A task parallel implementation of an RBF-generated finite difference method for the shallow water equations on the sphere[☆]

Martin Tillenius[a], Elisabeth Larsson[a,*], Erik Lehto[b], Natasha Flyer[c]

[a] *Dept. of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden*
[b] *Dept. of Mathematics, KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden*
[c] *National Center for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307-3000, USA*

Abstract

Radial basis function-generated finite difference (RBF-FD) methods have recently been proposed as very interesting for global scale geophysical simulations, and have been shown to outperform established pseudo-spectral and discontinuous Galerkin methods for shallow water test problems. In order to be competitive for very large scale simulations, the implementation of the RBF-FD methods needs to be efficient and adapted for modern multicore based computer architectures. The main computational operations in the method consist of unstructured sparse matrix-vector multiplications, which are in general not well suited for multicore-based computers. In this work, the method is implemented for clusters of multicore computers using a task-based parallel programming model. Performance experiments showed that our implementation achieves 71% of theoretical speedup within one computational node, and 90–100% of linear speedup between nodes. A speedup of 178 times compared with the original MATLAB implementation was achieved for a global shallow water problem with a 30km resolution.

Keywords: shallow water, scattered node, task parallel, distributed memory, multicore, radial basis function, RBF-FD
2000 MSC: 65Y05, 65Y10, 65M99

1. Introduction

In computational geoscience, models are often global in the sense that the considered domain is the atmosphere around the whole earth, the entire surface of the earth, or

[☆]The work of M. Tillenius and E. Larsson was supported by the Swedish Research Council through the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures. The work of E. Lehto was supported by The Dahlquist Research Fellowship. NCAR is sponsored by the National Science Foundation (NSF). N. Flyer acknowledges support of NSF grant DMS-0934317.

[*]Corresponding author
Email addresses: martin.tillenius@it.uu.se (Martin Tillenius), elisabeth.larsson@it.uu.se (Elisabeth Larsson), elehto@kth.se (Erik Lehto), flyer@ucar.edu (Natasha Flyer)

the mantle of the earth. Furthermore, there may also be a variety of coupled physical processes as in global climate simulations, where different models for the atmosphere, the land surface, the oceans, the large ice sheets, and others all contribute to the global results. These simulations are both time consuming and produce large volumes of data. Hence, they cannot be performed without the use of parallel processing. Some of the largest computer systems in the world are dedicated to weather and climate simulations.

Existing community models such as EC-Earth connected with the European Center for Mid-Range Weather Forecasts (ECMWF), and the Community Earth System Model (CESM) connected with the National Center for Atmospheric Research (NCAR), USA, use grid based numerical methods for the simulations. This makes it costly to resolve local solution features, since typically a larger region needs to be resolved in order to preserve grid integrity. Reported resolutions in current simulations are at a level of 30–100 km, but processes that are of importance for the global simulation result may occur on even finer scales.

An interesting alternative to using grid based methods are meshfree methods based on radial basis function (RBF) approximation. These methods work with scattered nodes and easily allow for local refinement of the numerical solutions. Global RBF approximation methods have been shown to be competitive compared with other commonly used numerical approaches in computational geosciences in a series of papers [16, 17, 43, 14]. However, for large node sets, it becomes expensive in terms of computational time and storage to work with the dense linear systems that arise from the global approximations.

By replacing the global RBF approximations with local RBF approximations, we can reduce the computational cost while retaining the flexibility with respect to geometry and the ability to work with scattered nodes. RBF-generated finite differences (RBF-FD) are similar in nature to finite difference methods, but with stencils that are built on scattered nodes instead of nodes in a grid, and an underlying interpolation that is RBF based instead of polynomial. In the paper [15], RBF-FD methods were compared with, and outperformed state-of-the-art methods for fluid flow test cases over the sphere. These comparisons where done with a sequential implementation. However, in order to demonstrate the viability of the RBF-FD method for large scale computational geoscience, we also need to show that it scales to large problem sizes and can be efficiently parallelized for large computer systems.

The RBF-FD method is relatively new, with an early mentioning in the conference paper [38] from 2000, and further development and analysis for example in [32, 42, 10, 44, 3, 9, 8, 5, 4]. There is still not much research done on parallelization of these methods. However, a parallel implementation of an RBF-FD method for the shallow water equations on the sphere for multiple CPUs and GPUs can be found in [6]. Each CPU runs an MPI process and handles the necessary communication. Local computations are off-loaded to the GPU attached to the CPU. Speedups of up to 7 compared with execution on one CPU are reported. An OpenMP parallelization of an RBF-FD method for a coupled thermal-fluid flow problem can be found in [24]. The stencils and problem sizes are small, and only two cores are used, but the results are good. The parallel method is further developed in [25] with experiments for up to 16 cores. An MPI implementation of a similar application problem and similar RBF-FD approach is presented in [11], where experiments are performed on a 36-node cluster. However, no performance results are reported. A preliminary version of the shared memory part of this paper can be found in [37].

In the MPI/OpenCL and MPI approaches used for parallelization in [6, 11], the programmer is responsible for explicitly expressing all data transfers and decide where in the system each computation takes place. With the OpenMP approach of [24, 25] the programmer works with pragma directives to the compiler inserted into the sequential code. The latter is significantly easier to implement, however the performance may be suboptimal due to the global synchronization points introduced by the fork-join style programming model.

Dependency-aware task-based parallel programming models are emerging as one of the most promising approaches to achieve high performance in scientific applications at a reasonable programming effort. Here, the programmer writes a sequential code in terms of tasks, while a run-time system provided by a programming framework is responsible for scheduling and executing tasks in parallel. There are several widely spread frameworks of this type such as OmpSs [13] and StarPU [1]. In this work we use the SuperGlue framework [35, 33, 34], developed by the first author. Its performance with respect to other frameworks was tested in [34], and SuperGlue was shown to be highly competitive.

Typical benchmark problems for task parallel programming are dense linear algebra operations such as the Cholesky factorization. These are amenable to parallelization on multicore architectures because they are compute-bound. In this work, we instead target a complete RBF-FD solver for the shallow water equations on the sphere, which generates sparse unstructured matrices. The main operations of the solver are sparse matrix-vector multiplications which are bandwidth-bound. With a straightforward implementation they are expected to scale badly on multicores.

The objectives of this paper are to establish that RBF-FD methods can be efficiently implemented in parallel for modern multicore based computer architectures. Furthermore, we want to demonstrate that task-based parallel programming is a tool that provides high programmer productivity and significantly facilitates the writing of scientific software without sacrificing absolute performance. How to implement adaptive node refinement will however not be pursued in this paper.

The paper is organized as follows: In Section 2 we define the shallow water problems that we use as benchmarks. Then in Section 3, we discuss the RBF-FD method. Section 4 contains the discrete formulation of the problems. The task-based programming model is explained in Section 5, and the parallel implementation of the solver is described in Section 6. Finally, Section 7 provides performance experiments and solution examples, and the conclusions are given in Section 8.

2. The shallow water equations on the sphere

For a new method to be adopted in any field it must first be shown to perform well for certain benchmark problems. In computational geosciences, and especially for global atmospheric simulations, the shallow water equations (SWE) constitute an important problem class for numerical testing. The SWE are a set of non-linear partial differential equations (PDE) which capture the main features of the horizontal dynamics of atmospheric flow around the earth.

In this paper, we will use two of the benchmarks that were used in [15] to evaluate the parallel performance of RBF-FD methods for the SWE. The first test case that we use is called “flow over an isolated mountain”. The initial condition is perfectly laminar flow

in the easterly direction, but due to the presence of a very large cone shaped mountain, the flow develops a wave pattern with time. A full description can be found in [39]. The second test case, described in [23] represents the evolution of a highly non-linear wave. The solution contains high frequency components and sharp gradients. Figure 1 shows the solutions of the two test cases at different times.

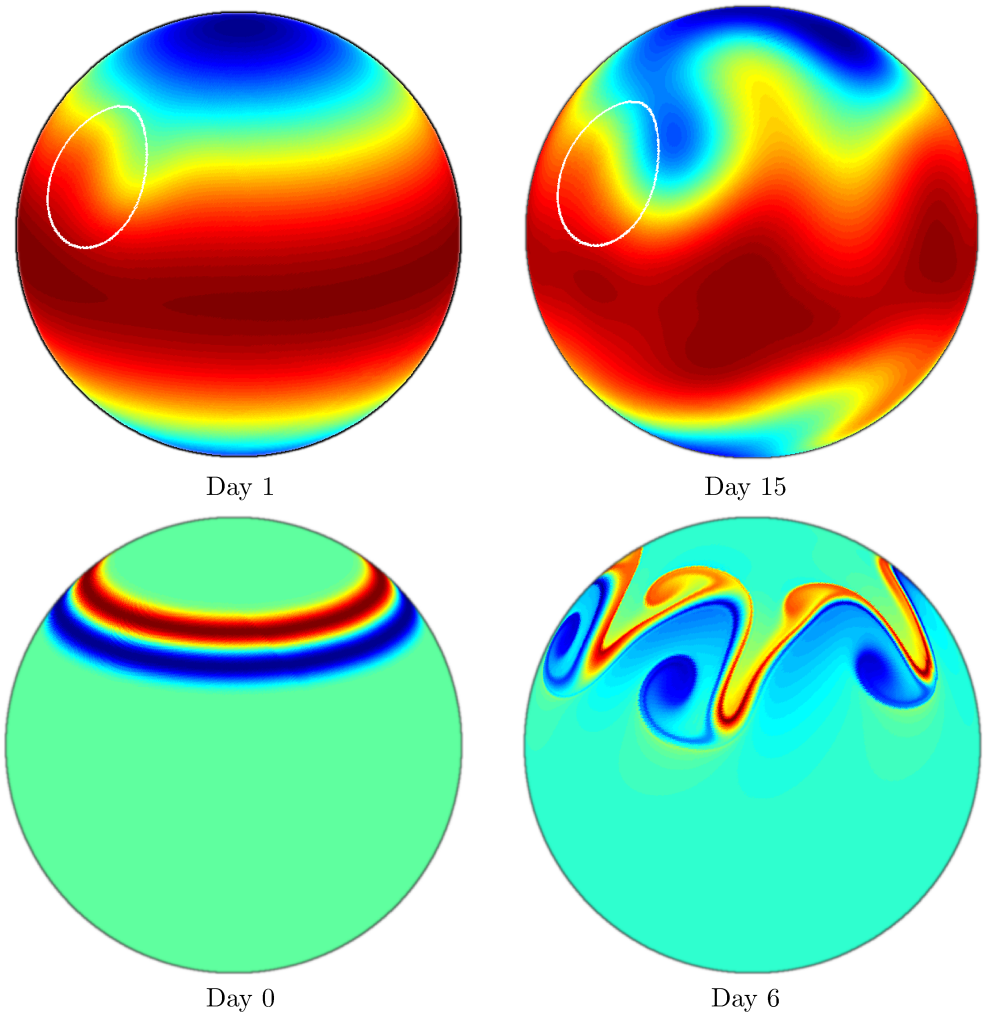


Figure 1: Solution to the shallow water test case "flow over an isolated mountain" (top) and the evolution of a highly non-linear wave (bottom) at different times. In the first case, the geopotential height is shown, and in the second case, the relative vorticity is shown. The white circle shows the location of the cone shaped mountain. In both cases, a perturbation of an initially laminar flow develops over time to affect the flow on a global scale.

In combination with grid based methods on the sphere, the SWE are often formulated in spherical coordinates. However, this introduces unphysical singularities at the poles. RBF based methods are oblivious to the coordinate system used and its orientation.

Hence, we choose to work with the Cartesian formulation of the SWE, which is singularity free and given by

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - f(\boldsymbol{x}\times\boldsymbol{u}) - g\nabla h,$$
$$\frac{\partial h}{\partial t} = -\nabla\cdot(h\boldsymbol{u}),$$

where $\boldsymbol{u} = (u, v, w)$ is the wind field, $\boldsymbol{x} = (x, y, z)$ is the location, $f = 2\Omega z$ is the Coriolis force, where Ω is the angular velocity of the earth, h is the geopotential height, and g is the gravity.

Using the Cartesian formulation, we instead need to project the operators onto the curved surface of the earth. This approach was introduced in [17], and is applied also in [15]. For practical purposes, we first scale the problem to the unit sphere. The projection operator onto the unit sphere is defined as $\boldsymbol{P} = \boldsymbol{I} - \boldsymbol{x}\boldsymbol{x}^T$. The projected form of the SWE then becomes

$$\frac{\partial \boldsymbol{u}}{\partial t} = -\boldsymbol{P}\left[(\boldsymbol{u}\cdot\boldsymbol{P}\nabla)\boldsymbol{u} + f(\boldsymbol{x}\times\boldsymbol{u}) + g\boldsymbol{P}\nabla h\right], \tag{1}$$
$$\frac{\partial h}{\partial t} = -\boldsymbol{P}\nabla\cdot(h\boldsymbol{u}). \tag{2}$$

3. The RBF-FD method and its properties

Here we will discuss the RBF-FD method. We start with the general framework, and then discuss different properties that are important for the approximations. For each property, we will in particular discuss aspects that may influence the ability of the method to scale to large problem sizes.

We start by defining the RBFs that we will use in the underlying approximations. A radial basis function $\phi(r)$ is a function in d dimensions whose value depends only on the distance $r = \|\boldsymbol{x} - \boldsymbol{c}\|$ from its center point $\boldsymbol{c}$. Many commonly used RBFs are also equipped with a so called shape parameter that is applied to the argument to influence the flatness of the RBF. Figure 2 shows the effect of scaling on a Gaussian RBF, $\phi(r) = e^{-\varepsilon^2 r^2}$.

Assume there are $N \gg n$ scattered nodes on the surface of the sphere (or in any other geometry of choice). A differential operator $\mathcal{D}$ is approximated at the location $\boldsymbol{x}_c$ by using a weighted combination of the function values f_k, $k = 1, \ldots, n$ at the n nearest neighboring nodes. That is,

$$\mathcal{D}f(\boldsymbol{x}_c) \approx \sum_{k=1}^{n} w_k f_k, \tag{3}$$

where the weights w_k are determined by requiring the approximation to be exact when the solution can be exactly represented by the basis underlying the approximation, which here consists of radial basis functions centered at the scattered nodes. Assuming that $f(\boldsymbol{x}) = \sum_{k=1}^{n} \lambda_k \phi(\|\boldsymbol{x} - \boldsymbol{x}_k\|) \equiv \sum_{k=1}^{n} \lambda_k \phi_k(\boldsymbol{x})$ yields the following linear system of

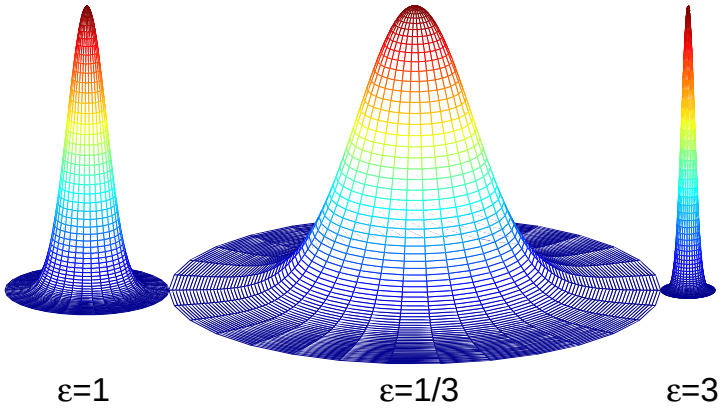


Figure 2: The shape of a Gaussian RBF for different values of the shape parameter ε.

equations for the weights

$$\begin{bmatrix} \phi_1(\boldsymbol{x}_1) & \phi_1(\boldsymbol{x}_2) & \cdots & \phi_1(\boldsymbol{x}_n) \\ \phi_2(\boldsymbol{x}_1) & \phi_2(\boldsymbol{x}_2) & \cdots & \phi_2(\boldsymbol{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_n(\boldsymbol{x}_1) & \phi_n(\boldsymbol{x}_2) & \cdots & \phi_n(\boldsymbol{x}_n) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \mathcal{D}\phi_1(\boldsymbol{x}_c) \\ \mathcal{D}\phi_2(\boldsymbol{x}_c) \\ \vdots \\ \mathcal{D}\phi_n(\boldsymbol{x}_c) \end{bmatrix}. \tag{4}$$

Figure 3 provides a graphical representation of a differentiation stencil on the sphere. The exemplified stencil is quite large with $n = 75$. Because the sphere is a periodic domain (no boundaries) we can use large stencils without running in to the particular problems that are associated with boundaries and high-order stencil approximations. It has been shown that RBF approximations in the limit are polynomial [12, 27, 31, 29]. Therefore, at least for small values of ε, the approximation error for a stencil for a first-order derivative with n points behaves as $\mathcal{O}(h^k)$, where

$$(k+1)^2 \leq n < (k+2)^2, \tag{5}$$

and $(k+1)^2$ is the number of degrees of freedom in a polynomial of degree k restricted to the sphere.

To assemble a global differentiation matrix that approximates the operator $\mathcal{D}$ at all node points given the function values at these points, we let $\boldsymbol{x}_c$ traverse all node points and then we enter the generated weights into the corresponding row in the global sparse differentiation matrix. A typical sparsity pattern for a differential operator approximation on the sphere is shown in the right part of Figure 3.

3.1. Stability for time-dependent PDEs

The most critical issue in scaling the RBF-FD method to large problem sizes is stability, in particular for hyperbolic PDEs (lacking a diffusion term) like the SWE. The discretization of the spatial operator may, as illustrated in [15], exhibit eigenvalues with a positive real part, leading to instability. A remedy for this was given in [19] in the form of hyperviscosity operators that can be added to the discrete operator in order to

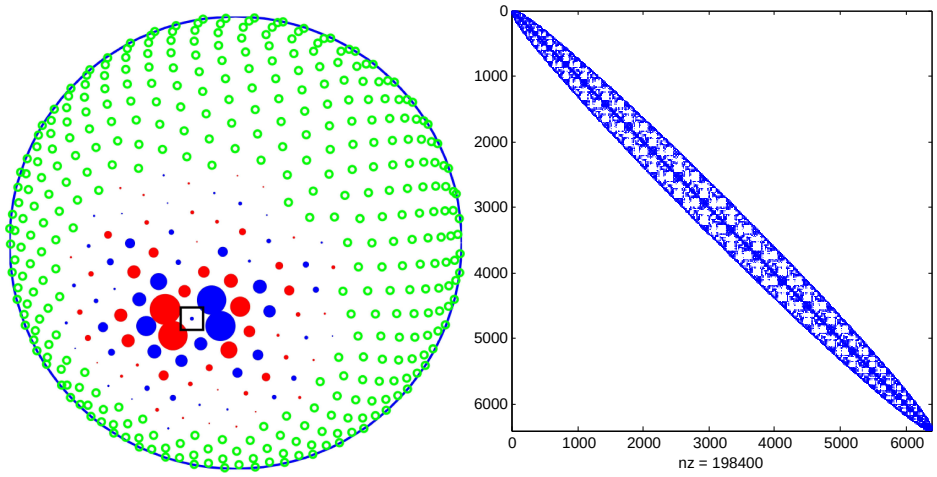


Figure 3: Left: This is an illustration of a 75 node stencil on the sphere. The differential operator is evaluated at the node marked with a square. The size of the markers indicates the magnitude of the stencil weights and the color indicates the signs. The nodes marked with green rings (further away) are not included in the stencil. Right: The structure of a global differentiation matrix for $N = 6400$ nodes. A bandwidth minimization algorithm has been applied to the matrix.

suppress the unstable modes. The hyperviscosity operator is of the type Δ^m, where m needs to be higher for larger stencil sizes (corresponding to higher order methods).

Using Gaussian RBFs, the hyperviscosity operators can be efficiently implemented [19], see also [28] for the flat limit version. Therefore, we will use Gaussian RBFs everywhere in this paper.

When we scale to large problem sizes, we can only add very small amounts of hyperviscosity, otherwise the solution quality is affected. Furthermore, if the stencils are large, we need high order hyperviscosity, which also makes the problem more sensitive.

3.2. Stencil sizes

The stencil size has already been discussed above and how the order of accuracy k relates to the stencil size n is given by equation (5). A larger stencil leads to higher accuracy, but more stability problems, and as will be discussed in Section 7 we were not able to use large stencil sizes for the largest problem size and the "evolution of a highly non-linear wave" test case.

The first test case, flow over an isolated mountain, has a non-smooth forcing term representing the conical mountain. Therefore, the accuracy is not improved by increasing the stencil size beyond a certain point. For this problem we will use the size $n = 31$ chosen in [15] for all experiments.

Following equation 5, it might seem reasonable to choose the stencil size equal to $(k + 1)^2$. However, it has been observed in experiments that certain stencil sizes have better stability properties than others. These special numbers are $n = 17$, 31, 50, and 101. We do not know why these sizes are better, but one theory is that they have some symmetry properties, depending on the layout of the nodes.

3.3. Node sets on the sphere

There are different approaches to create quasi uniform node distributions on the sphere. Minimum energy (ME) nodes [41] are of high quality, but only relatively small node sets can be generated. Maximum determinant (MD) nodes [40, 41] are also high quality and somewhat larger node sets can be generated. These are used for the smaller problem sizes in our numerical experiments. For the larger problem sizes, we are relying on icosahedral (ICO) node sets [2] that can be easily be generated for large problem sizes. However, the quality of the icosahedral node sets is a little bit worse. The number of close neighbors to a node point depends somewhat on the location in the original icosahedron. This does not matter very much for small problem sizes, but for the large problem sizes, it is one of the factors that influence stability. In fact, we have some stability problems for the ICO nodes that led us to also include a few node sets generated by DistMesh [30] (DM), which gives smoother results.

3.4. Scaling of stencils and RBFs

Stencil weights are computed by solving a linear system $Aw = b$. If we fix the node configuration, i.e, the number of stencil nodes and their relative location up to a scaling and translation, then we can focus on the effects of the scaling of the coordinates and the scaling of the RBFs. Define the stencil size h as the largest distance between two nodes in the stencil. We use the notation $A = A(h, \varepsilon)$ and $b = b(h, \varepsilon)$ in order to facilitate the discussion of the scaling issues. Due to the interchangeability of the scaling of the distances r and the shape parameter ε in the argument of the RBFs, we have the following relations

$$A(\alpha h_0, \varepsilon_0) = A(h_0, \alpha\varepsilon_0), \tag{6}$$

$$b(\alpha h_0, \varepsilon_0) = \alpha^q b(h_0, \alpha\varepsilon_0), \tag{7}$$

where q is the order of the differential operator $\mathcal{D}$.

We will discuss two main strategies for choosing the scaling parameters (i) keeping the shape parameter constant and (ii) stationary scaling. With the first strategy, theory tells us that the error will go to zero as $h \longrightarrow 0$ in exact arithmetic. However, the numerical stability of the computation of the stencil weights depends on the conditioning of the matrices A. Equation (6) tells us that decreasing h with a fixed ε is numerically equivalent to decreasing ε for a fixed h. For a fixed node set, the condition number of A goes to infinity as ε^{-2K} when $\varepsilon \longrightarrow 0$ as shown in [27] and [31]. Hence, if the weights are computed directly from (4), we get numerical stability problems for small ε and/or small h.

A number of numerically stable methods for evaluating RBF interpolants or approximants in the flat $\varepsilon \longrightarrow 0$ limit have been developed [22, 18, 20], see also [21] for global RBF approximations on the sphere. Common for these are that the cost for evaluating the stencil weights becomes higher (by a constant factor), but computations can be stably performed for any small value of ε. Some problems may arise when the stencils become very small in relation to the size of the spherical surface. The surface then is numerically flat, but we are trying to generate a stencil in a three-dimensional space. A work-around is to project the nodes onto the tangent plane and compute the weights in the two-dimensional plane instead (personal communication Bengt Fornberg).

The second strategy, the stationary scaling, is commonly adopted in practical applications, as, e.g, in [15]. By employing equation (6) for both arguments, we have that $A(h_0, \varepsilon_0) = A(\alpha h_0, \varepsilon_0/\alpha)$. That is, if we decrease h while at the same time increasing ε, the condition number stays constant. This is of course an advantage, but as $h \longrightarrow 0$ we will run in to a saturation error at some point, beyond which accuracy is not further improved. The stencil can be modified by inclusion of lower order polynomial terms, by which some order of convergence can be retained. How the stencil approximation errors depend on ε and h is discussed in [3] and explicit formulas are given for low order stencils. In the article [28], the two strategies for stencil scaling are compared with each other as well as with stencils that include polynomial terms. It can be seen that up to the point where the saturation error enters, the performance is similar.

In [15], RBF-FD approximations where computed for N up to $\mathcal{O}(10^5)$ nodes with maintained convergence for the flow over an isolated mountain test case. However, a slight change in the convergence trend for the largest problem size could indicate that the saturation error occurs in that regime. This might put in question the need for higher resolutions and parallel implementations. We will try to put this issue to rest by a hypothetical example. Assume that we want to approximate two different functions $f(x)$ and $g(x) = f(\beta x)$. Then any derivative of order q is scaled by β^q for $g(x)$ compared with the same derivative of $f(x)$ for the corresponding location. If we apply a stencil with the combination to (h_0, ε_0) to $f(x)$, we should apply a stencil with $(h_0/\beta, \beta\varepsilon_0)$ to $g(x)$. Then the function values at the node points for the two functions will be identical and the stencils will be identical up to a scaling factor of β^q (see eq. (7)), corresponding to the higher derivative of $g(x)$. The conclusion is that if saturation strikes at a certain h for $f(x)$, then it strikes at h/β for $g(x)$. Hence, a function with variations at shorter scales ($\beta > 1$) can be resolved more before reaching the saturation error. Furthermore, by using the stable methods, we can use scaling approach (i), which does not suffer from saturation errors.

4. The discrete formulation of the shallow water equations

To obtain a semi-discretization of the PDE, the RBF-FD stencils are used for approximation of the spatial PDE operators at each node point and the results are assembled into one sparse global differentiation matrix per operator involved. Here the spatial operators are the different components of $\boldsymbol{P}\nabla$, and we denote the differentiation matrices by D_N^x, D_N^y, and D_N^z. For details on how to apply the projected operators to the RBFs for the right hand side in the stencil weight computation (4), see [15].

We adopt a tensor notation, where each element of a matrix or a vector in turn is a matrix or a vector. An underlined quantity such as $\underline{\boldsymbol{x}} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)^T$ denotes a node function with a (matrix, vector or scalar) value at each node point. Operations that involve node functions are interpreted as being applied element wise for each node. The discrete gradient operator applied to a vector or a scalar node function yields

$$\boldsymbol{D}_N \underline{\boldsymbol{u}} = \begin{bmatrix} D_N^x \underline{u} & D_N^x \underline{v} & D_N^x \underline{w} \\ D_N^y \underline{u} & D_N^y \underline{v} & D_N^y \underline{w} \\ D_N^z \underline{u} & D_N^z \underline{v} & D_N^z \underline{w} \end{bmatrix}, \quad \boldsymbol{D}_N \underline{h} = \begin{bmatrix} D_N^x \underline{h} \\ D_N^y \underline{h} \\ D_N^z \underline{h} \end{bmatrix}, \tag{8}$$

and the discrete divergence operator similarly becomes

$$\boldsymbol{D}_N \cdot \underline{\boldsymbol{u}} = D_N^x \underline{u} + D_N^y \underline{v} + D_N^z \underline{w}. \tag{9}$$

With this notation, the semi-discrete form of the shallow water equations on the sphere becomes

$$\frac{\partial \underline{\boldsymbol{u}}}{\partial t} = -\underline{\boldsymbol{P}} \left[\underline{\boldsymbol{u}} \cdot \boldsymbol{D}_N \underline{\boldsymbol{u}} + f(\underline{\boldsymbol{x}} \times \underline{\boldsymbol{u}}) + g \boldsymbol{D}_N \underline{h} \right], \tag{10}$$

$$\frac{\partial \underline{h}}{\partial t} = -\left(\underline{\boldsymbol{u}} \cdot \boldsymbol{D}_N \underline{h} + \underline{h} \boldsymbol{D}_N \cdot \underline{\boldsymbol{u}} \right). \tag{11}$$

For the time-derivative, the classical fourth order Runge-Kutta method is used. The SWE are of hyperbolic type, which motivates the choice of an explicit time-stepping method. To ensure stability, small hyperviscosity terms $\gamma D_N^L \underline{\boldsymbol{u}}$ and $\gamma D_N^L \underline{h}$ respectively, are incorporated into the scheme. These represent a small amount of diffusion, which acts as stabilizer without significantly altering the solution values. As in [15], all operators are discretized using Gaussian RBFs, which allows for efficient evaluation of the hyperviscosity operator [19].

The computationally most expensive operations in evaluating the right hand side functions in the system of ODEs (10)–(11) resulting from the spatial discretization of the PDEs (1)–(2) are the applications of differentiation matrices to node functions. There are four solution components u, v, w, and h, and each is differentiated with respect to the three coordinate directions. Furthermore, we apply the hyperviscosity once to each component. This results in a total of 16 sparse matrix-vector multiplications for each evaluation of the right hand side functions.

5. Task based parallel programming

We have chosen to use a task-based programming model to parallelize the SWE solver. There are two main reasons for this choice. The first is the programmer productivity aspect. A key idea in task parallel programming is that it allows the programmer to write a sequential code in terms of computational tasks, and then a run-time system handles the parallel execution of the tasks. In scientific applications, tasks typically share some data and this leads to dependencies between tasks in terms of constraints on the order in which they can execute. The run-time system needs to respect these constraints when scheduling tasks for execution. Figure 4 shows a schematic representation of an algorithm in terms of dependent tasks.

The second reason to choose a task-based programming model is performance. Tasks are scheduled dynamically at run time to the cores, processors, and computational nodes. Load balancing, so far only in the shared memory setting, is also achieved at run-time by task stealing. No particular assumptions are made about the hardware beforehand except for the distinction between shared memory and distributed memory. The execution instead adapts to the conditions of the system where the software is running. Another important aspect is that in the task model, global barriers, which typically do not scale well can be avoided. Synchronization is instead fine grained between different

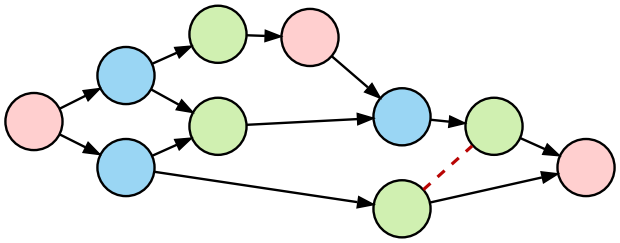


Figure 4: Example of a task graph from a 3×3 block Cholesky factorization.The circles represent tasks and the arrows represent a must-execute-before relationship. The dashed line connects two tasks that can run in any order, but not at the same time. The red tasks are factorizations, the blue are column updates, and the green are the matrix-multiplication updates in the trailing submatrix.

tasks. In [26] and [7] the suitability of different programming models and scheduling approaches was investigated for linear algebra problems on multicore based architectures, and dynamic scheduling was found to be beneficial for performance.

In this work, we use the SuperGlue[1] library, developed by the first author [35, 33, 34], for dependency-aware task-based parallel programming. The performance of SuperGlue has been compared with that of related efforts and was shown to have comparatively very low run-time overhead. In SuperGlue, the programmer defines tasks and also registers what type of access to shared data that each task performs. The types can be for example *read*, *write*, and *add*, where the latter is used for accesses that can be performed in any order, but not at the same time. The tasks are submitted to the run-time system, which infers the dependencies from the data accesses and schedules the tasks accordingly.

An important principle in our view of task parallelism is that tasks depend on data, not on other tasks. This allows us to add dependencies without having to synchronize with other tasks. We use a system with data versioning, where tasks depend on a particular version of the data. When the required version becomes available, the task can execute. It should be noted that we do not duplicate data. The versions replace each other as they become available. The versions are handled by a data object that we call a (data) handle. This gives additional flexibility and allows us to construct handles not only for consecutive blocks of data, but for arbitrary data structures, as well as for logical resources such as for example disk I/O.

6. The sequential and parallel implementations of the solver

In this section, we will describe the implementation, optimization, and parallelization of the RBF-FD method for the SWE step by step, starting with the sequential code and leading up to the parallel distributed code.

6.1. The sequential MATLAB code

The first implementation of the RBF-FD SWE solver was done in MATLAB. However, as reported in [15], even the pilot implementation was 4–10 times faster, when comparing

[1] `https://github.com/tillenius/superglue`

serial executions, than the latest C++ discontinuous Galerkin solver developed at the National Center for Atmospheric Research (NCAR), Boulder, CO, USA.

Listing 1 shows the main computational steps of the MATLAB implementation. First the necessary RBF-FD matrices are set up, and then the time stepping is performed in a loop. The variable `H` is an $(N \times 4)$ matrix with the node functions of the four solution components at the current time step as its columns.

```
% Build differentiation matrices
% and hyperviscosity operator
[DPx, DPy, DPz, L] = rbfmatrix_fd();

for i=1:timesteps
  % Runge-Kutta
  F1 = dt*rhs( H );
  F2 = dt*rhs( H + 0.5*F1 );
  F3 = dt*rhs( H + 0.5*F2 );
  F4 = dt*rhs( H + F3 );
  H = H + (1/6)*(F1 + 2*F2 + 2*F3 + F4);
end
```

Listing 1: Excerpt from the main program of the MATLAB implementation.

Profiling of the MATLAB program shows that the majority of the time (74% for the tested problem size) is spent in the evaluation of the right hand side, i.e, the `rhs()` function. Therefore, this is the part of the program that we target initially. Note that the initial computation of the different stencil weights is trivially parallelizable in the sense that for each operator, the N linear systems of size $(n \times n)$ are independent of each other.

Further examining the computations in the right hand side function, we find that over 90% of the time is spent in sparse matrix-vector multiplications between differentiation matrices and intermediate solution vectors. Hence, parallelizing this code efficiently amounts to handling sparse unstructured matrix-vector multiplications, which is a classical example of a bandwidth-bound operation that is hard to get to scale on multicore based computer systems.

6.2. The sequential C++ code

We consider the time-stepping loop only and start by performing optimizations and adjustments of the sequential code. A first step is to change from column major storage as used by MATLAB to row-major compressed sparse row (CSR) storage of the matrices. Next, we note that each instance of the solution value (u, v, w, h) is of length four, corresponding to one row in the variable `H`. This allows us to employ AVX (Advanced Vector Extensions to the x86 instruction set) SIMD instructions when multiplying the differentiation matrices with `H`. The SIMD instructions are significantly more efficient than performing four single instruction operations.

Furthermore, the differentiation matrices, denoted by `DPx`, `DPy`, and `DPz` in the MATLAB code, have identical sparsity patterns since they use the same stencil sizes. By storing them together in the aggregated operator `D`, we can reuse the sparsity pattern. Instead of performing three separate matrix-vector multiplications with `H`, we compute

Table 1: The execution time in millions of cycles and the speedup of (the different parts of) the right hand side computation when the sequential C++ code is compared with the sequential MATLAB code.

	MATLAB	C++	Speedup
Differentiation	8186	1441	5.7
Hyperviscosity	2606	679	3.8
Other rhs ops	790	200	4.0
Total	12062	2402	5.0

them all at the same time as `T=D*H`, where `T` has 12 values per node point. Technically, we use an array of structs instead of separate arrays.

The overall speedup from the MATLAB implementation to the sequential optimized C++ code was 5.0 times. For a breakdown into different parts, see Table 1. Note that much of the gain was achieved by exploiting the structure of this specific problem. We could not have achieved the same speedup by using a standard optimized library routine for sparse matrix-vector multiplication.

6.3. The task parallel shared memory C++ code

As discussed in Section 5, we will use the task parallel framework SuperGlue for the implementation of the parallel code. The first step is a shared memory parallel implementation, and the problem we will discuss is how to do the sparse-matrix vector multiplications efficiently using SuperGlue.

We need to formulate the algorithm in terms of tasks. This can be done in different ways for any given problem and it does have implications for performance. Task size is one important parameter. Too small tasks make the overhead from scheduling visible, whereas too large tasks are hard to schedule efficiently and may lead to reduced parallelism. Then the way that the data and the algorithm are partitioned also has an impact. Figure 5 shows two alternative ways of blocking the matrix in order to define tasks and the resulting access pattern of each task.

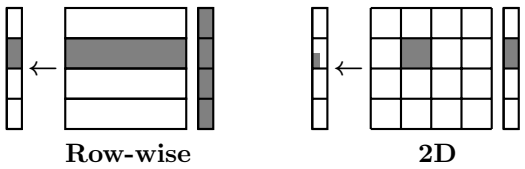


Figure 5: Two choices of blockings to divide the algorithm into tasks. The data that a task needs to touch is shaded.

The row-wise blocking has the advantage of producing tasks with an equal amount of work, since the number of non-zeros in each row equals the stencil size. The tasks are also larger than in the 2D-blocking case. However, each task needs to access the full vector. In the 2D-blocking case, we create tasks that only write to a single block of the output vector, in order to maximize parallelism. This leads to smaller tasks and different amounts of work per task, but also creates more potential parallelism.

Listing 2 shows the SuperGlue user code for the 2D-blocked matrix-vector multiplication. First, handles are created for the involved blocked data structures with `nb` blocks in each dimension. Then the block multiplication tasks are submitted in a loop. Finally, the

task is defined. The data accesses are registered in the constructor and the run method (which is not displayed here) implements the computational kernel.

```
BlockedMatrix D(nb,nb);
BlockedVector H(nb);
BlockedVector T(nb);

// T += D*H
for (int i = 0; i < nb; ++i)
  for (int j = 0; j < nb; ++j)
    submit(new mult_task(T(i), D(i,j), H(j)));

struct mult_task : public Task {
  mult(VectorBlock &T, MatrixBlock &D, VectorBlock &H) {
    registerAccess(add, T.handle);
    registerAccess(read, D.handle);
    registerAccess(read, H.handle);
  }
  void run() { /* T += D*H */ }
};
```

Listing 2: Blocked matrix-vector multiplication using SuperGlue. Note that the actual multiplication kernel is not shown.

To see how we can build a more complete application, we also show the code for the whole time-stepping loop in Listing 3. GenTasks is a task that submits other tasks. It calls a number of subroutines, which be listed below, that in turn submits the computational tasks. When all task for one time-step have been submitted, a new instance of GenTasks is added.

```
// Runge-Kutta step
GenTasks::run() {
  f(F1, H);                              // F1 = f(H)
  add(H1, H, 0.5*dt, F1); f(F2, H1);     // F2 = f(H + 0.5*dt*F1)
  add(H2, H, 0.5*dt, F2); f(F3, H2);     // F3 = f(H + 0.5*dt*F2)
  add(H3, H,     dt, F3); f(F4, H3);     // F4 = f(H + dt*F3)

  step(H, F1, F2, F3, F4);               // H = H + dt/6*(F1+2*F2+2*F3+F4)

  submit(new GenTasks(H));                  // Generate new tasks for next step
}

// evaluate ∂H/∂t
void f(dH, H) {
  mult(T, D, H);            // T = D*H
  rhs(dH, H, T);            // dH = ...
  mult(dH, L, H);           // dH = dH + L*H
}
```

Listing 3: The main loop in the task parallel code. Note that all the subroutines here submit tasks.

The implementation of the `mult` subroutine was indicated in Listing 2. The other helper routines are displayed in Listing 4.

What we would like to emphasize here is that the code complexity is quite similar to that of the original MATLAB implementation. Certainly, we need to write a wrapper

```
add(Htmp, a, H) {
  for (int r = 0; r < nb; ++r)
    submit(add_task(Htmp(r), a, H(r));
}

rhs(dH, H, T) {
  for (int r = 0; r < nb; ++r)
    submit(rhs_task(H(r), T(r)));
}

step(H, F1, F2, F3, F4) {
  for (int r = 0; r < nb; ++r)
    submit(step_task(H(r), F1(r), F2(r),
                     F3(r), F4(r)));
}
```

Listing 4: The helper tasks that submit the computational tasks for each block of the matrix `H`.

for each computational kernel to make it into a task, but it is still easy to follow the code and recognize the different steps in the algorithm.

6.4. The task parallel distributed memory C++ code

The management of the distributed environment and the communication between different computational nodes is implemented as a layer on top of SuperGlue, using MPI for the message passing calls. Because the distributed framework builds on the SuperGlue programming model, the user code for the distributed case is almost identical to the shared memory code. The main difference from the user perspective is that the handles and tasks now are upgraded to `MPI_Handle` and `MPI_Task` with some additional functionality. Therefore, when an `MPI_Handle` is created as opposed to the basic handle, the user must in addition specify the data owner process rank and the associated memory block. Each computational node submits the same sequence of tasks to the run-time system, which then decides which tasks will be executed locally and which to discard. The decision is based on the location of the output data from the task. The run-time system also detects for which tasks data transfers from other nodes are needed and inserts the appropriate communication tasks. One core at each node is dedicated to the MPI processing. We are also developing another version of the distributed task-based framework, DuctTeip [45], with the same programming model, and also built SuperGlue, but with slightly different design choices.

Figure 6 shows a simplified example of what the run-system does in a situation when a transfer is needed. The transfer is detected when the Copy task is submitted. For the process with rank 0, the owner of the data x, a task that will send the data is submitted. The process with rank 1 instead submits a PublishTask. This waits to receive the data an then copies the data to the handle. FinishedSendTask updates the version number of the data when the send is completed. In addition to keeping track of version numbers, the handles also register who last wrote to a certain data, and which nodes already have a copy of the latest version. When scheduling tasks, the run-time system gives priority to tasks that work on data that will be communicated.

```
x.set_owner_rank(0);
y.set_owner_rank(1);

submit(new Write(x));
submit(new Copy(x, y));
submit(new Write(x));
```

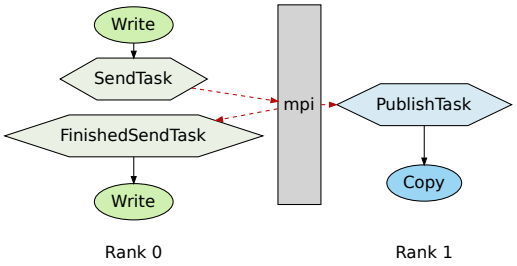


Figure 6: The code to the left shows the tasks that the programmer submits, and the figure to the right shows also the additional tasks that are submitted by the run time system and how they interact with the MPI process thread.

7. Experimental results

The numerical experiments are performed on the Tintin cluster at the UPPMAX facilities at Uppsala University. The cluster has 160 computational nodes equipped with dual AMD Opteron 6220 Bulldozer CPUs. Formally, this gives us 16 cores per node. However, it is important to note that each floating point unit is shared between two cores. Hence, for the type of floating point heavy simulations performed here, we can not expect a speedup of more than 8 times on one computational node.

The result section consists of two main parts, first we analyze the performance of the parallel implementation, and then we discuss the SWE solutions computed with the parallel RBF-FD method.

7.1. Parameter values in the experiments

For the "flow over an isolated mountain" test case, we use stencil size $n = 31$, hyperviscosity of order 4, and the amount of hyperviscosity $\gamma = -0.05 \cdot N^{-4}$. The node sets, shape parameters, time steps, and block sizes used are listed in Table 2. These are the values that were also used in [15]. However, for the largest problem size, we needed to increase the shape parameter value in order to have stability. This will decrease the accuracy of this solution compared to having the expected value $\varepsilon = 28$, and the last column in the table shows how this affects the product εh (which relates to the stationary scaling), assuming that h is proportional to $1/\sqrt{N}$.

Table 2: The node sets, shape parameter values, and time steps used for the numerical simulations of the "flow over an isolated mountain" test case. The last column indicates how the relation between the shape parameter and the stencil size changes.

N	Type	ε	Δt	n_b	$\varepsilon/\sqrt{N}$
6 400	MD	2.7	900	400	0.0338
25 600	MD	5.5	300	800	0.0344
40 962	MD	7.0	180	1000	0.0346
163 842	ICO	14.1	60	2000	0.0348
655 362	ICO	40.0	60	5000	0.0494

The second test case "evolution of a highly non-linear wave" is much more sensitive, and in order to get a stable and accurate solution, we needed to modify the parameter

Table 3: The parameter values for the numerical simulations of the second test case.

$n = 31$						$n = 101$					
N	Type	ε	m	γ_c	Δt	N	Type	ε	m	γ_c	Δt
6 400	MD	2.7	4	-0.05	600	6 400	MD	4.48	10	-5	600
25 600	MD	6.0	4	-0.05	300	25 600	MD	9.12	10	-5	300
40 962	MD	7.0	4	-0.05	180	42 768	DM	15.0	8	-0.5	60
163 842	ICO	14.1	4	-0.05	60	155 718	DM	32.0	6	-0.05	10
655 362	ICO	40.0	4	-0.10	5						

values. These are given in Table 3. Note that the hyperviscosity operator has the form $\gamma\Delta^m = \gamma_c\, N^{-m}\Delta^m$.

7.2. Performance analysis

All experiments in this section are performed for the "flow over an isolated mountain" test case. In each experiment, we run the code for 100 time-steps, which is enough to compute the speedup, but does not correspond to a complete run. For the largest problem with time step $\Delta t = 10$s, the total simulation time corresponding to 15 days requires 129 600 time steps. The sparse matrices are divided into blocks of different sizes depending on the problem size. These block sizes have not been optimized. For the largest problem we use a block size of 5000×5000, resulting in 130×130 blocks. However, the blocks that are empty are discarded.

We start with experiments for the shared memory case as this is the basis for the distributed memory code and the performance properties will partly be inherited. Figure 7 visualizes an execution trace for the largest problem size considered, with $N = 655\,362$ nodes. The tasks from different time steps do not need to synchronize globally, and the scheduler can take advantage of this to schedule any tasks that are ready to run, in parallel. In fact, the scheduler is very successful, with an idle time in the schedule of only 1.29%.

Table 4 shows the resulting speedups for different problem sizes. The parallel code with execution time T_p is compared both against the best serial (unblocked) code with execution time T_{best}, and against the parallel (blocked) code running on 1 core with execution time T_1. The estimated total execution time of the whole simulation T_{tot} is also given as this is the time that the end user will experience.

Table 4: Speedups for 100 time steps using the shared memory code. We compare against the best serial code, T_{best}, and execution on one core with the parallel code, T_1.

Size N	T_p (s)	Speedup T_{best}/T_p	Speedup T_1/T_p		Total T_{tot}
6 400	0.2	5.6	8.1	(101%)	2s
25 600	0.7	6.5	7.8	(98%)	30s
40 962	1.1	6.6	7.6	(95%)	1m 22s
163 842	5.2	6.4	6.8	(85%)	18m 35s
655 362	21.0	5.2	5.7	(71%)	1h 15m 31s

The theoretical optimal speedup is 8 on the system we are using, and with the very low idle time seen in the execution trace, we might expect a nearly perfect speedup. However,

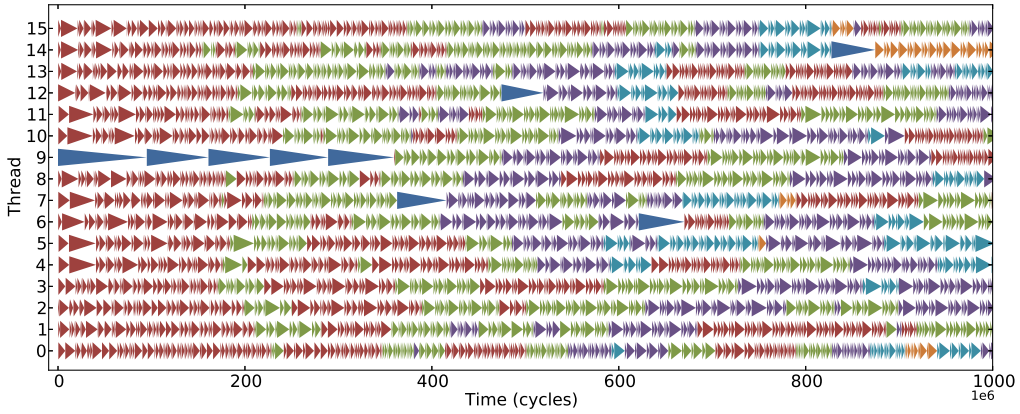


Figure 7: Part of an execution trace for $N = 655\,362$ nodes and 16 shared memory cores. Each triangle represents a task with start time at the base and finish time at the tip. Tasks of the same color belong to the same time-step. The longer tasks are the ones that submit the computational tasks.

this is where the bandwidth-bound characteristics of the sparse matrix-vector multiplication comes in. In current multicore processors, resources such as caches and memory bandwidth are shared between several cores. This may result in contention between the cores and consequently performance degradation when the resource is oversubscribed.

To evaluate the contention hypothesis, we break down the execution into different parts and then compute the total time spent in each part in the sequential case and in the parallel case (sum over the cores). Figure 8 shows the increase in total computational time for the parallel code compared with the sequential. The slowdown that we experience indicates that the cores are waiting for data to be delivered from memory. The problem is a combined effect of having few computations per data access, and not accessing data consecutively in memory. The two main parts of the execution time represent the application of the differentiation matrices and the hyperviscosity operator. Because we could aggregate the three differentiation operators into one, and reuse the sparsity pattern, this part becomes computationally heavier than the hyperviscosity operator and therefore scales slightly better.

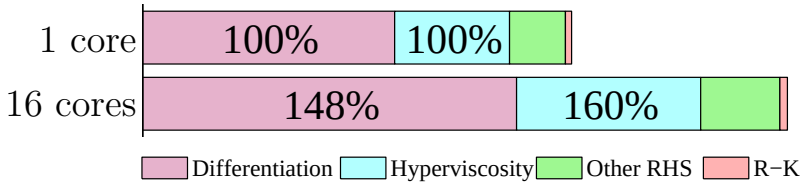


Figure 8: The total time spent in different parts of the computation in the serial code compared with the shared memory code. The increase in time in the parallel case is due to resource contention.

How to schedule to decrease the resource contention is discussed in [36]. However, this approach only improves the situation if there are tasks that do not need the resource in question that can be interleaved with the resource bound tasks. This is not the case here, since all the heavy tasks are similar in nature.

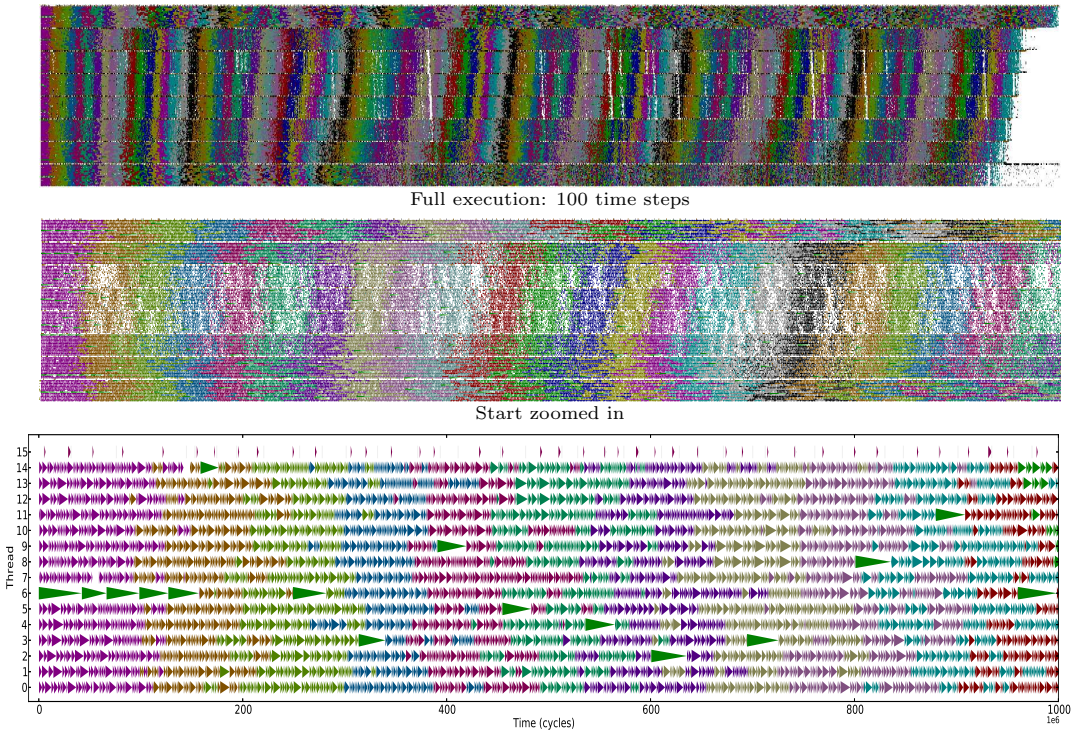


Figure 9: The execution trace for $N = 655\,362$ nodes using a distributed system with 8 computational nodes, each with 16 shared memory cores. The top subfigure shows the full execution trace for 100 time steps. There are 8 bands, each representing one computational node. In the middle subfigure, the trace is zoomed in. The lower subfigure shows a part of the trace only for the first computational node (the lower band in the other figures). The colors represent the different time steps.

Figure 9 shows execution traces for the first 100 time-steps of the problem with $N = 655\,363$ nodes for the distributed memory code. The total idle time is here 19.9%, which is larger than in the shared memory case. The MPI thread, which can be seen in the lower subfigure, running on core 15, is not included in the idle time evaluation. By inspecting the traces, we can see that there clearly is a load imbalance as the different computational nodes do not finish at the same time, and from the middle subfigure, we can also see that the task density (or the amount of work) is less for nodes 3–5. In fact, the idle time for node 7 (the uppermost band in the top subfigure) is only 2.16% (similar to the the shared memory experiment), while it is significantly higher for the other nodes.

The distribution of data between the computational nodes, and consequently the distribution of work is currently static, determined before the execution starts. By implementing a load-balancing strategy at the node level, this can be improved. We are currently investigating mechanisms for automatic load-balancing over the nodes.

Table 5 shows the speedup of the distributed code for different numbers of computational nodes. The execution time T_q for running on q computational nodes is compared both against the best shared memory code (without MPI) with execution time denoted

Table 5: Speedups for 100 time steps using the distributed memory code. We compare against the best shared memory code, T_{shared}, and execution on one node with the parallel code, T_1.

Nodes q	T_q (s)	Speedup T_{shared}/T_q	Speedup T_1/T_q	Idle time	T_{tot}
1	21.43	0.98	1.00	1.5%	1h 17m 11s
2	10.49	2.00	2.04 (102.2%)	2.8%	37m 46s
4	5.45	3.85	3.93 (98.3%)	7.4%	19m 38s
6	3.56	5.89	6.03 (100.5%)	5.7%	12m 48s
8	3.02	6.94	7.11 (88.9%)	19.9%	10m 51s

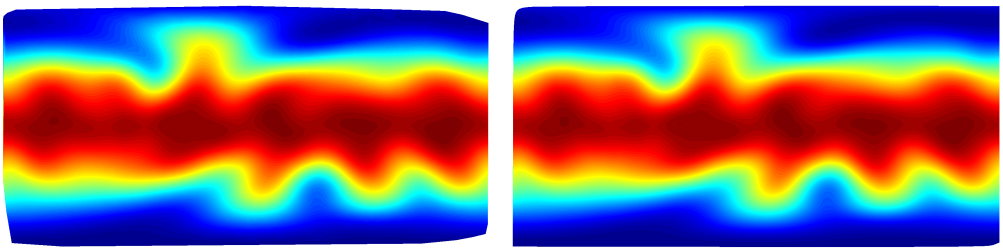
by T_{shared}, and against the distributed code (with MPI) running on one node, T_1.

Looking at these numbers, we find that the scaling of the parallel code is very close to the theoretical best speedup of q except for the last value. However, comparing with the amount of idle time, indirectly representing the load imbalance, we see that this is why the speedup is a bit lower. If we can reduce the idle time down to 7%, we would have a speedup over 8. This is entirely possible by redistribution of the work. From this we conclude that if the load balancing is handled, there is nothing to prevent further scaling to larger numbers of computational nodes from the parallelization perspective.

If we now look at the overall speedup that has been achieved, the distributed parallel code running on 8 computational nodes is currently 178 times faster than the original MATLAB implementation, 36 times faster than the optimized serial C++-code, and 7 times faster than the parallel shared memory code. Clearly, this enables both larger and faster SWE simulations using the RBF-FD method.

7.3. Shallow water equation simulation results

The "flow over an isolated mountain" test case is comparatively easy to solve, and visually, the solutions appear identical for all problem sizes. Figure 10 shows the solutions for the lowest resolution $N = 6\,400$, which corresponds to 300km, and the highest resolution $N = 655\,362$, which corresponds to around 30km. We do not carry out a convergence study here, but it was done in [15] for all problem sizes up to the next largest one that we use. Instead we focus on finding out if we can solve the larger problems with a qualitatively well behaved solution.



$N = 6\,400$ $N = 655\,362$

Figure 10: The computed geopotential height after 15 days for two different resolutions for the "flow over an isolated mountain" test case. The top boundary corresponds to the North Pole, the bottom boundary is the South Pole, and the zero longitude is located in the center of the horizontal axis.

For the "evolution of a highly non-linear wave" test case it is much harder to get a physically correct solution. It also takes longer to solve the problem because we need a smaller time step. The largest simulation took 4.8 times longer than the corresponding simulation for the first test case. In Figure 11, a number of solutions are displayed. Solutions for smaller node numbers can be found in [15] for comparison. The solution for $N = 163\,842$ nodes and stencil size $n = 31$ is closest to the expected behavior. For the larger stencil size $n = 101$, we needed to increase ε and decrease the order of the hyperviscosity operator in order to have a stable scheme. Both of these actions serve to decrease the accuracy of the solution, and as can be seen in the figures to the right, these solutions are of lower quality than the corresponding solutions for $n = 31$. The solution for the largest problem size was computed stably, but the right part of the solution does not look precisely as it should.

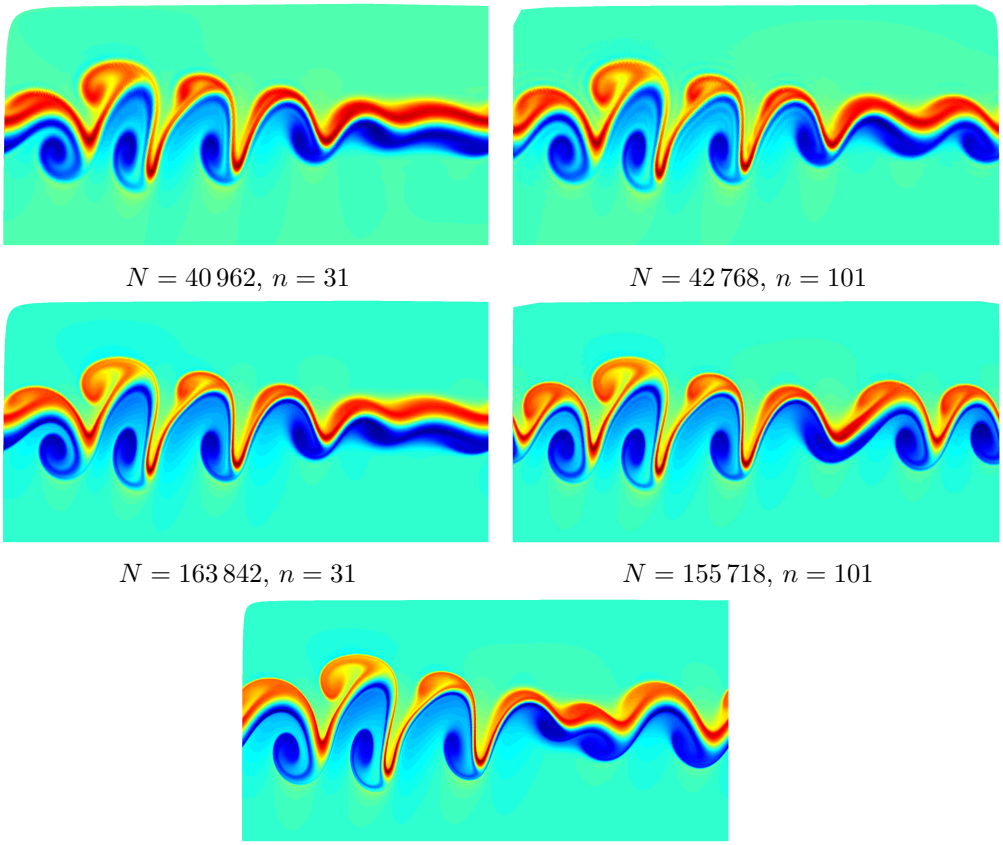


Figure 11: The computed relative vorticity after 6 days for different resolutions and stencil sizes for the "the evolution of a highly non-linear wave" test case. The top boundary corresponds to the North Pole, the bottom boundary is the Equator, and the zero longitude is located in the center of the horizontal axis.

We believe that the behavior can be improved by employing one of the stable methods,

RBF-QR [18] or RBF-GA [20], to compute stencils for smaller ε-values. This is something that we will implement and test.

8. Conclusions

Our two main objectives in this paper were to show that The RBF-FD method can be efficiently parallelized and that task-based parallel programming was a suitable approach both for productivity and performance.

Starting from the original MATLAB implementation, which was shown to perform well sequentially compared with a discontinuous Galerkin solver [15], we have achieved 178 times speedup, using only 8 computational nodes in a cluster. Part of this speedup comes from specific optimizations of the sparse matrix-vector multiplications. These are method and problem specific and can not be found in standard library routines. However, similar opportunities would arise in other RBF-FD solvers as well. Judging from the performance experiments, the code can scale to even larger computer systems, if needed. However, a better load-balancing strategy in the distributed case is needed.

The parallelization was done using the SuperGlue library for task-based parallel programming. The resulting code is quite similar to the original code except that it is restructured to work with matrix and vector blocks instead of the entire structures. The programmer does not need to pay any particular attention to the parallel aspects, except that in the distributed code, the data distribution is specified. The block size does affect performance to some extent, but it is not critical to tune it perfectly. Overall, the productivity goal is met compared with the effort it would take to hardcode all interactions and schedules.

The task-based approach resulted in a very efficient code. There is very little idle time, and the performance losses that are observed are due to resource sharing, which cannot be completely avoided for a multicore based system. We have no reason to believe that another parallel approach would be more efficient.

The aspects that still needs to be improved are related to numerical stability. We are currently not able to perform accurate computations with large stencil for the largest problem sizes. One potential improvement is to use special numerically stable methods [18, 20] to compute the stencil weights. Another is to choose the stencil nodes individually depending on the local node layout. Finally, introducing the local node refinement could also prove beneficial for stability, since all features would be resolved properly. However, this still needs to be investigated.

[1] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency Computat. Pract. Exper. 23 (2) (2011) 187–198.

[2] J. R. Baumgardner, P. O. Frederickson, Icosahedral discretization of the two-sphere, SIAM J. Numer. Anal. 22 (6) (1985) 1107–1115.

[3] V. Bayona, M. Moscoso, M. Carretero, M. Kindelan, RBF-FD formulas and convergence properties, J. Comput. Phys. 229 (22) (2010) 8281–8295.

[4] V. Bayona, M. Moscoso, M. Kindelan, Gaussian RBF-FD weights and its corresponding local truncation errors, Eng. Anal. Bound. Elem. 36 (9) (2012) 1361–1369.

[5] V. Bayona, M. Moscoso, M. Kindelan, Optimal variable shape parameter for multiquadric based RBF-FD method, J. Comput. Phys. 231 (6) (2012) 2466–2481.

[6] E. F. Bollig, N. Flyer, G. Erlebacher, Solution to PDEs using radial basis function finite-differences (RBF-FD) on multiple GPUs, J. Comput. Phys. 231 (21) (2012) 7133–7151.

[7] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Computing 35 (1) (2009) 38–53.
[8] O. Davydov, D. T. Oanh, Adaptive meshless centres and RBF stencils for Poisson equation, J. Comput. Phys. 230 (2) (2011) 287–304.
[9] O. Davydov, D. T. Oanh, On optimal shape parameter for Gaussian RBF-FD approximation of Poisson equation, Comput. Math. Appl. 62 (2011) 2143–2161.
[10] H. Ding, C. Shu, D. B. Tang, Error estimates of local multiquadric-based differential quadrature (lmqdq) method through numerical experiments, International Journal for Numerical Methods in Engineering 63 (11) (2005) 1513–1529.
[11] E. Divo, A. J. Kassab, An efficient localized radial basis function meshless method for fluid flow and conjugate heat transfer, J. Heat Transfer 129 (2) (2006) 124–136.
[12] T. A. Driscoll, B. Fornberg, Interpolation in the limit of increasingly flat radial basis functions, Comput. Math. Appl. 43 (3-5) (2002) 413–422.
[13] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, OmpSs: a proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (2) (2011) 173–193.
[14] N. Flyer, E. Lehto, Rotational transport on a sphere: local node refinement with radial basis functions, J. Comput. Phys. 229 (6) (2010) 1954–1969.
[15] N. Flyer, E. Lehto, S. Blaise, G. B. Wright, A. St-Cyr, A guide to RBF-generated finite differences for nonlinear transport: shallow water simulations on a sphere, J. Comput. Phys. 231 (11) (2012) 4078–4095.
[16] N. Flyer, G. B. Wright, Transport schemes on a sphere using radial basis functions, J. Comput. Phys. 226 (1) (2007) 1059–1084.
[17] N. Flyer, G. B. Wright, A radial basis function method for the shallow water equations on a sphere, Proc. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci. 465 (2106) (2009) 1949–1976.
[18] B. Fornberg, E. Larsson, N. Flyer, Stable computations with Gaussian radial basis functions, SIAM J. Sci. Comput. 33 (2) (2011) 869–892.
[19] B. Fornberg, E. Lehto, Stabilization of RBF-generated finite difference methods for convective PDEs, J. Comput. Phys. 230 (2011) 2270–2285.
[20] B. Fornberg, E. Lehto, C. Powell, Stable calculation of Gaussian-based RBF-FD stencils, Comput. Math. Appl. 65 (4) (2013) 627–637.
[21] B. Fornberg, C. Piret, A stable algorithm for flat radial basis functions on a sphere, SIAM J. Sci. Comput. 30 (1) (2007) 60–80.
[22] B. Fornberg, G. Wright, Stable computation of multiquadric interpolants for all values of the shape parameter, Comput. Math. Appl. 48 (5-6) (2004) 853–867.
[23] J. Galewsky, R. Scott, L. Polvani, An initial-value problem for testing numerical models of the global shallow-water equations, Tellus A 56 (5).
[24] G. Kosec, B. Šarler, Solution of thermo-fluid problems by collocation with local pressure correction, Int. J. Numer. Meth. Heat & Fluid Flow 18 (7/8) (2008) 868–882.
[25] G. Kosec, R. Trobec, M. Depolli, A. Rashkovska, Multicore parallelization of a meshless PDE solver with OpenMP, in: G. Haase, M. Liebmann (eds.), Parallel Numerics 11, Leibnitz, Austria, 2011, pp. 58–69.
[26] J. Kurzak, J. Dongarra, Implementing linear algebra routines on multi-core processors with pipelining and a look ahead, in: B. Kågström, E. Elmroth, J. Dongarra, J. Waśniewski (eds.), Applied Parallel Computing. State of the Art in Scientific Computing, vol. 4699 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 147–156.
[27] E. Larsson, B. Fornberg, Theoretical and computational aspects of multivariate interpolation with increasingly flat radial basis functions, Comput. Math. Appl. 49 (1) (2005) 103–130.
[28] E. Larsson, E. Lehto, A. Heryudono, B. Fornberg, Stable computation of differentiation matrices and scattered node stencils based on Gaussian radial basis functions, SIAM J. Sci. Comput. 35 (4) (2013) A2096–A2119.
[29] Y. J. Lee, G. J. Yoon, J. Yoon, Convergence of increasingly flat radial basis interpolants to polynomial interpolants, SIAM J. Math. Anal. 39 (2) (2007) 537–553.
[30] P.-O. Persson, G. Strang, A simple mesh generator in Matlab, SIAM Rev. 46 (2) (2004) 329–345.
[31] R. Schaback, Multivariate interpolation by polynomials and radial basis functions, Constr. Approx. 21 (3) (2005) 293–317.
[32] C. Shu, H. Ding, K. S. Yeo, Local radial basis function-based differential quadrature method and its application to solve two-dimensional incompressible Navier–Stokes equations, Comput. Methods Appl. Mech. Eng. 192 (2003) 941–954.

[33] M. Tillenius, Leveraging multicore processors for scientific computing, Licentiate thesis, Department of Information Technology, Uppsala University (Sep. 2012).
[34] M. Tillenius, SuperGlue: A shared memory framework using data-versioning for dependency-aware task-based parallelization, Tech. Rep. 2014-010, Department of Information Technology, Uppsala University, to appear (Apr. 2014).
[35] M. Tillenius, E. Larsson, An efficient task-based approach for solving the n-body problem on multicore architectures, in: PARA 2010: State of the Art in Scientific and Parallel Computing, University of Iceland, Reykjavík, 2010, 4 pp.
[36] M. Tillenius, E. Larsson, R. M. Badia, X. Martorell, Resource-aware task scheduling, Tech. Rep. 2014-001, Department of Information Technology, Uppsala University (Jan. 2014).
[37] M. Tillenius, E. Larsson, E. Lehto, N. Flyer, A task parallel implementation of a scattered node stencil-based solver for the shallow water equations, in: Proc. 6th Swedish Workshop on Multi-Core Computing, Halmstad University, Halmstad, Sweden, 2013, pp. 33–36.
[38] A. I. Tolstykh, On using RBF-based differencing formulas for unstructured and mixed structured-unstructured grid calculations, in: Proceedings of the 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation, Lausanne, Switzerland, 2000, p. 6 pp.
[39] D. L. Williamson, J. B. Drake, J. J. Hack, R. Jakob, P. N. Swarztrauber, A standard test set for numerical approximations to the shallow water equations in spherical geometry, J. Comput. Phys. 102 (1) (1992) 211–224.
[40] R. S. Womersley, I. H. Sloan, How good can polynomial interpolation on the sphere be?, Adv. Comput. Math. 14 (3) (2001) 195–226.
[41] R. S. Womersley, I. H. Sloan, Interpolation and cubature on the sphere, website `http://web.maths.unsw.edu.au/~rsw/Sphere/` (2003).
[42] G. B. Wright, Radial basis function interpolation: numerical and analytical developments, Ph.D. thesis, University of Colorado, Boulder (2003).
[43] G. B. Wright, N. Flyer, D. A. Yuen, A hybrid radial basis functionpseudospectral method for thermal convection in a 3-d spherical shell, Geochem. Geophys. Geosys. 11 (7) (2010) Q07003.
[44] G. B. Wright, B. Fornberg, Scattered node compact finite difference-type formulas generated from radial basis functions, J. Comput. Phys. 212 (1) (2006) 99–123.
[45] A. Zafari, M. Tillenius, E. Larsson, Programming models based on data versioning for dependency-aware task-based parallelisation, in: Proc. 15th International Conference on Computational Science and Engineering, IEEE Computer Society, 2012, pp. 275–280.