CS 5220 Homework 3 Initial Report - Group 12

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

November 19, 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

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

2 Parallel using MPI

In this section we discuss how we attempted to speed up the computation by using MPI. We start with simply paralleling the code with MPI. To tune the code for faster implementation, we changed the order of the loop in function 'square' and added blocks in a way that is similar to the first project. We implemented the MPI version. The core idea is that, in each step of the iteration, we need to compute the length of the shortest paths between every pair of nodes within 2^s steps, based on the results of the previous step, which is similar to the procedure of computing matrix multiplication. Let's call the shortest path matrix at step s $A^{(s)}$. In our implementation, in each step, we divide our work of computing the "product" of the current matrix and itself, meaning to compute

$$A^{(s+1)} = A^{(s)} \otimes A^{(s)}$$

into several sub-works and assign them to each processor. More specifically, if we have k processors, then we divide the matrix $A^{(s)}$ into k stripes, $A_1^{(s)}, A_2^{(s)}, \ldots, A_k^{(s)}$, so that

$$A^{(s+1)} = [A_1^{(s+1)}, A_2^{(s+1)}, \ldots, A_k^{(s+1)}] = A^{(s)} \otimes [A_1^{(s)}, A_2^{(s)}, \ldots, A_k^{(s)}].$$

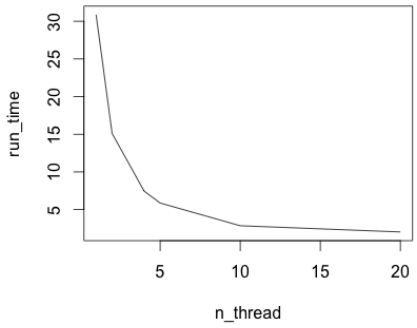

Figure 1: Plot of number of threads against run-time (second) with n=2000 using openMP

In processor i, we compute

$$A^{(s)} \otimes A_i^{(s)},$$

And then glue them together. We use the processor as our root to broadcast the current matrix $A^{(s)}$. Then we run the matrix multiplication of a whole matrix and a stripe matrix in each processor, and then use root to gather all of the stripe matrices and put them together. Notice that the whole process stops when there is no change after one step. In our implementation, only when there is no change for any of the stripe matrices, we can say that there is no change in this step.

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

2.1 Loop reordering

Similar to the matrix multiplication, we can change the loop order so that we can retrieve a contiguous memory block so that the program runs more efficiently. In our experiments, we show that the order k-j-i is the fastest, and the improvement verses the original j-i-k is remarkable.

Besides, that we also did vectorization on MPI version, and the efficiency was improved by about 20%. Here we compare the running time of the tuned version and the previous version in Figure 3.

2.2 Adding blocks

Remember when we implemented the matrix multiplication, the blocking strategy along with copy optimization makes it much faster. Here we utilize the

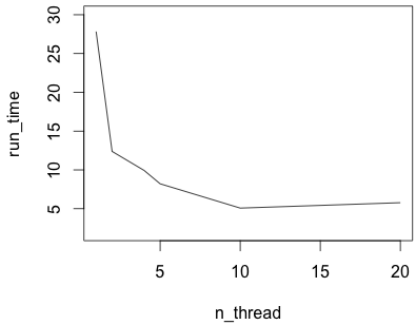

Figure 2: Plot of number of threads against run-time (second) with n=2000 using MPI

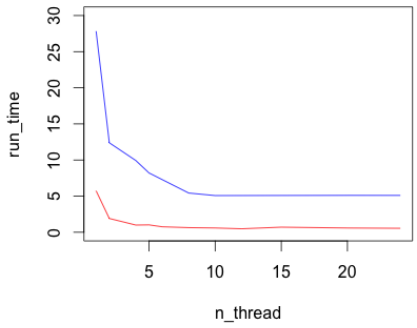

Figure 3: Red line plots the number of threads against run-time (second) after loop reordering; blue line plots the number of threads against run-time (second) before reordering. Both have n=2000 and use MPI

same strategy. In each processor, we divide both our whole matrix and the stripe matrix into small blocks, where the block size is carefully chosen so that it fits the cache. Inside each block, the data is stored in row-major, and the blocks are also in row-major. In this way, we can retrieve a contiguous memory block so that the program runs more efficiently. In the following graph we can see that, when the dimension increases, especially when $n \geq 3000$, the blocking MPI version is much more efficient then that without blocking as shown in 5. However, when n is small and when we run the program on many processors, the improvement is not outstanding, sometimes it is even slower. The reason is that, using blocking method, we have more data transfer and copy, on which

the processors spend a lot of time. When the total computation load is not very large, the improvement resulted by blocking method cannot offset the running time for those extra work.

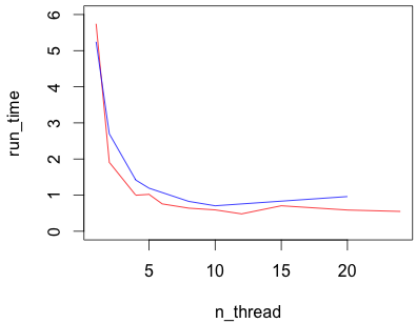

Figure 4: Blue line plots the number of threads against run-time (second) after adding blocks; Red line plots the number of threads against run-time (second) before adding blocks(but after loop reordering). Both have n=2000 and use MPI

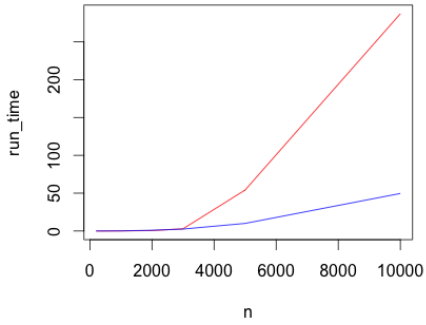

Figure 5: Blue line plots the number of nodes(n) against run-time (second) after adding blocks; Red line plots the number of nodes(n) against run-time (second) before adding blocks(but after loop reordering). Both have number of threads = 20 and use MPI

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