CS 5220 Homework 3 Initial Report - Group 12

Amiraj Dhawan (ad867) Weici Hu (wh343)
Sijia Ma (sm2462)

November 10, 2015

1 Profiling of Serial Performance

The table below summarizes the profiling result of the serial code. As expected, function square takes up the most of the run-time.

function	CPU time(s)
square_step	27.519
fletcher16	0.020
__intel_ssse3_rep_memcpy	0.01
gen_graph	0.01
genrand	0.01

2 Parallel using openMP

While the code is given by the instructor, we perform strong scaling and plot the result in figure 1.

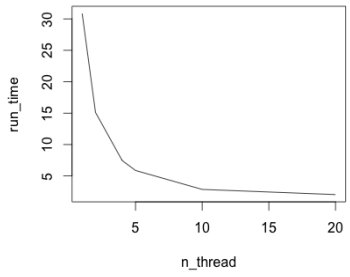

Figure 1:

3 Parallel using MPI

We specify thread 0 as our root which alone runs all the functions except the shortest_paths_mpi and square_stripe. We parallel the outermost loop in square_stripe by distributing the work to all the non-root thread according to their thread ID.

We also run strong scaling on the mpi code. The result is summarized in figure 2

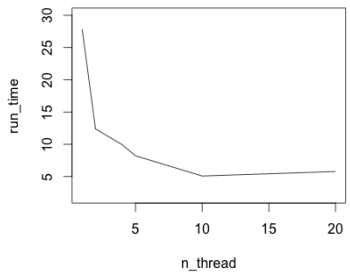

Figure 2:

4 Serial Tuning with OpenMP Code

The code provided used OpenMP library to parallelize the implementation of Floyd-Warshall algorithm with a time complexity of $O(Log(n) * n^3)$. The profiling of this code yields the following result:

Time as per the code: 3.17277 seconds
Amplxe Hotspots: Function: *square* → 43.093 seconds
memcpy → 0.029 seconds

Using this as the base performance, we tried to do the following optimizations:

i) Removed a redundant loop

In the function ***shortest_paths***, the following loop iteration iterates over the entire graph with time complexity of $O(n^2)$ which can be removed.

```
for (int i = 0; i < n*n; i += n+1)
      l[i] = 0;
```

We removed this loop and updated the ***infinitize*** function which was also looping over the same matrix to perform the above function.

```
static inline void infinitize(int n, int* l){
   for (int i = 0; i < n*n; ++i)
    if (l[i] == 0 && (i % (n + 1)) != 0)
       l[i] = n+1;
}
```

Performance after this step:

Time as per the code: 3.08317 seconds
Amplxe Hotspots: Function: *square* → 43.227 seconds
memcpy → 0.028 seconds

ii) Removed memcpy calls in shortest_paths function

In the function ***shortest_paths,*** after each iteration of the for loop in which the square function is called, variable lnew is copied in variable l which is of size n*n and takes time. The code becomes:

```
int* restrict lnew = (int*) calloc(n*n, sizeof(int));
int flag = 1;
for (int done = 0; !done; ) {
   done = flag ? square(n, l, lnew) : square(n, lnew, l);
   flag = !flag;
}
if(!flag) {
          memcpy(l, lnew, n*n * sizeof(int));
```

```
}
```

In the case, rather than always doing a memcpy from lnew to l, we call the function square with arguments in a swapped order alternatively. Finally at the end if a copy is required in the case if the last execution of the loop, the function square was called with the arguments as square(n, lnew, l), the memcpy function is called only once.

Performance after this step:

Time as per the code: 2.98006 seconds
Amplxe Hotspots: Function: *square* → 29.199 seconds
memcpy → 0.010 seconds

iii) Copy Optimization of the matrix for better cache hits

In the function ***square,*** we take the transpose of l matrix and store it in ltrans. The reason for this expensive operation is because the inner most loop of the original square function was accessing the matrix l in a way that would lead to a lot of cache misses.

```
//Original Code

int done = 1;
#pragma omp parallel for shared(l, lnew) reduction(&& : done)
   for (int j = 0; j < n; ++j) {
      for (int i = 0; i < n; ++i) {
         int lij = lnew[j*n+i];
         for (int k = 0; k < n; ++k) {
            int lik = l[k*n+i]; //This line will result in a lot of cache misses
            int lkj = l[j*n+k];
            if (lik + lkj < lij) {
               lij = lik+lkj;
               done = 0;
            }
         }
         lnew[j*n+i] = lij;
      }
   }
   return done;
```

Thus we create another variable to store the transpose of the matrix l to avoid these cache misses. The resultant code for the function square becomes:

```
//Optimized Code

//Copying the transpose of l into ltrans
int* restrict ltrans = malloc(n*n*sizeof(int));
   for (int j = 0; j < n; ++j) {
      for (int i = 0; i < n; ++i) {
         ltrans[i*n + j] = l[j*n + i];
```

```
    }
}

int done = 1;
        omp_set_num_threads(n_threads);
#pragma omp parallel for shared(l, lnew) reduction(&& : done)
for (int j = 0; j < n; ++j) {
   for (int i = 0; i < n; ++i) {
      int lij = l[j*n+i];
      for (int k = 0; k < n; ++k) {
         int lik = ltrans[i*n + k]; //This transpose results in a cache hit
         int lkj = l[j*n+k];
         if (lik + lkj < lij) {
            lij = lik+lkj;
            done = 0;
         }
      }
      lnew[j*n+i] = lij;
   }
}
free(ltrans);
return done;
```

This optimization increased the performance of the function quite a lot.

Performance after this step:

Time as per the code: 0.841073 seconds
Amplxe Hotspots: Function: *square* → 6.953 seconds
memcpy → 0.010 seconds

Optimizations performed table:

	Time as per the code	Time as per amplxe
Unoptimized	3.17277	Square: 43.093 memcpy: 0.029s
i) Removed redundant loop	3.08317	Square: 43.227s memcpy: 0.028s
ii) Removed memcpy calls	2.98006	Square: 29.199s memcpy: 0.010s
iii) Copy Optimization for better cache hits	0.841073	Square: 6.953s memcpy: 0.010s