CAAM 520: COMPUTATIONAL SCIENCE II
HOMEWORK 3.

WEI WU

1. Introduction

In this project, we build upon the previous project: we parallelize our linear system solver using OpenMP. This turns out to be significantly easier than using MPI.

1.1. **Partition of Parallel Work.**

Let us begin by looking at the part of the serial code where the actual computation of Jacobi method takes place.

```
while(residual > tol){
// update interior nodes using Jacobi
  for(int i=1; i<=N; ++i){
    for(int j=1; j<=N; ++j){
      /* Compute unew while accessing u and residual */
    }
  }
  /* Update u with unew */
}
```

The conceivable parallel region is inside the while loop, and if fact, we can use loop-parallelism on the double for-loop to achieve speed-up. We hence have the following code snippets.

```
 while(res2>tol*tol){
   res2 = 0.0;
   int i;
   int j;
   // update interior nodes using Jacobi
   #pragma omp parallel for shared(unew) private(i,j) reduction(+: res2)
   for(int i=1; i<=N; ++i){
     for(int j=1; j<=N; ++j){
      const int id = i + j*(N+2); // x-index first
      const double Ru = -u[id-(N+2)]-u[id+(N+2)]-u[id-1]-u[id+1];
      const double rhs = invD*(f[id]-Ru);
      const double oldu = u[id];
      const double newu = w*rhs + (1.0-w)*oldu;
      res2 += (newu-oldu)*(newu-oldu);
      unew[id] = newu;
      }
    }
   for (int i = 0; i < (N+2)*(N+2); ++i){
     u[i] = unew[i];
    }

    ++iter;
    if(!(iter%500)){
      printf("at iter %d: residual = %g\n", iter, sqrt(res2));
    }
  }
```

1.2. **OpenMP Directives Used.** In the code snippet above, I make *unew* and *res*2 a shared variable, since it will be accessed once during each iteration inside the parallel region. Loop indices i, j are set to be private, which is the default of OpenMP. And finally, to avoid data race, the reduction clause is used on variable *res*2 with summation operation.

1.3. **Correctness.** I compare the results of my code with the serial version in homework 1. For any given number of threads, my code finishes with the same number of iterations and reached the same Max error as in the serial version. Full results of the experiment could be found in slurm-7558200.out, or in the tables below.

Table 1. N = 32, tol = 1e-6

Num. Threads	Iter	Max Error	Time(s)
1	529	0.000149701	0.00983288
2	529	0.000149701	0.00761996
4	529	0.000149701	0.00778023
8	529	0.000149701	0.00636196
16	529	0.000149701	0.0144027

Table 2. N = 64, tol = 1e-6

Num. Threads	Iter	Max Error	Time(s)
1	1914	3.29097e-05	0.133607
2	1914	3.29097e-05	0.0907296
4	1914	3.29097e-05	0.0672218
8	1914	3.29097e-05	0.0659042
16	1914	3.29097e-05	0.0614588

TABLE 3. N = 128, tol = 1e-6

Num. Threads	Iter	Max Error	Time(s)
1	6964	3.03937e-06	2.07492
2	6964	3.03937e-06	1.30382
4	6964	3.03937e-06	0.872488
8	6964	3.03937e-06	0.650239
16	6964	3.03937e-06	0.693178

2. Strong Scaling

To test strong scaling of our code, we experiment with different problems sizes with threads 1,2,4,... on NOTS. Below are the results. They correspond to matrix sizes of 1024, 4096, and 16384.

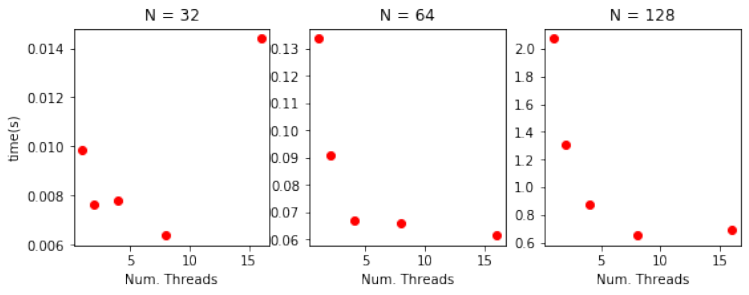

FIGURE 1. Scaling

For all three problem sizes, the time to finish decreases initially. Ideally it would keep decreasing, but the communication overhead will eventually take over if we apply a large number of threads. When the problem size is small, it takes much longer to finish if we apply many threads. For the other two problem sizes, they seem to reach their speedup plateau.