CS 5220 Introduction to Parallel Programming | Fall 2015
Kenneth Lim (kl545), Yalcin Ozhabes (ao294), Joshua Cohen (jbc264) | Project 2

1 Summary

Our final report centers around the constructs that resulted in significant speed improvements. The structure of the original codebase made parallelism difficult for the following reasons:

1. The evaluation window for each timestep is performed over the entire simulation grid. This does not scale well with the input size, nor does it allow us to exploit the hardware threads provided by our architecture.
2. The optimal "locations" in the code for parallelism (e.g. `Central2D::limdiff` or `Central2D::compute_step`) incurs a large initialization overhead because the outermost for-loop (which is not visible from the function context alone) creates and tears down the thread pool on every iteration.
3. The optimal location to initialize a thread pool (i.e. the parent context provided by `Central2D::run`) is not amenable for doing so because the child functions in the loop are called serially, and resource contention inevitably occurs when checking or updating the termination condition for the while-loop.

We solve (1) by partitioning the simulation grid into smaller blocks capable of indepdendent computation — a domain decomposition approach. Detailed benchmarking of (2) finds that it still benefits most from vectorization, and we reuse the dependency annotations described in our mid-term report. For (3), we set up a producer-consumer model that dispatches block computations (viz. domain decomposition) and accumulates results via a "control" thread.

2 Domain Decomposition

The problem input is subdivided into $n \times n$ square blocks stored in `BlockedSimulation`, where n is to be determined later. We also define `SimBlock`, which inherits from `Sim`, but performs the additional step of copying some radius g of ghost cells needed for computation lookahead from the neighboring cells around each block (expanding outwards with respect to g). This replaces the functionality previously provided by `Central2D::apply_periodic`. More importantly, the ghost cells in each block allows the functions responsible for advancing the time step and computing the propagation speeds to be evaluated independently for each block without loss of generality.

The functions responsible for checking the solution and initializing the problem are modified accordingly to support and encourage domain decomposition using the constructs described above. `BlockedSimulation::run` now maps over each block in the grid, creating an opportunity to execute each iteration of the loop in parallel. The nested for-loops provide some intuition for the selection of n, because a best-case scenario will use n^2 threads — one for each block. Conversely, and more importantly, given some number of available threads t, we can propose that:

$$\hat{n} = \lfloor \sqrt{t} \rfloor$$

where $\hat{n}$ is the optimal value for n, and t for an OpenMP system can be obtained by calling `omp_get_max_threads`. We discuss the specifics of parallelism in the eponymous section later. Note, however, that determining the maximum allowed time step is still dependent on the most recent global state of the simulation, and has to be evaluated serially.

```
for (int i = 0; i < nblocks; i++) {
  for (int j = 0; j < nblocks; j++) {
    blocks[i][j].limited_derivs();
    blocks[i][j].compute_step(0, dt);

    blocks[i][j].compute_fg_speeds(cx[i * nblocks + j],
                                   cy[i * nblocks + j]);

    blocks[i][j].limited_derivs();
    blocks[i][j].compute_step(1, dt);
  }
}
```

Figure 1: Evaluation of two time steps within the same loop.

There is one last opportunity for optimization: we are given that in-between any two time steps, the dependency of any value on the grid is bounded within its neighborhood. Specifically, it is only affected by points that are 3 units away. Thus, instead of naively using only the *immediate* ghost cells around the current block, we can set $g = 3$ to allow us to take two time steps within a single loop (see Figure 1) iteration before we have to invoke `BlockedSimulation::copy_ghosts` to re-copy the new values back in from the neighboring block.

3 Vectorization

A large part of vectorization for `Central2D` was implemented and described in our mid-term report, and the benefits obtained persist even for our current best attempt. While implementing domain decomposition introduces a large number of nested for-loops used for copying ghost cells, there is little performance improvement gained from vectorization because we are copying *across* strides[1]. In most cases, the speedup from forcing the use of intrinsics is lost because the compiler has to perform a large number of broadcast operations.

We have incorporated advice received from the mid-term peer review and experimented with explicit specification of the `restrict` attribute for `du` in `Central2D::limdiff`, which is the bottleneck function. However, we did not observe any noticeable change in performance.

4 Parallelism with OpenMP

In our mid-term report, we noted that adding `#pragma omp parallel for` annotations to loops in `Central2D` did not yield the improvements expected from running code in parallel. We also mooted the idea of a "master" thread of execution responsible for maintaining the threadpool, and dispatching parallelized for-loops within the master thread (see Figure 2). There are a number of problems with this approach:

1. Running `#pragma omp parallel for` *within* the while loop results in a new thread pool being initialized on every iteration of the loop. This causes an approximate two to three order of magnitude increase in wall-clock time incurred by initialization overheads.
2. The large critical section and presence of multiple atomic operations results in high thread contention, causing a thread to spend approximately 20% of its time contending for a lock in that section.
3. The `#pragma omp master` annotation prohibits spawning threaded processes within any encapsulated code, because it violates the invariant that only one thread context should be made available.
4. Exiting a while loop within a parallel execution context causes indeterminate behavior, even with mutual exclusion constructs in place.

After repeated investigation, we concluded that the following constraints **had** to be enforced when applying parallelism constructs:

[1]The authors are reminded, grudgingly, of `matmul-`, in which we struggled to perform the due diligence for the sole task of copying out blocks.

```
#pragma omp master
{
  bool done = false;
  real t = 0;

  while (!done) {
    real dt;
    for (int io = 0; io < 2; ++io) {
      #pragma omp parallel for schedule(auto)
      {
        ...

        #pragma omp critical
        if (io == 0) {
          dt = cfl / std::max(cx/dx, cy/dy);
          ...
        }

        ...

        #pragma omp atomic
        t += dt;
      }
    }
  }
}
```

Figure 2: Proposed parallelism schema from our mid-term report.

1. The thread pool created by `#pragma omp parallel` should not be within any kind of loop structure, and should be done as early as possible to give the threads time to spin up.

2. The while-loop *must* run in a single thread.

3. Within the while-loop, it is important to use barriers to guard computations which accumulate modifications over a sliding window. For example, while it is not necessary that `Central2D::compute_fg_speeds` is executed in loop index order, we want *all* such computations to complete before moving on to the next step. This requires either an implicit barrier provided by a `#pragma omp for` environment, or an explicit OpenMP barrier.

4. Loop termination variables must be globalized, either by placing them outside the parallel execution environment, or by using the `shared` keyword. The documentation leads us to believe that both approaches result in identical outcomes.

Unfortunately, (2) makes it difficult to actually parallelize any for-loop naively. Thankfully, our use of domain decomposition allowed us to approach the problem from a slightly different perspective. Firstly, the batching of time steps described in the previous sections unrolls the for-loop on Line 8 of Figure 2. Our remaining for-loops serve the purpose of iterating over the blocks in the simulation grid. If we consider the master thread which contains the while-loop to be a producer, and a thread pool to be the consumer, we can safely spin up OpenMP tasks using the for-loops, and guard the end of each for-loop block with a task-specific barrier provided by `#pragma omp taskwait`. This gives us the implementation described in Figure 3.

When initializing the thread pool, we clamp the number of threads to be no more than the expected number of tasks spawned by the nested for-loops ($\hat{n}$), plus one more to account for the master thread. Thus a more precise definition of $\hat{n}$ would be $\hat{n} = \lfloor \sqrt{t-1} \rfloor$. In theory, this is not necessary, because we could have used the loop indices to assign one set of tasks to run on the master thread. However, this design decision was made in order to ensure that the critical section equivalent we have created (by guarding sections of the code with explicit barriers) does not get delayed because a thread is in-between tasks. The results of this implementation are discussed in the section on scaling studies.

```c
bool done = false;
...

#pragma omp parallel num_threads(nblocks * nblocks + 1)
{
  #pragma omp master
  {
    while (!done) {
      for (int i = 0; i < nblocks; i++) {
        for (int j = 0; j < nblocks; j++) {
          #pragma omp task
          // Copy ghost cells and calculate speeds
        }
      }

      #pragma omp taskwait

      // Check loop termination conditions
      ...

      for (int i = 0; i < nblocks; i++) {
        for (int j = 0; j < nblocks; j++) {
          #pragma omp task
          // Batch run two time steps
        }
      }

      #pragma omp taskwait
      ...
    }
  }
}
```

Figure 3: Improved parallelism applied to the domain decomposition approach.

5 MIC Experiments

Our attempts at offloading computation to the coprocessors was flaky at best. We believe that there are two possible approaches:

1. Offload the computation associated with an entire frame
2. Keep the wrappers for the frame computation on the main processor, but offload expensive sections.

(1) is a more appealing solution, both in terms of implementation, and potential gains. The Intel Xeon Phi coprocessors work best with highly parallelized code, and provide 240 threads (4 hardware threads per core, for 60 cores, compared to a measly 24 threads on the main processor) per board. Distributing frames over the 2 coprocessors in an alternating manner sounded like an excellent idea.

The challenge with (1) was that C++ vectors are not bitwise copyable, which is a requirement for efficient data transfer to the coprocessors. We experimented with a "brute-force" copy method, but found that even at moderately large input sizes, the cost incurred from copying everything to and from the coprocessor outweighs the performance gains.

With (2), our approach was to try offloading bottleneck portions of the code to the coprocessor in an attempt to ease the load on the main processor. Unfortunately, offloading from within a parallel thread is challenging because one has to correctly and precisely target *all* functions and variables for compilation on the coprocessor architecture. A carpet bomb approach exists in the form of the compiler flag `-offload-target-attribute=mic`, which targets every file for coprocessor compilation, but this inevitably brings us back to the problem described in the previous paragraph. We speculate that even if we were able to successfully implement this approach, we would not get significant speedups by virtue of the fact that we are not fully saturating the coprocessor or exploiting any kind of cache locality.

6 Scaling Studies

Figure 4 and Figure 5 show the results of strong and weak scaling studies respectively for our best implementation, which implements good vectorization and parallelism optimizations. The results for the strong scaling study are somewhat interesting — while one expects a clean, exponential decrease in wall-clock time, the computation flattens out at 4 threads, 9 threads, and then again at 16 threads. The reason for this is because we have artificially clamped the number of threads in the thread pool to be a function of the number of blocks used in the computation. From 4 to 8 available threads, `BlockedSimulation` partitions 4 and only 4 blocks (2×2). From 9 to 15 threads, we

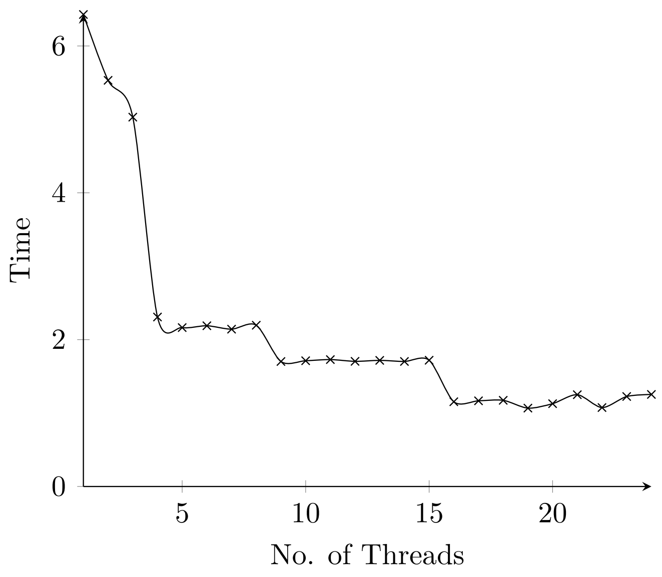

Figure 4: Strong scaling study for a wave-on-a-river simulation with 1000 cells per side and 100 frames. The time reported is wall-clock time.

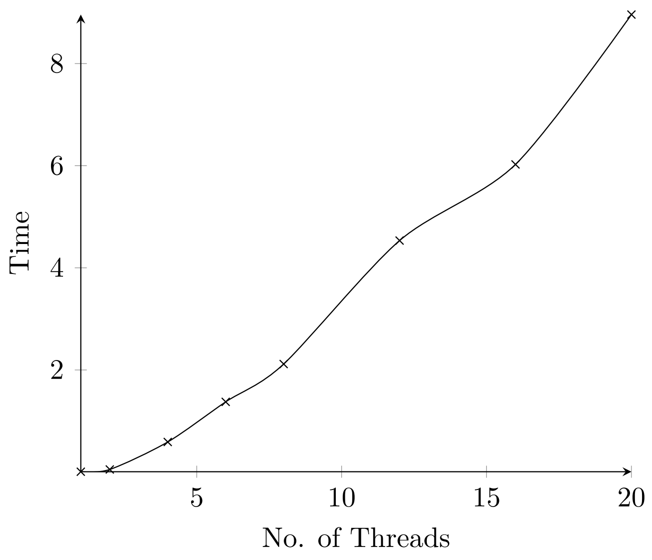

Figure 5: Weak scaling study for a wave-on-a-river simulation with 100 frames. The problem size is varied in terms of the number of cells per side, and is increased propotional to the number of threads starting at 100 frames for 1 thread, up till 2000 frames for 20 threads. The time reported is wall-clock time.

get 9 blocks (3×3), and from 16 threads onwards we get 16 blocks (4×4). Releasing the thresholding of the number of threads produces a graph which is somewhat more consistent with the expected trends for good strong scaling, but inevitably incurs more CPU idle time because threads are not being utilized. For the weak scaling study, we find that the trend is more or less as expected, and there is linear scaling when both problem size and processing units are increased proportionally.