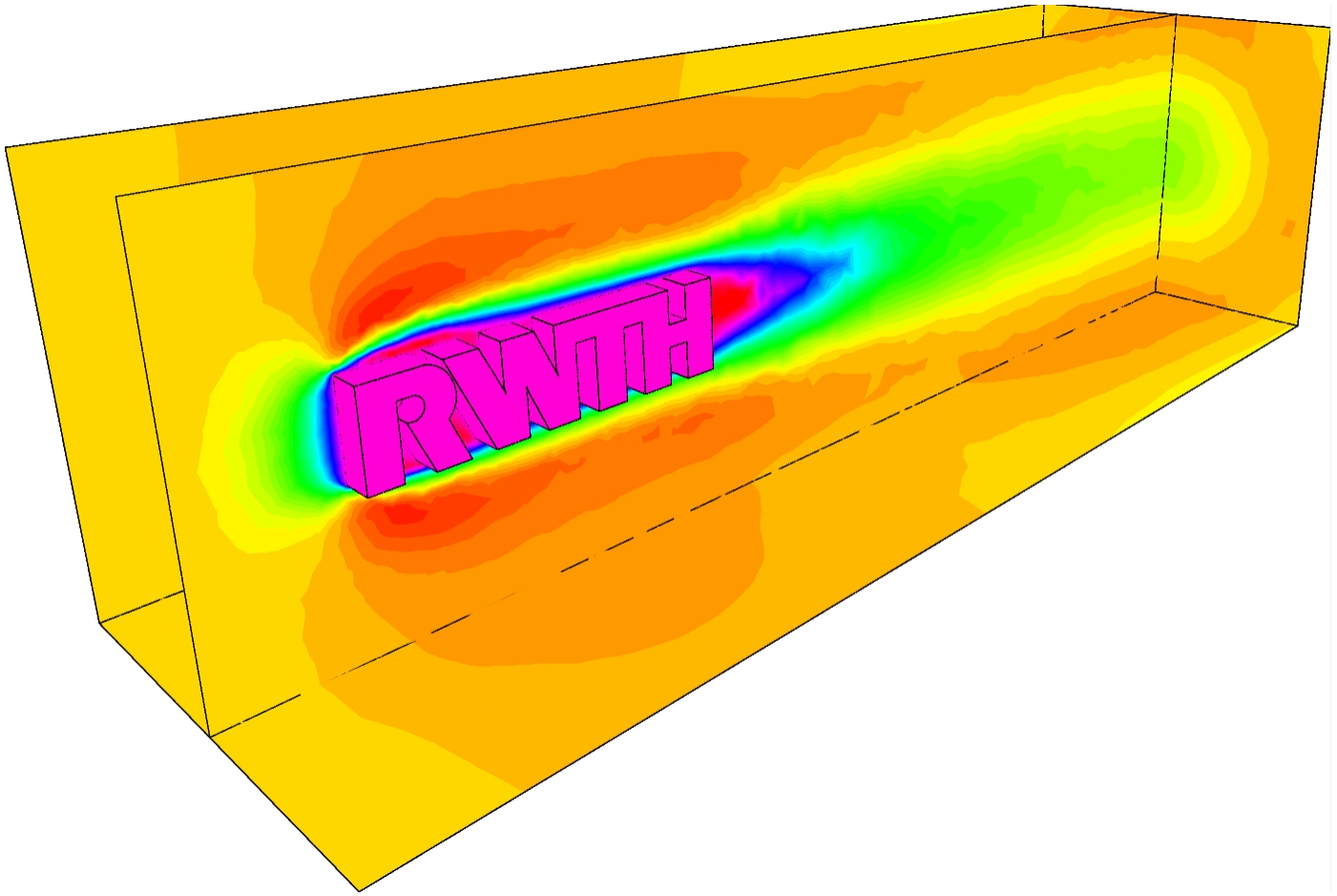
Parallel Computing for Simulation Sciences: Project 2

Jaeyong Jung (359804)

June 19, 2016



1. Comparing graph from interpolation method and original graph from coarse mesh.

(a) coarse mesh (b) fine mesh

Figure 1: Comparing graph of coarse and fine mesh

2. Description about how the program works

My program is designed to minimize iterative steps for finding corresponding coarse element of a fine node. The most outer loop is set to be parallel. That is because it is directive to distribute every fine nodes to every thread. Also, by parallelizing the most outer loop, overhead to wake up every thread can be prevented. If inner loop is parallel, threads have to be waken up every time when parallel loop is called. Also, inner loop is modified to minimize the number of iterations to search coarse elements with two tolerances.

At the beginning of for(l) loop, finding coarse element, the program starts finding elements with 0.2 tolerance. If it finds an element with 0.2 tolerance, it saves interpolated data on data_fine and try 0.0 tolerance right after that. flag_tol_l is set to be 1 since interpolation is done with larger tolerance. Nevertheless, the program still keep searching element with 0.0 tolerance until it finds an element with 0.0 tolerance. If an element for interpolation with 0.0 tolerance is found, interpolated data are saved on data_fine and flag_tol_s is set to be 1 and break from the loop. At last, those two flags are initialized and go for another iteration. This idea is summarized below.

- The case of the element with 0.0 tolerance to exist. Computer searches element with 0.2 tolerance firstly. If it finds an element with 0.2 tolerance, it also tries interpolation with 0.0 tolerance. If it is hit, breaks the loop after saving interpolated values. Otherwise, just keep searching on.
- On the other hand, the element with 0.0 tolerance do not exist. Computer tries interpolation with 0.2 tolerance. At some point, interpolation will be done and computer will still keep searching for. Finally, computer saves an interpolation with 0.2 tolerance.

Implementation of parallel algorithm is done on main function. C code:

```
#pragma omp parallel for firstprivate(tol, tol2, flag_tol_l,
    flag_tol_s) private(k, l, m, xx, yy, zz, xi_eta_zeta, xe, ye,
    ze, coarse_data) schedule(static)
        for (i = 0; i < nn_fine; i++) {
                //printf("Working on i= %d\n",i);
```

```
xx = mxyz_fine[i][0];
yy = mxyz_fine[i][1];
zz = mxyz_fine[i][2];
{ // Loop over degrees of freedom and interpolate,
    ndf = 4
  // TODO: Interpolate the data_fine[i][k]. This
     is the k-th degree of freedom of the i-th
     node.
  // In this project the number of degrees of
     freedom is 1. Use for interpolation the
     function
  // YOUR CODE STARTS HERE

for (l = 0; l < ne_coarse; l++) {
        // Put element node's coordinates
        for (m = 0; m < NEN; m++) {
                xe[m] = mxyz_coarse[mien_coarse[l
                   ][m]][0];
                ye[m] = mxyz_coarse[mien_coarse[l
                   ][m]][1];
                ze[m] = mxyz_coarse[mien_coarse[l
                   ][m]][2];
        }

if (flag_tol_l == 0)
{
if (check_with_tolerance(xi_eta_zeta, xe, ye, ze,
   xx, yy, zz, tol2) == 1)
{
        for (k = 0; k < ndf; k++) {
                for (m = 0; m < NEN; m++) {
                coarse_data[m] = data_coarse[
                   mien_coarse[l][m]][k];
}
                data_fine[i][k] = interpolate_data
                   (xi_eta_zeta, coarse_data);

}
                flag_tol_l = 1;
}
} // We have interpolation with larger tolerance
   but need to go for more iteration with 0
   tolerance.

if (flag_tol_l == 1) // We got that tolerance 0.2
```

```
                    is accepted in a specific case. Still, we neeed
                     to narrow it down with 0 tolerance.
                {
                if (check_with_tolerance(xi_eta_zeta, xe, ye, ze,
                   xx, yy, zz, tol) == 1)                          {
                for (k = 0; k < ndf; k++)
                {
                for (m = 0; m < NEN; m++)
                {
                                coarse_data[m] = data_coarse[
                                   mien_coarse[l][m]][k];
                }
                data_fine[i][k] = interpolate_data(xi_eta_zeta,
                   coarse_data);
                }
                                flag_tol_s = 1;
                break; // breaking from l loop, since we got
                   interpolation with 0 toleration.
                }
                }
                                                } // end l

                if (flag_tol_l + flag_tol_s == 0) {
                        printf("Impossible to interpolate points")
                           ;
                        exit(-1);
                }

                flag_tol_s = 0;
                flag_tol_l = 0;

        } // end k
} // end i
 // end of parallel loop
```

3. Study on scalability and scheduling

Scalability is investigated based on default "Static" scheduling firstly. The maximum number of threads are 12. After that, "Dynamic" scheduling experiments with varying chunk size have been done to improve scalability. Finally, it is found that it is possible to attain 100% parallel efficiency using "Dynamic" scheduling.

Scalability with respect to number of threads is studied using default "Static" scheduling option.

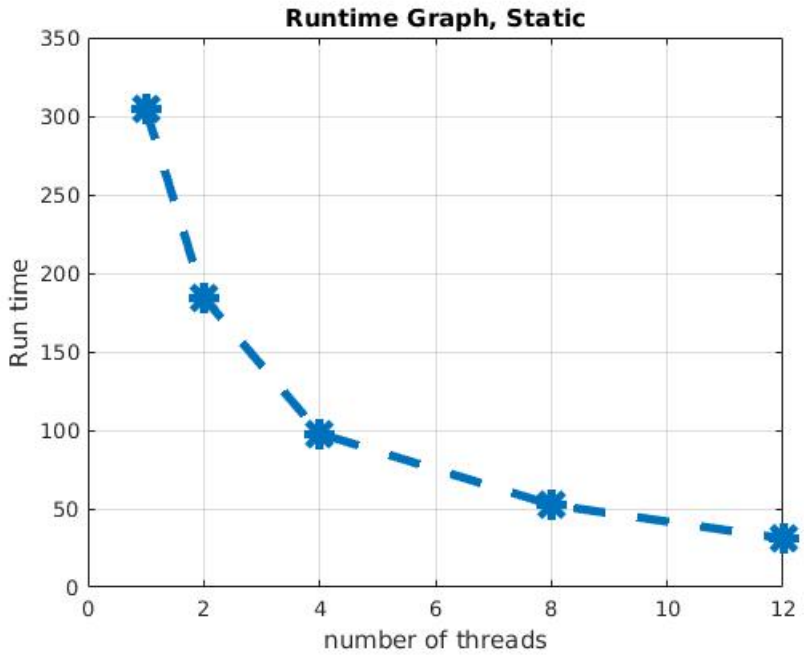


Figure 2: Runtime

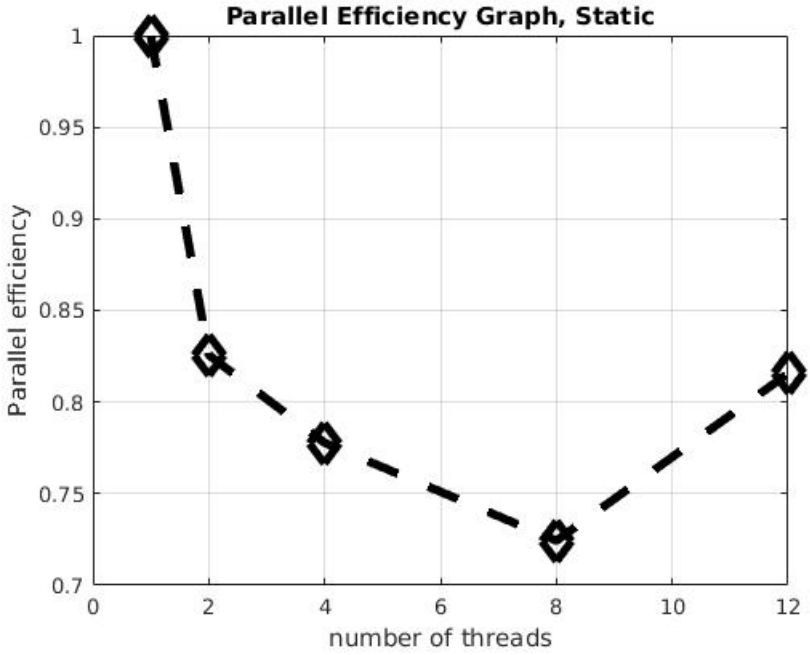


Figure 3: Parallel Efficiency

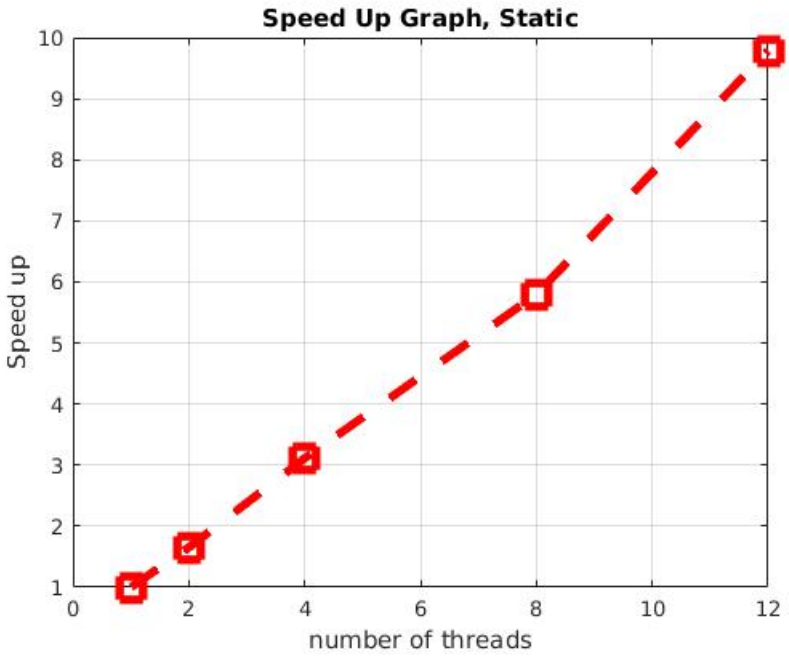


Figure 4: Speed Up

From those three graphs, it is observed that runtime decreases as number of threads increases. From Figure 3, parallel efficiency is below 0.85. This parallel efficiency is good in general. However, considering the most outer loop is parallel so that most of the code is parallel, this efficiency is not remarkable. One of the reason why the parallel efficiency is not so good is that every thread has different amount of computational load. To elaborate, every thread tries to do interpolation on coarse mesh element from first one to last one for every iteration. Depending on the distribution of fine nodes in coarse mesh matrix, each thread can be assigned with coarse elements near to first one or the last one. Since some threads may finish their job early and the others take more time, computational load can be biased. Therefore, it is necessary to randomly distribute computational work loads onto threads to run the program independently to coarse mesh matrix structure. "Dynamic" scheduling is alternative way to distribute computational load equally along threads.

For "Dynamic" scheduling test, number of threads is fixed as 12 and chunk size varies. The first test's chunk size number of fine nodes (110618) over number of threads (12). Based on this, the chunk size is divided with natural numbers for "normalization". Smaller chunk size allows threads to have computational load randomly and equally.

$$\boxed{\begin{array}{c} \frac{nn\_fine}{12 \times 1} \\ \frac{nn\_fine}{12 \times 2} \\ \frac{nn\_fine}{12 \times 3} \\ \frac{nn\_fine}{12 \times 4} \\ \frac{nn\_fine}{12 \times 5} \\ \frac{nn\_fine}{12 \times 6} \\ \frac{nn\_fine}{12 \times 20} \\ \frac{nn\_fine}{12 \times 50} \\ \frac{nn\_fine}{12 \times 100} \\ \frac{nn\_fine}{nn\_fine} = 1 \end{array}}$$

Chunk size

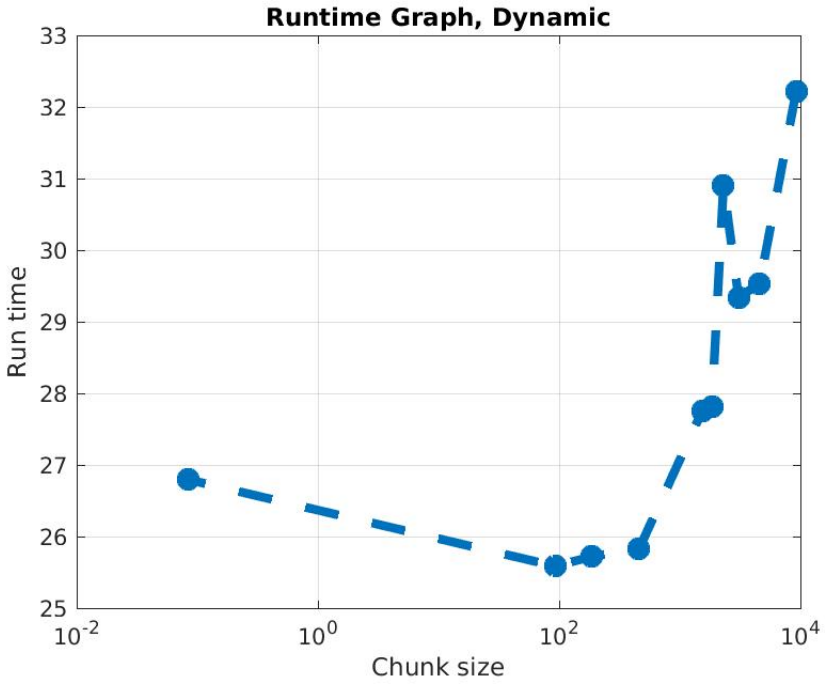


Figure 5: Runtime

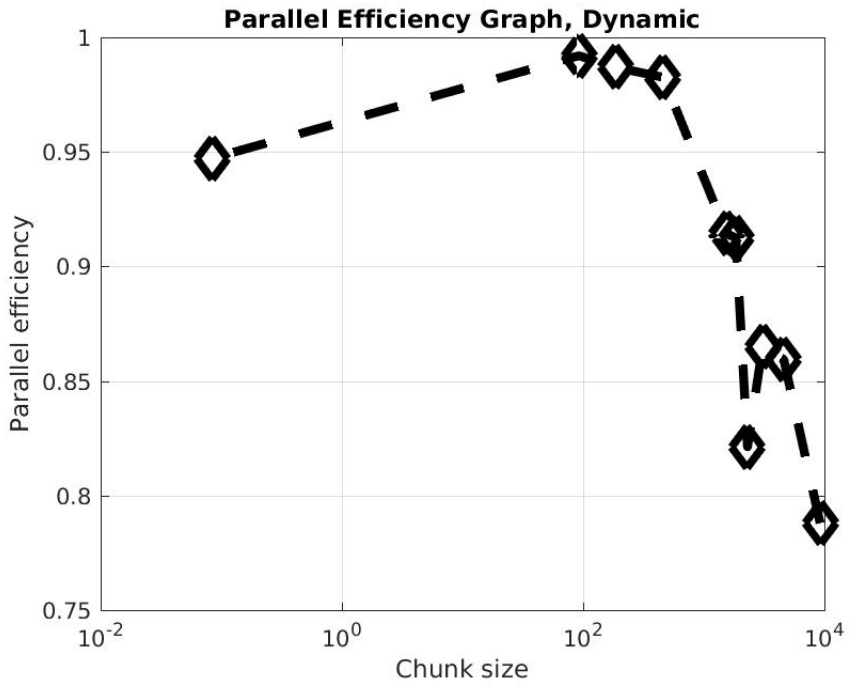


Figure 6: Parallel Efficiency

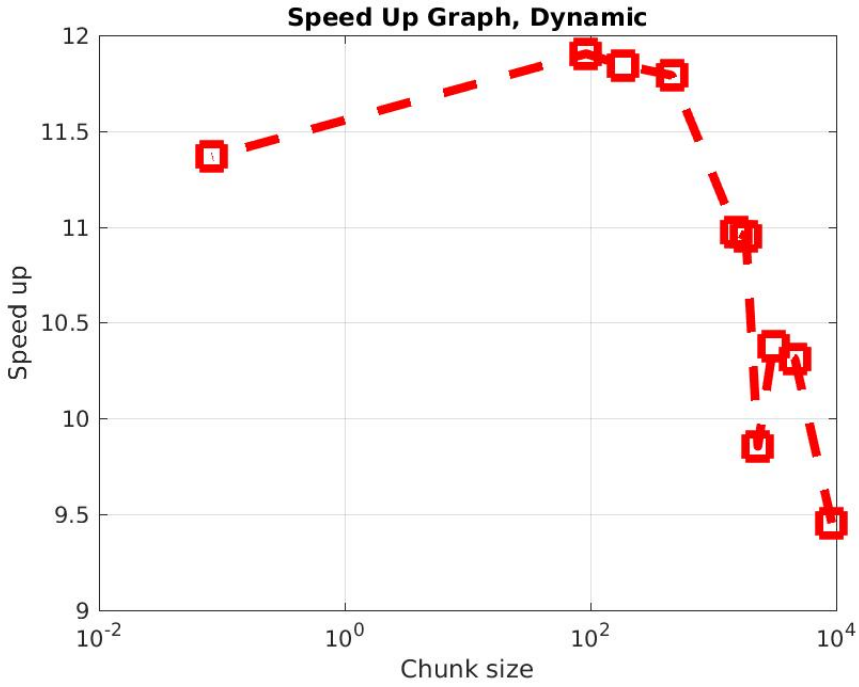


Figure 7: Speed Up

From Figure 6 and 7, the parallel efficiency is the highest when chunk size is $\frac{nn\_fine}{12 \times 100}$. The reason is that smaller chunk size makes threads to prevent having biased computational

load. It is unexpected that when the efficiency of chunk size 1 is a little bit slower than the maximum value. The smallest chunk size does not guarantee maximum parallel efficiency. "Overhead" is another factor that influences this phenomenon. If chunk size is too much small, master thread should allocate new jobs to others quite often. As time is wasted on assigning new jobs to threads, parallel efficiency will not be the best. Nevertheless, this "Dynamic" schedule's default chunk size option yields better computational performance than "Static" default does.

In conclusion, with 12 threads and "Dynamic" schedule, it is possible to make this interpolation program 11.9042 faster, from Figure 7, than it used to be with single thread.