Wavelet-Based Adaptive Solvers on Multi-core Architectures for the Simulation of Complex Systems

Diego Rossinelli, Michael Bergdorf, Babak Hejazialhosseini, and Petros Koumoutsakos

Chair of Computational Science, ETH Zürich, CH-8092, Switzerland

Abstract. We build wavelet-based adaptive numerical methods for the simulation of advection dominated flows that develop multiple spatial scales, with an emphasis on fluid mechanics problems. Wavelet based adaptivity is inherently sequential and in this work we demonstrate that these numerical methods can be implemented in software that is capable of harnessing the capabilities of multi-core architectures while maintaining their computational efficiency. Recent designs in frameworks for multi-core software development allow us to rethink parallelism as task-based, where parallel tasks are specified and automatically mapped into physical threads. This way of exposing parallelism enables the parallelization of algorithms that were considered inherently sequential, such as wavelet-based adaptive simulations. In this paper we present a framework that combines wavelet-based adaptivity with the task-based parallelism. We demonstrate good scaling performance obtained by simulating diverse physical systems on different multi-core and SMP architectures using up to 16 cores.

1 Introduction

The physically accurate simulation of advection dominated processes is a challenging computational problem. Advection is the main driver in the simulation of fluid motion in computational fluid dynamics (CFD), or in the animation of complex geometries using level sets in computer graphics. The difficulty arises from the simultaneous presence of a broad range of length scales (*e.g.* fine geometric features of a 3D model), and their interactions. Presently, the workhorse approach in simulating multiscale flow phenomena such as turbulent flows are Direct Numerical Simulations (DNS) [1] which use large uniform grids to resolve all scales of the flow. Large DNS calculations are performed on massively-parallel computing architectures and compute the evolution of hundreds of billions of unknowns [2,3].

DNS is not a viable solution for the simulation of flows of engineering interest [4] albeit the continuous increase in available high performance computers. Adaptive simulation techniques, such as Adaptive Mesh Refinement (AMR) [5], or multiresolution techniques using Wavelets [6,7,8] have been introduced in order to locally adjust the resolution of the computational elements to the different length scales emerging in the flow field. Wavelets have been employed largely for conservation laws but they have also recently been coupled with level sets for geometry representation [9].

In order to create simulation tools that go beyond the state-of-the-art, these adaptive numerical methods must be complemented with massively-parallel and multi-level parallel scalability. Hardware-accelerated wavelet transforms have a long history. The first GPU-based 2D Fast Wavelet Transform was introduced by Hopf and Ertl in 2000 [10]. Since then many different parallel implementations have been proposed and evaluated [11] both on multi-core architectures such as the Cell BE [12], and GPUs [13]. These efforts however, have been restricted to the *full* FWT. The effective parallel implementation of *adaptive* wavelet-based methods is however hindered by their inherently sequential nested structure. This difficulty limits their effective implementation on multi-core and many-core architectures and affects the development of per-thread-based wavelet software that can have both high performance and abstracts from a specific hardware architecture. An alternative approach for the effective implementation of wavelets for flow simulations is to specify parallel tasks instead of threads and then let an external library map logical tasks to physical threads according to the specific hardware architecture. Another emerging need in developing simulation software is the necessity to specify more than one granularity level for parallel tasks in order to combine multi-core computing with many-core accelerators such as GPUs.

In this paper we present multiresolution wavelet based computational methods designed for different multi-core architectures. The resulting computational framework, is flexible and can be used to simulate different natural phenomena such as transport of interfaces, reaction-diffusion, or compressible flows.

The paper is organized as follows: first we introduce briefly wavelets, followed by the presentation of our algorithms for the discretization of partial differential equations. We discuss how the framework can be employed for multi-core and SMP machines. The performance of the proposed simulation tools is then demonstrated on computations of level set based advection of interfaces and two dimensional compressible flows.

2 Wavelet-Based Grids

Biorthogonal wavelets can be used to construct multiresolution analysis (MRA) of the quantities being represented and they are combined with finite difference/volume approximations to discretize the governing equations. Biorthogonal wavelets are a generalization of orthogonal wavelets and they can have associated scaling functions that are symmetric and smooth[14]. Biorthogonal wavelets introduce two pairs of functions, ϕ, ψ for synthesis, and $\tilde{\phi}, \tilde{\psi}$ for analysis.

There are four refinement equations:

$$\phi(x) = \sum_m h_m^S \phi(2x+m), \quad \psi(x) = \sum_m g_m^S \phi(2x+m), \tag{1}$$

$$\tilde{\phi}(x) = \sum_m h_m^A \tilde{\phi}(2x+m), \quad \tilde{\psi}(x) = \sum_m g_m^A \tilde{\phi}(2x+m). \tag{2}$$

Given a function f, we compute $c_k^0 = \left\langle f, \tilde{\phi}_k^0 \right\rangle, d_k^l = \left\langle f, \tilde{\psi}_k^l \right\rangle$ and reconstruct f as:

$$f = \sum_k c_k^0 \phi_k^0 + \sum_{l=0}^{\infty} \sum_k d_k^l \psi_k^l. \tag{3}$$

The Fast Wavelet Transform (FWT) uses the filters h_n^A, g_n^A (analysis), whereas h_n^S, g_n^S are used in the inverse fast wavelet transform (synthesis):

$$c_k^l = \sum_m h_{2k-m}^A c_m^{l+1}, \quad d_k^l = \sum_m g_{2k-m}^A c_m^{l+1}, \tag{4}$$

$$c_k^{l+1} = \sum_m h_{2m-k}^S c_m^l + \sum_m g_{2m-k}^S d_m^l. \tag{5}$$

Active scaling coefficients Using the FWT we can decompose functions into scaling and detail coefficients, resulting in a MRA of our data. We can now exploit the scale information of the MRA to obtain a compressed representation by keeping only the coefficients that carry significant information:

$$f_{\geq\varepsilon} = \sum_k c_k^0 \phi_k^0 + \sum_l \sum_{k:|d_k|>\varepsilon} d_k^l \psi_k^l, \tag{6}$$

where ε is called threshold and is used to truncate terms in the reconstruction. The scaling coefficients c_k^l needed to compute $d_k^l : |d_k^l| > \varepsilon$, and the coefficients at coarser levels needed to reconstruct c_k^l are the active scaling coefficients. The pointwise error introduced by this thresholding is bounded by ε. Each scaling coefficient has a physical position. Therefore the above compression results in an adapted grid $\mathcal{K}$, where each grid node will represent an active scaling coefficient, representing a physical quantity. We then discretize the differential operators by applying standard finite-volume or finite-difference schemes on the active coefficients. One way to retain simple operations is to first create a local, uniform resolution neighborhood for a grid point, and then apply the operator on it. Such operators can be viewed as (non-linear) filtering operations on uniform resolution grid points, formally:

$$F(\{c_{k'}^l\}_{k'\in\mathbb{Z}})_k = \sum_{j=s_f}^{e_f-1} c_{k+j}^l \beta_j, \quad \beta_j^l \text{ function of } \{c_m^l\} \tag{7}$$

where $\{s_f, e_f - 1\}$ is the support of the filter in the index space and $e_f - s_f$ is the number of non-zero filter coeffcients $\{\beta_j\}$. In order to perform uniform resolution filtering we need to temporarily introduce artificial auxiliary grid points, so-called "ghosts". We need to ascertain that for every grid point k in the adapted set $\mathcal{K}$, its neighborhood $[k - k_f, k + k_f]$ is filled with either other points k' in $\mathcal{K}$ or ghosts g. Using this set of ghosts, which can be precomputed and stored or computed on the fly (see section 2), we are now able to apply the filter F to all points k in $\mathcal{K}$. The ghosts are constructed from the active scaling coefficients as a weighted average $g_i^l = \sum_l \sum_j w_{ijl} c_j^l$, where the weights w_{ijl} are provided by the refinement equations (4). It is convenient to represent the construction of a ghost as $g_i = \sum_j w_{ij} p_j$, where i is the identifier for the ghost and j represents the identifier for the source point p_j which is an active scaling coefficient in the grid. Calculation of the weights $\{w_{ij}\}$ is done by traversing a dependency graph associated with the refinement equations (see Figure 1). This operation can be expensive for two reasons: firstly if the dependency graph has loops, we need to solve a linear system of equations to compute $\{w_{ij}\}$, secondly, the complexity of calculating the values $\{w_{ij}\}$ scales with the number of dependency edges *i.e.* it grows exponentially with the jump in level between a point k and its neighbors in the adapted grid. The wavelets incorporated into the present framework are based on subdivision schemes and either interpolate function values or their averages [15]. Due to their construction, these wavelets do not exist explicitly in an analytic form. The primal scaling function can however be constructed by a recursive scheme imposed by its refinement equation. By virtue of their interpolatory property, the ghost reconstruction is straightforward (the ghost dependency graphs do not contain loops) and leads to very efficient reconstruction formulae.

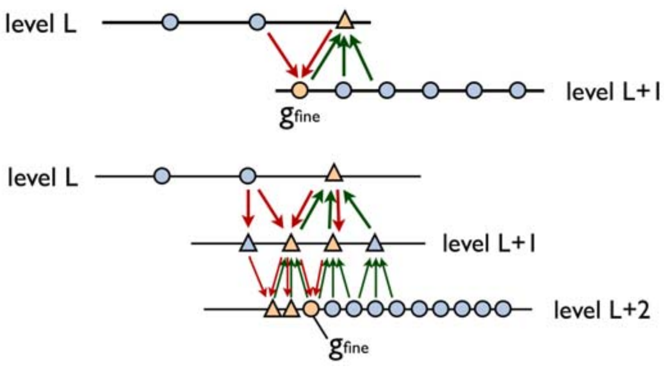

Fig. 1. Dependency graph for the ghost reconstruction g_{fine} with a jump in resolution of 1 (top) and 2 (bottom), for the case of a linear, non-interpolating, biorthogonal wavelet. The ghost generates secondary (triangles), temporary ghosts to calculate the needed weights and source points. The unknown ghosts are denoted in orange.

2.1 Using Wavelet-Blocks

The wavelet adapted grids for finite differences/volumes are often implemented with quad-tree or oct-tree structures whose leaves are single scaling coefficients. The main advantage of such fine-grained trees is the very high compression rate which can be achieved when thresholding individual detail coefficients. The drawback on the other hand is the large amount of sequential operations they involve and the number of indirections (or read instructions) they need in order to access a group of elements. Even in cases where we only compute without changing the structure of the grid, these grids already perform a great number of neighborhood look-ups. In addition, operations like refining or coarsening single grid points have to be performed and those operations are relatively complex and strictly sequential. To expose more parallelism and to decrease the amount of sequential operations per grid point, the key idea is to simplify the data-structure. We address this issue by decreasing the granularity of the method at the expense of a reduction in the compression rate. We introduce the concept of block of grid points, which has a coarser granularity by one or two order of magnitude with respect to the number of scaling coefficients in every direction *i.e.* in 3D, the granularity of the block is coarser by 3 to 5 orders of magnitude with respect to a single scaling coefficient.

Structure of the Wavelet-Blocks. A block contains scaling coefficients which reside on the same level and every block contains the same number of scaling coefficients. The grid is then represented with a tree which contains blocks as leaves (see Figure 2). The blocks are nested so that every block can be split and doubled in each direction and blocks can be collapsed into one. Blocks are interconnected through the ghosts. In the physical space the blocks have varying size and therefore different resolutions. The idea of introducing the intermediate concept of a block provides a series of benefits. The first benefit is that tree operations are now drastically accelerated as they can be performed in $\log_2(N^{1/D}/s_{block})$ operations instead of $\log_2(N^{1/D})$, where N is the total number of active coefficient, D the dimensions in consideration and s_{block} is the block size. The second benefit is that the random access at elements inside a block can be extremely efficient because the block represents the atomic element. Another very important advantage is the reduction of the sequential operations involved in processing a local group of scaling coefficients. We consider the cost c (in terms of memory access) of filtering per grid point with a filter of size KD ($c = KD$). In a uniform resolution grid the data access operations to perform the computation is proportional to KD. For a fine-grid tree the number of accesses is proportional to $c = KD\log_2(N^{1/D})$. Using the wavelet-blocks approach and assuming that s_{block} is roughly one order of magnitude larger than K, the ratio of ghosts needed to perform the filtering for a block is

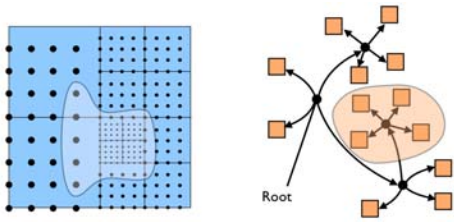

Fig. 2. The masked (left) region identifies the source points of the grid used to construct the ghosts, which are used to collapse/split the blocks in the tree (right masked region)

$$
\begin{aligned}
r &= \frac{(s_{block} + K)^D - s_{block}^D}{s_{block}^D} \approx D\frac{K}{s_{block}} \\
c(K) &= (1-r)KD + rKD\big(\log_2(N^{1/D}/s_{block})\big) \\
&= KD + KDr\big(\log_2(N^{1/D}/s_{block}) - 1\big). \qquad (8)
\end{aligned}
$$

We assume that none of the blocks is overlapping in the physical space. We improve the efficiency of finding the neighbors of a block by constraining the neighbors to be adjacent neighbors. Because of this constraint, the additional condition $s_{block} \geq w_{stencil}2^{L_j}$ appears, where $w_{stencil}$ is the filter size and L_j is the maximum jump in resolution.

Local compression and refinement of the grid. The re-adaptation of the grid is achieved by performing elementary operations on the blocks: block splitting to locally refine the grid, and block collapsing to coarsen the grid. Block splitting and block collapsing are triggered by logic expressions based on the thresholding of detail coefficients residing inside the block. To compute the detail coefficients of the block (i_b, l_b) we perform one step of the fast wavelet transform :

$$
d_k^{(l_b-1)} = \sum_m g_{2k-m}^A c_m^{l_b}, \quad k \in \{\frac{i_b \cdot s_{block}}{2}, \frac{(i_b+1)\cdot s_{block}}{2} - 1\} \qquad (9)
$$

Note that k is inside the block but m can be outside the block (see Figure 2). In the one dimensional case, when we decide to collapse some blocks into one, we just have to replace the data with the scaling coefficient at the coarser level:

$$
c_k^l = \sum_m h_{2k-m}^A c_m^{l+1} \quad k \in \{\frac{i_b}{2}\cdot s_{block}, (\frac{i_b}{2}+1)\cdot s_{block} - 1\} \qquad (10)
$$

The complexity of a single FWT step, which consists of computing all the details and the scaling coefficients, is approximately $D \cdot (c \cdot s_{block}/2)^D$ with $c = c(K)$ defined as in Equation 8 and K is the maximum stencil size of the FWT filters. If we consider the block collapse as the key operation for compressing, we can consider the block splitting as the key to capture smaller emerging scales. In the case where we split one block into two blocks, we perform one step of the inverse wavelet transform on the block (i_b, l_b):

$$
c_k^{l_b+1} = \sum_m h_{2m-k}^S c_m^{l_b}, \quad k \in \{2i_b \cdot s_{block}, 2(i_b+1)\cdot s_{block} - 1\} \qquad (11)
$$

Time refinement. Small spatial scales are often to be associated with small scales in time, especially if the equation under investigation contains nonlinear terms. The wavelets-adapted block grid can exploit time refinement [16,17]. Depending on the case, this can already accelerate the simulation by one or two orders of magnitude. There are however two shortcomings in exploiting different time scales. The first drawback comes from the fact that the processing of the grid blocks has to be grouped by their time scales and no more than one group can be processed in parallel. The second drawback is the increased complexity of reconstructing ghosts; when dealing with different time scales, ghosts have to be interpolated also in time.

3 Wavelet Algorithms for Multi-core Computing

Our first objective in designing the framework is to expose enough load balanced parallel tasks so that the amount of tasks is greater than the number of cores. The second objective is to express nested parallelism inside the parallel tasks, so that the idling cores could help in processing the latest tasks. Once these are met, we expect an external library to map our logical tasks into physical threads based on the specific SMP architecture. Grid adaptation is performed in two steps: before computing we refine the grid in order to allow the emergence of new smaller scales [18]. After the computation we apply compression based on thresholding to retain only blocks with significant detail. Figure 3.2 shows the program flow for the refinement stage. An important aspect of the framework is its ability to retain control over the maximum number of scaling coefficients and therefore, the memory; instead of keeping the threshold fixed, we can also control the number of scaling coefficients with an upper bound by changing the compression threshold.

3.1 Producing Many Parallel Tasks

Due to the design of the framework, the number of parallel tasks scales with the number of blocks. Parallel tasks can be grouped into *elementary* and *complex* tasks. A task of the *elementary* type operates exclusively on data inside the block, *i.e.* it is purely local. These tasks are inherently load-balanced, since every task operates on the same number of points. A *complex* task, however, involves the reconstruction of ghosts and block splitting, the FWT, or evaluating finite-difference operators. The cost of reconstructing a ghost is not constant and can be expensive as it depends on the structure of the tree. The efficiency of the ghost reconstruction is therefore paramount for load balance.

3.2 Fast Ghost Reconstruction

We recall that ghosts are computed as weighted averages of the scaling coefficients in its neighborhood. We call n_i the number of source points/weights that we need to construct the ghost g_i from:

$$g_i = \sum_j w_{ij} p_j, \; n_i = |\{j \in \mathbb{Z} : w_{ij} \neq 0\}|. \tag{12}$$

We can split the ghost construction into two stages: find the contributing source points/weights and evaluate the weighted average. The first stage will produce an array of weights and indices associated with grid points, which is successively sent as input to the second stage where the array will be used to produce the weighted summation. We call this array the instruction array. We note immediately that the first stage is computationally expensive because it needs recursive data structures. If we assume that we have to construct the same ghost several times, it is meaningful to store the instructions array and successively perform only the second stage, which is fast. For the same ghost of a block, we can use the same instructions array as long as the neighbor blocks do not change. The main shortcoming of this strategy is its memory consumption, as the instructions arrays tend to be big. However, the entries of the instruction arrays of two close ghosts are likely to be very similar. This coherence between instructions arrays allows us to compress them before they are saved. To compress them, we re-organize every instruction array of a block into streams of w_{ij}'s, j's and n_i's, and then pass them to an encoder (using either quantization based encoding or Huffman encoding). We noticed that overall, the compression factor of the instruction array varies between 1.5 and 5, and the compression of n_i is between 10 and 20.

4 Results

In this section we present the tools used to develop the framework and the performance of our algorithms on a set of benchmark problems. We then present a study of the effect of different block sizes, types of wavelets and the number of threads.

Software Details. The framework was written in C++ using generic programming and object oriented concepts. In our framework, Intel's Threading Building Blocks (TBB)[19] library was used to map logical tasks to physical threads. This library completely fulfills our requirements stated in Section 3, as it allows to specify task patterns and enables to easily express nested parallelism inside tasks. Furthermore TBB provides a very useful set of templates for programming in parallel which are independent of to the simulated phenomena. Most of the TBB calls were to `parallel_for`, which exploits recursive parallelism. We employed the `auto_partitioner` for an automatic grain size through the 'reflection on work stealing' [20], to dynamically change the grain size of tasks. In fact, the most efficient grain size varies with respect to the block size and the wavelet order but also with the number of cores.

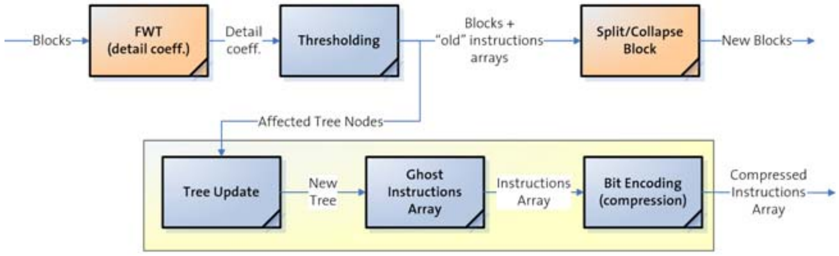

Fig. 3. Stages of refinement/compression of the grid. The region depicted in yellow is performed sequentially but executed in parallel for different blocks. The orange stages need to be optimally parallelized as they involve intensive processing of the data.

Computer architectures. Simulations were done on the ETH Zurich central cluster "Brutus", a heterogeneous system with a total of 2200 processors in 756 compute nodes. The compute nodes that meet our needs *i.e.* multi-threaded nodes are the eight so-called *fat* nodes each with eight dual-core AMD Opteron 8220 CPU's (2.8 GHz) and 64-128 GB of RAM. These nodes are inter-connected via a Gigabit Ethernet backbone. The Intel C++ compiler v10.1.015 (64-bit) was used along with the Intel TBB library (64-bit) to build the three different test cases used in this work. For some of the cases, we have also used an Intel Xeon 5150 machine, with 2 dual core processors with a core speed of 2.66 GHz and 4GB RAM.

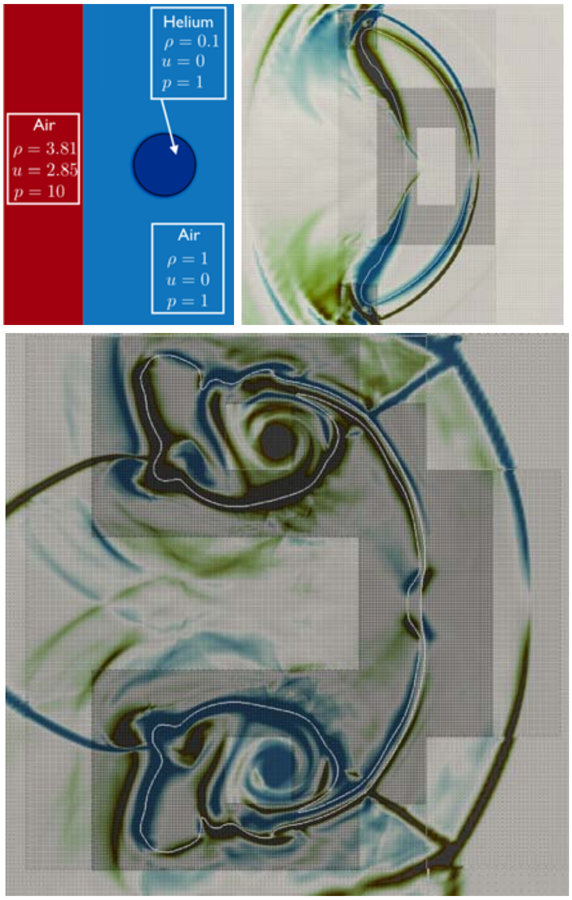

Fig. 4. Initial condition (top left) and zooms of the simulation to the high resolution region close to the bubble at different times (chronologically: top right, bottom). Blue/green depicts high positive/negative vorticity, whereas the interface of the bubble is depicted in white.

4.1 Benchmark Problems

Two-dimensional Compressible Flows. A common benchmark problem for compressible flow simulations consists of a circular bubble filled with helium, surrounded by air being hit by a shock wave (see Figure 4 top left). The shock deforms the bubble and leads to instabilities and fine structures on the helium-air interface. The governing equations are:

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) &= 0 \\ \frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u} + p\underline{\underline{I}}) &= 0 \\ \frac{\partial \rho E}{\partial t} + \nabla \cdot ((\rho E + p)\mathbf{u}) &= 0 \end{aligned} \tag{13}$$

with ρ, μ, $\mathbf{u}$, p and E being the fluid density, viscosity, velocity, pressure and total energy respectively. This system of equations is closed with an equation of state of the form $p = p(\rho)$. In this test case, each grid point stored the values ρ, ρu, ρv, ρE. The simulation was performed using a maximum effective resolution of 4096×4096 grid points with third order average interpolating wavelets, a block size of 32, and a maximum jump in resolution of 2. Derivatives and numerical fluxes were calculated using 3rd order WENO [21] and HLLE [22] schemes respectively. Time stepping was achieved through a 3rd order low storage TVD Runge-kutta scheme. The thresholding was a unified thresholding based on the bubble level set and the magnitude of the velocity. The simulation was computed on the Xeon machine with 4 cores, with a strong speedup of 3.74, that decreases to 3.55 when there are less than 80 blocks. The compression rate varied between 10 and 50.

Three-dimensional Advection of Level sets. We simulated the advection of level sets in 3D based on the advection equation $\frac{\partial \phi}{\partial t} + \boldsymbol{u} \cdot \nabla \phi = 0$. Derivatives

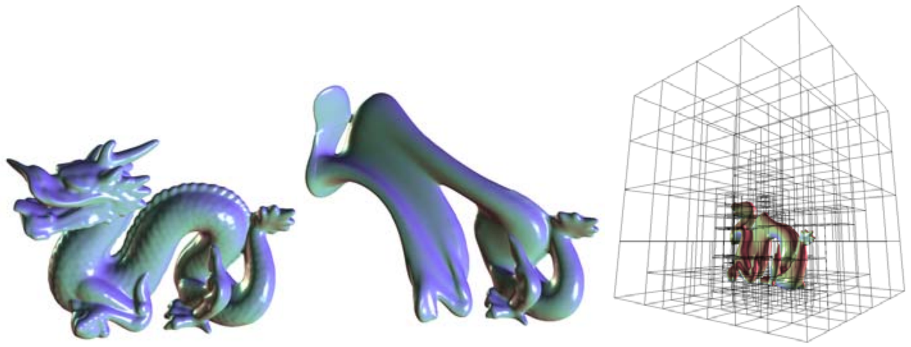

Fig. 5. Initial interface (left), advected with a velocity field of a vortex ring (middle) and distribution of the blocks in the grid (right)

and numerical fluxes were constructed using a 5th order WENO [21] and a Godunov flux respectively. Time integration was performed with a first-order time refinement, where given a fixed CFL number, each block is updated based on its own time step, *i.e.* for a block at level l we used a time step proportional to 2^{-l}. CFL number is defined as:

$$\mathrm{CFL} = \Delta t \frac{|\mathbf{u}|_{max}}{\Delta x_{\mathrm{local}}} \tag{14}$$

with $\Delta x_{\mathrm{local}} = 2^{-l} \frac{1}{\text{block size}}$.

The level set field was also re-distanced after each time step using the re-initialization equation $\frac{\partial \phi}{\partial t} + sgn(\phi_0)(|\nabla \phi| - 1) = 0$, proposed by Sussman, Smereka and Osher [23] with ϕ_0 being the initial condition of the re-initialization equation

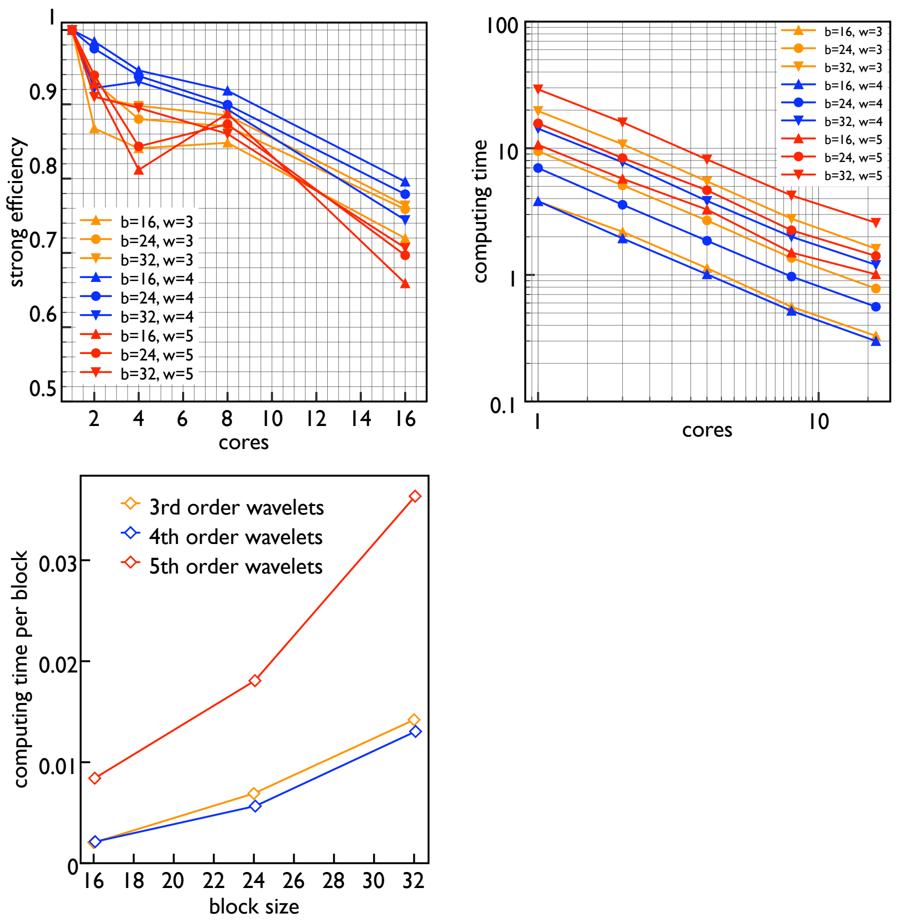

Fig. 6. Efficiency (left), computing times (middle) and time spent per block (right) for different interpolating/average interpolating wavelets and block sizes

which in general, does not have the desired signed distance property. For efficient level set compression we employed the detail-clipping trick introduced in [9]. As initial interface we used the dragon illustrated in Figure 5. Simulations were carried out for four different types of wavelets: 3rd and 5th order average interpolating wavelets, 2nd and 4th order interpolating wavelets, and for different block sizes ranging from 16 to 32 as well as for a range of cores from 1 to 16, while $\varepsilon_{\text{compression}}$ and $\varepsilon_{\text{refine}}$ were kept fixed. We measured the time spent in the computing stage. Figure 6 shows the strong efficiency obtained from these measurements. The largest decrease in performance is from 8 to 16 cores. In the case of 16 cores, the best result was achieved using a block size of 16 with the interpolating wavelet of order 4. With this settings we achieved a strong efficiency of 0.8, with a strong speedup of 12.75 with 16 cores. While with the fourth order interpolating wavelets we achieved a monotonic decreasing efficiency, the efficiency obtained with average interpolating wavelets of order 3 and 5 was better with 8 cores than with 4. We observe that the combination of fourth order interpolating wavelets with a block size of 16 also achieved the best timings (see Figure 6). Furthermore, we note that the lines do not cross: the change in the number of cores does not affect the ranking of the computing time for the different wavelets and block size. We also note that the second fastest setting was the third order average interpolating wavelets with a blocksize of 16, which did not show a good efficiency (0.72 with 16 cores). Regarding the time spent per block versus the block size, the best timing was achieved by the third order average interpolating wavelets. Theoretically, the time spent per block should increase roughly by a factor of 8 between a block size of 16 and 32. We measure an increase by a factor of about 7 for all three wavelets, meaning that for a block size of 16, the "pure computing" time per block was about 85% of the total time spent per block. We must however note that the time spent per block also includes some synchronization mechanism, which scales with the number of blocks which is high in the case of the third order average interpolating wavelets with a blocksize of 32 (Figure 6). Figure 7 depicts the time spent versus the

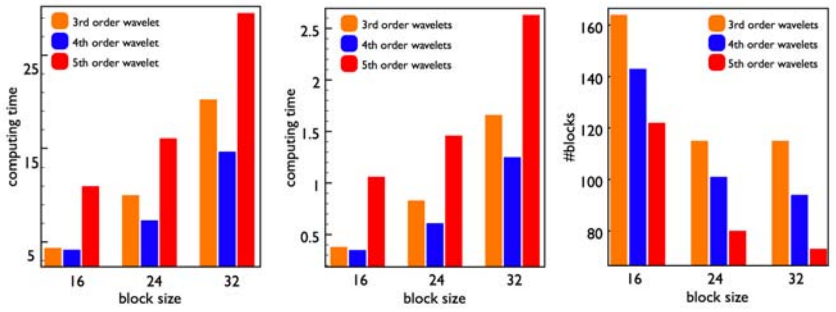

Fig. 7. Computing time versus different block sizes by using one core (left), 16 cores (middle) and the number of blocks (right)

block size with one core (left) and 16 cores (middle). We can see that with the third order wavelet we have a substantial decrease in performances with a block size of 32; this is due to the high number of blocks (Figure 6, right) and the cost associated with the reconstruction of the ghosts at the coarser resolutions.

5 Conclusions

We presented a wavelet-based framework for adaptive multiresolution simulations for solving time-dependent PDEs. The present framework demonstrates that it is possible to develop multiresolution simulation tools that can exploit the new multi-core technologies without degrading the parallel scaling. We have also shown the change in performance and compression rate based on the choice of the wavelets, the size of the block and the number of cores. We achieved the best performances in terms of scaling and timings using a block size of 16, the best wavelet was the fourth-order interpolating one, which gave a strong speedup of 12.75 of 16 cores. We observed that bigger block sizes can be efficiently coupled with high order wavelets, whereas smaller block sizes are more efficient when combined with small block sizes. We have also seen that the performance differences between combinations of block sizes and wavelets are independent of the number of cores, meaning that good efficiency is expected also for more than 16 cores. The current work includes an extension of the framework to support particle-based methods [24] and the incorporation of multiple GPUs as computing accelerators. On the application level we focus on extending the two-dimensional flow simulations to three-dimensional flows and on combining flow simulations and reaction-diffusion processes for adaptive simulations of combustion.

References

1. Ishihara, T., Gotoh, T., Kaneda, Y.: Study of High–Reynolds Number Isotropic Turbulence by Direct Numerical Simulation. Annual Review of Fluid Mechanics 41(1) (2009)
2. Yokokawa, M., Uno, A., Ishihara, T., Kaneda, Y.: 16.4–Tflops DNS of turbulence by Fourier spectral method on the Earth Simulator. In: Supercomputing 2002: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pp. 1–17. IEEE Computer Society Press, Los Alamitos (2002)
3. Chatelain, P., Curioni, A., Bergdorf, M., Rossinelli, D., Andreoni, W., Koumoutsakos, P.: Billion Vortex Particle direct numerical simulation of aircraft wakes. Comput. Methods Appl. Mech. Engrg. 197(13-16), 1296–1304 (2008)
4. Jiménez, J.: Computing high-Reynolds-number turbulence: will simulations ever replace experiments? Journal of Turbulence 4 (June 12, 2003); 5th International Symposium on Engineering Turbulence Modelling and Measurements, Mallorca, Spain, September 16-18 (2002)
5. Berger, M.J., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. J. Comput. Phys. 53(3), 484–512 (1984)
6. Harten, A.: Adaptive Multiresolution Schemes For Shock Computations. Journal of Computational Physics 115(2), 319–338 (1994)

7. Kevlahan, N.K.R., Vasilyev, O.V.: An adaptive wavelet collocation method for fluid-structure interaction at high reynolds numbers. SIAM Journal on Scientific Computing 26(6), 1894–1915 (2005)
8. Liu, Q., Vasilyev, O.: A Brinkman penalization method for compressible flows in complex geometries. Journal of Computational Physics 227(2), 946–966 (2007)
9. Bergdorf, M., Koumoutsakos, P.: A Lagrangian particle-wavelet method. Multiscale Modeling and Simulation 5(3), 980–995 (2006)
10. Hopf, M., Ertl, T.: Hardware accelerated wavelet transformations. In: Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym 2000, pp. 93–103 (2000)
11. Yang, L.H., Misra, M.: Coarse-grained parallel algorithms for multi-dimensional wavelet transforms. Journal of Supercomputing 12(1-2), 99–118 (1998)
12. Bader, D.A., Agarwal, V., Kang, S.: Computing discrete transforms on the cell broadband engine. Parallel Computing 35(3), 119–137 (2009)
13. Tenllado, C., Setoain, J., Prieto, M., Pinuel, L., Tirado, F.: Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting. IEEE Transaction on Parallel and Distributed Systems 19(3), 299–310 (2008)
14. Cohen, A., Daubechies, I., Feauveau, J.: Biorthogonal Bases of Compactly Supported Wavelets. Communication on Pure and Applied Mathematics 45(5), 485–560 (1992)
15. Donoho, D.: Interpolating wavelet transforms. Technical report, Preprint Department of Statistics, Stanford University (1992)
16. Ferm, L., Lötstedt, P.: Space–Time Adaptive Solution of First Order PDEs. J. Sci. Comput. 26(1), 83–110 (2006)
17. Jansson, J., Logg, A.: Algorithms and data structures for multi-adaptive time-stepping. ACM Transactions on Mathematical Software 35(3), 17 (2008)
18. Liandrat, J., Tchamitchian, P.: Resolution of the 1D regularized Burgers equation using a spatial wavelet approximation. Technical Report 90-83, 1CASE, NASA Contractor Report 18748880 (December 1990)
19. Contreras, G., Martonosi, M.: Characterizing and improving the performance of Intel threading building blocks. In: 2008 IEEE International Symposium on Workload Characterization (IISWC), Piscataway, NJ, USA, pp. 57–66. IEEE, Los Alamitos (2008)
20. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: 2008 IEEE International Symposium on Parallel & Distributed Processing, New York, NY, USA, vols. 1-8, pp. 598–605. IEEE, Los Alamitos (2008)
21. Harten, A., Engquist, B., Osher, S., Chakravarthy, S.: Uniformly high order accurate essentially non-oscillatory schemes.3 (Reprinted from Journal of Computational Physics, vol 71, p. 231 (1987). Journal of Computational Physics 131(1), 3–47 (Feburary 1997)
22. Wendroff, B.: Approximate Riemann solvers, Godunov schemes and contact discontinuities. In: Toro, E.F. (ed.) Godunov Methods: Theory and Applications, 233 Spring St, New York, NY 10013 USA, London Math Soc., pp. 1023–1056. Kluwer Academic/Plenum Publ. (2001)
23. Sussman, M., Smereka, P., Osher, S.: A Level Set Approach for Computing Solutions To Incompressible 2-Phase Flow. Journal of Computational Physics 114(1), 146–159 (1994)
24. Koumoutsakos, P.: Multiscale flow simulations using particles. Annual Review Of Fluid Mechanics 37, 457–487 (2005)