18.337/6.338: Numerical Computing With Julia

The Swept Rule for Breaking the Latency Barrier in Time-Advancing PDEs

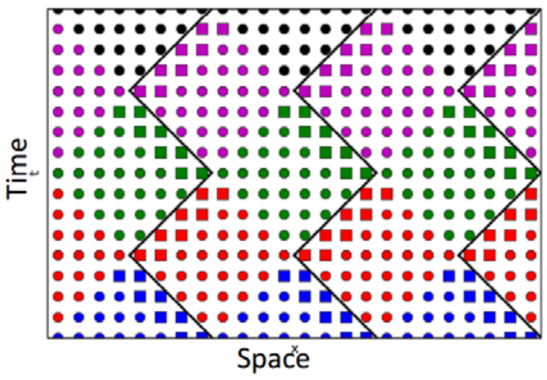
Project Report

Maitham Alhubail, Aziz Albaiz, Mohamad Sindi, Mohammad Aladwani

Introduction

Most parallel PDE solvers undergo severe limitations when it comes to strong scaling. Beyond a certain point, adding more resources, namely compute nodes, to solve a certain PDE problem does not decrease the time required to solve the PDE, and in many cases, the overhead may take over and result in poorer performance. The most fundamental limit to strong scalability is the network latency barrier. Because of the network latency, a delay is experienced whenever data is exchanged between compute nodes, and therefore, the speed of a solver is limited by the frequency at which it exchanges data between the compute nodes. To overcome this latency barrier, we invented the Swept Rule, which achieves superior performance by reducing how often inter-process communication is needed, and hence decreases the overall experienced network latency. This is achieved by decomposing both space and time among computing nodes with respect to the domains of influence and the domain of dependency [1].

The Swept Rule follows the domain of influence and the domains of dependency while explicitly solving PDEs. The communication latency is reduced by following the way that allows you to proceed further without communication. The domain is decomposed into smaller subdomains that are assigned to different computing nodes, or cores. Each core performs the PDE integration over time, not only for one time step, but for several time steps without the need to communicate data between these time steps. Then, a single communication to each neighboring core is made to send the set of data that is shared between the two. This is repeated until the system is completely solved.



Previous Work

After some theoretical analysis of the Swept Rule, a successful implementation for the Swept Rule in a single dimensional domain was done using C++ with MPI as the underlying parallel library. The outcome was remarkable, even when the code was run on cloud-computing clusters, which usually has higher network latency than dedicated clusters. The speed-up measured during our experiments almost matched what was obtained in the theoretical analysis. These promising results motivated us to implement the Swept Rule in higher dimensions. Currently, the Swept rule is being developed in 2D using C++ and MPI.

Objective

The main goal of the project is implement and evaluate the Swept Rule in 2D in Julia. This includes the following components: developing the local computing engine that performs all the intra-process operations, developing the inter-process communication and synchronization schemes to achieve efficient and correct transfer of data across processes, and writing sample elliptic PDE solvers using the Swept Rule, such as the convection–diffusion equation and the Poisson equation.

The implementation of the Swept Rule in Julia also allows us to compare Julia's parallel-computing performance to the performance of C++ and MPI.

Swept Decomposition for 2D Space and Time: Swept2D

As the swept decomposition scheme advances in time, the grid points assigned to each MPI process or worker thread change over time; this is true for swept decomposition in 1D, 2D, and 3D. We call the time it takes swept to go back to the initial grid points' assignment a Swept cycle. With the Swept2D implementation, the domain is decomposed into squares with sides of length n. Each complete Swept2D cycle advances the entire computation domain by $n/2$ time-steps. When all processes/threads perform a half-cycle, the entire computation domain is shifted by $n/2$ grid points in the x direction, and $n/2$ grid points in the y direction. As the Swept2D completes the second half of its cycle, the computation domain will be shifted back to its original position, by shifting $n/2$ grid points in the x direction and $n/2$ grid points in the y direction, but in an opposite direction to the shifts done in the first half of the cycle. The net shifting result of the entire cycle will be zero in both the x and y directions.

To simplify the implementation of the Swept2D decomposition, we break the whole scheme into small pieces: pyramids and bridges. The pyramids can either be pointing up or down, whereas the bridges can be either horizontal or vertical. Figure 1 illustrates the basic components of the Swept2D implementation components.

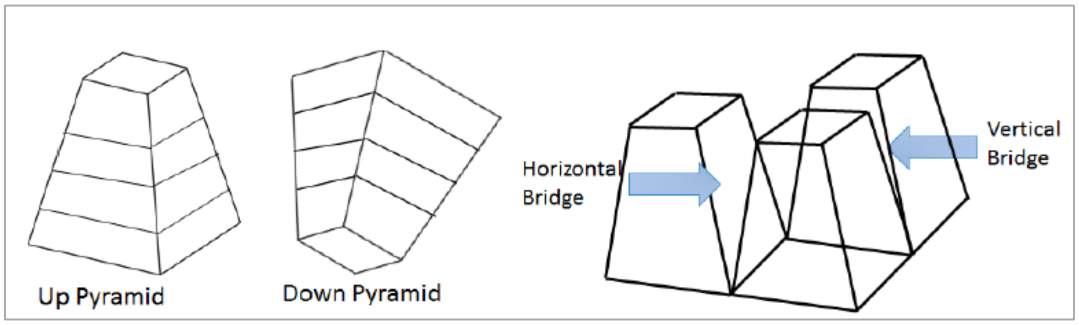


Figure 1: The basic building blocks of Swept2D cycle implementation.

We start with the 2D computational domain divided into squares. As the PDE solution advances in time and follows the domains of influence and dependency, it is noticed that the squares become foundations of pyramids that point upwards. The end result of such an up-pointing pyramid is four panels: North, South, East, and West. These panels will be communicated between the MPI processes and will be the ingredients to going further with the Swept2D and building the bridges, which fill the valleys generated when the up-pointing pyramids were built. This is the first communication that takes place in our Swept2D implementation. Figure 2(a) shows the panels of an unrolled up-pointing pyramid. Now, it might be obvious that given East and West panels, we should be able to fill a horizontal gap between two up-pointing pyramids. We refer to the result of this step as a Horizontal bridge, which consists of a pair of panels: a North panel and a South panel. The same goes for Vertical bridges, which fill the vertical gaps between the up-pointing pyramids with East and West panels. Notice that the panels generated by bridges are pointing down and not like the panels generated by the up-pointing pyramids, which are obviously pointing up. The second communication in our Swept2D implementation takes place when we start communicating the bridges' panels among the MPI processes or worker threads. Figures 2(b) and 3(a) show the inputs and outputs of building the horizontal and vertical bridges, respectively.

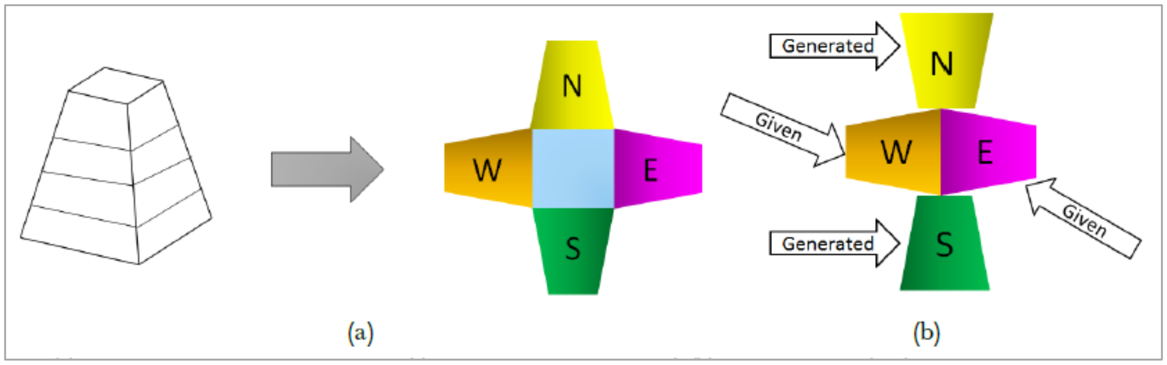


Figure 2: (a) Illustration of the panels generated by an up-pointing pyramid. (b) The horizontal bridge connecting two up-pointing pyramids.

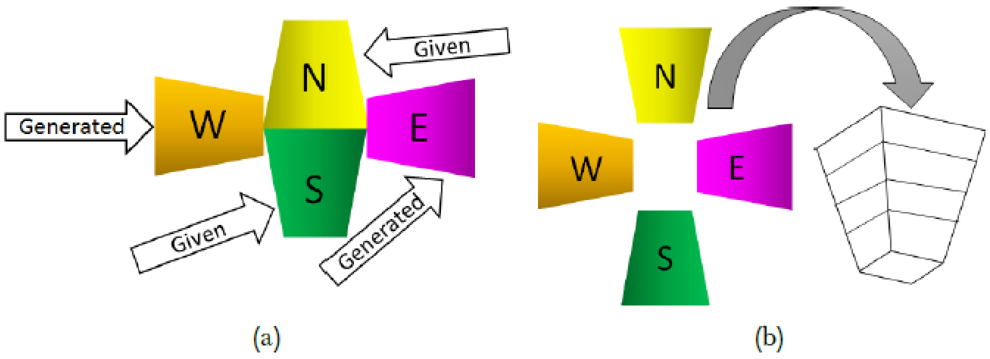


Figure 3: (a) The vertical bridge connecting two up-pointing pyramids. (b) The down-pointing pyramid of Swept2D.

After the second communication is done, each MPI process should have four down-pointing bridges; those bridges are the four sides, or panels, of what we call a down-pointing pyramid. It is clear that filling the gap between those down-pointing panels forms a down-pointing pyramid, which ends with a square foundation. Figure 3(b) shows the input and output of the down-pointing pyramid building process. By now, each participating MPI process or worker thread should have completed building one of each of the four components we defined for our Swept2D implementation using two communications. This stage completes half a cycle. The second half of the cycle proceeds exactly like the first half, except that the starting point will be the foundation of the down-pointing pyramid, which represent the shifted squares of the computational domain. By communicating the up-pointing and down-pointing panels properly, the end result of the second half of the Swept2D cycle—the set of foundations generated by building the down-pointing pyramids—will be squares containing the same grid points we initially started with but advanced n time-steps.

So, a complete Swept2D cycle needs only four total communications for n time-steps compared to n communications for n time-steps using the classical domain decomposition approach. Figure 4 demonstrates the stages that a Swept2D cycle goes through. Notice that the shifts of the computational domain are not shown in this figure.

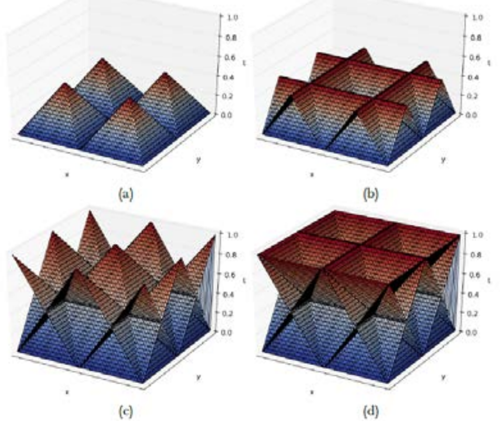


Figure 4: The steps of a Swept2D cycle.

Implementation

We have developed a `swept2D.jl` library in Julia implementing the Swept algorithm in 2D. The library is easy to include and use in your code to solve PDEs, you just need to setup your PDE of interest and its initial condition and the parallelization part is taken care of by our library. For the parallelization part we used Julia's low level remote calls. We didn't want to use MPI since it is C based and we wanted to keep everything Julia all the way down. The parallel remote calls we used in Julia are of the following syntax:

```
remotecall_fetch(procesesor id, function, args...)
```

The code in figure 5 shows an example of how the swept2D library is used to run a PDE solver in parallel using Swept Rule.

```
#Add number of parallel processors needed
addprocs(2*2);

#Set domain parameters needed for Swept library
@everywhere N       = 64;
@everywhere xNodes  = 2;
@everywhere yNodes  = 2;

#Include the swept2D library (inside this library you can set your PDE and its initial condition)
@everywhere include("swept2d.jl");

#3D array to store solution
solution = zeros(N*xNodes,N*yNodes,200); #This is 2D but 3rd dimension is time, 200 time steps

#Sets the initial condition, in this example it puts heat source in middle (modify as desired)
setInitCondition();

#Get the initial condition at time t=1 and store it in solution
solution[:,:,1] = getSolution();

#Loop remaining time starting at 2 and store solution for all time steps for later plotting
@time for i=2:200
    calculate();  #Does a full cycle for Swept
    solution[:,:,i] = getSolution();
end
```

Figure 5: Example of how the *swept2d.jl* library is used.

Performance Evaluation

The Swept rule is meant to break the latency barrier that the classical approaches suffers from. In the classical method, exchange of data between different computing units is done at the end of every time step. This results in repeated latency. The Swept Rule is designed with the domain of influence and domain of dependency in mind, reducing the number of communications between computing units, and resulting in an overall speed up by avoiding all the additional latency.

In order to evaluate the performance of Swept, we compare it with the classical method. The metric that is used for comparison is the elapsed time for running the solver for a fixed number of time steps.

Figure 6 shows the performance of Swept and classical method in their Julia implementation. The x-axis represents the number of parallel computing units, or cores, that were used to run the benchmark. It can be clearly seen that the performance of Swept is at least an order of magnitude better than the classical method. This is true for both small and large number of parallel cores.

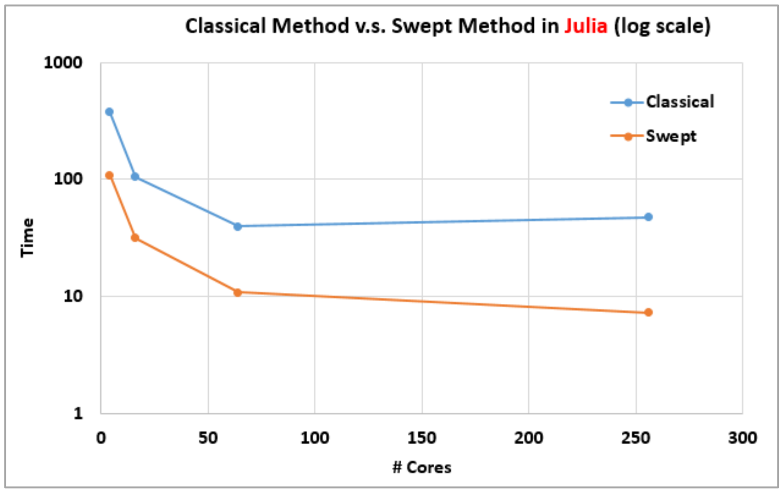


Figure 6: Elapsed time of the classical and swept methods in **Julia**.

When comparing the same case in the C++ implementation for Swept and classical methods, shown in figure 7, it is still the case that Swept is faster than the classical method. However, the speedup in C++ is not as large. One of the reasons for that is that the runtime in C++ is much faster than Julia for both methods, and therefore the difference between the two methods is smaller.

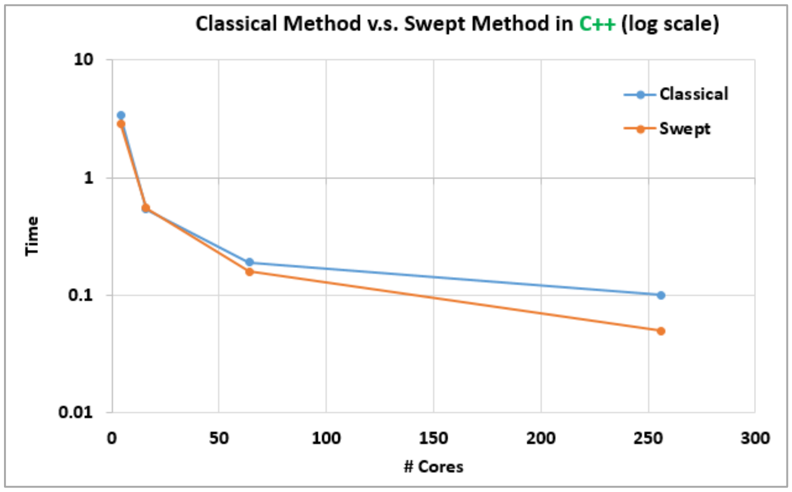


Figure 7: Elapsed time of the classical and swept methods in **C++**.

Another aspect that we looked it during our performance evaluation is the differences in performance between the C++ and Julia implementations of the Swept method. Both implementations were run to solve the same PDE over a number of timesteps, and the elapsed time over a different number of computing units, is shown in figure 8. There seems to be a large performance gap between Julia and C++. The Julia implementation was optimized to avoid any known performance issues (such as dynamic memory allocations). However, the performance gap is still at least an order of magnitude between the two.

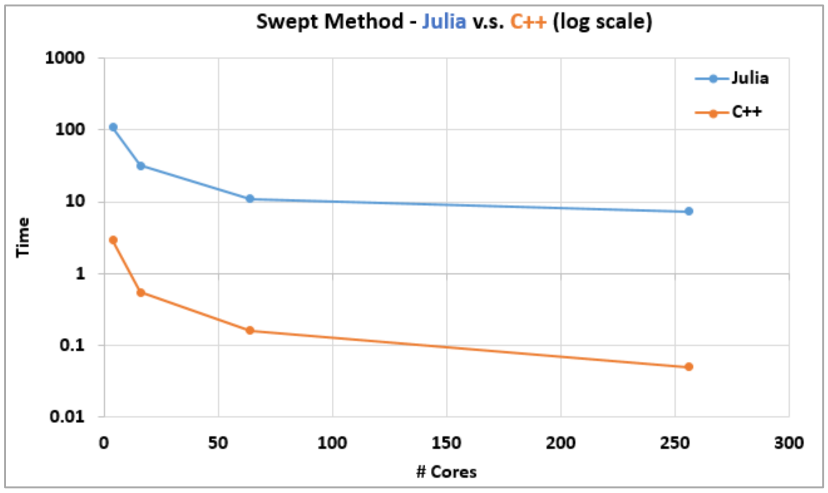


Figure 8: Elapsed time of the **Swept** method in **Julia** and **C++**.

In order to ensure that this performance gap is not related to an issue in our implementation, we have compared the performance of the classical method in Julia and C++ as well. As figure 9 shows, the performance gap is still there. While the scalability and speedup behavior are very similar when the number of cores is increased, Julia seems to have some performance overhead that is preventing it from achieving its potential performance, a performance that is comparable to C++.

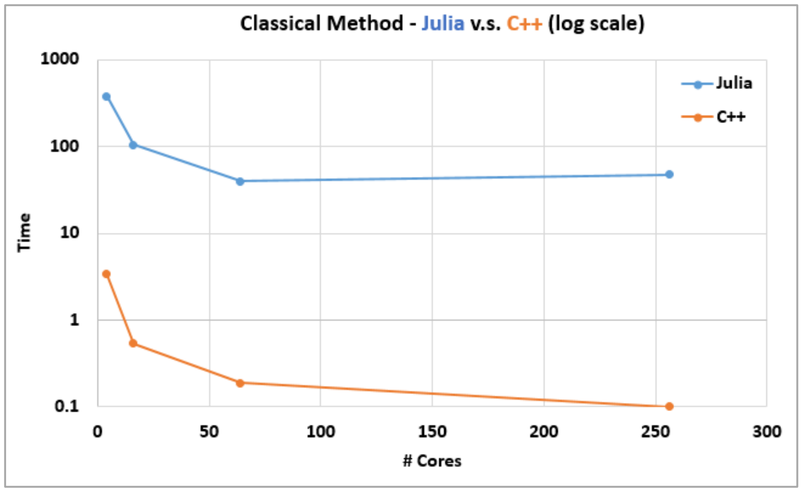


Figure 9: Elapsed time of the **classical** method in **Julia** and **C++**.

Challenges Encountered During the Project

The Julia `include` statement seems to be very slow when running on a large number of cores. For example when running on 256 cores, it took ~80 seconds just to execute the include statement, while the actually parallel computation only took 7 seconds. The include statements we used in Julia are of the following syntax:

```
@everywhere include("swept2d.jl");
```

We also had issues using the `machinefile` option when running the code distributed among several physical node as the option didn't seem to work properly. We had to construct the host string manually in the code and pass it to the `addprocs` function as a workaround.

Out of boundary errors in Julia were also a hassle to deal with as they were difficult to debug especially when running in parallel. The Julia debug info didn't provide proper line numbers, while using print statements to debug in parallel wasn't convenient when running on a large number of cores (e.g. 256 cores).

References

[1] Alhubail, Maitham Makki, and Qiqi Wang. "The swept rule for breaking the latency barrier in time advancing PDEs." *arXiv preprint arXiv:1504.01380* (2015).