Final Report

Enrique Rojas (elr96), Wenjia Gu (wg233) and Laura Herrle (lmh95)

In this document we are going to present different approaches to solve the shortest path problem using parallel programming. First we are going to present a short analysis of the performance of the original code provided by the instructor, then we will propose an MPI implementation and analyze its performance and finally we will use some tunning approaches to lower the processing time using OpenMP.

Profiling

Using VTune Amplifier, we generated several reports to find the parts of the code which consumes more CPU time. In Figure 1, we can see an extract of the hotspots report. Here we can see that the square function is the heaviest in term of computational work.

Function	Module	CPU Time
square	path.x	40.931s
__kmp_fork_barrier	libiomp5.so	9.011s
__kmp_barrier	libiomp5.so	5.587s
__kmpc_reduce_nowait	libiomp5.so	2.649s
fprintf	libc-2.12.so	0.881s
write_matrix	path.x	0.030s
fletcher16	path.x	0.030s
gen_graph	path.x	0.020s
__intel_ssse3_rep_memcpy	path.x	0.020s
infinitize	path.x	0.010s
pthread_create	libpthread-2.12.so	0.010s
genrand	path.x	0.010s
write_matrix	path.x	0.009s

Figure 1. Extract of the hotspots report.

Once the heaviest computational part of the code was identified, we proceed to run an scale analysis to see the response of the original implementation. In Figure 2 we can see a plot of the strong scaling for 2000 nodes and the same probability of 0.05. As expected, the decay in the values of time is significant in the strong scaling. To make the weak scaling, we begin with 1000 and multiply the number of nodes by the square root of the factor of increasing in the number of threads, with this we try to maintain approximately constant the amount of work per thread. We see that the relation in this case is almost linear.

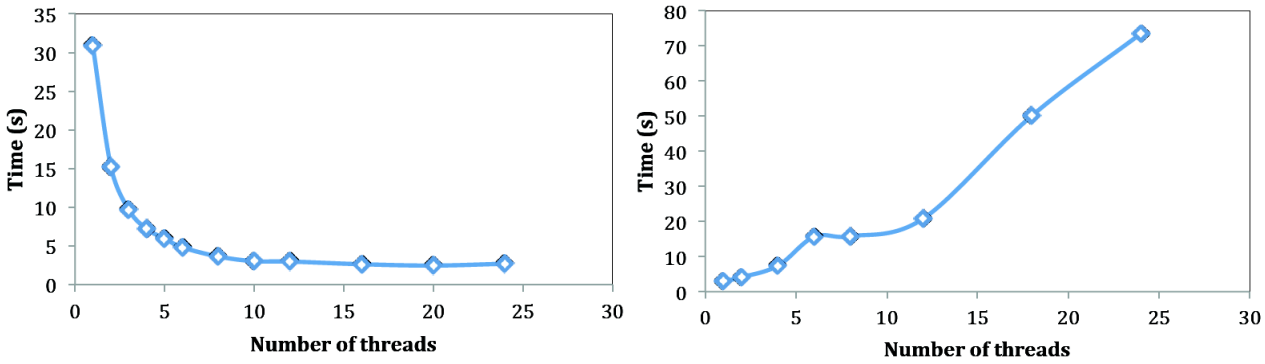

Figure 2. Right: Strong scaling. Left: Weak scaling. Results from original OpenMP code.

MPI Implementation

We followed the approach of the Group 14. The main idea here is to minimize the amount of modifications of the code. The first step is to make the function *square* able to select an specific number of columns, this way the the updates could be performed by independent processes column-wise. For further specifications, see the comments in the code. In Figure 3, we show the strong and weak scaling analysis made with this implementation.

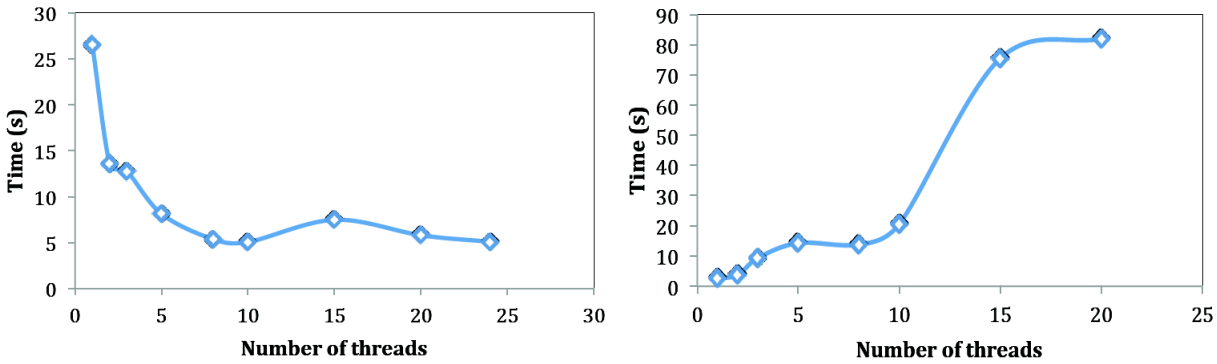

Figure 3. Right: Strong scaling. Left: Weak scaling. Results from MPI implementation.

Tunning

The O(n^3*log n) Floyd-Warshall Algorithm performs similarly with matrix multiplication. Therefore, we tuned the original OpenMP code in a similar way of the matrix multiplication optimization. The tuning process included three steps.

1. *Elimination of the redundant for-loop*
   The first for-loop in original *shortest_path* function (Figure 4) which requires a time complexity of O(n^2) can be included in the for-loop of function *infinitize* (Figure 5).

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

Figure 4: for-loop in shortest_path function

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

Figure 5: for-loop in infinitize function

   We edited the *infinitize* function so the first for-loop in *shortest_path* function can be eliminated (Figure 6).

```
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;
}
```

Figure 6: Modified infinitize function, and the first for-loop in shortest_path function has been deleted

2. *Elimination of the memcpy operation*
   In the original *shortest_path* function, there are two matrices – l and lnew. At the end of each iteration, lnew is copied to l which is of size n by n. We modified the code by swapping the roles of these two matrices so that the memcpy function doesn't need to be called every time.

```
int* restrict lnew = (int*) calloc(n*n, sizeof(int));
memcpy(lnew, l, n*n * sizeof(int));
for (int done = 0; !done; ) {
    done = square(n, l, lnew);
    memcpy(l, lnew, n*n * sizeof(int));
}
free(lnew);
deinfinitize(n, l);
```

Figure 7: Original code in shortest_path function

```
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));
}

free(lnew);
deinfinitize(n, l);
```

Figure 8: Tuned code in shortest_path function

3. *Copy optimization*

From the profiling process, we found that the *square* function took a lot of time. Further more, the worst performance happened in the innermost loop, which is because the original code was accessing matrix l in a way that led to a lot of cache misses. To improve this, we introduced a new matrix ltrans which is the transpose matrix of l. By using ltrans, we can access the matrix in a row-major order, which will help us to obtain better cache hits.

```
{
    int done = 1;

    //omp_set_num_threads(24);

    #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];
                int lkj = l[j*n+k];
                if (lik + lkj < lij) {
                    lij = lik+lkj;
                    done = 0;
                }
            }
            lnew[j*n+i] = lij;
        }
    }
    return done;
}
```

Figure 9: Original code of function square

```
{
    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(24);

    #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];
                int lkj = l[j*n+k];
                if (lik + lkj < lij) {
                    lij = lik+lkj;
                    done = 0;
                }
            }
            lnew[j*n+i] = lij;
        }
    }
    free(ltrans);
    return done;
}
```

Figure 10: Tuned code of function square

Each of the optimization methods explained above results in improvement of the performance of the code, which is shown as the decrease of time. We ran each code for five times and took the average time. The result is shown in Figure 11.

Original	Optimization 1	Optimization 1+2	Optimization 3
43.566	29.809	16.326	6.674

Figure 11: CPU time of square function (in s)

We also conduct the strong scaling analysis for the tuned code. As the result shown, the CPU time of square function has been well improved by the optimization process. Huge progress occurs when there is small number of threads. As the number of threads increase, the value of the time that we saved decreases, which is because general performance is improved by the increasing threads, while the difference can still be noticed.

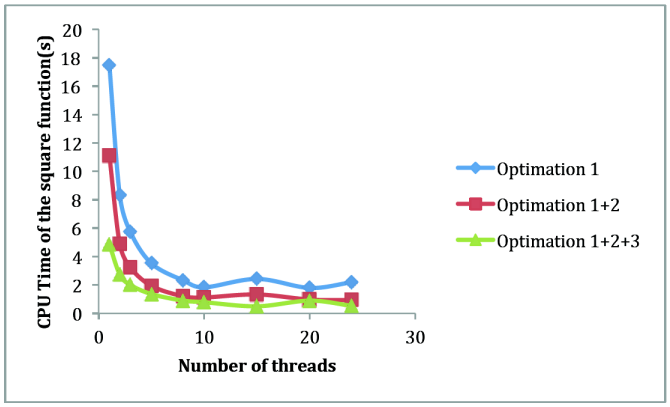

Figure 12: Strong scaling analysis for the tunned code.