High-Performance PDE Solver with ParallelStencil.jl in Julia

Zhang Wu

May 17, 2023

1 Github Repo

https://github.com/Peacockspider/18337

2 Background and Significance

Partial Differential Equations (PDEs) are a fundamental mathematical tool used to describe the behavior of many natural and physical systems, making them a crucial area of study in various scientific and engineering fields. Solving PDEs is essential in advancing our understanding of various phenomena, including fluid dynamics, electromagnetism, and quantum mechanics. PDEs often arise in real-world problems, and their solution provides valuable insights into the behavior of complex systems, making it a critical component in scientific research. However, solving PDEs is typically a computationally intensive task, requiring advanced numerical methods and high-performance computing. Therefore, the development of efficient PDE solvers is crucial in enabling the progress of scientific research.

Stencil numerical calculations are a type of algorithm used for solving partial differential equations (PDEs) numerically. Stencil numerical calculations work by discretizing a continuous PDE into a set of discrete equations that can be solved using a computer. The region of interest is discretized into square (2D) or cubic (3D) cells connected by grid points in order to perform stencil calculations. The basic idea behind this is to replace the derivatives in the PDE with finite difference approximations, which involves approximating the value of the function at a point using the values of the function at neighboring points. For example, the laplacian operator in a PDE can be approximated by the finite difference formula (1) as follows.

$$\nabla^2 f[i,j] \approx \frac{f[i+1,j] - 2f[i,j] + f[i+1,j]}{\Delta x^2} + \frac{f[i,j-1] - 2f[i,j] + f[i,j+1]}{\Delta y^2} \tag{1}$$

The Gray-Scott model is a mathematical model first proposed by James Gray and Scott Pearson which simulates the reaction-diffusion dynamics observed in various chemical systems. It consists of a set of partial differential equations that describe the time evolution of the concentration of multiple interacting chemical species in a spatial domain. For example, for the chemical reaction,

$$U + 2V \to 3V; V \to P \tag{2}$$

the mathematical formula for the overall behavior is shown in equation 3.

$$\frac{\partial u}{\partial t} = D_u \Delta u - uv^2 + F(1-u); \frac{\partial v}{\partial t} = D_v \Delta v + uv^2 - (F+k)v \tag{3}$$

where U, V and P are the three chemical species, u and v representing the concentrations of U and V respectively, ru and rv are the corresponding diffusion rates, k being the rate of conversion of V to P, and f being the rate of the process that feeds U and drains U, V and P.

Based on these formula, the Gray-Scott model is able to generate patterns and complex structures through the interplay of diffusion and reaction processes. By adjusting the parameters of the model, such as the F and k constants, different patterns can be generated, which can range from simple spots or stripes to more intricate structures like spirals or target-like arrangements.

3 Scientific Goals and Objectives

The goal of this project is to develop an efficient Gray-Scott PDE solver of high performance with scalability for large domain fields. Multi-threading with the employment of the ParallelStencil.jl package for shared memory parallelism and MPI communication with the use of the MPIHaloArrays.jl for distributed memory parallelism are explored in this project.

4 Serial Code

A naive serial code is implemented in `gs_serial.jl` with the constant parameters set at Du = 0.1, Dv = 0.05, F = 0.0545 and k = 0.062. A matrix U which represents the concentration distribution of the u species is initialized to be 1 everywhere except for a center square zone where u = 0.5, whereas the matrix V which represents the concentration distribution of the v species is initialized to be 0 everywhere and v = 0.25 in the center square zone. The center square zone occupies about 20% of the entire domain. A simulation result of the serial code is shown in Figure 1. The simulated pattern evolution in the spatial domain agrees with the general results available on https://pnavaro.github.io/python-fortran/06.gray-scott-model.html only in the beginning period. The periodic patter structures in the later stage are not reproduced. Therefore, I suspect that there're still some intrinsic implementation errors in the serial code which I couldn't figure out where up to now (may be something related to the periodic boundary conditions for the entire domain).

5 Multi-Threading Parallel Implementation with ParallelStencil.jl

To implement the parallelization of the algorithm with multi-threading, the ParallelStencil.jl julia package is imported. Also, the code frame is adopted from the diffusion2D_shmem_novis.jl example provided in the ParallelStencil.jl module tutorial on github (https://github.com/omlins/ParallelStencil.jl). For specifics, please see the `GS_stencil.jl`.

The parallelStencil.jl package incorporates multi-threading as a powerful mechanism for parallelization. When using multi-threading with parallelStencil.jl, the computational workload is divided into smaller tasks, and each task is assigned to a separate thread. These threads can then run concurrently on the available GPU cores, allowing for efficient utilization of computational resources. The package leverages Julia's built-in multi-threading capabilities, which include task scheduling and load balancing functionalities. The parallelStencil.jl package intelligently distributes the tasks among the threads, ensuring that the workload is evenly spread across the available cores. Moreover, the package takes advantage of shared memory access, allowing threads to communicate and share data efficiently, reducing the need for expensive data transfers. By exploiting multi-threading, parallelStencil.jl enables users to harness the power of parallel processing at a finer granularity, resulting in improved performance and accelerated stencil computations.

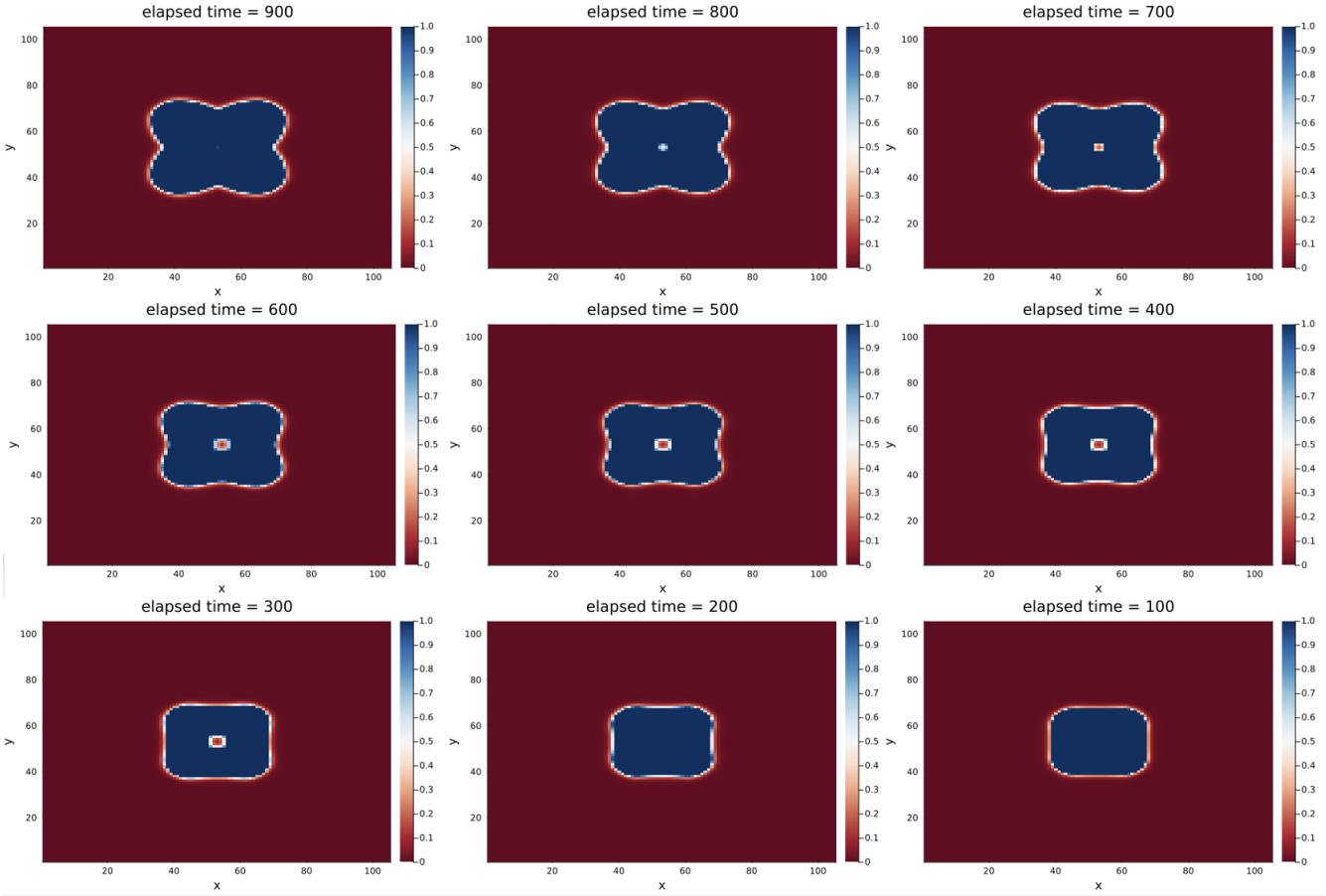

Figure 1: Simulation Result of the serial Gray-Scott Implementation.

6 Performance Benchmarks and Analysis

For the performance analysis, the GPU launch parameters are being varied to benchmark the corresponding effective main memeory access per iteration (A_eff, GB), execution time per iteration (t_it, s) and effective memory throughput (T_eff, GB/s), and the results are shown in Table 1.

	t_it	T_eff
$threads = (32, 32)$	6.74e-6	52.34421364985163
$threads = (16, 16)$	7.102e-6	49.67614756406646
$threads = (8, 8)$	8.662e-6	40.729623643500354
$threads = (4, 4)$	9.838e-6	35.86094734702176
$threads = (2, 2)$	1.2806e-5	27.54958613150086
$threads = (1, 1)$	3.131e-5	11.267965506228043

Table 1: Benchmark results for different GPU lauch parameters.

The execution time per iteration (t_it), speed-up and effective memory throughput (T_eff) with different threads parameters are plot in Figure 2, 3, 4 respectively. The speed-up is calculated by dividing

the reference execution time per iteration with threads=(1,1) by the actual execution time per iteration under specific thread parameters, i.e.,

$$Speedup[i] = \frac{t\_it[threads = (1,1)]}{t\_it[threads = (i,i)]} \tag{4}$$

From the speedup plot in Figure 3, we see that speedup of multithreading with ParallelStencil.jl does not increase linearly with the number of threads. Instead, the derivative of the speedup over the thread number decrease with the increase of the threads. And there seems to be a plateau around 4.5 speedup as the thread number increase further to 32. This can happen due to various factors which can affect the parallel performance. Here are some possible reasons explaining why the speedup doesn't scale linearly:

1. Memory Access Patterns: In the code, shared memory is used to store the local copies of the U and V arrays. Each thread reads from and writes to the shared memory, which introduces memory access patterns that are not easily optimized for parallel execution. As the number of threads increases, contention for accessing the shared memory can occur, as indicated by Figure 4 where the effective memory throughput doesn't scale linearly with the number of thread, therefore leading to increased overhead and slower performance.

2. Synchronization Overhead: The code uses the @sync_threads() macro to synchronize the threads at certain points to ensure correct data dependencies. Synchronization introduces overhead and can limit the parallel scalability of the code. As the number of threads increases, the cost of synchronization becomes more significant, affecting the overall performance.

3. Block Size: The block size used for parallel execution is defined by the threads variable. In the code, (32, 32) for example is used as the block size. Choosing an inappropriate block size can lead to inefficient GPU utilization or suboptimal thread distribution on CPUs. If the block size does not align well with the underlying hardware architecture, the performance may not scale linearly with the number of threads.

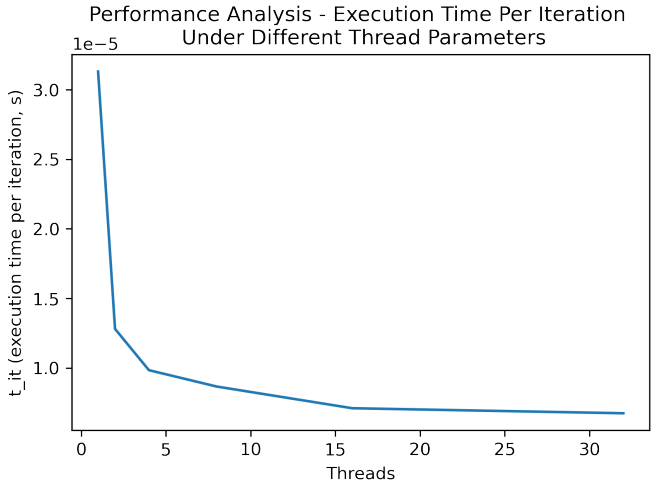

Figure 2: Execution Time Per Iteration Under Different Thread Parameters.

7 Future Work for Further Parallelization with MPIHaloArrays.jl

To further improve the performance of the algorithm, distributed memory parallelism could be exploited. The entire domain of interest can be divided into many subdomains and each subdomain can be distributed

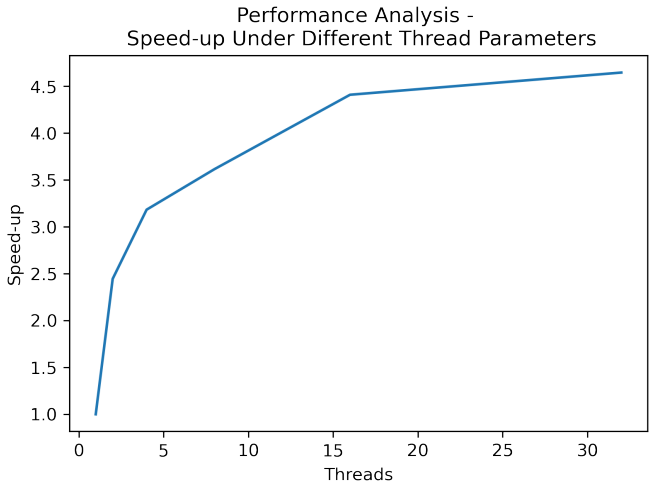

Figure 3: Speed-up Under Different Thread Parameters.

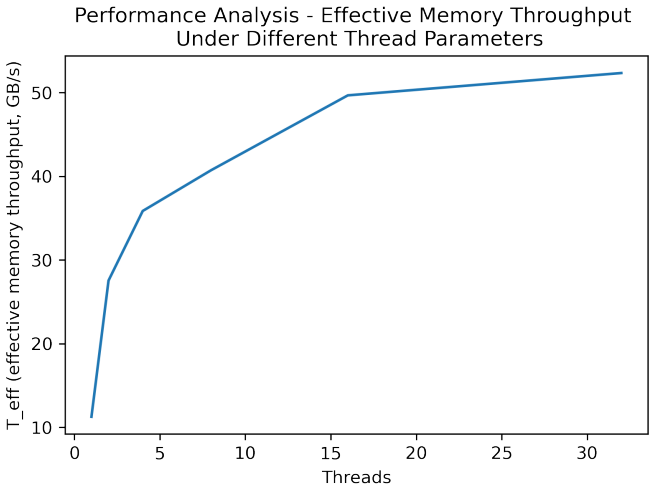

Figure 4: Effective Memory Throughput Under Different Thread Parameters.

to a single processor / rank for parallel computation, as depicted in Figure 5. Since the algorithm is based on 5-point stencil calculations, which involves the values at nearest neighbors, each two neighboring processes need to communicate in order to find the solution near the boundary. For example, process 0 needs to know the value of the solution in B in order to calculate the solution at the grid point A. Similarly, process 1 needs to know the value at point C in order to calculate the solution at the grid point D. These values are unknown by the processes, until communication between processes 0 and 1 occurs. Therefore, neighboring processes need to exchange information messages to append ghost cells to the boundaries of each subdomain MPI communication, as depicted by Figure 6. For the implementation, the pseudo code frame is adopted from the 04-diffusion2d.jl example provided in the MPIHaloArrays.jl module tutorial on github (https://github.com/smillerc/MPIHaloArrays.jl). For specifics, please see the `GS_mpi.jl`. The number of processes is temporarily set at 1 only for the purpose of testing.

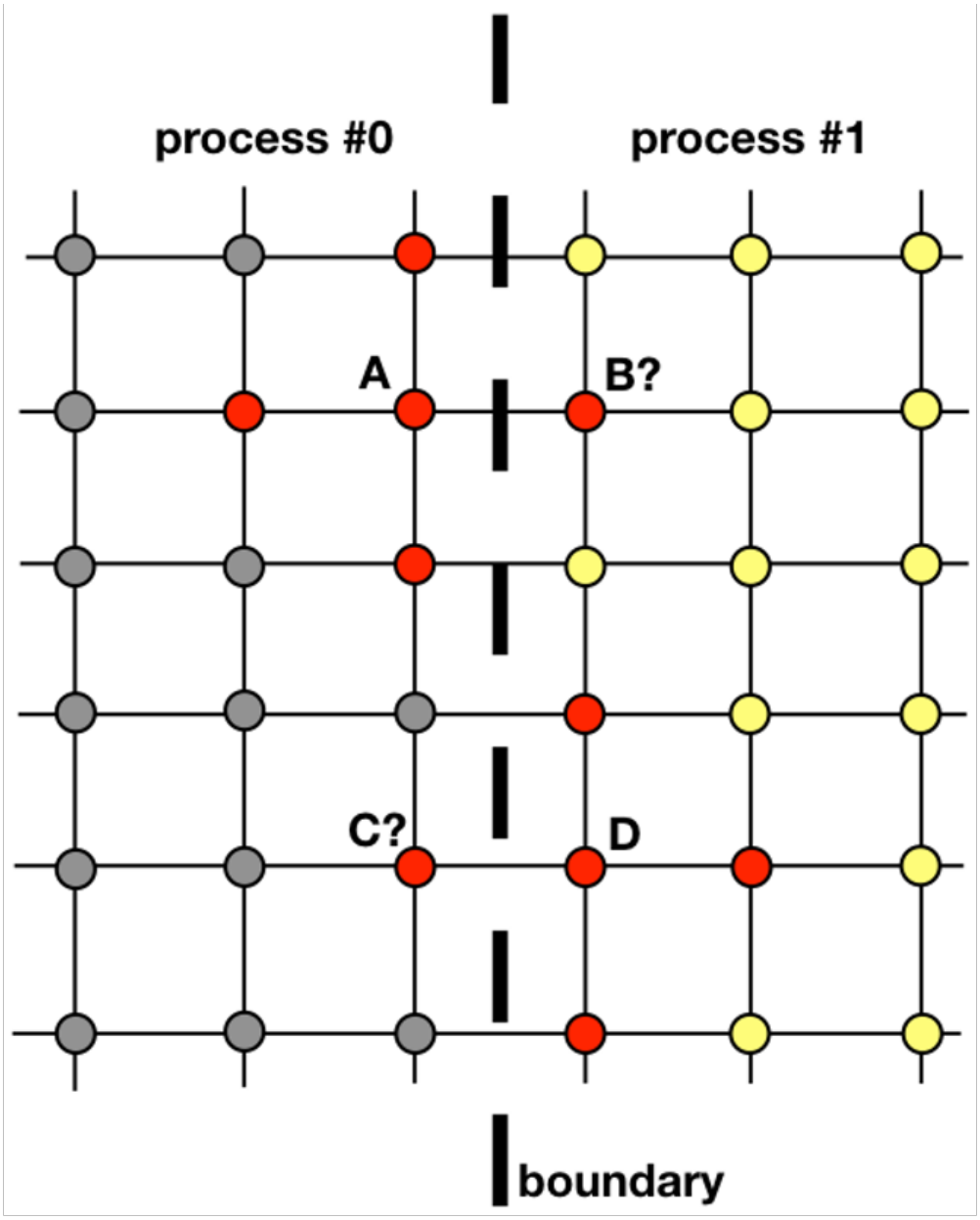

Figure 5: 5-point Stencil MPI Scheme.

References

[1] Hawick, K. A. and Playne, D. P. (2010). Automated and parallel code generation for finite-differencing stencils with arbitrary data types. Procedia Computer Science, 53(1): 1795-1803.

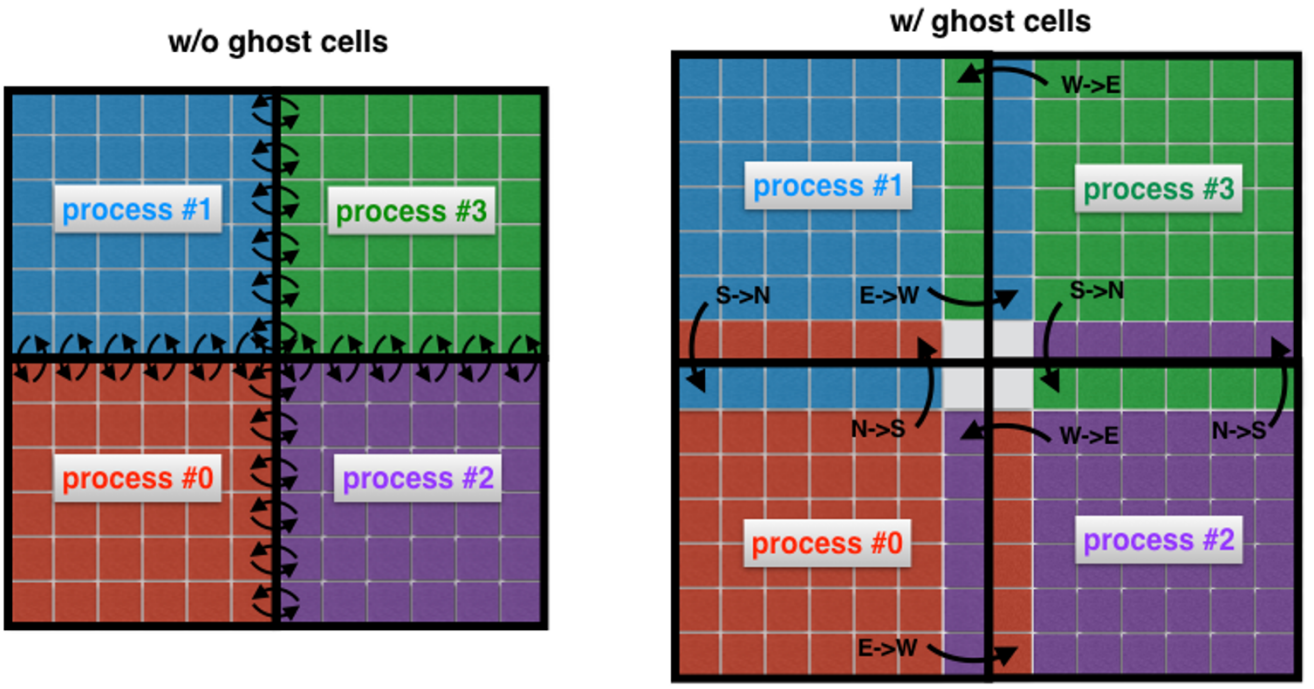

Figure 6: MPI communication with ghost cells.