CS 5220 Introduction to Parallel Programming — Fall 2015
Kenneth Lim (kl545), Batu Inal (bi49), Wensi Wu (ww382) — Project 3

1 Introduction

For compilation, we use a reasonable set of compiler flags:

```
-O3, -no-prec-div, -opt-prefetch, -xHost, -ansi-alias, -ipo -restrict
```

to handle the bare-bones of loop optimization, memory allocation/alignment, and architecture specific instructions.

1.1 Caveats

Issues with the Intel's VTune Amplifier prevented advanced hotspots analysis up till the time of submission. As a result, the timings reported in subsequent sections are either wall-clock times obtained via `omp_get_time` encapsulation of function calls, or CPU times obtained from VTune Amplifier's concurrency analysis mode.

2 Parallelism

The codebase ships by default with good OpenMP annotations that are difficult to beat. Table 1 shows a breakdown of the time spent per function for the untuned code when run on 2000 nodes generated with an edge inclusion probability of 0.05. Up to 86% of the CPU time spend by `square` kernel is ideal, and the bottleneck appears to be the implicit barrier at the end of the parallel region collating the shared and reduced variables. To some extent, this is an unavoidable overhead.

Note that the `memcpy` operation executed at the end of every iteration of the while-loop does not constitute a significant portion of the CPU time because the problem size is small. Instead, the difference in execution time for each iteration of the for-loop within `square` dominates. However, when the problem size is increased significantly, serial-bound operations takes up a larger proportion of the CPU time in comparison to the spinwait of the barrier because threading economies of scale apply. Table 2 shows a breakdown similar to that of Table 1 for 16,000 nodes, and attention is drawn to the difference in time contribution from `_intel_ssse3_memcpy` and `__kmpc_fork_barrier`. Unfortunately, the `memcpy` call is difficult to optimize as no parallelized version exists. In theory, it should be possible to either perform the copy operation using AVX instructions (where the grid is a multiple of 4, or padding to that end), or modify the values in place.

However, since the main bottleneck is still the `square` kernel, we ignore these minor optimizations in-lieu of focusing our efforts where there is larger room for improvement.

	Time (s)				
Function	Total	Idle	Poor	Ok	Ideal
`square`	45.143	0	4.032	2.080	39.030
`__kmp_barrier`	6.485	0	6.015	0.310	0.160
`__kmpc_reduce_nowait`	2.979	0	2.790	0.189	0
`__kmpc_fork_barrier`	2.553	1.420	0.972	0.132	0.030
`gen_graph`	0.030	0.010	0.020	0	0
`_intel_ssse3_memcpy`	0.030	0	0.030	0	0
`fletcher16`	0.030	0	0.030	0	0

Table 1: Concurrency analysis of untuned Floyd-Warshall APSP implementation with $n = 2000$ and $p = 0.05$. All times shown are CPU times.

	Time (s)				
Function	Total	Idle	Poor	Ok	Ideal
`square`	4753.975	5.341	230.318	0	4518.316
`_intel_ssse3_memcpy`	1.742	0.020	1.722	0	0
`fletcher16`	1.550	0	1.550	0	0
`gen_graph`	1.502	0.020	1.482	0	0
`__kmpc_fork_barrier`	0.439	0.010	0.429	0	0.030

Table 2: Abbreviated concurrency analysis of untuned Floyd-Warshall APSP implementation with $n = 16,000$ and $p = 0.05$. All times shown are CPU times.

The loop structure for this problem is somewhat similar to that of the previous project—the termination condition is enforced by a while-loop which *has* to run in a single-threaded environment, whereas the nested for-loop benefits from parallel execution.

In Project 2, the approach to parallelism was as follows:

1. Initialize the thread pool as early as possible, and outside any iteration scopes
2. Enforce a master thread that maintains the loop invariant
3. Allocate work via tasks created within the nested for-loop (from the master thread), and collate results using an implicit wait

The challenge with Project 3 is that the producer-consumer model created by items (2) and (3) is incompatible with the reduction construct provided by a `#pragma omp for` environment. Thus, a choice has to be made between incurring the overhead of thread pool initialization on every iteration of the while loop, or an atomic update of the `done` variable (by foregoing the reduction construct) causing thread contention. In our experiments, we found that the former resulted in significantly better timings, because the computation time associated with initialization of the thread pool is relatively long when n is large, whereas contending for access to a variable within a highly vectorized loop nullifies any performance improvements in that aspect.

3 Vectorization and Memory Access

A significant portion of our speed gains were obtained by vectorizing the `square` kernel. We go for the low-hanging fruit first by altering the loop iteration order from j, i, k, to j, k, i from outermost to innermost loops. This ensures that the innermost loop has unit stride, which itself provides only a modest speed boost (1.3x), but improves timing consistency as n changes.

Next, we enforce memory alignment to 64-byte boundaries on all allocated arrays by swapping out `malloc` and `free` calls for `_mm_malloc` and `_mm_free` calls respectively. This allows us to assert that all data structures in the innermost loop of `square` are aligned in memory for an added performance boost. The updated code is shown in Figure 1.

```
#pragma omp parallel for shared(l, lnew) reduction(&&:done)
for (int j = 0; j < n; ++j) {
  const int jn = j * n;

  for (int k = 0; k < n; ++k) {
    const int lkj = l[jn+k];

    #pragma vector aligned
    for (int i = 0; i < n; ++i) {
      const int lijOriginal = lnew[jn+i];
      const int lik = l[k*n+i];
      const int lijTest = lik + lkj;

      if (lijTest < lijOriginal) {
        lnew[jn+i] = lijTest;
        done = false;
      }
    }
  }
}
```

Figure 1: Updated `square` kernel with vectorization annotations

Note that we have cached some of the more frequently-used variables to minimize the number of operations carried out within the loop. The combined effect of the vectoriation tweaks made to `square` results in an approx. 4x speed-up on the wall-clock time. As an aside, it is interesting to note that merely enabling vectorization improves the performance of `infinitize` and `deinfinitize` by approx. 8x. This is somewhat expected because the loop operations are simple and predictable.

4 Blocking and Other Improvements

We investigated a blocked computation approach for `square`, but did not manage to obtain significant improvements. To be specific, we retrofitted code from the best performing groups in Project 1, swapping out the 4×4 matrix multiplication kernel for a simple addition loop. While doing so indeed improves performance, the effects are marginal, and only observed at large values of n where the overhead from copying the blocks is offset by the time taken to perform the calculation. High-level discussions with other groups who focused on blocking instead of parallelism and vectorization shows that the performance gains for both approaches are similar (approx. 4x overall).

An alternative approach to the problem is to evaluate a subset of the entire search space and terminate the computation as early as possible. The current implementation evaluates every point in the adjacency matrix, neglecting the fact that the distances are symmetric about the diagonal, and the distances *on* the diagonal are zero. That is, for some pair of nodes $0 \leq i \leq n$ and $0 \leq j \leq n$, where $i \neq j$, $P_{ij} = Pji$, where P is the path distance between the nodes identified in the subset notation. When $i = j$, $P_{ij} = 0$. Exploiting this relationship is equivalent to iterating over a triangular matrix, which still has the same time complexity as the original implementation, but benefits from a smaller constant factor as n increases.

In addition, the current implementation naively evaluates *every* node until it is certain that both `lnew` and `l` have converged. Instead of asserting `done = 1` at the start of every while-loop iteration, it might be possible to check for the reverse condition and break out of the nested for-loops once the arrays converge. This idea entails a different challenge: it is not possible to `break` or `goto` within an OpenMP parallel region. A known workaround is to guard the inner loops with a check for the global loop termination condition, as shown in Figure 2

While this method works as expected, the performance improvement obtained by early termination is lost almost immediately due to the overhead from the OpenMP `flush` calls, which are equivalent to a barrier. An alternative solution might be to exploit the new OpenMP 4.0 `#pragma omp cancel for` construct as shown in Figure 3.

The difficulty with this method is that canceling out of the for-loop leaves any reduction variables in an indeterminate state by default. This means that we *cannot* rely on the value of `done` to terminate the while-loop. We were unable to make further progress in this area by time of submission, but expect to investigate further leading up to the final report.

```
#pragma omp parallel for shared(l, lnew) reduction(&&:done)
for (...) {
  #pragma omp flush(done)
  if (!done) {
    for (...) {
      for (...) {
        if (converged) {
          done = true;
          #pragma omp flush(done)
        }
      }
    }
  }
}
```

Figure 2: Toy example of OpenMP flush constructs

```
#pragma omp parallel shared(l, lnew) reduction(&&:done)
{
  #pragma omp for collapse(3) schedule(dynamic)
  for (...) {
    for (...) {
      for (...) {
        if (converged) {
          done = true;
          #pragma omp cancel for
        }

        #pragma omp cancellation point for
      }
    }
  }

  #pragma omp cancellation point parallel
}
```

Figure 3: Toy example of OpenMP cancellation constructs

5 Offloading and MPI

Unfornately, we were unable to get MPI to work because of issues with the cluster. We expect to periodically reattempt a study with MPI, and will report accordingly if successful. Offloading is currently a work-in-progress.

6 Scaling Studies

We report strong and weak scaling studies for our current best attempt, which implements the vectorization and parallelism changes described above. To avoid spurious results at small values of n, we run each instance 10 times and average the result. Figure 4 shows the results for a strong scaling study in which we vary the problem size from $n = 500$ to $n = 8000$, doubling at each tick. The number of OMP threads is clamped to the maximum available in the system. The regression line is linear, which confirms that we are scaling correctly with the problem size. Figure 5 shows the results for a weak scaling study in which we hold the problem size constant at $n = 4000$ and vary the number of OMP threads from 1 to 24, doubling at each tick up to 16. The regression line demonstrates exponential decrease, which corroborates the speedup obtained from parallelizing the `square` kernel.

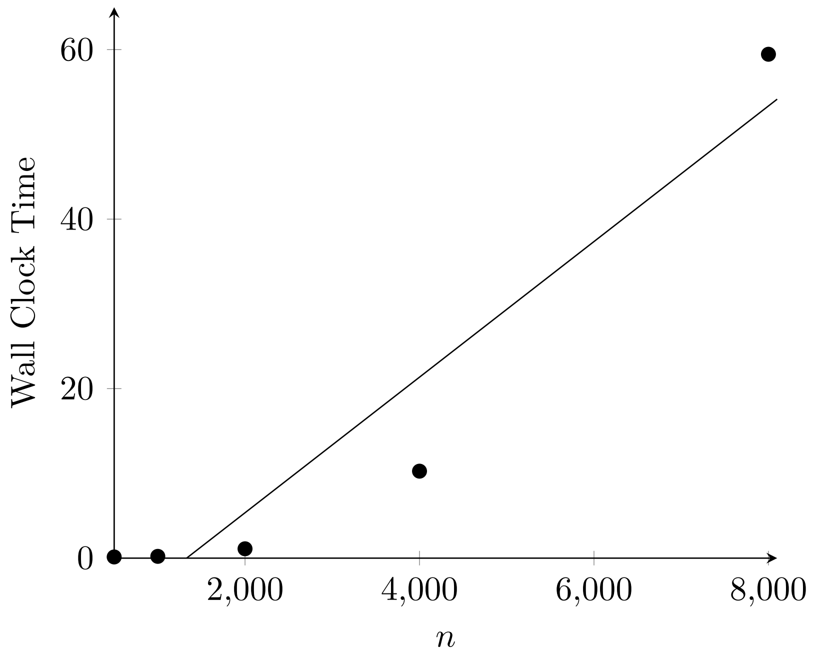

Figure 4: Strong scaling study varying n from 500 to 8000 with 24 OMP threads. The best-fit line is linear with $R^2 = 0.912$

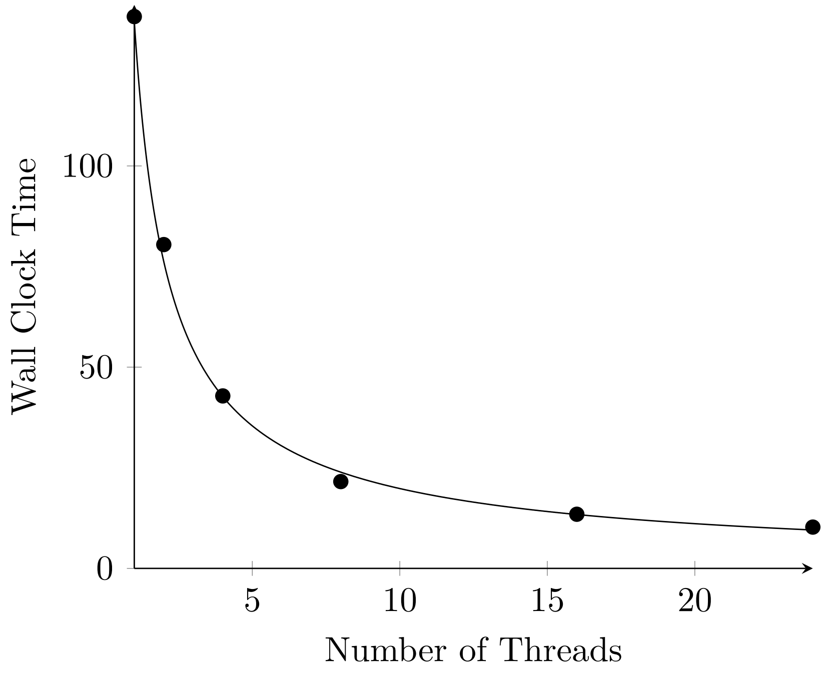

Figure 5: Weak scaling study varying the number of OMP threads from 1 to 24 with $n = 4000$. The best-fit line is exponential with $R^2 = 0.996$