Assignment #3 due 11/30 @11:59pm

WARNING - VERY LONG INSTRUCTIONS, BUT PLEASE READ MULTIPLE TIMES TO UNDERSTAND.

Part 0. Overview

Your goal is to take the scalar code that implements the Aliev-Panfilov cardiac simulation we discussed in class and parallelize it using MPI.
Your MPI implementation should scale to large numbers of cores, which you will test on Expanse (a real supercomputer) at SDSC. You may additionally try to parallelize the computation using vectorization or other techniques for extra credit. You will develop your code on the sorken cluster/Expanse's shared job queue and test it for small numbers of cores. Once you are satisfied with your performance on sorken cluster/Expanse's shared job queue, you can try your code on Expanse's dedicated compute queue which has ~93,000 cores available (YOU WILL USE ONLY A FRACTION OF THEM ~100's).

The assignment includes a report, which is, as usual, an important part of this assignment.

Your work will be assessed on 4 factors:

- performance - in terms of raw numbers and scalability with an emphasis on scalability and operating cost.
- correctness
- well displayed results including comments, explanations
- insights about how your program achieves (or doesn't achieve) its performance and how you got there.

Part 1. Get the code

As in assignment #1 and #2, we have provided some starter code. Here is the link:

https://classroom.github.com/a/UjXadTAJ

For this assignment you may work in group of 2. Groups are encouraged to increase the amount of work the group may do, increase debate and intellectual discussion and conserve resources.

Part 1: Starter Code

We've provided you with a starter code that implements the Aliev-Panfilov solver, which was discussed in class.

All of your changes will made to just two files:

```
solve.cpp       helper.cpp
```

Solve.cpp contains the solver, helper.cpp provides various helper functions such as storage allocation. (You may, therefore, change the storage allocation scheme if you wish.) The solver is computationally intensive, and so the focus of your parallelization and performance programming effort will be on this routine. You'll notice that the solver includes both the fused and unfused versions of the main computational loop. You can control which version to use via the FUSED macro, defined in solve.cpp.

To build your code, use the provided Makefile, which sets the appropriate flags for the compiler via the "arch" file located in$(PUB)/Arch/arch.gnu-c++11.generic. The Makefile produces a target called apf. Do not change the name of this target. While you may modify any files not listed above, remember that we will substitute fresh copies when we grade your assignment, so don't rely on any custom changes as they will be lost during the grading process.

To get you started, we have included some basic calls to MPI in the provided code. These calls are conditionally compiled under control of the _MPI_ C-preprocessor macro. To facilitate conditional compilation, the Makefile will cause this macro to be asserted when you set the mpi=1 on the Make command line.

```
make mpi=1
```

You can achieve the same effect by uncommenting the Makefile variable definition in the provided Makefile:

```
# If you want to compile with MPI enabled,

# uncomment this line
```

```
# mpi = 1
```

Additional code is included to show you how to restrict output to one MPI process, for example, when printing information to stdout. We have also included some very basic code to support SIMD intrinsics , which is enabled via the vec=1 flag (which asserts the SSE_VEC C pre-processor macro, enabling conditionally compiled code [the SSE name is leftover from pre AVX days.]).

We also have provided some sample slurm scripts. Be sure to use the correct batch file for the capability you require; for example, the parallel batch job won't run correctly if you haven't enabled MPI when building your code. It will deadlock, tying up resources for the specified duration of the job. The starter code enables you to input the processor geometry via two command line flags: -x and -y. The code will report an error if the product of x and y do not equal the number of cores you requested when invoking MPI via mpirun or srun (-n processess).

You'll notice that the provided code uses a different naming convention for the bounds of the computational box. In particular, while we use the single letter n (e.g N0, N1, N2) in this document, the code uses separate bounds for the x and y axes: the variables m and n. This convention enables the code to accommodate non-square local problem sizes that arise when the data has been partitioned on non-square processor geometries. Also, we "flatten" the arrays so that all solution arrays are 1-dimensional and are subscripted with just 1 index.

Also note the conventions used to index the 2D solution arrays: the indices are swapped, with the i index being the most rapidly varying index, that is, indexing along the last dimension of the array. Thus, array elements in the same row have different i indices and elements i-1, i and i+1 are adjacent in memory.

The simulator has various options, controlled via command line arguments (parsed in in cmdLine.cpp), with default values set in apf.cpp:

-n <int>	Number of mesh points in the x and y dimensions
-i <int>	Duration of simulation in iterations
-s <int>	Print statistics, L2 and L∞ norms of the excitation variable at specified regular intervals (in timesteps). This feature is useful when debugging.
-p <int>	Plot the excitation variable, at the specified regular intervals (in timesteps). Useful when debugging. Only works on non mpi code.

-x <int>

-y <int>

x and y-axis of the processor geometry (Used only for your MPI implementation)

-k Disable communication (Used only for your MPI implementation)

-d Enable debugging output. Sets a variable called `debug` that you may use to debug your code.

Thus, the following command line will conduct the simulation on a 150 × 150 box, run for 400 iterations, plot the excitation variable every 20 iterations, and print out summary values of the excitation variable every 10 iterations.

```
./apf -n 150 -i 400 -s -10 -p 20
```

If we leave off the -s and -p flags, then the output is as follows. The last line of output, beginning with the string "@" outputs performance data along with relevant parameters, that is suitable for automated extraction from a sequence of runs.

```
[Simulation begins] Wed Nov  4 14:30:41 2015


dt= 0.0639292, # iters = 400

m x n = 150 x 150

processor geometry: 1 x 1


Compiled with MPI ENABLED


[Simulation completes] Wed Nov  4 14:30:41 2015


End at iteration 400

Max norm: 9.928374e-01, L2norm: 6.125795e-01

Running Time: 0.216294 sec. [1.17 GFlop/sec]
```

```
  M x N    px x py Comm?    #iter   T_p, Gflops         Linf, L2

@  150  150      1  1         Y       400    0.2163  1.165      9.92837e-01
6.12580e-01
```

The plotting capability, controlled via the -p flag, uses gnuplot, and may be useful in debugging. Do not enable plotting when you are measuring running times for your performance studies. The plotter capability only works on a single core and is discussed in more detail later in this writeup (but you may make it work in the multi processor case for extra credit).

Part 2. Assignment Development Process

Here are some suggestions on how to approach this problem. Read this section carefully as it will guide you on the development process. It also will give you the data you need to generate the report (see Performance Study and Analysis below).

For instructions on how to do development (e.g. interactive sessions) on expanse, see

https://sites.google.com/eng.ucsd.edu/cse260fall2022/compute-resources/expanse

1. Parallelize the code using MPI

You will need to support arbitrary one and two-dimensional processor geometries consistent with the command-line flags -x, and -y. (see cmdLine.cpp). We suggest you start with one-dimensional geometries which should work fine for small numbers of cores (<=8). You may find it easier to deal with one-dimensional geometries (the ones with unit stride) while you debug your ghosting/halo code and then expand that code to handle the 2D case.

The program needs only support square meshes (n x n). However, the square meshes will not usually be integer multiples of your processor geometries which may be rectangular. So your program will need to deal with rectangular sub-meshes.

Your code must correctly handle the case when the processor geometry doesn't divide the mesh evenly. (It may be difficult to ensure that, for certain nearby values of n, performance may vary by 25% or more.)

In particular, your program must divide the mesh as evenly as possible using partitions that have the following property:

The amount of work assigned to the most heavily and most lightly-loaded process along a given dimension will differ by not more than one row (and/or column) of the solution array.

Note: you do not need to collect the result matrices as we are only going to look at the error bounds (Linf and L2). In the real world, you would, of course, be interested in the final data array and would output it to a display or file (see extra credit).

You must distribute the initial conditions to the worker processes from process 0 (rank 0). The program sets up an initial condition that consists of one-half plane of 0's and one-half plane of 1's. **Pretend** as if the initial condition values were read from a file on process 0 and distribute the values to the other worker ranks rather than hardcoding the initial conditions in each rank. That is, rank 0 provides the initial condition to all the worker ranks.

2. Optimize your code

The goal of this assignment is to get decent performance and strong scaling. We will give you some guidance as to expected performance. We have provided a stripped binary mpi reference in :

$PUB/HW/hw3/apf-ref

NOTE: $PUB should be : /share/public/cse260-sp22 on Sorken

$PUB should be : /expanse/lustre/projects/csd720/bwchin/public on Expanse

1. Optimize communication to enable the application to scale strongly: for sufficiently large problems, performance should be proportional to the number of cores.
2. Optimize the inner loops. There are two (non-exclusive) possibilities.
    1. Improve cache locality
    2. Get the code to vectorize (play around with vectorization options, but don't get hung up on this. aggressive vectorization applies to the extra credit).

3. Conduct strong scaling studies and find the best geometry. You'll run over different ranges of parallelism:

1. 1 to 16 cores (N0 : n = 800)

2. 16 - to 128 cores (N1: n = 1800)
3. 128 to 384 cores (note: we might limit this depending on how busy the expanse is and how much CPU money we have available). (N2: n = 8000)

Each of the above ranges should use a different value of n, which we'll call N0, N1 and N2. We suggest an iteration value i in the description below. You may need to adjust that "i" when performing scaling runs to either larger or smaller so that your program executes in between 10 seconds and 1 minute. For a strict strong scaling experiment, all parameters should remain the constant for the different numbers of processors.

Do not attempt to run beyond 16 cores unless you have observed robust strong scaling results on 16 cores (goal #1). Similarly, do not attempt the 16 to 128 core goal unless you have obtained robust strong scaling results on 16 cores. This procedure is intended to save our allocation, and it's important to never run a poorly performing program on larger numbers of cores, especially at the highest end of >=256 cores (When user's apply for large allocations on supercomputers, they are required to demonstrate good scaling) Each student has a restricted number of SUs (340 per account). Each core consumes 1 SU if the core runs for an hour. Our jobs run significantly less than one hour so we should consume fewer SUs. When you run on the **`shared`** queue you pay only for the cores you use. When you run on the **`compute`** queue, you are charged for one compute node which is 128 cores whether you use all 128 cores or not. SUs are charged based on the estimated time in your sbatch script. If you request 128 cores for 30 minutes, you will be charged 64 SUs. We will always request no more than 3 minutes of time, typically 1 minute.

Use **`expanse-client user`** to see how many SUs you have and how many are remaining. The remaining amount shown is our entire allocation (not your own allocation).

THE FOLLOWING ASSUMES SORKEN IS WORKING PROPERLY. AS OF THIS WRITING, SORKEN CAN BE USED TO PROVE FUNCTIONALITY BUT IT IS NOT PERFORMING AS EXPECTED (so you won't see proper scaling). YOU SHOULD DEBUG FUNCTIONALITY ON SORKEN (no SUs required!) BUT YOU WILL HAVE TO DO YOUR SCALING EXPERIMENTS ON EXPANSE. USE THE SHARED OR DEBUG partition ON EXPANSE FOR UP TO 16 CORES SCALING EXPERIMENTS.

a) Start on Sorken. Set n=800 (N0) and # iterations (niter) i=~2000; the provided code will complete in about 9 seconds on one core of Sorken. Compare the single processor performance of your parallel MPI code against the performance of the original provided code to assess any overheads introduced in parallelizing the code. These overheads should be negligible; if not, investigate the source of the overhead and eliminate it.

b) Next, conduct a strong scaling study on sorken: observe the running time as you successively increase the number of cores, starting at p=1 core, p=2, p=4, p=8, p=12, p=16, while keeping N0 fixed. We've picked a small value of N0 so that your runs should scale nearly perfectly to 16 cores. Your code should run in under 10 seconds on 1 core of sorken and run in about 1 second on 16 cores. (There is a batch script called N0_sorken-1.slurm) Use a 1D processor geometry to avoid the need to search for the optimal geometry. Your goal here is to gain insight into using MPI effectively and to collect and collate your performance data.

c) Measure communication overhead on 16 cores. To do this, use the indirect method, described below, for timing the simulation without communication. You should find that there is relatively little or no communication overhead. Why is this the case?

d) Alternatively, to do this on expanse in case sorken is not working correctly, use the sample batch file **N0_expanse-16.slurm. Modify the ntasks in the sbatch file to match the number of cores you are using.**

NOTE: For initial development on Expanse you should be using 16 cores or less and use Expanse's shared partition or debug partition, not the compute partition which blocks the entire node. Performance should scale absolutely linearly for up to 16 cores. Then you can shift to the compute partition for performance studies on larger core counts. Reduce the search space by doing experiments on smaller numbers of cores and eliminating some candidate geometries. Minimize large scale runs on expanse.

e) Repeat steps (b) and (c) on Expanse for 16 to 128 cores and measure communication overhead. Keeping i constant (for our model, we needed to set i ~100,000 to get reasonable run times), set n=1800 (N1). We have set-up an example batch file called **N1_expanse-16.slurm. (sbatch N1_expanse-16.slurm)** ***WE HAVE SET A DEFAULT TIME LIMIT IN THIS SCRIPT SO YOUR PROGRAM WON'T USE TOO MUCH ALLOCATION. MAKE SURE YOU DO NOT CHANGE THIS LIMIT.***

Once you are running beyond 16 cores, you will need to enable 2D processor geometries. For each value of p, determine the optimal geometry, and use that geometry when reporting performance. You may find that more than one geometry yields maximum performance. **For the largest value of p (128), report all interesting geometries that yield the top performance.** For our purposes, we'll treat the top contenders as those coming within 10% of the top performer(s). Can you see a pattern emerge in the choice of geometry? This may help you prune the search for optimal geometry at the next level of parallelism (step (f), coming next). It should not be necessary to conduct an exhaustive study of optimal geometry, as you may be able to eliminate obvious non-performers, in particular 1D geometries. Avoid these and save time for more interesting studies!

Develop an intuition about geometry selection based on empirical observations and share your insight on Piazza.

f) Next attempt to scale to 128, 256, 384 cores (depending on machine usage, we may increase this limit to 512) . **Get as far as you can, but proceed carefully as our bank account is charged for # cores * time for each run!** At 384 cores you should be running at ~ 2 TFlops/sec). Start with P=`128. Keeping i=~8000 (or adjust as needed to get reasonable (10-20s) run times for this series of experiments), set n to 8000 (N2).

Copy and modify **N1_expanse-16.slurm** to create submission scripts (e.g. N2_slurm-64.slurm). You will need a different job submission script for each different number of processing **nodes** (see slurm script nodes and tasks per node). Each 128 cores requires one more node. So nodes = 1 for 128 cores, nodes = 2 for 256 cores. Don't run 128 core simulations with more than 1 node as this will waste your allocation. Avoid wasting processors (cores and nodes) to ensure that all cores the script asks for get used by the job. **Do not allow the time limit for the job to exceed 3 minutes**. **This limit has been set up in the job script and should not be changed.**

Use your intuition from studying 16 to 64 processors to develop a strategy for picking the optimal geometries. Avoid exhaustive search and do not do geometry searches at scale Before attempting this study, be sure that your code scales strongly on up to 128 cores and that all known bugs have been fixed.

g) Once you have done the above, run only once the optimal geometry obtained in step f on 128, 256 and 384 cores for n=1800(N1) and i~100000. Plot these values on the same graph as step e and comment on their differences from the results obtained in step f.

h) Based on your execution time in part (f) - Calculate the **product of (Execution Time * #Cores)** used for i=8000 N2=8000 for each of 128, 256 and 384 cores. What does this result indicate about the cost of compute as a function of core count? In your report, make sure to address scaling behavior. Is it sub linear, close to linear or superlinear with respect to increasing core count.

Part 3: Testing your code

Your code will be tested for correctness and performance against various combinations of the n, niter (i) and processor geometry. Test your code by comparing it against the provided implementation. You can catch obvious errors using the graphical output (see below), but a more precise test is to compare two summary quantities reported by the simulator: the L2 norm and the L∞ norm. (These are included in the 1 line performance

output). The former value is obtained by summing the squares of all solution k values on the mesh, dividing by the number of points in the mesh (normalized), and taking the square root of the result. For the latter measure, we take the "absolute max," which is the maximum of the absolute value of the solution, taken overall points. While there may be slight variations in the last digits of the reported quantities, variations in the first few digits indicate an error.

For example, you may find that after adding a new performance optimization that your code indeed ran faster but now the L2 of the errors differ in the 5th decimal place, while the Max errors are the same. There is probably an error in your code. The L2 norm is giving a clearer picture of what is going on here, as it takes the square root of the sums of the squares of the differences (look at the code to see how the norms are computed) the max takes the max of the absolute values of those differences. So what's happening is you are probably introducing small errors in many elements of the solution mesh, but these are masked by some maximum value and hence not visible with the max norm.

The error is small, so any penalty (if we even assess in this case) would not be severe. Write up your results and take some time time to ask 3 questions, which you should add as an addendum to your plots:

1. Where is the error coming from? Make your best guess by examining the code. Show us code fragments as needed to clarify
2. How severe is it? Try successively doubling the number of iterations with the -i flag, say for 1800, 4000 and 8000 iterations and see how much the errors grows.
3. How much of an increase in performance did you get at the cost of introducing this error? Was it worth it? If you got say only 10% performance improvement, probably not worth it. But if you went much faster,it could be worth it.

The code and Makefile have been set up to enable you to compile without MPI, which may come in handy when debugging. To compile without MPI, be sure that the following line has been commented in the Makefile:

```
mpi=1
```

This setting defines the _MPI_ macro which causes certain code to be conditionally compiled. To take advantage of this feature in any code you write, follow the convention used in the provided starter code.

Plotting capability

As mentioned previously, the code includes plotting capability to observe the simulation dynamics which can be helpful when debugging.

The simulations will produce a spiral pattern that evolves with time. However, if you set the number of iterations (-i) to a sufficiently large value, the organized spiral pattern will break up. This indicates non-physical behavior, and the length of time that the simulator produces physically meaningful results depends on the mesh size N.

Be sure that, while collecting performance data, that you have not enabled plotting (-p) or statistics reporting (-s).

The plotter (Plotting.cpp) interfaces with gnuplot, which is available on all platforms in use by this course (and is freeware). Remote visual displays can be slow, so either work from a campus machine, or increase the frequency to at least 20 (-p 20). The summary statistics may be more useful than plotting in some cases. If a display window does not appear on your screen, then your ssh display settings may not be set correctly. If you are connecting to the remote machine from a Linux platform using ssh, use the -X option:

```
$ ssh -X yourlogin@sorken.ucsd.edu
```

If you are connecting from a Windows platform, and are using a 'secure shell client' AND some xserver. We like mobaxterm as it's easy to install. (Others include xwin32, or cygwin).

On macs, install XQuartz and log in from a terminal window with ssh -X.

If you see a large number of warning messages to an interaction between OpenMPI and Unix Pipes, such as following text:"An MPI process has executed an operation involving a call to the "fork()" system call to create a child process." To avoid this message set the following shell environment variable before running mpirun (e.g. via your shell startup scripts):

```
export OMPI_MCA_mpi_warn_on_fork="0"
```

Part 4: How to take timings

Use the non-interactive batch queues to collect timings. Use the interactive queues (when available) to tune your code's performance. On Sorken or Expanse, use an interactive job node to debug your code for functionality . On Expanse, use the MPI batch queues (shared and compute. Use shared when debugging and using a small number of cores) to collect timings. Instructions for submitting jobs via the batch queue are described in batch queue notes. Pay attention to the time limit (don't abuse this - leave it alone) and nodes. Each node on Expanse has 128 cores, so for a 256 core simulation you will have

#SBATCH --nodes=2

Take timings using the provided timer code, which measures wall clock time. With MPI enabled, the timer uses MPI_Wtime() otherwise it uses gettimeofday() as in the first assignment. Both timers collect wall clock time. (Some old, but potentially useful Instructions for how to use the timer to take timings in MPI are described in the document Using MPI on Bang. The man page for MPI_Wtime is available using the man command.

The indirect method for measuring communication

The running time for our application is the time it takes for the last processor to finish. We time the entire run, not individual iterations. However, you will see in the Driver code we exclude MPI init and finalize. The driver code also excludes Init() which sets up the problem. You may want to exclude this time as well when you distribute the problem to the different ranks.

It can be difficult to measure communication times accurately, owing to the short time durations involved. Therefore, we will use an indirect measurement technique. We do this by disabling the communication of ghost cells and subtracting the resultant time from that obtained from a run that includes communication. The -k command line option (cmdLine.cpp) can be used to measure communication using the indirect method; when the option is asserted, your code should disable all MPI calls that handle ghost cell communication (e.g. Send, Isend, Ireceive and Receive), as well as any code needed to pack and unpack message buffers. Do not disable the calls to MPI_Init and MPI_Finalize as these are needed to enable your program to run under MPI (and we are not timing them anyway). Similarly keep any other message-passing calls that don't involve the ghost cell communication directly but are essential for the correct operation of your code. Note that, when you shut off ghost cell communication, your code will produce incorrect results. This is to be expected and won't introduce errors in measured performance unless numerical exceptions are generated. (You'll know when this happens; the L2 and Linf errors will appear as nan or -nan).

You may sometimes see interference from other jobs on Expanse as it's a shared resource. If you observe greater variation in run times, try averaging a few runs. If this fails to mask the variation there may be another cause and this should be investigated. Exclude outlier timings in your plots, but note them in your report if significant. Report the minimum running time rather than the average.

When doing performance measurements, you may need to adjust the iteration count so that the running time never drops below 2 seconds.

Part 5: Development Environment

To build the starter code, be sure that your environment has been set up as explained in compute resources.

Both Sorken and Expanse support interactive and non-interactive access. Use Sorken for development but do performance experiments at scale on Expanse. To run interactively on Expanse, use the handy alias `inter (as in assignment #2)`. You can find this alias in $PUB/PROFILES. (set $PUB first to /expanse/lustre/projects/csd720/bwchin/public).

It is not possible to run MPI jobs on a login node on either Sorken or Expanse; any attempt will result in an error message reported by the provided code. Be courteous to other users signing out an interactive node when you no longer need it (and saving your SU budget on Expanse). You can always tell when you are using a login node by running hostname.

$ hostname // on sorken

sorken.ucsd.edu

$ hostname // on expanse

Login01 or Login02

MPI Documentation

https://mpitutorial.com/tutorials/

Man pages for MPI are available on both systems as well as here : http://www.mpich.org/static/docs/latest/www3/

Note that on sorken and expanse we are using openmpi but you are welcome to experiment with other mpi packages like mvapich2 and other compilers (we're using gcc but expanse has other compilers, notably the amd compiler aocc and intel). https://www.sdsc.edu/support/user_guides/expanse.html#compiling

Prof. Baden also has some nice MPI resources indexed here:

http://cseweb.ucsd.edu/~baden/Doc/mpi.html

Part 6. Grading

The report should be **6 to 10** pages including plots. (please limit the length to 10 pages!)

This is a general overview - see report template for actual data and sections we want.

Section 1 - Performance Study

We would like to see strong scaling exhibited by your code. We will measure the following:

- We want to see the performance with n=1800 (N1) for 16, 32 and 64 and 128 cores. We would like to see strong scaling of at least ~90% on the intervals 16->32, 32->64, 64->128
- We want to see performance with n=8000 (N2) for 128, 256, and 384 cores. We want to see strong scaling ~1 (or at least 95%) from 128->256, You may get less strong scaling from 128 to 256 (you will want to describe why in your analysis) or super scaling on some intervals. You will want to theorize about this in your report.
- We want to see approximately ~>1.5 - 2 TF performance at 384 cores (N2: n=8000, i=8000). You may see greater or less so you will want to explain what you think might be happening.
- **Note**: For full points you should achieve ~200GF or more on 16 cores (N1) and ~700GF or more on 64 cores **and** achieve good (~90) scaling as mentioned above. If you do not meet these performance goals but get good scaling, points will be deducted. Similarly, good performance (above these thresholds) but poor scaling will result in point deduction.

We will run your code on Expanse using our own makefile and your OPTIONS.TXT.

Section 2 - Analysis

How does your program work? A complete description should include:

- setting up the initial conditions amongst all the processes
- how do you handle ghost cell exchanges
- how do you handle boundary cases on the edges of the global matrix
- optimizations to the computation kernel (vectorization, cache affects, etc).
- how does your program evenly distribute the work such that no processor has more than 1 row or column more work than any other processor
- What MPI calls are you using and how do you use them.
- Consider tools such as TAU to **justify or gain insight** into your analysis/report (see https://sites.google.com/eng.ucsd.edu/cse260-spring-22/compute-resources/expanse).

Use of pseudo code, flow charts, diagrams that help describe your program are encouraged but **DO NOT INCLUDE FULL SOURCE code listings.**

Feel free to include data to support your results from section 1 (e.g. profiling tools).

Part7 EC-a.: Extra Credit (up to 2.5 pts)

The current plotting routine only works on a single thread. Devise an efficient method of plotting results on a multi-core (MPI) application. Demonstrate your code by showing

snapshot updates (at least 3) of an evolving simulation. In your implementation, does the node that does the plotting also do computation? If so, is this a problem worth solving?

Part 7 EC-b.: Extra Credit (up to 2.5 pts)

Using the amd compiler (aocc), get the code to vectorize using AVX2 using compiler hints or hand vectorization and report the performance increase. Describe what you needed to do (flags, code changes, etc). to get the code to vectorize and show that it actually did vectorize. (provide us with an easy way to turn these optimizations on or off - default should be off). Show the speedup on scalar code and on 64, 128, 256 and 384 cores. Does the code scale as well as it did before aggressive vectorization? Explain why it does or does not.

You may also vectorize (SIMDize) your code by hand if you wish. Be sure to list any assumptions about mesh size if your code has some limitations.

Part 8: Requirements

You must use double precision floating point numbers (64 bits).

Part 9: Turn it in

When you have completed your assignment, you will submit your final versions of the code. You should be submitting periodically with comments so we may see your progress. The steps for final submission are no different.

1. run *git status* to see what files have changed.

2. use *git add* to stage your changes for commit.

3. use *git commit -m "FINAL SUBMISSION"* to commit your staged files into your local repository. The -m FINAL SUBMISSION lets us know that you intend this to be your final

submission. If you submit again before the deadline, we will also consider the last submission before the deadline as your final (but please also tag that with FINAL SUBMISSION message).

4. push your changes into github with *git push origin main* . For those more familiar with git, you may also be merging branches at this point. **We want the final code to be on the main branch. Don't forget this step. IF you don't push your changes, we won't see them.**

5. Surf over to your repository or clone a new copy to see if your changes are really there.

Things to watch out for:

- Watch out for (and resolve) merge conflicts before your final push!!

If you and your partner have modified files on multiple computers, you may have merge issues. This happens because your code from one repository is out of sync with your code from another and git doesn't know who to merge the files. MERGE CONFLICTS will prevent your code from compiling so be sure to resolve these before you do your final push.

Don't let these problems bite you at the end. If you add/commit/push often, this is less likely to happen.

- As a team, push just the team repository. You shouldn't have any others.
- Be careful with the 'FINAL SUBMISSION' message. You shouldn't have multiple repositories (but if you do somehow), make sure only the one you want graded as this message
- You can submit multiple FINAL SUBMISSIONS before the deadline. We will grade the latest one before the deadline.
- Late work will be accepted with a 10% deduction per 24 hours up to 72 hours. After 72 hours, we will consider the last submission before the deadline tagged as 'FINAL SUBMISSION' or the last submission before the deadline. (basically no late work after 72 hours).
- DO NOT commit binaries - just source code. PDF and XLS files are okay.

Required files in the submission

Source code

Will will be using our framework. But do include all of your source files (including your Makefile and slurm scripts) in case we have trouble compiling, we can refer to your environment. Include a file called OPTIONS.TXT if you use other than the default Makefile options.

We expect to be able to do the following:

make `cat OPTIONS.TXT`

to make your program.

Two text documents

You must include the two following documents (10% penalty will be accessed if forms are missing or not filled out properly). We will not report a grade without these forms.

1. teameval.txt - completed self-evaluation discussing your work. If in a team this form addresses the division of labor and other aspects of how the team worked or didn't work.

2. A file named DECLARATION.txt. This was included in the starter code. Just sign (type) your name(s) and dates and acknowledge compliance with the academic integrity policy.

Lab report, pdf document

Name this file according to your repository name. For example PA3-jdoe-jbeeb.pdf

Check the report into the gradescope as well.

An important part of your report is your *analysis*. That is, to provide insight that explains your results. For this reason, if you are running out of time, put more effort into the writeup and your analysis rather than trying yet another performance enhancement. It can take time to collect, tabulate and plot results, as well as offer a reasonable explanation. We prefer a deeper analysis involving a few program optimizations over a shallower analysis over a larger number of optimizations. Spend more time discussing the optimizations that made a difference and less on those that made less of a difference. Negative results are also especially valued if you carefully document them, *with the underlying causes*, or best guesses if the causes are unclear.

Do not include full code listings in your writeup as these will be in your turnin.

Take care in how you plot your data, plotting only what's necessary to get your points across. Choose log and linear plots carefully, to take advantage of the plotting space. **You must also include any plotted data in tabular form. (this may be in a separate text file).**

Cite any written works or software as appropriate in the text of the report. Citations should include any software you used that was written by someone else, or that was written by you for purposes other than this class. In such cases you must talk with us (and any code co-authors) prior to get approval.

Assignment #3 background for details on Aliev-Panfilov and grid methods.