Optimization of a Shallow Water Simulation

Ravi Patel (rgp62), Saurabh Netravalkar (sn575), Greg Granito (gdg38)

Profiling Results

Original Code: The amplxe profiler was used with hotspots to observe the running times of different functions. The functions which take longer to run and are a potential bottleneck are:

1. limit\_derivs (CPU time: 1.155s)
2. compute\_step (CPU time: 0.5688)
3. compute\_fg\_speeds (CPU time: 0.191)

Naïve Parallelization: Loop unrolling was performed and OpenMP parallel for pragmas were naively added in front of the looping constructs of each of the three functions.

Code Snippet:

Before:

for (int iy = 1; iy < ny\_all-1; ++iy)

for (int ix = 1; ix < nx\_all-1; ++ix) {

After:

#pragma omp parallel for

for(int n = 0; n < (ny\_all - 2)\*(nx\_all - 2); ++n) {

int iy = n / (nx\_all - 2);

int ix = n - iy \* (nx\_all - 2);

++iy;

++ix;

Observations:

1. The parallel code runs slower for the 2002 cell problem than the serial un-tuned code. The reason is the inherent read after write and write after read dependencies among the loop bodies. Hence, the waiting time for each thread combined with the added overhead of communication costs blows up the time of execution.
2. The run time for loop unrolling alone for the bottlenecks above were found to be higher:
   1. limit\_derivs (CPU time: 1.480s)
   2. compute\_step (CPU time: 0.717s)
   3. compute\_fg\_speeds (CPU time: 0.397s)
3. The run time for loop unrolling and naïve parallelization (24 threads) in the above functions were also found to be higher. About 20s of additional time was spent on overhead for amplxe.
   1. limit\_derivs (CPU time: 1.925s)
   2. compute\_step (CPU time: 1.315s)
   3. compute\_fg\_speeds (CPU time: 0.703s)
4. Weak and strong scaling analysis was performed on the naïve parallelized code:

Strong Scaling (for 4002 threads)

|  |  |
| --- | --- |
| Problem size (cells) | Total time |
| 1 | 19.3s |
| 8 | 6.2s |
| 24 | 7.4s |

Weak Scaling (for 2002 cells/core)

|  |  |
| --- | --- |
| Threads | Total time |
| 1 | 2.5s |
| 4 | 7.5s |
| 16 | 62.5s |

* 1. From strong scaling, performance improved using 8 cores over 1 (39% efficiency), but 24 cores results in worse performance.
  2. From weak scaling, this code shows 33% efficiency using 4 cores over 1, but extremely poor efficiency using 16 cores.
  3. Naïve parallelization results in some improvements for low thread counts, but scales poorly for higher thread counts.

Future optimizations:

Domain decomposition: We are currently implementing the strategy below for domain decomposition. We expect to obtain the largest gains with this.

1. Remove data dependencies by tiling the grid and padding the tiled grids with ghost cells as well as the necessary cell layers of adjacent tiles.
2. For time ts in blocks of a step:

For Each Processor pi, operate in parallel on a block bi of the board:

Copy into local storage board of each processor along with ghost cell info

Advance by making the processors compute some steps independently

Copy out to a new board

Barrier synchronization

Swap the board pointers, i.e. new board becomes the old board and vice-versa

Serial Optimizations:

* Sizing blocks to fit in cache considering the sharing into account
* Align memory for the blocks
* Compute wave speeds every few steps and choose dt more conservatively