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

1 Domain Decomposition

We implement a tiled version of the Floyd-Warshall algorithm[1] that builds on the blocking concept investigated by-and-large in the mid-term report. At each step, the algorithm traverses the diagonal of the adjacency matrix, first expanding computation outwards in the four cardinal directions, and then into the four quadrants demarcated by the previous expansion. Figure 1 provides an overview of the computation process, where the notation c_{ij} represents a matrix block whose top left corner starts at i, j in the parent matrix c. In this case, c is the input matrix. `fwi` and `fwi_abc` are min-plus matrix multiplication kernels to be later defined.

```
for (int k = 0; k < num_blocks; ++k) {
  // Phase 1: Diagonal Block
  fwi(BLOCK_SIZE, c_kk, c_kk, c_kk);

  // Phase 2: Left and Right
  for (int j = 0; j < num_blocks; ++j)
    if (j != k)
      fwi(BLOCK_SIZE, c_kk, c_kj, c_kj);

  // Phase 3: Up and Down
  for (int i = 0; i < num_blocks; ++i)
    if (i != k)
      fwi(BLOCK_SIZE, c_ik, c_kk, c_ik);

  // Phase 4: Quadrants
  for (int i = 0; i < num_blocks; ++i)
  for (int j = 0; j < num_blocks; ++j)
    if (i != k && j != k)
      fwi_abc(BLOCK_SIZE, c_ik, c_kj, c_ij);
}
```

Figure 1: Overall evaluation schema for Tiled FW-APSP.

Since the main arithmetic operation is min-plus matrix multiplication (MPMM) on three operands, the size B of each block is chosen to be such that a triplet of such blocks

[1]Described by Han et. al in *Program Generation for the All Pairs Shortest Path Problem*

fits within the L3 cache, i.e. $3B^2 = \mathrm{L}_3$. This works out to a familiar value of $B = 64$. Within each block, we employ a secondary level of partioning using a "block" size of 16 when iterating through the loop. This is implemented as a loop-unrolling factor, but has the effect of ensuring that the operands of the inner loops fit within the L2 cache. These correspond to `UNROLL_*` in Figure 2 and Figure 3, where $*$ is the context of the loop. In a more robust implementation, these values could be the target of an ATLAS optimization procedure.

There are two cases for the MPMM operation: either all three operands are mutually distinct, or they are not. This dichotomy is important because the former case allows us to change the canonical kij loop (where k is the path exploration step) to a jik loop, which exhibits much better stride and data locality. The latter, unfortunately, still remains unchanged. Figure 2 shows the mutually distinct case, in which `step` is the innermost loop, whereas Figure 3 shows the non-distinct case, in which `step` *has* to be the outermost loop.

```
1 for (int row  = 0; row  < n; row  += UNROLL_ROW)
2 for (int col  = 0; col  < n; col  += UNROLL_COL)
3 for (int step = 0; step < n; step += UNROLL_STEP)
4   for (int step_p = step; step_p<step+UNROLL_STEP; ++step_p)
5   for (int row_p = row; row_p < row + UNROLL_ROW; ++row_p)
6   for (int col_p = col; col_p < col + UNROLL_COL; ++col_p)
7     int new_path = a[row_p][step_p] + b[step_p][col_p];
8     c[row_p][col_p] = min(c[row_p][col_p], new_path);
```

Figure 2: `fwi_abc`: The mutually distinct case.

```
1 for (int step = 0; step < n; ++step)
2 for (int row  = 0; row  < n; row += UNROLL_ROW)
3 for (int col  = 0; col  < n; col += UNROLL_COL)
4   for (int row_p = row; row_p < row + UNROLL_ROW; ++row_p)
5   for (int col_p = col; col_p < col + UNROLL_COL; ++col_p)
6     int new_path = a[row_p][step] + b[step][col_p]
7     c[row_p][col_p] = min(c[row_p][col_p], new_path);
```

Figure 3: `fwi`: The non-distinct case.

To optimize memory allocation, we batch allocate four scratch blocks on 64-byte boundaries with dimensions $B \times B$ to be reused for the lifetime of the computation. We also tried to use `scalable-aligned-malloc` from Intel's TBB library instead of `_mm_malloc` in the hope of exploiting the number of hardware threads available, but were ultimately unable to proceed because we encountered configuration problems on the cluster[2].

[2]Specifically, `tbb/tballocator.h` insists on importing the C++ standard type library instead of the C

2 Vectorization

We rewrote the codebase to utilize Intel's Cilk Array Notation when performing array operations. In exchange for a certain amount of memory locality (across the minor axis) that is lost because entries on the minor axis in the 2-dimensional array are not necessarily adjacent to each other in memory, we obviate the need to add dependency-relaxation annotations to for-loops. This is particularly useful when copying data to-and-from blocks because we can use the full width of intrinsic functions rather than doing an elementwise copy. Figure 4 shows the code we use to extract a block from the input matrix, where `i_start` / `i_end` and `j_start` / `j_end` indicate the region of interest.

```
block[0:BLOCK_SIZE][0:BLOCK_SIZE]
  = c[i_start:i_end][j_start:j_end];
```

Figure 4: Using Cilk array notation to copy from the parent matrix c to a block.

The loop-unrolling structure of `fwi` and `fwi_abc` could also be substitued with explicit 8-way AVX instructions. Han et. al describe a schema for MPMM that uses the same number of operations as the un-vectorized version—a broadcast and an addition, followed by a minimum comparison. While we implemented this version and found that there were tangible speed improvements at small input sizes, we did not have the time to run benchmarks to determine if our observations were consistent, or the result of spurious events. As such, our final submission leaves this job to the compiler, which still manages to produce a decent output by virtue of our efforts with Cilk.

3 Parallelism and Offloading

The domain decomposition we have used lends itself well to parallelism because iterations in each phase can be distributed independently to workers. Our only constraint is that we must collect the results from each phase before proceeding to the next. We explore both MPI and OpenMP in our implementation.

3.1 MPI

Our work with MPI primarily uses `MPI_Gatherv` to distribute MPMM jobs at each phase and collate the result. The advantage is that the collation process eliminates the need to copy-back from blocks into the input array because we can specify that the displacement on collation be equal to the block size. The disadvantage of MPI is that its main purpose is to facilitate memory sharing across processes, whereas the tiled

equivalent. Compiling with icpc instead of icc produced small explosions which we were reluctant to mitigate at 4am on Thursday morning.

FW-APSP approach we have used already eliminates that invariant. Therefore, to some extent, we are maintaining the MPI communication overhead without actually utilizing it. Our final implementation favors OpenMP over MPI, though one postulates that a hybrid implementation (which some groups have attempted) could capture the best of both worlds.

3.2 OpenMP

The situation with OpenMP is flipped—the overhead is incurred in the form of copying to and from blocks, but jobs distributed to threads can be executed without undue concern for ordering or data sharing. We create a master thread of execution which dispatches OpenMP tasks within the for-loops for each phase, proceeding to the next only when a barrier condition is fulfilled. Lastly, we observe that the amount of work that could be done in each task is variable because an update may or may not happen on each iteration. Therefore, we use a static scheduler with the chunk size set to a small multiple of the total number of blocks.

3.3 Offloading

We experimented with offloading the entire tiled algorithm to the coprocessor. Our motivations for doing so were that a fine-grained offloading at the MPMM level, while possible, incurs a large amount of memory allocation overhead, and does not fully take advantage of the large number of hardware threads (236, by `omp_get_max_threads`) available on the coprocessor. Instead, we dump the entire computation to the coprocessor immediately upon program initialization.

To ensure that our timing measurements are accurate, we also warm the coprocessor by performing a dummy offload prior to main program initialization. We also specify the KMP thread affinity parameter to map threads on the main processor to the coprocessor in accordance with the main processor architecture. Thus we have the following export variables:

```
1 export KMP_AFFINITY="granularity=core,type=scatter"
2 export OFFLOAD_INIT=on_start
```

The cutoff point for offloading is determined by the number of blocks that are being considered. Given a block size of 64 and 24 threads on the main processor, it is safe to say that any computation whose dimensions are significantly larger than $64 \times 24 = 1536$ will benefit from the additional number of threads available on the coprocessor. For our purposes, we use a weak criteria where we offload for any computation whose dimensions are larger than 1024. This criteria is weak because the economies of scale likely kick in only at significantly larger problem sizes where most of the 236 hardware threads on the

coprocessor are being saturated. Thus, in a more robust implementation, this threshold could be the subject of an ATLAS optimization routine.

4 Scaling Studies

The base case considered here is the implementation that shipped with the assignment. We measure speedup as the ratio of the wall-clock time taken for the optimized case versus the wall-clock time for the base case. For ease of measuring scaling performance, we disable offloading so that we need only consider the range of 1 to 24 threads and a corresponding problem size. For strong scaling shown in Figure 6, we use a slightly larger problem size ($n = 4096$) in order to stay well above the threshold at which blocking/tiling is efficient.

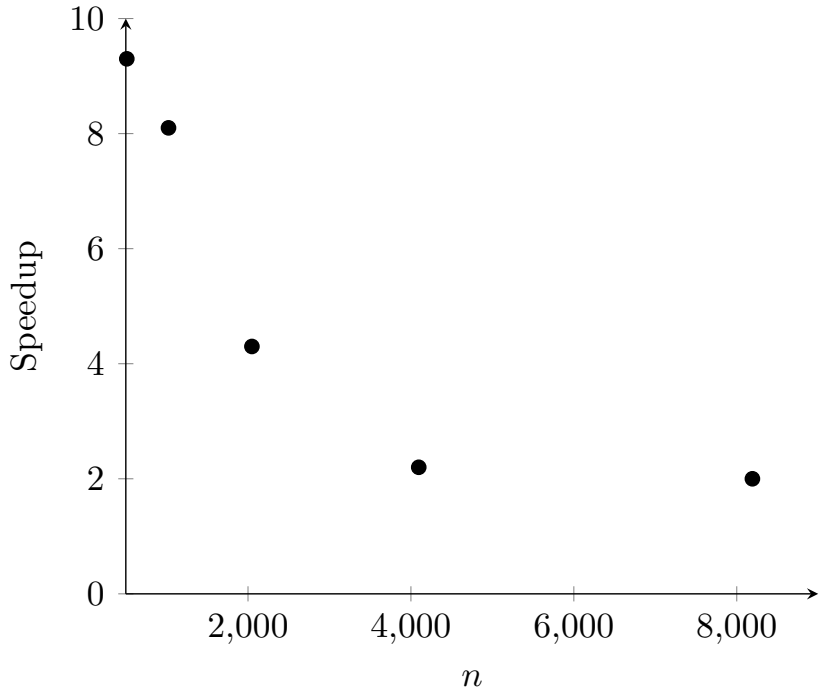

Figure 5: Varying n from 512 to 8192 (powers of 2), and varying the number of OpenMP threads from 1 to 24.

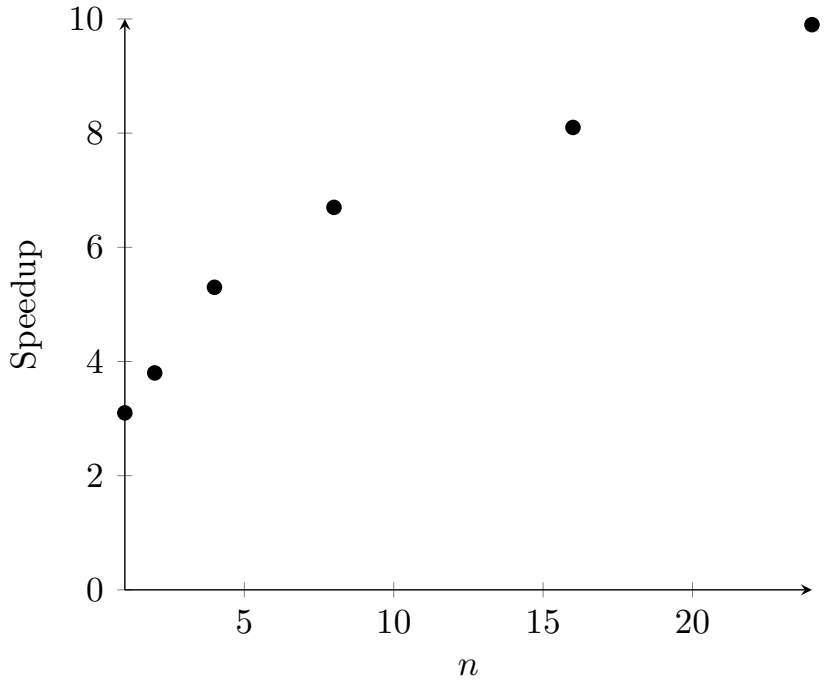

Figure 6: Fixing $n = 4096$, and varying the number of OpenMP threads from 1 to 24.