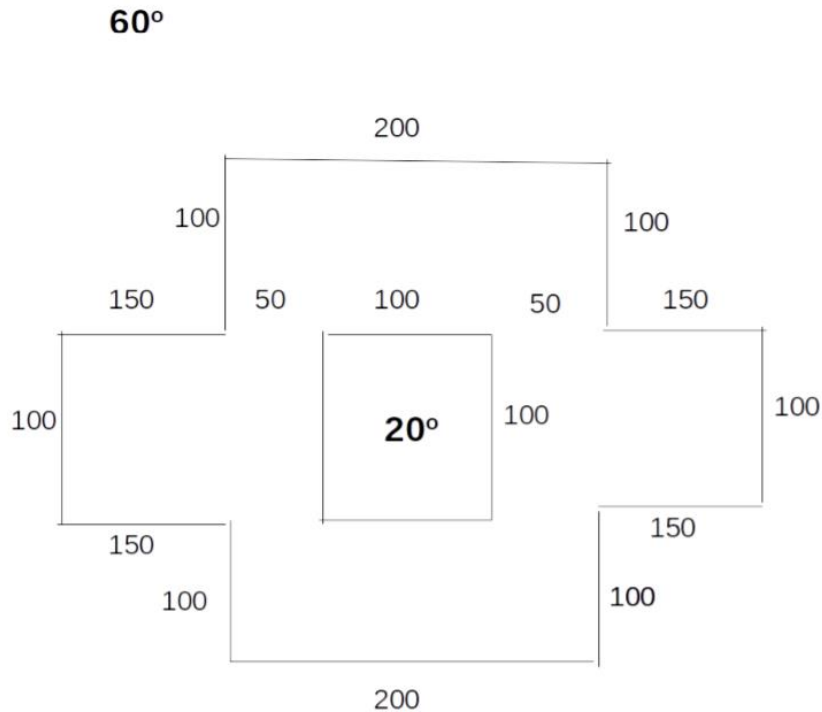


Abstract

In this project, we've been asked to calculate the Heat Equation for a specific geometric shape (mentioned below) using MPI library in C language.

Notes:

- We couldn't run the full size of the shape, so we divided the whole shape by 50 (As DR. Guy said), so we performed the equation size on the next data:
- Number of rows: 6.
- Number of columns: 10.
- We chose that the temperature of the inside is 30.
- Number of Processors = 4 (as requested).
- The temperature outside the shape = 60.
- *Note – The code is attached in another file calls "HeatEquation.c" -which can be opened in any kind of notepad/text editor.

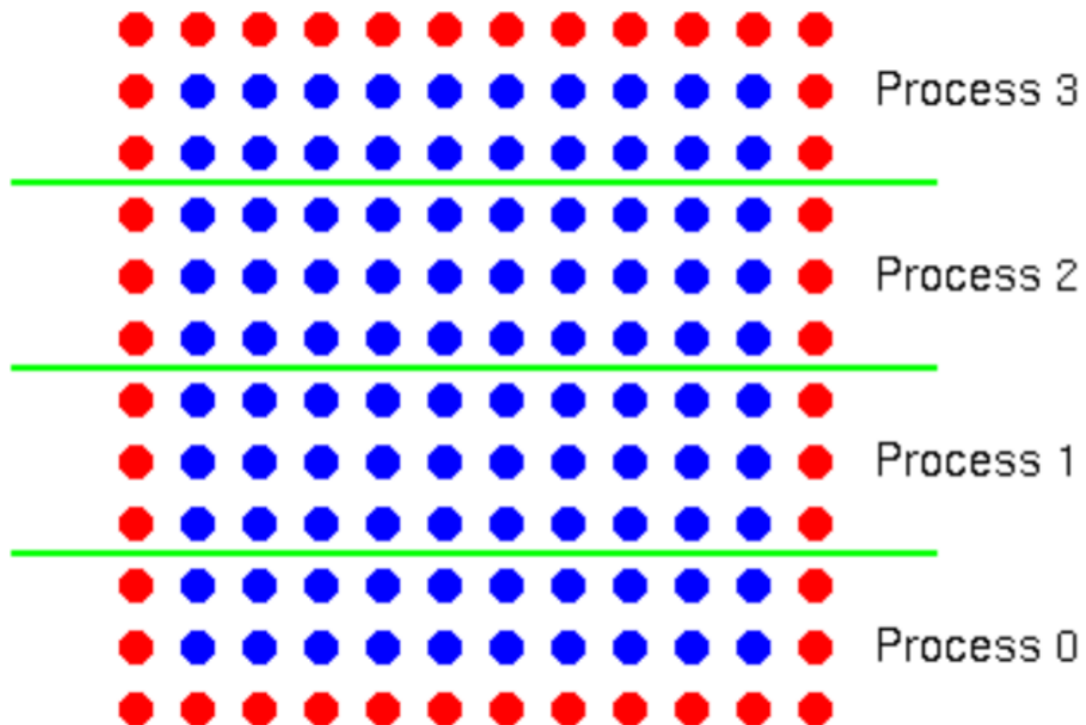


Pic 1. The shape we've been asked to perform the Heat Equation.

Domain Decomposition

We broke the shape into 4 processors. There is only one set of boundary values, but when we distribute the data, each processor needs to have an extra row for the data distribution.

We'll try to present an example of how we separate the data with the picture from DR. Guy's lecture:



Pic 2: An example from the lecture.

Sending Multiple Elements & Array Ordering

For the first time we want to send multiple elements. In this case, a whole row or column of data. That is exactly what the count parameter is for.

The common use of the count parameter is to point the Send or Receive routine at the first element of an array, and then the count will proceed to strip off as many elements as we specify.

This implies that the elements are contiguous in memory. That will be true for one dimension of an array, but the other dimensions will have a stride.

We iterate over the last (j) variable in the inner loop.

Jacobi Iteration & Break Point

The laplace equation on a grid states that each grid point is the average of its neighbors.

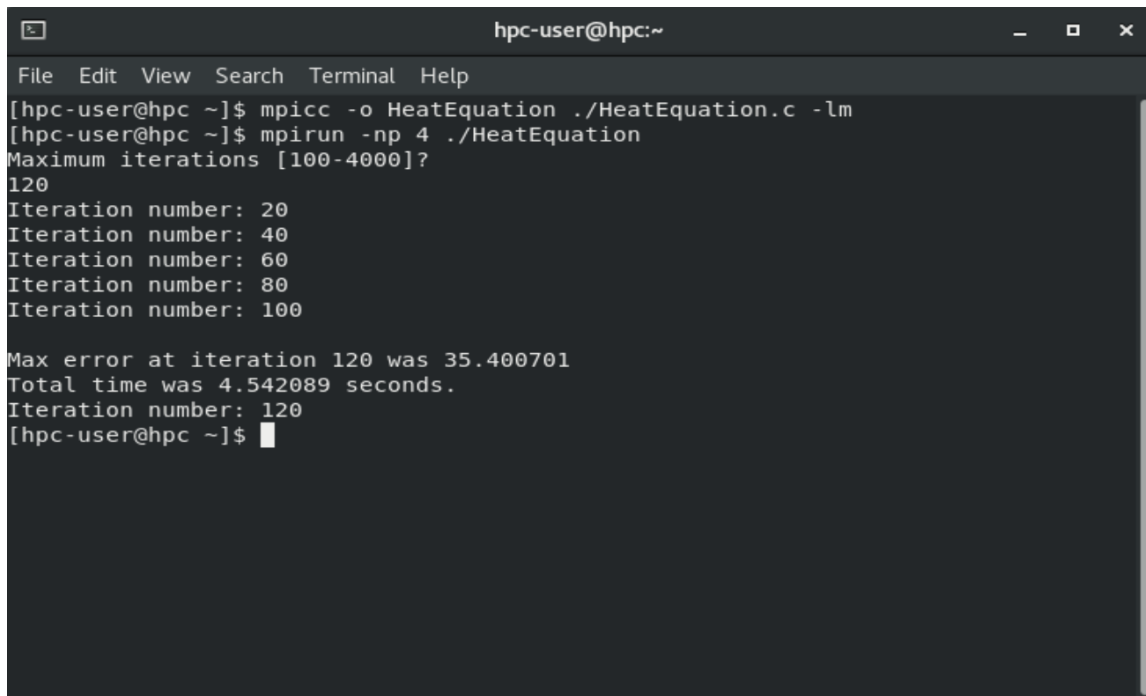
We can iteratively converge to that state by repeatedly computing new values at each point from the average of neighboring points.

We keep doing this until the difference from the pass to the next is small enough for us to tolerate (in Our code we set tolerance to 0.01).

How To Run The Code:

- The code will execute regular by compiling the code with: `mpicc -o HeatEquation ./HeatEquation.c -lm`
- Then, you need to run the code with: `mpirun -np 4 ./HeatEquation`
- The program asks you to input the number of iterations you want to run on.
- The program will run while the delta will be bigger than the tolerance and iteration number is less then max iteration.
- The program will create two files: The input – which gives the exact shape of the question, and the output – which shows the output of the program after the code.
- The program will print in the end the total time of the program and the max error.
- The heatmap of the temperature created in Python, the code and the screenshots is attached/ will be shown next.

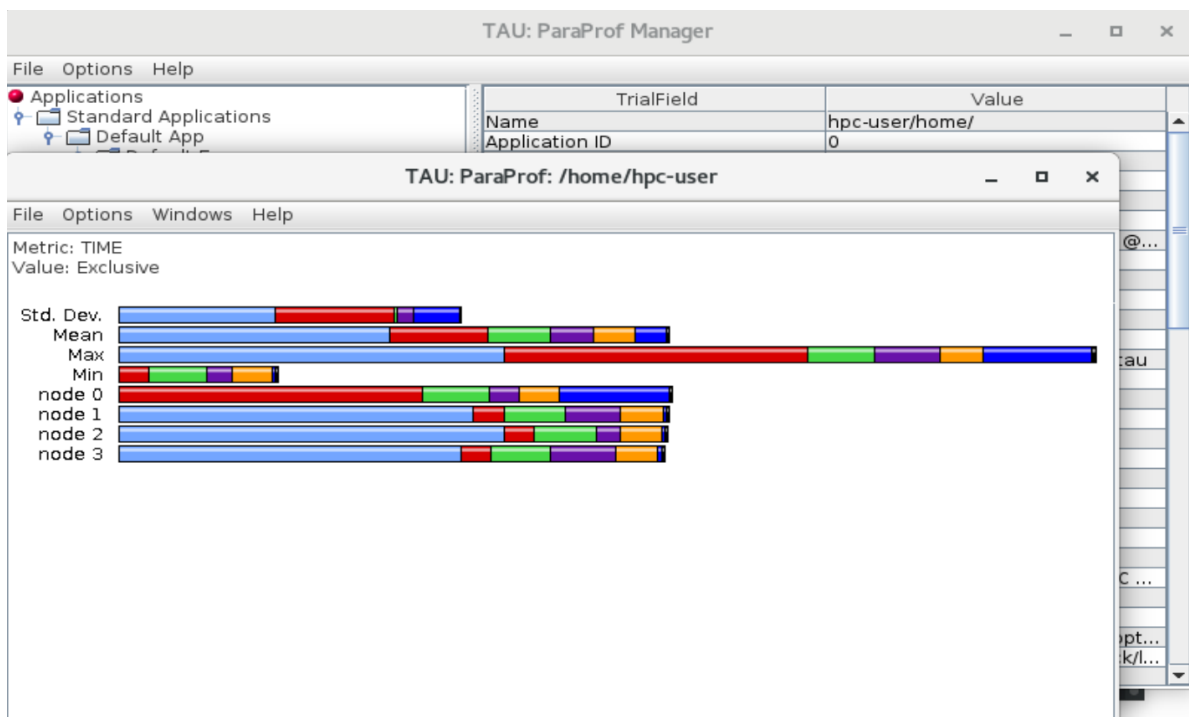
Screenshots of the output in the terminal:



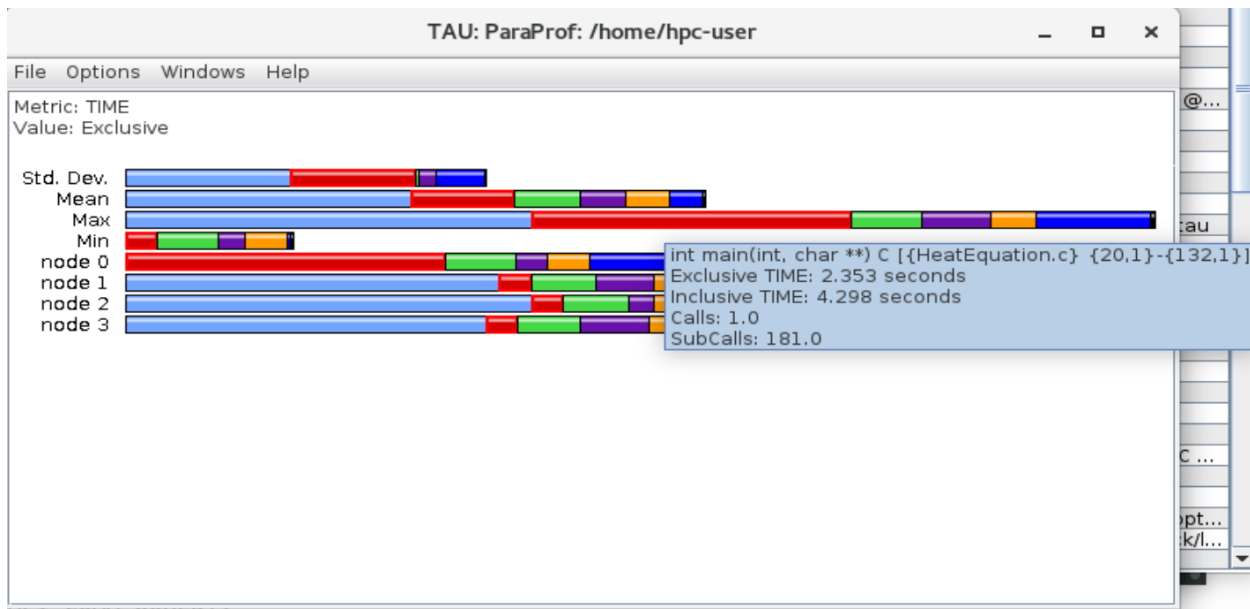
```
hpc-user@hpc:~  
File Edit View Search Terminal Help  
[hpc-user@hpc ~]$ mpicc -o HeatEquation ./HeatEquation.c -lm  
[hpc-user@hpc ~]$ mpirun -np 4 ./HeatEquation  
Maximum iterations [100-4000]?  
120  
Iteration number: 20  
Iteration number: 40  
Iteration number: 60  
Iteration number: 80  
Iteration number: 100  
  
Max error at iteration 120 was 35.400701  
Total time was 4.542089 seconds.  
Iteration number: 120  
[hpc-user@hpc ~]$
```

Pic 3: Output in the terminal

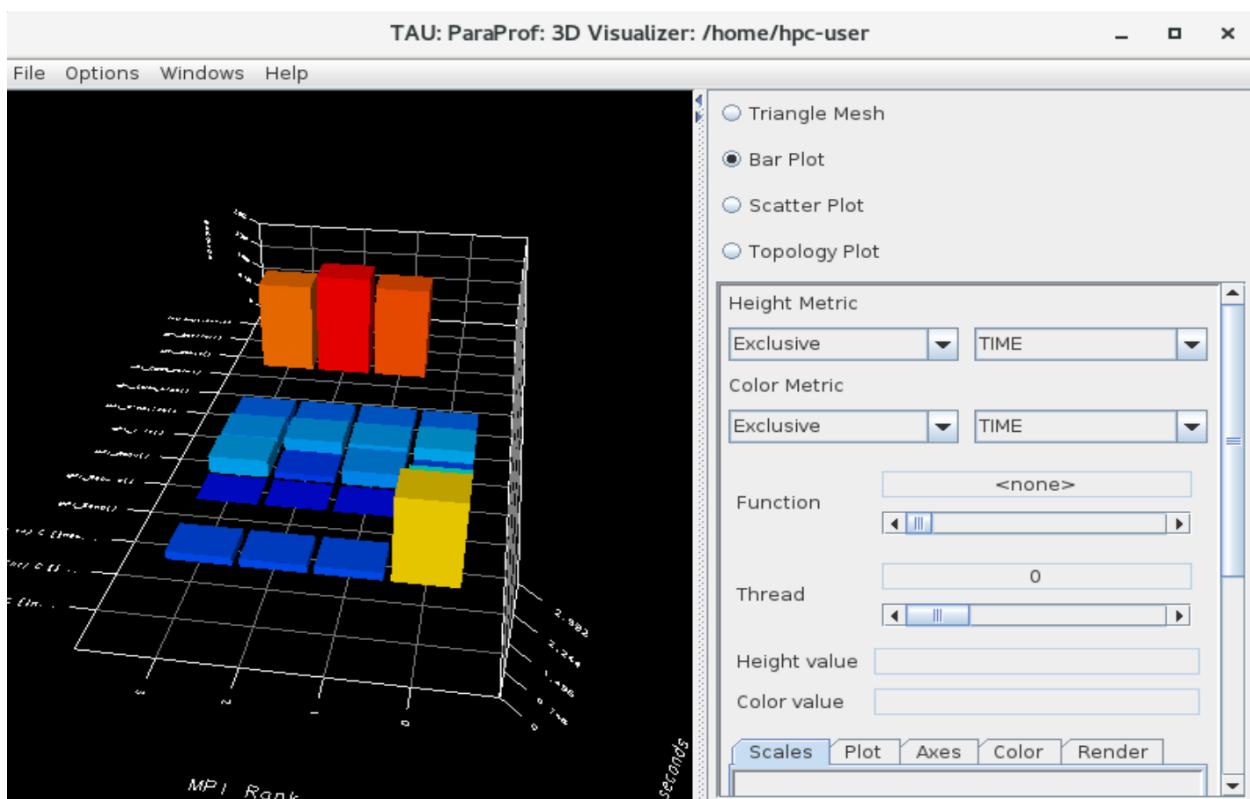
Jumpshot Visualization outputs for the program



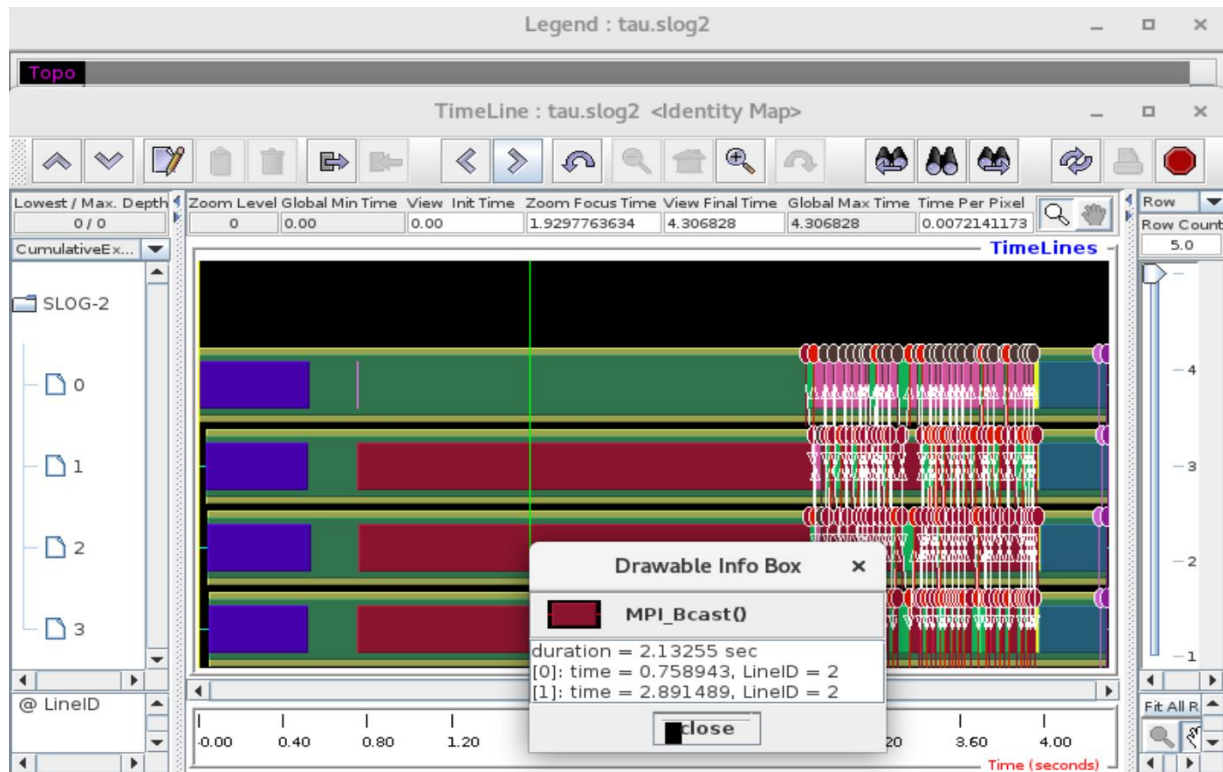
Pic 4: Jumpshot basic descriptive statistics.



Pic 5: Jumpshot basic descriptive statistics, Zoom in.

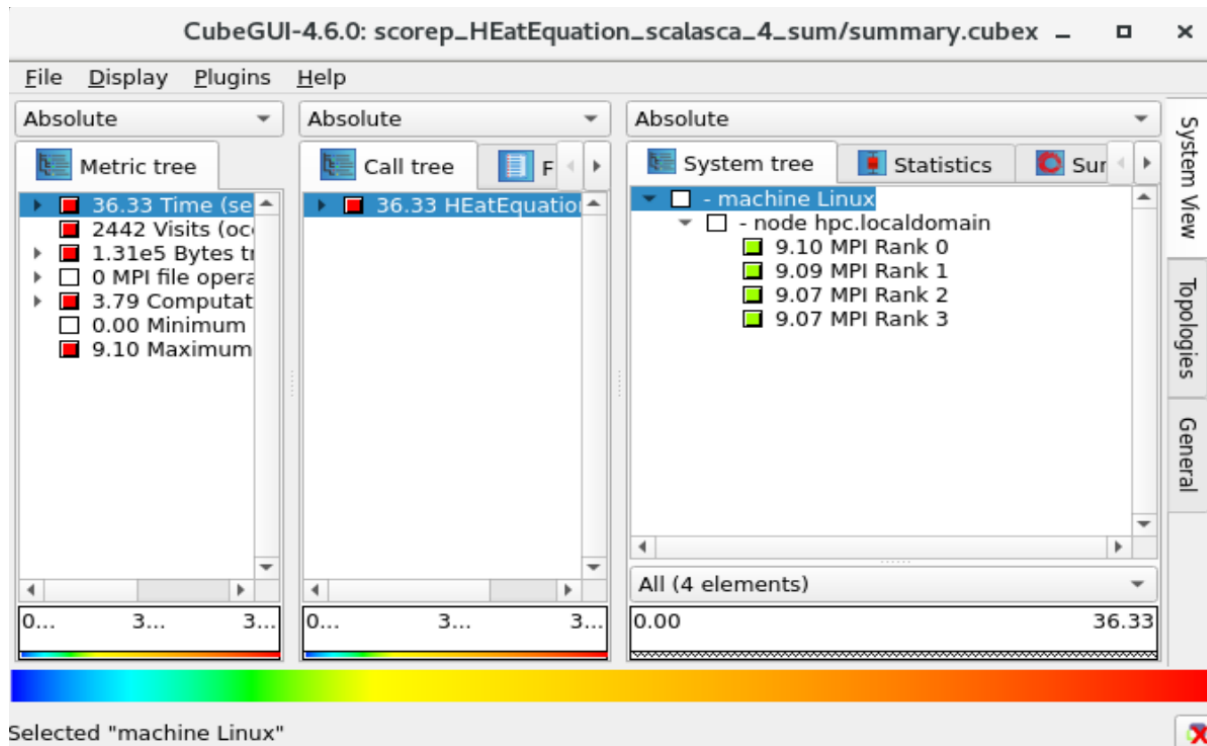


Pic 6: Bar plot for all program parts distribution.

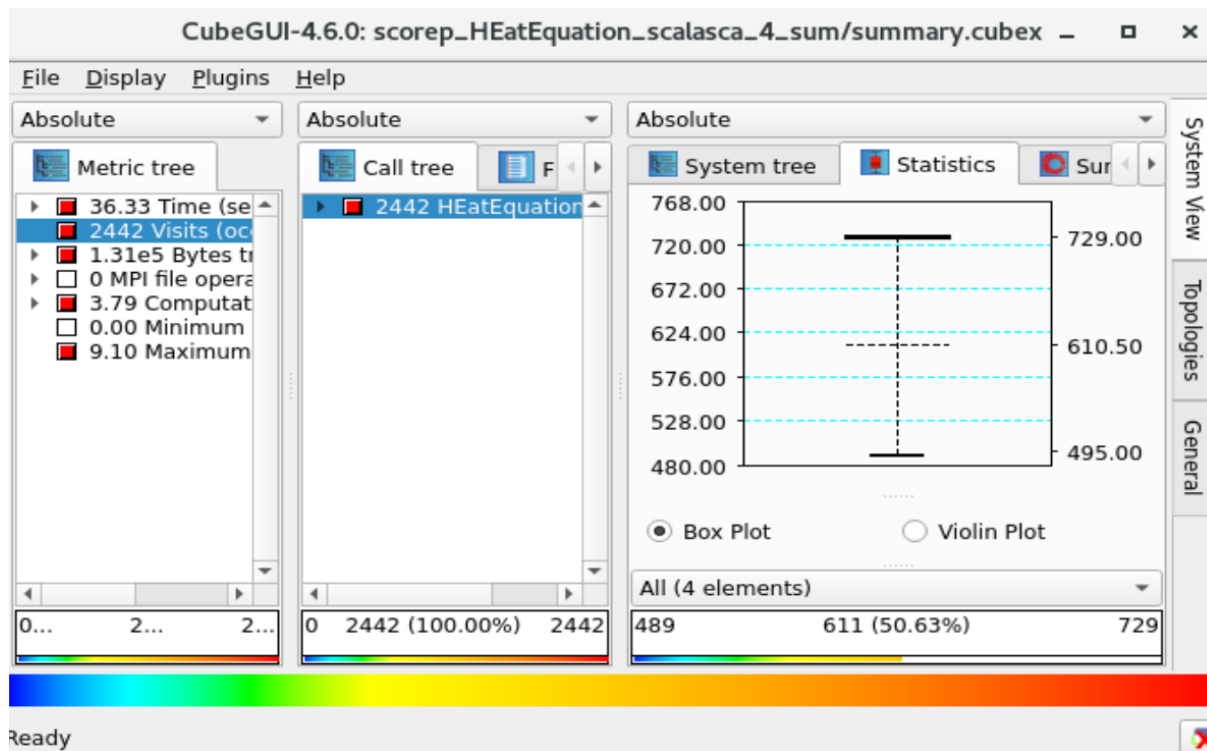


Pic 7: MPI_Bcast() function, takes the longest run time, and we can see that we have a lot of communication between the processors (due to they are sending the data each iteration).

Scalasca Screenshots:



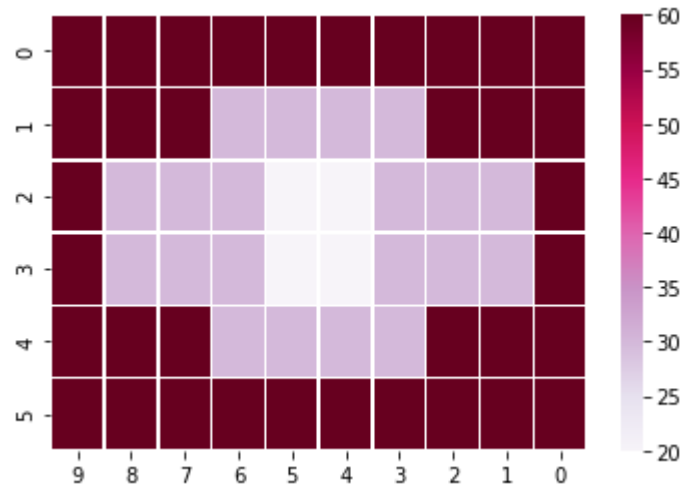
Pic 8: Scalasca system tree of run time screenshot.



Pic 9: Scalasca boxplot for visit distribution.

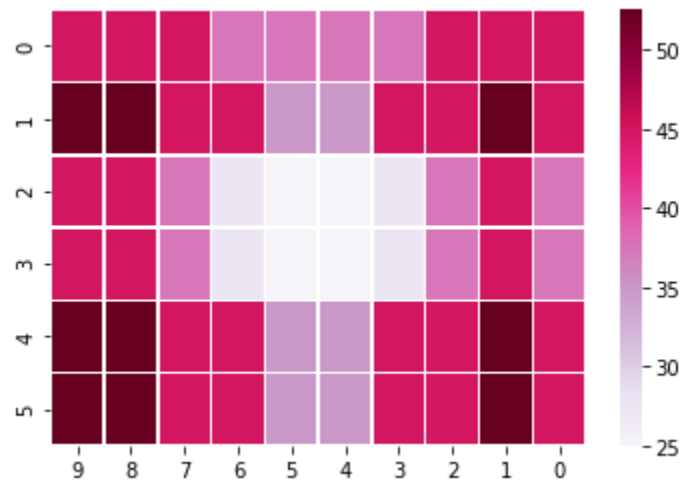
Heat Map of the temperature for the input & output data using Python

Input



Pic 10: Heat Map of the input data (before performing laplace equation).

Output



Pic 11: Heat Map of the output data (after performing laplace equation).

Conclusions

- The function MPI_Bcast takes the most time for the program, which is also the bottleneck for this program. We can see that also from the graphs and from the charts, it almost distributes uniformly among the “workers”, and equal to zero among the master (which makes sense).
- We can see that the number of “visits” is higher in processor 2 & 3 more than processors 0 & 4 – we can assume that it happened because the first processor takes the same number of data, and the remaining is going to the last processor. Processor 0 is the master and therefore he wasn’t involved much in the program execution.
- In Scalasca, we can notice the number of bytes that each processor sends. We can also see in Scalasca the number of time that every task takes to each processor (which we can also see in good visualization in Jumpshot graphs).
- This kind of program can present good performance among this kind of tasks such as image processing or doing a lot of calculation between data that needs to be “send” and “receive” to be continued.

2 Open Question on what we've learned so far:

***Right answers will be marked in YELLOW**

Question number 1:

If you call MPI_Send and there is no matching receive, which of the following are possible outcomes?

1. The message disappears.
2. The send fails.
3. The message is stored and delivered later on (if possible) or waits until a receive is posted (potentially waiting forever).
4. The send times out after some system-specified delay (e.g., a few minutes).

Question number 2:

The MPI receive routine has a parameter "count" – what does this mean?

1. The size of the incoming message (in bytes).
2. The size of the incoming message (in terms, e.g., integers).
3. The size you have reserved for storing the message (in bytes).
4. The size you have reserved for storing the message (in terms, e.g., integers).