CSE 260 PA3

Gehua Qin A53283886 qgehua@eng.ucsd.edu Xinyu Xie A53285829 x1xie@eng.ucsd.edu

Section 1 - Performance Study

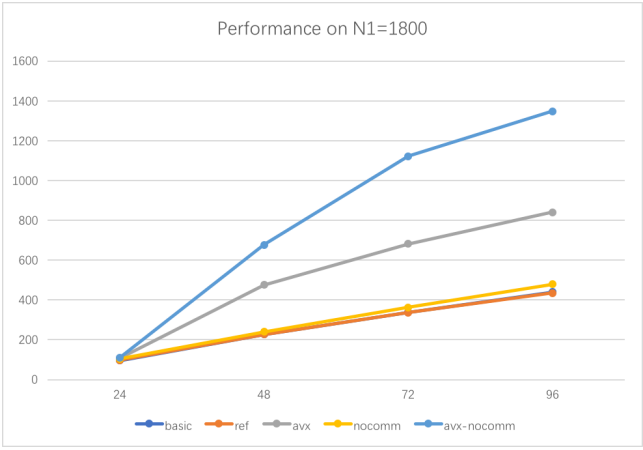

Figure 1 - performance of N1=1800 (24, 48, 72, 96 cores)

The performance result can be seen in Figure 1. In general, for $n = 1800$ with 24, 48, 72, and 96 cores,the scaling is aound 1. We have also shown our results on avx2 auto vectorization, the performance is greatly better than original version.

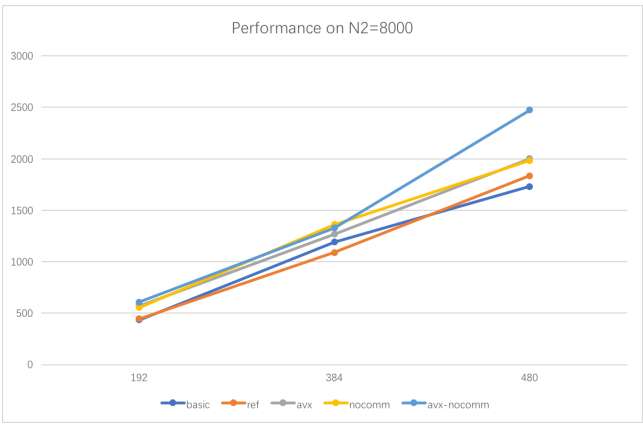

Figure 2 - performance of N2=8000 (192, 240, 480 cores)

In Figure 2, the performance with $n = 8000$ for 192, 384 and 480 also achieves strong scaling around 1. Furthermore, for $np = 480, n = 8000$, and $i = 2000$, the performance is greater than 1TF. The deteiled numerical value is illustrated in the following table.

n	cores	basic	ref	avx	nocomm	avx-nocomm
1800	24	95.25	97.48	107.5	103.5	110.6
1800	48	226.8	226.1	475.9	239.7	676.4
1800	72	337.6	336	681.2	362.8	1122
1800	96	438.8	435.3	839.7	478.6	1347
8000	192	435.1	444.5	572.9	556.2	607.4
8000	384	1190	1090	1268	1361	1327
8000	480	1732	1834	2003	1982	2474

Section 2 - Analysis

Aliev-Panfilov cardiac simulation

Setting up the initial conditions

The simulation algorithm begins with initializing the excitation matrix \(E\), \(E*{prev}\) and the recovery matrix \(R\). The elements in the matrix denote the respective state parameter value of the cell with that index. The initial values of \(E{prev}\) and \(R\) are a half-plane of 0's and 1's, with \(E_{prev}\) being divided horizontally and \(R\) divided vertically. Because the algorithm requires the four neighbors (north, south, east and west) to update the values at each step, both matrices are zero padded by 1 row on both directions and 1 column on both directions (i.e. a m-by-n matrix is padded to (m+2)-by-(n+2) by filling padding area with 0). In the following sections, for convenience, we call the original non-padded matrix as '*computational area*'. Also for space for 'ghost cells', we padded each sub-matrix in the same way.

Boundary cases on the edges of the global matrix

As mentioned above, we padded the initial matrices so that every element in the computational area have exactly the four neighbors. To avoid special IF cases that would have to be tested for every stencil point, instead we insert an extra boundary layer around the global grid. This way, even the tiles (and stencil points inside of them) that have no neighbors on one or two sides do not need any special handling. The trick is to set the boundary values so that the partial derivative in the normal direction is 0, which is described in detail in the **Ghost cell exchange** part.

Even distribution on processors

Because we suppose that the initial values are read from a file on process 0, process 0 needs to distribute the values to all the other workers. If the processor geometry exactly divides the size of the matrix, it's automatically distributes the global matrix evenly onto all processors.
However, if the processor geometry cannot exactly divides the matrix's dimension, we need to take severl steps to ensure each processor does not get allotted more work than a row or a column of elements so that we don't need to worry about to additive trade-off.
We accomplish this by taking two steps:

1. Given a n-by-n matrix, each processor is initially assigned a sub-matrix with \(\lfloor n/py \rfloor\) rows and \(\lfloor n/px \rfloor\) columns.
2. At this time, the global matrix has \(n\%py\) rows and \(n\%px\) columns rest to be assigned. To achieve our goal, we make use of the processors's rank.
   a). If \(rank/px<n\%py\), which means this sub-matrix hasn't reach the vertical boundary of global matrix and we haven't divide all rows, we increment the sub-matrix's rows by 1.
   b). If \(rank\%py<n\%px\), which means this sub-matrix hasn't reach the horizonal boundary of global matrix and we haven't divide all columns, we increment the sub-matrix's column by 1.

Hence, we have the whole matrix divided almost evenly with each processor not getting allotted more work than a row or a column of elements. For quick understanding, an example of divide a 10-by-10 matrix to 3*4 processor geometry is shown as follow:

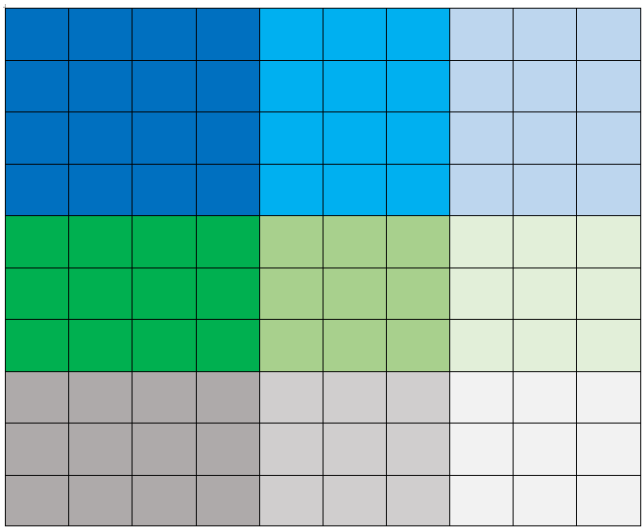

Figure 3 - An evenly division of a 10-by-10 matrix by 3*4 processor geometry

Ghost cell exchanges

Since the computation of every element in \(E\) relies on the four neighbours, we have to deal with the boundary case. When we compute the value of an element at computational boundary, we need the so-called 'ghost cells' to act as the neighbours of this element. There're two cases we need to take into consideration:

1. The 'ghost cell' falls outside the computational area (i.e. the target element is at the edge of non-padded matrix \(E\)). In this case, the target 'cell' simply uses the opposing neighbor's value instead. For example, if the cell's north neighbor is missing, which means it's at the first row in the non-padded matrix \(E\), then it uses its south neighbor as north neighbor by filling the top padding cell with the value of its south neighbor.
2. A more complicated case is that the 'gohst cell' falls within the computational area but outside the sub-matrix assigned to the processor. In this case, the processor must retrieve the value of the 'ghost cell' from the corresponding neighbor processor. In application, we can achieve this goal using MPI calls to send and receive the values among processors, which will be described in the next part.

MPI implementation

As described above, when the required neighbor cells is held by neighbor processors, we take use of MPI calls to transfer data. In our implementation, we use ***MPI_Isend*** and ***MPI_Irecv*** because they are non-blocking and asynchronous. Before going into the computation kernel, a processor checks if it requires data from neighbor processors. If so, the processor create puts the required elements into a send buffer and send to the neighbor with ***MPI_Isend***. Simultaneously, the processor call ***MPI_Irecv*** to indicate that it's ready to receive data and save into a receive buffer. After each send or receive call, the processor calls ***MPI_Wait*** to let the data trasmission complete.
Then the program enters the computation kernel to update cells' values. When the computation loop ends, the processor calls ***MPI_Reduce*** twice. The first is to calculate the local sum of the sub-matrix assigned to the processor, and the second fetch the global maximum value. Both values are transmitted to the processor of rank 0 so that the final L2 norm is computed.

Optimizations to the computation kernel

Vectorization

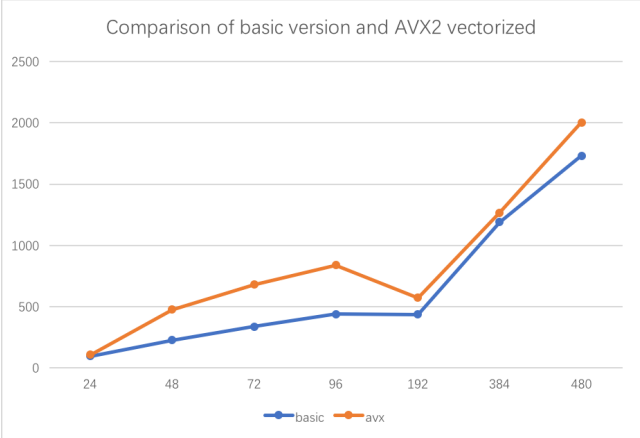

Figure 4 Comparison of basic version and AVX2 auto vectorized version

In Figure 7, we illustrate the comparison between basic version (automatically vectorized, but not on AVX2) and AVX2 vectorized version. It is quite obvious that the version on AVX vectorization is always superior to the basic version. Moreover, at 480 cores, the performance reaches 2 TF.

However, the AVX2 vectorized version does not scale as well as the original version on 24, 48, 72 and 96 cores. The reason could be that AVX2 vectorization would only decrease the computation time, but not the communication time. The overhead becomes significant on the scaling performance.

MPI overhead analysis

We first examing our program on Bang with a small 400-by-400 matrix using only one core. Our experiment result suggests that there's almost no differences between MPI and non-MPI in this case. The communication overhead is only 0.0028s. It's because with only one core, there's no trade-off caused by communications among processors. The very slight difference results from the communication with process 0 which take place at the final ***MPI_Reduce*** call, and the cost of memory free due to our implementation (see **Section 3**).

Then we gradually scale to 8 cores on Bang. From Figure 5 and Table 1, we still find the overhead is very small and even negligible. This may because when the core number is little, every processor deals with a lot of computation but only a few communication for 'ghost cells', which means the computation part dominates the communication time. Thus, the communication has very slight effects on the performance of program. Also, another possiblt reason is that on Bang, the processors are very close to each other, which makes the data transition very fast.

Core	Gflops/sec	Communication overhead
1	0.455	0
2	0.908	0.0055
4	1.81	0.0113
8	3.58	0.0183

Table 1. Communication overhead when N0=400 with core = 2,4,8

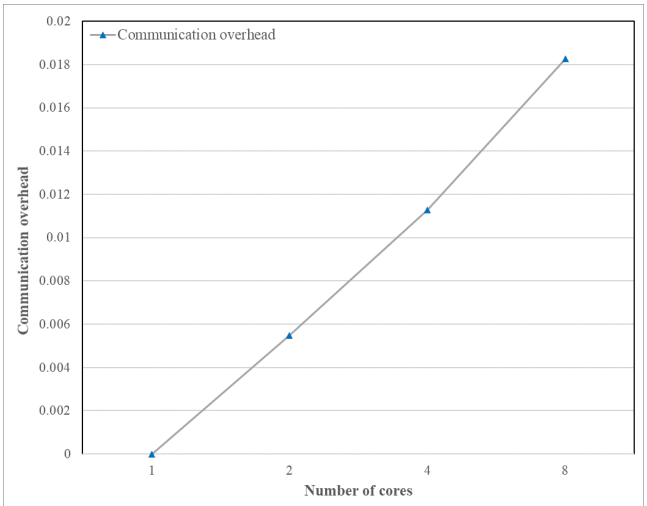

Figure 5. Communication overhead on different cores when N=400

MPI overhead for large number of cores on Comet

Next we scale to 24, 48, 96 cores with a 1800-by-1800 matrix. As shown in Figure 6, there's obviously an increasing of communication overhead as the number of cores grows. This is because as the number of cores increases, when computing the matrix of the same size, each processor is dealing with a smaller sub-matrix while a larger amount of communication to fill in 'ghost cells'. And because of sychronization, we always have to let each processor wait for completion of data transmission. Hence, the computation becomes faster but the cost of communication becomes greater, which results in a more significant effects of communication on performance.

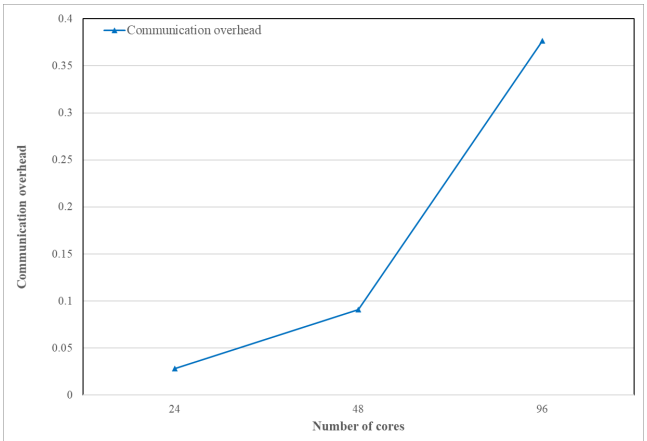

Figure 6. Communication overhead on different cores when N=1800

Section 3 - Development Process

Implementation of Aliev-Panfilov cardiac simulation

Our work is implemented follow the procedure described in **Section 2**. During the implementation of Aliev-Panfilov cardiac simulation algorithm, we generally fix two problems:

Matrix division

Almost the most difficult part is how to tile the global matrix to the given geometry and correctly send each sub-matrix to a processor. Although the algorithm is just as described in **Section 2**, many details requires to be solved. For our implementation of this part, please refer to the code file as we typically write detailed comments for easy understanding.

Work distribution

At the very beginning, we followed the framework of starter code using ***cblock*** for sub-matrix information and boradcasting the whole \(E\), \(E_{prev}\) and \(R\) matrices to each processor. However, as the number of processor grows, we found that passing the whole matrices reduces the performance a lot. To keep the definition of ***cblock*** unchanged, we defined a new struct called ***myblock*** for holding only the sub-matrix and its dimension. When initialization, we let process 0 only send sub-matrix using ***myblock***. This strategy showed a great improvement of performance. Also, it brings new trade-off to free the memory for the copied sub-matrices, but this trade-off is very small when matrix is large.

Optimizations of computation kernel

Vectorization

According to A Guide to Vectorization with Intel C++ compilers, §6.2.2, at http://tinyurl.com/qc46qjs, for the inner loop of computing E and R, we can provide SIMD auto compiling information (`#pragma simd`) to the compiler to automatically generate machine code on AVX2 instruction set. In the compiling process, we need additionally add compiling flag `C++FLAGS+=-march=core-avx2` to actually compile it on AVX2 instruction set.

Memory access

We tried some small tricks to optimiza on register and cache. Initially, the access to \(E\), \(E_{prev}\) and \(R\) is accomplished using pointers to the whole sub-matrix. We changed it into global variables so that to make full use of registers instead of always accessing memory. Also we edited the unfused computation procedure to include computation of PDE and ODE into one outer loop in order to increase the cache hit rate.
These two optimization did show a slight improvement on performance, but meanwhile it turns out that the performance is more unstable than before, which prevents us from going on with them. We guess the reason actually has something to do with our second change. Although to some extent the cache hit rate increases, because PDE and ODE are included in one outer loop, the compiler's auto vectorization actually does not work as well as before.

Section 4 - Determine Geometrics

Case study 1 - geometry of 96 cores

We study the geometry of 96 cores to see how geometry affects the program's performance and raise our hypothesis which may account for the observation.

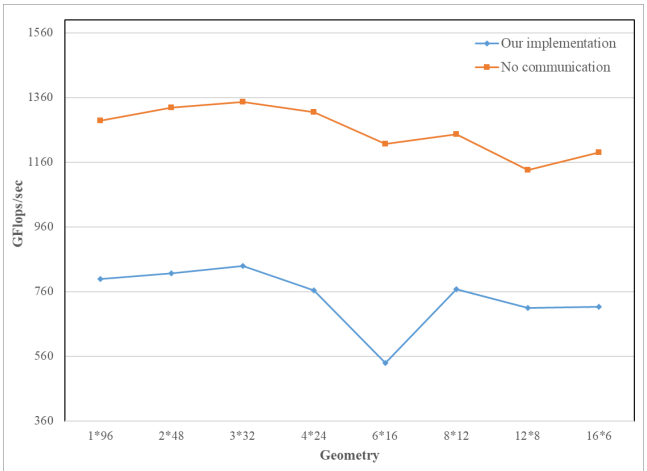

Figure 7. Performance of different geometries on 96 cores

Geometry	Gflops/sec	Computation ratio	Communication overhead
3*32	839.7	0.6233	0.3767
2*48	816.1	0.6149	0.3851
8*12	767	0.6154	0.3846
4*24	763.4	0.5810	0.4190
16*6	712.6	0.5982	0.4018

Table 2. Top performance among different geometries of 96 cores

As shown in Figure 7 and Table 2, we see that as \(px\) increses and \(py\) decreases (the sudden drop at 6*16 seems an accidental case due to the machine since we see similar case at 24 and 48 cores as well for some time, but we have no time to rerun many other experiments),the performance first increases and then decreases. This is due to three facts:

1. The processor in x-dimension should be fewer than that in y-dimension because according to our implementation, a smaller \(px\) means more columns in a sub-matrix, which reduces cache missing (See *Even distribution on processors* part in **Section 2**).
2. If \(px\) is too small, which means the row number of submatrix is very small, there will be too much 'ghost cell' exchange between top and bottom. Similarly if \(py\) is too small, there will be too much 'ghost cell' exchange between left and right, which causes the situation even worse. Figure 8 can prove our hypothesis. It shows that when we select a geometry of \(3*32\), we achieved the highest computation ratio (i.e. the ratio of time that the program spends on computation rather than communication) which also means the smallest communication overhead.

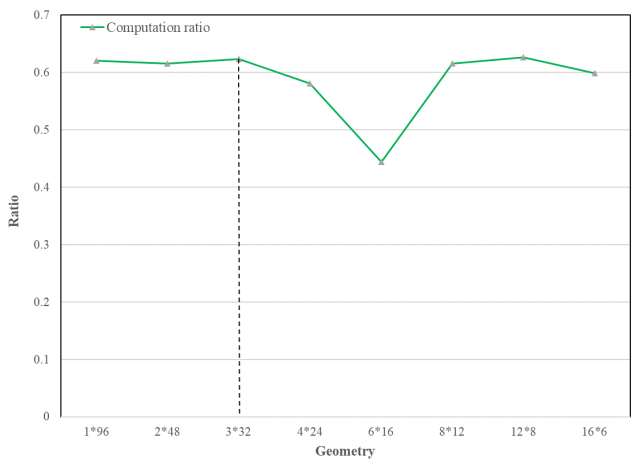

Figure 8. Computation ratio of programming of different geometries on 96 cores

Therefore, it's reasonably to have a curve like Figure 7 and a best geometry is always found at a relatively small \(px\) and a relatively large \(py\).

Case study 2 - geometry of 480 cores

To support our hypothesis, we report the measurement of the case of 480 cores, which is a strong scale of 96 cores. As shown in Figure 9, Figure 10 and Table 3, we can draw the same conclusion as the 96-core case, that a peak performance is achieved at a relatively small \(px\) and a relatively large \(py\), and usually at this point the communication overhead is the least.

Geometry	Gflops/sec	Computation ratio	Communication overhead
15*32	2003	0.8414	0.1586
16*30	1988	0.7537	0.2463
12*40	1983	0.8024	0.1976
6*80	1934	0.7201	0.2799
20*24	1899	0.8257	0.1743

Table 3. top performance among different geometries of 480 cores

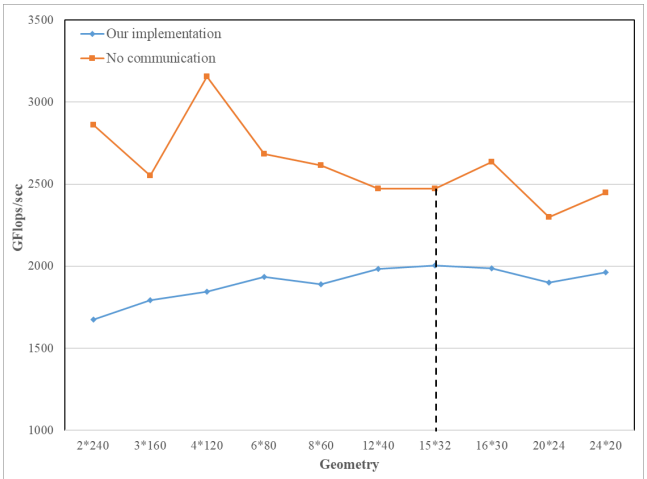

Figure 9. Performance of different geometries on 480 cores

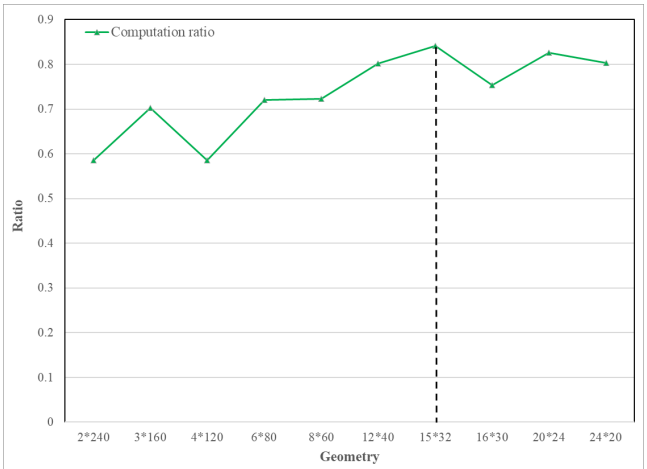

Figure 10. Computation ratio of programming of different geometries on 480 cores

Section 5 - Future Work

We would like to try manually vectorize the inner loop, and compare the performance between manual vectorization and auto vectorization on AVX2.

Section 6 - References

[1] https://sites.google.com/eng.ucsd.edu/cse260-winter-2019/assignments/assignment-3?authuser=0 [2] A Guide to Vectorization with Intel C++ compilers, §6.2.2, at http://tinyurl.com/qc46qjs.