Abstract

In this project, we've been asked to perform a Matrix Multiplication using MPI library in C language.

Some notes about the generic code:

- We had to use static memory only & not dynamic memory variable.
- Master does not include in the calculation of the matrix.
- The process will stop running if number of processes < 2.
- The code works for any number of rows, even if number of rows % processes != 0.
- We've ran the code on the Virtual Machine.

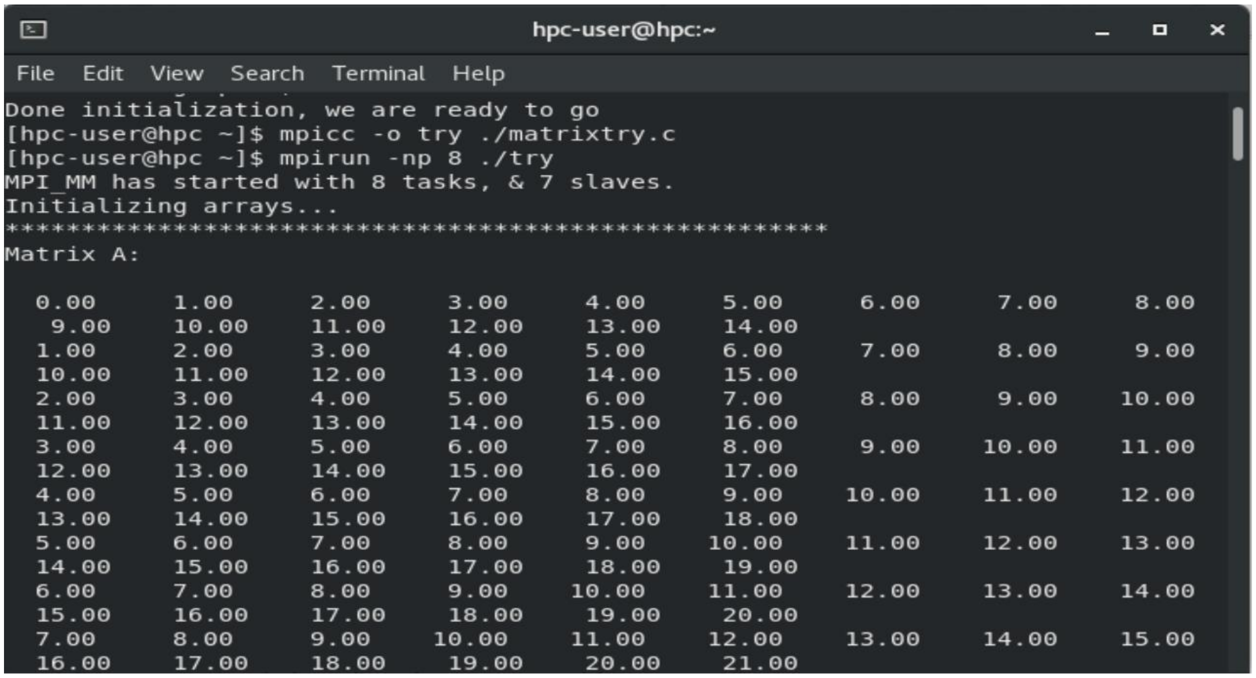
Some notes about the specific variable for this code (as requested):

- Number of rows in A (NRA) is 62.
- Number of columns in A (NCA) is 15.
- Number of columns in B (NCB) is 7.
- $A_{i,j} = i*j$
- $B_{i,j} = i+j$
- $C = A*B$

- We've ran this code on 8 processors.

*Note – The code is attached in another file calls "matrixtry.c" -which can be opened in any kind of notepad.

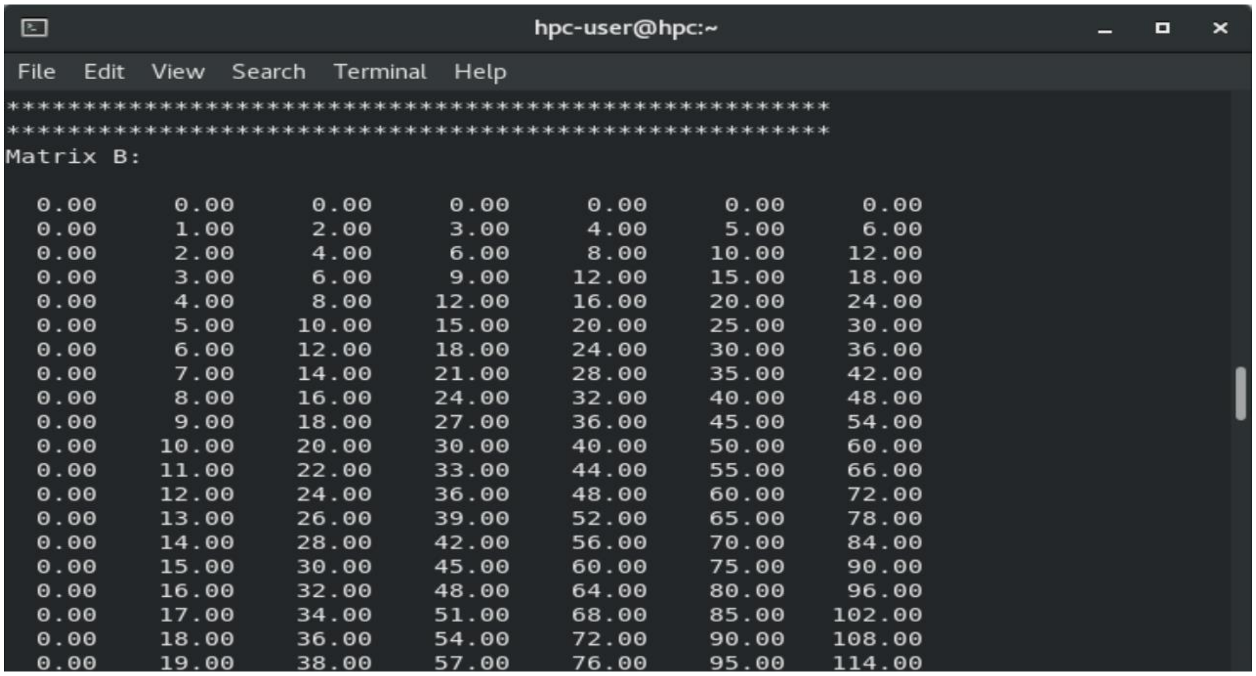
Screenshots of the output of the code for example (not the full output):



```
Done initialization, we are ready to go
[hpc-user@hpc ~]$ mpicc -o try ./matrixtry.c
[hpc-user@hpc ~]$ mpirun -np 8 ./try
MPI_MM has started with 8 tasks, & 7 slaves.
Initializing arrays...
******************************************************
Matrix A:

   0.00      1.00      2.00      3.00      4.00      5.00      6.00      7.00      8.00
    9.00     10.00     11.00     12.00     13.00     14.00
   1.00      2.00      3.00      4.00      5.00      6.00      7.00      8.00      9.00
  10.00     11.00     12.00     13.00     14.00     15.00
   2.00      3.00      4.00      5.00      6.00      7.00      8.00      9.00     10.00
  11.00     12.00     13.00     14.00     15.00     16.00
   3.00      4.00      5.00      6.00      7.00      8.00      9.00     10.00     11.00
  12.00     13.00     14.00     15.00     16.00     17.00
   4.00      5.00      6.00      7.00      8.00      9.00     10.00     11.00     12.00
  13.00     14.00     15.00     16.00     17.00     18.00
   5.00      6.00      7.00      8.00      9.00     10.00     11.00     12.00     13.00
  14.00     15.00     16.00     17.00     18.00     19.00
   6.00      7.00      8.00      9.00     10.00     11.00     12.00     13.00     14.00
  15.00     16.00     17.00     18.00     19.00     20.00
   7.00      8.00      9.00     10.00     11.00     12.00     13.00     14.00     15.00
  16.00     17.00     18.00     19.00     20.00     21.00
```

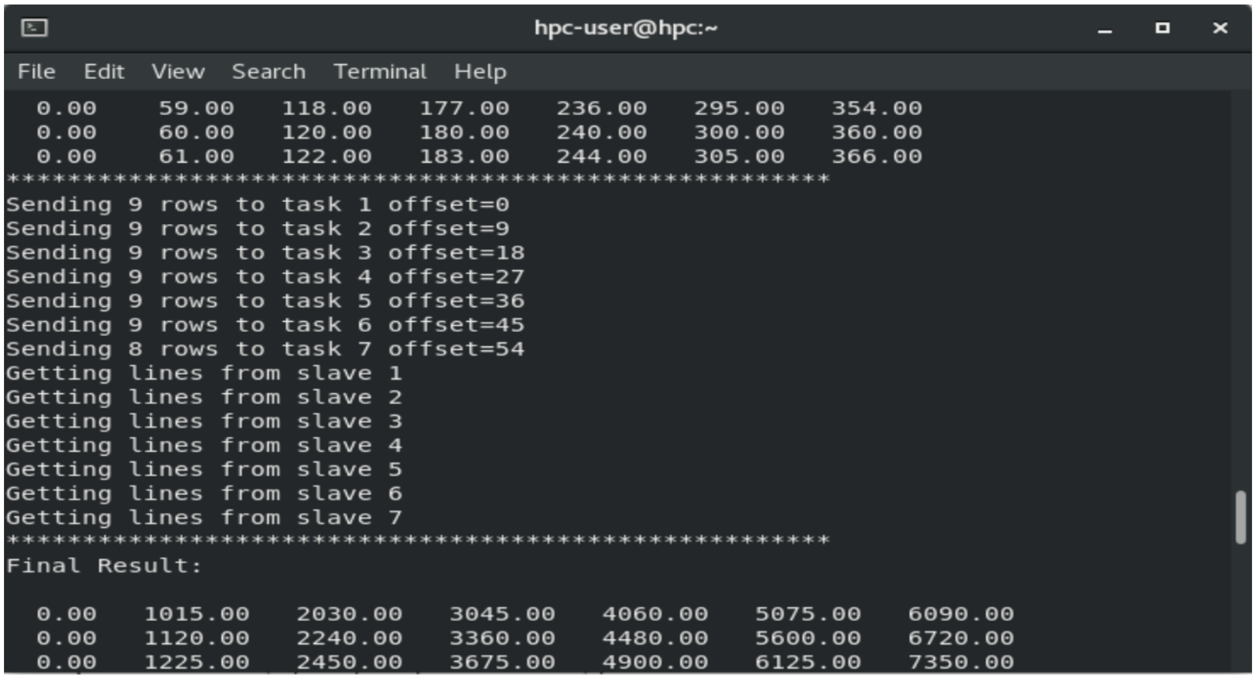
Pic 1: First screen shot for output.



```
******************************************************
******************************************************
Matrix B:

   0.00      0.00      0.00      0.00      0.00      0.00      0.00
   0.00      1.00      2.00      3.00      4.00      5.00      6.00
   0.00      2.00      4.00      6.00      8.00     10.00     12.00
   0.00      3.00      6.00      9.00     12.00     15.00     18.00
   0.00      4.00      8.00     12.00     16.00     20.00     24.00
   0.00      5.00     10.00     15.00     20.00     25.00     30.00
   0.00      6.00     12.00     18.00     24.00     30.00     36.00
   0.00      7.00     14.00     21.00     28.00     35.00     42.00
   0.00      8.00     16.00     24.00     32.00     40.00     48.00
   0.00      9.00     18.00     27.00     36.00     45.00     54.00
   0.00     10.00     20.00     30.00     40.00     50.00     60.00
   0.00     11.00     22.00     33.00     44.00     55.00     66.00
   0.00     12.00     24.00     36.00     48.00     60.00     72.00
   0.00     13.00     26.00     39.00     52.00     65.00     78.00
   0.00     14.00     28.00     42.00     56.00     70.00     84.00
   0.00     15.00     30.00     45.00     60.00     75.00     90.00
   0.00     16.00     32.00     48.00     64.00     80.00     96.00
   0.00     17.00     34.00     51.00     68.00     85.00    102.00
   0.00     18.00     36.00     54.00     72.00     90.00    108.00
   0.00     19.00     38.00     57.00     76.00     95.00    114.00
```

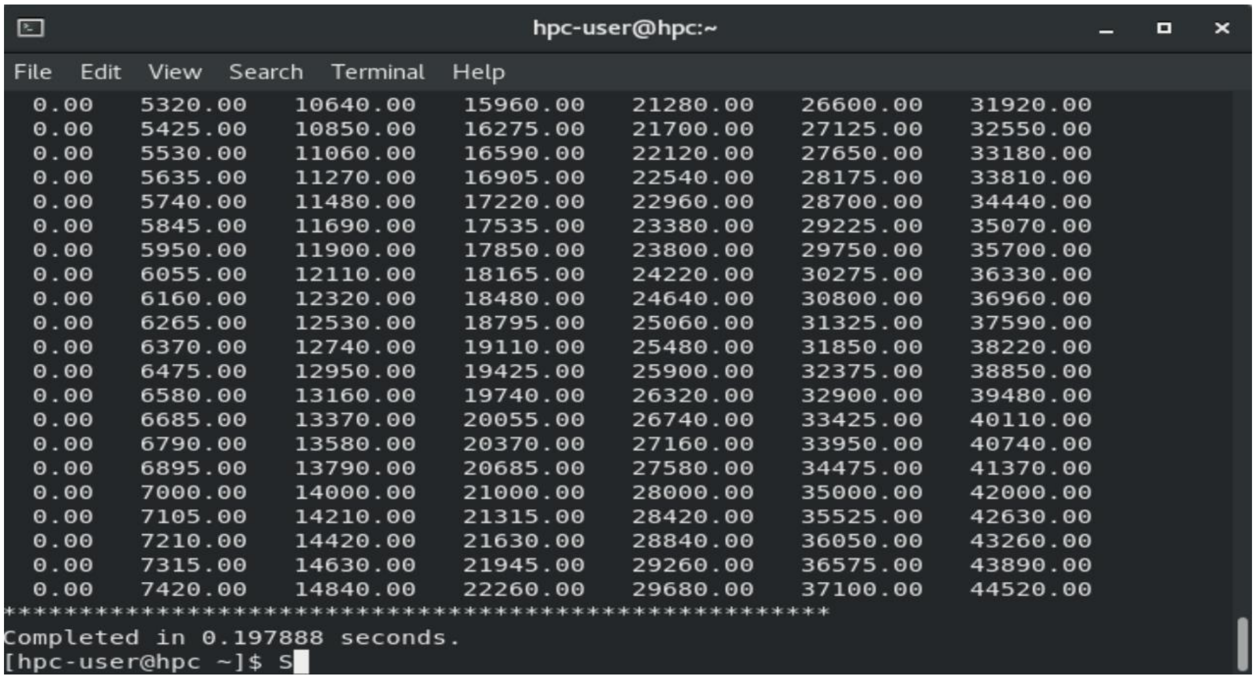
Pic 2: Second screen shot for output.



```
  0.00    59.00   118.00   177.00   236.00   295.00   354.00
  0.00    60.00   120.00   180.00   240.00   300.00   360.00
  0.00    61.00   122.00   183.00   244.00   305.00   366.00
******************************************************
Sending 9 rows to task 1 offset=0
Sending 9 rows to task 2 offset=9
Sending 9 rows to task 3 offset=18
Sending 9 rows to task 4 offset=27
Sending 9 rows to task 5 offset=36
Sending 9 rows to task 6 offset=45
Sending 8 rows to task 7 offset=54
Getting lines from slave 1
Getting lines from slave 2
Getting lines from slave 3
Getting lines from slave 4
Getting lines from slave 5
Getting lines from slave 6
Getting lines from slave 7
******************************************************
Final Result:

  0.00   1015.00   2030.00   3045.00   4060.00   5075.00   6090.00
  0.00   1120.00   2240.00   3360.00   4480.00   5600.00   6720.00
  0.00   1225.00   2450.00   3675.00   4900.00   6125.00   7350.00
```

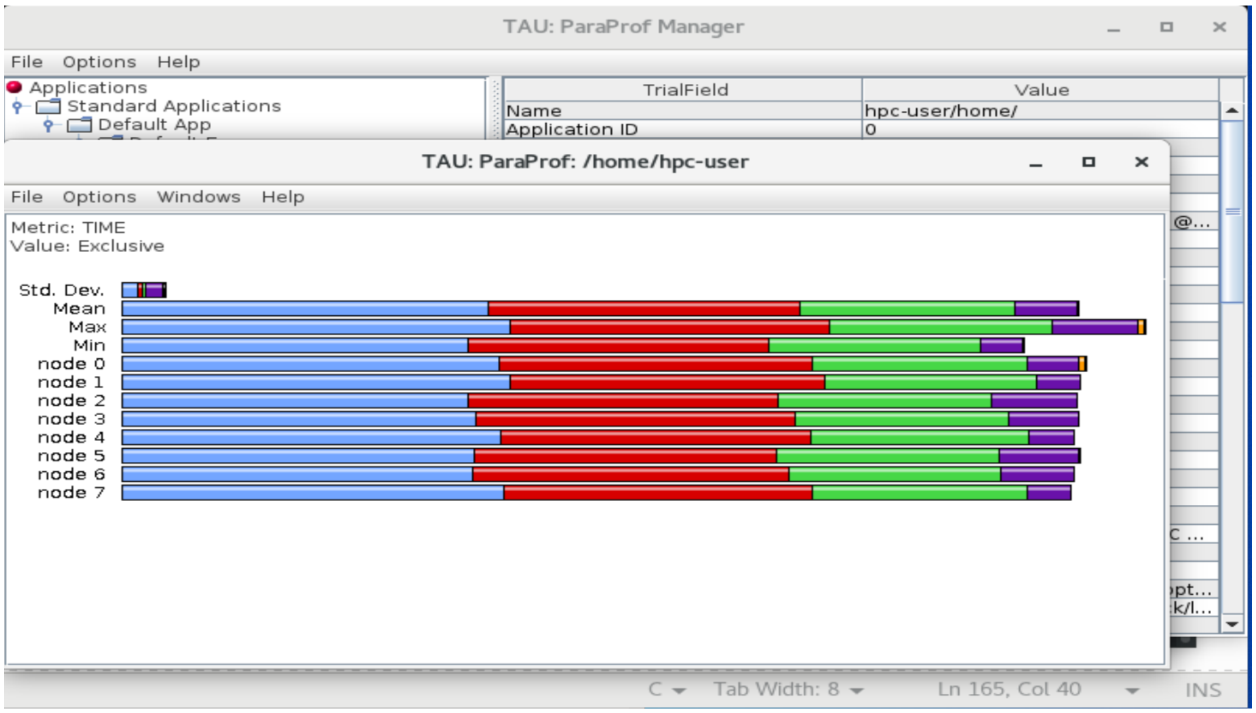
Pic 3: Third screen shot for output.



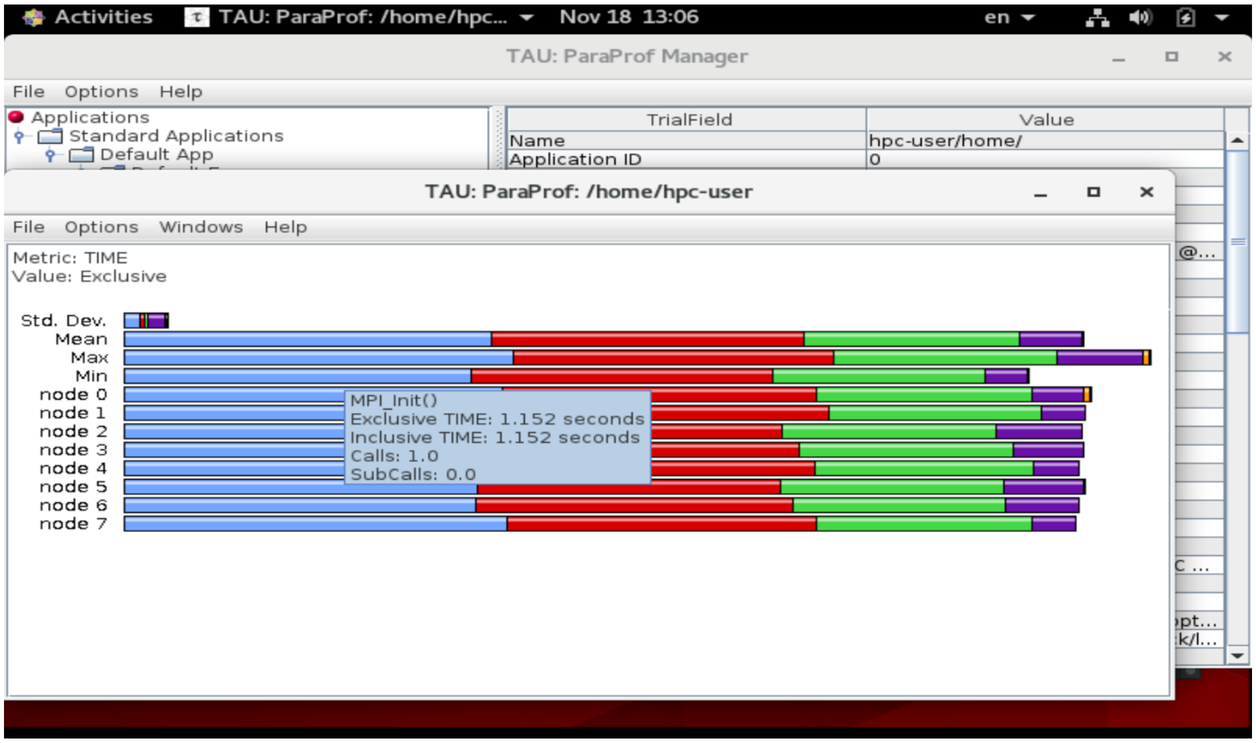
```
  0.00   5320.00   10640.00   15960.00   21280.00   26600.00   31920.00
  0.00   5425.00   10850.00   16275.00   21700.00   27125.00   32550.00
  0.00   5530.00   11060.00   16590.00   22120.00   27650.00   33180.00
  0.00   5635.00   11270.00   16905.00   22540.00   28175.00   33810.00
  0.00   5740.00   11480.00   17220.00   22960.00   28700.00   34440.00
  0.00   5845.00   11690.00   17535.00   23380.00   29225.00   35070.00
  0.00   5950.00   11900.00   17850.00   23800.00   29750.00   35700.00
  0.00   6055.00   12110.00   18165.00   24220.00   30275.00   36330.00
  0.00   6160.00   12320.00   18480.00   24640.00   30800.00   36960.00
  0.00   6265.00   12530.00   18795.00   25060.00   31325.00   37590.00
  0.00   6370.00   12740.00   19110.00   25480.00   31850.00   38220.00
  0.00   6475.00   12950.00   19425.00   25900.00   32375.00   38850.00
  0.00   6580.00   13160.00   19740.00   26320.00   32900.00   39480.00
  0.00   6685.00   13370.00   20055.00   26740.00   33425.00   40110.00
  0.00   6790.00   13580.00   20370.00   27160.00   33950.00   40740.00
  0.00   6895.00   13790.00   20685.00   27580.00   34475.00   41370.00
  0.00   7000.00   14000.00   21000.00   28000.00   35000.00   42000.00
  0.00   7105.00   14210.00   21315.00   28420.00   35525.00   42630.00
  0.00   7210.00   14420.00   21630.00   28840.00   36050.00   43260.00
  0.00   7315.00   14630.00   21945.00   29260.00   36575.00   43890.00
  0.00   7420.00   14840.00   22260.00   29680.00   37100.00   44520.00
******************************************************
Completed in 0.197888 seconds.
[hpc-user@hpc ~]$ S
```

Pic 4: Fourth screen shot for output.

Jumpshot Visualization outputs for the program

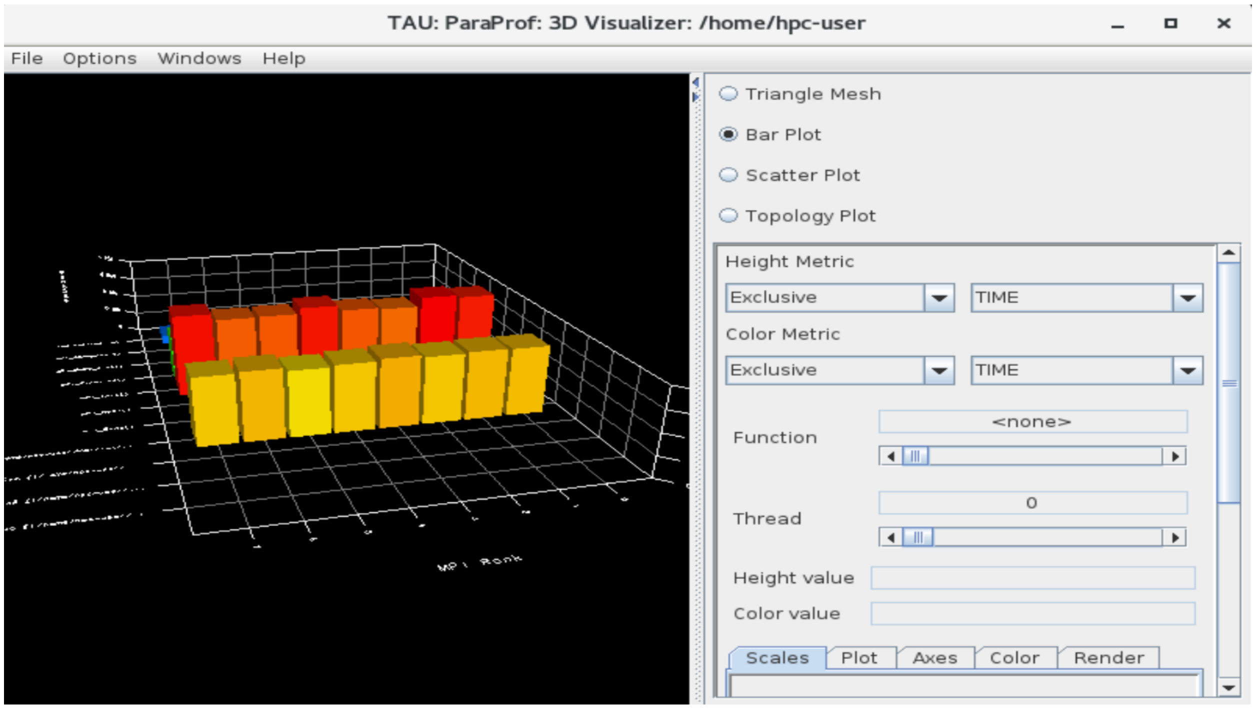


Pic 5: Jumpshot basic descriptive statistics.



```
MPI_Init()
Exclusive TIME: 1.152 seconds
Inclusive TIME: 1.152 seconds
Calls: 1.0
SubCalls: 0.0
```

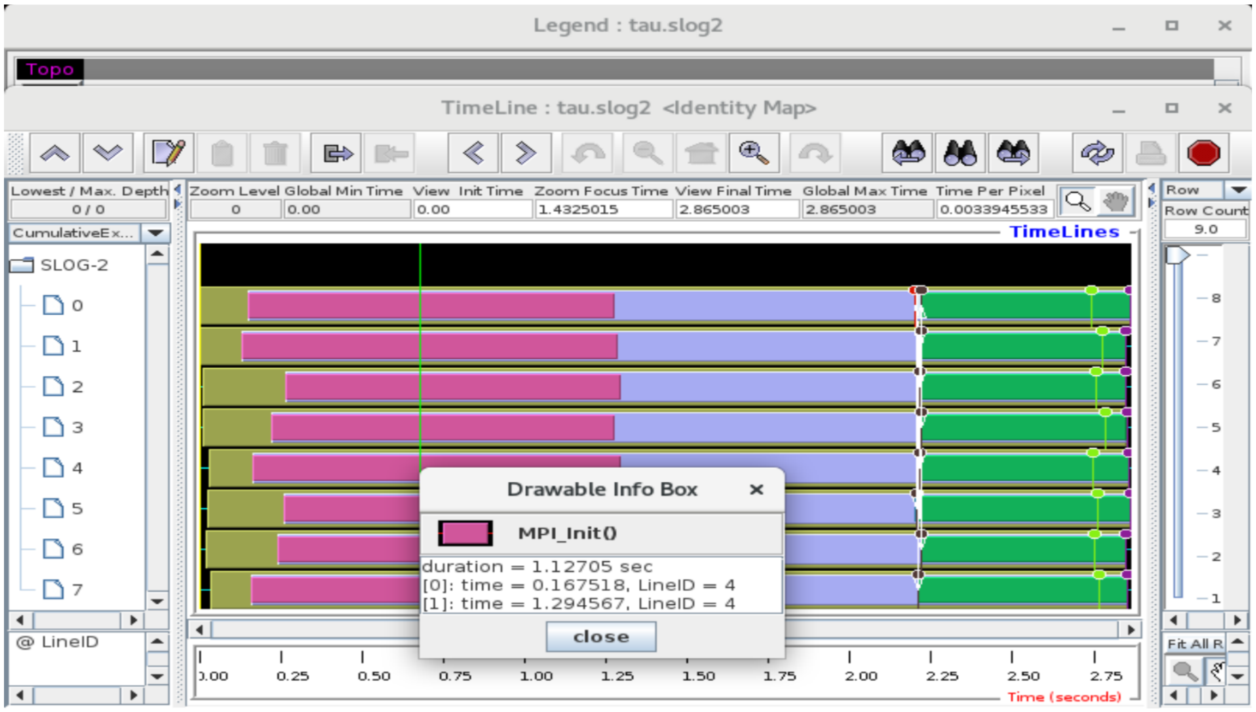
Pic 6: Jumpshot basic descriptive statistics, Zoom in.


TAU: ParaProf: 3D Visualizer: /home/hpc-user
File Options Windows Help
MPI Rank
Scales Plot Axes Color Render

( ) Triangle Mesh
(●) Bar Plot
( ) Scatter Plot
( ) Topology Plot

Height Metric: Exclusive, TIME
Color Metric: Exclusive, TIME
Function: <none>
Thread: 0
Height value:
Color value:

Pic 7: Bar plot for all program parts distribution.



Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel	Row Count
0 / 0	0	0.00	0.00	1.4325015	2.865003	2.865003	0.0033945533	9.0

CumulativeEx...

Drawable Info Box

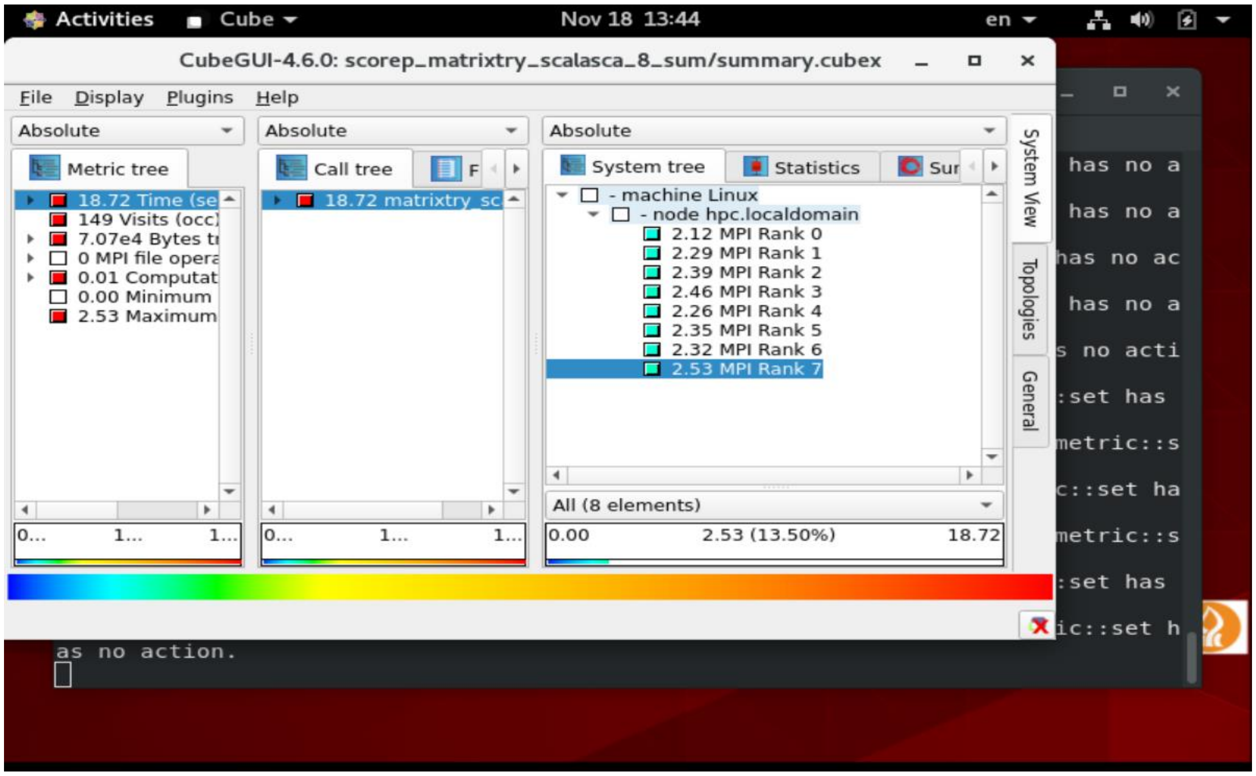
MPI_Init()

```
duration = 1.12705 sec
[0]: time = 0.167518, LineID = 4
[1]: time = 1.294567, LineID = 4
```

close

Pic 8: MPI_Init() function, takes the longest run time.

Scalasca Screenshots:



```
CubeGUI-4.6.0: scorep_matrixtry_scalasca_8_sum/summary.cubex
```

Metric tree:

```
18.72 Time (se
149 Visits (occ)
7.07e4 Bytes tı
0 MPI file opera
0.01 Computat
0.00 Minimum
2.53 Maximum
```

Call tree:

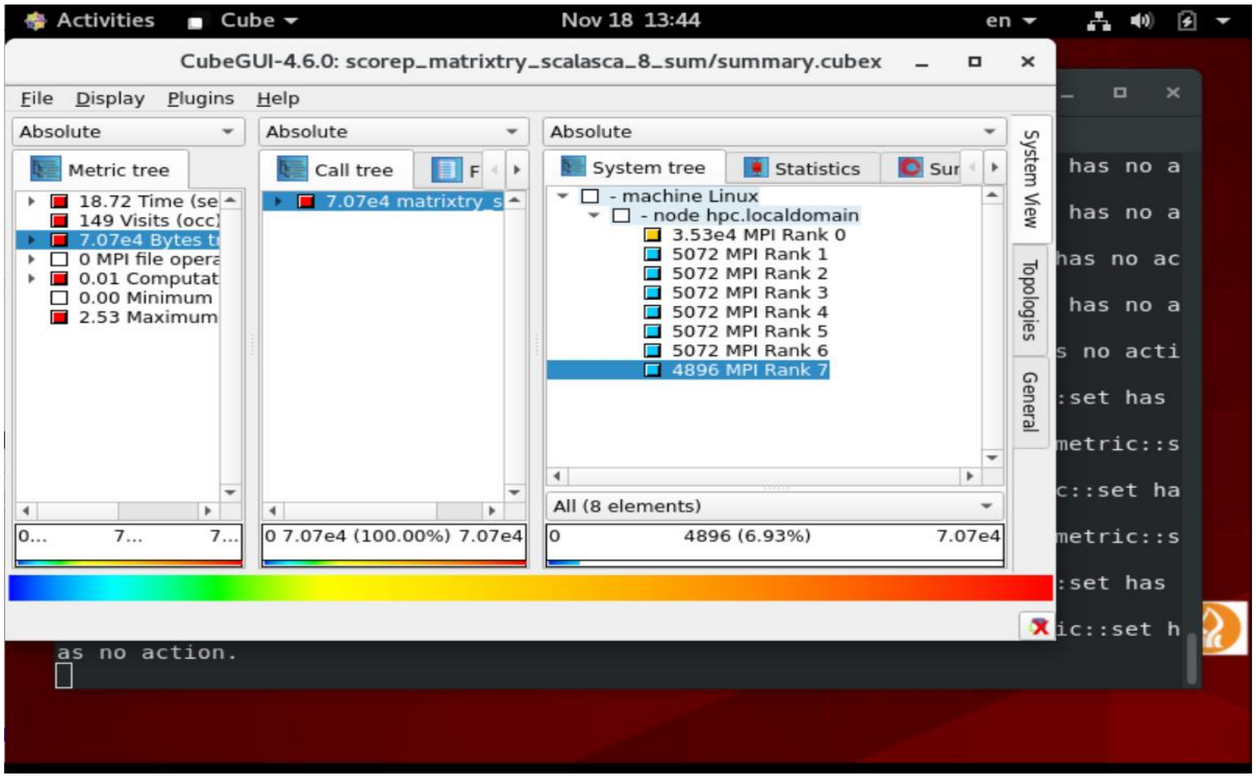
```
18.72 matrixtry_sc
```

System tree:

```
- machine Linux
  - node hpc.localdomain
      2.12 MPI Rank 0
      2.29 MPI Rank 1
      2.39 MPI Rank 2
      2.46 MPI Rank 3
      2.26 MPI Rank 4
      2.35 MPI Rank 5
      2.32 MPI Rank 6
      2.53 MPI Rank 7
All (8 elements)
0.00        2.53 (13.50%)        18.72
```

```
has no a
has no a
has no ac
has no a
s no acti
:set has
metric::s
c::set ha
metric::s
:set has
ic::set h
as no action.
```

Pic 9: Scalasca system tree of run time screenshot.



```
- machine Linux
  - node hpc.localdomain
      3.53e4 MPI Rank 0
      5072 MPI Rank 1
      5072 MPI Rank 2
      5072 MPI Rank 3
      5072 MPI Rank 4
      5072 MPI Rank 5
      5072 MPI Rank 6
      4896 MPI Rank 7
```

Pic 10: Scalasca system tree memory taken of each processor.

Conclusions

- The function MPI_Init 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.

- In the functions MPI_Init, main, MPI_Finalize – the difference between the MAX to Min is very low, we can assume from that they are almost distributes uniformly, and it takes to all processes almost the same time to preform these functions and tasks.

- We can see MPI_Recv only in node 0, which is the master, which it also makes sense that only the master receives the data from the slaves/workers in this program.

- 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).

- Parallel programing for matrix multiplication for huge number of rows & columns is very effective in the way we mentioned in the code.

- This parallel programing can be also effective to image processing while making any task on a huge number of matrix/arrays/pixel/etc.

- For these kinds of programs such as: image processing, manipulating huge matrices – I'll recommend using MPI_Barrier between any task/manipulation for some reasons:
    - If we want to gather all the data we've worked on till now and make a "layer" before we start manipulating the data again.

- If we get into a situation that two processes (GPU) want to transmit data to each other at the same time is very likely to happen, which could badly reduce the program efficiency.

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

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

Question number 1:

After initiating an MPI program with "mpirun -np 4 ./mympiprogram", what does the call to MPI_Init do ?

1. Create the 4 parallel processes.
2. Start program execution.
3. Enable the 4 independent programs subsequently to communicate with each other.
4. Create the 4 parallel threads.

Question number 2:

If you call MPI_Recv and there is no incoming message. What happens?

1. The Recv fails with an error.
2. The Recv reports that there is no incoming message.
3. The Recv waits until a message arrives (potentially waiting forever).
4. The Recv times out after some system-specified delay (e.g a few minutes).