HMW03 Report

Jacob Engel

March 6, 2019

This is the first parallelized section of the code. Omp collapse is being used to take advantage of the fact that there are 2 independant, square forloops being used. This allows to have nested parallelism without having to actually make the nested call.

```
# pragma omp collapse(2)
{
for (int j = 0; j<N+2; j++){
        for (int i = 0; i<N+2; i++){
        int pos = i+j*(N+2);
        u1[pos] = 0;
        ue[pos] = 0;
        double x = a+i*h;
        double y = a+j*h;
        if (i == 0 || j == 0 || i == N+1 || j == N+1)
                b[pos] = 0;
                u[pos] = 0;
        else{
                b[pos] = sin(x*pi)*sin(y*pi);
                u[pos] = 1;
        }
}
}
```

This section of the code actually computes the vector. OMP collapse is being used here in the exact same way as before, looping over the square forloops to avoid having to do nested parallelization.

```
double E = tol+1;
while (E >=tol){
        # pragma omp collapse(2)
        {
        for (int j = 1; j<N+1; j++){
                for (int i = 1; i<N+1; i++){
                        int pos = i + j*(N+2);
                        int lpos = i-1 + j*(N+2);
                        int rpos = i+1 + j*(N+2);
                        int bpos = i + (j-1)*(N+2);
                        int upos = i + (j+1)*(N+2);

                        double sum = 0;
                        sum += (-1/pow(h, 2))*(u[lpos]
                + u[rpos] + u[bpos] + u[upos]);
                        u1[pos] = (w*0.25*(pow(h,2)))*
                (b[pos] - sum) + (1-w)*u[pos];

                        }
                }
                }

                # pragma omp collapse(2)
                {
                for (int j = 1; j<N+1; j++){
                        for (int i = 1; i<N+1; i++){
                          int pos = i + j*(N+2);
                          int lpos = i-1 + j*(N+2);
                          int rpos = i+1 + j*(N+2);
                          int bpos = i + (j-1)*(N+2);
                          int upos = i + (j+1)*(N+2);
                          ue[pos] = fabs((1/(h*h))*
                          (-1*(u[lpos]+u[rpos]+u[bpos]
+u[upos]) + 4*u[pos])-b[pos]);

                        }
                }
                }
                # pragma omp critical
                E = ue[0];
                for (int i = 1; i<pow(N+2,2); i++){
                        if (ue[i] > E){
                                E = ue[i];
                        }
                        u[i] = u1[i];
                }

                iter++;
        }
```

This code was run on the cpu for a 4x4 system and yielded the same results on the same number of iterations as the serial code for the 4x4 system. They both took around 10e-5 seconds and took 158 iterations to get below the residual threshold of 10e-6.

```
#!bin/bash
#SBATCH --job-name=Myjob
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#SBATCH --time=00:10:00
```

For the cpus-per-task command, it changes depending on how many cpus are being used for the given executable.

The columns of this table show the time required to run on NOTS. Each column is a mesh size (10,50,100) and they were run on 1, 2, 3, and 4 cpus on NOTS.

	1	2	3	4	% difference betweeen 4 and 1 cpus
n = 10	5.62e-3	5.19e-3	5.18e-3	5.11e-3	9.62
n = 50	2.618	2.584	2.274	2.285	12.3
n = 100	36.510	35.775	36.432	35.469	0.112

This code barely scaled at all. The more cpus there were, generally, the less time it took, but the change was very minimal, and even though 4 CPUs always took faster than 1 cpu, there were instances where the code ran on one more cpu and took longer. An example of this is for when n=50, 3 CPUs took 9/1000 seconds shorter than 4 CPUs. However, this code is correct, despite its apparent costliness. The errors were the same for each number of interior points, and the iterations needed were the same as well.