A multi-platform scaling study for an OpenMP parallelization of a discontinuous Galerkin ocean model

Balthasar Reuter, Vadym Aizinger, Harald Köstler

Article info
Article history:
Received 14 February 2015
Accepted 23 May 2015
Available online 1 June 2015

Keywords:
Discontinuous Galerkin finite element method
3D shallow water equations
OpenMP
Intel Xeon Phi
IBM PowerPC
IBM Power6

Abstract
We present a cross-platform scaling investigation for an OpenMP parallelization of UTBEST3D – a coastal and regional ocean code based on the discontinuous Galerkin finite element method. The study is conducted for a real life application on an unstructured computational mesh of the Northwest Atlantic with realistic topography and well resolved coast line on a broad selection of current computing platforms. Four numerical setups of increasing physical and computational complexity are used for comparison: barotropic with no vertical eddy viscosity, barotropic with an algebraic eddy viscosity parametrization, baroclinic with an algebraic eddy viscosity, and baroclinic with $k-c$ vertical turbulence closure. In addition to Intel Xeon and IBM Power6/PowerPC architectures, we also include Intel’s new MIC processor Xeon Phi in the evaluation. Good scalability is found across all platforms with Intel Xeon CPUs producing the best runtime results and Xeon Phi demonstrating the best parallel efficiency.

1. Introduction
Numerical ocean models are an essential component of the coupled simulations performed in the course of climate studies; they also represent one of the exascale challenge applications. The sheer size of the computational domain (comprising the global ocean or large parts of it) and its highly complex geometry prescribed by the bottom topography and coast lines call for a highly efficient numerical method that works well on unstructured meshes. A number of physical processes acting on multiple scales must be simulated in real-life applications; this puts high demands on the computational resources and requires good scalability in all parts of the numerical model.

Several aspects of ocean applications make them particularly interesting test subjects for a parallelization study. Computational domains employed in such models are comparatively thin with a typical aspect ratio of vertical to horizontal dimensions of 1:100 or more. The specific anisotropy of the problem is connected with the direction of the gravity force; it justifies the hydrostatic assumption under which the pressure is computed as a function of the free surface elevation rather than being a solution to the saddle point problem as it would be the case if the incompressible Navier–Stokes equations were considered instead. Thus, fully unstructured 3D meshes can be avoided and so-called 2.5-dimensional meshes can be used to simulate 3D ocean dynamics. This simplification has the potential to greatly benefit the design of highly scalable ocean simulators but, at the same time, requires well thought out data structures and algorithms to achieve desirable computational efficiency without sacrificing the flexibility of the numerical model.

Due to the high demands on computational resources arising from typical applications of ocean models it is not new to parallelize the simulation of ocean dynamics. In fact, a number of parallelization and performance studies exist for the established frameworks and models (see, e.g., HYCOM [1], POP [2], FVCOM [3], ADCIRC CG [4,5], HOMME [6], MPAS [7], POM [8,9], and recently for a Western Mediterranean Sea Model [10]). Moreover, for HYCOM [11,12], detailed performance models were developed from runtime measurements to predict response to changing system parameters such as the network speed or the CPU clock rate. All of them employ MPI as parallelization tool for distributed memory platforms; some also offer support for shared memory systems using OpenMP (e.g., HYCOM, POP, HOMME).

Rather new, however, is the use of accelerators like GPUs in the context of ocean or atmospheric simulations. Mak et al. [13] compared CUDA, OpenMP, and MPI implementations of ROMS and found the GPU version to deliver reasonable performance on much cheaper hardware. Another current project focuses on porting the
dynamical core of ICON to GPUs [14]. Since accelerator cards are
found increasingly often in newer supercomputers the importance
of the ability to exploit this type of architecture increases.

When porting existing code to GPUs the fact that core compo-
nents must be rewritten using, e.g., CUDA or OpenCL C to make
use of the accelerator cards is a serious flaw. Along with necessary
changes in data structures this can result in extensive code modi-
fications. Intel's new Many Integrated Cores (MIC) architecture
marketed as Intel Xeon Phi accelerator cards does not share this
drawback, it allows to employ the widely used programming lan-
guages C/C++ and Fortran in combination with standard parallel-
ization techniques such as OpenMP and MPI that are often
already being employed by the ported software. This was discussed
by Heinecke et al. [15] who showed that Intel's coprocessor pro-
duced better performance results for a data mining algorithm than
an Nvidia Tesla card. Early experiences with MIC were also docu-
mented by Schulz et al. [16] and Schmidl et al. [17] concluding that
porting of existing applications is straightforward but may not pro-
duce satisfactory performance for all kinds of algorithms.

WARIS, a framework for atmospheric simulation, was one of the
first applications from the area of climate studies which was
ported to the MIC architecture [18]. Yet, to our knowledge, our
DG code UTBEST3D is the first ocean simulator ported to and
assessed on the MIC architecture.

To analyze the performance of a numerical code on modern
computing hardware it is necessary to consider the memory hier-
archy (CPU-caches, shared memory with potentially non-uniform
access, distributed memory), possibly heterogeneous computing
units (e.g., compute nodes with accelerators), as well as the com-
unication and I/O hardware. This hardware-centered analysis has
to be interwoven with an investigation of computation/scaling
properties of the critical parts of the numerical algorithm.

For a fully featured ocean code simulating a realistic application
this type of analysis is a complex and labor intensive undertaking
we are not attempting to accomplish in this work. Rather a much
more narrow focus on the newly implemented shared-memory
parallelization and its cross-platform comparison was chosen for
this study. All single-core and MPI performance issues were thus
left out – as far as they did not affect our stated goals. The motiva-
tion behind this particular choice of approach boils down to the
following main points:

- conduct a performance and scaling evaluation of our OpenMP
parallelization of various parts of a fully featured regional ocean
simulator on a real-life computational domain using an
unstructured mesh;
- cover the whole range of important model setups of various
degrees of physical sophistication ranging from a simple baro-
tropic one-layer run nearly equivalent to a 2D case up to a fully
featured baroclinic multi-layer scenario with a two-equation
vertical turbulence closure;
- test and compare this OpenMP implementation on a represen-
tative selection of current architectures, including Intel Xeon,
IBM Power, and Intel Many Integrated Cores (MIC) systems.

The results of this study provide useful insights for our ongoing
work on efficient hybrid MPI/Open MP parallelizations (the main
features of the pure MPI parallelization of UTBEST3D were
presented in [19]) and address portability concerns of the present
implementation, including those for the new Intel MIC
architecture.

The rest of this paper is organized as follows. In the next section,
we give the mathematical formulation of the 3D shallow water
equations, briefly describe the DG discretization of this system,
and specify the test problem benchmarked in our scaling runs.
We then present the algorithms arising from numerical
approximations of different complexity levels and detail our
OpenMP parallelization. In Section 4, an analysis of runtime por-
tions and scaling results for five different platforms are compared
and discussed. Last, some conclusions and a brief outlook round off
this work.

2. Numerical method and test setup

2.1. 3D baroclinic shallow water equations

For \( a \in \mathbb{R}^d, b \in \mathbb{R}^d \), we denote by \( ab \) the tensor-product of \( a \) and
\( b \). Our system includes the equations for elevation (1), momen-
tum (2), continuity (3), temperature/salinity transport (4), and
transport of turbulent quantities (5). The primary unknowns are the
total height of the water column \( h = z - z_0 \), where \( z, z_0 \) are the
values of the vertical coordinate at the free surface and the sea bed,
correspondingly. \( u_y = (u, v) \) the horizontal velocity, \( r \in \{\theta, s\} \) the
temperature/salinity, \( m \in \{k, \psi\} \) the turbulence variables. The ver-
tical velocity component \( w \) is computed diagnostically from (3).

\[
\begin{align*}
\partial_t h + \nabla \cdot \mathbf{u}_y &= 0, \quad (1) \\
\partial_t \mathbf{u}_y + \nabla \cdot (u_y \mathbf{u}_y) &= \partial_z (v_z \mathbf{u}_y) + \nabla \gamma (gh + p) - f_y \mathbf{k} \times \mathbf{u}_y = F_u - \nabla g z_0, \quad (2) \\
\nabla \cdot \mathbf{u} &= 0, \quad \text{(3)} \\
\partial_t \mathbf{r} + \nabla \cdot (\mathbf{u} \mathbf{r}) &= -F_m, \quad \text{(4)} \\
\partial_t m - \partial_z (v_m \partial_z m) &= F_m. \quad \text{(5)}
\end{align*}
\]

Here, \( p \) is the baroclinic pressure correction

\[
p(x, y, z) = \frac{g}{\rho_0} \int_0^z (\rho(\theta(x, y, \hat{z}), s(x, y, \hat{z})) - \rho_0) d\hat{z}. \quad \text{(6)}
\]

\( \rho_0 \) the reference density, and \( \rho(\theta, s) \) the density computed from the
equation of state [20] for temperature in degrees Celsius \( \theta \) and
salinity in ppt \( s \)

\[
\rho(\theta, s) = 999.83 + 0.808 s - 0.0708 (1 + 0.068 \theta) \theta - 0.003 (1 - 0.012 \theta) (35 - s) \theta. \quad \text{(7)}
\]

The atmospheric pressure gradient and tidal potential are combined
into a body force term \( F_u, F_m, F_m \) denote sources/sinks in the trans-
port equations for temperature/salinity and turbulence variables,
correspondingly. \( \nabla g z = (\partial_z, 0) \) is the horizontal gradient operator,
\( f_y \) the Coriolis coefficient, \( k = (0, 0, 1) \) a unit vertical vector, \( g \)
acceleration due to gravity, \( v_x, v_y, v_m \) are the vertical eddy viscosity/
diffusivity coefficients in the momentum, transport, and vertical
turbulence equations, correspondingly.

The following boundary conditions were specified for the system:

- At the bottom boundary \( \partial \Omega_{bot} \), we have no normal flow
\( \mathbf{u}(z_0) \cdot \mathbf{n} = 0 \), where \( \mathbf{n} = (n_x, n_y, n_z) \) is an exterior unit normal
to the boundary, and the quadratic slip condition for the horizontal
velocity components \( v_z \partial_z \mathbf{u}_y(z_0) = C_f \sqrt{v^2(z_0) + w^2(z_0)} \mathbf{u}_y(z_0) \),
where \( C_f \) is the friction coefficient.
- The free surface boundary conditions in the case of no wind are
just \( \nabla u(z) \cdot \mathbf{n} = \nabla v(z) \cdot \mathbf{n} = 0 \).
- Land boundary: No normal flow \( \mathbf{u} \cdot \mathbf{n} = 0 \).
- Open sea boundary: Prescribed surface elevation \( \zeta_{sw}(t, x, y) \).

In addition, homogeneous Neumann boundary conditions for
temperature and salinity were used where necessary.

UTBEST3D provides vertical eddy viscosity parameterizations of
various levels of computational and conceptual complexity. In
order of increasing sophistication, those include a constant eddy
viscosity coefficient, algebraic (zeroth order) models, as well as
one- and two-equation turbulence closures. In this work, only an algebraic and the k–ε schemes were employed.

The algebraic vertical eddy viscosity closure was modeled using the quadratic formula given by Davies in [21]:

\[ \nu_t = K_i (\bar{u}^2 + \bar{v}^2) \sqrt{\nu_t}, \]  

(8)

where \( \bar{u}, \bar{v} \) are the depth averaged horizontal velocities, \( \nu_t \) a typical long wave frequency taken as \( 10^{-4} \text{s}^{-1} \), and \( K_i = 2 \times 10^{-4} \) a dimensionless coefficient.

The two-equation turbulence closure implemented in UTBEST3D is based on the generic turbulence length scale model proposed by Umfaff and Burchard in [22] and extensively evaluated by Warner et al. [23]. The main advantage of this formulation is the ability to switch between several two-equation models, including k–ε and Mellor–Yamada [24], by changing a few constant parameters. Due to space constraints, we refer to [19] for a detailed description of our version of the two-equation vertical eddy viscosity model.

2.2. Computational mesh

Let \( \Omega(t) \subset \mathbb{R}^3 \) be the time dependent shallow water domain. In the case of simulations that do not account for wetting/drying – as in the present study – we assume that the top boundary of the domain \( \partial \Omega_{\text{top}}(t) \) is the only moving boundary. The bottom \( \partial \Omega_{\text{bot}} \) and the lateral \( \partial \Omega_{\text{lat}}(t) \) boundaries are assumed to be fixed although the height of the lateral boundaries can vary with time following the movements of the free surface. We denote by \( \Omega_{\text{xy}} \) the projection of \( \Omega(t) \) onto the xy-plane. In the non-wetting case, domain \( \Omega_{\text{bot}} \) does not change with the time.

The geometry of the 3D domain used for our finite element discretization is fully specified by a 2D unstructured triangular mesh and the values of free surface elevation and bathymetry at the nodes of this mesh. This domain with piecewise linear top and bottom boundaries is subdivided into layers, thus producing a vertical column of prismatic 3D elements for each triangle of the 2D mesh. The topmost and the bottommost faces in each column are permitted to be non-parallel to the xy-plane to provide a better representation of the free surface and the bathymetry. In the default case, vertical layers are defined to be equidistant and correspond to local z-levels; therefore, the local number of layers depends on the local bathymetry. A user specified mesh grading parameter controls the minimum height of the bottommost elements as a fraction of the full layer height. The elements in the bottom layer whose shortest vertical edge is smaller than this value are merged with the neighbor immediately above – possibly producing hanging nodes.

An additional mesh feature is activated in the case of one- and two-equation turbulence models. Those equations are discretized on one-dimensional vertical segments that pass through the quadrature points of the prismatic elements (due to the alignment of the 2D and 3D meshes, the quadrature points on prisms lie on vertical lines passing through the quadrature points of the corresponding 2D triangles). The lengths of those segments are dynamically adjusted to account for the movements of the free surface.

2.3. Semi-discrete formulation

In the framework of the local discontinuous Galerkin (LDG) methods [25], all second order terms are approximated with the help of an auxiliary flux variable similarly to the mixed method. E.g., the second order transport Eq. (4) will now change to

\[ \partial_t r + \nabla \cdot (ur) - \partial_z q_x = F_r, \]  

(9)

\[ q_x = v_r \partial_z r. \]  

(10)

Similar expressions hold for other second order equations in (1)–(5). The semi-discrete DG formulation of our system is obtained by subdividing the domain into some partition \( T \), defining piecewise polynomial (discontinuous) approximation spaces on \( T \), multiplying the equations with test functions spanning these polynomial spaces, and, finally, integrating the resulting equations by parts over elements of \( T \). The main steps of this procedure performed for a transport equation are detailed in Appendix A, and a full derivation of the semi-discrete system is presented in [19].

We approximate \( c(t, .) = (h, \mathbf{u}_g, q_{m, z}, w, r, q_g, m, q_{m, z}) \), a solution to the mixed version of (1)–(5), with a discontinuous finite element approximation \( \mathbf{C}(t, .) = (\mathbf{H}, \mathbf{U}_g, \mathbf{Q}_{m, z}, W, R, \mathbf{Q}_z, \mathbf{Q}_m) \) whose detailed definition is also given in Appendix A. Introducing \( a(\cdot, \cdot) \) – the (trilinear) advection, \( d(\cdot, \cdot) \) – the (bilinear) diffusion, \( l(\cdot) \) – the linear right hand side, \( g_d(\cdot) \) – the linear gravity forcing and continuity operators and denoting by \( \partial_{\text{xy}} \) the \( L^2 \)-scalar product on a domain \( \Omega \subset \mathbb{R}^2 \), a compact form of (1)–(5) can be written as

\[ (\partial_t H, \varphi_h)_2 + a_h(\mathbf{U}_g, H, \varphi_h) = 0, \]  

(11)

\[ (\partial_t \mathbf{U}_g, \varphi_{\mathbf{U}})_3 + a_h(\mathbf{U}_g, \mathbf{U}_g, \varphi_{\mathbf{U}}) + g_d(H, \varphi_{\mathbf{U}}) + d_h(Q_{m, z}, \varphi_{Q_{m, z}}) = l_h(\varphi_{\mathbf{U}}), \]  

(12)

\[ (Q_{m, z}, \lambda_{Q_{m, z}})_3 = c_m(\mathbf{U}_g, \lambda_{Q_{m, z}}), \]  

(13)

\[ (\partial_t w, \varphi_w)_4 = 0, \]  

(14)

\[ (\partial_t \mathbf{R}, \varphi_{\mathbf{R}})_4 + a_h(\mathbf{U}_g, \mathbf{R}, \varphi_{\mathbf{R}}) + d_h(Q_{z}, \varphi_{Q_z}) = l_h(\varphi_{\mathbf{R}}), \]  

(15)

\[ (Q_z, \lambda_{Q_z})_4 = c_m(\mathbf{R}, \lambda_{Q_z}), \]  

(16)

\[ (\partial_t M, \varphi_{M, z})_4 + a_h(\mathbf{Q}_{m, z}, M, \varphi_{M, z}) = l_h(\varphi_{M, z}), \]  

(17)

\[ (Q_{m, z}, \lambda_{Q_{m, z}})_4 = c_m(M, \lambda_{Q_m}), \]  

(18)

where \( \varphi_h, \varphi_{\mathbf{U}}, \lambda_{Q_{m, z}}, \varphi_w, \lambda_{Q_z}, \varphi_{\mathbf{R}}, \lambda_{Q_m}, \lambda_{Q_m} \) are the test functions. To illustrate the correspondence between various components of the mathematical formulation presented above and the algorithmic changes for different test cases given in Section 3 we have marked the operators in (11)–(18) accordingly. The unmarked operators and equations are common to all test cases.

2.4. Time discretization and slope limiting

The UTBEST3D implementation currently supports discontinuous Galerkin space discretizations with piecewise constant, linear, or quadratic polynomials. At each time step, the solution is advanced in time using a semi-implicit method based on TVB (total variation bounded) Runge–Kutta methods as described in [26]. The order of the time stepping scheme is chosen according to the order of spatial discretization, i.e., for piecewise constant discretization, a forward Euler method is used for temporal integration; for piecewise linear approximations, a second order Runge–Kutta method is used, etc. In the present study, piecewise linear approximations and a second order, two-stage Runge–Kutta were employed.

The vertical velocity component \( W \) is computed diagonally from the values of \( U \) and \( V \) using the discrete continuity Eq. (14). Since operator \( b_h(\cdot) \) is linear with regard to \( W \) this computation is performed as a linear equation system solve in a vertical column of prismatic elements.

In baroclinic runs, operator \( g_d(\cdot) \) includes the baroclinic pressure correction term (6).
Once in a full time step, we solve implicitly for the vertical diffusion terms. A user-specified parameter controls the update frequency of the system matrix for the diffusion operator. The rest of the system, including the horizontal diffusion terms not considered here, is advanced in time explicitly (with local implicit solves for the mass matrix and vertical velocity).

If a transported variable (e.g., temperature) is approximated by linear or quadratic polynomials it must be postprocessed after each stage of the Runge–Kutta method using slope limiters. The purpose of this procedure is to prevent violations of the maximum principle that may produce unphysical extrema. In this work, we employ nodal slope limiters described in [27,28].

2.5. Test problem

As benchmark for our numerical algorithm and implementation we chose a tidal scenario in a part of the Atlantic Ocean adjoining the eastern sea board of the North America, Gulf of Mexico, and the Caribbean between ca. 10° and 45° North. In the east, the domain is bounded by an artificial open boundary with a specified tidal elevation. The geometry of the computational domain provides a sufficiently good representation of the highly irregular coast line and bottom topography to produce accurate tidal simulations (see Fig. 1).

The horizontal finite element mesh consisting of ca. 98k (Fig. 2) triangles of sizes varying between less than 1 km on shallow coastal shelves and 120 km in deep ocean is strongly graded to fit the domain geometry (Fig. 3). The mesh has been optimized towards producing a good resolution of local features on the continental shelf and providing a possibly uniform distribution of the CFL numbers across all mesh elements (the advective CFL number is proportional to the gravity wave propagation speed given as $\sqrt{gh}$).

3. Algorithms

3.1. Main algorithm

As outlined in Section 2.4, UTBEST3D employs a semi-implicit time stepping method resulting in a two-part algorithm for every time step. First, two Runge–Kutta stages are carried out in which body forcings, density forcings, and advection terms in (11), (12), (15) are evaluated and linear systems are built and solved. That way the unknowns are advanced to the next time step explicitly. Each stage also includes the computation of the vertical velocity by Eq. (14). In the presence of vertical eddy viscosity, an implicit computation of the diffusion contributions from (12), (13), and (15)–(18) is added. Due to the nonlinear nature of the involved integrals they cannot be precomputed but must be evaluated numerically in every time step or stage of the Runge Kutta method, respectively. Hence, a number of quadrature rules were defined on the relevant mesh entities for computing numerically the contributions to right hand sides and diffusion matrices. The resulting solution procedure with some explanations is given in Algorithm 1.

3.2. Algorithmic changes for different numerical models

For our scaling study, we compared four different numerical models of increasing physical accuracy and, due to the additional terms, of increasing computational cost. In Section 2.3, certain operators were marked to be relevant only for some of the four
numerical models resulting in some simplifications to Algorithm 1, the latter being the complete algorithm for the most sophisticated setup. In the simplest case, we omitted all vertical diffusion terms, computed with a single layer of prisms, and did not update the free surface. Moreover, the computation was only carried out as a barotropic simulation, hence no variables for temperature and salinity and no computation of density forcings were necessary. In the algorithm, the free surface update, density forcings, and variables $T, S, K, \Psi$ dropped out altogether and so did the entire block labeled ‘implicit’ making the time stepping method a nearly explicit one.

The second setup was enhanced by the algebraic eddy viscosity model (8) and included up to 20 equidistant layers increasing the total number of degrees of freedom to 200k. Additionally, the free surface update was carried out every tenth time step. The resulting scheme is essentially Algorithm 1 without the baroclinic variables $T, S, K, \Psi$.

In the third test, we switched from barotropic to baroclinic simulation, hence adding temperature/salinity transport and the density forcing (6). The turbulence closure utilized the algebraic model from the second test case.

The fourth and most complex setting replaced the algebraic eddy viscosity with the $k$–$c$ closure introducing two 1D unknowns $K, \Psi$

In addition to the mentioned changes visible in the algorithmic structure, more terms were involved in the integration routines of the different mesh entities with every new model. The total number of the degrees of freedom involved in each of the four scenarios, can be easily obtained by considering the fact that the 3D unknowns are approximated using four local degrees of freedom on each prism (piecewise linear DG spaces), whereas the turbulence unknowns need eight degrees of freedom per prism.

### 3.3. Implementation issues for shared memory parallelization

While implementing shared memory parallelization using OpenMP two major challenges had to be faced: (i) identifying and parallelizing suitable parts of the code and (ii) avoiding data races due to parallel updates of degrees of freedom.

---

**Algorithm 1.** UTBEST3D algorithm. See Appendix A for notation.

<table>
<thead>
<tr>
<th>Algorithmic steps</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input global parameters</strong></td>
<td>2D mesh provides $x, y$ node coordinates, node topography (bathymetry), and triangle connectivity</td>
</tr>
<tr>
<td><strong>Input 2D mesh</strong></td>
<td>As columns of non-parallel prisms vertically aligned with 2D mesh triangles</td>
</tr>
<tr>
<td>Create 3D mesh</td>
<td><strong>Done by weighted average of free surface elevation at 2D mesh nodes</strong></td>
</tr>
<tr>
<td>$t \leftarrow t_0$</td>
<td>See Section 2.4</td>
</tr>
<tr>
<td><strong>while</strong> $t &lt; t_{end}$</td>
<td>Includes tidal forcings, sources/sinks, etc</td>
</tr>
<tr>
<td><strong>Update free surface elevation (only every 10th time step)</strong></td>
<td>Part of the baroclinic model, evaluates the equation of state (7)</td>
</tr>
<tr>
<td>$i \leftarrow 0$</td>
<td>Involves solution of Riemann problems on lateral faces (see Appendix A)</td>
</tr>
<tr>
<td><strong>implicit</strong></td>
<td>Linear system solves on each vertical column</td>
</tr>
<tr>
<td><strong>while</strong> $i &lt; n_{stages}$</td>
<td>Evaluation of nonlinear operators</td>
</tr>
<tr>
<td>Perform slope limiting on diffusion variables $K, \Psi$</td>
<td>Evaluation of nonlinear operators</td>
</tr>
<tr>
<td>Build diffusion matrices and right hand sides for diffusion variables :</td>
<td></td>
</tr>
<tr>
<td>Integrate over all horiz. faces $(F^v_i \text{ terms})$ in $a_{i}(\ldots)$</td>
<td>Simple 1D slope limiting</td>
</tr>
<tr>
<td>Integrate over prisms $(E_1 \text{ terms})$ in $a_{i}(\ldots)$</td>
<td>Involves computation of diffusion coefficients</td>
</tr>
<tr>
<td>Compute the next stage of Runge–Kutta method for $H, U, V, T, S$</td>
<td>Linear system solves on each vertical column</td>
</tr>
<tr>
<td>$i \leftarrow i + 1$</td>
<td></td>
</tr>
<tr>
<td><strong>explicit</strong></td>
<td></td>
</tr>
<tr>
<td>Integrate over interior and top horiz. faces $(F_i^{\text{top}} \text{ terms})$ in $a_{i}(\ldots)$</td>
<td></td>
</tr>
<tr>
<td>Integrate over prisms $(E_i \text{ terms})$ in $a_{i}(\ldots)$</td>
<td></td>
</tr>
<tr>
<td>Compute the next stage of Runge–Kutta method for $H, U, V, T, S$</td>
<td></td>
</tr>
<tr>
<td>$t \leftarrow t + dt$</td>
<td></td>
</tr>
</tbody>
</table>

The second setup was enhanced by the algebraic eddy viscosity model (8) and included up to 20 equidistant layers increasing the total number of prisms to 200k. Additionally, the free surface update was carried out every tenth time step. The resulting scheme is essentially Algorithm 1 without the baroclinic variables $T, S, K, \Psi$.

In the third test, we switched from barotropic to baroclinic simulation, hence adding temperature/salinity transport and the density forcing (6). The turbulence closure utilized the algebraic model from the second test case.

The fourth and most complex setting replaced the algebraic eddy viscosity with the $k$–$c$ closure introducing two 1D unknowns $K, \Psi$.

In addition to the mentioned changes visible in the algorithmic structure, more terms were involved in the integration routines of the different mesh entities with every new model. The total number of the degrees of freedom involved in each of the four variables defined on the 2D mesh such as the total water height $H$ are shared by an entire column of prisms. To avoid data races for the degrees of freedom corresponding to 2D unknowns the integration was parallelized with respect to columns of prisms or faces instead of single mesh entities ensuring that only a single thread
was active within a column and able to access the associated 2D degrees of freedom. This approach might potentially increase the risk of load imbalances for test cases where the number of 3D elements within a column differs very much. However, in none of our tests such problems were encountered.

The local nature of the employed discontinuous Galerkin scheme provided for very sparse block matrices that could be efficiently factorized directly. The mass matrix was even block diagonal and could be inverted using a simple LDL-decomposition for which both, decomposition and substitution, were easily parallelizable with respect to the diagonal blocks.

Diffusion matrices were very sparse as well but with an irregular sparsity pattern dependent on the element numbering; however, only dependencies between elements sharing a face within a column of the 3D mesh produced a non-zero block. By choosing an element ordering scheme that numbered elements in a column subsequently we produced diffusion matrices with no couplings between blocks corresponding to elements in different columns. This permitted the parallel LU-decomposition and -substitution methods to be easily parallelized with respect to mesh column-blocks.

4. Performance results

4.1. Dynamic analysis

After implementing the OpenMP parallelization with the necessary modifications described in the previous section a dynamic analysis of the runtimes of all code parts within each time step was performed. Measurements were carried out with four threads using Intel VTune Amplifier XE 2013 with the help of Intel’s task annotation API on an Intel Core i7-2720QM processor (2.2 GHz, 4 cores, HT disabled) with 8 GB RAM.

In the first test case, a time step only consisted of two stages of the Runge–Kutta method. Here, the integration of advection terms to build up the right hand sides dominated the overall execution time – the linear solvers for the computation of the vertical velocity or mass matrix inversion took each less than 4% of the runtime. The most time consuming part of the code was the integration over vertical faces and prisms. A graphical illustration of the runtime portions within a time step is given in Fig. 4. Within the integration routines, many operations of the form $y = ax + y$ (axpy) were executed on the local degrees of freedom of the current element.

Fig. 3. Zoom-in of the 2D mesh. Gulf of Mexico (left), detail at the Louisiana coast (right).

Fig. 4. Time flow of a time step in Algorithm 1 for each of the four test cases (measured using Intel VTunes on four threads).
was confirmed by a bottom-up analysis of a time step showing that the axpy-operations were the routine with most accumulated runtime but nevertheless not with a major share of the runtime, indeed the axpy-operations were the routine with most accumulated runtime but nevertheless not with a major share of the runtime. Some time was also spent in the HLLC Riemann solver. Table 1 shows execution times of the five most time consuming routines in a bottom-up analysis of a time step showing that the advection (both RK-stages) and the diffusion parts of the system accounted for approximately half of the execution time. Within the latter, the integration of the diffusion variables – all combined in the block ‘linear solver’ in Fig. 4. A bottom-up analysis (see Table 1) demonstrated that large parts of the execution time were now spent on linear algebra routines, such as matrix–matrix-products and LDL-substitution – all parts of the computation of the diffusion terms. But, due to the complexity of the algorithm, there was no single compute kernel dominating the execution time. Although more time was spent on OpenMP synchronizations in this test case the overall portion of overhead was even lower than in the first case. Some slight workload imbalance due to free surface updates

Table 1
The five most time consuming routines in a bottom-up analysis of a time step (measured using Intel VTunes on four threads). Presented are the accumulated runtimes T for each routine as well as the portion of the total runtime T.T. of a time step.

| Test case 1 (T = 61.1 ms) | axpy on local DOFs | 11.0 | 18 |
| | RK-stage: integrate horiz. face | 4.00 | 6.6 |
| | Access to local quadrature data | 3.00 | 4.9 |
| | OpenMP synchronization | 3.00 | 4.9 |
| Test case 2 (T = 311 ms) | Diffusion: integrate prism | 22.0 | 7.1 |
| | Update of quadrature data | 20.0 | 6.4 |
| | axpy on local DOFs | 20.0 | 6.4 |
| | General matrix multipl. (GEMM) | 14.0 | 4.5 |
| | LDLSubstitution on diff. matrix | 29.0 | 5.4 |
| | OpenMP synchronization | 10.0 | 3.2 |
| Test case 3 (T = 541 ms) | Diffusion: integrate prism | 46.1 | 8.5 |
| | axpy on local DOFs | 50.1 | 6.7 |
| | General matrix multipl. (GEMM) | 44.1 | 5.9 |
| | LDLSubstitution on diff. matrix | 29.0 | 5.4 |
| | OpenMP synchronization | 34.1 | 6.3 |
| Test case 4 (T = 752 ms) | Diffusion: integrate prism | 59.1 | 7.9 |
| | axpy on local DOFs | 50.1 | 6.7 |
| | General matrix multipl. (GEMM) | 44.1 | 5.9 |
| | Dot-product of local DOFs | 31.0 | 4.1 |
| | LDLSubstitution on diff. matrix | 28.0 | 3.7 |
| | OpenMP synchronization | 53.1 | 7.1 |

Table 2
Overview of the used systems with total number of physically available cores Nc and virtual hardware threads Nt.

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>System</th>
<th>Nc</th>
<th>Nt</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC</td>
<td>1x Intel Xeon Phi 5110P, 8 GB RAM, 60 cores, 4-way HT, 1.053 GHz</td>
<td>60</td>
<td>240</td>
<td>icpe</td>
</tr>
<tr>
<td>WEX</td>
<td>4x Intel Xeon E7-4870, 128 GB RAM, 80 cores, 2-way HT, 2.4 GHz</td>
<td>40</td>
<td>80</td>
<td>icpe</td>
</tr>
<tr>
<td>SEP</td>
<td>2x Intel Xeon E5-2690, 64 GB RAM, 16 cores, 2-way HT, 2.9 GHz</td>
<td>16</td>
<td>32</td>
<td>icpe</td>
</tr>
<tr>
<td>PWR6</td>
<td>16x IBM Power6, 4 GB RAM (DKRZ's Blizard)</td>
<td>64</td>
<td>128</td>
<td>xlc++_r</td>
</tr>
<tr>
<td>PWA2</td>
<td>IBM PowerPC A2, 16 GB RAM (JSC's JUQUEEN)</td>
<td>64</td>
<td>128</td>
<td>bgxlc++_r</td>
</tr>
</tbody>
</table>

Table 3
Runtimes and speed-up for the four test cases on all tested architectures with varying number of threads Nt. The given speed-up numbers were calculated by \( s = T(1)/T(N_t) \).

<table>
<thead>
<tr>
<th>Nt</th>
<th>Test case 1</th>
<th>Test case 2</th>
<th>Test case 3</th>
<th>Test case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIC</td>
<td>6.28e3</td>
<td>3.61e3</td>
<td>4.51e4</td>
<td>6.20e4</td>
</tr>
<tr>
<td>WEX</td>
<td>3.00e2</td>
<td>1.20e3</td>
<td>2.18e3</td>
<td>3.02e3</td>
</tr>
<tr>
<td>SEP</td>
<td>3.47e2</td>
<td>1.87e2</td>
<td>1.75e2</td>
<td>2.18e3</td>
</tr>
<tr>
<td>PWR6</td>
<td>1.28e3</td>
<td>3.03e2</td>
<td>3.12e2</td>
<td>3.09e3</td>
</tr>
<tr>
<td>PWA2</td>
<td>1.31e2</td>
<td>3.00e2</td>
<td>3.12e2</td>
<td>3.16e3</td>
</tr>
</tbody>
</table>

may be affecting the results of this test case. Since the free surface update was not executed in every time step the average OpenMP overhead was even less pronounced than the one given in Table 1.

The third test case turned on the baroclinic simulation whereas all other parameters remained the same as in the previous model. This introduced a density forcing term along with temperature and salinity as two additional unknowns. Consequently, the integration routines and solvers for linear systems took approximately twice as long as in the previous test case – due to the fact that now four instead of two primary unknowns were considered. However, the relative portions of the execution times were comparable to the previous model as shown in Fig. 4. A similar outcome was visible in the bottom-up analysis given in Table 1. Noteworthy is the slightly increased OpenMP overhead due to the additional density forcing computation which was also parallelized with respect to columns of prisms and, consequently, suffered from a slight imbalance in the workload.

In the last model, the algebraic turbulence scheme was replaced by the two-equation $k$–$\varepsilon$ model. This change affected the computation of diffusion variables whereas the explicit part of the time step remained the same as in the previous case. The new turbulence closure became noticeable mainly in the more involved integrations over prisms and horizontal faces during the implicit part of the time step – the differences are clearly visible in Fig. 4. The bottom-up analysis results in Table 1 confirmed that the integration over prisms was the most time consuming part of the time step in this case.

In summary, two points can be seen from this dynamic analysis: (i) there is not a small compute kernel that is responsible for a major share of the execution time but a number of very different integration routines which are getting more expensive to compute as the model gets more complicated and (ii) OpenMP overhead is reasonably small. This is a mandatory requirement for good parallel performance and was one of the primary goals during the implementation of the OpenMP parallelization. However, the first point suggests that our runtime is not limited by a single routine that could be locally optimized to improve the overall performance but that the complexity of the algorithm requires a many different operations which appear to be balanced in our implementation.

4.2. Parallel performance

Parallel performance was tested on a number of shared memory architectures commonly found in current compute clusters and high performance computers. The specific test environments are listed in Table 2, along with the used compiler and its version number. All measurements were done using appropriate pinning, i.e., OpenMP threads were placed as close to each other as possible and without employing SMT/HT unless the chosen number of threads was higher than the number of physically available cores. The resulting runtimes for all four test cases are listed in Table 3, strong scaling results are visualized in Fig. 5, and parallel efficiency plots are in Fig. 6.

When comparing runtimes for equal numbers of threads, significant differences between the different architectures are visible. Best runtimes were obtained on Intel Xeon processors with Sandy Bridge EP being the fastest, presumably due to the higher clock rate. Similar times were obtained on IBM’s Power6, but computations took significantly longer on IBM’s PowerPC A2 and Intel’s MIC. Considering the substantial differences in clock rate between these processors we find a strong dependency of the overall runtime on the clock rate of the processor.
Regarding the parallel efficiency we found reasonable scaling behavior for all test cases on the MIC and Power processors – as long as the number of threads did not exceed the number of physically available cores. With 8 or 10 threads the parallel efficiency was between 76% and 97% and even higher for lower thread counts. Highest parallel efficiency values throughout all test cases were achieved on Intel MIC and IBM PowerPC A2 which, however, had the longest computation times. Intel’s Xeon processors produced lower speed-up numbers but still achieved 68–74% of parallel efficiency with 8 or 10 threads. The deviation from linear scalability can be explained with the OpenMP synchronization overhead measured in Section 4.1 which turned out to be small but not negligible and potentially became more severe with a higher numbers of threads.

Noteworthy is the fact that, within the region of physically available cores, best speed-up results were obtained with the simplest test case 1 on all architectures; the three multi-layer scenarios achieved lower but among them similar parallel efficiency. This is also in agreement with the findings of Section 4.1, where a higher OpenMP overhead was measured for the latter test cases. Furthermore, while all other architectures showed none or only little speedup when using SMT/HT, IBM’s PowerPC A2 processors still produced a factor of 1.6–1.7 for 2-way and 2.2–2.4 for 4-way SMT. Intel’s MIC, the second best performer in the SMT/HT region, only achieved a factor of 1.3–1.4 for 2-way and none additional speedup for 4-way HT.

Considering the runtimes per machine, Intel’s Westmere EX processors produced the fastest times to solution in all cases owing to the greater number of available cores compared to the Sandy Bridge EP processors, even though the latter had a better per-core performance. The same is true for the two slowest architectures, IBM’s PowerPC A2 and Intel’s MIC, where the former showed a better per-core performance but the much higher number of available threads allowed for a faster time to solution of the latter.

5. Conclusions and outlook

The results of Section 4.2 showed very good parallel efficiency of our code on a real-world application for a wide range of current shared-memory parallel platforms and moderate thread counts. Further speed-up was achieved with very high thread numbers but at the cost of reduced parallel efficiency. When integrating the OpenMP parallelization into the already existing MPI-parallel code to support hybrid parallelization, this suggests a choice of up to 10 threads per MPI-rank to give the best performance. The integration and evaluation of both parallelization strategies will be our next step from which an additional performance gain on recent HPC clusters can be expected.

The runtimes achieved with Intel’s new Xeon Phi accelerator cards were not satisfactory. Possibly due to the low clock rate it produced the longest runtimes in our study and was only able to catch up with IBM’s PowerPC A2 when exploiting the greater number of available threads. In none of the four test cases it was able to challenge Intel’s Xeon processors from the Westmere EX or Sandy Bridge EP family. One of the most serious bottlenecks on current platforms is an insufficient vectorization of the whole code. Here, we have to explore in more detail when the compiler can vectorize our code automatically and where it becomes necessary to do it manually, e.g., by adding explicit platform-dependent intrinsics to our implementation.
Additionally, as part of this study, we have identified the performance critical parts of our ocean simulation. It will be interesting to investigate which of them benefit most from using additional accelerators such as GPUs. Another possibility to decrease the overall runtime would be to automatically generate specialized code for evaluating the different kinds of integrals.

**Acknowledgments**

This work was supported in part by the German Research Foundation (DFG) Grant Al 117/1. We acknowledge the Regional Computing Center Erlangen (RRZE), Jülich Supercomputing Center (JSC) and the German Climate Computing Center (DKRZ) for providing computational resources.

**Appendix A. Construction of a semi-discrete LDG-approximation**

Prior to presenting the semi-discrete formulation of the numerical scheme we introduce some additional notation. For a bounded domain \(\Omega \subset \mathbb{R}^d, d \in \{1, 2, 3\} \), let \(\mathcal{T} = \{\mathcal{T}_h(\Omega), \Delta x > 0\} \) be a family of non-overlapping, regular but not necessarily conforming partitions of \(\Omega\) into elements \(e\). The space \(V_d\) on \(\mathcal{T}_h\) is defined as follows:

\[
V_d = \mathcal{L}^2(\Omega) \cap \{v: v \in H^1(e), \forall e \in \mathcal{T}_h\}.
\]

Let \(\Gamma: \mathcal{H}^1(e) \to \mathcal{H}^1(\partial e)\) be the trace operator on \(e\), and let \(\gamma_{id} = \partial e \cap \partial \Omega \neq \emptyset\) for some \(e, f \in \mathcal{T}_h\) such that \(e\) has lower index in \(\mathcal{T}_h\) than \(f\). For \(u \in V_d\), we define the jump operator on \(\gamma_{id}\) as follows:

\[
[u] = (\Gamma u)|_{\gamma_{id}} - (\Gamma u)|_{\gamma_{id}}.
\]

We also fix the direction of a unit normal \(n = (n_x, n_y, n_z)^T\) on \(\gamma_{id}\) so that points away from the element with lower index and set it to the exterior unit normal on \(\partial \Omega\). In this paper, we will use standard notation \((\cdot, \cdot), <\cdot, \cdot>\) for the \(\mathcal{L}^2\) inner products on elements \(e \subset \mathbb{R}^d\) and \(d - 1\) dimensional surfaces \(\gamma\), correspondingly. For \(d = 1\), operator \((\cdot, \cdot)\) denotes a function evaluation at a boundary point.

Given a mesh of \(\Omega(t)\), we define the following sets of elements and faces:

- \(E_1\) − set of vertical 1D segments in \(\Omega(t)\) (one- and two-equation turbulence models only);
- \(E_2\) − set of 2D elements in \(\Omega(t)\);
- \(E_3\) − set of 3D elements in \(\Omega(t)\);
- \(F_{i1}, F_{i2}\) − sets of interior/interior segment endpoints (one- and two-equation turbulence models only);
- \(F_{22}, F_{23}\) − sets of interior/interior 2D edges;
- \(F_{31}, F_{32}\) − set of interior/exterior 3D faces;
- \(F_{i1}^{int}, F_{i2}^{int}\) − set of lateral (side) interior/exterior 3D faces;

We approximate \(c(t, \cdot) = (h, u_{bg}, q_{bg}, w, r, q_t, m_{bg})\), a solution to the mixed version of (1)–(5), with \(c(t, \cdot) = (H, U_{bg}, Q_{bg}, W, R, Q_t, M, m_{bg})\) in \(H^1 \subset U^1 \subset V^2 \subset Q_{bg}^2 \subset Q_{bg} \subset V_3 \subset V_3, W_4 \subset V_3, R^4 \subset V_3, Q_t^4 \subset V_3, M^4 \subset V_1\) are approximation spaces defined on \(T\). This quite general setting has one important constraint in the case of our prismatic 2D/3D meshes, namely that the partition of the 3D domain \(\Omega(t)\) must consist of prismatic elements stacked on top of each other so that each such vertical column exactly corresponds to a 2D element in the partition of \(\Omega(t)\).

The semi-discrete LDG formulation for the temperature/salinity transport Eqs. (9) and (10) is then given by

\[
\sum_{e \in F_{i1}} (\partial \varphi_e) \cdot n_e + \sum_{e \in F_{i2}} \bigl( \quad \widehat{K}_{r,m}(U_e, R_e, U_e, R_e), \varphi_e \quad \bigr) + \sum_{e \in F_{i1}} (U \cdot \nabla) \varphi_e - \sum_{e \in F_{i2}} (Q_{r,m} \cdot \nabla) \varphi_e - \sum_{e \in F_{i1}} (Q_{r,m} \cdot \nabla) \varphi_e - \sum_{e \in F_{i2}} (Q_{r,m} \cdot \nabla) \varphi_e - \sum_{e \in F_{i1}} (Q_{r,m} \cdot \nabla) \varphi_e = \sum_{e \in F_{i2}} (F_e, \varphi_e) + \sum_{e \in F_{i1}} (Q_{r,m} \cdot \nabla) \varphi_e - \sum_{e \in F_{i2}} (Q_{r,m} \cdot \nabla) \varphi_e - \sum_{e \in F_{i1}} (Q_{r,m} \cdot \nabla) \varphi_e - \sum_{e \in F_{i2}} (Q_{r,m} \cdot \nabla) \varphi_e.
\]

where \(\varphi_e \in R^4, \lambda_e \in Q_t^4\) are the test functions; \(\widehat{K}_{r,m}(C, C)\) is an approximation to the normal advective flux (i.e., solution to the Riemann problem). Since we are not considering the horizontal diffusion in this work the diffusive fluxes denoted by \(\lambda\) are equal to zero on vertical faces. On the interior horizontal faces, they are computed using an anti-symmetric stencil in which the values of primary unknowns in Eq. (22) are taken from the element below \((\downarrow)\) whereas the values of the auxiliary (flux) variables in Eq. (21) are evaluated on the element above \((\uparrow)\). This scheme reduces the matrix bandwidth in the semi-implicit formulation of the vertical eddy viscosity/diffusivity – we only need the blocks corresponding to neighbors sharing a horizontal face.

Introducing a trilinear form for the advective terms and bilinear forms for the diffusive/flux operators:

\[
a_{e}(U, R, \varphi_e) = \sum_{e \in F_{i2}} \bigl( \quad \widehat{K}_{r,m}(U_e, R_e, U_e, R_e), \varphi_e \quad \bigr) + \sum_{e \in F_{i1}} (U \cdot \nabla) \varphi_e - \sum_{e \in F_{i2}} (Q_{r,m} \cdot \nabla) \varphi_e
\]

\[
d_{e}(Q_{r,m}, \varphi_e) = \sum_{e \in F_{i1}} (Q_{r,m} \cdot \nabla) \varphi_e = \sum_{e \in F_{i2}} (Q_{r,m} \cdot \nabla) \varphi_e
\]

the scalar product on \(\Omega \subset \mathbb{R}^3\), and a linear form for the right hand side

\[
(A, B)_1 = \sum_{e \in F_{i2}} (A, B)_e, \quad l_e(\varphi_e) = \sum_{e \in F_{i1}} (F_e, \varphi_e)
\]

we can re-write Eqs. (21) and (22) in a compact form as in (15) and (16) with similar expressions holding for the rest of the system.

A derivation of the semi-discrete formulation for the rest of the system can be found in [19].

**References**


