Solving the Shallow Water Equations using Finite Volume Method and Domain Decomposition in Julia

Felix Köhler
Technical University Munich
Chair of Scientific Computing in Computer Science
Munich, Germany
f.koehler@tum.de

***Abstract*—Research in computational science and engineering mostly incorporates the usage of two programming languages, a high-level dynamic language for prototyping and a low-level language for the final implementation and its parallelization. The *Julia* language aims to be a solution to this by offering close to *C* performance with many high-level features in a Just-In-Time compiled fashion. In this paper, we analyze this statement in the context of Tsunami simulations modelled by the Shallow Water Equations. The equations are solved using a Finite Volume Scheme with an *HLLE* Riemann solver. The program's implementation process is described. Performance comparisons against a reference (highly-optimized) *C++* implementation are drawn for the sequential case and a parallelization case using domain decomposition. The *Julia* code showed competitive performance in both cases making it a viable option for certain tasks in scientific computing.**

***Index Terms*—High Performance Computing, Julia, Just-in-Time Compilation, Tsunamis, Shallow Water Equations, Domain Decomposition, Parallel Computing**

I. Introduction

Traditional approaches in scientific computing are suffering from the problem that two languages are used throughout the process of creating an application or performing research. Typically, a very high-level dynamic language like *Python* is used for prototyping the algorithm and a low-level language like *C++* is then used for implementing the code efficiently and scaling it for a High Performance Computing (HPC) context.

The *Julia* language aims to be a remedy out of the double nature of programming languages. It promises close to *C* performance while keeping much of the conveniences of modern scripting languages. Therefore, it inherits concepts from languages as *MATLAB*, *Python* and *Lisp*. The built-in matrix processing and strong Linear Algebra package make it well suited for solving partial differential equations. Since its debut in 2009, the *Julia* community has grown and many scenarios did show its applicability and scalability [1].

This paper analyzes *Julia*'s capabilities in the context of solving the Shallow Water Equations (SWE). This is done by porting an existing suite of Shallow Water Equation solvers written in *C++* [2] to *Julia*.

The paper will briefly introduce the *Shallow Water Equations*. We then cover the difficulties and observations when implementing a Finite Volume Method algorithm together with an approximate Riemann solver in *Julia*. Afterwards, two major comparisons are presented. First, we reason about the sequential performance and the specialties of the new language in relation to the *C++* code. The second part comprises the parallelization with domain decomposition and message passing. Here we compare *Julia*'s native *Distributed* package with the *MPI*-based implementation of the *C++*-code.

II. The Shallow Water Equations

The *Shallow Water Equations* are a set of coupled non-linear partial differential equations (PDE). They form a hyperbolic system in conservative form with a source term.

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghb_x \\ -ghb_y \end{bmatrix} \tag{1}$$

Here, we use a vector of conserved quantities $\vec{q} = [h, hu, hv]^T$ (wave height, momentum in x, momentum in y). The source term considers the influence of varying bathymetry b. The spatial domain is two-dimensional, hence $(\cdot)_x$ and $(\cdot)_y$ denote the partial derivative w.r.t. the first and the second axis, respectively. g denotes the acceleration due to gravity.

In the context of solving these equations numerically, we employ a Cartesian grid to discretize the domain into cells of finite volume. The equations (and hence the conserved quantities) are spatially averaged over these cells, $\vec{q} \to \vec{Q}_{i,j}$. With an additional forward Euler discretization of the time derivative $(\cdot)_t$ we get the update routine for each cell.

$$\begin{aligned} \vec{Q}_{i,j}^{[n+1]} = \vec{Q}_{i,j}^{[n]} &- \frac{\Delta t}{\Delta x}\left(\mathcal{F}_{i-\frac{1}{2},j}^{[n]} + \mathcal{F}_{i+\frac{1}{2},j}^{[n]}\right) \\ &- \frac{\Delta t}{\Delta y}\left(\mathcal{F}_{i,j-\frac{1}{2}}^{[n]} + \mathcal{F}_{i,j+\frac{1}{2}}^{[n]}\right) \end{aligned} \tag{2}$$

In this update over time-step Δt and cell width Δx and Δy, $\mathcal{F}$ denotes the numerical flux which we evaluate at the cell edges. To achieve a high-accuracy in this Godunov-type scheme we employ an approximate Riemann solver at each edge. More details on the used *HLLE*-solver and the Finite Volume Method can be found in [3]. The time-step is adaptively chosen based on the maximum wave speed, the cell widths and a given CFL-condition (set to 0.4 here).

For our numerical experiments and the comparison with the reference implementation, we use the scenario of a radial dam break with an outflow boundary condition. Initially, the water

is at rest, i.e. $hu(t = 0, x, y) = hv(t = 0, x, y) = 0$. Only in the center of our two-dimensional domain $\Omega = [0, 1000] \times [0, 1000]$, there is a circular peak in water height.

$$h(t = 0, x, y) = \begin{cases} 15 & \sqrt{(x - 500)^2 + (y - 500)^2} < 100 \\ 10 & \text{else} \end{cases} \tag{3}$$

The simulation spans over $t \in [0, 15]$ with 20 checkpoints along the way to save the fields for postprocessing.

III. Sequential Implementation

A. Porting the SWE solver to Julia

As already mentioned in the introduction, Julia advertises to be a solution of the two language problem. We can confirm this statement during the process of porting the SWE solver. The ability to quickly prototype language specialties in the REPL[1] aided in the process of learning the language. Additionally, *Julia* is a garbage-collected language making memory management simpler than in *C++*.

Julia provides many convenience functions known from *MATLAB* and *Numpy* when working with high-dimensional data structures. Functions and operators can be broadcasted to apply to all elements of a multi-dimensional array. The Just-In-Time (JIT) compiler together with the *LLVM* backend then transforms these operations into efficient *SIMD* instructions.

Another advantage making *Julia* a modern programming language is its integrated package system. As of June 2020, there are $3'000$ registered *Julia* packages. This is less than *Python*'s $240'000$ but they cover many important fields. For the process of porting, we used a package to write *NetCDF* files. This package system is simpler to use than big build systems for *C++*, e.g. *CMake*.

To make up for possibly missing package functionalities, *Julia* provides a no-overhead call of *C* and *FORTRAN* libraries. This makes many classical numerical libraries available to the user. Similar to *NumPy* and *SciPy*, *Julia*'s integrated *LinearAlgebra* library provides highly-optimized *BLAS* and *LAPACK* routines.

Over the course of translating the program, we experienced performance difficulties that are probably caused by our first exposure to the language. *Julia*'s documentation with the hints on performance improvements was considered of great help. We see these four points to be the biggest bottlenecks.

1) *Julia*'s builtin arrays always allocate on the heap. This is costly, especially in numerical kernels (in our case the Riemann solver) that are called frequently. With the *StaticArrays* package, one can define stack-allocated arrays improving the performance in our case by almost one order of magnitude.
2) When coming from *Python* one might never explicitly declare the type of a variable in *Julia*. However, in certain cases this can be helpful for the compiler. In our example code, we were able to speed up the implementation by almost 50% by fixing the type of strategic variables.
3) *Julia*'s builtin array operations can be convenient but in the case of slice operations, *Julia* will allocate a new array on the heap for the requested excerpt. This is beneficial for long computations since *Julia* can make sure that the memory is aligned. However, in simple assignments this can have a negative impact on performance. Applying the `@views` macro, *Julia* only uses a reference to the elements in memory. Instead of

```
array_1[2, 2:101] = array_2[1, 2:101];
```

one would write

```
@inbounds @. @views array_1[2, 2:101] =
    array_2[1, 2:101];
```

The `@.` macro defines the upcoming operations to be broadcasted making them easier to vectorize for the JIT compiler. In this scenario, we additionally deactivated any bounds-check by the use of a macro. One can declare an entire scope to be expended by a set of macros to reduce the code footprint.
4) A common issue in the *Julia* community is the so-called "time-to-plot" meaning the duration between starting a *Julia* program and getting a result. Especially for short-running computations, the impact of the JIT compiler, which is called every start of the program, can be noticeable. One can use the *PackageCompiler* library to precompile frequently used libraries. This was not used in this project and we will report the time influence of compilation and startup.

B. Performance comparison against reference C++ code

Now, we will compare its performance with the implementation in *C++* (which is highly optimized). The *C++* variant uses single precision floating point numbers and *Julia* double precision. The machine is a 16 core *Intel Xeon* (*Haswell*) processor with a $2.3GHz$ clock speed accompanied with $16GB$ of memory. The *C++* code was compiled with *GCC*. We deactivated vectorization for both programs[2].

Table I shows the execution times of different scenarios. Here we differentiate between:

- "J(ulia) wrap": The naive execution time of the *Julia* code. The whole program call was wrapped in a *Unix* `time` command.
- "J(ulia) main": Is the time for the program logic. This excludes the startup time but still includes JIT compilations occurring during the first time any code statement is executed. We also exclude IO here.
- "J(ulia) main p(ure)": Is the time for the program logic excluding all overhead caused by startup and compilation. We also exclude IO here.

[1]Read-Evaluate-Print-Loop, the interactive Julia mode

[2]The vectorization of both *Julia* and *C++* was poor since the auto-vectorization scenario was too difficult for *LLVM* and *GCC*, respectively. To keep the comparison fair we did not use the *Intel* compiler which was able to almost perfectly vectorize the *C++* code reducing its runtime by about 70%.

- "S(WE) wrap": The naive execution time of the reference implementation captured similarly with the `time` command.
- "S(WE) core": The main simulation task, excluding IO.

TABLE I: Time measurements in $[s]$ for the sequential implementations with a varying number of cells per axis.

#	J wrap	J main	J main p	S wrap	S core
50	20.21	4.97	0.23	0.05	0.01
100	20.32	5.12	0.10	0.14	0.05
200	21.01	5.71	0.68	0.69	0.49
400	25.24	10.13	5.16	4.50	3.91
800	61.98	46.73	41.40	32.57	30.74
1600	364.42	349.32	336.82	410.19	239.70

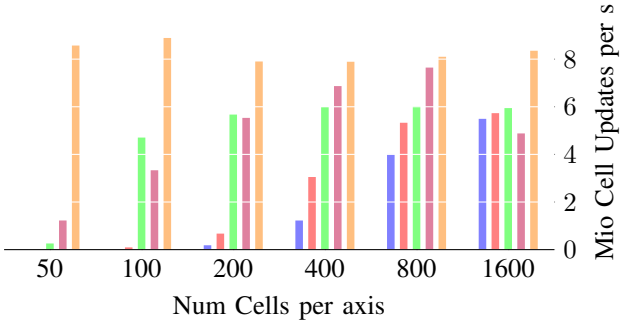

Fig. 1: Cell Updates per second for the scenarios given in table I over various numbers of cells per axis.

Looking at the three *Julia* scenarios, we can see the impact of the two major overheads. Starting up the *Julia* compiler, reading in all included files and compiling the used libraries takes approx. $15s$ and is constant over the problem size. The JIT compilation of the actual code takes approx. $5s$ and is also problem size independent. As a comparison, the compilation of the *C++* binary took $7s$. However, this is a one-time cost. The presented overhead for the *Julia* code is present every time it is started. On the other side, *Julia* can make sure that the executed instructions are always optimal for the underlying hardware. For bigger problem sizes and hence longer compute time, the ratio of the *Julia* overhead becomes smaller.

Ignoring the overhead, we see an almost similar performance. For the biggest problem size of 1600×1600 cells the pure *Julia* code is only 29% slower than the core *C++* code. For some scenarios in scientific computing the gained flexibility of a highly-dynamic language could be worth this price. Usually, performances of languages as *Python* are at least one order of magnitude slower than *C++* code.

We do also observe that the implementation of the IO mechanisms is more efficient in *Julia* than in the *C++* implementation. This explains why, for the biggest problem size, the wrapped *Julia* code runs faster.

IV. Parallel Implementation

It is important to have a fast sequential implementation to then create efficient parallel applications that scale well. In this section we compare the performance when decomposing the domain into patches and assigning them to individual worker processors. In the *C++* code, a parallelization using *MPI* was chosen. *Julia* uses an own primary/worker based distribution model. We do not use any shared-memory parallelism (i.e. multithreading e.g. by *OpenMP*). The exchange of halo layer ghost cells is achieved through explicit message passing. For the simplicity of our analysis we also did not consider distributed computing over multiple network-connected nodes.

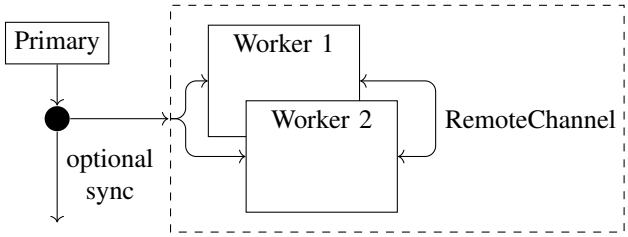

Fig. 2: One-sided communication with *Julia*'s builtin primitives. One primary process controls many worker processes (possibly also on network-connected nodes) by assigning tasks to them. Halo cells between workers are exchanged with RemoteChannels.

A. The Parallel Model in Julia

Decomposing the computational domain into subdomains is a common parallelization pattern in discretization based solutions to partial differential equations. One process owns a certain subset of the collection of Finite Volume cells and exchanges its boundary values with its adjacent domain.

Julia uses a primary/worker approach to parallel computing[3]. This means that one primary process spawns tasks on worker processes. We instantiate the field arrays for each block on a separate process and the primary process keeps remote references to later spawn tasks that work directly on this data. Remote references are implemented as futures in *Julia*.

Additionally, we have to set up so called remote channels for each interior domain boundary that are used for communicating the halo cells between the blocks. Every block writes its own *NetCDF* file. Fig. 2 depicts the program flow. After each solution step, there can be optional synchronization constructs which is here not necessary because the RemoteChannel's implementation is blocking implementing a natural barrier in each iteration loop.

When comparing the implementation to the *C++*, one has to again note that the prototyping in *Julia* is convenient due to expressive macros. This also improves the readability of the code and most certainly also its maintainability.

B. Performance comparison against the reference MPI implementation

In the previous section on the sequential performance, we saw that the *C++* implementation of the *NetCDF* writer is a bottleneck in comparison to the *Julia* code which is the reason we decided to deactivate IO for the upcoming analysis. We again deactivated auto-vectorization of both *Julia* and the *GCC* compiler.

[3]Additionally, *Julia* offers shared memory parallelism similar to *OpenMP* and coroutines similar to *Go*, *JavaScript* and modern *C++*.

We now compare the execution time and speedup for various numbers of processors for a fixed problem size of a domain with 2000 cells in each axis. The numbers of processors are varied from 1 to 16. The test machine consists of two sockets with 8 cores each resulting in 2 NUMA domains.

We only consider two scenarios:

- "J(ulia) main pure no IO": Similar to the sequential implementation we do not include startup ($\approx 15s$ for startup and compilation of libraries) and compilation ($\approx 8s$ to compile the code on each processor) overhead since we were able to show that this is close to the sequential analysis. Additionally, no IO is performed.
- "SWE wrap no IO": The `mpirun` was wrapped inside a `time` statement. IO was disabled in the code.

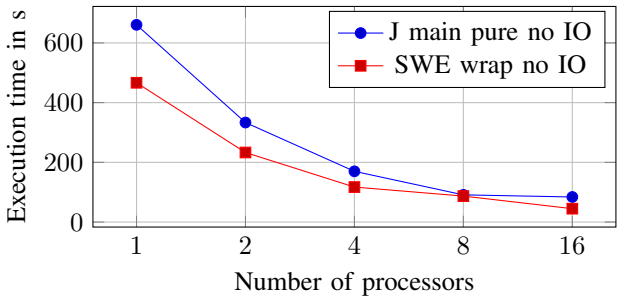

Fig. 3: Execution time in $[s]$ over the number of processors.

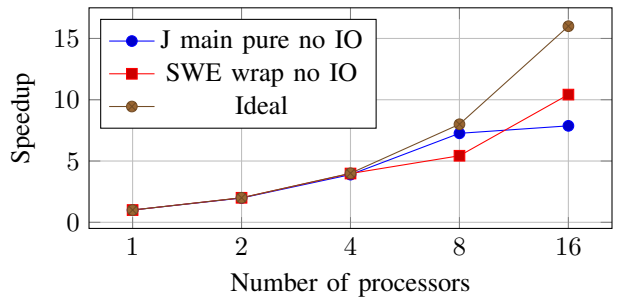

Fig. 4: Speedup of the *Julia* implementation and the *C++* implementation over various numbers of processors.

By looking at Fig. 3, we observe a similar result for the 1-core-case as in the sequential scenario. Keep in mind, that the parallel implementation with one core is slightly different from the sequential implementation for both *Julia* and *C++*. Still, the *Julia* code performs only 30% slower than its reference.

The *Julia* code's absolute performance scales better than the reference with *MPI* but only until 8 cores. The converged time, also visible in the speedup graph of Fig. 4, can be caused by the greater overhead of the primary/worker approach and a potentially weaker implementation of message passing over two *NUMA* domains than the more mature *MPI* implementation. Still, we achieve comparable scaling for the *Julia* code. For an easier porting process, one could have also used the *Julia MPI* bindings.

V. Conclusion & Future Work

In this paper we ported an existing suite of Shallow Water Equations solvers from *C++* to *Julia* for a comparison of their performance in the sequential and parallel case. Over the course of translating, we experienced *Julia* to be a beginner-friendly language offering many of the known conveniences from high-level interpreted languages. However, we have to note that we spent almost as much time tuning the code as we spent implementing. We think this is caused by writing the first project in *Julia*. The documentation helped to improve on the major bottlenecks.

With this *Julia* code we were able to show that the sequential execution time is only about 30% slower than the *C++* code which combining with the faster development time and all the other benefits of the *Julia* language could be worth it for some applications. The Just-In-Time compilation of *Julia* results in an overhead of $\approx 20s$ each startup which is noticeable for small problem sizes.

The parallel implementation was able to scale well until block sizes went below 500 by 500 cells whereas the *MPI* variant of the *C++* code kept speeding up. We found *Julia*'s builtin execution model to be expressive and easy to use. However, for scenarios of domain decomposition for scientific computing we would resort to the *MPI* bindings since this more naturally fits the parallelization strategy.

Another way could be changing to an actor-based distribution model which reduces the spawning overhead and could be implemented with *Julia*'s *Distributed* package that was also used in this paper.

The *Julia* community is aware of the overhead in running code. For the future, it is planned to create an ahead-of-time compile option creating a binary out of a *Julia* script which reduces the overhead time to almost zero similar to a compiled *C++* executable. Until then, a potential remedy is outsourcing some functions and numerical kernels into own packages and use *Julia*'s *PackageCompiler* library to precompile them. However, the JIT approach has the advantages that the generated machine instructions are always optimal for the underlying hardware and with each improvement in the *LLVM* backend, the *Julia* code also improves in performance.

In conclusion, *Julia* holds up to its promise of solving the two language problem. Especially in modern times when computing becomes cheaper, a faster prototyping time and the advantage to not maintain two code-bases might outweigh the 30% slowdown.

The code is available open source via Github:

```
github.com/Ceyron/Tsunamis.jl
github.com/TUM-I5/SWE
```

References

[1] J. Bezanson and A. Edelman and S. Karpinski and V.B. Shar, "Julia: A fresh approach to Numerical Computing", SIAM Review 2017 59:1, 65-98

[2] A. Breuer and M. Bader, "Teaching Parallel Programming Models on a Shallow-Water Code", 2012 11th International Symposium on Parallel and Distributed Computing, Munich, 2012, pp. 301-308, doi: 10.1109/ISPDC.2012.48.

[3] D.L. George, "Finite Volume Methods and adaptive refinement for Tsunami propagation and inundation", University of Washington, Diss. Ph. D. Thesis, 2006