PA3 - Computing using MPI

https://github.com/cse260-fa22/pa3-camcginley-rskarande

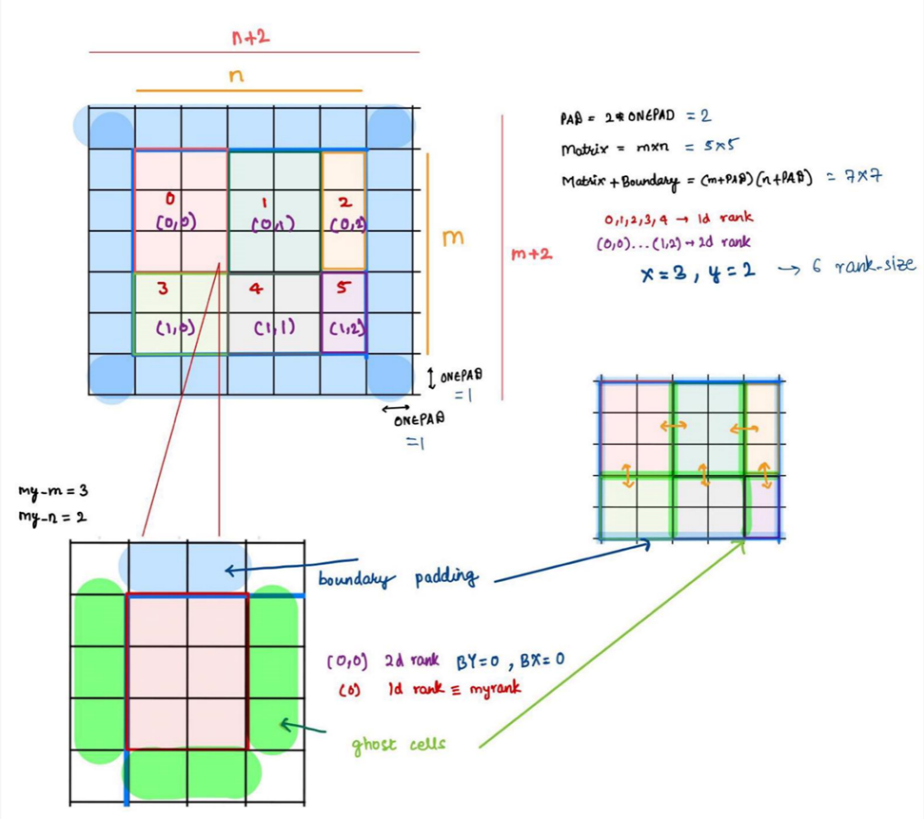
Conventions used in the report

- (m,n) are the number of rows and columns in the global grid
- The grid is partitioned into blocks for computation
    - A block maps to one rank.
    - There are (x,y) blocks in X and Y direction respectively
    - Total processes, thus would be x*y
    - (BX, BY) refers to the 2d rank of the process. Row major convention is used for ranking the processes in 2d.
- async version - no blocking communication and splitted computation
- sync version - partially blocking communication and non-splitted computation.
- Geometry (x,y)

Section (1) - Development Flow

Q1.a) Describe how your program works (pseudo code is fine, do not include all of your code in the write-up). [ghost cell exchanges, boundary cases, even distribution, MPI calls etc.]

We present an example of matrix M = 5x5 to give an overview and then present the pseudo code.



Matrix(M) = 5x5 matrix when padded becomes a 7x7 matrix. M is subdivided in 6 blocks. One process operates on one block.

```
void init(){
    // send intial conditions from rank0 to all other ranks
}

void communicate_async()
{
    if (BY == IS_TOP_BLOCK)  // top padding
    else // top-down exchange send top row, receive bottom row | using Irev, Isend

    if (BY == IS_BOTTOM_BLOCK) // bottom padding
    else // bottom-top exchange | using Irev, Isend

    if (BX == IS_LEFT_BLOCK) // left padding
    else // left-right exchange | first pack and then send |  using Irev, Isend

    if (BX == IS_RIGHT_BLOCK) // right padding
    else // right-left exchange | first pack and then send | using Irev, Isend
}

void solve()
{
    for (int niter = 0; niter < cb.niters; niter++)
    {
        if (!cb.noComm)
            communicate_async(E_prev); // issue async request for exchanging data amongst neighbours

        // Do computation only for all the rows and columns that don't depend on ghost cell communication to be completed
        for (j = innerBlockRowStartIndex + (my_n + 2); j <= innerBlockRowEndIndex - (my_n + 2); j += (my_n + 2))
            for (i = 1; i < my_n - 1; i++)
                // compute E_tmp[i], R_tmp[i]

        if (!cb.noComm)
        {
            MPI_Waitall(counter, recvReqs, statuses); // wait for the async requests to be completed
            if (BX != IS_LEFT_BLOCK) // unpack in_left and store in E_prev
            if (BX != IS_RIGHT_BLOCK) // unpack in_right and store in E_prev
        }

        // Do computation for top and bottom row
        for (j = innerBlockRowStartIndex; j <= innerBlockRowEndIndex; j += (my_n + 2) * (my_m - 1))
            for (i = 0; i < my_n; i++)
                // compute E_tmp[i], R_tmp[i]

        // Do computation for left and right column
        for (j = innerBlockRowStartIndex + (my_n + 2); j <= innerBlockRowEndIndex - (my_n + 2); j += (my_n + 2))
            for (i = 0; i < my_n; i += my_n - 1)
                // compute E_tmp[i], R_tmp[i]
        }

    // collect the stats using MPI_Reduce
}
```

Initialization

1. Rank 0 processes propagatest the initial values of **E_prev** and **R**
2. All other processes wait to receive the initial values and then proceed with the computation

Division of work

1. The original matrix is computed using p = (y*x) processes. Thus each process operates on a block of the original matrix.

2. We ensure that the blocks are evenly divided. The illustration shows show this division takes place for a 5x5 matrix with 6 processes (x=3, y = 2)

Communication among neighboring processing

1. We issue an async request using MPI primitives MPI_Isend and MPI_recv to the neighboring blocks.
2. While this communication is happening we proceed with the computation for inner rows and inner columns of the matrix. This can be done as these elements have all the elements necessary to perform stencil operation.
3. Then we await for the communication to be completed. Once completed, the values are unpacked and stored appropriately in E_prev.
4. We then proceed with the computation for boundary rows and columns
5. After all the computation is completed for all the blocks / processes, we do a reduce operation to compute the L2 and Linf norms.
6. To ensure correctness, we compare the L2 and Linf norms with what when we have 1 process only.

Note: We have implemented sync mode communication as well. For sync mode, we wait for all the communications to happen and then proceed with the computation. This way we do not have to split the computation but have to await for all the exchanges to take place.

Improving efficiency using vectorization

1. In order to improve the efficiency, we used a couple of approaches
    1. Enforced native auto vectorization.
    2. Vectorization by hand.
2. Using vectorization, we club 4 stencil operations as 1 using AVX2 intrinsics.
3. This results in significant speedup. The results are discussed in the bonus section

Q1.b) What was your development process? What ideas did you try during development?

1. We mostly worked on sorken in the initial phase.However, we figured out that MPI could be installed locally. Our machine had 6 cores. So we could initiate 6 processes in parallel. This was very helpful for debugging purposes and for rapid development.
2. Once we had functional code, then we tried to run it on expanse.
3. Our development phase included the following phases:
    a. We extensively use macros as it makes the code much more readable and print statements for debugging. We ensured that the L2 and Linf norms match when we scale.
    b. Initially, we had hardcoded the initialization of the matrices.
    c. We then implemented a sync mode communication for matrix sizes that are divisible by x and y.

d. Later on we made all the communications requests asynchronous and splitted the computation in regions that are dependent on communications and otherwise.
e. Once this worked, we implemented generic matrix sizes and any grid geometry.
f. Finally we changed the initialization such that the rank 0 process passes the parameters to other processes for initialization.
g. Finally, we decided to vectorize the code - manually and using compiler hints.

Q1.c) If you improved the performance of your code in a sequence of steps, document the process. It may help clarify your explanation if you summarize the information in a list or a table. Discuss how the development evolved into your final design.

1. We started with the sync version of the code - blocking communication calls. We used MPI_Send and MPI_Recv initially.
2. Later on we replaced them with MPI_Isend and MPI_Irecv.
3. However, we were still waiting for the communication to be completed before doing the computation.
4. We then decided to split the computation in 2 parts - communication dependent computation and communication independent computation. Communication independent computation was done prior to awaiting for async calls to complete.
5. We also tried fusing and unfused implementations, but did not notice much of a difference.
6. For further improvement, we used AVX2 intrinsics and compiler hints for further improvements.
7. We wanted to use **openmp** for multi-threading but we had to skip it due to time constraints.

Section (2) - Result

Q2.a) Compare the single processor performance of your parallel MPI code against the performance of the original provided code. Measure and report MPI overhead from 1 to 16 cores).

Cores	Geometry	GF/s (no comm)	GF/s (with comm-sync)	GF/s (with comm-async)	Comm Overhead (with sync) %
1 (orig.)	N/A	11.57	N/A	N/A	N/A
1	1 x 1	12.77	9.448	12.41	26.01
2	1 x 2	19.3	18.07	24.48	6.37
3	1 x 3	38.8	33.55	36.58	13.53
4	1 x 4	52.18	45.22	42.38	13.33

5	1 x 5	66.07	63.47	56.82	3.93
6	1 x 6	79.63	75.6	74.17	5.06
7	1 x 7	88.11	88.01	83.51	0.11
8	1 x 8	105.2	102	95.91	3.04
9	1 x 9	114.1	113.1	109.3	0.87
10	1 x 10	123.4	112.5	122	8.83
11	1 x 11	134.3	122	128	9.15
12	1 x 12	157.2	130.9	139.6	16.73
13	1 x 13	172.1	161.1	155.9	6.39
14	1 x 14	183.5	172	166	6.26
15	1 x 15	189.5	159.3	176	15.93
16	1 x 16	194.9	180.5	191.4	7.38

Looking at the first two lines of the table, we compare the original code's performance on one core versus our code on one core, specifically the async version. The original code performs much better, which is to be expected. For our version of the code, even when communication is disabled, there is still additional overhead for setting up computation. For example, in the async version, computation is split into multiple sections. There is also setup overhead in the Init function that builds the matrix to be ready for computation.

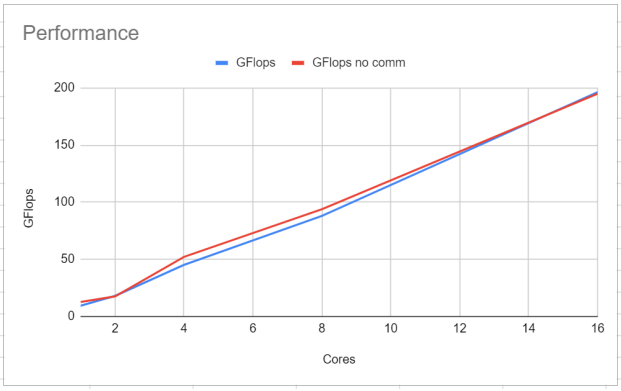
We can see that MPI overhead has an upward trend. Thus one could say that communication for initial distribution and and for boundary exchanges show their presence in a multi core setting.

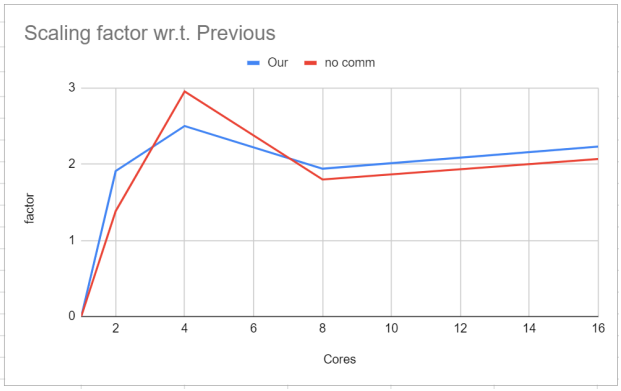
Q2.b) Conduct a strong scaling study: observe the running time as you successively double the number of cores, starting at p=1 core and ending at 16 cores on Expanse, while keeping N0 fixed.

The below table shows performance and geometries of this experiment. Additionally, the scaling is shown, which is the current GFlops divided by the previous run's GFlops. It is worth mentioning, for this and for all future tables, that results are collected from a single run on Expanse, as opposed to collecting an average from multiple, given the constraint on how much are allowed to run on the supercomputer. Analyzing this test, we know that in an ideal world everything would scale perfectly, i.e. 2 cores would perform at exactly double that of 1 core, and so on for larger numbers of cores. Our results (and our initial, more long-term tests on Sorken) indicate we achieve a good scaling.

One factor to consider in scaling is the overhead of MPI setup and communication. Even for 1 core, there is still some overhead even though MPI does not come into play and thus receives no benefit. This could impact the extra differing performance of 1 and 2 cores (aside from the obvious boost from using more cores), because of an overhead without benefit that no other number of cores will experience.

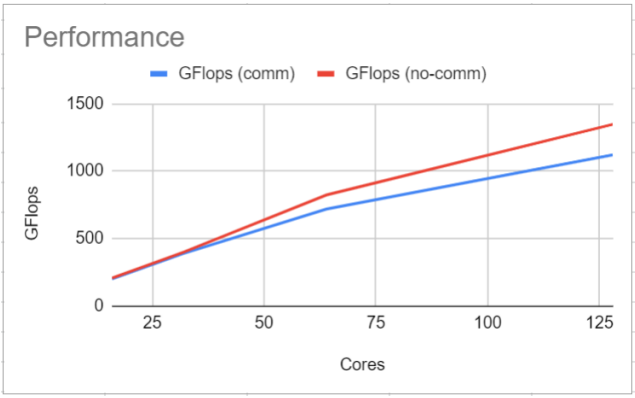
Cores	Geometry (x, y)	GFlops	GFlops no comm	Scale From Previous (Our)	Scale From Previous (no comm)
1	1 x 1	9.448	12.77	N/A	N/A
2	1 x 2	18.07	17.67	1.91	1.38
4	1 x 4	45.22	52.18	2.50	2.95
8	1 x 8	88.01	93.87	1.94	1.79
16	1 x 16	196.3	194.9	2.23	2.06

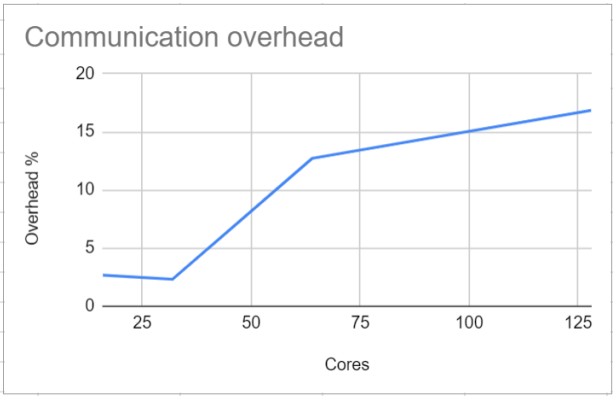




Q2.c) Conduct a strong scaling study on 16 to 128 cores on Expanse with size N1. Measure and report MPI communication overhead on Expanse. Supplement your discussion of scaling /overhead (i.e. what is the cost of communication).

Cores	Geometry	GFlops (comm)	GFlops (no-comm)	Scale From Previous (Ours)	Scale From Previous(no-comm)	Comm. Overhead(%)
16	1 x 16	201.5	207	N/A	N/A	2.65
32	1 x 32	389.8	399	1.93	1.9275	2.30
64	2 x 32	718.8	823.5	1.84	2.06	12.71
128	2 x 64	1120	1347	1.5581	1.63	16.85

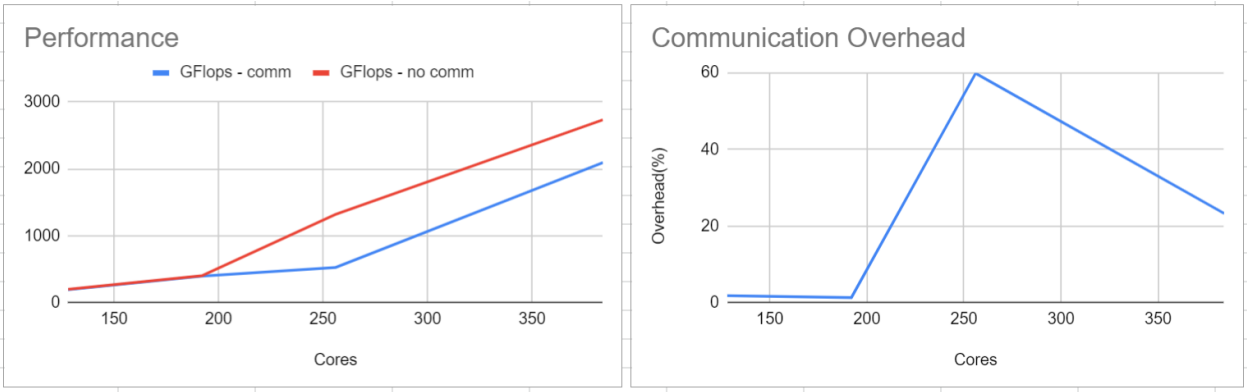




It can be seen from the graphs the performance with and without communication increases as we increase the number of cores. Thus there is an overall benefit in having a higher number of cores insteading of doing the whole computation on a single core. However, it can be noticed that as the scaling increases so does the communication overhead. The slowest nodes turn out to be the bottleneck which happen to be those processes that receive data slowly (those who are away in the grid) or those who are doing heavy computation. One more reason for linear rise in performance is that communication is happening inside the node as there are 128 cores per node in expanse and hence the communication overhead is visible as such.

Q2.d) Report the performance study from 128 to 384 cores with size N2. Measure the communication overhead. Use the knowledge from the geometry experiments in Section (3) to perform these large core count performance studies. Don't do an exhaustive search of geometries here as that will eat up your allocation. Please report your geometries in this Google Form: (as well as this table in your report) https://forms.gle/2CNW9Kzr9mNTEnCp8

Cores	Geometry	GFlops - comm	GFlops - no comm	Comm. Overhead(%)
128	4 x 32	199.4	203.3	1.91
192	12 x 16	401	406.7	1.40
256	16 x 16	530.2	1322	59.89
384	4 x 96	2095	2733	23.34



An interesting result is the superlinear performance growth, especially between 256 and 384 cores. Assuming a performance of 200 GF at 128 cores, then a purely linear growth would give us 300 GF at 192 cores, 400 GF at 256 cores, and 600 GF at 384 cores, but our performance at 384 is more than triple this.

One explanation for this is better memory usage. In the case of 128 cores with N=8000, each core must work on 500,000 cells. With more cores, this number decreases linearly. So, because of a linear increase in cores, and a linear decrease in memory required for each process, the effect compounds and results in the superlinear growth we see above. Based on this table we can not say for sure where the optimal memory utilization on N2 will fall, but further scaling results shown below, where we test these number of cores on N1, indicate that 384 cores may fall close to optimal for N2.

Another explanation would be geometries. Row dominant geometries favors row major communication. The cost of packing and unpacking smaller values in columns is less.

Q2.e) Explain the communication overhead differences observed between <=128 cores and >128 cores.

As described in the above development sections, we specifically sought to minimize such overhead when using MPI by performing as many calculations as possible, i.e. did not depend on ghost cell data, before waiting on messages to be received.

It can be seen that the communication overhead between <= 128 cores and N1 as 1800 increases linearly whereas for N2 it increases from 192 to 256 and then decreases from 256 to 384. It can be seen that adding more cores benefits as the computation gain is more than communication overhead. The geometry of nodes plays a significant role in communication overhead.

Q2.f) Report cost of computation for each of 128, 256 and 384 cores for N2.

The below table reports computation cost as the product of the number of cores multiplied by the computation time in seconds. All tests performed with the optimal geometry found in Q2.d, and with N=8000 and i=8000.

Cores	Geometry	Computation Cost = #cores x computation time
128	4 x 32	9191.99
192	12 x 16	6895.45
256	16 x 16	7041.69
384	4 x 96	2756.58

The overall computation cost was found to be least with 384 cores, by a wide margin. This follows on from Q2.d, where our performance saw huge gains when moving to 384 that was not seen in other numbers of cores. So, based on this lowest cost then it is most optimal to run with 384 cores.

This is pretty much what we expected. With a large number of cores, we get the benefit of the many individual memory buses to decrease our computation time on top of the extra processing power.

Section (3) - Determining Geometry

Q3.a) For p=128, report the top-performing geometries at N1. Report all top-performing geometries (within 10% of the top).

The following geometries measure within 10% of the highest performing measurement of 1053 GFlops, i.e. 947.7 and above.

N = 1800

Cores	Geometry	GFlops
128	1 x 128	973.7
128	2 x 64	1120
128	4 x 32	1022
128	8 x 16	949.4

Q3.b) Describe the patterns you see and hypothesize the reasons for those patterns. Why were the above-mentioned geometries chosen as the highest performing? Use the knowledge from these geometry experiments to perform the large core count performance study.

Given we operate on a fixed number of cores, it follows that the computation cost is least for the greatest performer. Communication overhead however varies for the geometries. Following the above pattern of geometry changes (becoming more square before moving to rectangles in the opposite direction), the performance continues to decrease.

As for the rectangular geometries in the opposite direction, where X is larger and Y is smaller, performance decreases are a result of poorer communication in this direction. Small X

and large Y, which we find optimal, favors east-west communication. This actually came as a surprise to us, because we know that east-west communication has the challenge of taking a column of data in a row major matrix. For north-south communication, the data is easier to access, which would seemingly then be more favored. It is possible that our specific implementation contains some flaw that hurts performance for north-south communication, but on a deep analysis of our code and attempted changes, we did not see noticeable differences in the results.

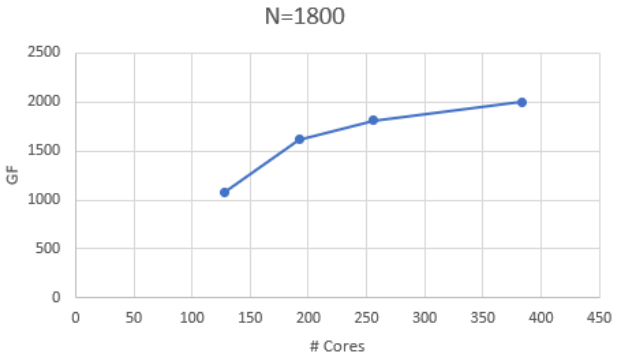
One of the possible reasons could be that taller geometries are more cache optimal w.r.t to a stencil computation operation as the top, down ,left, right elements are pretty near the cell and can leverage better cache use.

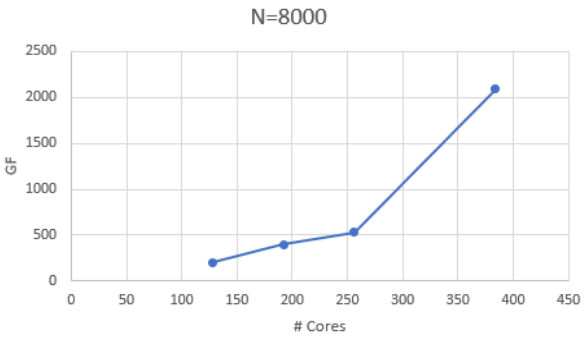
Section (4) - Strong and Weak Scaling

Q4.a) Run your best N1 geometry (or as close as you can get) at 128, 192, 256 and 384 cores. Compare and graph your results from N1 and N2 for 128, 192, 256 and 384 cores.

Cores	Geometry (N1)	GFlops (N1)	GFlops (N2)
128	2 x 64	1083	199.4
192	2 x 96	1619	401.0
256	2 x 128	1810	530.2
384	2 x 192	1999	2095

We chose geometries of 2xN/2 for all number of cores. This geometry pattern tended to perform the best or near best based on our results for N1 with 16 to 128 cores. We set I to 200000 to ensure >10s runtime for all tests.





Q4.b) Explain or hypothesize differences in the behavior of both strong scaling experiments for N1 and N2?

The results shown in Q4.a differ quite significantly from our previous results shown Q2.d for N2. Previously, for N2 with optimal geometry we found a superlinear growth. Now, with the

same core count but running on N1 with a roughly optimal geometry pattern discovered against 16 to 128 cores, the growth becomes sublinear.

Just as we discussed earlier, memory may impact this. Given we are now running against N=1800, which is significantly smaller than N=8000, the percentage of data a process can hold in memory is much higher. Our results indicate that memory has little if any impact on performance here given that decreasing block sizes are not improving performance. This also explains why performance is greater at 128, but roughly equal at 384. Memory is a limiting factor in only N2 at 128 cores, and it appears that neither N1 nor N2 is memory limited at 384 cores.

Expanse has 128 cores per node, so when we go beyond that the communication cost becomes more significant. As a result the rate of performance increase drops for cores more than 128. This can be seen for N = 1800.

However, for N = 8000, we would have less work elements per core, when cores are 384. Thus capacity misses would not be a big issue which seems contributing towards super linear growth.

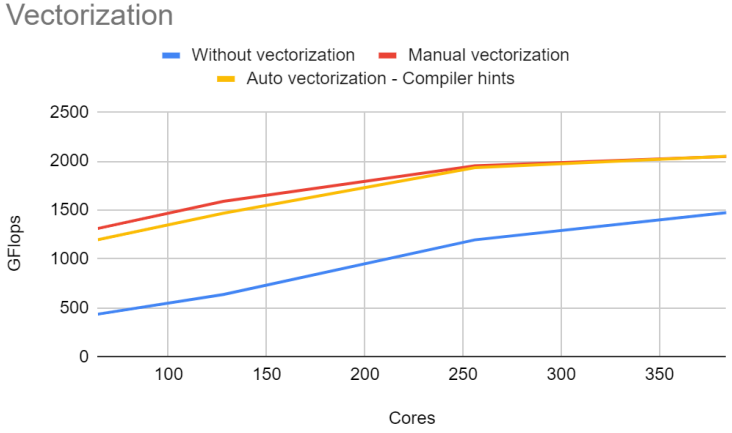
Thus, our results purely rely on a growing number of cores. In a perfect world we may see a perfectly linear growth, but due to expected overhead and other factors our graph falls below.

Section (5) Extra Credit

EC-b) Vectorization

N=1800

Cores	Geometry	Without vectorization	Manual vectorization	Auto vectorization
64	2x32	437.9	1313	1198
128	2x64	638.4	1590	1469
256	2x128	1197	1954	1934
384	2x192	1474	2048	2050



- It can be seen that there is a huge performance improvement with vectorization.
- `objdump -d apf | grep xmm | wc -l` can be used to see how many vectorization instructions were there
- **No vectorization**
  - `C++FLAGS += -fno-tree-vectorize`
  - ```
    make clean
    make mpi=1 novec=1
    ```

- **Manual vectorization**
  - Was done for both sync and async versions of the code. We have reported the performance of the sync version of the code.
  - For manual vectorization you will have to copy solve_sync_vec.cpp or solve_async_vec.cpp and overwrite solve.cpp
  - xmm_count

    `C++FLAGS += -fno-tree-vectorize -DSSE_VEC -mavx2 -march=native`

    ```
    make clean
    make mpi=1 vec=1
    ```

- **Auto-vectorization**
  - seems to be more efficient than manual vectorization as it vectorizes a few other loops. This can be inspected using `-fopt-info-vec-optimized` flag
  - ```
    C++FLAGS += -DAUTOVEC
    C++FLAGS +=  -ftree-vectorize -march=native -ffast-math -freciprocal-math
    ```
  - ```
    make clean
    make mpi=1 autovec=1
    ```

Section(6) - Potential Future work

- Try more with compiler optimizations
- More grid searching for finding optimal geometries
- Further splitting of async calls and make more overlapping with computation (use primitives like MPI_Waitany instead of MPI_Waitall)
- Try MPI+X model (MPI with multithreading - distributed memory + shared memory model)

Section (7) - References

[1] https://mpitutorial.com/tutorials/

[2] https://mpi.deino.net/mpi_functions/index.htm
    ```