OpenMP Project: Temperature of a Metal Plate

Nikolaos Zafeiriadis

February 2025

1 Introduction

In this project, the Equilibrium Heat Equation of a metal plate is going to be solved:

$$\nabla^2 T(x, y) = 0$$

This function is a classic Laplacian Equation and it will be solved using the method of Jacobi for linear systems. The code is parallelized using OpenMP and is going to run with different number of threads, in order to measure the speedup of the program.

2 Theoretical Background

2.1 Jacobi Method

The Jacobi method is an iterative technique for numerically solving linear systems. If there is a system of N linear equations of N variables, each term is written in the form:

$$v_i = f(c, v_1, v_2, ..., v_{i-1}, v_{i+1}, ..., v_N)$$

By setting random initial values for every variable (for example all $v_i = 0$), previous formula is used for all the variables in order to obtain an approximation of each one. At every iteration, their values are updated using the results of the previous approximation. By calculating the approximate error of each variable, the system can be solved with a desired accuracy. The convergence of this method is $\mathcal{O}(N^2)$.

2.2 Amdahl's Law

Amdahl's Law suggests that while increasing the resources of a system, only a fraction of it can actually run faster. This law is expressed by the following formula for calculating the expected speedup of a program:

$$Speedup = \frac{1}{(1-\rho) + \frac{\rho}{n}}$$

where ρ is the fraction of the program that can be made parallel and n is the number of threads used.

3 Implementation

The implementation of this paper happened for a square metal plate with a side of length $a = 3m$. The boundaries of this plate were the following:

- Top Side: 200^oC
- Left Side: 150^oC
- Bottom Side: -20^oC
- Right Side: -40^oC

The program was written in C language and was made parallel using OpenMP. Since the laptop of usage had a core of 4 real cores, the calculation took place from 1, all the way to 8 threads. Each dimension of the square had a grid of $N = 1200$ points. The C code was the following:

```c
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#define a 3.0 //Meters
#define Tbottom (273.15-20) //Kelvin
#define Ttop (273.15 + 200) //Kelvin
#define Tleft (273.15 + 150) //Kelvin
#define Tright (273.15 - 40) //Kelvin
#define TOLERANCE 0.01
#define maxIterations 10000000
#define NR_END 1
#define FREE_ARG char*


// Numerical Recipes.
void nrerror(char error_text[])
/* Numerical Recipes standard error handler */
{
    fprintf(stderr,"Numerical Recipes run-time error...\n");
    fprintf(stderr,"%s\n",error_text);
    fprintf(stderr,"...now exiting to system...\n");
    exit(1);
}

double **dmatrix(long nrl, long nrh, long ncl, long nch)
/* allocate a double matrix with subscript range m[nrl..nrh][ncl
    ..nch] */
{
    long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
    double **m;
    /* allocate pointers to rows */
```

```
34      m=(double **) malloc((size_t)((nrow+NR_END)*sizeof(double*)))
        ;
35      if (!m) nrerror("allocation failure 1 in matrix()");
36      m += NR_END;
37      m -= nrl;
38      /* allocate rows and set pointers to them */
39      m[nrl]=(double *) malloc((size_t)((nrow*ncol+NR_END)*sizeof(
        double)));
40      if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
41      m[nrl] += NR_END;
42      m[nrl] -= ncl;
43      for(i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;
44      /* return pointer to array of pointers to rows */
45      return m;
46  }
47
48  void free_dmatrix(double **m, long nrl, long nrh, long ncl, long
        nch)
49  /* free a double matrix allocated by dmatrix() */
50  {
51      free((FREE_ARG) (m[nrl]+ncl-NR_END));
52      free((FREE_ARG) (m+nrl-NR_END));
53  }
54
55  // Main code.
56  int main(int argc, char **argv){
57      int i,j,c; //Looping.
58      double diff, maxDiff; //Stopping.
59      int N=100; // Default.
60
61      // Choosing N before running.
62      for (i=1;i<argc;i++){
63          if(argv[i][0]=='-'){
64              switch(argv[i][1]){
65                  case 'N': sscanf(argv[i+1],"%d",&N);
66                          break;
67              }
68          }
69      }
70      printf("N = %d\n", N); // After parsing N
71      printf("Starting iterations...\n"); // Before the iteration
        loop
72
73      double h = a/(N-1);
74
75      //Allocating Matrices.
76      double **T= dmatrix(0,N-1,0,N-1);
77      double **T_prev= dmatrix(0,N-1,0,N-1);
78
79
80
81
82      // Boundary Conditions plus setting the interior = 0 for
        Jacobi Method.
83      for (i=0;i<N;i++)
84      {
85          for (j=0;j<N;j++)
```

```
86          {
87              if (j==0){
88                  T_prev[i][j] = Tbottom;
89              }
90              else if (j==N-1){
91                  T_prev[i][j] = Ttop;
92              }
93              else if (i==0){
94                  T_prev[i][j] = Tleft;
95              }
96              else if (i==N-1){
97                  T_prev[i][j] = Tright;
98              }
99              else{
100                 T_prev[i][j] = 0;
101             }
102         }
103     }
104
105     for (c=0;c<maxIterations;c++){
106         //Next approximation.
107         #pragma omp parallel shared(T,T_prev) private(i,j)
    firstprivate(N) default(none)
108         {
109             #pragma omp for collapse(2)
110             for (i=1;i<N-1;i++)
111             {
112                 for (j=1;j<N-1;j++)
113                 {
114                     T[i][j] = (T_prev[i-1][j] + T_prev[i+1][j] +
    T_prev[i][j-1] + T_prev[i][j+1])/4;
115                 }
116             }
117         }
118         maxDiff=fabs(T[1][1]-T_prev[1][1]);
119         //Updating the T array for next iterations.
120         #pragma omp parallel for collapse(2) shared(T,T_prev)
    private(i,j,diff) firstprivate(N) reduction(max:maxDiff)
    default(none)
121         for (i=1;i<N-1;i++)
122         {
123             for (j=1;j<N-1;j++)
124             {
125                 diff = fabs(T[i][j]-T_prev[i][j]);
126                 if(diff>maxDiff){
127                     maxDiff=diff;
128                 }
129                 T_prev[i][j] = T[i][j];
130             }
131         }
132
133         if (maxDiff<TOLERANCE){
134             break;
135         }
136     }
137     printf("Iterations: %d", c+1);
138     //Deallocating.
```

```
139     free_dmatrix(T,0,N-1,0,N-1);
140     free_dmatrix(T_prev,0,N-1,0,N-1);
141     return 0;
142 }
```

4 Results and Discussion

By measuring the execution time of the program for each thread, the following graph was created:

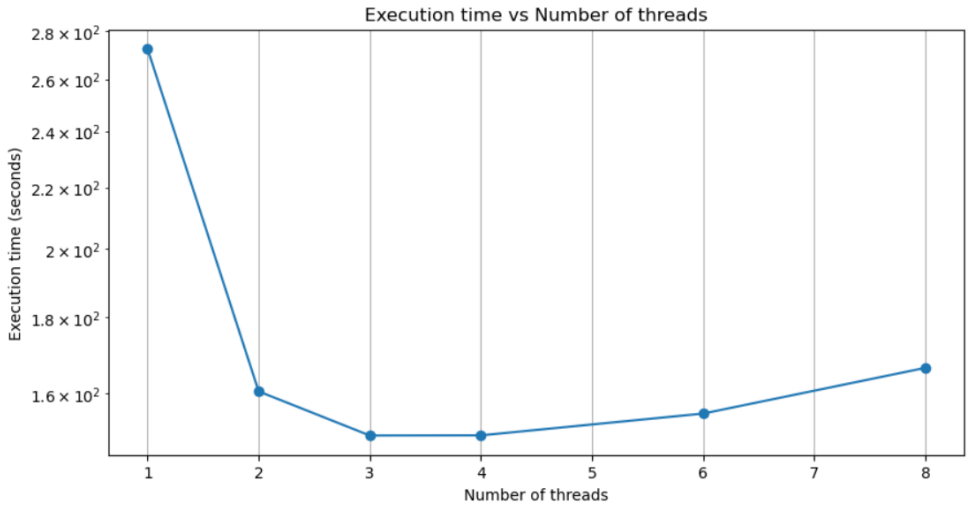

Figure 1: Execution time for different number of threads.

The resulting graph is making clear that the actual speedup occurs roughly at the number of threads as the number of physical cores of our system. As mentioned in earlier, the laptop of this implementation contains 4 cores, so the largest speedup was indeed expected around 4 threads.

At the following part of this project, the speedup of each number of threads was calculated and fitted using Amdahl's Law, in order to calculate the fraction of the program that is actually parallelizable. The result was:

$$\rho \approx 0.59$$

, meaning that the theoretical maximum speedup that we can get is $\sim 250\%$, while $n \to \infty$. With that in mind, the next graph presents the actual speedup, along with the Amdahl's Law predictions and the ideal case.

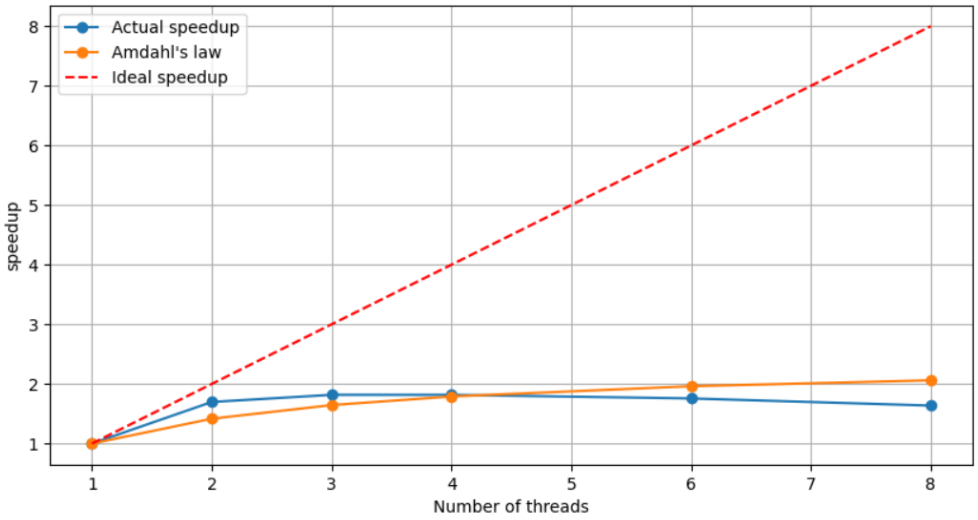

Figure 2: The speedup as a function of the number of threads.

As it becomes clear, the actual speedup fully aligns with the theoretical plateau. In the same context, the following figure presents the plateau of 250% speedup by plotting the expected speedup as a function of the number of threads:

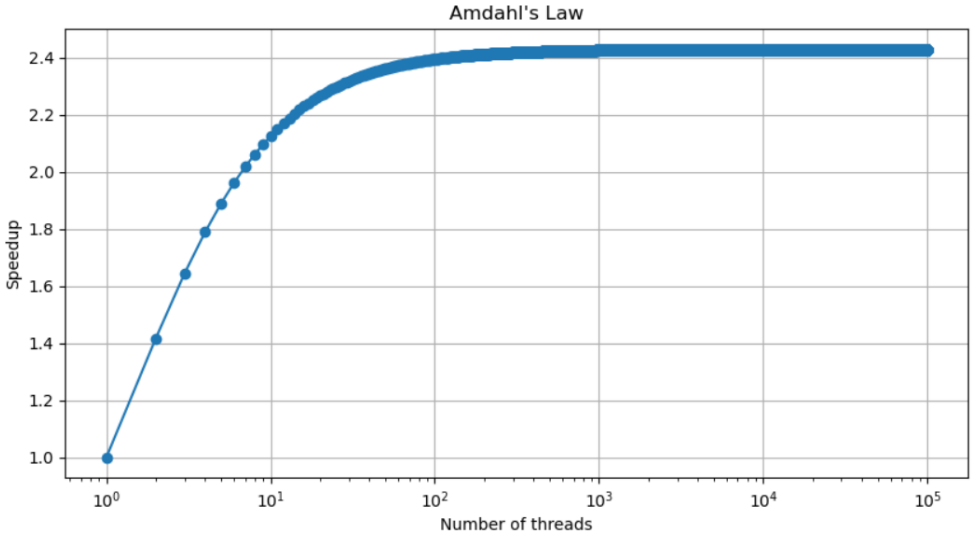

Figure 3: The expected speedup as a function of the number of threads, using Amdahl's Law.

The efficiency of the parallel code is calculated using:

$$Parallel\,Effficency = \frac{Speedup}{Number\,of\,threads} \cdot 100\%$$

In the following figure, the efficiency of the code is presented for each number of threads that it was used for implementation.

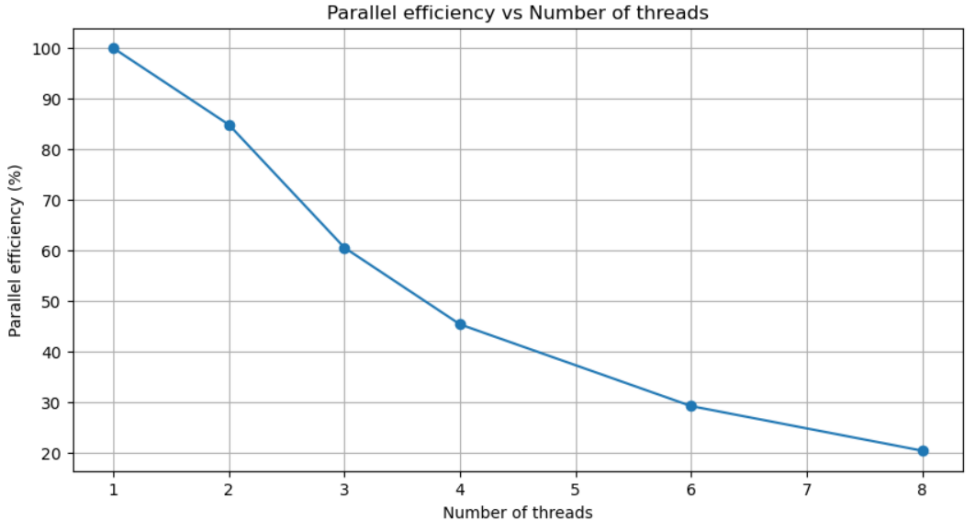

Figure 4: The efficiency of the parallel code with respect to the number of threads.

The last thing that needed to be checked was the convergence of our implementation. By solving the problem for many different number of grid points (N) and keeping a count of how many iterations it took, we made a graph of the convergence iterations with respect to N^2 (since N grid points exist on one line of the plate, the whole plate is separated into N^2 grid points):

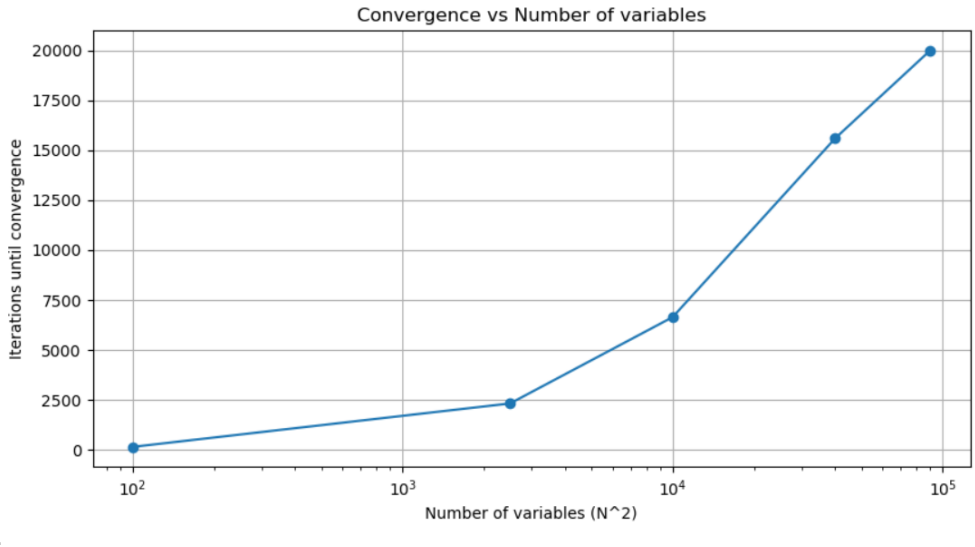

Figure 5: The convergence of the Jacobi method in our implementation.

As made clear from the figure, our method truly converges with a rate of $(\mathcal{O})[(N^2)^2]$, as implied from the theory. That means that our implementation sufficient to explain the temperature of this metal plate at an equilibrium state.

5 Conclusion

Concluding our findings, it becomes clear that the theory predicts perfectly the parallelization parameters and numerical solution of the heat equation of a metal plate at equilibrium, using the Jacobi method. Therefore the efficiency, as well as the convergence of the method, are categorized as accepted.