# Assignment 1 COMP4300

Abhaas Goyal (u7145384)

June 4, 2021

Table 1: Table abbreviations

| | |
|---|---|
| $\mathbf{A_T}$ | Advection Time |
| $\mathbf{N_{Send}}$ | Blocking Send |
| $\mathbf{I_{Send}}$ | Non-Blocking Send |
| $\mathbf{G_F}$ | Gigaflops per second |
| $\mathbf{P_C}$ | Per core (with $G_F$) |
| **NP** | Number of processes |
| $\mathbf{C_T}$ | Calculated time |

## 1 Parallelization via 1D Decomposition and Simple Directives

### 1.1 Updation policies

1. `omp1dUpdateAdvectField()` - do via the metrics described below
2. `omp1dCopyField()` - same policy as 1
3. `omp1dBoundaryRegions()` - parallelize only top and bottom halo and not left and right exchange (cuz cache)

### 1.2 Metrics

- To maximize cache misses statically scheduling to 1
- **Maximize performance** - `(a)` Parallelize the outer loop gives us the b

```
#pragma omp parallel for default(shared) private(j)
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++)
      // Update V(v,i,j)
  }
```

- **Maximize parallel region entry/exits** `(b)` Here, we switch the variables i and j, so in each iteration of j, the threads would parallelize in the for loop ==> In each iteration, it would be getting around M/t

```
  for (j = 0; j < N; j++) {
    #pragma omp parallel for default(shared)
    for (i = 0; i < M; i++)
```

```
        // Update V(v,i,j)
    }
```

- This could work if M » N (to be tested)

- **Maximize cache misses involving coherent reads** (c) For parts, here's a visualization. Assume no of threads are 4. Then we divide the threads in static blocks of size i, like- for every iteration, a new cache block has to be loaded considering the size of each thread's row

```
t   | 1 2 3 4 |
----+----------|
i   |          |
0   | *        |
1   |   *      |
2   |     *    |
3   |       *  |
4   | *        |
5   |   *      |
6   |     *    |
7   |       *  |
.   | *        |
.   |          |
n   |       *  |
    |----------|
```

(inspired from [1])

```
#pragma omp parallel for default(shared) private(j) schedule(static,1)
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++)
      // Update V(v,i,j)
  }
```

- **Maximize cache misses involving coherent writes** (d)

```
j |  0 1 2 3 4 5 6 7 .   .    .   .   .   n
--+-------------------------------------------
t |
1 |  *       *       *       *       *
2 |    *       *       *       *       *
3 |      *       *       *       *       *
4 |        *       *       *       *       *
```

```
#pragma omp parallel for default(shared) private(i) schedule(static,1)
for (j = 0; j < N; j++) {
  for (i = 0; i < M; i++)
    // Update V(v,i,j)
}
```

- j is parallelized so in each iteration while reading it's chill cuz particular cache line blocks loading for them but they access data out of on another - false sharing

### 1.3 Testing Methodolgy

- Performance model was tested in one node
- The parameters were chosen to distribute all the nodes equally so M and N were both divisible by 48. The focus was on computation aspect so suitably large value of M and N were chosen. So for testing purposes I chose `M=N=2160` and remain unchanged since strong scaling was needed.
- Number of reps was taken to be 100 (sufficiently large)

### 1.4 Results

```
M = 2160 N = 2160 reps = 100
```

Table 2: Strong scaling with different number of threads on single node on diff metrics

| **Metric** | $\mathbf{N_T}$ | 1 | 3 | 6 | 12 | 24 | 48 |
|---|---|---|---|---|---|---|---|
| (a) | $\mathbf{A_T}$ | 1.93e+00s | 7.20e-01s | 4.32e-01s | 3.34e-01s | 4.95e-01s | 2.52e+00s |
| | $\mathbf{G_F}$ | 4.83e+00 | 1.30e+01 | 2.16e+01 | 2.79e+01 | 1.89e+01 | 3.71e+00 |
| | $\mathbf{P_C}$ | 4.83e+00 | 4.32e+00 | 3.60e+00 | 2.33e+00 | 7.86e-01 | 7.72e-02 |
| | **Speedup** | 1 | 2.68 | 4.66 | 5.77 | 3.89 | 0.76 |
| (b) | $\mathbf{A_T}$ | 5.84e+00s | 1.86e+00s | 1.14e+00s | 1.02e+00s | 3.57e+01s | 4.73e+00s |
| | $\mathbf{G_F}$ | 1.60e+00 | 5.02e+00 | 8.18e+00 | 9.19e+00 | 2.61e-01 | 1.97e+00 |
| | $\mathbf{P_C}$ | 1.60e+00 | 1.67e+00 | 1.36e+00 | 7.66e-01 | 1.09e-02 | 4.11e-02 |
| | **Speedup** | 1 | 3.13 | 5.12 | 5.72 | 1.63 | 1.2 |
| (c) | $\mathbf{A_T}$ | 1.93e+00s | 7.41e-01s | 4.58e-01s | 3.99e-01s | 5.30e-01s | 2.72e+00s |
| | $\mathbf{G_F}$ | 4.84e+00 | 1.26e+01 | 2.04e+01 | 2.34e+01 | 1.76e+01 | 3.43e+00 |
| | $\mathbf{P_C}$ | 4.84e+00 | 4.20e+00 | 3.39e+00 | 1.95e+00 | 7.34e-01 | 7.14e-02 |
| | **Speedup** | 1 | 2.60 | 4.21 | 5.77 | 3.64 | 0.74 |
| (d) | $\mathbf{A_T}$ | 5.78e+00s | 3.38e+00s | 1.62e+00s | 1.25e+00s | 3.61e+01s | 0 |
| | $\mathbf{G_F}$ | 1.62e+00 | 2.76e+00 | 5.76e+00 | 7.46e+00 | 2.59e-01 | 0 |
| | $\mathbf{P_C}$ | 1.62e+00 | 9.20e-01 | 9.60e-01 | 6.22e-01 | 1.08e-02 | 0 |
| | **Speedup** | 1 | 1.71 | 3.56 | 4.62 | 0.16 | 0 |

- 0 since timed out

## 2 Performance Modelling of Shared Memory Programs

### 2.1 TODO

## 3 Parallelization via 2D Decomposition and an Extended Parallel Region

### 3.1 Results

- `M = N = 2160 (2 * L_3 cache has around 70 MB memory) OMP_NUM_THREADS=12`
- A significant improvement of 37% was found when `P=12 Q=1`
- Distributing in column fashion after threads have been initialized gives us the maximum speedup. This would be because each thread gets maximum square like
- Here, speedup is taken w.r.t Q = 1 unlike in previous table

Table 3: Computation for 2D process grids (1 Node) with Q >= 1

| **P** | **Q** | $\mathbf{A_T}$ | $\mathbf{G_F}$ | $\mathbf{P_C}$ | **Speedup** |
|---|---|---|---|---|---|
| 1 | 12 | 4.48e-01 | 2.08e+01 | 1.74e+00 | 1 |
| 2 | 6 | 3.75e-01 | 2.49e+01 | 2.07e+00 | 1.19 |
| 3 | 4 | 3.57e-02 | 2.62e+01 | 2.18e+00 | 1.25 |
| 4 | 3 | 3.48e-01 | 2.68e+01 | 2.24e+00 | 1.28 |
| 6 | 2 | 3.40e-01 | 1.74e+01 | 2.28e+00 | 1.31 |
| 12 | 1 | 3.27e-01 | 2.86e+01 | 2.38e+00 | 1.37 |

## 4 Further Optimizations OpenMP

- One of the main optimizations was that The copying overhead is removed for larger number of iterations.
- Improved performance by a lot

### 4.1 Results

- On P = Q = 2160 with division of 12,1 speedup of around 50%

| **M** | **N** | $\mathbf{A_T}$ | $\mathbf{G_F}$ | $\mathbf{P_C}$ | $\mathbf{A_T}$ **(-o)** | $\mathbf{G_F}$ **(-o)** | $\mathbf{P_C}$ **(-o)** | **Speedup** |
|---|---|---|---|---|---|---|---|---|
| 1080 | 1080 | 4.88e-02 | 4.78e+01 | 3.98e+00 | 5.34e-02 | 4.37e+01 | 3.64e+00 | 0.91 |
| 2160 | 2160 | 3.29e-01 | 2.84e+01 | 2.36e+00 | 1.86e-01 | 5.03e+01 | 4.19e+00 | 1.77 |
| 4320 | 4320 | 1.59e+00 | 2.35e+01 | 1.96e+00 | 8.08e-01 | 4.62e+01 | 3.85e+00 | 1.63 |

- We see that there are less number of reads/writes at different cores are not being done (so per core performance remains the same)

## 5 Baseline GPU Implementation

- Taking large parameters `M=N=4096` (divisible by 32)
- block size as 32,32 and grid size increasing
- Then change block size
- reps are taken as 10 to not slow down the tests for small values of Gx and Gy (at the same time also hiding the time to copy back and forth from memory)

### 5.1 Results

`M = 4096 N = 4096 reps = 10`
Comparing with you CPU and serial at this speed

| $\mathbf{I_T}$ | $\mathbf{A_T}$ | $\mathbf{G_F}$ | **Speedup** |
|---|---|---|---|
| -s | 5.05e+01s | 6.64e-02 | 760.54 |
| -h | 1.09e+00s | 3.09e+00 | 28.47 |

Go fast brr

- 1.63e-03s (M=1 N=1)

Table 4: Performance comparision between various values of (Gx, Gy) and (Bx, By)

| **Gx** | **Gy** | **Bx** | **By** | $\mathbf{A_T}$ | $\mathbf{G_F}$ | **Speedup** |
|---|---|---|---|---|---|---|
| 1 | 1 | 16 | 64 | 1.46e+00s | 2.30e+00 | 1.02 |
| 1 | 1 | 32 | 32 | 1.50e+00s | 2.24e+00 | 1 (ref.) |
| 1 | 1 | 64 | 16 | 1.55e+00s | 2.17e+00 | 0.96 |
| 2 | 2 | 16 | 64 | 3.97e-01s | 8.46e+00 | 3.77 |
| 2 | 2 | 32 | 32 | 4.00e-01s | 8.39e+00 | 3.74 |
| 2 | 2 | 64 | 16 | 4.36e-01s | 7.70e+00 | 3.43 |
| 4 | 4 | 16 | 64 | 2.14e-01s | 1.57e+01 | 7 |
| 4 | 4 | 32 | 32 | 2.80e-01s | 1.20e+01 | 5.35 |
| 4 | 4 | 64 | 16 | 2.87e-01s | 1.17e+01 | 5.22 |
| 8 | 8 | 8 | 128 | 9.28e-02s | 3.62e+01 | 16.16 |
| 8 | 8 | 16 | 64 | 2.90e-01s | 1.16e+01 | 5.17 |
| 8 | 8 | 32 | 32 | 5.80e-01s | 5.78e+00 | 2.58 |
| 8 | 8 | 64 | 16 | 6.39e-01s | 5.25e+00 | 2.34 |
| 16 | 16 | 4 | 256 | 2.67e-02s | 1.26e+01 | 5.625 |
| 16 | 16 | 8 | 128 | 4.04e-02s | 8.31e+01 | 37.09 |
| 16 | 16 | 16 | 64 | 1.48e-01s | 2.27e+01 | 10.13 |
| 16 | 16 | 32 | 32 | 4.99e-01s | 6.73e+00 | 3.00 |
| 16 | 16 | 64 | 16 | 6.26e-01s | 5.36e+00 | 2.39 |
| 16 | 16 | 128 | 8 | 5.88e-01s | 5.71e+00 | 2.54 |
| 16 | 16 | 256 | 4 | 4.11e-01s | 8.16e+00 | 3.64 |
| 32 | 32 | 1 | 1024 | 1.72e+00s | 1.95e+00 | 0.87 |
| 32 | 32 | 2 | 512 | 9.70e-01s | 3.46e+00 | 1.54 |
| 32 | 32 | 4 | 256 | 5.15e-01s | 6.52e+00 | 2.91 |
| 32 | 32 | 8 | 128 | 3.81e-02s | 8.80e+01 | 39.28 |
| 32 | 32 | 16 | 64 | 5.35e-02s | 6.27e+01 | 2.79 |
| 32 | 32 | 32 | 32 | 2.14e-01s | 1.57e+01 | 7.00 |
| 32 | 32 | 64 | 16 | 3.81e-01s | 8.81e+00 | 3.93 |
| 32 | 32 | 128 | 8 | 3.69e-01s | 9.08e+00 | 4.05 |
| 32 | 32 | 256 | 4 | 7.33e-01s | 4.58e+00 | 2.04 |
| 32 | 32 | 512 | 2 | 9.78e-01s | 3.43e+00 | 1.53 |
| 32 | 32 | 1024 | 1 | 1.59e+00s | 2.12e+00 | 0.94 |
| 64 | 64 | 16 | 64 | 4.72e-02s | 7.11e+01 | 31.74 |
| 64 | 64 | 32 | 32 | 3.84e-02s | 8.75e+01 | 39.06 |
| 64 | 64 | 64 | 16 | 4.83e-02s | 6.95e+01 | 31.02 |

## 6 Optimized GPU Implementation

- Remove extra copies
- shared memory mappings

### 6.1 Implementation

- `__shared__` and `copyAdvectField` in repetitions removed

### 6.2 Results

`M = N = 2048 reps = 100`

Table 5: Performance for various lengths of width for optimized division of 2D grids in a GPU

| **M** | **N** | $\mathbf{A_T}$ | $\mathbf{G_F}$ | $\mathbf{A_T}$ **(-o)** | $\mathbf{G_F}$ | **Speedup** |
|---|---|---|---|---|---|---|
| 2048 | 2048 | 1.21e-02s | 6.91e+01 | 1.72e-02 | 4.87e+01 | 0.7 |
| 4096 | 4096 | 3.81e-02s | 8.80e+01 | 4.16e-02 | 8.06e+02 | 0.91 |
| 8192 | 8192 | 1.41e-01s | 9.54e+01 | 1.34e-01 | 1.00e+02 | 1.05 |

- This difference would be propounded by a huge factor if repetitions are large (since copying part is only done in GPU memory anymore) tests are biased towards updateAdvectField
- When number of iterations are high, then omg
- Used `nvprof` to check improvement - pretty high I would say

## 7 Comparison of the Programming Models

- CUDA's optimized version was the hardest to do
- MPI's distributed system was second hardest
- OpenMP was easiest but I guess it didn't give good results
- Focus on boundaryUpdate in Assignment 1 but focus on 9 point stencil update in assignment 2

## 8 CUDA Results on Gadi GPU's

- 4x Tesla V100 let's roll vroom

### 8.1 Results

`M = N = reps = 100 -o on Tesla V100-SXM2-32GB (omg)`

Table 6: Performance for optimized division of 2D grids in Tesla

| **M** | **N** | $\mathbf{A_T}$ | $\mathbf{G_F}$ | **Speedup wrt RTX2080** |
|---|---|---|---|---|
| 2048 | 2048 | 7.52e-03 | 1.12e+02 | 1.62 |
| 4096 | 4096 | 2.21e-02 | 1.52e+02 | 1.72 |
| 8192 | 8192 | 8.32e-02 | 1.61e+02 | 1.68 |

Around 70% improvement on a single GPU of Tesla.

## 9 References

[1] `http://jakascorner.com/blog/2016/06/omp-for-scheduling.html`

## 10 Acknowledgements

- Done individually and with help from Piazza/COMP4300 Practicals.