HPCSE II - Exercise 1

Anian Ruoss

March 2, 2019

All experiments were run on a node obtained by running the following command:

```
bsub -n 24 -R fullnode -R "select[model=XeonE5_2680v3]" -Is bash
```

Task 4

a) See the first row in Table 1.

b) The optimal number of grids is 6 as illustrated in Table 1.

Table 1: Iterations and running times for different numbers of grids for DownRelaxations = 1 and UpRelaxations = 1.

GridCount	Iterations	Running Time	Verification
1	199204	150.165s	Passed
2	43522	57.425s	Passed
3	32457	46.268s	Passed
4	32102	46.524s	Passed
5	30625	44.926s	Passed
6	22973	33.810s	Passed
7	33171	48.768s	Passed
8	51762	75.945s	Passed

c) The optimal combination of UpRelaxations and DownRelaxations is determined in a greedy way. The optimal number of DownRelaxations in terms of running time is 3 as illustrated in Table 2. Although, values of 5 and 7 produce less iterations, the time per iteration is significantly higher.

Table 2: Iterations and running times for different numbers of DownRelaxations given GridCount = 6 and UpRelaxations = 1.

DownRelaxations	Iterations	Running Time	Verification
1	22973	33.770s	Passed
2	37152	72.471s	Passed
3	13467	33.071s	Passed
4	20885	61.631s	Passed
5	12068	41.526s	Passed
6	14811	58.096s	Passed
7	10130	44.835s	Passed

The optimal number of UpRelaxations is 1 as illustrated in Table 3.

Table 3: Iterations and running times for different numbers of UpRelaxations with GridCount = 6 and DownRelaxations = 3.

UpRelaxations	Iterations	Running Time	Verification
1	13467	33.055s	Passed
2	13467	34.066s	Passed
3	13467	35.200s	Passed
4	13467	36.909s	Passed
5	13467	37.349s	Passed
6	13467	38.467s	Passed
7	13467	39.613s	Passed

d) The optimal configuration with respect to the results from tables 1, 2, and 3 is given by GridCount = 6, DownRelaxations = 3, and UpRelaxations = 1. To find the global minimum one would have to run a grid search on all three parameters. However, since the exercise asked for a greedy approach and since the local minimum performed fairly well, the grid search was omitted.

e) The solution reaches its equilibrium state for low frequencies faster on coarser grids as every cell covers more area (it is also cheaper to compute a full pass on coarse grids). Larger frequencies can only be resolved on finer grids and thus the solution is alternatingly computed on finer and coarser grids to increase convergence speed while maintaining a small discretization error.

f) Using more DownRelaxations increases the number of iterations in an alternating manner (the TA could not provide an explanation for this phenomenon during the exercise class). Furthermore, the running time decreases slightly from 1 to 3 because at every grid level the solution can converge to its equilibrium state before passing on its values to the next grid. However, using more relaxation steps doesn't lead to a better convergence as it has already converged on the grid with fewer relaxations and thus just increases the time spent on every grid. The number of UpRelaxations doesn't seem to affect the iterations at all and just increases the time spent at every grid.

Task 5

a) *Loop Interchange* can be applied to a prevent bad memory access patterns, i.e. swapping the i- and j-loops. *Loop Fusion* can be applied to increase data locality, i.e. performing the computations of two subsequent for-loops that rely on the same data in a single for-loop instead to reduce cache misses.

b) Every grid should be allocated in its entirety before moving on to the next one as this prevents cache thrashing. This is an example where *Loop Fusion* should not be applied as the different for-loops operate on different data.

c) As smoothing on the 0-th grid takes the most time, the maximum possible speedup can be obtained by optimizing the **applyJacobi** method. Thus, to determine the effect of *Loop Blocking*, the for-loops in **applyJacobi** can be replaced with the blocked for-loops:

```
for (int ii = 1; ii < g[l].N - 1; ii += B) {
  for (int jj = 1; jj < g[l].N - 1; jj += B) {
    for (int i = ii; i < std::min(ii + B, (int)g[l].N - 1); ++i) {
      for (int j = jj; j < std::min(jj + B, (int)g[l].N - 1); ++j) {
        g[l].U[i][j] =
            (g[l].Un[i - 1][j] + g[l].Un[i + 1][j] + g[l].Un[i][j - 1] +
             g[l].Un[i][j + 1] + g[l].f[i][j] * g_l_h_squared) *
            0.25;
      }
    }
  }
}
```

However, even when trying different block sizes $B \in \{8, 16, 32, 64, 128\}$ (larger block sizes are not applicable due to the size of the largest grid), the running time does not decrease. In fact, since the grid sizes are so small that enough data can fit into the cache, the overhead induced by the two additional for-loops causes the program to run slower. For this reason, *Loop Blocking* was not used for the final implementation.

Task 6

a)
- Division operator replaced with multiplication of inverse.
- Multiplication by 1 removed.
- *pow* $(x, 2)$ replaced with $x \cdot x$.
- Constants pre-computed and moved out of loops.
- Expressions simplified with associativity.
- Copy of array avoided by pointer swap.

Task 7

To generate the optimization reports, the following line in the Makefile has to be changed to:

```
CFLAGS = -O3 -D NOALIAS -D NOFUNCCALL -qopt-report=3 -qopt-report-phase=vec
```

a) Most loops cannot be vectorized due to vector dependence. This means that the compiler cannot know if the data accessed in one iteration are also changed in another iteration (which would case a race condition if vectorized).

b) This problem can be fixed by adding **#pragma** ivdep in front of the loop if the vectors are in fact independent (which they are as the results are computed from and stored into different vectors).

c) SIMD operations start working at the beginning of a block of memory and thus if the arrays are not aligned with the block starts the operations need to gather data from different blocks which is very inefficient. During the exercise class the following grids allocation was recommended:

```
g[i].U = (double**) _mm_malloc (sizeof(double*) * g[i].N, 16);
for (int j = 0; j < g[i].N; j++) {
    g[i].U[j] = (double*) _mm_malloc (sizeof(double) * g[i].N, 16);
}
```

However, even in combination with **#pragma** vector aligned this allocation slowed down the implementation (probably because it only aligns pointers and not the underlying data). Hence, the original allocation was used for the final implementation.

Table 4: Iterations and running times for the three problems after all (beneficial) optimizations have been performed.

Problem	Iterations	Running Time	Verification
1	7844	1.422s	Passed
2	13467	8.238s	Passed
3	12865	7.824s	Passed