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. 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.

3 Vectorization

4 Blocking

5 Ongoing Work