HW3
'An Explicit In-Time Solver With Finite Difference Solver Using OpenMP'

Mehmet Şamil Dinçer (2236248)
Elvin Gültekinoğlu (2446169)

6/Dec/2023

1 Abstract

Parallelization is a technique which is utilized in computational applications these days. The main purpose is to divide a complex task into smaller ones to decrease the overall computational time and increase the efficiency accordingly. In that sense, OpenMP is a standard parallel programming API and it is used frequently. This paper represents the use of OpenMP in parallelization of the explicit in-time finite difference solver. With the comprehensive explanation of the methodology used, results for different mesh resolutions are also reflected throughout this paper.

2 Introduction

Parallelization has gained an important place in engineering applications and solving related problems with the increasing complexity of these. Until now, different methods and standards have been offered by researchers, scientific programmers, and engineers. One of these is OpenMP and it is prominent due to its wide use in parallelization. In Chandra and Rohit (2001) [2], the motivation behind its use is explained as the improvement of speed and performance it offers, which is also an important advantage. In another study, Ayguade et al. (2008) [1] define OpenMP as expressive, flexible, and carrying high potential in converting serial solutions into parallel ones.

A serial solver has been prepared previously to tackle the linear advection equation. This solver is now being converted to a parallel one by the use of OpenMP. The main method is the finite difference method in this solver. To perform computations, all steps are utilized in a file called "Advection.c" and it is compiled and run along with an input file. The mesh resolution in this input file is changed to observe the effect of it on performance. Another performance criterion, which is the duration of calculations, is also recorded and evaluated.

In the second part, the infinity norm of the resulting vector is found by parallelization as the case in the main solver. Three different approaches that are adapted can be classified as critical regions, atomic operations, and reductions. These are compared according to the timing of the calculations. These methods have different pros and cons which are specific to the problem considered.

In this paper, parallelization techniques utilized in solvers and calculations are observed and evaluated in terms of performance considering certain criteria.

3 The Theory and Methodology

In this section, the theory behind the solver and parallelization is explained in detail.

3.1 The Solver

The linear advection euqation is given in Equation 1.

$$\frac{\partial q}{\partial t} + \nabla(uq) = 0 \tag{1}$$

To get a clear understanding of this equation, the longer version is given in Equation 2.

$$\frac{\partial q}{\partial t} + \frac{\partial (uq)}{\partial x} + \frac{\partial (vq)}{\partial y} = 0 \tag{2}$$

The crucial part of preparing solver by finite difference method is the determination of mesh system. This has already been done in the previous assignment and it is also given in the code file so it will not be explained here in detail. A sample mesh system can be seen in Figure 1 .

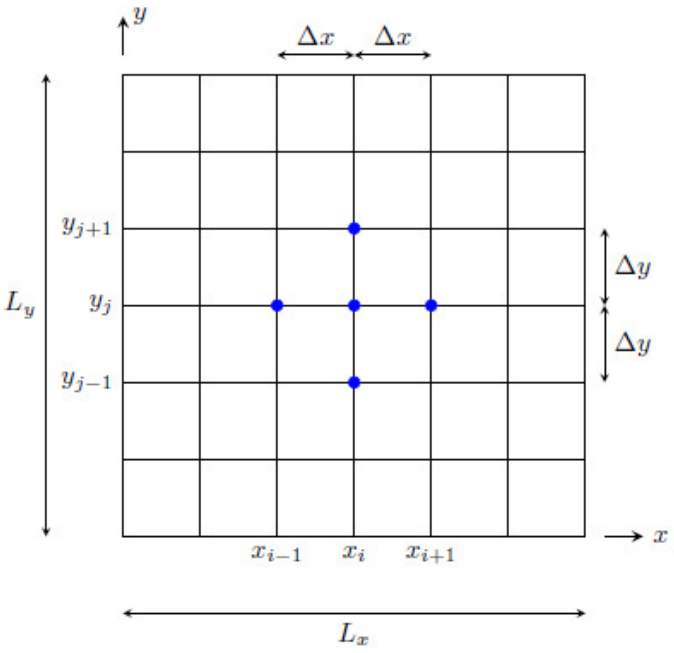

Figure 1: Finite Difference Mesh

Considering the orientation of adjacent nodes, all are expressed in terms of single location instead of two. This is done by utilizing the following equation,

which is Equation 3.

$$n = j \times NX + i \tag{3}$$

After the mesh system is defined in a compact manner, certain parameters are defined in the following manner, as presented in Equation 4 and 5.

$$\frac{\partial(uq)}{\partial x}_{i,j} \approx \begin{cases} \frac{u(i,j)p(i,j)-u(i-1,j)q(i-1,j)}{x(i,j)-x(i-1,j)}, & \text{for } u(i,j) \geq 0 \\ \frac{u(i+1,j)p(i+1,j)-u(i,j)q(i,j)}{x(i+1,j)-x(i,j)}, & \text{for } u(i,j) < 0 \end{cases} \tag{4}$$

$$\frac{\partial(vq)}{\partial x}_{i,j} \approx \begin{cases} \frac{v(i,j)p(i,j)-v(i-1,j)q(i-1,j)}{y(i,j)-y(i-1,j)}, & \text{for } v(i,j) \geq 0 \\ \frac{v(i+1,j)p(i+1,j)-v(i,j)q(i,j)}{y(i+1,j)-y(i,j)}, & \text{for } v(i,j) < 0 \end{cases} \tag{5}$$

Once $\frac{\partial(vq)}{\partial x}_{i,j}$ and $\frac{\partial(vq)}{\partial x}_{i,j}$ are defined, $\frac{\partial q}{\partial t}_{i,j}$ can be calculated using Equation 6:

$$\frac{\partial q}{\partial t} = \text{rhs}(q) = \left(-\frac{\partial(uq)}{\partial x} + \frac{\partial(vq)}{\partial y} \right) \tag{6}$$

Using all these definitions and variables, the solver is completed by the addition of the Runge-Kutta method.

3.2 The Parallelization Using OpenMP

The serial implementation of the solver is made according to the steps explained in Section 3.1. In this section, the parallelization using OpenMP is explained. The main changes are made on the computationally intensive part of the solver, which is the "RhsQ" function. The integration method is kept the same, the only change is made in the structure of the processes.

In the first step, the current and total core numbers are stored in a variable to be used later. In addition to these, the current node number is also calculated and stored in another variable. This calculation is explained in 3.1 and the related equation is given in Equation 3. This part of parallel solver in the code file is given below:

```
int my_rank = omp_get_thread_num();
int thread_count = omp_get_num_threads();
```

```
for(int j=0; j<msh->NY; j++){
    for(int i=0; i<msh->NX; i++){
      const int idn = j*msh->NX + i;
```

At this point, it is important to notice that the calculation of current node number (idn) is performed under a loop. This reason of this is to consider whole space which is represented in terms of nodes. The operations on core numbers are outside the loop; however, this is not a problem as it will be later seen that this function will be called again and again by parallelizing in the main section of the code.

The meaning of storing current and total core numbers appears here. In order to make parallel calculation in the related core, current node number (idn) is divided by total core number and the remainder is compared with the current core number. If this equality check returns true, then the operation is performed. Otherwise, the code moves on. This part of the code is given below:

```
for(int j=0; j<msh->NY; j++){
    for(int i=0; i<msh->NX; i++){
      const int idn = j*msh->NX + i;
      const double unx = solver->u[2*idn + 0];
      const double uny = solver->u[2*idn + 1];

      if ((idn % thread_count) == my_rank){ //paralel part

        // neighbor elements
        int elmE = msh->N2N[4*idn + 0];
        int elmN = msh->N2N[4*idn + 1];
        int elmW = msh->N2N[4*idn + 2];
        int elmS = msh->N2N[4*idn + 3];

        // neighbor velocities
        double uxE = solver->u[2*elmE + 0];
        double uyN = solver->u[2*elmN + 1];
        double uxW = solver->u[2*elmW + 0];
        double uyS = solver->u[2*elmS + 1];

        // Find spacing just in case it is not uniform
        double hip1 = fabs(msh->x[elmE] - msh->x[idn] );
        double him1 = fabs(msh->x[idn]  - msh->x[elmW]);

        double hjp1 = fabs(msh->y[elmN] - msh->y[idn] );
        double hjm1 = fabs(msh->y[idn]  - msh->y[elmS]);

        double dfqdx = unx> 0 ? (unx*solver->q[idn]- uxW*solver->q[elmW])/him1
        ↪  :  (uxE*solver->q[elmE]- unx*solver->q[idn])/hip1;
        double dfqdy = uny> 0 ? (uny*solver->q[idn]- uyS*solver->q[elmS])/hjm1
        ↪  :  (uyN*solver->q[elmN]- uny*solver->q[idn])/hjp1;

        double rhsq   = -(dfqdx +dfqdy);

```

```
        // Time integration i.e. resq = rk4a(stage)* resq + dt*rhsq
        double resq = tstep->rk4a[stage]*tstep->resq[idn] + tstep->dt*rhsq;
        // Update q i.e. q = q 6 rk4b(stage)*resq
        solver->q[idn]  +=  tstep->rk4b[stage]*resq;

        tstep->resq[idn] = resq;
        tstep->rhsq[idn] = rhsq;
      }
    }
  }
```

This parallelized code is called in the main section by specifying parallel programming information beforehand. This specifications and the code are given below:

```
//paralel programing information
  int thread_count;
  thread_count = strtol(argv[2], NULL, 10);
  omp_set_num_threads(thread_count);
```

```
// ********************Time integration*************************************/
  // for every steps
  for(int step = 0; step<Nsteps; step++){
    // for every stage
    for(int stage=0; stage<tstep.Nstage; stage++){


      #pragma omp parallel
        // Call integration function
        RhsQ(&advc, &tstep, stage);

    }
```

To record timings at the end of solving process a timer is initiated and later it is terminated. The elapsed time is calculated and stored in a variable. This implementation is also given below:

```
// Start the timer
  double start_time = omp_get_wtime();
```

```
// Stop the timer
double end_time = omp_get_wtime();

// Calculate and print the elapsed time
double elapsed_time = end_time - start_time;
```

This completes the parallelization of a serial solver through OpenMP. To evaluate the performance of this solver, different resolutions are provided in the input file and corresponding timings are also recorded and they will be given in Section 5.

4 The Calculation of Infinity Norm

Infinity norm simply indicates how large a vector is by comparing its magnitude with the largest entry. In the parallelized explicit in-time solver with finite difference solver, this norm can be found by three different methods. These methods belong to OpenMP, and they can be listed as "Critical Regions", "Atomic Operations" and "Reductions". All these methods have pros and cons compared to each other. However, in this text, the timing of computation is taken as base to evaluate their performance.

These methods are applied as separate functions taking results of solver and performing comparison to obtain the infinity norm. The details of these functions will be provided throughout this section.

4.1 Critical Regions

The logic behind using Critical Regions to find infinity norm is based on comparing the result of solver by a predefined value, then recording it to a variable if it is larger than this predefined value. In order to implement this on the main code, a function is created. At this point, it is important to make sure that this runs for every step and two values, current and predefined, are compared to each other. This is also ensured by the use of for loop.

The problem with this method is the computational time. Since it makes the code more serial than it was, the computational time is more compared to atomic and reduction methods, which is less efficient. This will be also represented in Section 5 in more detail.

```
1  double inf_norm_C(solver_t *solver,double Val){
2  mesh_t *msh = &solver->msh;
3
4  double maxVal=Val;
5
6  #pragma omp parallel for shared(maxVal)
7      for (int idn = 0; idn < msh->NX*msh->NY; idn++) {
8        #pragma omp critical
9        {
10         if (solver->q[idn] > maxVal) {
11
12                     maxVal = solver->q[idn];
13                 }
14               }
15     }
16 return maxVal;
17 }
```

4.2 Atomic Operations

Atomic Operations compensates the serialized effect of Critical Regions, so it is safer to use this to calculate infinity norm. However, it is possible to encounter hardware restrictions in the utilization of this method. The computational time is less compared to Critical Regions, so it is more efficient in that sense.

```
double inf_norm_A(solver_t *solver, double Val) {
    mesh_t *msh = &solver->msh;

    #pragma omp parallel for
    for (int idn = 0; idn < msh->NX * msh->NY; idn++) {
      if (Val < solver->q[idn]){
        #pragma omp atomic write
        Val = solver->q[idn];
      }
      //else {
       // #pragma omp atomic write
       // Val = Val;
     // }
    }

    return Val;
}
```

4.3 Reductions

Reductions are more preferred compared to other methods since it improves the parallelism characteristic. It is the one having the least computational time and also the most effective in that sense.

```
double inf_norm_R(solver_t *solver,double Val){
mesh_t *msh = &solver->msh;
double maxVal=Val;
#pragma omp parallel for reduction(max:maxVal)
    for (int idn = 0; idn < msh->NX*msh->NY; idn++) {
        if (solver->q[idn] > maxVal) {
                    maxVal = solver->q[idn];
                }
    }
return maxVal;
}
```

At this stage, it is important to point out the way followed to run the code using one of these options. The operation performed is specified by the input from the user. In that sense the following numbers are defined to represent four different options.

1: Only running parellel rshQ function
2: Infinity norm calculation with critical sections
3: Infinity norm calculation with reduction
4: Infinity norm calculation wih atomic

By specifying the desired number, the desired operation can be initiated. To compile the code and run the code, the followings should be made:

- To compile the code: `gcc -o advection advection.c -lm -fopenmp`
- To run the code: `./advection input.dat core-number desired-analysis`

---

5 Results

5.1 Part 1: Parallelizations

"How is parallelization applied" is explained in Section 3. In this section, the results are shown. As seen in Figure 2-3-4, also expected result is plotted. The expected result assumes that the main required time is 1 core computational time of that resolution. If the parallelization is 100% efficient, then 2 cores computational time has to be half of 1 core computational time. This calculation is repeated for 3 and 4 cores computational time.

5.1.1 Recorded Timings for Different Mesh Resolutions

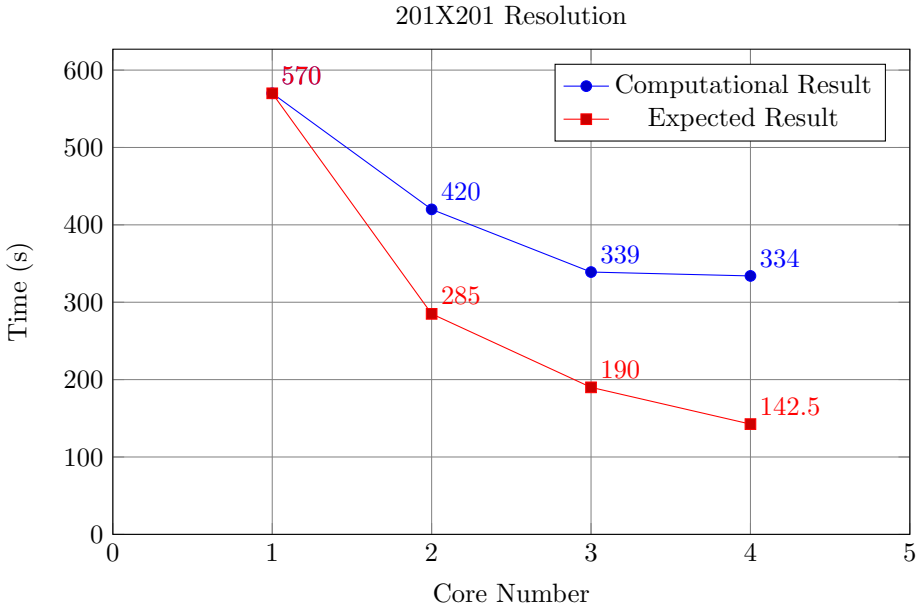

Figure 2: Calculation results with the expected result for 201X201 resolution

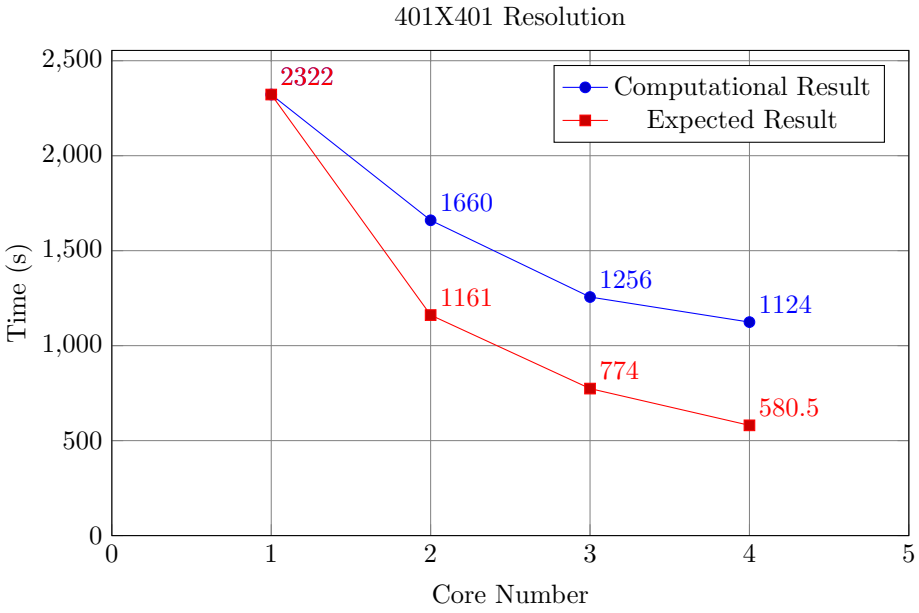

Figure 3: Calculation results with the expected result for 401X401 resolution

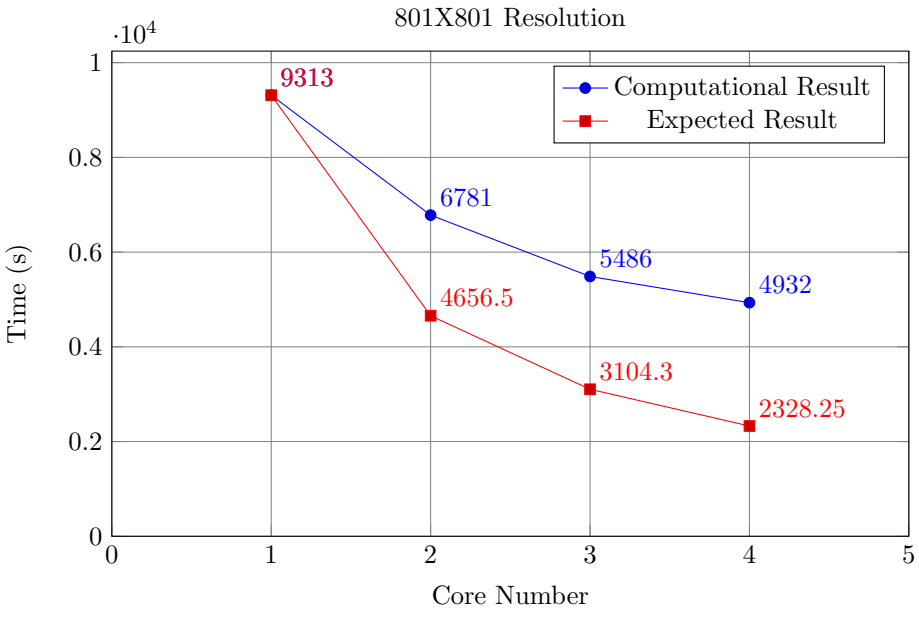

Figure 4: Calculation results with the expected result for 801X801 resolution

5.2 Part 2: Infinity Norms

"How is the infinity norm calculated" is explained in Section4. As seen in Figure5, the infinity norm is changed with a total number of nodes. This is expected due to the nature of the advection problem. If mesh resolution increases, critical regions are calculated more accurately. But these critical regions can be also singular points. So that if most part of the solution is not changed with increasing mesh resolution, the solution can be accepted.

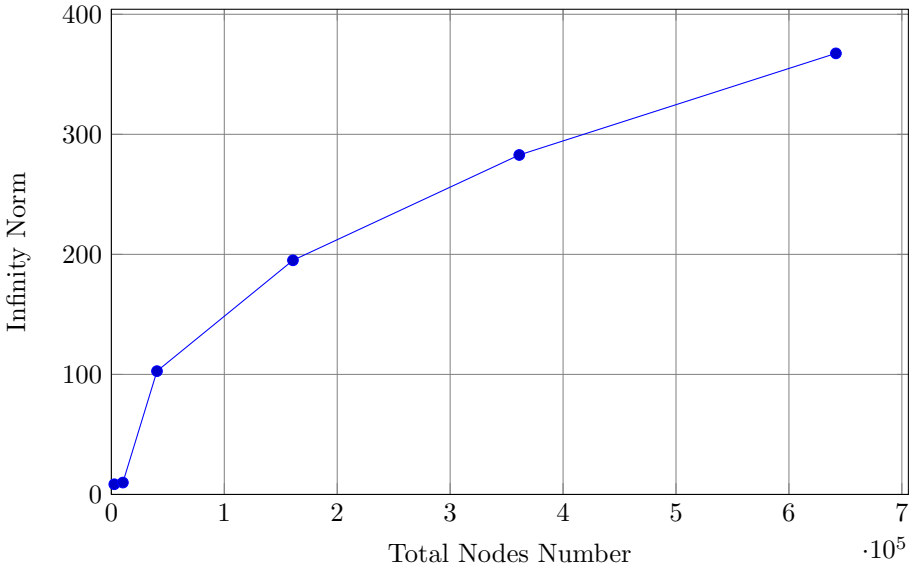

Figure 5: Infinity Norm Result of Different Total Nodes Number

In the next section, the times for calculating the infinity norm with three approaches are shown. Since every approach found the same result, the results are not shown again and can be seen in figure5.

5.2.1 Recorded Timings for Different Mesh Resolutions

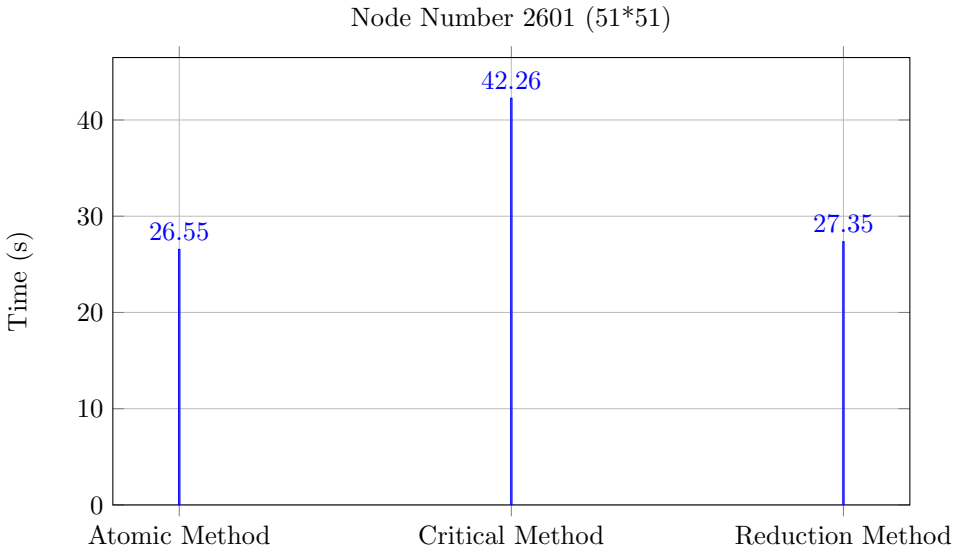

Figure 6: Required time for 51x51 mesh resolution with three parallelization approach

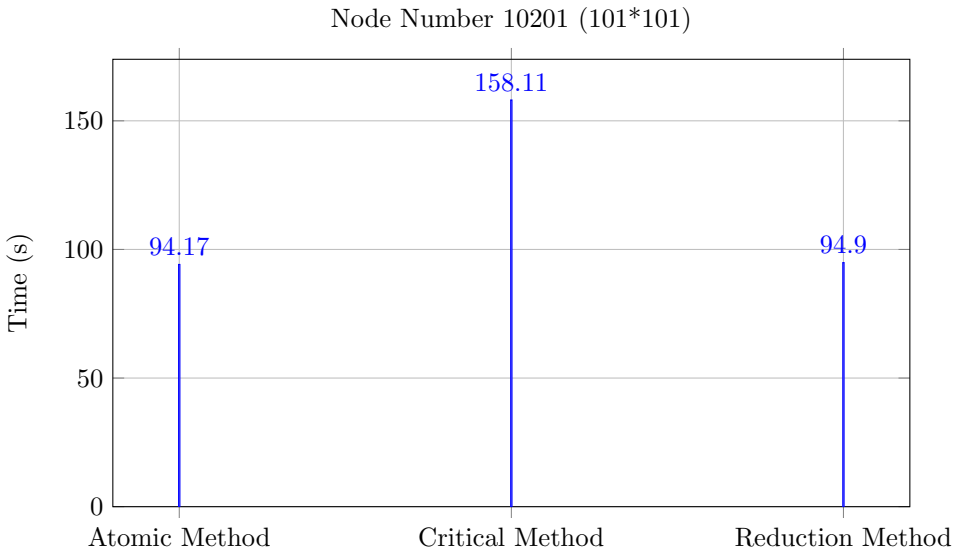

Figure 7: Required time for 101x101 mesh resolution with three parallelization approach

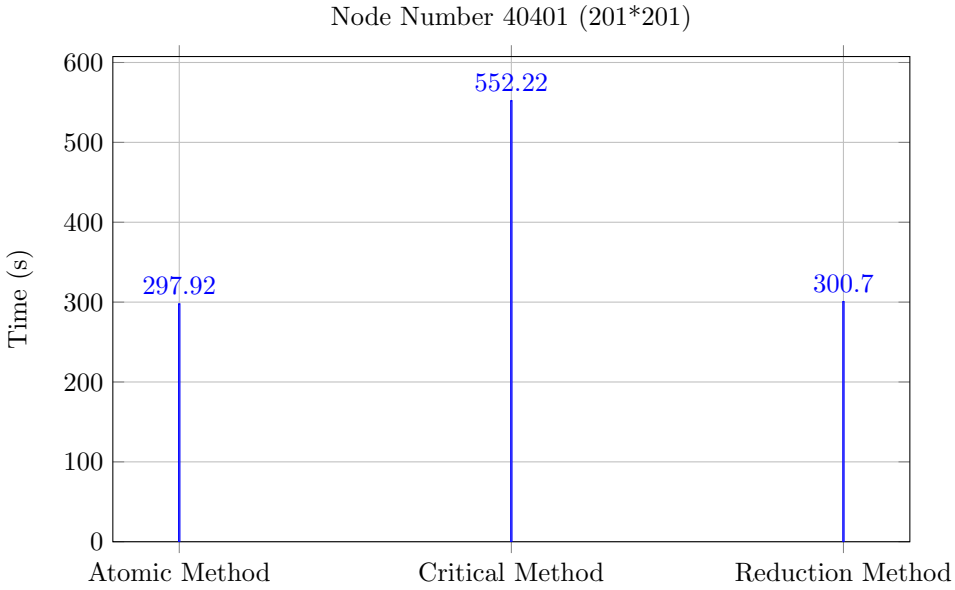

Figure 8: Required time for 201x201 mesh resolution with three parallelization approach

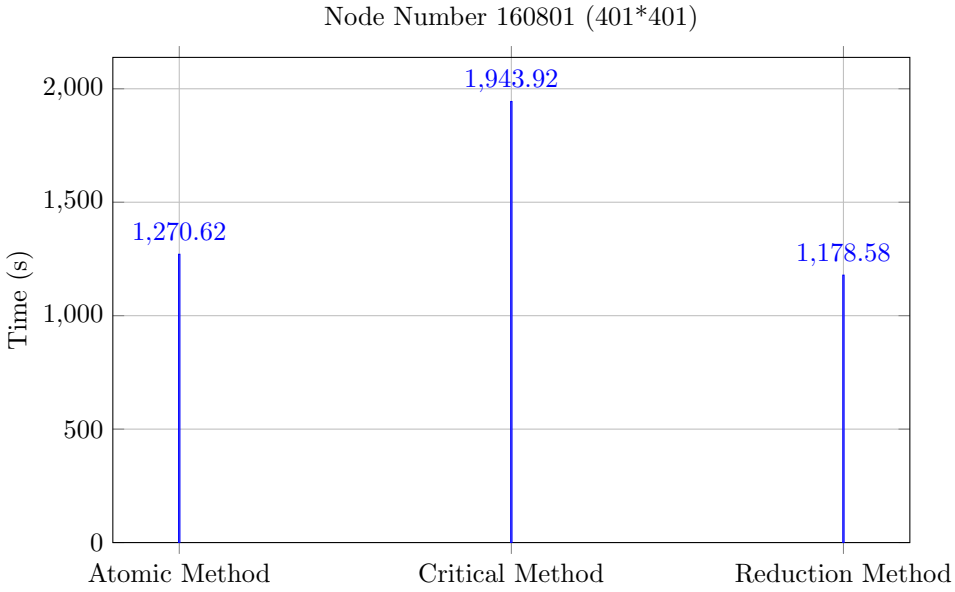

Figure 9: Required time for 401x401 mesh resolution with three parallelization approach

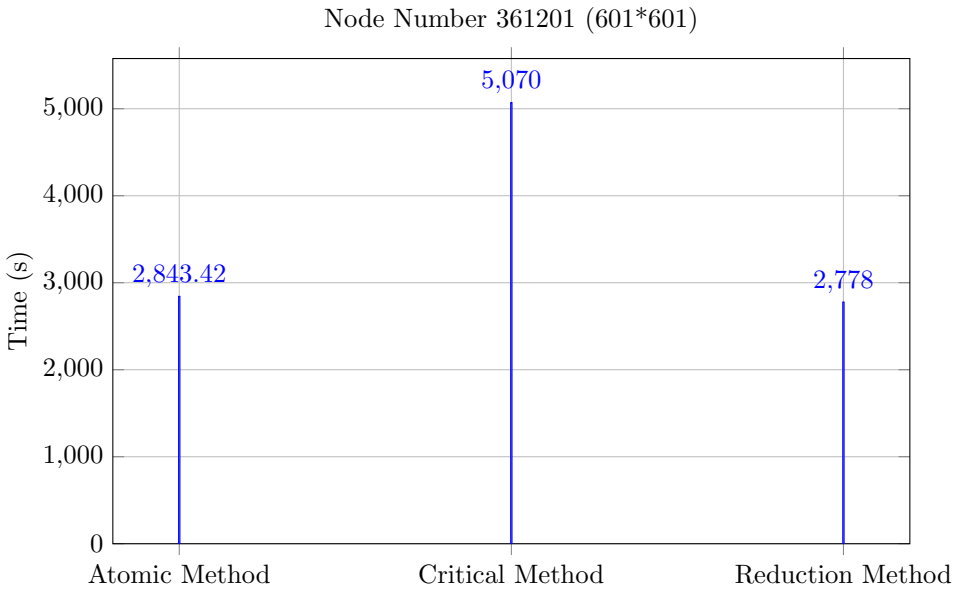

Figure 10: Required time for 601x601 mesh resolution with three parallelization approach

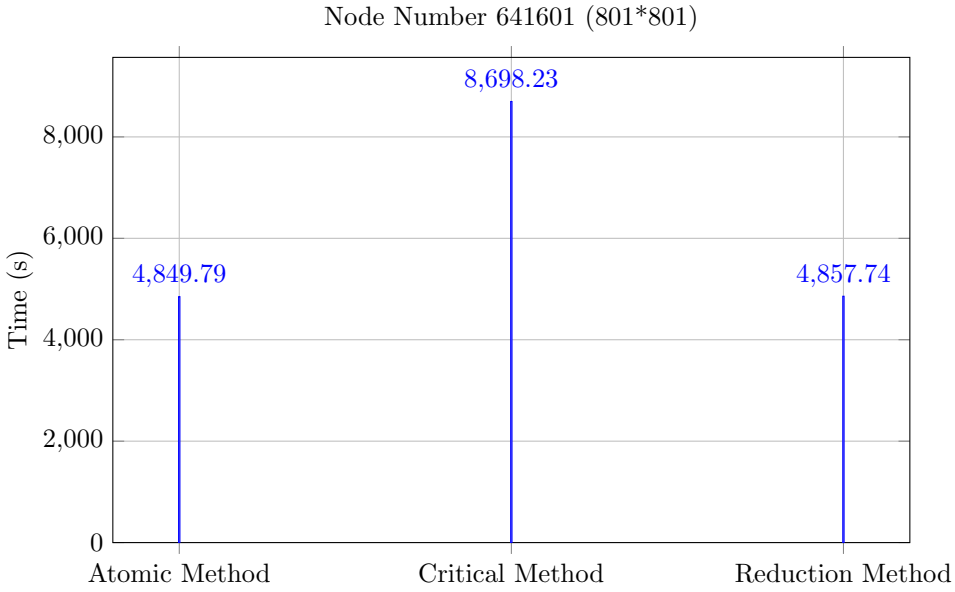

Figure 11: Required time for801x801 mesh resolution with three parallelization approach

6 Conclusion and Comments

In conclusion, in this paper, the effect of parallelization on a serialized solver is represented by considering the recorded timings. To evaluate the performance, the solver is run for different mesh resolutions and thread numbers. Infinity norm is also found by using the parallel code along with three different methods. These methods are critical regions, atomic and reductions. The methodology is explained and the results are presented throughout the paper.

As seen in figures 2-3-4, parallelization is never 100% percent efficient. This is expected because only the solver part is parallelized. Also, there are lot of communication time in solver. This problem cannot be solved with parallelization. However, as a general comment, it can be said that increasing the number of threads reduces the computation time, which is a desirable outcome. This amount of reduction does not match with the theoretical amount, which can be explained considering external effects such as hardware capability and the nature of solver.

As seen in figure5, the infinity norm has not converged yet even though the mesh resolution is very high. This makes us think there can be singular points in the system. The infinity norm position and time have to be checked.

As seen in figure 6-7-8-9-10-11, required time for atomic and reduction methods are almost same. However, the required time for critical regions method is always higher than others. This is expected indeed since critical regions method makes the code similar to the serialized one due to its nature.

References

[1] Eduard Ayguadé et al. "The design of OpenMP tasks". In: *IEEE Transactions on Parallel and Distributed systems* 20.3 (2008), pp. 404–418.

[2] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.