Parallelizing Aliev-Panfilov Cardiac Simulation Solver using MPI

Parallel Computation (CSE 260), Assignment III

Mithun Chakaravarrti Dharmaraj
A53235626, mdharmar@eng.ucsd.edu

Naga Harshini Voruganti
A53247771, nvorugan@eng.ucsd.edu

December 1, 2017

Motivation

Simulations play an important role in science, medicine, and engineering. A cardiac electrophysiology simulator can be used for clinical diagnostic and therapeutic purposes. To reduce computational costs, simplifying assumptions are often made to make the simulations feasible. In our clinical example, time may be of the essence in determining an appropriate life-saving treatment. Cell simulators often entail solving a coupled set of a equations: a system of Ordinary Differential Equations (ODEs) together with a Partial Differential Equation (PDE). In this assignment we have parallelized the heart electrophysiology simulator using the Message Passing Interface, MPI.

Assumptions & Notations

- M and N to denote the mesh size in X & Y directions in the global problem.
- Number of mesh points in the X & Y directions are equal ie., M = N.
- Parameters (processor geometry & mesh size) are chosen in such a way that compute resources are utilized to the fullest.
- P_x and P_y to denote the x and y axis of the processor geometry. Thus no. of processes $P = P_x \times P_y$.
- p_{ij} to denote the process in process-row i and process-column j.
- We divide the global problem into P_y rows and P_x columns of subproblems each handled by a separate process.
- m_{ij} and n_{ij} to denote the mesh size in X and Y directions in local computational grid for process p_{ij} .

1. Environment

The Aliev-Panfilov solver was optimized for the following runtime environments. All benchmarks & performance analysis were carried out on hosts with the following configuration.

Compute Node on Bang cluster	Compute Node on Comet cluster
8 x Intel® Xeon® CPU E5345 @ 2.33GHz	24 x Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz
2 sockets & 4 cores-per-socket	2 sockets & 12 cores-per-socket
L1i - 32K ; L1d - 32K ; L2 - 4096K	L1i - 32K ; L1d - 32K ; L2 - 256K ; L3-30720K

2. The Simulator Algorithm

The simulator maintains two *state variables* characterizing the cell electrophysiology model: *excitation* (E) and *recovery* (R). Since we are using the method of *finite differences* to solve the problem, we *discretize* the variables E and R by considering the values only at a regularly spaced set of discrete points. We represent the variables as 2D arrays: E[][] and R[][], respectively. The PDE solver also maintains the voltage at the previous timestep, E_{prev}.

2.1 PDE Step

$$E[i,j] = E_{prev}[i,j] + \alpha*(E_{prev}[i+1,j] + E_{prev}[i-1,j] + E_{prev}[i,j+1] + E_{prev}[i,j-1] - 4*E_{prev}[i,j])$$

2.2 ODE Step

$$E = E - dt*(k*E * (E-a) * (E-1)+E * R)$$

$$R = R + dt*(\varepsilon + \mu_1*R / (E+\mu_2)) * (-R-k*E * (E-b-1))$$

3. Basic Aliev-Panfilov Solver

Let's conduct a performance study on the serial Aliev-Panfilov (APF) solver running the simulator algorithm. Also, to simplify our performance studies, we will run our simulations for a given number of iterations rather than a given amount of simulated time.

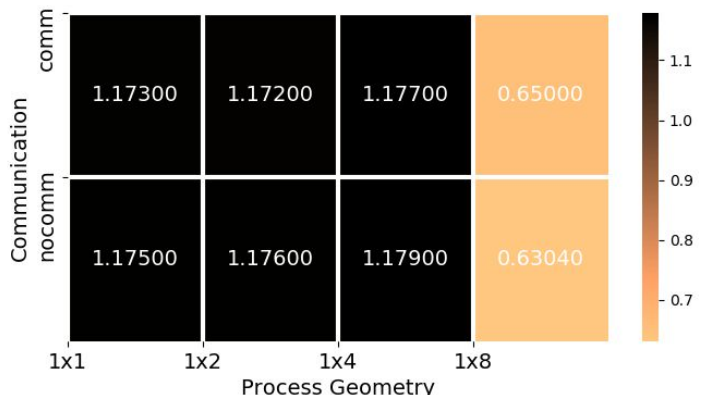


Fig 3.1 Performance of basic solver in GFlop/s for **M=N=400** running for **500 iterations**

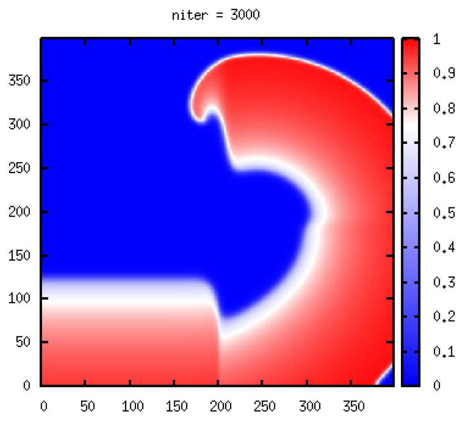


Fig 3.2 Plot for mesh size M=N=400 after 3000th iteration

The above heatmap clearly shows the performance of the serial APF solver running on an 8-core compute node (Bang). Since the algorithm isn't parallelized it doesn't benefit at all with the increase in processes. This means that all the processes run the whole of APF solver in this case which is clearly seen by a huge drop in performance when it's scaled to 8 processes. Fig 3.2 shows the plot of the mesh of size 400 x 400 after 3000th iteration of the basic Aliev-Panfilov algorithm.

4. Optimizations

The solver routine is computationally intensive with `(PDE + 2 x ODE) x #iterations`. To take advantage of the multiple cores, we can parallelize our problem into multiple sub-problems. These problems can be computed independently by their own processes and the results can be communicated to neighbouring process via MPI. The optimum parallelized algorithm is the one that divides the global problem into sufficient local problems to strike a balance between computational overhead within each process and the communication overhead between processes. Let's see the improvements & optimization techniques in detail.

4.1 Dividing Labor Among Processes

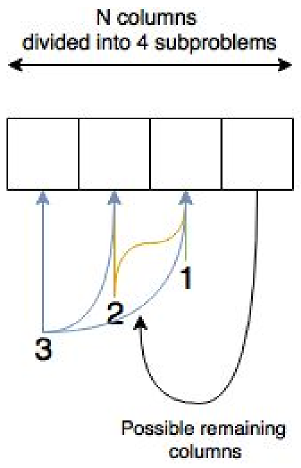


Fig 4.1 Equal division of labor among processes

We divide our global computational grid into smaller subproblems each of which is handled by a separate process. Let's consider one dimensional geometry (P_x x 1) for simplicity of this explanation. To divide N columns efficiently among P_x processes. Two extremes are possible: (To best understand this see Fig 4.1)

1. N is perfectly divisible by P_x and each process gets assigned with N / P_x columns. (Best case)
2. N leaves a maximum remainder of P_x - 1, and we have (P_x - 1) processes to equally distribute the columns. (Worst case)

This method of division stipulates the following condition on our processes.

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 (or column) of the solution array.

Processes are addressed in row-major order, with process of rank 0 is in the top-left corner and the process of rank P_x - 1 in the upper right corner in the processes' grid geometry.

Figure 4.2 shows how our division of labour works for the following parameter space:

`M = N = 1; Px = 3; Py = 2`

Each color in the figure to the right corresponds to a separate process. The matrix shown to the right can either be E_{prev} or R in our problem context. It's also worth noting how this method distributes the work load more or less evenly among processes.

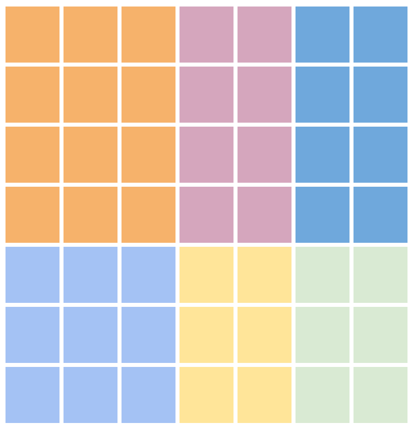


Fig 4.2 Division of Labor in computational grid space.

4.2 Halo Exchange between Processes

Dynamic equations in computing PDE require the values in 4 Manhattan directions: North, South, West, East. When we decompose the domain into subregions, one per process, we can compute the inner mesh values easily. But for the annulus (we call it **Halo Regions** or **Ghost Cells**) we will be needing the values from neighbouring processes. So now, each process does the following two steps in it's solve routine in the same order.

- Transmit halo regions between processes.
- Compute inner region after communication completes.

Loop carried dependencies impose a strict ordering on communication and computation.

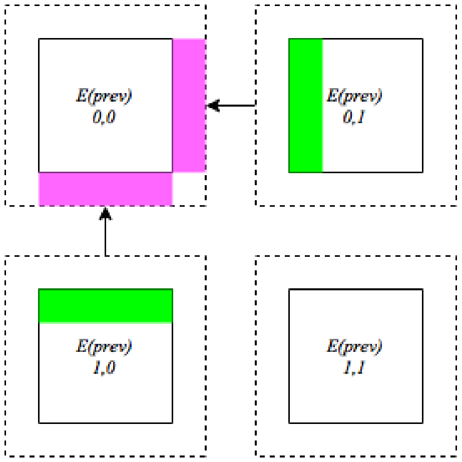


Fig 4.3 Graphical Representation of Halo Regions (Ghost Cells)
idea drawn from Scott B. Baden / CSE 160 / Winter 2010

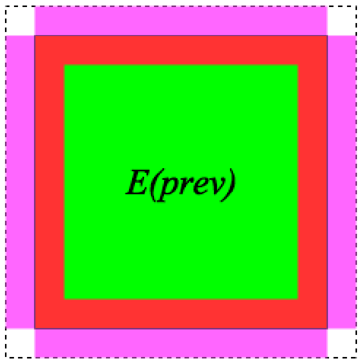


Fig 4.3 Subregions in the domain owned by each process
idea drawn from Scott B. Baden / CSE 160 / Winter 2010

4.2.1 Can we avoid this strict ordering?

Yes. Noticing closely, we observed that only a subset of computations in the domain exhibits loop carried dependencies with respect to the halo regions. We can subdivide this subproblem into - inner region, annulus and halo region to remove dependencies. Now we can sweep the inner region in parallel with communication & the annulus after the communication finishes. This creates a balance between the computation & communication overhead. But however, this won't provide significant improvement in case of smaller mesh sizes with crowded process grid geometry because the computations involved in smaller meshes are not too intensive to take advantage of the communication overhead. Our current implementation stipulates strict ordering and doesn't parallelize inner region computations with communications.

4.2.2 Reasoning our implementation

In our implementation, we account for the ghost cells by simply padding each computational block with an extra border of entries. This renders the numerical integration a simple task. Since memory is not shared between processes, we use MPI to facilitate ghost cell communication. Since the west and east ghost cells are non-contiguous in our padded subproblem, we must pack the outgoing message into compact form before sending. Similarly, we must unpack the incoming message once it is received.

This doesn't sound too convincing, but consider the following alternative:

Instead of extending the computational block, we may instead choose to hold the ghost cells in a separate array and forgo padding. However, this type of strategy introduces a different type of overhead. With padding, we have consistent strides in our memory accesses during the integration phase of the algorithm. However, if the ghost cells are maintained separately, the strides in our data-accesses are inconsistent, which leads to poor cache coherency. Additionally, we must handle the borders of the computational grid in a unique manner (even after the ghost cells are communicated). This introduces more branches into the program path, which are never a good thing for performance. Ultimately, we decided to stick with the padding strategy, although we did not take any measurements to weigh the benefits and pitfalls of the two strategies.

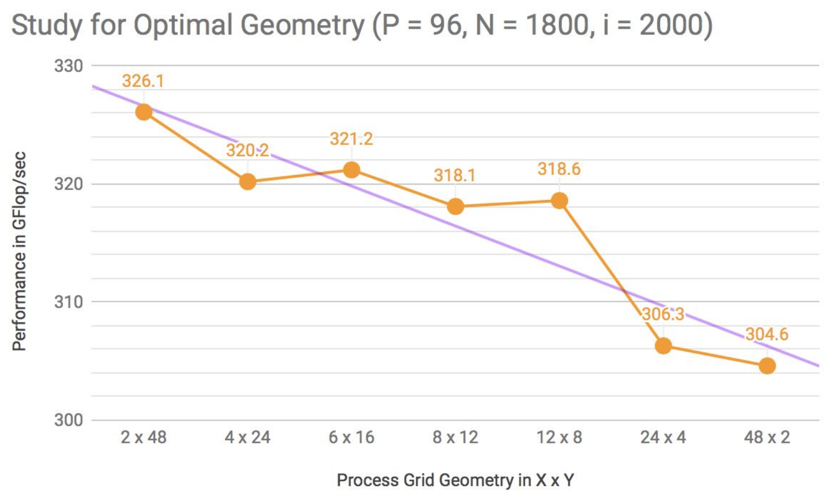
4.2.3 Contention between Asynchronous & Blocking communication

To compute dynamic equations in PDE we use 5pt stencil algorithm which involves halo exchanges between processes. We use MPI-2 for our Halo region communications. Communication is a 2 stage process (East/West Boundary, North/South Boundary). We initially started off using blocking communication routines MPI_Send and MPI_Recv which have inbuilt MPI_Wait. Even though it worked on smaller mesh sizes (M x N), it started to hang indefinitely when run on large values of M and N. Turns out, blocking communications drops messages if mesh is very large without any form of indication that the message was dropped other than a complete indefinite hang. These sorts of behaviour made the debugging extremely tedious, which makes sense as to why memory coherency is used more often than message passing algorithms. However, the idea of discrete message passing is brilliant, because we're left only transferring data that is absolutely necessary. A solution to this is the non-blocking asynchronous communication routines MPI_ISend and MPI_IRecv, which has immediate return. These routines performed flawlessly on all values on M x N.

5. Performance Study

We have parallelized the Aliev-Panfilov algorithm using MPI. Let's conduct a performance study to check the efficiency of our implementation. We first identify the optimal processor grid geometry for which we get the maximum performance on a handful of mesh sizes and then choose that geometry to do following scaling study. In this section, we analyze our results and why they concur with our expectations.

5.4.1 Choosing the Optimal Processor Geometry

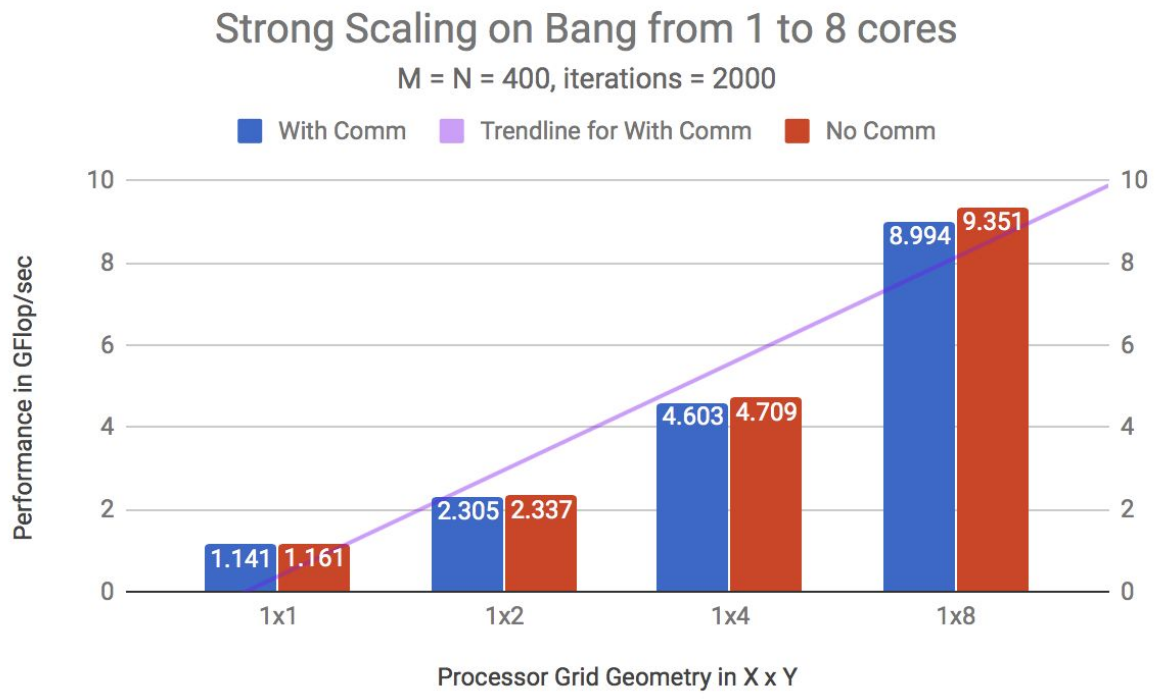


On measuring with multiple mesh sizes our optimal process grid geometry comes out to be rectangular with more processes in Y direction ($P_y >> P_x$). This contradicts with our logical assumption of square geometries having higher performance. But it makes sense for our implementation because, With more processors dividing the rows of the global computation problem, we have lesser values to unpack/pack for the west and east boundaries.

This means lesser computation time in each process. Based on this result, we shall be using $P_x = 2$ while conducting the linear strong scaling study. The above figure shows one such study conducted on 96 cores.

5.4.1 Linear Performance Scaling

Initially we had our APF solver run on a compute node at Bang cluster to conduct scaling studies from 1 to 8 cores. Following is the graph that summarizes the study.



We can see that it scales pretty much linearly with the increase in the number of cores in Bang. The values at 8 cores are approximately 14 times higher than the naive APF solver which doesn't parallelize. However it's interesting to note that with 1 x 1 geometry our solver's performance is lower than basic APF serial solver. (about 1.028 times). This is probably because of the computations and the branchings we do in parallelized solver.

5.4.1.1 Optimizing for Communication cost in Comet

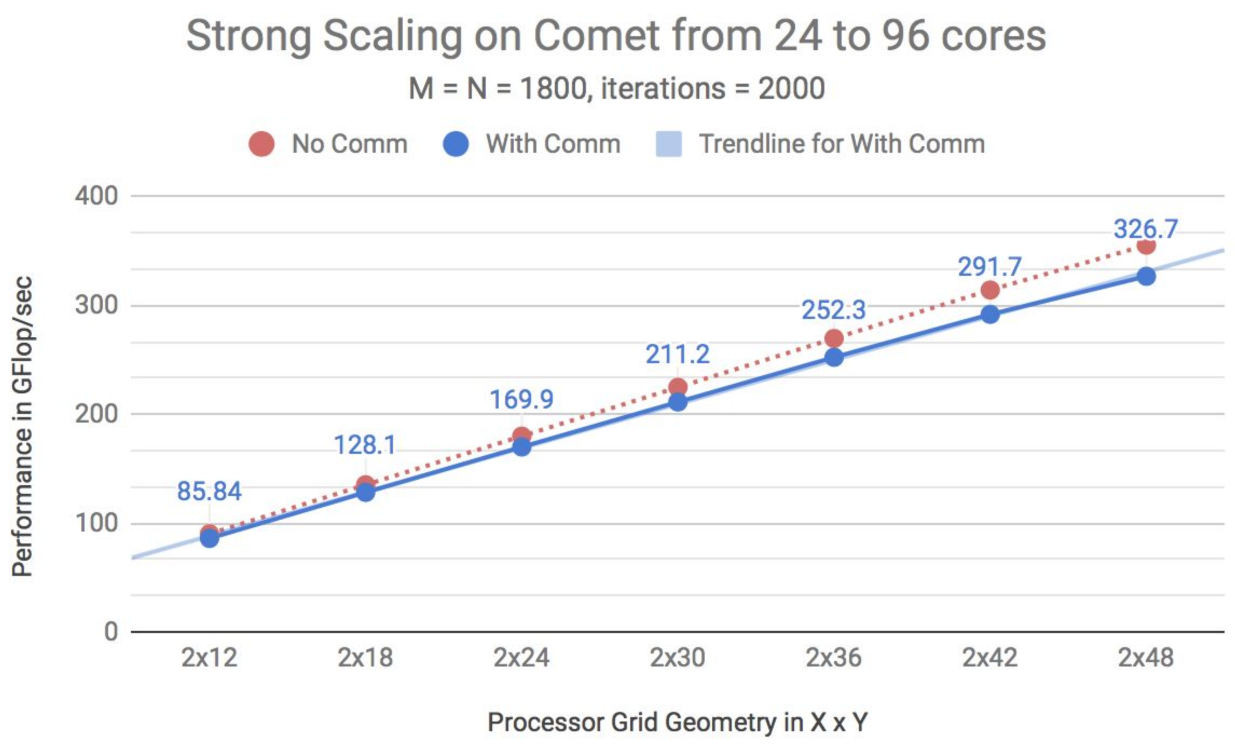
Comet has 24 cores per compute node. When we submit a job to Comet in a .slurm file, we get allocated multiple compute nodes along with a provision of having upto 24 tasks per node. Communication overhead is much more when process from one node is trying to communicate to a process in another node than when two processes within a node are trying to communicate. This is evident from the following observation which shows almost a 2x decrease when the tasks per node drops. By having less number of tasks per node we kind of increase the communications between nodes.

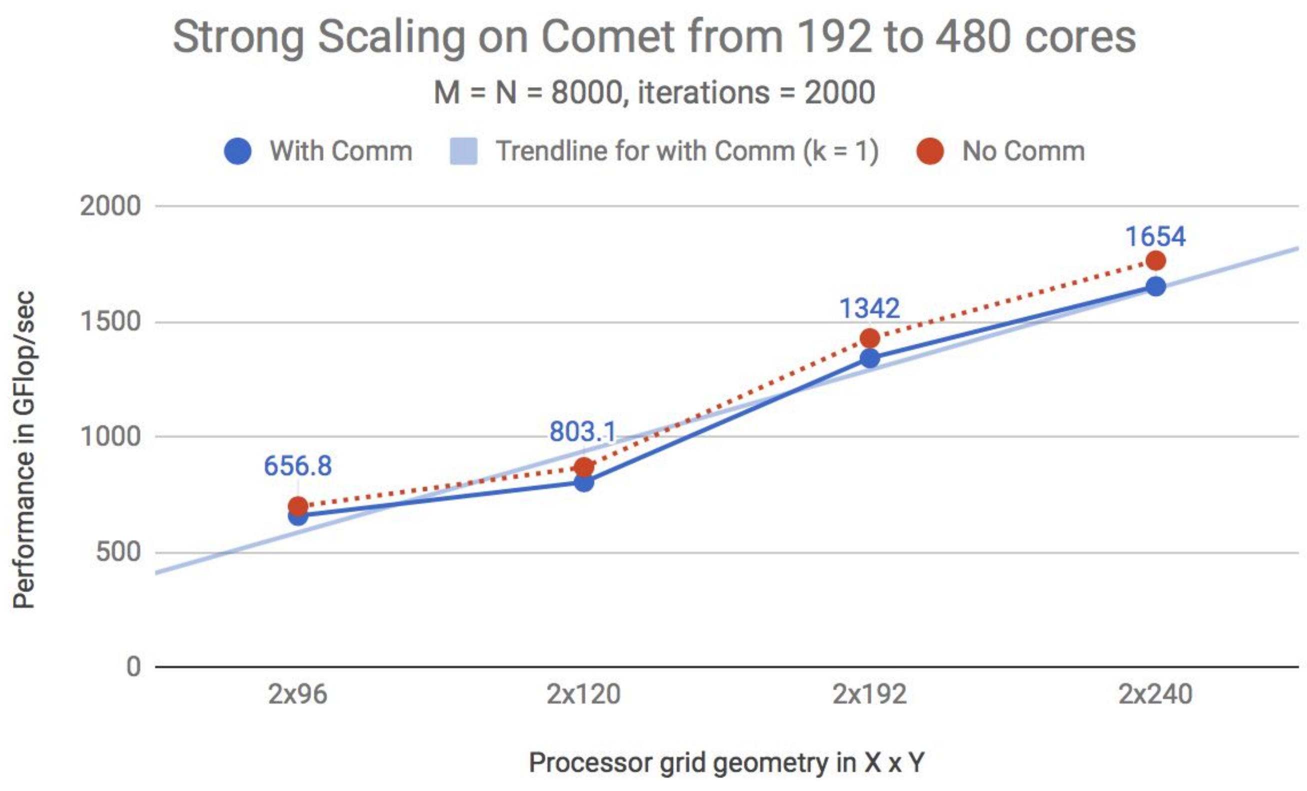
Nodes	Tasks per Node	Parameter	Performance in GFlop/sec

40	24	M = N = 8000
		iterations = 2000
			656.8

60	16		377.7

Let’s see how our code performs on Comet nodes - **Number of Nodes = 20. Tasks per node = 24.**

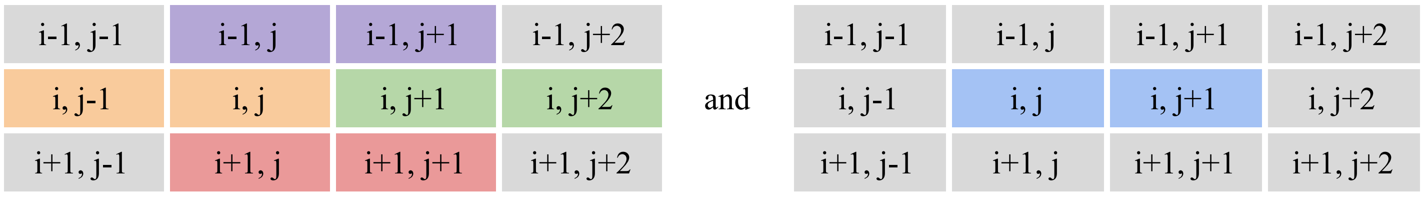




4.3 Vectorization

4.3.1 Using SSE Intrinsics

We used SSE intrinsics to perform a method analogous to loop unrolling. Instead of computing dynamical equations and updates for one cell at a time, we do it for 2 cells at a time. With a 5 pt. stencil based solver, computing $E_{i,j}$ requires $E_{i-1,j}$, $E_{i,j-1}$, $E_{i+1,j}$ and $E_{i,j+1}$. Similarly the next contiguous element, $E_{i,j+1}$ requires the surrounding cells from 4 Manhattan directions.



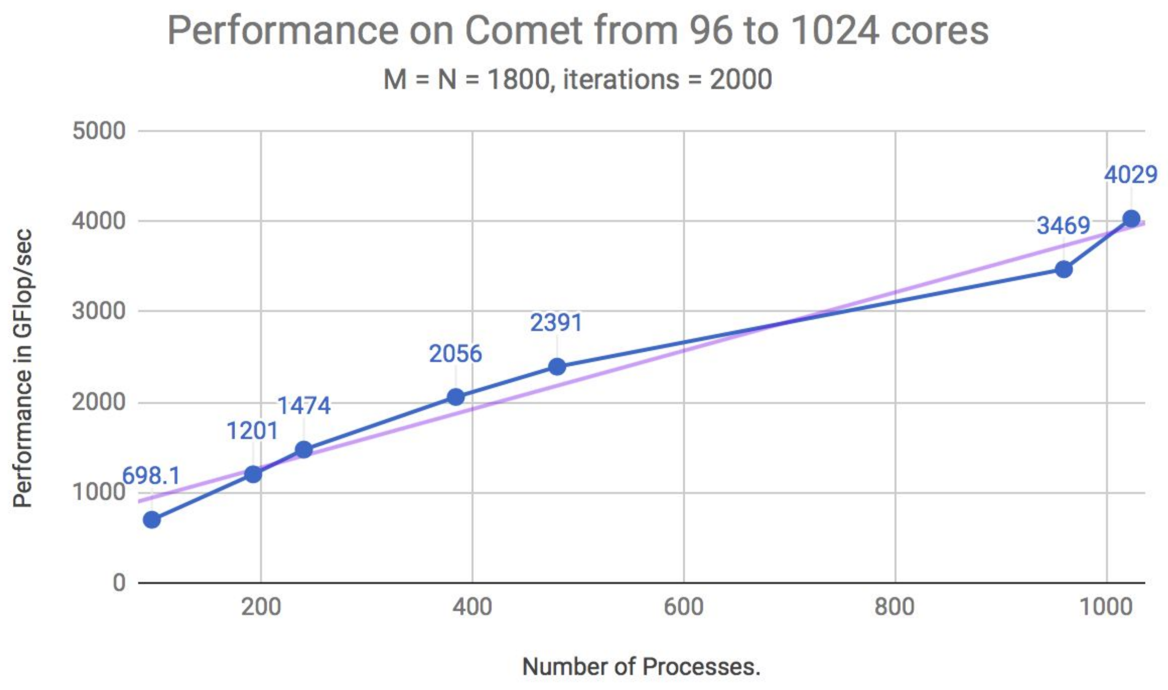
Implementation wise, this means we can load five 128-bit words of E_{prev} or the previous time-step E to do the PDE. Ordinary Differential Equations solver equations can also be vectorized in a similar trivial way. We do include and load the East boundary ghost cells in the update using SSE instructions to avoid branching. This means that at the end of each iteration, the east boundary ghost cells are updated to some random values and also, this won't cause errors in L2 and Linf norms because before these values are being used up, we fill it up with appropriate values from neighbouring processes.

4.3.2 Aligned vs Unaligned loads

In our implementation of parallelized APF solver we're using unaligned loads. This gave a significant boost over the previous solver that was discussed in section 3. We can make our loads aligned by padding the subproblems to even size. If we align E, $E_{prev.}$ and R to 16 bytes then we can make ⅗ loads 16 byte aligned. This even padding to sub problems required to change the signature of `solve`, which means modifying the apf.cpp which is against the instructions. Besides, padding strategy for aligned loads ended up being slower than the SSE baseline, probably because of larger stencil stride, so we sticked with unaligned loads in our implementation.

4.3.3 Using AVX intrinsics

As discussed above for SSE, we would have a huge uplift in performance if we vectorize our code. We moved from SSE to AVX such that we would be computing 256 bits at once with more registers. This implementation almost gave us a more than 90% increase in performance as compared to a parallelized solver with no vectorization. Following graph shows the strongly scaled performance of our vectorized code from 96 to 1024 cores in Comet (nodes = 20, tasks per node = 24)


Performance on Comet from 96 to 1024 cores
M = N = 1800, iterations = 2000
Performance in GFlop/sec
Number of Processes.
0
1000
2000
3000
4000
5000
200
400
600
800
1000
698.1
1201
1474
2056
2391
3469
4029