ID1217 project report

Erik Hanstad, Lucas Villaroel

March 2021

1 Introduction

The project is to develop and to evaluate efficient parallel programs for a grid computation to solve partial differential equations (a.k.a. PDE solvers), conduct timing experiments to analyze the performance of the written programs. Laplace's equation is an example of two-dimensional partial differential equation. The iterative techniques used are Jacobi and multigrid.

The programs are written in c with the OpenMP library for the parallel programs. This report will on a deeper level explain how the programs work, evaluate the performance of the programs and analyze the results.

2 Programs

In this section the different programs will be described.

2.1 Jacobi sequential

The sequential program takes two parameters, grid size and number of iterations. Allocations in the memory for the grids, represented as two double pointers, are made using the malloc system call. The next step is to initialize the grids by putting ones on the boundary points and the rest as zeroes. The program then calls the function the group named jacobi, it takes parameters of the two grids. In a tripple loop, where the outermost goes from 1 and adds 2 to the number iterations chosen by the user. The second loop represents the grids´ rows and third the third goes through each element in that row excluding the boundary points and uses the jacobi formula T[i,j] = (G[i,j-1] + G[i-1,j] +G[i+1,j] + G[i,j+1]) / 4 for both of the grids. When the grids are calculated a new function is called which calculates the maximum difference between the two grids. By going through all the elements again excluding the boundary points and takes the absolute value of the difference and ether saves it, if it's lager than the saved value, or keeps the old value. Then the program prints the resulting grid to a new file and the execution time and maximum difference on the terminal.

```
int boundary = gridSize - 1;
  for (int iter = 1; iter < numIters; iter++) {
    for (int i = 1; i < boundary; i++) {
      for (int j = 1; j < boundary; j++) {
        new[i][j] = (grid[i - 1][j] + grid[i + 1][j] + grid[i][j -
    1] + grid[i][j + 1]) * 0.25;
      }
    }
    for (int i = 1; i < boundary; i++) {
      for (int j = 1; j < boundary; j++) {
        grid[i][j] = (new[i - 1][j] + new[i + 1][j] + new[i][j - 1]
     + new[i][j + 1]) * 0.25;
      }
    }
```

2.2 Jacobi parallel

The parallel and the sequential are very similar except from a few but important aspects. The first one being the boilerplate code for using OpenMP. The parallel program takes one more parameter in addition to those stated in the sequential section the number of workers. As stated in the assignment the second for-loop in the jacobi function representing the grids´ rows are divided among the workers and calculated simultaneously. The program uses the OpenMP's 'pragma omp parallel for' and moves the integers i and j used in the for-loops outside the for-loop and explicitly flags j as private. So i is shared among all threads but j is private. The same thing is done on the maxDiff function. The only other difference between the parallel and sequential is the timer, the parallel program uses OpenMP´s built-in timer.

```
int i, j;
  int boundary = gridSize - 1;
  for (int iter = 1; iter < numIters; iter++) {
#pragma omp parallel for private(j)
    for (i = 1; i < boundary; i++) {
      for (j = 1; j < boundary; j++) {
        new[i][j] = (grid[i - 1][j] + grid[i + 1][j] + grid[i][j -
    1] + grid[i][j + 1]) * 0.25;
      }
    }
#pragma omp parallel for private(j)
    for (i = 1; i < boundary; i++) {
      for (j = 1; j < boundary; j++) {
        grid[i][j] = (new[i - 1][j] + new[i + 1][j] + new[i][j - 1]
     + new[i][j + 1]) * 0.25;
      }
    }
```

2.3 Multigrid sequential

The sequential program takes two parameters from the user, the of the smallest grid and the number of iterations. Then it calculates the sizes of the four different grids. The second courest grid is the size of the courset grid times

two plus one and so on. And since the group already implemented Jacobi, that will be relaxation method. The grids then gets initialized by allocating memory and giving boundary points. The group follows the steps on page 550 in [1] Foundations of multithreaded, parallel, and distributed programming. So the program starts with the finest grid and updates points for a few iterations using Jacobi. The program then restricts the results to a coarser grid. The program does this by calling a function restrictGrid. This function goes through all points of the smaller of the grids and calculates the corresponding position and updates the coarsepoints as follows: *coarse[i][j] = fine[x][y] * 0.5 + (fine[x - 1][y] + fine[x][y - 1] + fine[x][y + 1] + fine[x + 1][y]) * 0.125*

```
void restrictGrid(double** fine, double** coarse, int coarseSize) {
  int x, y, i, j;
  for (i = 1; i <= coarseSize; i++) {
    x = i * 2;
    for (j = 1; j <= coarseSize; j++) {
      y = j * 2;
      coarse[i][j] = fine[x][y] * 0.5 + (fine[x - 1][y] + fine[x][y
     - 1] + fine[x][y + 1] + fine[x + 1][y]) * 0.125;
    }
  }
}
```

Then takes the second finest and repeats the same steps until all four have been updated. After that the program interpolates the coarse grid back to the finest grid using the V cycle, illustrated on page 552 in Foundations of multithreaded, parallel, and distributed programming [1]. The program calls the function interpolate. This function assign the coarse grid points to the corresponding fine grid points, updates the fine grid points in columns that were updated and lastly updates the grid points that are left.

```
  int x, y;
  // assign coarse grid points to corresponding fine grid points
  for (int i = 1; i <= coarseSize; i++) {
    x = i * 2;
    for (int j = 1; j <= coarseSize; j++) {
      y = j * 2;
      fine[x][y] = coarse[i][j];
    }
  }


  // update fine grid points in columns that were updated
  for (int i = 2; i <= fineSize; i += 2) {
    for (int j = 1; j <= fineSize; j += 2) {
      fine[i][j] = (fine[i - 1][j] + fine[i + 1][j]) * 0.5;
    }
  }

  // update other fine grid points
  for (int i = 1; i <= fineSize; i++) {
    for (int j = 2; j <= fineSize; j += 2) {
      fine[i][j] = (fine[i][j - 1] + fine[i][j + 1]) * 0.5;
    }
  }
```

2.4 Multigrid parallel

The differences between the sequential and the parallel program for multigrid are just as Jacobi very similar. Instead of using the sequential version of Jacobi we use the parallel version. In the functions restrictGrid and interpolate the program uses *#pragma parallel for* and sets j, x and y as private and for the ones only using i and j, j is set as private.

3 Performance Evaluation

The programs were evaluated on two different computers both running with Intel processors. The results can be seen in figure 2, 4, 1 and 3. Each evaluation ran the program with each input parameter 5 times after which the median execution time was chosen.

3.1 Jacobi

When looking at the benchmarks it's obvious the parallel program performs better than the sequential version. Furthermore the efficiency in the parallel program is increased when more workers were added, this is especially apparent in the larger matrix where the time is virtually halved when using 4 workers.

gridSize	**iterations**	**time(ms)**
100	500 000	31536
200	125 000	31749

(a) Sequential Jacobi

gridSize	**iterations**	**workers**	**time(ms)**
100	500 000	1	32720
100	500 000	2	18623
100	500 000	3	22614
100	500 000	4	17376
200	125 000	1	32360
200	125 000	2	17309
200	125 000	3	22079
200	125 000	4	16879

(b) Parallel Jacobi

Figure 1: Performance evaluation Jacobi (Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 4 course and 2 threads per core)

gridSize	iterations	time(ms)
100	250 000	28812
200	60 000	27787

(a) Sequential Jacobi

gridSize	iterations	workers	time(ms)
100	250 000	1	30655
100	250 000	2	19525
100	250 000	3	16318
100	250 000	4	14705
200	60 000	1	29437
200	60 000	2	18323
200	60 000	3	14760
200	60 000	4	12935

(b) Parallel Jacobi

Figure 2: Performance evaluation of Jacobi (Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 8 cores 2 threads per core)

3.2 Multigrid

The multigrid program is a bit more complex when it comes to performance. Since the program uses four levels where each level has a grid of a different size, the performance could fluctuate depending on which level the program is doing calculations for. For example, the smallest grid could be a bottleneck for the parallel performance, while the larger grids should generally perform better. This can be seen in the results of the performance test on both computers, for the program where the smallest grid is 12 the parallel program performs poorly. Also, for this grid, adding workers increases the execution time. This is likely due to the overhead of running concurrently being proportionally bigger than the benefits gained when performing calculations on very small grids.

gridSize	iterations	time(ms)
12	12 500 000	31033
24	3 000 000	29089

(a) Sequential Multigrid

gridSize	iterations	workers	time(ms)
12	12 500 000	1	39683
12	12 500 000	2	49393
12	12 500 000	3	49763
12	12 500 000	4	51673
24	3 000 000	1	26978
24	3 000 000	2	20874
24	3 000 000	3	23798
24	3 000 000	4	20956

(b) Parallel Multigrid

Figure 3: Performance evaluation Multigrid (Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 4 course and 2 threads per core)

gridSize	iterations	time(ms)
12	15 000 000	32139
24	4 000 000	33652

(a) Sequential multigrid

gridSize	iterations	workers	time(ms)
12	15 000 000	1	41777
12	15 000 000	2	55457
12	15 000 000	3	55980
12	15 000 000	4	66885
24	4 000 000	1	31826
24	4 000 000	2	27530
24	4 000 000	3	25803
24	4 000 000	4	27189

(b) Parallel multigrid

Figure 4: Performance evaluation of multigrid (Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 8 cores 2 threads per core)

4 Conclusion

The report has analyzed the implementation and parallelization of two PDE-solvers for the Laplace's equation. The two methods used in the programs were the Jacobi method and the multigrid method. The report has shown that a Jacobi program can be effectively parallelized using OpenMP and for-loop-parallelism. The parallelization of the multigrid gave some performance benefits, although not as substantial as the performance benefit in the Jacobi implementation.

This project has provided the authors with deeper knowledge regarding concurrent programming with OpenMP. As well as how to analyze and evaluate concurrent programs with regards to workload and number of workers.

References

[1] G. R. Andrews, *Foundations of multithreaded, parallel, and distributed programming.* Reading, Mass.: Addison-Wesley, 2000.