HPC: Scaling Aliev-Panfilov Electro-Cardiac Simulation Using OpenMPI

Teofilo Erin Zosa IV
Department of Computer Science and Engineering
The University of California, San Diego
La Jolla, CA 92092
PID: A53242430
tzosa@ucsd.edu

Abstract

In this assignment we were tasked with optimizing the Aliev-Panfilov electro-cardiac simulation algorithm using OpenMPI to run on the COMET supercomputer at the San Diego Super Computer Center, with the goal of achieving Scaling efficiency of 1.0 and roughly 1 Tflops/s using 480 processor cores. This program achieves a peak of 1.82 Tflops/s via the use of asynchronous non-blocking communication and the exploitation of spatial and temporal locality inherent to computed elements.

1 Background

The Aliev-Panfilov algorithm involves a series of partial differential equations and ordinary differential equations over discrete points in a mesh where each point represents a cardiomyocyte in a piece of cardiac tissue. At each time step, the algorithm computes two state variables: the excitation (E) of a cell via a two-dimensional 5-point stencil over a cell and its neighbors at time $t-1$, and the recovery (R) of a cell at time t Aliev and Panfilov [1996].

2 Method

2.1 Benchmarking Machine(Core Count≤ 8): Bang

For small scale studies (i.e. eight or fewer cores), performance metrics were obtained on the UCSD Bang computing cluster, whose computing units are Dell PowerEdge 1950 servers with two quad-core Intel Xeon E5345 processors. Each processor runs at a clock speed of 2.33 GHz and has two 4MB (each shared between two cores) unified L2 caches (16-way set associative). Each core has its own 32KB L1i cache (8-way set associative) and 32KB L2d cache (8-way set associative). All caches have a line size of 64 bytes.World [2017a][1]

2.2 Benchmarking Machine(Core Count≥ 24): COMET Supercomputer

For large scale studies (viz. 28 to 480 cores),Performance metrics were obtained on the COMET supercomputer located at the San Diego Supercomputer Center, whose computing units are Dell dual socket server nodes with 64 GB of DDR4 DRAM and a single dodecacore Intel Xeon E5-2680 v3 processor per socket. Each node also has 320 GB of SSD local scratch memory which is shared between socketsSupercomputer Center [2017]. Each

[1]Additional characteristics gleaned from 'lscpu' and 'cat /proc/cpuinfo' commands

processor runs at a clock speed of 2.50 GHz and has a shared 30 MB unified L3 cache (20-way set associative), 12 256KB unified L2 caches (8-way set associative) and 12 32KB L1i caches (8-way set associative) in addition to eight 32KB L1d caches (8-way set associative). All caches have a line size of 64 bytes.World [2017b]

2.3 Program Behavior

Algorithm 1 Parallel Aliev-Panfilov Algorithm

```
$p_{row}=\frac{myrank}{n_p}$
$p_{col}=myrank \mod n_p$
$submesh_n=\frac{n}{n_p}$
$submesh_m=\frac{m}{m_p}$
$ExtraRows=m - submesh_m * m_p$
$ExtraCols=n - submesh_n * n_p$
if $p_{row} < ExtraRows$ then
    $submesh_m{++}$
if $p_{col} < ExtraCols$ then
    $submesh_n{++}$
for $i = 0; i < iters; {++}i$ do
    for $Neighbor \in [Top, Bottom, Left, Right]$ do
        if $has(Neighbor)$ then
            $MPI\_Isend(CompCellsTo_{Neighbor}, dst = Neighbor)$
            $MPI\_Irecv(GhostCellsFrom_{Neighbor}, src = Neighbor)$
    for $Neighbor \in [Top, Bottom, Left, Right]$ do
        if $has(Neighbor)$ then
            $MPI\_Wait$
    $Update(GhostCells)$
    $RunKernel()$                                        ▷ to compute $E_t, R$
```

2.3.1 Initialization of State Variables

The program begins by initializing the excitation parameters (E_t and E_{t-1}) and the recovery parameter (R_t) of each point in the mesh. Both E and R are represented by matrices whose indices represent that index's respective state parameter value. The initial values for E_{t-1} and R are a half-plane of 0's and 1's, with E being subdivided down the column dimensions (i.e. vertically) and R down the row dimension (i.e. horizontally). As the 5-point stencil requires neighboring values to compute the value at a point, both matrices are zero-padded (i.e. cells immediately outside the computational mesh are initialized to 0).

2.3.2 Division of the Computational Mesh

The resultant matrices are then divided evenly according to a user specified 2-D processor geometry such that each processor has a unique computational window corresponding to their respective place in the geometry.

Each processor is initially assigned $\lfloor \frac{m}{m_p} \rfloor$ rows and $\lfloor \frac{n}{n_p} \rfloor$[2] columns. If a processor's row or column is less than the number of leftover rows or columns, respectively, it is assigned an extra corresponding dimension. This way, each processor is responsible for a maximum of 1 column and/or 1 row more or less than any another processor in the work group.

2.3.3 Distribution of Submeshes

To simulate a master node reading in data, the first processor (i.e. the processor with rank 0) sweeps over the initial excitation grid and sends copies of submeshes to corresponding

[2] m_p is the number of processor rows and n_p is the number of processor columns

processors. Each processor then copies the submesh onto their local copies of E_t, E_{t-1}, and R, respectively.

2.4 Ghost Cell Exchange

As the value of E at a point relies on neighboring values at E_{t-1}, the computation of E for cells at computational boundaries represent a special case. These cells rely on cells we term 'ghost cells' as they are not defined in the computational window. If a ghost cell falls outside of the computational mesh entirely, a cell simply uses the opposing neighbor's value instead (i.e. the east neighbor if no west neighbor exists, the north neighbor if no south neighbor exists, etc.). If, however, the ghost cell is within the computational mesh but outside a processor's submesh, the processor must retrieve that value from the corresponding neighbor processor. In the program implemntation, this was done through non-blocking asynchronous I/O via MPI (viz. 'MPI_Isend's and 'MPI_Irecv's). Immediately preceding an iteration of computation, processors would determine if they had neighboring processors from whom they needed to send and receive data. If so, the processor would put elements it needed to send into a buffer and transmit that buffer via 'MPI_Isend'. It would simultaneously indicate that it is ready to receive data into its own local buffer. After all send and receive calls are made, the processor issues 'MPI_Wait' commands, one for a each 'MPI_Isend' and 'MPI_Irecv' command it has issued. It then waits for the completion of the data transmissions, updates all the ghost cells in its submesh, and precedes to calculate the values of E_t and R_t for its elements.

At the end of the final iteration, the processors call 'MPI_Reduce' twice; once to sum the local sum of squares, and again to fetch the global maximum value. Both output values are transmitted to the rank 0 processor and the true L2 norm is computed via the global sum of squares.

2.4.1 Kernel Optimizations

Algorithm 2 Fused Kernel

```
$cols = submesh_n + 2$
$rows = submesh_m + 2$
**for** $i = 1stComputationalCell; i <= lastComputationalCell; i{+}{=}submesh_n$ **do**
    $E_{tmp} = E[i]$
    $E_{tmp_{prev}} = E_{prev}[i]$
    $E_{top_{prev}} = E_{prev}[i + cols]$
    $E_{bottom_{prev}} = E_{prev}[i - cols]$
    $R_{tmp} = R[i]$

    **for** $j = 0; j < submesh_n; i{+}{+}$ **do**
        $left = E_{tmp_{prev}}[j - 1]$
        $curr = E_{tmp_{prev}}[j]$
        $right = E_{tmp_{prev}}[j + 1]$
        $top = E_{top_{prev}}[j]$
        $bottom = E_{bottom_{prev}}[j]$
        $r = R_{tmp}[j]$

        $E_{tmp}[j] = compute\_PDE_E(left, curr, right, top, bottom) + compute\_ODE_E(curr, r)$
        $R_{tmp}[j] = compute\_ODE_R(curr, r)$
```

In the original implementation of the algorithm, there were two computational kernels, one that separates the PDE (5-point stencil) and the ODE into separate loops, and another that attempts to update both within the same loop. I will refer to the former version as the 'unfused' kernel and the latter version as the 'fused' kernel.

The first optimization made was involved the exploitation of temporal locality by moving frequently accessed array elements into local variables that were declared with the 'register'

Algorithm 3 Unfused Kernel

```
$cols = submesh_n + 2$
$rows = submesh_m + 2$
**for** $i = 1stComputationalCell; i <= lastComputationalCell; i += submesh_n$ **do**
    $E_{tmp} = E[i]$
    $E_{tmp_{prev}} = E_{prev}[i]$
    $E_{top_{prev}} = E_{prev}[i + cols]$
    $E_{bottom_{prev}} = E_{prev}[i - cols]$
    $R_{tmp} = R[i]$

    **for** $j = 0; j < submesh_n; i{+}{+}$ **do**
        $left = E_{tmp_{prev}}[j - 1]$
        $curr = E_{tmp_{prev}}[j]$
        $right = E_{tmp_{prev}}[j + 1]$
        $top = E_{top_{prev}}[j]$
        $bottom = E_{bottom_{prev}}[j]$

        $E_{tmp}[j] = compute\_PDE_E(left, curr, right, top, bottom)$

    **for** $j = 0; j < submesh_n; i{+}{+}$ **do**
        $curr = E_{tmp_{prev}}[j]$
        $r = R_{tmp}[j]$

        $E_{tmp}[j] {+}{=} compute\_ODE_E(curr, r)$
        $R_{tmp}[j] = compute\_ODE_R(curr, r)$
```

keyword, to encourage the use of keeping these values in fast memory. This included the excitation value of the current point at time $t-1$, the recovery value. To exploit spatial locality, this same strategy was repeated with pointers to the north and south rows of the current point. This optimization was performed for both kernels. For the 'fused' kernel, the PDE and ODE were calculated simultaneously for a single assignment into E_t (see 2). For the 'unfused' kernel, I then moved the ODE inner loop into the same outer loop as the PDE inner loop, producing a 'semi-fused' kernel (see 3). For nomenclature consistency, I consider the 'semi-fused' kernel a type of 'unfused' kernel and continue to refer to it in this paper as so.

2.5 Experimental Setup

To determine whether the program implementation was sufficiently scaling, tests were carried out on meshes with dimensions 400 x 400 for N = 1, 2, 4, and 8 on bang. On Comet, scaling efficiency tests were conducted on meshes with dimensions 1800 x 1800 (N = 24, 48, 96) and 8000 x 8000 (N = 96, 192, 384, 480). For the following versions:

1. APF-Fused
2. APF-Unfused
3. APF-Ref

. Performance values were taken as the average of three tests for each matrix dimension.

2.5.1 2-D Processor Geometries COMET Tests

For tests with mesh dimensions 1800 x 1800 and N = 24, 48, and 96, 7, 8, and 9 geometries, respectively, were tested. These geometries included values in the range Nx1 to 1xN, with successive geometries decreasing in the y dimension and increasing in the x dimension, usually by some integer multiple (i.e. Nx1, $\frac{N}{2}$x2, etc.), and a value where the x dimension was equal to the number of nodes requested (N_{nodes}) and the y dimension was equal to the number of tasks per node (N_{tasks}) which was always 24 as each node contained 24 physical processor cores. From initial tests, it was hypothesized that some geometry centered around the latter value would produce the best performance at that core count.

For tests with mesh dimensions 8000 x 8000 and N = 96, 192, 384, 480, 5 geometries were tested for each. These geometries had the characteristic that the central value had dimensions x = N_{nodes} and y =N_{tasks} and the dimensions to either side of a geometry increased or decreased by a factor of 2 (i.e. the left value has x = $\frac{N_{nodes}}{2}$ and y =$N_{tasks} \times 2$).

3 Results

Table 1: $\leq$ 96 Cores: Kernel Performance(Gflops/s)

	w/ Communication			w/o Communication		
N	Fused	Unfused	Ref	Fused	Unfused	Ref
1 (NO MPI)	N/A	N/A	N/A	1.24	1.88	0.46
1	1.24	1.89	0.46	1.24	1.9	0.46
2	2.48	3.78	0.91	2.49	3.83	0.91
4	4.91	7.48	1.81	4.98	7.65	1.82
8	9.64	14.50	3.60	9.96	15.30	3.64
24	101.90	104.20	67.95	108.80	115.30	77.70
48	201.10	296.50	168.10	244.00	345.00	172.70
96	362.70	545.10	317.90	494.7	679.30	341.80

Table 2: $\leq$ 96 Cores: Communication Overhead

N	Fused	Unfused	Ref
1	0.0%	0.4%	0.0%
2	0.4%	1.3%	0.0%
4	1.4%	2.2%	0.5%
8	3.2%	5.2%	1.1%
24	6.3%	9.6%	12.5%
48	17.6%	14.1%	2.7%
96	26.7%	19.76%	1.2%

Using a single CPU with N = 400 for 150 iterations, optimizations improved the performance of the 'fused' kernel by about 0.8 GFlops/s (1.24 total) and the 'unfused' kernel by about 1.42 GFlops/s (1.88 total) relative to the reference kernel (see table 1).

When using up to eight cores on bang, we see from table 1 that APF-Unfused achieves the highest performance values for all core counts,outperforming APF-Fused by about 51% on average, which in turn beats APF-Ref by almost 200% on average.

As we can see from table 2, these performance values come at the cost of increased performance overhead. This is most likely due to the fact that the constant communication time penalty incurred per request begins taking up a larger percentage of time relative to computation, and is consequently unavoidable.

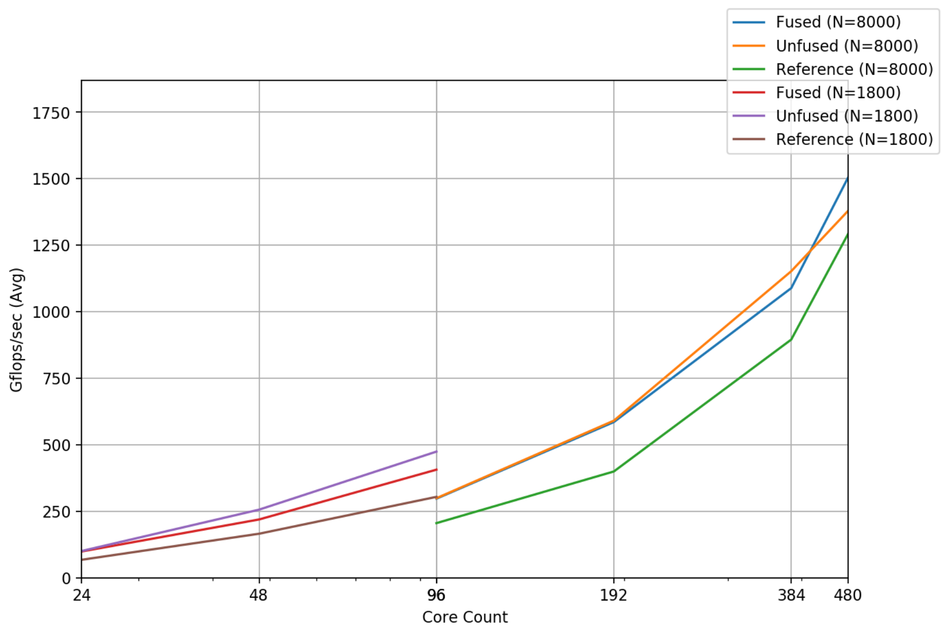

Figure 1: Scaling With Increasing Processors

As we can see from figure 1, on average, APF-Fused and APF-Unfused outperformed the reference implementation at every core count for both $N = 1800$ and $N = 8000$

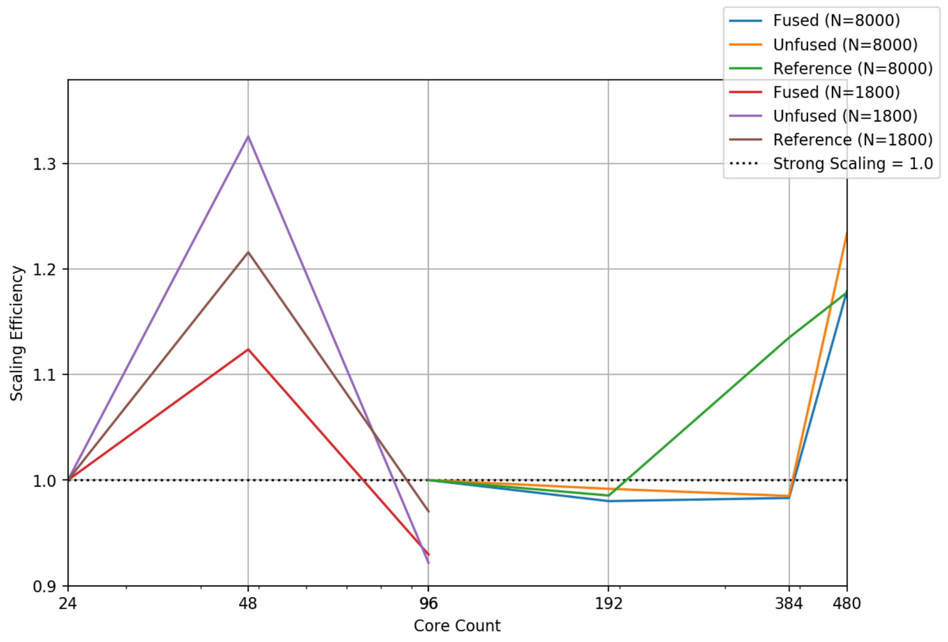

Figure 2: Scaling Strength For N = 1800 and N = 8000

Since geometries were tested with the intention of achieving the highest performance, to compute scaling efficiency, the best value for each core count was chosen as the reference value at that count. As we can see from figure 2, strong scaling was achieved for $N = 1800$ when going from 24 cores to 48 cores, for all implementations. Efficiency was highest for APF-Unfused, followed by APF-Ref, and then APF-Fused. Scaling dips slightly below the threshold for all implementations when going from 48 to 96 cores. For $N = 8000$, scaling efficiency when going from 96 to 192 cores is just under the threshold for all implementations, off by about 0.2. When increasing cores to 384, this continues for both APF-Fused and APF-Unfused while APF-Ref exceeds this threshold. Finally, increasing usage to 480 cores shows very strong scaling efficiency for all implementations.

Table 3: Top Five Processor Geometries For 96 Cores (m_pxn_p)

Rank	Fused	Unfused	Ref
1	12x8	48x2	96x1
2	96x1	24x4	48x2
3	6x16	96x1	24x4
4	48x2	6x16	12x8
5	8x12	12x8	16x6

From table 3 we see that the APF-Fused performed best when dimensions approached a square or column-vector. APF-Unfused tended to prefer more rectangular shapes that approached column-vectors. APF-Ref most consistently showed a preference having more rows than columns.

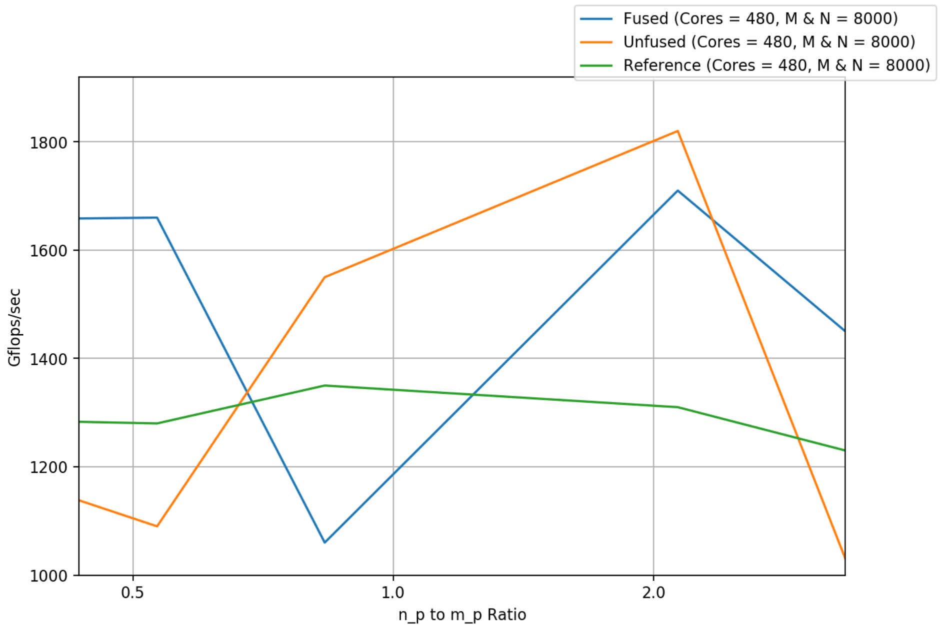

Figure 3: Gflops/s as a Function of n_p to m_p Ratio (480 Cores)

When using 480 cores, from figure 3 we see that the poorest performing geometries were those where n_p was between $\frac{1}{2}$ and 1 times m_p, and the best geometries were generally around the point where n_p was twice the amount of m_p.

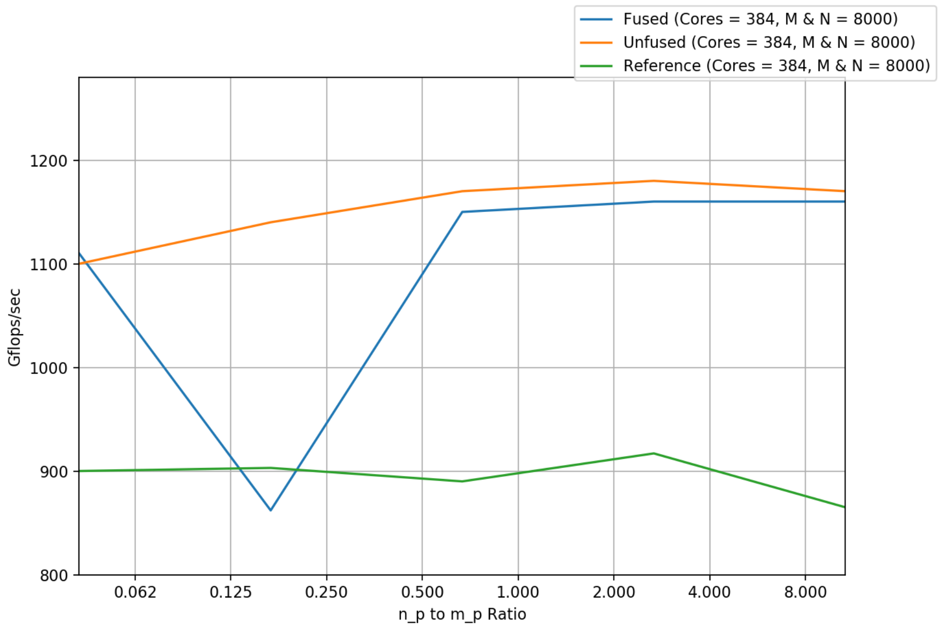

Figure 4: Gflops/s as a Function of n_p to m_p Ratio (386 Cores)

When using 384 cores, we see a trend towards more columns relative to rows producing better performance, similar to the one we observed when using 480 cores, though the difference in magnitude is not as severe (see figure 4).

3.1 Analysis

A few notable findings were produced by the experiments. First, though APF-Fused should theoretically make better use of caching behavior due to the combination of the inner loops, it performs worse relative to APF-Unfused a possible explanation for this phenomenon is that the compilers on our benchmarking machines (gcc 4.8.4. on bang, icc 14.0.2) were able to vectorize the inner loops of APF-Unfused better, for instance, through the use of AVX instructions, which may be especially efficient when calculating the ODE of E due to the spatial and temporal locality of operands.

Second, the increase in communication overhead for APF-Fused and APF-Unfused went up quite significantly with increasing core counts. Though some of this increase is unavoidable due to a greater amount of runtime being due to communication costs (as mentioned in section 3), some of this could possibly be ameliorated by interleaving more computation and communication. For instance, when the kernel is computing elements, when values needed by a neighboring processor have been computed, we could issue the send request for those cells at that time (i.e., for the top computational row). Additionally, we could issue a receive request for ghost cells only when we require those cells for computation (i.e., for the bottom computational row).

Next, the scaling efficiency achieved by APF-Fused and APF-Unfused was less than that achieved by APF-Ref. As both APF-Fused and APF-Unfused outperformed APF-Ref, a possible explanation for the reference implementation showing stronger scaling efficiency at certain intervals is the fact that it has not been optimized as well as the other implementations, so it gains more speedup from the greater parallelism inherent in increasing the core count. On the contrary, APF-Unfused or APF-Fused may be achieving much of their speedup via on-chip optimizations and are bottlenecked by the data transfer speed between processors. Since they already achieve such high performance, they may also have much less room to improve via increasing the number of processors used.

Lastly, when utilizing 384 and 480 cores, the optimal geometries for APF-Fused and APF-Unfused being characterized by fewer rows and more columns was a paradoxical finding. As ghost cell transfers between processors of different rows are less computationally intensive than those between processors of different columns due to the need to iterate over local send and receive buffers, it was hypothesized that geometries with fewer rows and more columns would produce performance penalties. This may be due to the fact that

3.2 Development Process

3.2.1 Division of the Computational Mesh

Originally, work was divided such that each processor was assigned $\lceil \frac{m}{m_p} \rceil$ rows and $\lceil \frac{n}{n_p} \rceil$ columns of the global computational mesh to ensure that all elements were included in subcomputations. However, I soon realized that this approach may leave a processor with up to $m\%m_p$ fewer computational rows and $n\%n_p$ fewer computational columns relative to other processors, since in the worst case a division may leave that many extra rows and columns left over. However, since there can be no more than $m\%m_p$ leftover rows and $n\%n_p$ leftover columns, quantities dictated by the number of processors in a row and in a column, respectively, I discovered that I could assign leftover rows and/or columns one at a time to satisfy the requirement that no processor computed more than one extra row or column than any other processor assigned to the job.

3.2.2 Sending Submeshes to Processors

A major roadbloack occurred when I was trying to code the sending of submeshes to processors from the rank 0 processor. Some processors were receiving incorrect submeshes. I had originally assumed my index calculations were broken or that I was incorrectly using the different variations of MPI's 'send' and 'recv' calls. I tried different combinations of these calls and different ways of filling the buffers, to no avail. I then discovered that the problem was rooted in the fact that I am writing back to the E_{t-1} and R matrices. When capturing submeshes sequentially, the rank 0 processor already has its own submesh and can immediately write its submesh to its own copies of E_{t-1} and R; copies that I would later use to determine the submeshes of other processors. In other words, this was overwriting values in other computational windows that I had not yet transmitted. I remedied issue this by having the rank 0 processor write to copy its submesh at the end of the data distribution phase, rather than the beginning.

3.2.3 Inter-Process Communication During The Computation Phase

To send and receive ghost cells from neighbors, I utilized calls to 'MPI_ISend' and 'MPI_IRecv', respectively. Initially, these calls would hang. I later discovered that this was due to the fact that I was using the incorrect tags when sending messages to neighbors. I had erroneously assumed that the 'sendToLeftNeighbor' and 'recvFromLeftNeighbor'tag could be combined, not realizing that the latter tag must instead match with that processor's 'sendToRightNeighbor' tag. Since each processor had at most one send and one receive call to any other processor, I remedied this issue by using the same tag for all exchanges. Given the fact that I used asynchronous communication, I also moved the wait calls for sends and receives to separate blocks, away from their respective requests, so that all send and receive calls could complete prior to the processor having to wait for results.

3.2.4 Kernel Optimizations

Next, I attempted to optimize the kernels. As outlined in section 2.4.1, I did this first through the exploitation of temporal locality, moving oft-used values into local variables, and then through the exploitation of spatial locality by having pointers to the top and bottom rows of a cell and moving contiguous elements into local variables (so consequent iterations would already have the value in fast memory). I tried to tune this further (e.g. through the use of a local variables to hold even more interim values), but this resulted in negative performance returns, presumably due to register spilling and/or the compiler being unable to identify optimization opportunities it had previously exploited.

3.2.5 Ensuring Correct L2 Norm Calculation

Finally, I conducted the experiments mentioned in 2.5. I was not aware that the reference implementation may not calculate the L2 Norm correctly (e.g. when N = 1800) which led me to scrutinize my code for a lengthy period of time to determine the location of the bug. Once I was informed that L2 norms should be compared to those computed via the provided single-threaded code, I realized that my programs produced the correct results and there was no bug to begin with.

3.2.6 Intel Compiler Vectorizations

The Intel compiler automatically attempts to vectorize given the default optimization flag used in our program ('-O3'). Specifying a vector reporting level of 5 reports the details of its automatic vectorization analysis, including assumed dependencies (i.e. flow, anti, and output dependence) between code segments in loops. This feedback offers programmers a chance to further optimize code by structuring it according to this feedback so that the autovectorizer can identify more optimization targets (i.e. by using pragmas). I did not attempt any further code optimization based on this feedback.

4 Future Work

Given more time, I would have worked with the Intel compiler autovectorization tools a little more to see if I could improve performance further. In general, I would plan to explore the use of other software tools to more easily discover optimization targets and fine tune performance for specific systems.

References

Rubin R Aliev and Alexander V Panfilov. A simple two-variable model of cardiac excitation. *Chaos, Solitons & Fractals*, 7(3):293–301, 1996.

CPU World. Intel xeon e5345 specifications, 2017a. URL `http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5345%20-%20HH80563QJ0538M%20(BX80563E5345A%20-%20BX80563E5345P).html`.

San Diego Supercomputer Center. Comet user guide: Technical summary, 2017. URL `http://www.sdsc.edu/support/user_guides/comet.html#tech_summary`.

CPU World. Intel xeon e5-2680 v3 specifications, 2017b. URL `http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2680%20v3.html`.