Contents

1 Deliverable 5

1.1 Gauss Seidel

1.1.1 Parallelization Strategy

To make our code parallel we observe that we can use a block decomposition strategy, as each iteration of the original loop only depends on the modified (i-1,j), (i,j-1) squares, and the unmodified (i+1,j) and (i, j+1) squares. Hence if we divide the computing space into blocks we naturally generate dependencies on the left and upper block. Hence our code will look like this:

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
    double unew, diff, sum=0.0;
    int howmany=8;
    #pragma omp parallel for ordered(2) private(unew, diff) reduction(+:sum)
    for (int blockid = 0; blockid < howmany; ++blockid) {
        for (int blockid1 = 0; blockid1 < howmany; ++blockid1) {
            int i_start = lowerb(blockid, howmany, sizex);
            int i_end = upperb(blockid, howmany, sizex);
            int j_start = lowerb(blockid1, howmany, sizey);
            int j_end = upperb(blockid1, howmany, sizey);
            #pragma omp ordered depend(sink: blockid-1,blockid1)  \
            depend(sink: blockid,blockid1-1)
            for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                for (int j=max(1, j_start); j<= min(sizey-2, j_end); j++) {
                    unew= 0.25 * ( u[ i*sizey + (j-1) ]+  // left
                                   u[ i*sizey + (j+1) ]+  // right
                                   u[ (i-1)*sizey + j ]+  // top
                                   u[ (i+1)*sizey + j ]); // bottom
                    diff = unew - u[i*sizey+ j];
                    sum += diff * diff;
                    u[i*sizey+j]=unew;
                }
            }
            #pragma omp ordered depend(source)
        }
```

```
    }

    return sum;
}
```

We have divided the blocks into equal parts per thread, and specified the dependencies of each block. if we run this changes and use a diff to compare the output of this code and the sequential one we can verify it is indeed correct.

In addition, to properly synchronize each thread we have simply used the reduction clause.

1.1.2 Scalability

To check the scalability of this problem we can check the following figure: We observe that although the speedup is higher, it is not as close to the perfect parallelization as we would like, this is due to the fact that this problem is not perfectly parallelizable, hence it doesn't scale up to directly proportional to the number of threads.

To check for the optimal blocksize we have simply binary searched over the answer, we see that if divide it into too many blocks we have too much of an overhead on syncing and forking, whereas if we use a little amount of blocks, due to our dependency graph we cannot fully use our parallelization potential. Hence after taking the mean of several executions having eliminated extreme value we observe that the optimal blocksize for 8 threads is into 8*8 blocks.

1.2 Optional

Instead of using the for ordered directive we can also use the task directive we have seen in previous labs. The result is the following:

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;
    int howmany=omp_get_max_threads();
    int dep[howmany+1][howmany+1];
    #pragma omp parallel
    #pragma omp single
    for (int blockid = 0; blockid < howmany; ++blockid) {
        for (int blockid1 = 0; blockid1 < howmany; ++blockid1) {
```

```
            int i_start = lowerb(blockid, howmany, sizex);
            int i_end = upperb(blockid, howmany, sizex);
            int j_start = lowerb(blockid1, howmany, sizey);
            int j_end = upperb(blockid1, howmany, sizey);
            #pragma omp task depend(in: dep[blockid][blockid1+1], dep[blockid+1][block
            for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
                double = 0;
                for (int j=max(1, j_start); j<= min(sizey-2, j_end); j++) {
                    unew= 0.25 * ( u[ i*sizey + (j-1) ]+  // left
                                   u[ i*sizey + (j+1) ]+  // right
                                   u[ (i-1)*sizey + j     ]+  // top
                                   u[ (i+1)*sizey + j     ]); // bottom
                    diff = unew - u[i*sizey+ j];
                    aux += diff * diff;
                    u[i*sizey+j]=unew;
                }
                #pragma atomic
                sum += aux;
            }
        }
    }

    return sum;
}
```

We again observe that it generates the same output data as the sequantial version.