Optimization of a Shallow Water Simulation

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

Profiling

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.

Observations

1. The parallel code runs slower for the 200^2 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:
    a. limit_derivs (CPU time: 1.480s)
    b. compute_step (CPU time: 0.717s)
    c. 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.
    a. limit_derivs (CPU time: 1.925s)
    b. compute_step (CPU time: 1.315s)
    c. compute_fg_speeds (CPU time: 0.703s)

4. Weak and strong scaling analysis was performed on the naïve parallelized code:

Strong Scaling (for 400^2 threads)

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

Weak Scaling (for 200^2 cells/core)

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

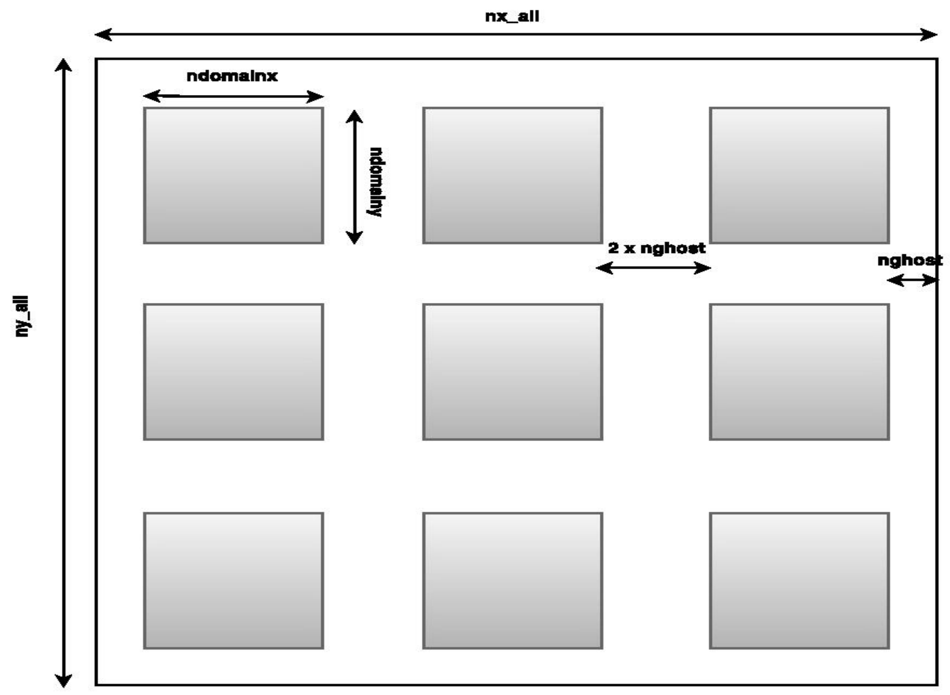
a. From strong scaling, performance improved using 8 cores over 1 (39% efficiency), but 24 cores result in worse performance.
b. From weak scaling, this code shows 33% efficiency using 4 cores over 1, but extremely poor efficiency using 16 cores.
c. Naïve parallelization results in some improvements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

High Level Strategy for using Domain decomposition

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 t_s in blocks of a step:
    - 2a. For Each Processor p_i, operate in parallel on a block b_i of the board:
        - i. Copy into local board of each processor along with ghost cell info
        - ii. Advance by making the processors compute some steps independently
        - iii. Copy out to a new board
        - iv. Barrier synchronization
    - 2b. Swap the board pointers, i.e. new board becomes the old board and vice-versa

Further Potential Optimizations

1. Sizing blocks to fit in cache considering the sharing into account
2. Align memory access for the blocks
3. Compute wave speeds every few steps and choose 'dt', the time step more conservatively



Board Padded with Ghost Cells

Parallelization

Workflow: The workflow as implemented, for parallel execution of the shallow water wave simulation using domain decomposition is as follows:

1. The original board is divided into square tiles/blocks
2. Each block is padded on all 4 sides with layers of ghost cells as shown in the figure.
3. Each processor is given its own tile padded with ghost cells and works in parallel by time stepping on its own tile.
4. A single processor initially performs the ghost cell updates for the entire board, i.e. updating ghost cells values by copying out rows from left and right and then top and bottom, by wrapping around the board boundaries. Now, the processors are ready to compute their own time step advances in parallel.
5. Each processor also computes the local wave speeds 'cx' and 'cy' within their own tile and write the values to a shared array, indexing the array by their own unique thread identifiers which range from 0 to number of threads - 1.
6. At the end of 'n' time steps, ('n' = number of layers of ghost cells – 3, since a minimum of 3 ghost cells are required to compute one entire time step in parallel without requiring data from adjacent tiles), the processors perform a barrier sync.
7. A single processor traverses through the array of local tile wave speeds and computes the maximum speed among them in both x and y directions, and uses this value to perform the time stepping increment.
8. Once the time steps are completed, the true data values are copied back from the ghost cell padded board into the original board.

Further Experiments

Rectangular Tiling: We also extended the code to be able to work with rectangular tiles and experimented with it, but found that it did not behave as expected. The boundary conditions and ghost cell updates typically change when we have rectangular tiles and it creates complexities and produces strange locally tile wise shifted wave simulation results.

Blocking at the Driver Level: We also created an entire implementation of domain decomposition where we divided our original n x n grid into four n/2 x n/2 tiles at the driver level. We then padded layers of ghost cells, and then instantiated these 4 tiles by their own Physics class and allowed these 4 simulations to run in parallel by doing a barrier synchronization between time steps, in order to communicate between processors and to update ghost cell values.

References

https://github.com/amirajdhawan/water : Held meetings with him to understand their group's approach, brainstormed and reasoned about potential further optimizations.