Teaching Parallel Programming Models on a Shallow-Water Code

Alexander Breuer, Michael Bader
Department of Informatics, Technische Universität München, Germany
Email: \{breuer,bader\}@in.tum.de

Abstract—We present a software package that supports teaching different parallel programming models in a computational science and engineering context. It implements a Finite Volume solver for the shallow water equations, with application to tsunami simulation in mind. The numerical model is kept simple, using patches of Cartesian grids as computational domain, which can be connected via ghost layers. The Finite Volume method is restricted to piecewise constant approximation in each grid cell, but the computation of fluxes between cells can be based on the simple Lax-Friedrichs method, as well as on versatile approximate Riemann solvers, which allows realistic simulations. We present how this code can be used to study parallelization with CUDA, MPI, OpenMP, and hybrid approaches – and is useful for both introductory lectures in parallel computing and more advanced courses.

Keywords—parallel computing; parallel programming models: MPI, OpenMP, CUDA; CSE education; shallow water equations

I. INTRODUCTION

Parallel platforms, from multicore processors via mid-size clusters to supercomputers, have become ubiquitous in scientific computing, such that parallel programming is now a key technology for computational science and engineering (CSE). Also, platforms with multi-level parallelism (clusters of shared-memory compute nodes, or clusters of GPUs, for example) and the use of hybrid parallel programming models are of increasing importance. As a consequence, academic education, especially in CSE and related disciplines, can no longer focus on a single programming model, but has to teach a larger range of parallel programming approaches – including established paradigms as message passing (with MPI as dominant model) or shared-memory programming (using OpenMP, for example) as well as more recent approaches for parallel accelerator hardware (such as CUDA or OpenCL for GPUs).

In this paper, we present a simple, yet versatile software package for teaching different parallel programming models on a concrete simulation scenario. It provides a Finite Volume solver for the shallow water equations, with tsunami simulation as intended application scenario. Implemented in C++, because of the wide-spread availability of parallel programming models, the package was designed to study different paradigms for parallel programming, and to compare them in the same problem setting. It also supports a wide range of teaching levels – from introductory courses in parallel computing, where the focus is “to make it work” (loop parallelism in OpenMP, data exchange in MPI, executing CUDA kernels) via testing and comparing different options for parallelization up to advanced courses that include hardware-aware optimization of serial and parallel performance (vectorization, load balancing, overlapping communication and computation, etc.). By focusing on the shallow water equations and tsunami simulation, we address a problem of high relevance that should be more motivating than to just address so-called “toy problems”.

II. SHALLOW WATER MODEL

The proposed two-dimensional Shallow Water Equations (SWEs) represent an intuitive system of equations, corresponding to a very general class of hyperbolic partial differential equations, namely the class of nonlinear systems of conservation laws with an optional source term:

\[
\begin{bmatrix}
    h \\
hu \\
hv
\end{bmatrix}_t + \begin{bmatrix}
    hu \\
hu^2 + \frac{1}{2}gh^2 \\
huv
\end{bmatrix}_x + \begin{bmatrix}
    hv \\
hu^2 + \frac{1}{2}gh^2
\end{bmatrix}_y = S(t, x, y).
\]

(1)

In the above formulation, \( t \) is the time, \( h \) the height of the water column, \( u \) and \( v \) represent the velocities in spatial \( x \)- and \( y \)-direction and \( g \) is the gravitational constant (here, \( g = 9.81 \text{ m/s}^2 \)). The term \( S(t, x, y) \) accounts for the contribution of a source term, which might include bathymetry or additional forces due to friction or coriolis effects for example. In the current version of the code, we restrict ourselves to bathymetry source terms.

For courses in computer or computational science, the shallow water model has the advantage of an intuitive problem setting, and comparably easy motivation of the modeling (resulting from classical conservation laws). From a mathematical point of view the most important features of nonlinear hyperbolic systems can already be studied in this model. Also, despite the simplicity of the formulation (1), the numerical solution of the equations is a challenging task of ongoing research [1]. The solution strategy, using Riemann-problems (see Sec. III) and the underlying integral formulations, learned on the SWEs can be easily transferred to other important systems, such as the Euler equations or the elastic wave equations.

Nonetheless, the most important factor for choosing the SWEs is their motivational character, because they are the basic equations for the tsunami simulation ([1], [2], [3], e.g.).

Accepted for Proceedings of the ISPDC 2012, to be published by IEEE.
III. NUMERICAL MODELS

The numerical formulation is kept as simple as possible, in order to keep the teaching focus on parallelization, and allow students to work quickly into the method and implementation. The discretization is based on a Cartesian grid with piecewise constant cell data, and explicit time stepping is used. In that aspect, the package takes up the approach used by Griebel et al. [4] for their code Nast (for the 2D incompressible Navier-Stokes equations), but comes by with a slightly easier numerical formulation.

A. Patch concept

The computational domain is represented by a collection of Cartesian grid blocks with a globally uniform grid resolution. Cartesian grid blocks are a frequently used concept for CSE codes – examples being the patch concept of the lattice-Boltzmann code WaLBerla [5], [6] or the CFD-code Nast [4] (with its several descendants). The approach is also close enough to block-adaptive tsunami simulation codes, such as GeoCLAW [7], to make the techniques learned in our package relevant for production-quality codes in CSE.

In our code, the Cartesian grid blocks are mainly used to define parallel tasks – grid blocks will be assigned to MPI processes or OpenMP threads, for example, or they will discriminate between parallelization models in hybrid approaches (OpenMP/CUDA implementation within blocks, message passing between blocks). Figure 1 shows a simulation where the grid blocks were used for an MPI-parallel simulation run.

B. Finite Volume approach

As numerical approach, the package assumes a Finite Volume scheme, where an explicit time step $t^n \rightarrow t^{n+1}$ is formulated in the abstract form:

$$Q^{n+1}_{i,j} = Q^n_{i,j} - \Delta t \left( A^+ \Delta Q_{i-1/2,j} + A^- \Delta Q^n_{i+1/2,j} - B^+ \Delta Q_{i,j-1/2} + B^- \Delta Q^n_{i,j+1/2} \right).$$

(2)

Here $Q^n_{i,j} = [h_{i,j}, (hu)_{i,j}, (hv)_{i,j}]$ is the vector of conserved quantities at time step $t^n$, $\Delta t$ the length of the timestep and $\Delta x$ resp. $\Delta y$ denote the size of the grid cells. The terms $A^\pm \Delta Q_{i-1/2,j}$ correspond to the solution of the Riemann problem located at the edge lying on the left resp. right side of cell $(i,j)$, and the terms $B^\pm \Delta Q^n_{i,j+1/2}$ to the Riemann solution located at the edge below resp. above cell $(i,j)$. The above formulation was introduced in the context of Wave Propagation solvers using the net-update formulation in [8] and will be used in this text for solvers using numerical fluxes, such as the Lax-Friedrichs solver (see section III-C), instead of net-updates as well. All of the previously mentioned variables except for the quantity $Q_{i,j}$ are defined or calculated within one of the implemented Riemann solvers. In the later implementation, most of the numerical complexities can thus be hidden in the implementation of Riemann solvers, whereas the crucial edge/element loops for the update of unknowns reduce to a few lines of code that can be understood easily.

C. Riemann solvers

The edge-local Riemann problem can be reduced to a one-dimensional problem and is stated at initial time $t = 0$ as:

$$q(x,0) = \begin{cases} q_l, & \text{if } x < 0 \\ q_r, & \text{if } x > 0 \end{cases},$$

(3)

with

$$q_l = \begin{bmatrix} h_l \\ (hu)_l \\ b_l \end{bmatrix}, \quad q_r = \begin{bmatrix} h_r \\ (hu)_r \\ b_r \end{bmatrix},$$

(4)

where $q_l$ and $q_r$ summarize the cell variables in normal direction on the left resp. right side of the edge, including an optional bathymetry value $b_l$ resp. $b_r$.

A variety of Approximate Riemann solvers, which solve problem (3) on the basis of a linearization followed by an eigendecomposition of the one-dimensional SWEs exist – see [9]. The implemented solvers require different levels of mathematical background, if a proper understanding should be part of the overall teaching goal, if not they can be used as black-box-solvers too. So far the following solvers with increasing complexity in terms of numerics and computational effort are implemented:

(Local) Lax-Friedrichs solver

The Lax-Friedrichs method computes a single numerical flux on each edge, using a simple averaging plus adding a fixed amount of artificial diffusion for reasons of stability. In the local Lax-Friedrichs method (also referred to as Rusanov method), this diffusion is computed from a local wave speed – see [10].

The Lax-Friedrichs methods are used in our most basic implementation, mainly because they are simple to understand. However, they lead to poor quality of the solution, and are therefore typically replaced (in an almost black-box manner) by the following, more versatile solvers.

Figure 1: Propagating Tsunami, generated by the 2011 Tohoku earthquake. The white lines mark the domain boundaries of the 16 MPI processes used for the simulation.
f-Wave solver

The f-Wave solver uses a slightly different formulation of the Riemann problem (3). The common eigenvector decomposition takes place in terms of flux variables, which has several advantages in comparison to the classical Roe solver for example. The bathymetry source term is directly included into the flux decomposition.

One important drawback of the solver is that it is not “positivity preserving”. Details can be found in [11], [10].

Augmented Riemann solver

The class of Augmented Riemann solvers extends the standard Riemann solvers, which work with a decomposition of the initial data or the corresponding initial flux in the case of the f-Wave solver, only. Mathematically the Riemann problem (3) is augmented by the momentum flux of the f-Wave formulation and a steady state wave contributing the effect of the bathymetry. This leads to a four-dimensional Riemann problem. The solver combines several advantages of different standard solvers and is capable of handling wetting/drying problems, as required to simulate inundation.

Important features like preserving positivity or correctness in terms of a natural entropy fix are proven.

Drawbacks of the solver are the extensive background knowledge needed for a proper understanding and the expensive computation time. Details can be found in [12], [13].

Hybrid solver

The combination of the f-Wave solver and the Augmented Riemann solver is called hybrid solver in the following chapters. In general, the idea is to use the computational expensive Augmented Riemann solver for “difficult” Riemann problems only (for example, where inundation occurs). This solution was first presented as part of the GeoClaw software [7] and leads to interesting challenges regarding load balancing.

IV. SOFTWARE DESIGN OF SWE

The software design of our teaching code has to achieve a couple of goals, part of which are in conflict with each other:

- The software structure should not be overly complicated, such that students can understand the code quickly, and do not require too deep knowledge of the programming language. Hence, we chose C++, but tried to restrict object-oriented constructs to just a few basic concepts (abstract and derived classes, inheritance), and did not use templates, for example.

- The software design should allow the combination of programming models – as, for example, hybrid parallelization with MPI+OpenMP, or MPI+CUDA – and also an easy integration of students’ code into a more complicated problem setting. For example, students would concentrate on MPI parallelization in their exercises, but be able to include an externally-provided OpenMP-parallelized kernel for further experiments, or implement a GPU-based kernel, which will later be used in a hybrid MPI+CUDA setting.

- The design should offer the integration of more versatile software components – such as improved modeling variants, output routines for visualization software (ParaView [14]), support to read input data for realistic problem settings, or even an instant OpenGL-visualization of simulation results (see Section V-E).

We chose to tackle these conflicting goals by a straightforward hierarchical class design with an abstract class SWE_Block as the core, providing the core data structures, a basic set of interface functions to control the simulation, and basic helper methods for input and output. Derived classes extend this functionality to implement simulations via different parallel programming models or using different modeling variants.

A. The abstract base class: SWE_Block

At the core of our class design, the abstract base class SWE_Block provides the following data structures and functionality:

- Four 2D arrays for the water height $h$, momentum components $hu$ and $hv$, and bathymetry $b$ in each grid cell (including ghost layers, see Fig. 2): Note that the class intentionally does not include additional arrays for temporary variables, such as flux components or old/new variables for time stepping, as these can depend on the applied numerical model (wave propagation methods compute different components for each edge than the simple Lax-Friedrich method, for example) or on the applied programming model (for GPU computing, such temporary variable might be only needed in GPU memory).

- Virtual methods for the core components of a simulation run, such as

  ```cpp
  setBoundaryConditions(...),
  getMaxTimestep(...),
  simulateTimestep(...), or
  simulate(...) (to run the simulation until the next check point is reached):
  ```

- Students need to implement these methods in derived classes for specific models or programming models, or override a provided implementation, for example by an optimized implementation.

- Information on block sizes, grid resolution, time step size, etc., are stored as class variables, and are
Figure 2: Cartesian grid block represented by an SWE_Block object: White cells are ghost-layer cells, which are used to implement boundary conditions or to replicate unknowns from adjacent blocks. The dark-blue cells mark the copy layer, which may be replicated in an adjacent block’s ghost layer.

thus uniform for all SWE_Block objects. Therefore, SWE_Block also determines the Cartesian grid blocks used for discretization, as well as ghost layers and copy layers that organize the data transfer between blocks – see the illustration in Fig. 2.

B. A proxy class for ghost-layers: SWE_Block1D

A typical approach to implement MPI communication between ghost and copy layers would be to introduce a specific boundary condition to mark that a certain boundary is adjacent to a remote SWE_Block. In the method setBoundaryConditions(...), respective MPI send and receive calls then need to be implemented for this specific boundary condition. Due to its simplicity, we recommend this simple approach, if a course only considers pure MPI parallelization of the code. For hybrid parallelization, however, such an approach leads to problems: for example, OpenMP- and CUDA-implementation would both implement a derived class from SWE_Block. To allow their integration into an MPI-parallel setting, they would both need to implement setBoundaryConditions(...), which can easily lead to inconsistencies.

We therefore provide a mechanism to implement communication between ghost and copy layers from a container class or main program that uses SWE_Block as components: We introduced two methods, registerCopyLayer(...) and grabGhostLayer(...), in SWE_Block that return a proxy object for a copy or ghost layer at a specified boundary. A program component can then read and update unknowns in ghost and copy layers by respective calls to the proxy object. Typically, it will send and receive the respective data to and from an adjacent MPI process. By “grabbing” a ghost layer, the respective boundary conditions is set such that the respective SWE_Block will no longer actively set boundary conditions in this ghost layer – instead, it will rely on the calling component to provide respective data, for example via a call to MPI_Recv(). However, GPU implementations of SWE_Block are responsible to transfer ghost-layer data from main memory (where it is modified by container classes or programs) into GPU memory in each time step.

C. Providing problem scenarios – the SWE_Scenario interface

To allow both simple and complex problem settings in a uniform way, we implement the initialization of variables via an interface class SWE_Scenario. It basically consists of virtual functions to obtain initial values of unknowns at a given position, such as getWaterHeight(float x, float y). Derived classes will implement these according to specific scenarios. These can be analytical settings such as a “radial dam break scenario”, where water height and bathymetry are computed from analytical functions, and initial velocities are given by 0; however, we also provided a wrapper class that provides an interface to a versatile server for geo-information (ASAGI), such that realistic geometries can be read – see Section V-E.

V. PARALLELIZATION

In the following, we shortly describe the parallelization of the shallow water solver using NVIDIA CUDA, MPI, and OpenMP, which are currently our main focus in teaching parallel programming models. We will provide some performance results, as well; however, note that it is not the intent of this paper to demonstrate optimal performance in any sense. Our interest is rather in highlighting typical parallelization approaches, and also road blocks for parallelization. We provide a short overview on some further activities in Sec. V-E.

A. CUDA

A suitably efficient CUDA implementation of our shallow water solver can already by accomplished by a very basic approach. The two main steps that need to be parallelized are the computation of Riemann solutions for each edge, and the combination of flux components (or net-updates) for each cell. Both come down to a loop over all edges or cells, respectively. In CUDA, this is realized by defining a kernel to compute the Riemann problem for a given edge, or resp. a kernel that combines the net-updates for a single cell. We then define a separate thread to compute each edge/cell, where the threads will be grouped into 2D blocks (each of which will compute a grid tile of size TILE_SIZE×TILE_SIZE, with TILE_SIZE typically 8 or 16), and the thread blocks laid out as a 2D grid – see
the kernel calls given in Fig. 3. In the first exercises of a course, these kernels can be implemented one after another, transferring the unknowns to GPU memory and back before and after each kernel call.

The first goal for a course will be to do the entire computation on the GPU, and only transfer memory from and to GPU for input/output purposes. In addition to the kernels given in Fig. 3, this requires kernels to set boundary conditions. Further exercises then include the implementation of a time-step control (which requires a maximum reduction on the velocities), and to exploit shared memory between thread blocks (or registers) for the update of unknowns (and computation of fluxes) in order to improve performance. The applied techniques are quite similar to those described in the textbook by Kirk and Hwu [15], for matrix multiplication – presenting these techniques in a lecture and working on the shallow water code in practical exercises has proven to be a good combination.

B. MPI

A MPI parallelization will typically use as many SWE_Block objects as MPI processes are desired, and will assign one grid block to each process – compare Fig. 1. The students’ job will then be to implement the exchange of messages to update the ghost layers after each time step. With the proxy objects, as introduced in Sec. IV-B, a respective implementation will look similar to the code example provided in Fig. 4. Note that MPI_ROW, in this example, denotes an MPI data structure for row data in a column-major format, i.e., a stride access to memory. A simpler implementation might also use a separate buffer to send and receive data. These buffers can also be part of the proxy objects for copy/ghost-layers.

After the update of the ghost layers, the computation of the Riemann solutions and therefore the computation of the numerical fluxes resp. net-updates within a block is completely independent. To implement an adaptive time step control, a short synchronization step to exchange the maximum allowed time step size in each block is necessary, before the cells can be updated regarding to equation (2) independently again.

The I/O-routines provided with SWE_Block make massibe use of the efficient netCDF-libraries. For the input data a geo-information server (ASAGI, see Sec. V-E) is used, which implements a block-wise caching approach in respect to distributed memory in an efficient manner. Due to this implementation the code is able to read arbitrary bathymetry and even dynamic displacement data of nearly every size.

As test case, the first 300 seconds in simulation time of the 2011 Tohoku Tsunami have been calculated. The spatial domain is shown in Fig. 1 and reaches from Japan to Hawaii. An initial static displacement was used, with a Cartesian grid of 14000×8000 cells with a cell size of 493m×488m, which is a reasonable resolution for the research of far-field tsunamis.

In total 1.078 · 10^{11} edge-local Riemann-problems have been solved in the required 481 time steps, the hybrid solver was able to use the more simple f-Wave solver in 51.92% of all cases with the switch proposed in the kernel routines of GeoClaw [7]. We have performed a speedup study of the respective computations on the SGI-ICE cluster of the Leibniz Supercomputing Centre. SGI-ICE consists of 32 dual-socket quad-core Nehalem nodes with 24GB of Memory per node. Hence, we placed eight MPI processes per node (i.e., one per core).

On a larger number of cores, the execution time, as shown in Fig. 5, becomes short enough to allow feasible response times of the entire simulation. On 128 cores, a bit more than 200 seconds are required, if the Hybrid solver is used. Fig. 6 shows the parallel speedup (relative to 16 cores) achieved by these simulation runs.

The less-than-optimal speedup observed in Fig. 6 is to a large part caused by an interesting property of the Hybrid solver, which becomes apparent, when the corresponding CPU times are taken into account. As shown in Fig. 7,
C. Hybrid Parallelization (MPI + OpenMP)

For hybrid MPI+OpenMP parallelization, we can keep the existing MPI-parallel program described in Sec. V-B, but replace the serial implementation of Cartesian blocks (of class `SWE_WavePropagationBlock`) by a subclass of `SWE_Block` that provides OpenMP parallelization. We only used simple loop parallelization of the two main loops, which compute the net updates resp. numerical fluxes (see Sec. III-C) and the cell updates – see equation (2).

We observed an overall performance improve of 20%, on average, if one MPI process with 8 OpenMP threads is spawned per node on the SGI-ICE (compared to having 8 single-thread MPI processes). The parallel speedups (again relative to 16 cores) of the hybrid MPI+OpenMP version are shown in Fig. 8 for the Hybrid solver and in Fig. 9 for the Augmented solver. The reason for the lower speedups in the case of the Hybrid solver again lies in the increasing load imbalance. However, even in the highly unbalanced case of eight MPI processes, as shown in Figure 7, the Hybrid solver is still 14% faster than the Augmented solver. In the case of fewer partitions, much better factors, up to 25%, are observed.
#pragma omp parallel for collapse(2), schedule(dynamic)
for (int i=0; i<blockX; i++) {
    for (int j=0; j<blockY; j++) {
        splash[i][j]->updateCells(dt);
    }
}

Figure 10: OpenMP-parallel code segment to update multiple Cartesian grid patches.

D. OpenMP and Dynamic Scheduling

To tackle the load balancing problem identified in the MPI speedup tests, we tried an OpenMP-parallel implementation that parallelizes on the SWE_Block level: split the domain into substantially more grid blocks than compute cores are available, and parallelize the loops over this block array. A typical code segment is given in Figure 10. Note that we used the schedule(dynamic) clause to dynamically schedule each individual block update (each block being a 100×100 grid, e.g.) to cores. On a two-socket system equipped with Intel Westmere-EX processors, dynamic scheduling achieved an ≈20% improvement (on 16 cores) compared to using schedule(static), which again suffers from the load imbalance caused by the Hybrid Riemann solver. On the other hand, dynamic scheduling of blocks to cores is likely to suffer from NUMA effects on larger machines with shared-memory support. Here, a parallel programming model would be required that takes the memory location of SWE_Block s into account, and schedules them preferably to “local” cores.

E. Other Activities/student projects

In this paragraph, we give a short overview of further activities around the SWE code.

As part of a seminar course on parallel programming within the Ferienakademie, an annual summer school organized by three German universities1, we used the SWE code as the basis for discussing parallel programming models. For the MPI implementation, one student tested several options for optimizing the communication, such as overlapping communication and computation, or modifying the code to use two ghost layers, which only need to be exchanged every second time step. Another participant optimized the CUDA implementation, for example by fusing the kernels for flux computations on edges and update of cell-located unknowns. In that way, the flux components only need to be stored for a single tile of cells, and can thus be kept in GPU shared memory. In a third student project, we studied the parallelization using Intel’s Threading Building Blocks and Array Building Blocks (ArBB). The ArBB implementation, for example, was able to achieve an 8.8× speedup compared to the unoptimized core on a dual-core laptop – the considerable extra speedup being the result of much better vectorization and code optimization by the compiler.

As part of a student project, Tobias Schnabel implemented a package for direct visualization of the simulation data. The package is intended for the CUDA implementation and exploits the fact that all simulation data is already present in GPU memory. The water surface and bathymetry are then plotted using respective OpenGL functionality. Fig. 11 shows a respective snapshot. On a standard desktop computer, equipped with an NVIDIA Quadro 2000 GPU, the simulation and visualization can interactively simulate and visualize problems of 800×800 grid cells (interactively meaning that an entire simulation can be simulated in 1–2 minutes). This corresponds to more than 30 Mio element updates per second achieved by the CUDA code.

As part of his master’s thesis, Sebastian Rettenberger has implemented a parallel server for geo-information, ASAGI, which is able to read and provide bathymetry data or initial water displacements obtained from measurements or rupture simulations for other simulation packages. To our SWE package, this service is connected via a simple wrapper class that implements the SWE_Scenario interface using ASAGI’s functionality.

VI. SUMMARY/OUTLOOK

The SWE package, developed for a lecture in winter 2010/11, has already proven to be a highly useful and flexible code platform for teaching parallel programming, and has been used in several different teaching scenarios. SWE is available under GPL – for further information, see http://www5.in.tum.de/SWE/.

Plans for the near future include to test SWE on GPU clusters (with hybrid MPI+CUDA parallelization), and to try additional programming models, especially OpenCL. Also,
we intend to use SWE within the parallel programming mini-course of the Gene Golub SIAM Summer School 2012 on “Simulation and Supercomputing in the Geosciences”.

ACKNOWLEDGMENT

The authors would like to thank all students that experimented with the SWE software – in particular, Diana Gudu, Philipp Schmid, Tobias Schnabel, and Amirhesam Shahvarani. We would especially like to thank Hans Pabst and Alexander Heinecke for their support concerning Intel TBB and ArBB.

REFERENCES


