A task parallel implementation of a scattered node stencil-based solver for the shallow water equations*

Martin Tillenius and Elisabeth Larsson
Dept. of Information Technology
Uppsala University, Box 337
SE-751 05 Uppsala, Sweden
{martin.tillenius,elisabeth.larsson}@it.uu.se

Erik Lehto and Natasha Flyer
National Center for Atmospheric Research
P.O. Box 3000
Boulder, CO 80307-3000, USA
{lehto,flyer}@ucar.edu

ABSTRACT

The shallow water equations (SWE) provide a basic model for atmospheric flow and are used as a standard benchmark problem for climate simulation codes. Solving the SWE globally is computationally challenging due to the problem size and the need to resolve local features at different scales. The problem size can be partly addressed by parallel computing, whereas the local adaptivity is a methodological problem. Radial basis function-generated finite difference methods (RBF-FD) have been proposed as a competitor with great potential to the established discontinuous Galerkin and pseudo-spectral methods. In this work, we consider the parallel programming aspects of this problem. The RBF-FD method results in (unstructured) sparse matrix-vector operations, and is hence completely bandwidth bound. The basic algorithm consists of (explicit) time-stepping, which is data-parallel in the sense that the same operation is performed for all data, but also inflicts frequent (local) synchronization points. The problem has been implemented using a dependency-aware task-based programming model. We have used the library SuperGlue which provides a run-time system for dynamic scheduling of dependent tasks. For the problem sizes we have used, the algorithm scales well up to about 6 cores.

Categories and Subject Descriptors

D.1 [**Programming techniques**]: Concurrent programming—*parallel programming*; G.1.8 [**Numerical Analysis**]: Partial differential equations—*Hyperbolic equations*

General Terms

Algorithms, performance

*This work was supported by the Swedish Research Council through the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures.

1. INTRODUCTION

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. Some of the most widely spread general purpose frameworks for task parallel programming are OmpSs [3] and StarPU [1]. The research on features of task parallel programming is active, and in two local efforts, we explore alternative formulations. In [9, 8] we employ data versions for dependency tracking, instead of the more common directed acyclic graphs (DAGs). The programming model in [6] combines abstractions for both data and work partitioning.

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 full partial differential equation (PDE) solver, which generates sparse unstructured matrices. The main operations of the solver are sparse matrix-vector multiplications which become bandwidth-bound. With a straightforward implementation they are expected to scale badly on multicores.

The numerical method, RBF-FD, that we are targeting is relatively new and there is no available parallel software. Therefore, we have a double interest in showing that RBF-FD methods can be implemented efficiently in parallel, as well as showing that task parallel approaches can be successful also for, from the multicore point of view, very challenging scientific applications. A parallel implementation (not using tasks) of an RBF-FD method for the SWE on multiple CPUs and GPUs can be found in [2], and a GPU implementation of a global RBF method is described in [7].

2. THE SHALLOW WATER PROBLEM

The SWE are non-linear partial differential equations (PDE) that describe flow for example in the atmosphere. The Cartesian form of the SWE is 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 is the Coriolis force, h is the geopotential height, and g is the gravity.

Whenever a new method or implementation is proposed for climate simulations involving the SWE, it is required to pass a number of SWE benchmark problems [11]. Figure 2, which is an illustration from [4] shows the solution to the test case "Flow over an isolated mountain".

In global climate simulations, the computational domain is the surface of the earth. The equations can be formulated in spherical coordinates, which, however, introduces unphysical singularities at the poles. We choose instead to work in Cartesian coordinates, which means that the equations need to be projected onto the curved surface. An approach for this was derived in [5] and used also in [4].

3. THE NUMERICAL METHODS

Many numerical methods that are used for solving PDEs are mesh-based. However, the surface of a sphere cannot be covered by a uniform or regular mesh that is singularity-free. This makes meshfree methods an attractive alternative for simulations over the earth. Here we use the RBF-FD method from [4], which relies only on scattered nodes, and allows for local refinement.

Assume there are $N > n$ scattered nodes. A differential operator D is approximated at the location x_c by using a weighted combination of the function values u_k, $k = 1, \ldots, n$ at the n nearest neighboring nodes. That is,

$$(Du)\big|_{x=x_c} \approx \sum_{k=1}^{n} w_k u_k, \tag{1}$$

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 $u(x) \approx \sum_{k=1}^{n} \lambda_k \phi(\|x - x_k\|) \equiv \sum_{k=1}^{n} \lambda_k \phi_k(x)$ yields the following linear system of equations for the weights

$$\begin{bmatrix} \phi_1(x_1) & \phi_1(x_2) & \cdots & \phi_1(x_n) \\ \phi_2(x_1) & \phi_2(x_2) & \cdots & \phi_2(x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_n(x_1) & \phi_n(x_2) & \cdots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} D\phi_1(x_c) \\ D\phi_2(x_c) \\ \vdots \\ D\phi_n(x_c) \end{bmatrix}.$$

Figure 1 provides a graphical representation of one differentiation stencil. To solve the actual PDE, the stencils are used for approximation of the spatial PDE operator at each node point and the results are assembled into a global differentiation matrix. 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. However, stability cannot be guaranteed in the scattered node setting and therefore a hyperviscosity operator of order four is also incorporated into the scheme. This represents a small amount of diffusion, which acts as stabilizer without significantly altering the solution values.

4. THE SEQUENTIAL MATLAB CODE

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

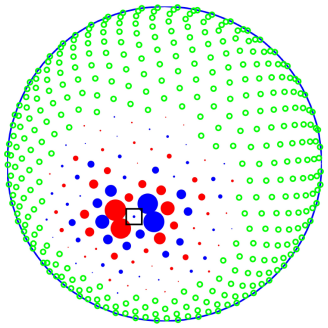

Figure 1: This is an illustration of a 75 node stencil. The differential operator is evaluated at the node marked with a square. The size of the markers indicates the magnitude of the stencil weights and the color indicates the signs. The nodes marked with green rings (further away) are not included in the stencil.

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

The main elements of the code consist of first setting up the necessary matrices and then performing the time stepping in a loop as shown in the listing below.

```
% 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
```

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.

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, to parallelize this code efficiently amounts to handling sparse unstructured matrix-vector multiplications, which is a classical example of bandwidth bound operations that are hard to get to scale on multicore processors.

5. THE PARALLEL C++ CODE

We consider the time-stepping loop only and start by performing basic optimizations of the sequential code. For example, different differentiation matrices that have the same access patterns in the algorithm are stored together in the same structure. Furthermore, by noting that each instance of the solution value is of length four (u, v, w, h), we were able to employ AVX (Advanced Vector Extensions to the

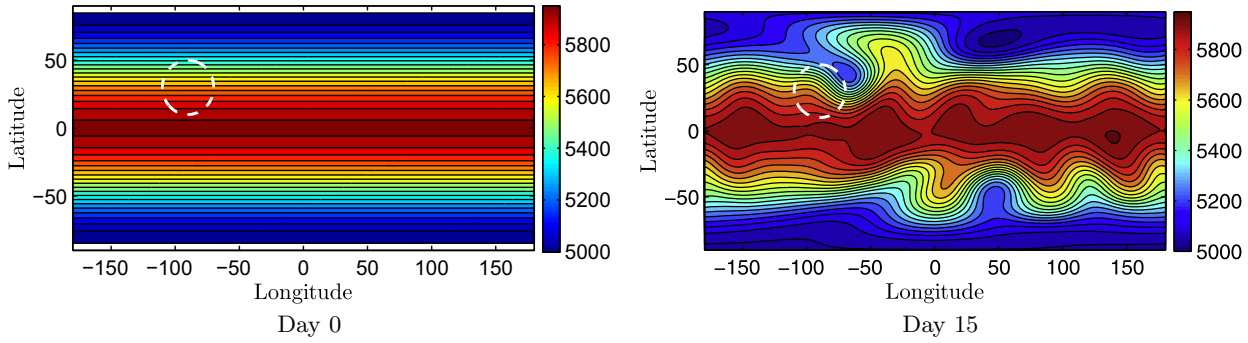

Figure 2: A solution to the shallow water test case "Flow over an isolated mountain". Left: The initial condition with laminar flow. Right: After two weeks, the mountain (dashed circle) has caused global scale perturbations in the flow.

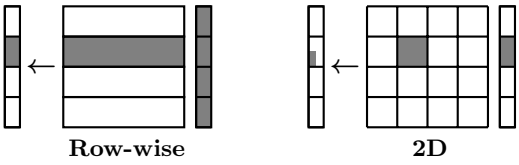

Figure 2: Two choices of blockings to divide the algorithm into tasks. The data that a task needs to touch is shaded.

x86 instruction set) SIMD instructions in the sparse matrix-vector multiplications (SpMV), which are significantly more efficient than performing four single instruction operations. The overall speedup from the MATLAB implementation to the sequential optimized C++ code was 3.3 times.

For the parallel implementation, we use the SuperGlue library [9, 8] for dependency-aware task-based parallel programming, which has been developed by the first author. In the SuperGlue run-time system, tasks are scheduled dynamically. This reduces the burden of the programmer significantly, it allows for flexibility in where tasks are executed, and the run-time overhead has shown to be negligible. 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, which could be described as tasks depending on futures, then when the future is realized, the task can execute. It should be noted that we do not duplicate data. The versions replace each other as they become available. Other properties of the SuperGlue implementation is that dependencies are deduced at run-time, there is one ready task queue per worker thread, load-balancing is achieved by task-stealing, and waiting tasks are queued at the required data.

A computational problem can be divided into tasks in different ways. Too small tasks make the overhead from scheduling visible, whereas too large tasks are hard to schedule efficiently and may lead to reduced parallelism. Figure 2 shows two alternative ways of blocking the matrix in order to define tasks. The choice of blocking determines which data each task needs to access, which in turn has impact on the performance. The row-wise blocking has the advantage of producing tasks with an equal amount of work, since the number of nonzeros in each row equals the stencil size. The task 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. When the blocks are small enough, we can use 16 bit indices instead of 32 bit indices within the blocks, to save memory bandwidth.

6. EXPERIMENTAL RESULTS

We present experimental results for two different problem sizes $N = 6400$ and $N = 25\,600$. The first problem should be considered as very small, which makes it even more difficult to run efficiently in parallel. In both cases, the stencil size is $n = 31$. The simulation is carried out for one day (real time), which corresponds to 144 time steps with 10 minute steps. The actual execution time is a few seconds. Figure 3 shows a section of the execution traces for each problem. For the very small problem, the best result is achieved when using row blocking, with the same number of block rows as the number of cores. This results in a very regular schedule even though it is dynamically created. For the larger problem size, the 2D-blocking with more diagonal blocks than cores is most successful. This results in a more chaotic schedule. However, in both cases, the achieved parallelism is high. That is, the schedule is tight and the amount of idle time is small, less than 8% for the small problem and less than 5% for the large problem.

Even though the scheduling looks very good, we need to look at the actual numbers to find out if the algorithm scales. Table 1 shows the speedup over the serial C++ code. For the small problem, the result for 6 cores is really good, but the problem is too small to run on a larger system than that. The larger problem uses more memory, hence the lower speedup at 6 cores, but it scales better.

As discussed earlier, the main reason we do not get linear speedup is the bandwidth contention. To understand in more detail how this affects performance, we break down the execution in different parts and then compute the total time spent in each part in the sequential case and in the parallel case. Figure 4 shows the increase in computational time

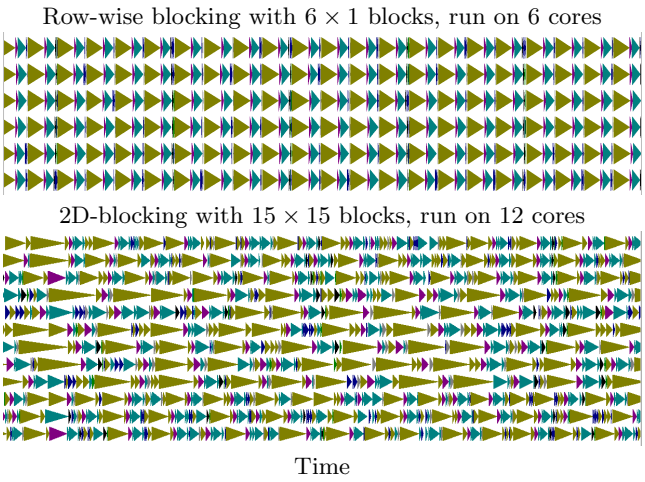

Figure 3: Parts of the execution traces for two different examples. Each triangle represents a task. Different colors indicate different types of tasks.

Table 1: Scaling at different problem sizes. Speedup, percentage of linear speedup, and best block configuration.

Cores	$N = 6400$			$N = 25600$		
1	1.0	100.0%	(1 × 1)	1.0	100.0%	(1 × 1)
6	5.2	86.7%	(6 × 1)	4.5	75.2%	(6 × 1)
12	5.9	49.2%	(12 × 1)	6.3	52.5%	(15 × 15)

for different configurations, and in fact, this reflects the increased time to access memory when several cores are using the memory bandwidth concurrently. The two main parts of the execution time represent the application of the differentiation matrices and the hyperviscosity operator. The latter is computationally heavier and therefore scales slightly better.

How to schedule in order to decrease the resource contention is discussed in [10]. 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.

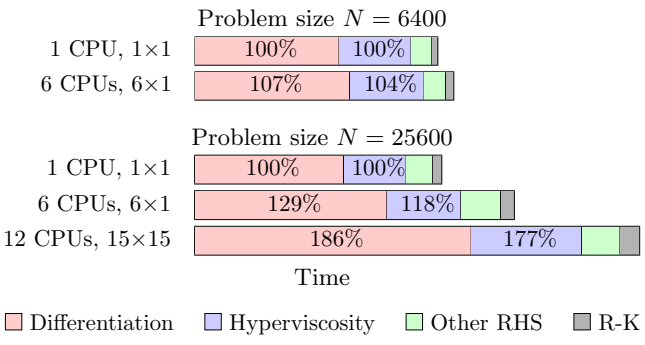

Figure 4: The total time spent in the different computational tasks. The increase in time in the parallel cases is due to resource contention.

7. CONCLUSIONS

We have shown that we can use our task based framework to parallelize a real application problem solved by the RBF-FD method, and that we can achieve reasonable speedup even though the problem is sparse and bandwidth limited.

Compared with the original MATLAB code we have achieved a speedup of more than 20 times through the combination of code optimizations (3.3×) and parallelization (6.3×). Future work involves much larger problems, distributed memory systems, and hybrid parallelizations.

8. REFERENCES

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

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

[3] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[4] N. Flyer, E. Lehto, S. Blaise, G. B. Wright, and A. St-Cyr. A guide to RBF-generated finite differences for nonlinear transport: shallow water simulations on a sphere. *J. Comput. Phys.*, 231(11):4078–4095, 2012.

[5] N. Flyer and 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):1949–1976, 2009.

[6] E. H. Rubensson and E. Rudberg. Chunks and tasks: a programming model for parallelization of dynamic algorithms. arXiv:1210.7427 [cs.DC], 2012.

[7] J. Schmidt, C. Piret, N. Zhang, B. J. Kadlec, D. A. Yuen, Y. Liu, G. B. Wright, and E. O. D. Sevre. Modeling of tsunami waves and atmospheric swirling flows with graphics processing unit (GPU) and radial basis functions (RBF). *Concurrency Computat. Pract. Exper.*, 22:1813–1835, 2010.

[8] M. Tillenius. *Leveraging Multicore Processors for Scientific Computing*. Licentiate thesis, Department of Information Technology, Uppsala University, Sept. 2012.

[9] M. Tillenius and 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.

[10] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. Resource aware task scheduling. In *Proc. HiPEAC: PARMA Workshop*. ACM Press, New York, 2013, 6 pp.

[11] D. L. Williamson, J. B. Drake, J. J. Hack, R. Jakob, and P. N. Swarztrauber. A standard test set for numerical approximations to the shallow water equations in spherical geometry. *J. Comput. Phys.*, 102(1):211–224, 1992.