CSE 260 Programming Assignment 3

Honam Bang, Myron Liu

Tuesday 24th November, 2015

1 Notation

In this document we will use the following notation:

- $\{M, N\}$ are the number of rows and columns (respectively) in the *global* computational grid.
- We partition the global problem into y rows and x columns of subproblems, each handled by its own process. **We will refer to this as the process-grid geometry.**
  - We denote the number of processes as $p = xy$.
- $\{m_{ij}, n_{ij}\}$ are the dimensions of the computational grid for the process in process-row i and process-column j. **We will refer to this as the process-geometry.**
  - We will sometimes simply use $\{m, n\}$ when it is unambiguous to which process we are referring
- $\{E_G, R_G\}$ are the scalar fields for the global problem.
- $\{E_{ij}, R_{ij}\}$ are the scalar fields for subproblem ij
  - We also have $Eprev_{ij}$, which serves as a write buffer to be swapped with E. Since E is governed by a partial differential equation, our numerical integration must pass a stencil over each element; this precludes us from updating E in place.

2 Implementation

2.1 Division of Labor among Processes

We divide the global problem into a $y \times x$ grid of subproblems, each of which is handled by a separate process. To allocate the work more evenly, we stipulate that the number of rows for all subproblems differs by no more than one (and similarly for the number of columns). We adopt the row-major convention, so that process 0 (blue in our figure below) is responsible for the upper left corner of the global problem, and process $x - 1$ (in pink) is responsible for the upper right corner. For instance, with a process-geometry of $\{y, x\} = \{2, 3\}$ and a global problem size $M = N = 5$ we have the following division of labor:

A_{00}	A_{01}	A_{02}	A_{03}	A_{04}
A_{10}	A_{11}	A_{12}	A_{13}	A_{14}
A_{20}	A_{21}	A_{22}	A_{23}	A_{24}
A_{30}	A_{31}	A_{32}	A_{33}	A_{34}
A_{40}	A_{41}	A_{42}	A_{43}	A_{44}

where each color corresponds to a unique process, and A_{ij} can either refer to $E_{prev,ij}$ or to R_{ij}. Notice how, as promised, the computation is distributed more or less evenly.

2.2 Ghost Cell Communication

When we compute the term $(\nabla^2 Eprev)_{ij}$ in the dynamical equations, we require the values of $Eprev$ in each of the four cardinal directions: $Eprev_{i-1,j}, Eprev_{i+1,j}, Eprev_{i,j-1}, Eprev_{i,j+1}$. This poses a problem. To update the values of E on the border of the subproblem, we require values of $Eprev$ that lie outside of the process's computational grid. Therefore, we must pull these values from the computational block of adjacent processes. The diagram below illustrates this idea. The elements in black constitute the computational grid for process i, j. To update the system, we require the values in red, which come from neighboring processes. We call the cells in red *ghost cells*.

Note: In the diagram below, we assume that the neighboring processes also have the same geometry, but in general this need not be the case.

...	$Eprev_{20}^{i-1,j}$	$Eprev_{21}^{i-1,j}$	$Eprev_{22}^{i-1,j}$	...
$Eprev_{02}^{i,j-1}$	$Eprev_{00}^{ij}$	$Eprev_{01}^{ij}$	$Eprev_{02}^{ij}$	$Eprev_{00}^{i,j+1}$
$Eprev_{12}^{i,j-1}$	$Eprev_{10}^{ij}$	$Eprev_{11}^{ij}$	$Eprev_{12}^{ij}$	$Eprev_{10}^{i,j+1}$
$Eprev_{22}^{i,j-1}$	$Eprev_{20}^{ij}$	$Eprev_{21}^{ij}$	$Eprev_{22}^{ij}$	$Eprev_{20}^{i,j+1}$
...	$Eprev_{00}^{i+1,j}$	$Eprev_{01}^{i+1,j}$	$Eprev_{02}^{i+1,j}$	...

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 noncontiguous 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 attractive, 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 pros and cons of the two strategies.

To numerically enforce the condition that $\hat{n} \cdot \nabla E = 0$ on the global problem boundaries, we set the *external* ghost cells to those two rows (or columns) into the computational grid's interior. As needed, for our *rectangular* problem grid, this satisfies:

$$(\hat{n} \cdot \nabla E)\Big|_{\text{north boundary}} = \frac{\partial E}{\partial y}[0, j] \sim \frac{E[1, j] - E[-1, j]}{2} = 0$$

since $E_G[1, j] = E_G[-1.j]$ by construction. The same is true for the east, west, and south global boundaries.

2.3 SSE Vectorization

We leveraged SSE using a very naive method analogous to loop unrolling. Instead of numerically integrating one cell at a time, we integrate two cells. We do this by loading the following five 128-bit words of E (color-coded) and then using them to update the two cells at the center (E_{ij} and $E_{i,j+1}$) in parallel.

$$\begin{array}{ccccccc}
 & \vdots & \vdots & \vdots & \vdots & & \\
\cdots & E_{i-1,j-1} & \mathbf{E_{i-1,j}} & \mathbf{E_{i-1,j+1}} & E_{i-1,j+2} & \cdots & \\
\cdots & \mathbf{E_{i-1,j-1}} & \mathbf{E_{i,j}} & \mathbf{E_{i,j+1}} & \mathbf{E_{i,j+2}} & \cdots & \text{and} \\
\cdots & E_{i+1,j-1} & \mathbf{E_{i+1,j}} & \mathbf{E_{i+1,j+1}} & E_{i+1,j+2} & \cdots & \\
 & \vdots & \vdots & \vdots & \vdots & &
\end{array}
\qquad
\begin{array}{ccccccc}
 & \vdots & \vdots & \vdots & \vdots & & \\
\cdots & E_{i-1,j-1} & E_{i-1,j} & E_{i-1,j+1} & E_{i-1,j+2} & \cdots & \\
\cdots & E_{i-1,j-1} & \mathbf{E_{i,j}} & \mathbf{E_{i,j+1}} & E_{i,j+2} & \cdots & \\
\cdots & E_{i+1,j-1} & E_{i+1,j} & E_{i+1,j+1} & E_{i+1,j+2} & \cdots & \\
 & \vdots & \vdots & \vdots & \vdots & &
\end{array}$$

The ordinary differential equations are also vectorized in the trivial manner. Even though we used unaligned loads, SSE still gave us great boosts in performance: this can be seen in figure 2. To avoid branch statements, we obliviously integrate the east ghost cells as well. Although the stencil's *shape* is not preserved when we lay it over the east ghost cells (and more importantly, we aren't even interested in updating the ghost cells) this does not violate correctness. Only one computational cell is interested in each east-ghost-cell, and this computational cell is already updated by the time we wantonly update the associated east-ghost-cell. By the time we finish one iteration of the problem, the ghost cells are obsolete, and newly communicated values in the next iteration will overwrite the gibberish.

One way to make most of our loads aligned is to pad the subproblems to even size. If we align E_{00}^{padded}, $Eprev_{00}^{\text{padded}}$, R_{00}^{padded} on 16 bytes, then we can make three out of our five 16 byte loads (*i.e* the **red** , **pink** , and **yellow** cells) aligned.

- Note: We actually implemented this padding strategy, but for programmatic reasons that aren't particularly interesting, it requires us to change the signature of

  solve(double**, double**, double*, double, double, Plotter*, double&, double&)
  ... to ...
  solve(double**, double**, double**, double, double, Plotter*, double&, double&))

  This requires that we modify *apf.cpp*, which is contrary to the instructions. For this reason our **master** branch contains the unpadded vectorization. Our padded vectorization can be found in the branch **offsetSSE**.

We also tried to align our loads by padding to even width, and then tacking on an additional column of zeros on the left of each process. But this was actually slower than our *base* SSE implementation, probably because the strides in the stencil become larger. For this reason we abandoned this approach in favor of the unpadded strategy.

3 Results

We demonstrate linear performance scaling (in GFLOPS) for our MPI implementation of the Aliev-Panfilov equations. We also measured the communication cost as a function of the process-grid-geometry $\{y, x\}$. To find the optimal process-grid geometry (and indirectly, the process geometry), we compared different choices for $\{y, x\}$ given a handful of problem sizes. In this section, as well as the next, we analyze our results and why they concur with our expectations.

3.1 Linear Performance Scaling

As figure 1 shows, the overall performance for our MPI implementation increases as we increase the number of processors (unless the problem size becomes too small). We also show the increase in performance from vectorizing our code with SSE intrinsics.

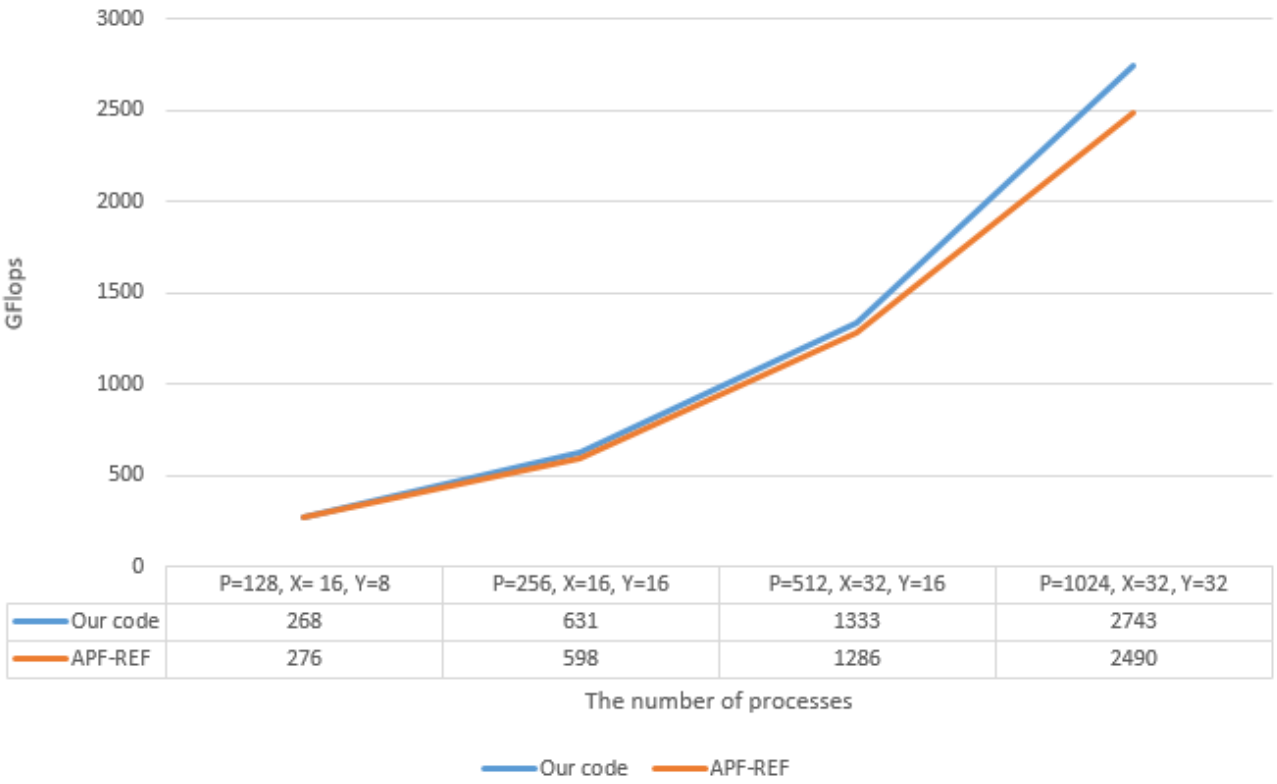


Figure 1: The plot above shows the performance of our **unvectorized** implementation versus the performance of the reference binary. The problem size is $M = N = 8000$ integrated for 2000 iterations. The performance of our *unvectorized* MPI implementation scales linearly as we increase the number of processes (note that the figure is actually a bar graph, which is the source of the apparent curvature in the line).

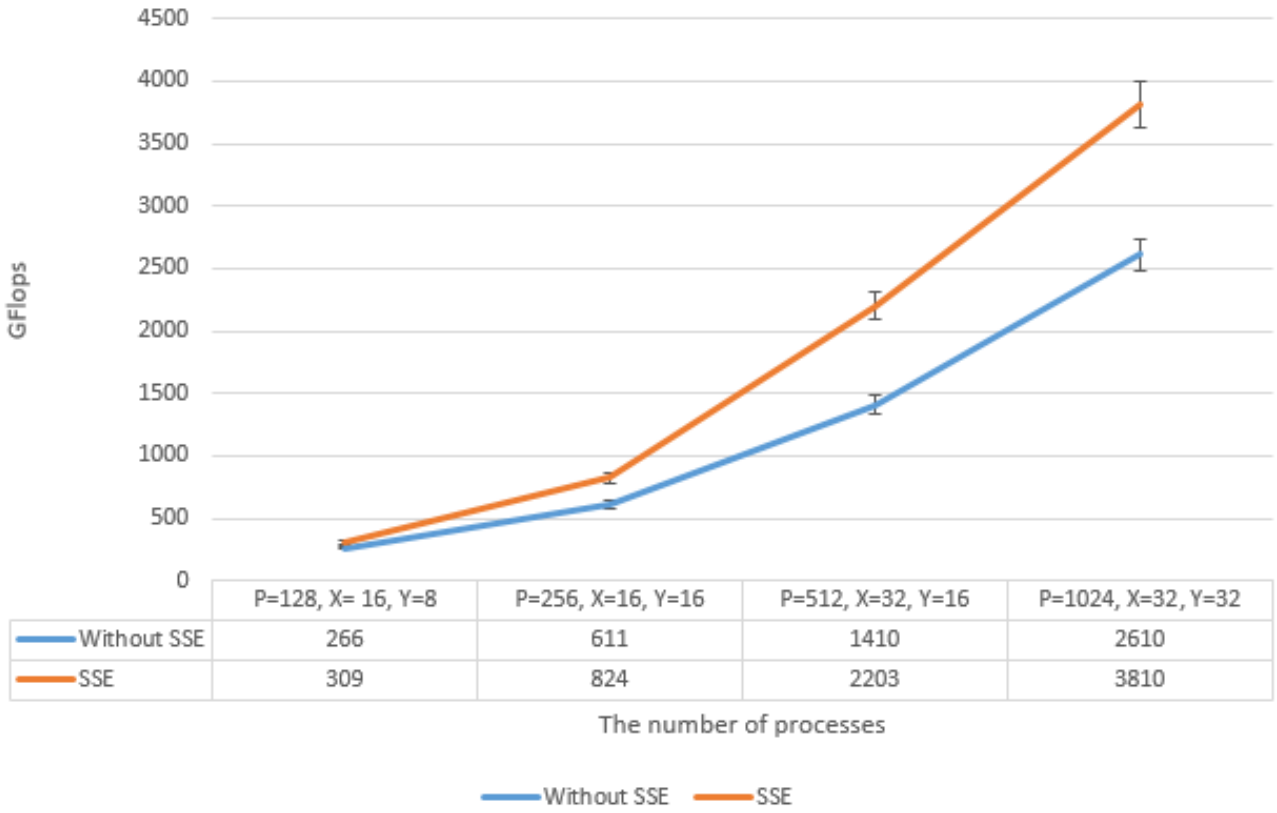


Figure 2: The plot above shows the speedup gained from vectorizing the computation phase of our program.

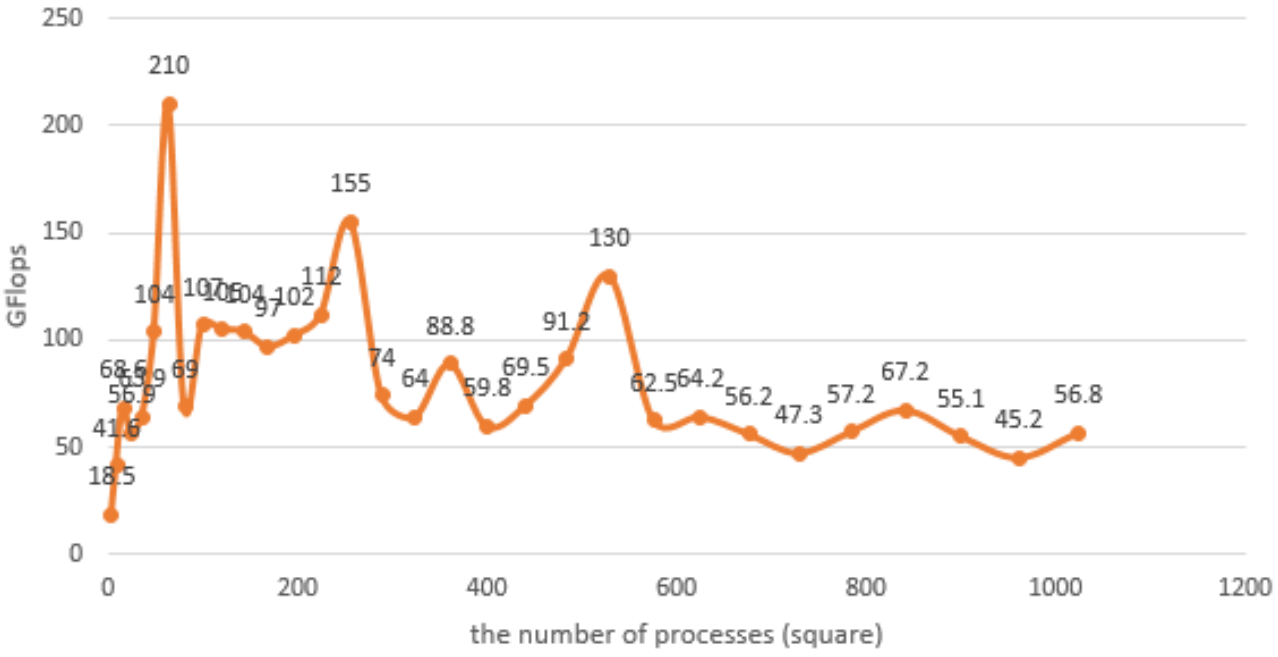


Figure 3: The plot above shows the performance of our vectorized code for a problem size of $M = N = 512$ and $i = 2000$ iterations. All trials were performed with a square process-grid (*i.e.* $x = y$). We observe that the overall trend is for performance to increase rapidly and then tail off gradually. This falloff is probably due to the fact that as the number of processes grows, the computational intensity starts falling. Eventually the increased parallelism in arithmetic operations is overtaken by the overhead of data-motion.

The reason why the performance varies linearly with the number of processes is primarily due to the increased number of arithmetic operations we can perform in parallel, along with the shorter stalls for each process to communicate ghost cell data. We will discuss these gains, along with the drawbacks for excessively increasing the number of processes, in the Discussion section.

3.2 Communication cost

Communication cost is mainly affect to the performance since it also increases by the number of processes. Therefore, for giving N, choosing proper size of p is important to achieve maximum performance.

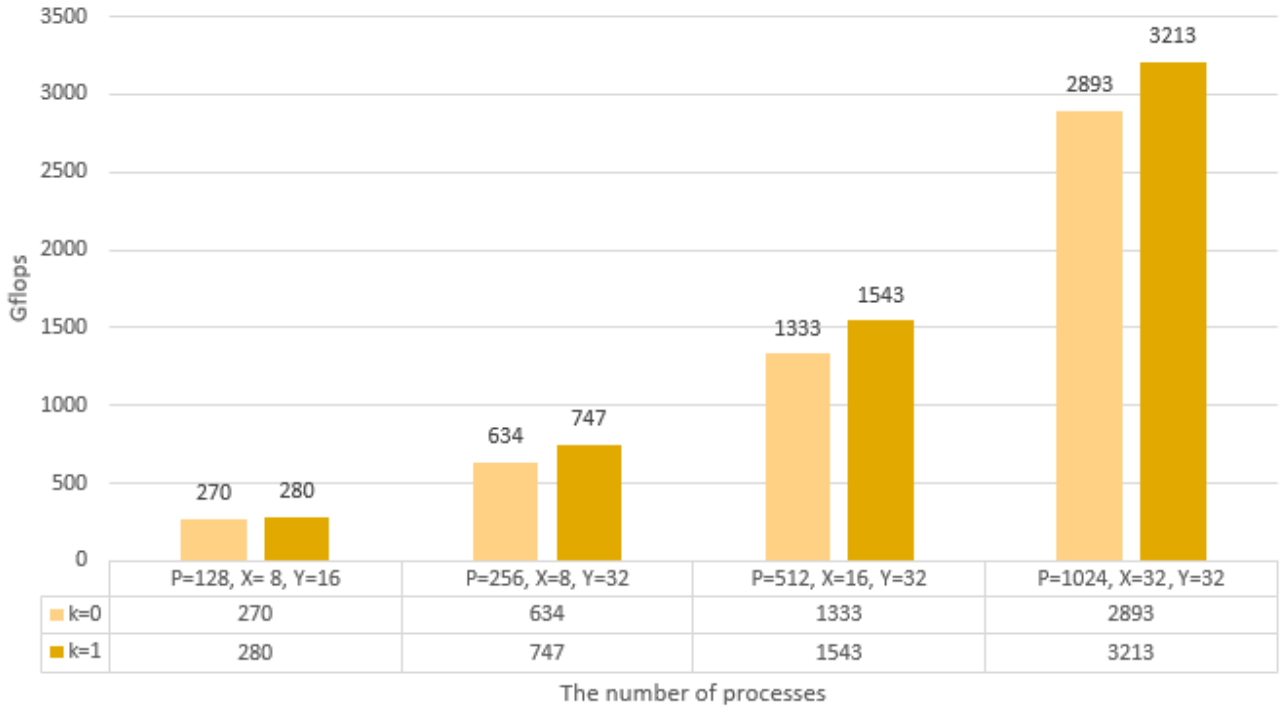


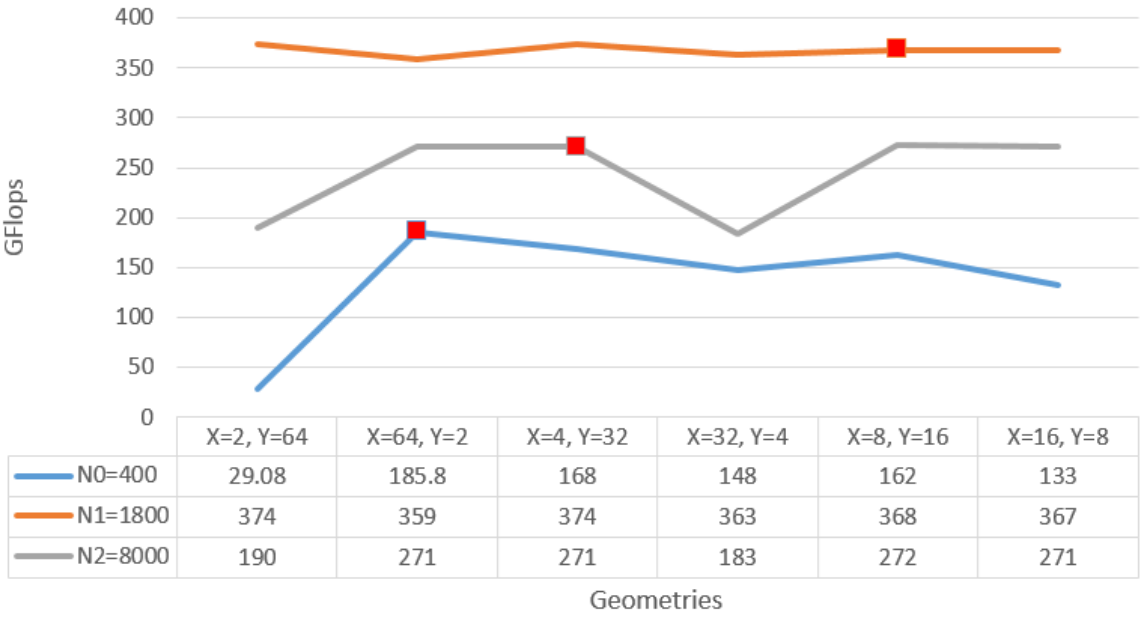
Figure 4: Plot showing communication cost for an increasing number of processes. As the number of processes grows larger, the communication cost also increase.

For the figure 4, the communication cost increases as p grows larger. This is because the amount of data transfer increases as the number of processes increases. Similar to how our lungs have very large surface area due to complexly-folded internal structures, the combined perimeter of all the processes grows as our subproblems become smaller (for fixed global problem size). We will see in the Discussion section how this affects the total amount of data motion, along with the implications for per-process computational intensity q.

3.3 Optimal Process and Process-Grid Geometry

Just as important as the process size (inversely proportional to the number of processes), is the process geometry (inversely proportional to process-grid geometry). We measured for an handful of problem sizes, how performances varies with process-grid geometry. We found that in general, square process geometries are optimal. This is mainly because data-transfer per process grows in

direct proportion to the perimeter of the computational blocks, while the number of arithmetic operations grows as the area: more on this later.



	X=2, Y=64	X=64, Y=2	X=4, Y=32	X=32, Y=4	X=8, Y=16	X=16, Y=8
N0=400	29.08	185.8	168	148	162	133
N1=1800	374	359	374	363	368	367
N2=8000	190	271	271	183	272	271

Figure 5: The plot above shows how performance varies as we vary the process-grid geometry and the problem size with a fixed number of processes $p = 128$. The optimal geometry from our trials is shown with the marker: ■. Note how for the smallest problem $N_0 = 400$, the tall-narrow process geometry (corresponding to process-grid geometry $x = 64, y = 2$) is actually more optimal than the near-square geometry. This, we conjecture, is due to the fact that the *stenciled cells* are spatially local, and so fit more contiguously into memory.

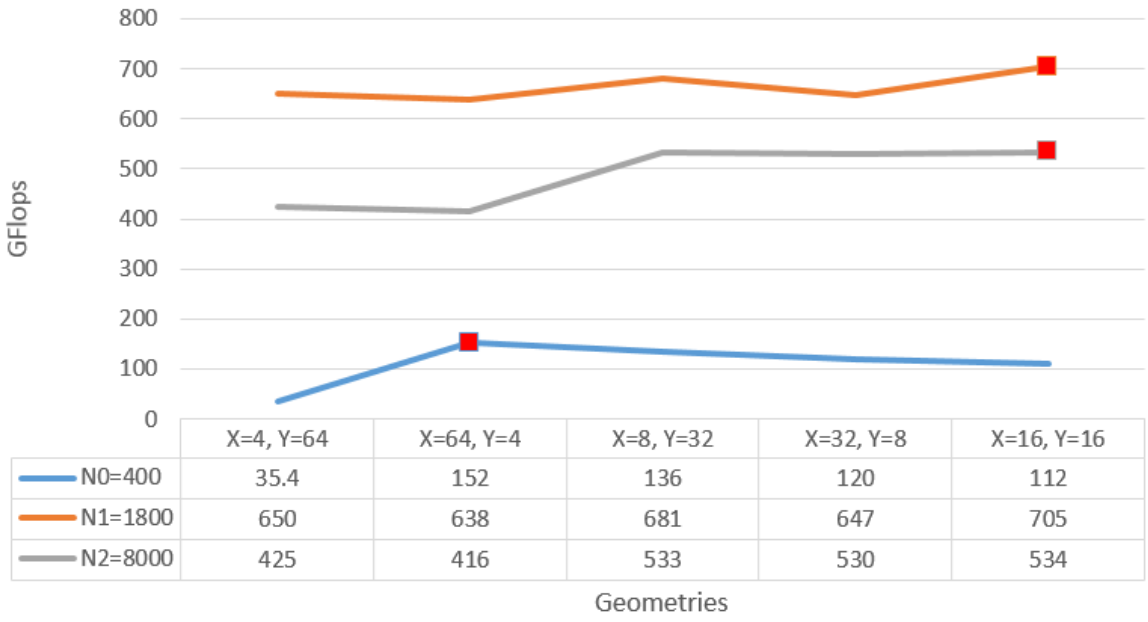
In the subsequent section we discuss the aforementioned results and why we might expect such phenomena.

4 Discussion

4.1 Selecting process geometries

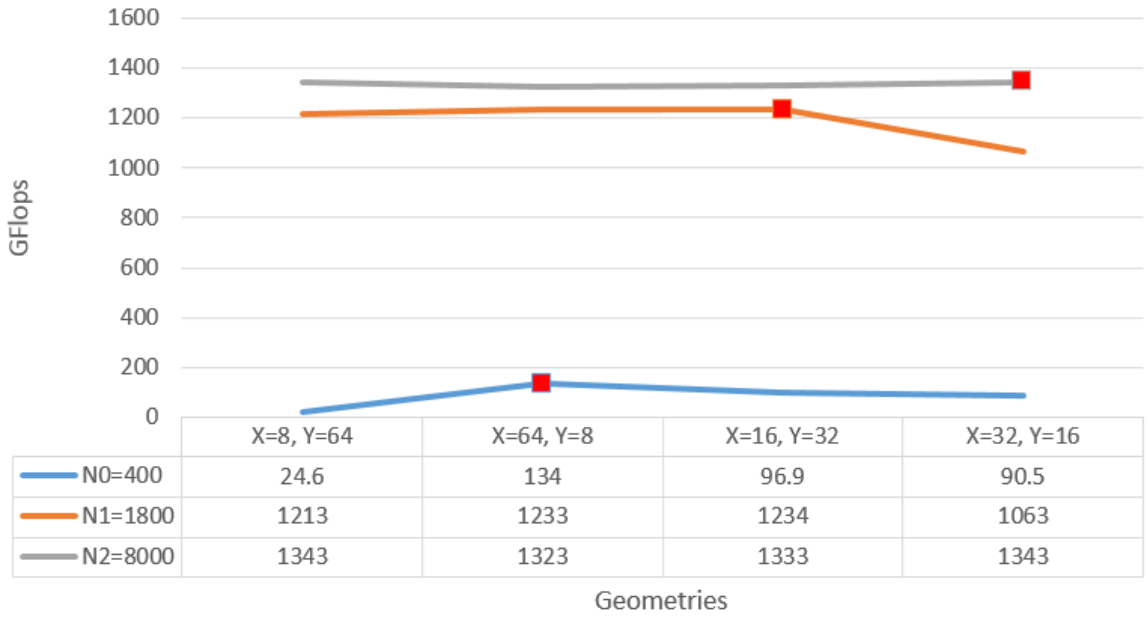
Recall that no matter the size of the subproblem, our ghost cells always form a border one cell wide. Therefore, data-transfer per process grows in direct proportion to the *perimeter* of the computational blocks. On the other hand, computation per process grows as the *area* of the computational block. Since data motion is much slower than floating point operations, we want to maximize the *computational intensity* q: the ratio of arithmetic operations to the number of memory accesses. More specifically, for *fully interior* subproblems, we have:

$$
\begin{aligned}
n_{\text{DATA}} &= 2(m+n) \\
n_{\text{FLOP}} &\propto mn \\
\implies q \equiv \frac{n_{\text{FLOP}}}{n_{\text{DATA}}} &\propto \frac{mn}{m+n}
\end{aligned}
$$



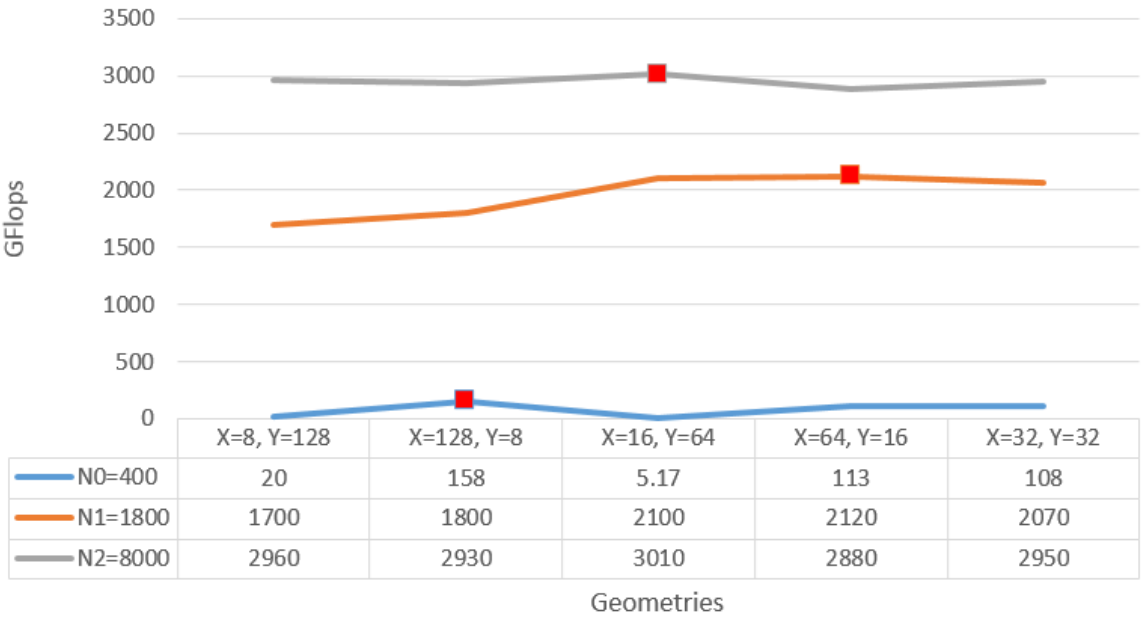
	X=4, Y=64	X=64, Y=4	X=8, Y=32	X=32, Y=8	X=16, Y=16
N0=400	35.4	152	136	120	112
N1=1800	650	638	681	647	705
N2=8000	425	416	533	530	534

Figure 6: Optimal process-grid geometry plot for $p = 256$. The graphs of N1 and N2 show that the maximum performance is achieved by choosing square geometries, which is in line with our intuition for reasonably large problem sizes.



	X=8, Y=64	X=64, Y=8	X=16, Y=32	X=32, Y=16
N0=400	24.6	134	96.9	90.5
N1=1800	1213	1233	1234	1063
N2=8000	1343	1323	1333	1343

Figure 7: Optimal process-grid geometry plot for $p = 512$. The graphs of N1 and N2 show that the maximum performance is achieved by choosing square geometries, which is in line with our intuition for reasonably large problem sizes.



	X=8, Y=128	X=128, Y=8	X=16, Y=64	X=64, Y=16	X=32, Y=32
N0=400	20	158	5.17	113	108
N1=1800	1700	1800	2100	2120	2070
N2=8000	2960	2930	3010	2880	2950

Figure 8: Optimal process-grid geometry plot for $p = 1024$. In this case, square blocks are not the most optimal. Although squares minimize the perimeter-to-area ratio, we must remember that there are other factors that affect performance. Skinnier processes have smaller strides in the stencil, while wider processes could potentially have shorter communication latencies despite their increased perimeter. Furthermore, it may be possible that particular strides in the stencil may access cache in a way that is more conducive than a shorter strides.

Or, if we let $k = mn$ be the size of the computational array...

$$q \propto \frac{k}{k/n + n}$$

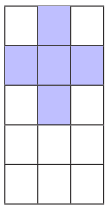
Maximizing this with respect to n gives $n_{\text{optimal}} = \sqrt{k}$. This result just tells us that we should shoot for square (or nearly square) computational blocks, which is exactly what we expect.

- This sort of *surface-area-to-volume ratio effect* is prevalent in nature, which is an apt analogy for our particular problem. Arctic animals tend to have rounder bodies and shorter limbs. This is because the rate at which body heat is lost is proportional to the surface area of the animal, while the amount of body heat that can be stored, is proportional to the volume of the animal. In contrast, equal-volume animals in tropic regions tend to have elongated bodies with long limbs (as a strategy for dissipating heat). Think of the number of floating point operations as the volume of the animal and the memory accesses as the surface area of the animal. Clearly, we want our processes to be adorable, rotund arctic seals.

We note that the situation is a bit more hairy for *exterior* subproblems, which don't need to communicate the global boundary conditions. In this case, the computational intensity diverges from the aforementioned expression, depending on the particular position of the process. Nonetheless, as long as the process size is small relative to the problem size, most processes will be fully interior, so our analysis is still helpful for the typical case. More importantly, the exterior subproblems will almost certainly complete each iteration faster, so performance hinges on the slowest process, which is likely an interior process: it must receive messages from all four neighbors to proceed with the computation. Therefore, interior processes are of greater interest to us.

What we have said so far might convince one to always opt for nearly square processes. But skinny processes do have their advantages. We consider two extreme cases, one where the process is tall and narrow, and one where the process is wide and flat.

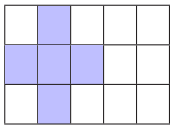
- The advantage of a tall and narrow process is that once the ghost cells are filled, applying the dynamical equations is fast (unless one implements SSE vectorization in our fashion). This is because all entries under the stencil are spatially local. In the following diagram, we show a 3×1 subproblem along with its ghost cells.

  In particular, note how no two entries beneath the stencil vary by more than $6 \cdot 8$ bytes$= 48$ bytes. This is the power of tall-narrow processes. The *stenciled* entries fit neatly into a minimal number of cache lines. Indeed, for smaller problem sizes, we saw that sometimes tall-narrow blocks do in fact perform better than our hallowed square blocks (see figure 5).

  The downside of such a configuration (relative to wide and flat processes) is that message (un)packing for westward and eastward ghost cell communication is more expensive.

- The advantage of a wide and flat process is that message (un)packing, in contrast to tall and narrow processes, is fast. The disadvantage is that the the stencil has large strides.



Here the distance between the *north* stencil entry and the *south* stencil entry is 10·8 bytes= 80 bytes: still small enough for cache coherency, but one can imagine that the problem is quickly exacerbated as the process-width increases.

This last point is interesting. If our processes are small, then we might be able to maintain both cache coherency and bypass the message (un)packing problem by using wide-flat processes. Although, relative to square blocks, there is more data-motion between processes, the fact that the messages are contiguous and short enough to fit into a couple of cache lines suggests that wide-flat processes might work well in certain scenarios, particularly for small problem sizes. However, we were not able to verify this hypothesis. Indeed, in figures 5, 6, and 7, wide processes actually performed the worse. Either we did not try sufficiently small problem sizes, or our conjecture is incorrect. Regardless, this is something that we would like to test in the future.

In general, though, for any reasonably large problem size, we found square process geometries to work better than lopsided geometries, which is the content of figures 5, 6, and 7.

4.2 Impact of the Number of Processes on Performance

Increasing the number of processes is favorable to an extent. The most obvious advantage is that more processes allows us to perform more arithmetic operations in parallel: this is responsible for the approximately linear performance-scaling in figure 1. The other big advantage is that, fixing the geometric *proportions*, the data-strides in the stencil become smaller. Finally, per-process sends/receives have decreased latency, since message (un)packing becomes cheaper. Yet, despite these gains, there are also crucial drawbacks to going overboard with the number of processors.

The first of these is the communication cost. Suppose we have a square problem and square processes. Furthermore, suppose that n evenly divides N. The total number of processes is $p = (N/n)^2$. Therefore, the total amount of data communicated between processes is approximately:

$$p \cdot 4n \sim p\frac{n}{N}N = p \cdot p^{-1/2}N \propto \sqrt{p}$$

Hence, data motion between processes grows as the square root of the number of processes. The actual functional form of communication cost is a bit more difficult to obtain. In particular, we did not decouple the overhead of message (un)packing from overhead of MPI sends/receives, which act in opposing directions (increasing p *increases* overall data-transfer, but *decreases* (un)packing cost per process). As shown in figure 4, we observe greater communication overheads for more processes.

Another more important reason why it is unwise to recklessly increase the number of processes is that it decreases the computational intensity per process. We mentioned earlier that we can increase the perimeter-to-area ratio of our processes (a bad thing) by choosing lopsided geometries. We can also achieve the same effect by making the subproblems smaller, which is a direct consequence of increasing the number of processes. Eventually, the amount of data-motion outstrips

the amount of arithmetic operations, and performance begins to fall off. This is demonstrated in figure 3.

5 Conclusion

There are numerous considerations when choosing an optimal configuration for an MPI implementation of the Aliev-Panfilov dynamical equation. For reasonably large problem sizes, we found nearly square process geometries to work best, presumably because they maximize the computational intensity. We also observed that performances scales approximately linearly with the number of processors until the subproblems become so small that data motion begins to overshadow computation.