the three antiderivatives $F_s^S(\cdot,\cdot)$, $F_0^S(\cdot,\cdot)$, and $F_t^S(\cdot,\cdot)$ (and S_1, S_2, S_3 as well) can be computed at once at about the cost of computing $F_s^S(\cdot,\cdot)$.

3 Vectorized implementation

Numerical evaluation of the boundary element integrals is a generally very expensive operation from the computational point of view. Whether using a fully numerical approach with a large number of quadrature points or the discussed analytic formulae requiring often conditional evaluation of expensive arithmetic operations, a naive implementation would lead to unacceptable computational times for even moderately sized meshes. Although the quadratic complexity of the classical BEM can be reduced to nearly linear using some fast BEM approach, an efficient implementation of the kernel integration routines is still necessary. In [21] we have presented a parallelized and vectorized implementation of the fully numerical BEM quadrature routines based on regularized four-dimensional integrals [24]. In what follows we focus on the efficient implementation of the analytic evaluation described in the previous section.

Operations like floating point division, square root, logarithm, or arctangent belong to the most time consuming ones. Although the fully numerical evaluation does not have to use all of these functions, it usually requires sampling the kernel in a very large number of quadrature nodes. Thus, the analytic evaluation is a competitive alternative. In Table 3.1 and Table 3.2 from [27] we provide latency in core clock cycles and reciprocal throughput (processor cycles per instruction) for some of the functions occurring in the formulae. One can see that the instructions for the reciprocal square root (VRSQRTPD) or fused multiple-add operations (VFMADD) present in modern instruction sets can lead to a significant performance gain. However, only with proper vectorization and parallelization the full potential of current processors can be reached. The wide SIMD (Single Instruction Multiple Data) registers of the Intel Xeon or Xeon Phi processors are able to process up to four or eight, respectively, double precision operands simultaneously.

Table 3.1: Haswell instructions latency and throughput.

instruction	instruction set	latency	reciprocal throughput
FMUL	x87 f. p.	5	1
FDIV	x87 f. p.	10–24	8–18
FSQRT	x87 f. p.	10–23	8–17
FPATAN	x87 f. p.	96–156	—
VDIVPD	AVX	19–35	16–28
VFMADD	AVX	5	0.5
VSQRTPD	AVX	28–29	16–28

Table 3.2: Knights Landing instructions latency and throughput.

instruction	instruction set	latency	reciprocal throughput
VDIVPD	AVX512	32	32
VSQRTPD	AVX512	37	16
VRSQRTPD	AVX512	7	2
VFMADD	AVX512	6	0.5

Although some level of automatic vectorization is supported by modern compilers, it is usually necessary to significantly modify the original code and data structures and to assist the compiler by using OpenMP directives to achieve optimal SIMD behaviour. In combination with the shared memory parallelization this can lead to speedups in orders of tens to hundreds compared to sequential and scalar code.

In the next sections we describe the combination of OpenMP threading and vectorization for the assembly of the boundary element matrices following the so-called VIPO (Vectorize Innermost, Parallelize Outermost) approach [28]. The presented approaches are implemented in the boundary element library BEM4I [29] developed at the IT4Innovations National Supercomputing Center.

3.1 OpenMP threading

One of the characteristics of the Intel Xeon Phi family of (co)processors is the relatively high number of available cores compared to Intel Xeon series widely used in HPC centers and high-performance servers. On top of that, to achieve optimal behaviour it is often recommended to employ hyper-threading (up to 4-way). This means that to perform well on Intel Xeon Phi the code should scale well up to tens or even hundreds of threads.

Similarly as in finite-element codes, in BEM4I the element-based strategy is used to assemble the boundary element matrices V_h, K_h from (2.7), (2.8). Since the assembly routines are almost identical for both matrices, in the following we concentrate on V_h only. In our setting, the approximated local contribution to V_h from the elements τ_ℓ, τ_k reads

$$\mathbb{R}^{3\times 3} \ni [\mathsf{V}_h^{\ell,k}]_{m,n} := \frac{1}{4\pi} \Delta_\ell \sum_{q=1}^{Q} w_q \varphi^{\mathrm{pw}}_{3(\ell-1)+m}(x^q) \int_{\tau_k} \varphi^{\mathrm{pw}}_{3(k-1)+n}(y) \frac{1}{\|x^q - y\|} \,\mathrm{d}s_y \tag{3.1}$$

with Δ_ℓ denoting the area of τ_ℓ, and quadrature weights and points w_q, $x^q \in \tau_\ell$, respectively. The quadrature points are transferred to τ_ℓ from the reference element

$$\hat{\tau} := \left\{ \mu \in \mathbb{R}^2 \colon \mu_1 \in (0,1),\ \mu_2 \in (0, 1-\mu_1) \right\} \tag{3.2}$$

by the affine mapping $R^\ell \colon \hat{\tau} \to \tau_\ell$,

$$x := R^\ell(\mu) := y^{\ell_1} + \begin{bmatrix} y^{\ell_2} - y^{\ell_1} & y^{\ell_3} - y^{\ell_1} \end{bmatrix} \mu \tag{3.3}$$

with ℓ_m denoting the global index of the m-th node of the triangle τ_ℓ. Note that in the provided code listings we switch to the zero-based indexing.

The assembly strategy is outlined in Listing 3.1 – we loop over the pairs of triangles, assemble a local matrix and map it to the global matrix. OpenMP threading is employed for the outer loop over triangles, so that the granularity of each task is not too fine. All auxiliary buffers used in `getLocalMatrix` and described in the following section are allocated on a per-thread basis.

```
#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 );
    globalMatrix.add( tau_l, tau_k, localMatrix );
} }
```

Listing 3.1: Threaded element-based assembly.

```
#pragma omp parallel for schedule( static, 1 )
for( int rank = 0; rank < omp_get_num_threads( ); ++rank ){
  offset = rank * chunkSize;
  representationFormula( points + offset, result + offset );
}
```

Listing 3.2: Threaded evaluation of the representation formula.

Thus, only the `globalMatrix` is shared among threads. Note, however, that due to the supports of the used trial and test functions being limited to a single triangle, no thread-private (`atomic` or `critical` OpenMP clauses) operations are necessary in the `add` function.

For the implementation of the discretized representation formula the threading is rather simple. The array of evaluation points is split into chunks and every thread is responsible for a subset of points, see Listing 3.2.

3.2 OpenMP vectorization

Another level of parallelism in BEM4I is provided by the SIMD paradigm. The vectorization of code can be achieved in different ways with varying ease of use. Starting from direct interaction with vector instructions via the assembly language, over the compiler-specific intrinsic functions, or an external compiler- and architecture-independent library [19, 20], the most user-friendly option is to leverage the current OpenMP specification [22] similarly as in the case of threading. In the following, our aim is to vectorize the loop generated by the quadrature sum displayed in (3.1). The local assembly, i.e., the function `getLocalMatrix` from Listing 3.1, consists of several steps including

1. transferring reference quadrature nodes from $\hat{\tau}$ to τ_ℓ by (3.3),
2. setting up of the local coordinate system of τ_k, see Figure 2.1,
3. obtaining local coordinates of the quadrature nodes, see Figure 2.1,
4. evaluation of boundary integral operators as in Sections 2.2, 2.3.

In the following we take a closer look at the implementation of individual phases resulting in the implementation of `getLocalMatrix` provided in Listing 3.7.

Let us first comment on the initial step of mapping the quadrature points from $\hat{\tau}$ to τ_ℓ. The vectorized code as implemented in BEM4I is hinted in Listing 3.3. The following text should be read in conjunction with Figure 3.1 depicting the described optimizations.

First of all, the arrays `y1`, `y2`, `y3` of the length 3 contain the coordinates of the vertices of the current inner triangle τ_k. More important is the structure of the arrays `mu1`, `mu2` containing the coordinates of the 2D quadrature points defined in the reference triangle (3.2). A classical way

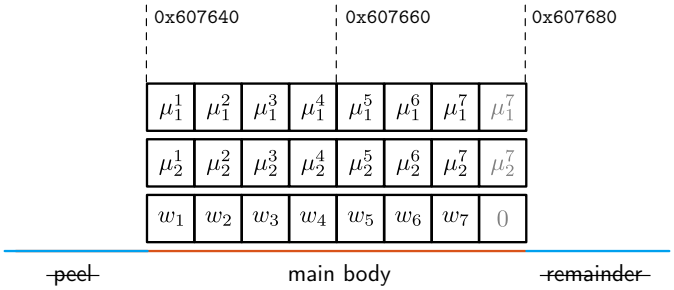


Figure 3.1: Aligned and padded quadrature nodes in the structure-of-arrays (SoA) format.

```
void getGlobalQuadratureNodes(
 double * y1, double * y2, double * y3
 ) {
// transform reference points to the triangle y1, y2, y3
#pragma omp simd \
aligned( x1, x2, x3, mu1, mu2 : align ) \
simdlen( width )
  for ( int i = 0; i < QPad; ++i ) {
    x1[ i ] = y1[ 0 ] + ( y2[ 0 ] - y1[ 0 ] ) * mu1[ i ]
                      + ( y3[ 0 ] - y1[ 0 ] ) * mu2[ i ];
    ... // compute x2, x3
} }
```

Listing 3.3: Transforming reference quadrature points to the current triangle.

of storing coordinates of multiple points, say $\mu^1, \mu^2, \ldots, \mu^q$, is the so-called array-of-structures (AoS) format corresponding to the vector

$$\left[\mu^1, \mu^2, \ldots, \mu^q\right] = \left[\mu_1^1, \mu_2^1, \mu_1^2, \mu_2^2, \ldots, \mu_1^q, \mu_2^q\right].$$

For better memory accesses utilizing unit-strided loads and stores, however, it is usually advisable to restructure the data into the structure-of-arrays format (SoA), i.e.,

$$\left[\mu_1^1, \mu_1^2, \ldots, \mu_1^q, \mu_2^1, \mu_2^2, \ldots, \mu_2^q\right].$$

This is due to the fact that it is more common for scientific codes that the same coordinates of different points interact rather than different coordinates of the same point. Thus, by storing the first coordinates of all reference nodes in `mu1` and similarly for `mu2` we ensure that the memory is loaded in unit-strides (instead of two-strided loads for the AoS format of 2D reference quadrature nodes). Similar SoA strategy is employed for the storage of the resulting quadrature nodes stored in the arrays `x1`, `x2`, `x3` and thus also memory stores are performed in unit strides (instead of strides of three for the AoS storage of 3D nodes in the current boundary element).

Another important detail to point out is that all the arrays processed in a vectorized fashion should be aligned in the memory. For the current Intel Xeon and Xeon Phi (co)processors the size of the cache line is 64 bytes and proper alignment can be achieved by using the clause `__attribute__( ( aligned( 64 ) ) )` in case of static allocation or by the `_mm_malloc` operator instead of the classical `malloc` or `new`. This ensures that the compiler will not be forced to create the so-called peel loop for elements stored in front of such a boundary and not filling the whole vector register. A counterpart to the peel loop possibly performed at the end of the iterations is the remainder loop processing aligned elements but again not filling the whole register. This can be overcome by data padding. For the semi-analytic assembly of BEM matrices this is

crucial, since the usual size of quadrature used in BEM4I is $Q = 7$, see [30], and with AVX512 employed this would lead to the whole loop being processed in the remainder part. Padding to the multiple of a cache-line size is also beneficial to avoid false sharing between multiple threads. Thus, for the instruction sets of interest (SSE4.2, AVX2, IMCI, AVX512) we set the value `QPad` to 8 for $Q = 7$. The dummy padding values in the quadrature point arrays are duplicates of the last quadrature point coordinates, the arrays of weights is filled with 0 so that the final result is not influenced, see Figure 3.1.

Lastly, to tell the compiler that the loop can be processed in a vectorized fashion we add `#pragma omp simd` with the relevant clauses. These include the `aligned` clause stating that the listed arrays are aligned at the `align`-byte boundary and the `simdlen` clause specifying that the length of each vector should be `width`. The `width` parameter is set to 2, 4, 8 for SSE4.2, AVX2, and AVX512 (or IMCI), respectively, corresponding to the number of double-precision operands fitting into the SIMD register.

The next step in the assembly is to build the local system of coordinates corresponding to τ_k as described in Section 2. For simplicity, we assume here that the local coordinate system based on the origin `y1` is suitable for all quadrature points (see Remark on page 8). This leads to a simple series of computations given in Listing 3.4 and no vectorization is employed here. In case that different local coordinate systems (based on `y2` or `y3`) have to be used for different quadrature nodes, the vectors `r1`, `r2` and the triangle parameters `alpha1`, `alpha2`, `stau` would have to be set up individually for each quadrature point. The optimizations listed above including alignment, padding, AoS to SoA conversion and vectorized processing would be used in a similar fashion in this case.

```
void setUpLocalCoordinateSystem(
 ) {
  r2[ 0 ] = y3[ 0 ] - y2[ 0 ];
  r2[ 1 ] = y3[ 1 ] - y2[ 1 ];
  r2[ 2 ] = y3[ 2 ] - y2[ 2 ];

  tk = std::sqrt( dot( r2, r2 ) );

  r2[ 0 ] /= tk; r2[ 1 ] /= tk; r2[ 2 ] /= tk;

  ... // set up r1, n (coordinate system)
  ... // set up stau, alpha1. alpha2 (triangle properties)
}
```

Listing 3.4: Setting up the local coordinate system.

```
void getLocalQuadratureNodes(
  double * y1
  ) {
// transform global coordinates into local
#pragma omp simd \
aligned( x1, x2, x3, sx, tx, ux : align ) \
simdlen( width )
  for ( int i = 0; i < QPad; ++i ) {
    sx[ i ] = ( x1[ i ] - y1[ 0 ] ) * r1[ 0 ]
            + ( x2[ i ] - y1[ 1 ] ) * r1[ 1 ]
            + ( x3[ i ] - y1[ 2 ] ) * r1[ 2 ];
    ... // compute tx, ux
} }
```

Listing 3.5: Transforming global coordinates into local for each quadrature point.

```
1
2  double evaluatePrimitive(
3    double s, double sx, ...
4  ) {
5  ...
6
7
8    // do not add to f in special case
9    if ( abs( s - sx ) > _EPS ) {
10     if ( tmp2 < 0.0 ) {
11       // masked evaluation of sqrt
12       tmp3 = hh1 / ( sqrt(
13         tmp1 * tmp1 + q_sq ) - tmp2 );
14     } else {
15       // masked evaluation of sqrt
16       // same argument as above!
17       tmp3 = tmp2 + sqrt(
18         tmp1 * tmp1 + q_sq );
19     }
20
21
22
23     // masked evaluation of log
24     f += ( s - sx ) * log( tmp3 );
25 }
26 ...
27   return f;
28 }
```

(a) Scalar code.

```
1  #pragma omp declare simd simdlen(
       width )
2  double evaluatePrimitive(
3    double s, double sx, ...,
4  ) {
5  ...
6    // unmasked evaluation of sqrt
7    tmp4 = sqrt( tmp1 * tmp1 + q_sq );
8    // do not add to f in special case
9    if ( abs( s - sx ) > _EPS ) {
10     if ( tmp2 < 0.0 ) {
11       // masked division only
12       tmp3 = hh1 / ( tmp4 - tmp2 );
13
14     } else {
15
16       // masked addition only
17       tmp3 = tmp2 + tmp4;
18
19     }
20   } else {
21     tmp3 = 1.0;
22   }
23   // unmasked evaluation of log
24   f += ( s - sx ) * log( tmp3 );
25
26 ...
27   return f;
28 }
```

(b) Vectorized code.

Listing 3.6: Typical excerpt from the evaluation of the antiderivative.

Since the analytic evaluations from Sections 2.2, 2.3 are based on the local coordinates of the quadrature points, the global coordinates of the Q evaluation points stored in `x1`, `x2`, `x3` have to be translated to the local ones stored in `sx`, `tx`, `ux` as shown in Listing 3.5. Since the transformation only consists of three dot products, all optimizations described above can easily be applied and the loop can be efficiently vectorized by `#pragma omp simd`.

Up to this point, the vectorization approach is quite similar to the one employed for the numerical assembly discussed in [21]. What makes the semi-analytic assembly more complicated is the evaluation of the auxiliary functions D_j, S_j from (2.19), (2.30), respectively. Except for the singularity for $x \to y$ one has to take care of special configurations when, e.g., the arguments of the log function approach zero or when dividing by $u_x \to 0$. A typical excerpt from the original implementation of these functions is shown in Listing 3.6a. In the scalar version, the function `evaluatePrimitive` is applied to a single instance of the variables `s`, `sx` and exactly one branch of the `if` statement in line 10 is executed – thus, at most one of the `sqrt` functions from lines 12 and 17 is called. In the vectorized call, however, the function will be evaluated for a whole vector of these variables. This leads to situations that the `if` condition from line 10 is only satisfied for some elements of the vector and the compiler generates masked instructions, see Figure 3.2 for an illustration of masked addition. Thus, the `sqrt` would be called twice with the same vector, but different parts of the result would be filtered out. The remedy is rather simple, in the optimized version from Listing 3.6b we evaluate the square root without a mask for the whole vector of arguments, see line 7. Instead, the mask is applied to filter out the result stored in `tmp4` in lines 12, 17, which is a much faster operation.

A similar strategy is applied to the call of `log` from line 24 in Listing 3.6a. Note that the `log` function is not a single CPU instruction but a series of elementary instructions including addition,

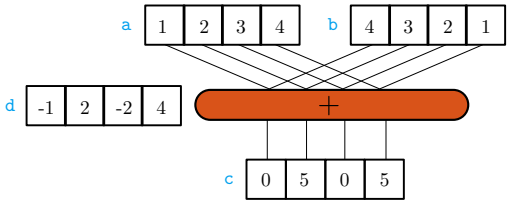


Figure 3.2: Masked evaluation of `c(d>0) = a + b`.

multiplication, or division, performing, e.g., a Taylor series to approximate the result. From our experience unmasked calls to `log` or `atan` are more efficient than their masked counterparts. To avoid the masked version we set the argument `tmp3` to a dummy value of `1.0` in line 21 of Listing 3.6b if the condition from line 9 is not satisfied. Thus, the result of the evaluation in line 24 is unchanged since $\log(1) = 0$. Note that the strategy of cheap masked evaluations of arguments instead of masked calls to expensive functions can be applied to `atan` in the same way. Finally, the function is annotated with the `#pragma omp declare simd` clause telling the compiler to generate its vectorized version.

Collecting the optimizations above results in the final assembly code presented in Listing 3.7. The function `collocationSingleLayer` represents the vectorized unified evaluation of S_1, S_2, S_3 from (2.30) and calls a function similar to the computationally expensive `evaluatePrimitive` from Listing 3.6b. The vectorization of the quadrature loop is enforced by `#pragma omp simd`.

```
1  getLocalMatrix(
2   int tau_l, int tau_k, FullMatrix & localMatrix
3  ) {
4  ...
5    // transform reference points to the triangle y1, y2, y3
6    getGlobalQuadratureNodes( y1, y2, y3 );
7    // set up local coordinate system r1, r2, n,
8    // and auxiliary variables stau, alpha1, alpha2
9    setUpLocalCoordinateSystem( );
10   // set up the coordinate systems and local coordinates
11   getLocalQuadratureNodes( y1 );
12   // compute local matrix entries
13 #pragma omp simd \
14 reduction( + : entry11, ..., entry33 ) \
15 private( ret1, ret2, ret3 ) \
16 aligned( sx, tx, ux, wx : align ) \
17 simdlen( width )
18   for ( int i = 0; i < QPad; ++i ) {
19     // evaluation of S_1, S_2, S_3
20     collocationSingleLayer( ret1, ret2, ret3, stau,
21       alpha1, alpha2, sx[ i ], tx[ i ], ux[ i ] );
22     // multiplication by weights
23     ret1 *= w[ i ]; ret2 *= w[ i ]; ret3 *= w[ i ];
24     // multiplication by test functions \phi_1, \phi_2, \phi_3
25     entry11 += phi1 * ret1;
26     ... // compute entry{12,13,21,22,23,31,32}
27     entry33 += phi3 * ret3;
28 }
29   localMatrix.set( 0, 0, entry11 );
30   ... // write entry{12,13,21,22,23,31,32}
31   localMatrix.set( 2, 2, entry33 );
32 }
```

Listing 3.7: Vectorized assembly of the local matrix.

When using BEM for solving certain problems, e.g., of shape optimization only the boundary traces of the solution are of importance. On the other hand, sometimes it is also necessary to

evaluate the discretized representation formula (2.9) in a relatively high number of evaluation points in the domain. The computation is very similar to the assembly of the BEM matrices as in Listing 3.7, however, with the difference that instead of looping over the quadrature points in line 18 we loop over all evaluation points assigned to a single OpenMP thread. This results in longer loops (without the artificial padding as in the case of matrix assembly) and more efficient vectorization which is also observed in the scalability experiments summarized in the following section.

4 Experiments

In this section we provide details on the performance of the boundary element codes presented above on different representatives of Intel multi- and many-core (co)processors.

4.1 Simultaneous evaluation of boundary integral operators

Firstly, let us concentrate on the speedup obtained by the unified evaluation of the singular integrals as presented in Sections 2.2.4, 2.3.4. For this purpose, an experimental C code outside of BEM4I was implemented. In the test we evaluated the functions D_j, S_j from (2.19), (2.30), respectively, in 125,000 evaluation points on a mesh with 28,952 surface triangles. The resulting number of calls to the unified functions is thus $3.619 \cdot 10^9$. The test was performed on a single node of the Salomon cluster located at the IT4Innovations National Supercomputing Center equipped with two 12-core Intel Xeon E5-2680v3 (Haswell) processors. We tested both GNU 6.3.0 and Intel 2017.0.2 C compilers with the respective compile lines `-O3 -fopenmp -std=c99` and `-xcore-avx2 -O3 -qopenmp -std=c99`. Each run was repeated five times, the presented results represent the average run.

The results are summarized in Table 4.1, where the individual columns represent time in seconds necessary for the evaluation of the collocation integrals by both the original and unified approaches and by both compilers. Clearly, the speedup due to the unified evaluation ranging from 2.35 to 3.15 justifies the effort invested in the analytic work. The timings also show that the Intel compiler is more benevolent to optimizations since the speedup with respect to the GNU version reaches 1.67 and 2.71 for D_j and S_j, respectively.

Table 4.1: Xeon E5-2680v3 (dual socket), evaluation times for the original vs. unified approach.

t[s] (speedup)	orig. (GNU)	simult. (GNU)	orig. (Intel)	simult. (Intel)
D_1, D_2, D_3	308.68	117.67 (2.62)	165.94	70.54 (2.35)
S_1, S_2, S_3	549.93	186.47 (2.95)	216.35	68.71 (3.15)

4.2 BEM4I – OpenMP threading

The second part of the performance experiments is devoted to the behaviour of BEM optimized for multi- and many-core (co)processors with wide SIMD registers as implemented in the BEM4I library. We tested both the assembly of the BEM matrices V_h, K_h from (2.7), (2.8) for a mesh consisting of 12,288 surface elements and the evaluation of the discrete solution u_h by (2.9) in 100,001 evaluation points. The chosen quadrature size for the assembly is $Q = 7$, for the purposes of vectorization the size of the underlying arrays (`QPad`) is padded to 8. The experiments

were performed on three Intel architectures, namely on the Salomon's two 12-core Xeon E5-2680v3 (Haswell) processors, 61-core Xeon Phi 7120 (Knights Corner) coprocessor available on Salomon's accelerated nodes, and on the second generation 64-core Xeon Phi 7210 (Knights Landing) processor available on the Taurus supercomputer at the Center for Information Services and High Performance Computing, TU Dresden. A unique feature of the Knights Landing generation is the presence of an on-chip high-bandwidth 16 GB MCDRAM memory, which served as the last-level cache shared by all cores for our experiments.

Let us firstly concentrate on the scalability of the code with respect to the number of OpenMP threads employed. Recall that the rather simple threading strategy is described in Listings 3.1, 3.2. The following results represent average timings computed from four subsequent runs. The compile line for the Intel compiler 2017.0.2 reads `${SIMD} -O2 -qopenmp -std=c++11` with the variable `SIMD` set to the most advanced vector instruction set available including `-xcore-avx2`, `-mmic`, and `-xmic-avx512`.

In Tables 4.2, 4.3, 4.4 we provide the assembly and evaluation times for all architectures and different number of OpenMP threads and the most advanced vector instruction set available on the given (co)processor. On the multi-core Xeon processor the speedup reaches the almost optimal values of 23.67 (23.56, 23.78) for the computation of V_h (K_h, u_h) when scaling from a single thread to the full team of 24 threads.

More interesting is the behaviour of the code on many-core Xeon Phi architectures. The speedup achieved on 61 threads of the first generation Knights Corner coprocessor reads 56.75 (58.43, 55.01) for V_h (K_h, u_h). However, due to the dual-issue nature of the dated Knights Corner's cores (based on Intel Pentium) the hyper-threading (up to four-way) is crucial. This results in the very reasonable speedup of 106.23 (124.33, 92.46).

Table 4.2: Xeon E5-2680v3 (dual socket), assembly times with different numbers of OpenMP threads.

t[s]	1	2	4	8	12	16	24
V_h	198.79	99.53	49.94	24.98	16.72	12.54	8.40
K_h	231.14	116.16	58.05	29.26	19.47	14.63	9.81
u_h	390.67	195.89	98.39	49.06	32.69	24.50	16.43

Table 4.3: Xeon Phi 7120P, assembly times with different numbers of OpenMP threads.

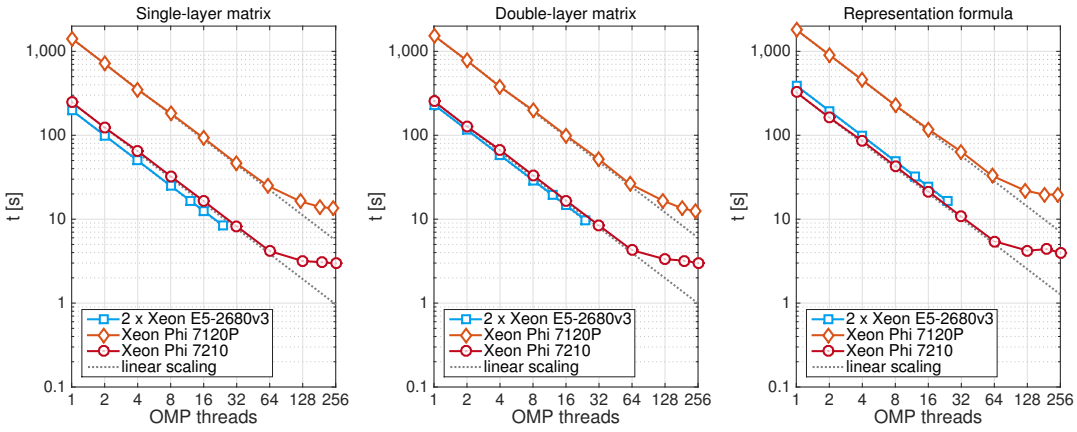
t[s]	1	2	4	8	16	32	61
V_h	1425.59	711.94	350.30	182.55	92.53	46.06	25.12
K_h	1539.15	777.04	377.81	198.50	99.59	51.57	26.34
u_h	1811.34	908.79	455.99	226.85	118.04	63.73	32.93
	122	183	244				
V_h	16.29	14.03	13.42				
K_h	16.68	13.49	12.38				
u_h	21.96	19.59	19.74				

The newer Knights Landing processor is based on more recent Intel Atom Airmont cores. The speedup without hyper-threading for V_h (K_h, u_h) reaches the value of 59.30 (59.45, 59.47) and with 4 threads per core the speedup reaches 83.55 (84.90, 81.18). For graphical visualization of

Table 4.4: Xeon Phi 7210, assembly times with different numbers of OpenMP threads.

t[s]	1	2	4	8	16	32	64
V_h	247.30	123.26	64.88	32.49	16.42	8.23	4.17
K_h	253.84	126.65	66.94	33.35	16.70	8.41	4.27
u_h	324.73	162.44	85.53	42.78	21.45	10.73	5.46
	128	192	256				
V_h	3.17	3.07	2.96				
K_h	3.33	3.21	2.99				
u_h	4.22	4.39	4.00				

the achieved results see Figure 4.1 summarizing the behaviour on all three architectures and the comparison to the optimal linear scaling (relevant up to the number of physical cores or 2-way hyper-threading in the case Xeon Phi 7120P).



(a) V_h. (b) K_h. (c) u_h.

Figure 4.1: Assembly times with different numbers of OpenMP threads.

4.3 BEM4I – OpenMP vectorization

Similarly as in the previous section, SIMD performance was evaluated with the Intel compiler 2017.0.2. The scalability of the code with respect to the width of the SIMD register was tested for various configurations. First of all, the experiments were run on the most efficient number of OpenMP threads, i.e., 24, 244, and 256 for Xeon E5-2680v3, Xeon Phi 7120P, and Xeon Phi 7210, respectively. The vector extensions were switched by setting the compile line variable `SIMD` to `-xsse4.2`, `-xcore-avx2`, `-xmic-avx512` based on the availability on the given architecture. The only option available and necessary for Xeon Phi 7120P is `-mmic`. The baseline for the test is given by compiling the code with the option `-no-vec -no-simd -qno-openmp-simd` instead of `${SIMD}` neglecting the OpenMP SIMD pragmas and forbidding vectorization by the compiler. However, note that this configuration also forbids the use of new instructions brought, e.g., by AVX512, albeit used in scalar (masked) mode. Thus, we also provide results obtained

Table 4.5: Xeon E5-2680v3 (dual socket), assembly times with different SIMD strategies.

t[s]	scalar	sse4.2(1)	sse4.2(2)	avx2(1)	avx2(2)	avx2(4)
V_h	16.52	16.51	13.28	15.47	12.48	8.40
K_h	17.13	17.09	13.46	16.53	13.39	9.81
u_h	33.28	33.05	23.53	31.53	23.82	16.43

Table 4.6: Xeon Phi 7120P, assembly times with different SIMD strategies.

t[s]	imci(1)	imci(2)	imci(4)	imci(8)
V_h	48.37	76.28	40.60	13.42
K_h	53.13	43.15	26.47	12.38
u_h	111.62	161.15	72.32	19.74

by the combination of the `SIMD` option and vectorization switched off. Lastly, we control the length of the vector processed simultaneously by setting the `width` variable from the listings in Section 3.2 to 2, 4, and 8 (if possible).

In Table 4.5 we provide the assembly and evaluation times obtained on two Haswell processors. Here, one can utilize the SSE4.2 and AVX2 instruction sets, which can operate on 2 and 4 double-precision operands, respectively. The label 'scalar' in the header of the table stands for the scalar code without any vector extension set allowed. The labels 'sse4.2(`width`)' represent versions with SSE4.2 and the vector width set by the `simdlen(width)` clause. Since the `width` parameter cannot be set to 1, we use `-xsse4.2 -no-vec -no-simd -qno-openmp-simd` to enforce SSE4.2 in the scalar (masked) mode. The same description holds for the remaining vector instruction sets and (co)processor architectures.

The speedup of the code utilizing AVX2 instructions with the native vector length of 4 with respect to the purely scalar version reads 1.97 (1.75, 2.03) for the processing of V_h (K_h, u_h). Theoretically, the expected speedup between AVX2 code and the scalar version should be 4. However, one has to take into account the lower frequency of the CPU when performing these instructions, and also different latencies. In Table 3.1 the latency of the standard square root is 10–23, while for AVX2 it raises to 28–29. The obtained results thus seem reasonable.

From the vectorization point of view the many-core Xeon Phi (co)processors are much more interesting since they provide registers able to accommodate 8 double-precision operands. The computation results for the first generation Xeon Phi 7120P are summarized in Table 4.6. In this case it is not possible to obtain pure scalar measurements as the code has always to be compiled with `-mmic` enforcing the IMCI instruction set and one can thus only change the `width` variable. The vectorized version with the vector length of 8 reaches the speedup 3.60 (4.29, 5.65) for V_h (K_h, u_h), which already proofs that neglecting the SIMD features of modern processors would lead to a rather poor performance.

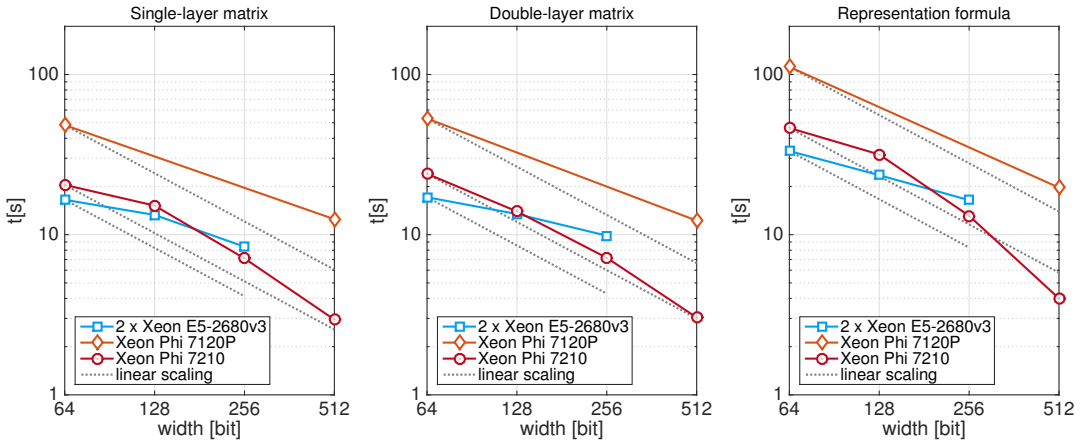
The effect of vectorization is most evident on the second generation of Xeon Phi processors codenamed Knights Landing. The addition of the AVX512 instruction set does not bring wider SIMD registers when compared to IMCI, however, the addition of new instructions plays an important role. Moreover, the AVX512 set is not specific to the Xeon Phi family and is also supported by the new multi-core Xeon processors (e.g., Skylake). The results obtained on Xeon Phi 7210 are summarized in Table 4.7. Comparing the code compiled with AVX512 and `width` set to the native value of 8 leads to the speedup 6.91 (8.03, 11.59) for V_h (K_h, u_h). Comparing only the effect of switching the `width` from 1 to 8 and using AVX512 gives the speedup 4.95 (4.95,

Table 4.7: Xeon Phi 7210, assembly times with different SIMD strategies.

t[s]	scalar	sse4.2(1)	sse4.2(2)	avx2(1)	avx2(2)	avx2(4)
V_h	20.46	20.55	15.19	20.49	15.18	7.11
K_h	24.00	24.04	13.97	23.97	14.01	7.17
u_h	46.36	46.63	31.68	46.66	31.78	13.12
	avx512(1)	avx512(2)	avx512(4)	avx512(8)		
V_h	14.64	14.34	6.66	2.96		
K_h	14.79	13.97	7.21	2.99		
u_h	31.00	30.97	12.69	4.00		

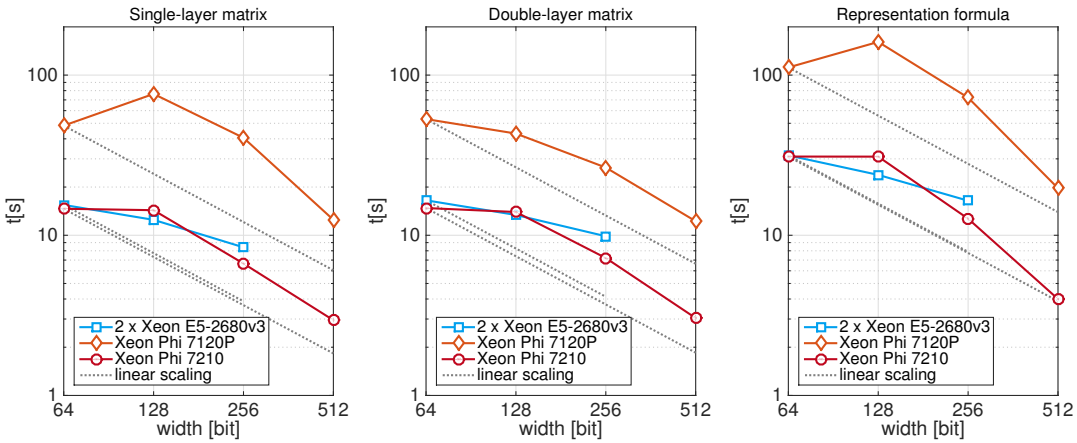
7.75). The better performance of the evaluation of the representation formula for u_h compared to the assembly of V_h and K_h is due to the easier unrolling commented on in Section 3.2. The length of these loops in BEM4I can be modified and the results presented here correspond to the length 32, i.e., 4 vectors of the size 8 are processed by a single loop. The situation is a bit more complicated for the object-oriented implementation of the assembly of the matrices but the results hint us where further optimizations are possible.

The relation between the computation time and the width of the SIMD register is again presented graphically in Figures 4.2, 4.3. In Figure 4.2 the `width` parameter is always changed together with the vector instruction set, for Xeon Phi 7210 we thus compare the versions 'scalar', 'sse4.2(2)', 'avx2(4)', and 'avx512(8)' from Table 4.7. In Figure 4.3 we always stick to the most advanced vector instruction set available on the given architecture and only change the parameter `width`, which for Xeon Phi 7210 translates to 'avx512(1)', 'avx512(2)', 'avx512(4)', and 'avx512(8)'. On both Xeon Phi (co)processors we can observe the overhead brought by vectorization, i.e., by changing `width` from 1 to 2. However, from this point on (comparing `width` ranging from 2 to 8) the line follows (or even outperforms) the optimal linear scaling trend.



(a) V_h. (b) K_h. (c) u_h.

Figure 4.2: Assembly times with different vector register widths and different vector extension sets – scalar (64 bit), SSE4.2 (128 bit), AVX2 (256 bit), IMCI/AVX-512 (512 bit).



(a) V_h. (b) K_h. (c) u_h.

Figure 4.3: Assembly times with different vector register widths and the most advanced vector extension set available on the given architecture – AVX2 (Xeon E5-2680v3), IMCI (Xeon Phi 7120P), AVX-512 (Xeon Phi 7210).

Finally, let us comment on the performance comparison between individual architectures. The theoretical dual-precision floating-point performance of a processor can be deduced from the formula

$$P := f \cdot N_{\text{core}} \cdot N_{\text{PD}} \cdot N_{\text{VPU}} \cdot N_{\text{FMA}}, \quad [P] = \text{FLOP/s}$$

with f denoting the core frequency, N_{core} the number of cores, N_{VPU} the number of vector processing units per core, and N_{FMA} equal to 2 if the fused multiply-add instruction is available and 1 otherwise. For the tested architectures, namely two Xeon E5-2690v3 processors, Xeon Phi 7120P, and Xeon Phi 7210, the performance respectively reads 960 GFLOP/s, 1,210.24 GFLOP/s, and 2,662.4 GFLOP/s. Theoretically, the performance gain obtained by the deployment of the code on the Knights Landing processor should read 2.77 and 2.20 when compared to the two Haswell and Knights Corner (co)processors, respectively. The speedup obtained in practice is given in Table 4.8, where we provide the ratios of best evaluation times reached on the respective architectures. The results hint that the actual gain can be better than expected, which may be caused by better processing of vector instructions in AVX512 and also by the available high-bandwidth memory serving as the last-level cache in case Xeon Phi 7210.

Table 4.8: Performance gain achieved by Xeon Phi 7210 over Xeon E5-2680v3 and Xeon Phi 7120P.

Xeon Phi 7210 vs.	Xeon E5-2680v3	Xeon Phi 7120P
V_h	2.84	4.53
K_h	3.28	4.14
u_h	4.11	4.94

5 Conclusion

The aim of the paper was twofold. Firstly, we described the simultaneous evaluation of boundary integral operators with piecewise linear trial functions. As suggested in [23], the local system of coordinates had to be set up three times individually for each local trial function supported on the triangular element. On the contrary, the approach described here in Section 2 uses the same coordinate system for all three evaluations which leads to a faster implementation. In Section 4.1 we showed that the speedup of the new code with respect to the original version ranges from 2.35 to 3.15.

The second part was devoted to the parallel and vectorized implementation of the assembly of BEM matrices and the evaluation of the representation formula which are two crucial steps of a boundary element simulation. The results in Section 4.2 show that the presented threading strategy with thread-private buffers works very well up to tens or hundreds of OpenMP threads both on multi- and many-core (co)processors. Specifically, the speedup on the 64-core Knights Landing processor reaches more than 80 when comparing 4-way hyper-threading to a serial run. The SIMD vectorization by OpenMP pragmas proves efficient especially on the Xeon Phi (co)processors with wide SIMD registers. The speedup with respect to the scalar code presented in Section 4.3 reaches 4.95 for the semi-analytic matrix assembly and the almost optimal value of 7.75 for the analytic evaluation of the representation formula. This result gives us a hint for further optimization for the assembly by unrolling the loop over the outer elements.

Although the theoretical performance of current processors is still rising, this is no longer due to the increased clock frequency. To efficiently utilize the resources, more programming effort is requested to utilize all available cores and SIMD lanes. The presented boundary integral operators are specific for the Laplace equation, however, the same procedure can be applied to the singular part of operators related to the Helmholtz operator in connection with the numerical scheme for the regular part, see, e.g., [9], or to the Lamé equation modelling linear elasticity problems. The analytical formulae for the single-layer potential of the Lamé equation are quite similar, see [4, Section C.2.3], and the double-layer potential can be represented by the single-layer potentials of the Lamé equation and by single- and double-layer potentials of the Laplacian, see [31] and [32, 33] for its use in fast methods. Moreover, the optimizations employed here, including AoS to SoA conversion, alignment of data, data padding, or unit-strided accesses to the memory, can be used in different scientific codes in a very similar fashion.

Acknowledgements

The presented research was supported by the project 'Efficient parallel implementation of boundary element methods' provided jointly by the Ministry of Education, Youth and Sports and OeAD under the grant numbers '7AMB17AT028' and 'CZ 16/2017'. JZ and MM further 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', the Large Infrastructures for Research, Experimental Development and Innovations project 'IT4Innovations National Supercomputing Center – LM2015070', and the grant SP2017/165 provided by VŠB – Technical University of Ostrava.

References

[1] E. E. Okon, R. F. Harrington, The potential integral for a linear distribution over a triangular domain, Internat. J. Numer. Methods Engrg. 18 (12) (1982) 1821–1828. doi:10.1002/nme.1620181206.

[2] D. E. Medina, J. A. Liggett, Exact integrals for three-dimensional boundary element potential problems, Comm. Appl. Numer. Methods 5 (8) (1989) 555–561. doi:10.1002/cnm.1630050809.

[3] M. Maischak, The analytical computation of the Galerkin elements for the Laplace, Lamé and Helmholtz equation in 3D-BEM, Tech. rep., Universität Hannover (2000).

[4] S. Rjasanow, O. Steinbach, The fast solution of boundary integral equations, Mathematical and Analytical Techniques with Applications to Engineering, Springer, New York, 2007.

[5] S. Nintcheu Fata, Explicit expressions for 3D boundary integrals in potential theory, Internat. J. Numer. Methods Engrg. 78 (1) (2009) 32–47. doi:10.1002/nme.2472.

[6] A. Salvadori, Analytical integrations in 3D BEM for elliptic problems: Evaluation and implementation, Int. J. Numer. Meth. Engrg. 84 (5) (2010) 505–542. doi:10.1002/nme.2906.

[7] M. J. Carley, Analytical formulae for potential integrals on triangles, ASME. J. Appl. Mech. 80 (4) (2013) 041008–041008–7. doi:10.1115/1.4007853.

[8] S. G. Mogilevskaya, D. V. Nikolskiy, The use of complex integral representations for analytical evaluation of three-dimensional BEM integrals—potential and elasticity problems, Q. Jl Mech. Appl. Math 67 (3) (2014) 505–523. doi:10.1093/qjmam/hbu015.

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

[10] L. Einkemmer, Evaluation of the Intel Xeon Phi 7120 and NVIDIA K80 as accelerators for two-dimensional panel codes, PLOS ONE 12 (6) (2017) 1–16. doi:10.1371/journal.pone.0178156.

[11] 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.

[12] L. Szustak, K. Rojek, T. Olas, L. Kuczynski, K. Halbiniak, P. Gepner, Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor, Scientific Programming 2015 (2015) 10.

[13] A. Lastovetsky, L. Szustak, R. Wyrzykowski, Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing, IEEE Transactions on Parallel and Distributed Systems 28 (3) (2017) 787–797. doi:10.1109/TPDS.2016.2599527.

[14] M. A. A. Farhan, D. K. Kaushik, D. E. Keyes, Unstructured computational aerodynamics on many integrated core architecture, Parallel Computing 59 (2016) 97–118, theory and Practice of Irregular Applications. doi:http://dx.doi.org/10.1016/j.parco.2016.06.001.

[15] I. Hadade, L. di Mare, Modern multicore and manycore architectures: Modelling, optimisation and benchmarking a multiblock CFD code, Computer Physics Communications 205 (2016) 32 – 47. doi:http://dx.doi.org/10.1016/j.cpc.2016.04.006.

[16] I. Z. Reguly, E. László, G. R. Mudalige, M. B. Giles, Vectorizing unstructured mesh computations for many-core architectures, Concurrency and Computation: Practice and Experience 28 (2) (2016) 557–577. doi:10.1002/cpe.3621.

[17] M. Merta, L. Riha, O. Meca, A. Markopoulos, T. Brzobohaty, T. Kozubek, V. Vondrak, Intel Xeon Phi acceleration of hybrid total FETI solver, Advances in Engineering Software (2017) –doi:10.1016/j.advengsoft.2017.05.001.

[18] 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.

[19] 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

[20] A. Fog, C++ vector class library (2017).
URL http://www.agner.org/optimize/vectorclass.pdf

[21] J. Zapletal, M. Merta, L. Malý, Boundary element quadrature schemes for multi- and many-core architectures, Computers & Mathematics with Applications 74 (1) (2017) 157–173, 5th European Seminar on Computing ESCO 2016. doi:10.1016/j.camwa.2017.01.018.

[22] OpenMP Architecture Review Board, OpenMP application program interface (7 2013).
URL www.openmp.org/mp-documents/openmp-4.5.pdf

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

[24] S. A. Sauter, C. Schwab, Boundary Element Methods, Springer Series in Computational Mathematics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-540-68093-2_4.

[25] I. Bronstein, K. Semendjajew, G. Musiol, H. Mühlig, Taschenbuch der Mathematik., 3rd Edition, Verlag Harri Deutsch, Frankfurt am Main, 1997.

[26] O. Steinbach, Galerkin– und Kollokations–Diskretisierungen für Randintegralgleichungen in 3D —Dokumentation—, internal report (2004).

[27] A. Fog, Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, Copenhagen University College of Engineering.
URL http://www.agner.org/optimize/instruction_tables.pdf

[28] R. Geva, Code Modernization Best Practices: Multi-level Parallelism for Intel® Xeon™ and Intel® Xeon Phi™ Processors, software.intel.com/en-us/articles/idf15-webcast-code-modernization-best-practices, [Online; accessed 11/5/2017] (2015).

[29] M. Merta, J. Zapletal, BEM4I, IT4Innovations National Supercomputing Center, VŠB – Technical University of Ostrava, Studentská 6231/1B, 708 33 Ostrava-Poruba, Czech Republic (2013).
URL http://bem4i.it4i.cz/

[30] J. Radon, Zur mechanischen Kubatur, Monatsh. Math. 52 (4) (1948) 286–300. doi:10.1007/BF01525334.

[31] V. D. Kupradze, T. G. Gegelia, M. O. Baseleisvili, T. V. Burculadze, Three-dimensional problems of the mathematical theory of elasticity and thermoelasticity., North-Holland Series in applied Mathematics and Mechanics. Vol. 25. Amsterdam, New York, Oxford: North-Holland Publishing Company, 1979.

[32] G. Of, O. Steinbach, W. L. Wendland, Applications of a fast multipole Galerkin boundary element method in linear elastostatics, Comput. Vis. Sci. 8 (3–4) (2005) 201–209. doi:10.1007/s00791-005-0010-9.

[33] M. Bebendorf, R. Grzhibovskis, Accelerating Galerkin BEM for linear elasticity using adaptive cross approximation, Math. Methods Appl. Sci. 29 (14) (2006) 1721–1747. doi:10.1002/mma.759.

Erschienene Preprints ab Nummer 2015/1

2015/1 O. Steinbach: Space-time finite element methods for parabolic problems

2015/2 O. Steinbach, G. Unger: Combined boundary integral equations for acoustic scattering-resonance problems problems.

2015/3 C. Erath, G. Of, F.–J. Sayas: A non-symmetric coupling of the finite volume method and the boundary element method

2015/4 U. Langer, M. Schanz, O. Steinbach, W.L. Wendland (eds.): 13th Workshop on Fast Boundary Element Methods in Industrial Applications, Book of Abstracts

2016/1 U. Langer, M. Schanz, O. Steinbach, W.L. Wendland (eds.): 14th Workshop on Fast Boundary Element Methods in Industrial Applications, Book of Abstracts

2016/2 O. Steinbach: Stability of the Laplace single layer boundary integral operator in Sobolev spaces

2017/1 O. Steinbach, H. Yang: An algebraic multigrid method for an adaptive space-time finite element discretization

2017/2 G. Unger: Convergence analysis of a Galerkin boundary element method for electromagnetic eigenvalue problems

2017/3 J. Zapletal, G. Of, M. Merta: Parallel and vectorized implementation of analytic evaluation of boundary integral operators