BOĞAZİÇİ UNIVERSITY

Cmpe 478 - Parallel Processing

Project 2

Can ÖZTURAN

Baran Deniz KORKMAZ - 2015400183

Egemen GÖL - 2016400009

SPRING 2020

Contents

1 Introduction

Solving Poisson's equation is one of the common fundamental tasks in parallel processing. The project aims at solving the equation system that arise from the discretization of Poisson's equation on a cube domain using Message-Passing Interface (MPI). A general description about the problem, and the proposed solution is given in this section.

The program takes a parameter N and discretize the domain into a (N + 1) x (N + 1) x (N + 1) cube grid with (x,y,z) belonging to a square domain $[0, 1]^3$. Informally, we can consider the input as a parameter specifying how many splits are we going to divide one dimension of the grid, i.e. h=$\frac{1}{N}$ stands for grid spacing. Explicitly, when N = 5, we are going to form a 6 x 6 x 6 cube where the values belonging to a constant axis, keeping others constant, will increase by $\frac{1}{N}$ = 0.2 in every step, i.e. 0, 0.2, 0.4, 0.6, 0.8, and 1.

Poisson's equation enables us to approximate the values of each grid point in terms of a function 'u' by the following equation:

$$u_{i,j,k} = \frac{1}{6}[u_{i+1,j,k}+u_{i-1,j,k}+u_{i,j+1,k}+u_{i,j-1,k}+u_{i,j,k+1}+u_{i,j,k-1}] - \frac{1}{6N^2}f_{i,j,k}$$

and,

$$f(x, y, z) = u_{xx} + u_{yy} + u_{zz}$$

where $f_{i,j,k} = f(x_i, y_j, z_k)$ and the shorthand notation for the coordinates is as $(x_i, y_j, z_k) = (ih, jh, kh), (0 \leq i, j, k \leq N)$.

Assuming that the existence of function u(x,y,z) (exact) returns the solution value at the boundary points, our program will find the approximate values for non-boundary points over the iterations with a converging error value.

The discretization of cube is key in terms of parallel processing. Below, a figure is given to describe our technique. We split the cube into subcubes in terms of number of processors assigned into the program. Our proposal ensures the efficient use of the fundamentals of parallel processing, benefits from the main advantages of parallelism. **Please note that**, we assume that the size of cube (in one dimension) will be evenly divided by the cube root of number of processors, to ensure a perfect division of cubes among processes, i.e. N+1 is evenly divided by $\sqrt[3]{np}$.

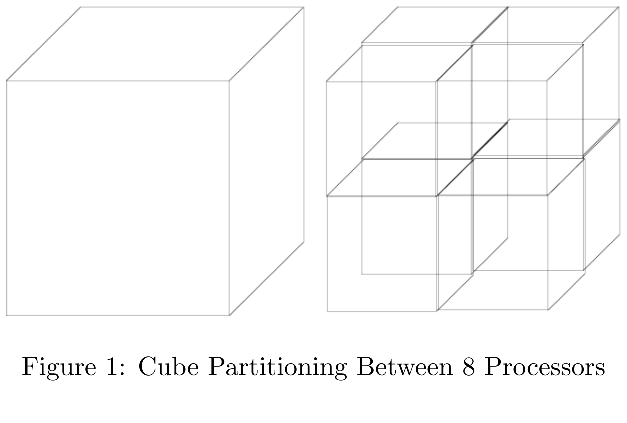


Figure 1: Cube Partitioning Between 8 Processors

2 Program Execution

2.1 Compilation

Compilation can be done in two ways. The user can select the longer version if it wants to determine the name of executable.

1. **Short Version:** The executable will be named **a.out** as default. This option is explicitly given since the scripts used in **2.3 Measure** and **2.4 Plot** assume that the executable is named **a.out**.

   ```
   >> mpic++ main.cpp
   ```

2. **Long Version:** Below, the executable named **main** will be created.

   ```
   >> mpic++ main.cpp -o main
   ```

2.2 Execution

Execution can be done by the following command.

```
>> mpirun -np [P] [--oversubscribe] [EXECUTABLE] [N] [--print]
```

Arguments:

1. P: Number of Processors
2. –oversubscribe: Nodes are allowed to be oversubscribed, even on a managed system, and overloading of processing elements. This option must be enabled whenever the allowed number of processors exceeds the desired number of processors in the program.
3. EXECUTABLE: Name of executable.
4. N: Program input which adjusts the grid spacing.
5. –print: Prints the actual and calculated cubes for providing observations. This options must be used only when the user wants to visualize the differences between the actual and calculated cubes.

Assumptions:

1. N is a valid positive integer. (N > 0)
2. N + 1 is evenly divided by $\sqrt[3]{P}$
3. In case the use of third argument (–print), it must be typed correctly as the way stated above. This option must only be used for visualization of outputs.

In all other uses, the argument control mechanism will print the error warning for invalid argument usages.

An example is given below where N equals 39, and the number of processors is 8. **Note that**, the example below is given for the case where executable is named **a.out**.

```
>> mpirun -np 8 --oversubscribe a.out 39
```

2.3 Measure

Requirements: The executable must be named as **a.out**. Make sure the parameters below are appropriate for you that can be set manually in the script **measure.py**.

- `sizes`
- `processors`
- `nof_reps`

Run:

```
>> python3 measure.py
```

Measurements are in `results.csv` file in the folder.

2.4 Plot

The script **plot.py** provides the plots showing the error value over the iterations,by using the standard output provided by our executable file. By error value, we imply the difference between the entire cube in the current and previous iterations. Below, the command sequence and an example are given:

```
>> mpirun -np [P] [--oversubscribe] a.out [N] > [OUTPUT_FILE]
>> python3 plot.py [OUTPUT_FILE]
```

Example:

```
>> mpirun -np 8 -- oversubscribe a.out 39 > 39_8
>> python3 plot.py 39_8
```

Output: Plots in the following format will be provided in the folder as images.

1. `[OUTPUT_FILE]_log`: Logarithmic Scale
2. `[OUTPUT_FILE]_norm`: Normal Scale

3 Implementation Details

The implementation details is presented in the subsections given below in detail:

3.1 Overall Structure

The overall program can be documented in the following steps:

1. Initialize the main MPI communicator, `MPI_COMM_WORLD`.
2. Every processor initializes its own grid of size $(\frac{N+1}{\sqrt[3]{np}})x(\frac{N+1}{\sqrt[3]{np}})x(\frac{N+1}{\sqrt[3]{np}})$, by mapping the points of their own subcubes into the actual cube, by using **get_actual_boundary** function. We must note that only the values of boundary points will be initialized, the non-boundary points have the value 0 in the beginning.
3. Every processor sends its partitioning slices into its neighbors.
   Message-Passing Mechanism: Every processor is assumed to have 6 neighbors in total from each directions, i.e. top, down, left, right, front, behind. For those who lie within the strict boundaries of the cube, it is considered as if the cube has continuous dimensions, that is it starts from the other point again for a given direction from where it ends. This assumption is made in order to avoid the hard-coded regions within the code. Instead of considering different number of communications per processors based on their locations within the cube, every processor performs exactly 6 send and 6 receive operations in this way. Please observe that, this assumption will not affect the final result of jacobi iterations, since the boundary points will remain unchanged over the iterations. The 6 send and 6 receive operations are done by **non-blocking synchronous communication**, to avoid buffer usage. The operations take place in two subsequent blocks and corresponding **MPI_Waitall** calls in order to ensure that the correct slices are gathered into the correct variables by restricting that per one dimension in 3D space, the communication in one certain way will take place. The remaining communication in other direction per dimension (X, Y, or Z-axis) will take place in second block. This is required as we do not know in which direction in a given dimension the send and receive will take place, and same processor might be both the left and the right neighbor for a given processor for 8 number of processors in a small cube. In this case, the slices might be directed into wrong variables, if we do not let the communication occur one by one per dimension.

   For sharing their partitioning slices into the neighbors, **Derived Data Types** provided by MPI are utilized. Plus, every processor obtains the unique id of their neighbors by **get_neighbor** function.
4. Every processor form its own ghost grid of size $(\frac{N+1}{\sqrt[3]{np}}+2)x(\frac{N+1}{\sqrt[3]{np}}+2)x(\frac{N+1}{\sqrt[3]{np}}+2)$, by combining the slices received from the neighbors into one ghost grid

to perform jacobi iterations by considering the correct orientations of slices gathered.

5. Every processor performs Jacobi iterations for the non-boundary points by using the Poisson's formula given in **Introduction**.
6. Every processor computes the sum of the norm of the difference between the previous and current values of points on its own grid. Then the final summation is obtained by **MPI_Allreduce**.
7. Go back to step 3, until the difference over the subsequent Jacobi iterations becomes smaller than a predetermined value 10^{-5}.
8. Finalize the MPI communicator, `MPI_COMM_WORLD`.

3.2 Auxiliary Functions

The auxiliary function utilized in the project are listed below:

1. **int get_actual_boundary(int proc_num, char dim)**
   Description: Returns the minimum index boundary for a given dimension on the actual cube for a given processor.
   Parameters:
   - **proc_num**: The rank of processor.
   - **dim**: The dimension $\in$ {`'x'`,`'y'`,`'z'`}.
2. **int get_proc_index(int proc_num, char dim)**
   Description: Returns the processor index for a given dimension. Explicitly, per dimension the index value is in the interval $[0, \sqrt[3]{np})$,i.e. the processor with rank 7 on the rightmost upper corner on the cube has indices (2,2,2) if the number of processors is eight.
   - **proc_num**: The rank of processor.
   - **dim**: The dimension $\in$ {`'x'`,`'y'`,`'z'`}.
3. **int get_proc_num(int x,int y,int z)**
   Description: Returns the rank of processor, given the processor indices described above.
4. **int get_neighbor(int proc_num,char where)**
   Description: Returns the rank of desired neighbor.
   - **proc_num**: The rank of processor.
   - **where**: The direction $\in$ {`'l'`,`'r'`,`'f'`,`'b'`,`'t'`,`'d'`}, respectively for left, right, front, behind, top, and down neighbors.
5. **bool is_boundary(int i,int j,int k)**
   Given the actual (mapped into the actual cube) coordinates of a point, returns true if the point is on the boundary of actual cube.

4 Input & Output

4.1 Input

The program arguments are as follows:

1. N: Adjusts the grid spacing.
2. –print: Enables printing the actual and computed cube. **(Optional)**

Assumptions:

1. N + 1 is evenly divided by $\sqrt[3]{P}$, P: Number of Processors
2. N is a positive integer.

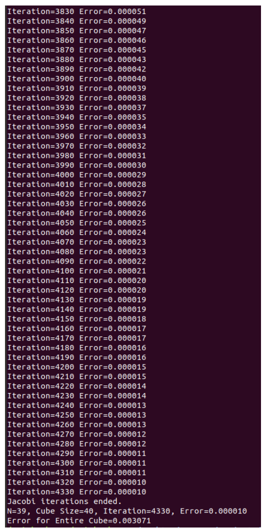
The user can manually change the functions **u(x,y,z)** and **f(x,y,z)**, if it wants to make calculations for differing functions. **Note that**, the predefined functions u(x,y,z) and f(x,y,z) are set as below for the following examples, unless otherwise specified.

- u(x,y,z) = xyz,
- f(x,y,z) = 0.0

4.2 Output

The output format is illustrated by the example given below, by the example where the number of processors is set 8 and N is 39.

```
>> mpirun -np 8 --oversubscribe main 39
```



```
Iteration=3830 Error=0.000051
Iteration=3840 Error=0.000049
Iteration=3850 Error=0.000047
Iteration=3860 Error=0.000046
Iteration=3870 Error=0.000045
Iteration=3880 Error=0.000043
Iteration=3890 Error=0.000042
Iteration=3900 Error=0.000040
Iteration=3910 Error=0.000039
Iteration=3920 Error=0.000038
Iteration=3930 Error=0.000037
Iteration=3940 Error=0.000035
Iteration=3950 Error=0.000034
Iteration=3960 Error=0.000033
Iteration=3970 Error=0.000032
Iteration=3980 Error=0.000031
Iteration=3990 Error=0.000030
Iteration=4000 Error=0.000029
Iteration=4010 Error=0.000028
Iteration=4020 Error=0.000027
Iteration=4030 Error=0.000026
Iteration=4040 Error=0.000026
Iteration=4050 Error=0.000025
Iteration=4060 Error=0.000024
Iteration=4070 Error=0.000023
Iteration=4080 Error=0.000023
Iteration=4090 Error=0.000022
Iteration=4100 Error=0.000021
Iteration=4110 Error=0.000020
Iteration=4120 Error=0.000020
Iteration=4130 Error=0.000019
Iteration=4140 Error=0.000019
Iteration=4150 Error=0.000018
Iteration=4160 Error=0.000017
Iteration=4170 Error=0.000017
Iteration=4180 Error=0.000016
Iteration=4190 Error=0.000016
Iteration=4200 Error=0.000015
Iteration=4210 Error=0.000015
Iteration=4220 Error=0.000014
Iteration=4230 Error=0.000014
Iteration=4240 Error=0.000013
Iteration=4250 Error=0.000013
Iteration=4260 Error=0.000013
Iteration=4270 Error=0.000012
Iteration=4280 Error=0.000012
Iteration=4290 Error=0.000011
Iteration=4300 Error=0.000011
Iteration=4310 Error=0.000011
Iteration=4320 Error=0.000010
Iteration=4330 Error=0.000010
Jacobi iterations ended.
N=39, Cube Size=40, Iteration=4330, Error=0.000010
Error for Entire Cube=0.003071
```

Figure 2: Output Example

As seen above, the program output prints the error value in every tenth iterations. Whenever, the Jacobi iterations are over, it prints the final results for the given parameters. **Finally**, the processors compute the difference between their computed subcube and the actual subcube given by using exact function 'u' by summing up the norm of element-wise differences. The **Final Error** over the entire cube is given as seen above by the expression **Error for Entire Cube=0.003068**. We can conclude that our algorithm works effectively to approximate non-boundary point values in terms of exact function.

The program output can be analyzed in 3 steps. These steps are related testing the output correctness and efficiency of parallelism.

1. Convergence
2. Output Analysis
3. Time Measurements

4.2.1 Convergence

We used the sum of absolute differences between every point for evaluating the difference between two cube states. **Every tenth iteration**, we print the difference between the current and the previous state. Note that, this computation is done **in every iteration**, but printed only in every tenth iteration. After the jacobi iterations finished, we obtain the cumulative results and plot them by a Python script named **plot.py** described in **2.4 Plot**.

Below, you will see the convergence plot for changing number of processors, where the parameter N is fixed into 47. The reason for picking 47 is that 48 (N+1) can be divided evenly by 2, 3, and 4 to show how plots differ for various number of processors.

Again, **note that**, the predefined functions u(x,y,z) and f(x,y,z) are set as below for the following plots, unless otherwise specified.

- u(x,y,z) = xyz,
- f(x,y,z) = 0.0

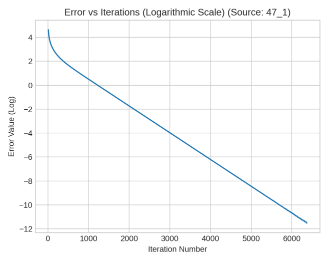


Figure 3: N=47 Number of Processors=1 (Logarithmic Scale)

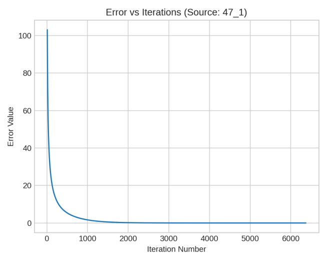


Figure 4: N=47 Number of Processors=1

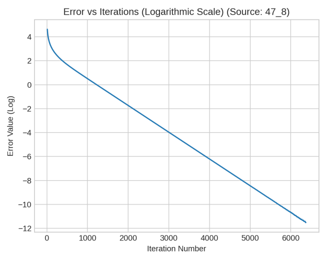


Figure 5: N=47 Number of Processors=8 (Logarithmic Scale)

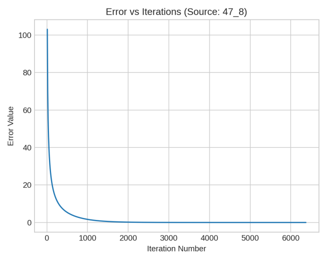


Figure 6: N=47 Number of Processors=8

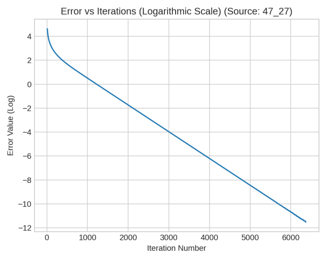


Figure 7: N=47 Number of Processors=27 (Logarithmic Scale)

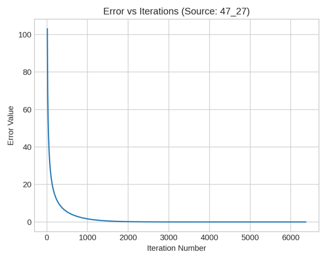


Figure 8: N=47 Number of Processors=27

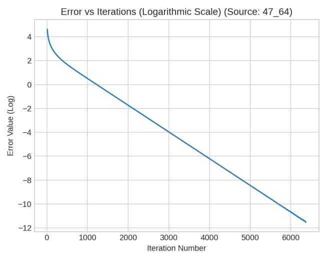


Figure 9: N=47 Number of Processors=64 (Logarithmic Scale)

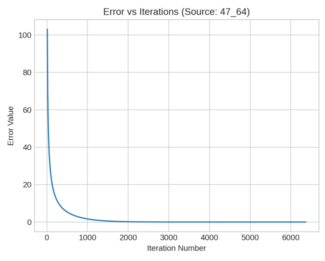


Figure 10: N=47 Number of Processors=64

The plots on left are given in the logarithmic scale for the y-axis. Note that, the Jacobi iterations stop if the difference between iterations becomes too small, our threshold is 10^-5.

Conclusion: We observe that the plots above are just the same for changing number of processors. Considering our task, this result is expected. We are actually moving towards the same iterations every time we run the program for a given N value. Changing number of processors only affect the run-time.

The program we use for these plots is `plot.py`, can be used as:

```
mpirun -np [P] --oversubscribe a.out 47 > [OUTPUT_FILE]
python3 plot.py [OUTPUT_FILE]
```

The plots will be provided as `[OUTPUT_FILE]_norm.png` and `[OUTPUT_FILE]_log.png` in the folder.
IMPORTANT NOTE: Plot analysis must be done without enabling –**print** option.

4.2.2 Output Analysis

In the previous section, we have tested whether our algorithm converges as desired. Now, in the current section, we test whether the algorithm provides reliable approximations about the actual values of exact function in our grid.

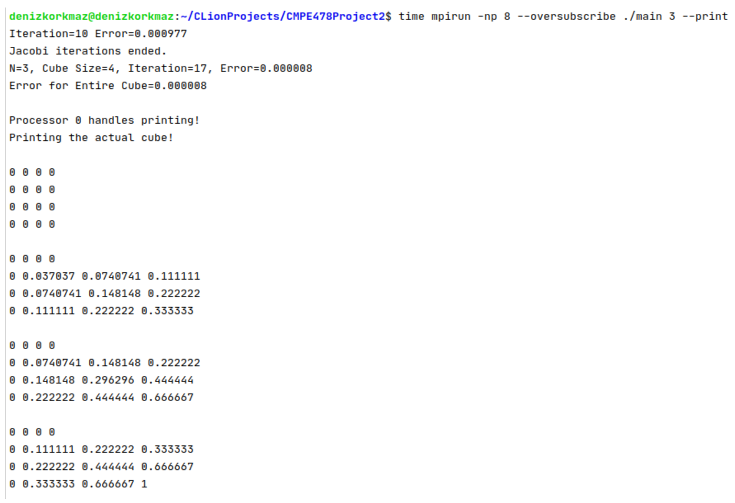
Output analysis is done in two stages:

1. **Final Error**
   After the Jacobi iterations are over, every processor computes the sum of absolute differences between the exact function $u(x, y, z)$ and our simulation's output values for every point in its cube space. Then, by **MPI_Allreduce** the sum is computed to be assigned as **Final Error**. Please note that, **Final Error** differs from the sum of the absolute differences between the current grid and the previous grid over the iterations. **Final Error**, compares the calculated cube to the actual cube, while other one measures the differences over the iterations to stop when converged.

2. **Visualization of Actual & Calculated Cube**
   If the option –**print** is used, when the Jacobi iterations are over, the processor with rank 0 forms the calculated cube by receiving the subcubes from the processors and embedding these pieces properly. The user then can see the visualizations of actual cube and calculated cube to compare.

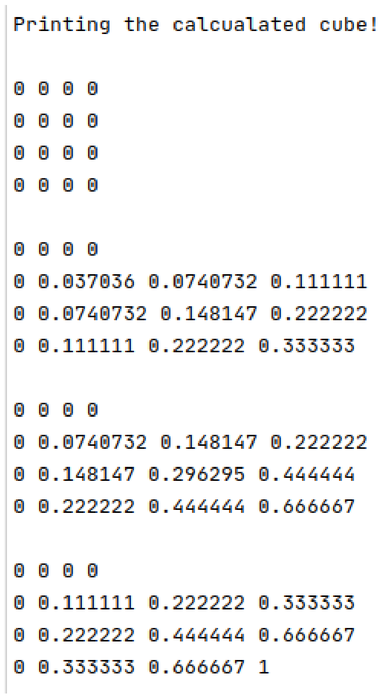
   Below, the procedure for output analysis is given by the example. The program used for the plots are given as:

```
>> mpirun -np 8 --oversubscribe main 3 --print
```



```
denizkorkmaz@denizkorkmaz:~/CLionProjects/CMPE478Project2$ time mpirun -np 8 --oversubscribe ./main 3 --print
Iteration=10 Error=0.000977
Jacobi iterations ended.
N=3, Cube Size=4, Iteration=17, Error=0.000008
Error for Entire Cube=0.000008

Processor 0 handles printing!
Printing the actual cube!

0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

0 0 0 0
0 0.037037 0.0740741 0.111111
0 0.0740741 0.148148 0.222222
0 0.111111 0.222222 0.333333

0 0 0 0
0 0.0740741 0.148148 0.222222
0 0.148148 0.296296 0.444444
0 0.222222 0.444444 0.666667

0 0 0 0
0 0.111111 0.222222 0.333333
0 0.222222 0.444444 0.666667
0 0.333333 0.666667 1
```

Figure 11: N=3 Actual Cube



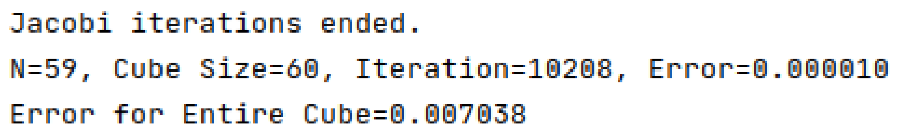
```
Printing the calcualated cube!

0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

0 0 0 0
0 0.037036 0.0740732 0.111111
0 0.0740732 0.148147 0.222222
0 0.111111 0.222222 0.333333

0 0 0 0
0 0.0740732 0.148147 0.222222
0 0.148147 0.296295 0.444444
0 0.222222 0.444444 0.666667

0 0 0 0
0 0.111111 0.222222 0.333333
0 0.222222 0.444444 0.666667
0 0.333333 0.666667 1
```

Figure 12: N=3 Calculated Cube

Let us stating the conclusion based on our observations. **Keeping N constant,** increasing the number of processors does not affect the number of iterations, or final error value. This remark is logical considering that we are actually moving through the same iterations and cube states, only the run-time changes as the number of processors changes. Let us present two examples to illustrate our remarks:

Example 1: Changing the number of processors, as N remains constant (N = 59). Note that $u(x, y, z) = xyz$ and $f(x, y, z) = 0.0$.

- Error is 0.007038 for N=59 with 1 processors and 10208 iterations.
- Error is 0.007038 for N=59 with 8 processors and 10208 iterations.
- Error is 0.007038 for N=59 with 27 processors and 10208 iterations.
- Error is 0.007038 for N=59 with 64 processors and 10208 iterations.

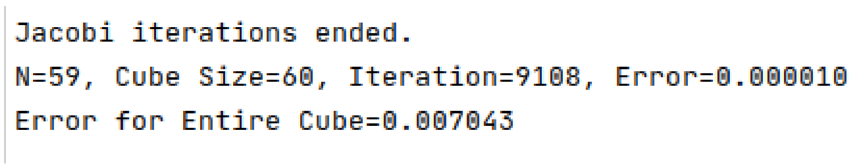


```
Jacobi iterations ended.
N=59, Cube Size=60, Iteration=10208, Error=0.000010
Error for Entire Cube=0.007038
```

Figure 13: Result for N = 59 (P:1,8,27,or 64 & u(x,y,z) = xyz)

Example 2: For different exact functions u, and so functions f. Note that in this example, N = 59 again to compare the results into **Example 1**.

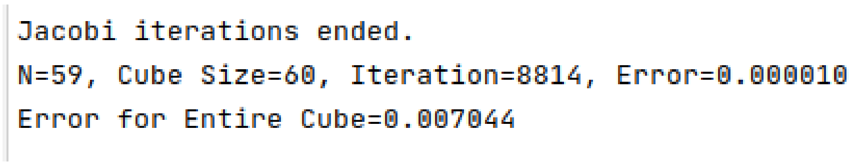
1. $u(x, y, z) = x^2y^2z^2$ and $f(x, y, z) = 2y^2z^2 + 2x^2z^2 + 2x^2y^2$



```
Jacobi iterations ended.
N=59, Cube Size=60, Iteration=9108, Error=0.000010
Error for Entire Cube=0.007043
```

Figure 14: $u(x, y, z) = x^2y^2z^2$

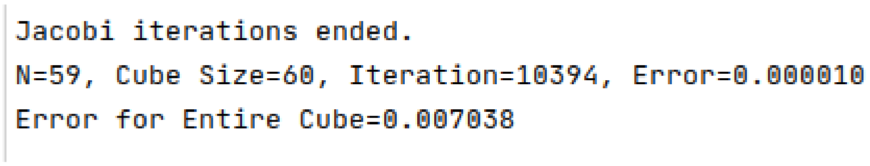
2. $u(x, y, z) = x^2y^2z^3$ and $f(x, y, z) = 2y^2z^3 + 2x^2z^3 + 6x^2y^2z$



```
Jacobi iterations ended.
N=59, Cube Size=60, Iteration=8814, Error=0.000010
Error for Entire Cube=0.007044
```

Figure 15: $u(x, y, z) = x^2y^2z^3$

3. $u(x, y, z) = x^2y^2 + xyz^2$ and $f(x, y, z) = 2y^2 + 2x^2 + 2xy$



```
Jacobi iterations ended.
N=59, Cube Size=60, Iteration=10394, Error=0.000010
Error for Entire Cube=0.007038
```

Figure 16: $u(x, y, z) = x^2y^2 + xyz^2$

As seen by the examples, we have successfully done plenty of tests from various aspects, and we conclude that our program works without any error.

4.2.3 Time Measurements

We use Python to make the time measurements. We invoke mpiexec command with python subprocess module, then we use python timeit module to measure the runtime. Our measurement script is called measure.py.

In order to overcome caching penalties, we run the code once without measuring at the beginning, this ensures exclusion of slow-starts.

We average multiple runs, in order to being robust to unexpected variances. This repetition number can be configured with a variable inside the program measure.py.

There are three main parameters for measure.py.

1. Cube Sizes: User adjusts the sizes list in the beginning of the program indicating the cube sizes she wishes to take measurements for. N+1 must be divisible by the cube root of every processor number the user wishes to measure with. Example for processor numbers [1,8,27,64], size - 1 must be a multiple of 24 (least common factor): sizes = [23, 47, 71]

2. Processor Counts: Since the program uses sub-cubes, the number of processors must be a cube of a natural number. Example processors list: [1, 8, 27, 64]

3. Repetition Count: We average the time measurements for robustness to unexpected variability.

After taking the measurements, we compose them into a .csv table, then save it into results.csv file into the root folder.

An example results.csv table:

N	T1	T8	T27	T64	S8	S27	S64
23	0.6	0.42	0.98	1.73	1.4	0.6	0.3
47	19.7	5.54	9.47	11.94	3.6	2.1	1.7
71	159.05	45.59	66.59	57.94	3.5	2.4	2.7

IMPORTANT NOTE: Time measurements must be done without enabling –**print** option, as expected.

REMARK: From the results above in the table, we observe **extremely low speed-up values**. This fact stems from the number of available processors we have on the machine used. When the required number of processors become available, we can see actual speed-up values that we can expect to arise.