ID1217 Project report - Oliver S. Lehtihet

PDE Solver

Introduction

The problem was to solve Laplace partial differential equations by developing an efficient parallel program which uses single and multi- grid computations. This can be done by updating a two dimensional matrix based on some initial state and update instructions. These updates were done by using Jacobi and multigrid iterative techniques. After the final stage of the program has been reached, the maximum difference between the two grids used should be calculated. To understand whether the solution was efficient or not, several timing experiments were conducted and the speedup was studied. The report will show that an almost optimal speedup was achieved for large grid sizes, but for smaller ones the overhead of parallelism was too costly.

Programs

Four programs were developed to solve this problem but all of them in some way rely on Jacobi iterations. In short, Jacobi iterations work by going over two grids: “new” and “grid”. For each element e = (x,y) of “new”, new[x, y] = (grid[x-1, y] + grid[x+1, y] + grid[x, y-1] + grid[x, y+1]) * 0.25. Then the process is repeated except where “new” and “grid” have changed places. This is done a specified numIters amount of times.

Program 1, sequential Jacobi:
“new” and “grid” are both initialized with border values = 1 and interior values = 0. Then the simple Jacobi iteration function described earlier is called and finally the maximum difference in the matrix is calculated by simply iterating over all elements in the matrices and returning the largest measured difference between the two matrices' respective elements. The program is optimized slightly by multiplying the sum described earlier with 0.25 instead of dividing it by 4. This is better since multiplication is faster than division.

The following picture show the code for the Jacobi iteration update and the maximum difference calculation function.

```
double jacobi() {

  for (int n = 0; n < numIters; n++) {

    // Compute new values for all interior points
    for (int i = 1; i < gridSize-1; i++) {
      for (int j = 1; j < gridSize-1; j++) {
        new[i][j] = (grid[i-1][j] + grid[i+1][j] + grid[i][j-1] + grid[i][j+1]) * 0.25;
      }
    }

    // Compute values gain for interior points
    for (int i = 1; i < gridSize-1; i++) {
      for (int j = 1; j < gridSize-1; j++) {
        grid[i][j] = (new[i-1][j] + new[i+1][j] + new[i][j-1] + new[i][j+1]) * 0.25;
      }
    }


  }

  // Compute the maximum difference
  double *maxdiff = malloc(sizeof(double));
  *maxdiff = 0;
  double diff = 0;

  for (int i = 1; i < gridSize-1; i++) {
    for (int j = 1; j < gridSize-1; j++) {
      diff = fabs(grid[i][j]-new[i][j]);

      if (diff > *maxdiff) *maxdiff = diff;

    }
  }

  return *maxdiff;

}
```

Program 2 parallel Jacobi:
This program is structured the same as program 1. The only difference is that optimizations have been made by adding parallelization support via the OpenMP library. All for-loops have been optimized this way as can be seen in the following picture which again displays the jacobi and max diff function. Note the pragma declaration before each for loop.

```
double jacobi() {

  int i, j;
  for (int n = 0; n < numIters; n++) {

    #pragma omp parallel for private(j)
      for (i = 1; i < gridSize-1; i++) { // Compute new values for all interior points
        for (j = 1; j < gridSize-1; j++) {
          new[i][j] = (grid[i-1][j] + grid[i+1][j] + grid[i][j-1] + grid[i][j+1]) * 0.25;
        }
      }
      // Have to wait for all threads to terminate so grid isn't
      // updated while being used in the previous for loop

    #pragma omp parallel for private(j)
      for (i = 1; i < gridSize-1; i++) { // Compute values gain for interior points
        for (j = 1; j < gridSize-1; j++) {
          grid[i][j] = (new[i-1][j] + new[i+1][j] + new[i][j-1] + new[i][j+1]) * 0.25;
        }
      }

  }

  // Compute the maximum difference
  double *maxdiff = malloc(sizeof(double)); // To be returned
  *maxdiff = 0;
  double diff;
  int id;
  double maxList[numWorkers];

  #pragma omp parallel private(diff, j, id)
  {
    id = omp_get_thread_num();
    maxList[id] = 0;
    diff = 0;

    #pragma omp for
      for (i = 1; i < gridSize-1; i++) {
        for (j = 1; j < gridSize-1; j++) {
          diff = fabs(grid[i][j]-new[i][j]);

          if (diff > maxList[id]) maxList[id] = diff;

        }
      }
  }

  for (i = 0; i < numWorkers; i++) {
    if (maxList[i] > *maxdiff) *maxdiff = maxList[i];
  }
```

Program 3 sequential multigrid Jacobi:
The multigrid program is a bit different from the previous single grid ones. First, instead of one “new” and “grid”, we create four with different sizes. Given a base grid size ‘x’, the next

grid size is 2x+1, the one after 2(2x+1)+1 and so on. Using a “V-cycle” described by G. Andrews with four levels, we start by doing Jacobi iterations on the finest grid (largest grid size) with a total of four iterations, then the 2nd finest grid is initialized using the finest grids’ values by a special restriction function which. The restriction function takes, for each (x,y) in the grid to be initialized, the sum of half the value of the finer grid at the same place (x,y), and 1/8th of each element on the finer grid directly to the sides of (x,y).

After this is done, the 2nd finest grid (which we just initialized) goes through the same process, where it computes four jacobi iterations and then is used to restrict the 3rd finest grid etc. Until finally the coarsest grid is reached. At that point, it goes through numIters Jacobi iterations and is then used to “interpolate” back to the next finer grid. And the “restriction followed by a few Jacobi-iterations” process is done “in reverse” so we eventually get back to the finest grid.

The following pictures show the restriction, interpolation, and parts of the main function.

```
void restriction(double** fine, double** coarse, int gridSize) {
  // Restrict from fine (large size) to coarse (smaller size). gridSize is of the course grid
  int i, j;
  for (int x = 1; x < gridSize-1; x++) {
    i = x*2;
    for (int y = 1; y < gridSize-1; y++) {
      j = y*2;
      coarse[x][y] = fine[i][j]*0.5 + (fine[i-1][j] + fine[i][j-1] + fine[i][j+1] + fine[i+1][j])*0.125;
    }
  }
}
void interpolation(double** fine, double** coarse, int gridSizeF, int gridSizeC) {
  int i, j;
  // First, assign coarse grid points to corresponding fine grid points
  for (int x = 1; x < gridSizeC-1; x++) {
    i = x*2;
    for (int y = 1; y < gridSizeC-1; y++) {
      j = y*2;
      fine[i][j] = coarse[x][y];
    }
  }

  // Second, update the other fine grid points in columns that were just updated
  for (i = 2; i < gridSizeF-1; i += 2) {
    for (j = 1; j < gridSizeF-1; j += 2) {
      // Start at column 1 instead of 2 (y=1; j=y*2) as in the first update
      fine[i][j] = (fine[i-1][j] + fine[i+1][j])*0.5;
    }
  }

  // Finally, update the rest of the fine grid points
  for (i = 1; i < gridSizeF-1; i++) {
    for (j = 2; j < gridSizeF-1; j += 2) {
      fine[i][j] = (fine[i][j-1] + fine[i][j+1])*0.5;
    }
  }
}
```

And the relevant main() part:

```
jacobi(grid1h, new1h, gs4, fineIters);
restriction(grid1h, grid2h, gs3);
jacobi(grid2h, new2h, gs3, fineIters);
restriction(grid2h, grid4h, gs2);
jacobi(grid4h, new4h, gs2, fineIters);
restriction(grid4h, grid8h, gridSize);
jacobi(grid8h, new8h, gridSize, numIters);

interpolation(grid4h, grid8h, gs2, gridSize);
jacobi(grid4h, new4h, gs2, fineIters);
interpolation(grid2h, grid4h, gs3, gs2);
jacobi(grid2h, new2h, gs3, fineIters);
interpolation(grid1h, grid2h, gs4, gs3);
jacobi(grid1h, new1h, gs4, fineIters);
```

Program 4 parallel multigrid Jacobi:
Much like the difference between program 1 and program 2, program 3 and program 4 share the same structure. The difference is that program 4 has had OpenMP functionality added to make the program support parallelism by adding OpenMP pragma directives before for loops.

Performance evaluation
Since I had problems connecting to the RHEL6 servers at KTH, I decided to use my own computer instead since it is a multicore computer. I ran the program inside a VM running Ubuntu 20.04.4 with 2GB of RAM allocated and access to 4 cores, with an AMD Ryzen 7 5800H.

Each input set was tested 5 times and the median time of each program was selected for the evaluation.

Single grid:
First the number of iterations was determined that gave a running time of ~30s. The following table shows the time for the sequential Jacobi iteration program (Program 1):

grid size	iterations	time (seconds)
100	400k	28.5
200	100k	28.5

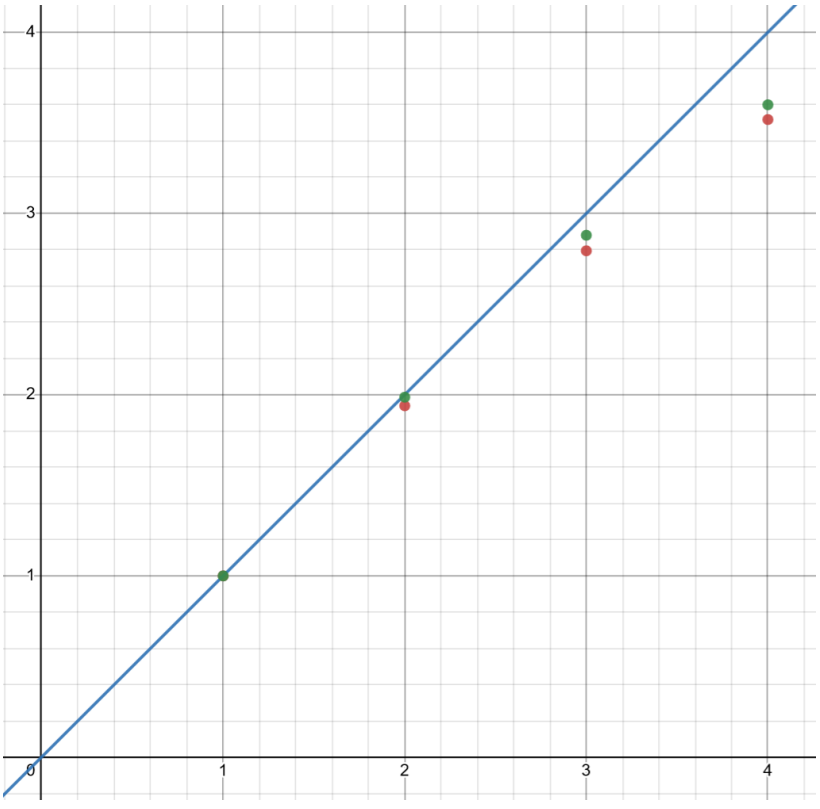
The parallel program (Program 2) was tested with four different workers:

grid size	iterations	workers	time (seconds)
100	400k	1	28.5
		2	14.7

		3	10.2
		4	8.1
200	100k	1	28.8
		2	14.5
		3	10.0
		4	8.0

Speedup is determined by the time it takes to run the program sequentially divided by the time it takes to run the same program in parallel. An optimal linear speedup means the speedup factor is equal to the number of workers. In other words, having 2 workers halves the time of the sequential program, and 10 workers gives an execution time of only 1/10th the sequential etc.

The following graph shows: an optimal linear speedup (blue line), and the actual speedup for both grid sizes (red=grid 100, green=grid 200). The x-axis represents the number of workers and the y-axis is the speedup factor:



We can see that both grid sizes are very close to optimal, however as the number of workers increases we get further away from optimal speedup. This is likely the result of the overhead required to create and maintain the threads. We also see that the larger grid size is closer to optimal than the smaller one, which is likely because creating a new thread takes time, and if the grid size is larger, then the amount of time it takes to create a new thread is going to be less as a portion of the total time it takes to go through the grid. As an example, if it takes 1

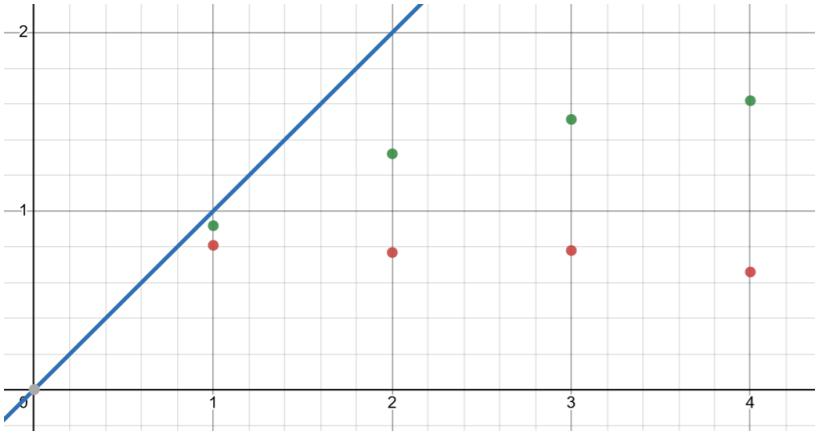
second to create a thread, but only 2 seconds to go through the grid then the speedup is going to be smaller than if it takes 1 second to create a thread, and 100 seconds to go through the grid.

Multi grid:

grid size	iterations	time (seconds)
12	30m	32.4
24	7m	28.0

grid size	iterations	workers	time (seconds)
12	30m	1	40.1
		2	42.2
		3	41.6
		4	49.2
24	7m	1	30.5
		2	21.2
		3	18.5
		4	17.3

The same type of graph as shown in the earlier comparison with green=24 grid size and red = 12:



We see that the smaller grid size has a speedup less than 1, meaning it slows down compared to the sequential version. And further, it slows down more if more workers are added. This is likely explained by the same reasoning for why a grid size of 200 had a larger speedup than 100 in the comparison between program 1 and 2. That the small grid size does not justify the overhead cost of creating and maintaining a bunch of threads. In fact the overhead seems so great that it just slows down the program. While the grid size of 24 does show a speedup, it is clearly far below the optimal.

Conclusion

The report has shown that parallelizing this type of program is easy to do and can lead to very good speedup. However you have to carefully consider whether the problem is large enough to where multiple threads are justified, and that the overhead costs aren't too great. I have learned that you can never assume parallelizing a program is going to lead to any kind of positive speedup, instead the scale of each program has to be considered.