**Lab 2**

**Testing Conditions**

The programs were tested on a machine with the following specifications:

* 6 core Intel(R) Xeon(R) CPU E5-2640 0 @ 2.50GHz.
* x86\_64 architecture
* CPU Clock Frequency: 2500.00 MHz.
* Ubuntu 16.04.1 LTS (Xenial Xerus) operating system.

Times for their execution were determined using the time command to run each command.

**Limitations:**

Quoting from the man page for the time command:

*The elapsed time is not collected atomically with the execution of the program; as a result, in bizarre circumstances (if the time command gets stopped or swapped out in between when the*

*program being timed exits and when time calculates how long it took to run), it could be much*

*larger than the actual execution time.*

We attempted to remove this limitation of our measurement tool by executing each test multiple times. While this doesn’t necessarily remove the error in our measurement it at least improves our precision.

**Question 1**

**Implementation**

**Results**

**Discussion**

**Question 2**

**Implementation**

The decomposition method used mirrored heavily what the professor suggested in slide 37 of lecture 8: Programming Overview. The method allocates a set of rows to each processor. Space is allocated to hold two *ghost rows* one which contains the top row of data in the subsequent row of the matrix and one which contains the bottom row of the previous row in the matrix. Each process will send their top and bottom rows to the ghost rows of their neighbors. Subsequently the program will calculate the current value at each node in the matrix by following the equations outlined in the problem statement.

**Results**

The program’s output was tested on a 512 X 512 grid using 1, 2, 4, 16 and 32 processes.

The executable was run using the following command:

time mpirun -np <NP> grid\_512\_512 5000

Where <NP> is equal to the number of processes that were used for that particular run.

This resulted in the following results:

|  |  |  |
| --- | --- | --- |
| **Trial** | **Number of Processes** | **Real Time (s)** |
| 1 | 1 | 16.404 |
| 2 | 1 | 16.368 |
| 3 | 1 | 16.371 |
| 4 | 1 | 16.339 |
| 5 | 2 | 8.779 |
| 6 | 2 | 8.770 |
| 7 | 2 | 8.731 |
| 8 | 2 | 8.479 |
| 9 | 4 | 4.668 |
| 10 | 4 | 4.610 |
| 11 | 4 | 4.663 |
| 12 | 4 | 4.842 |
| 13 | 8 | 4.928 |
| 14 | 8 | 5.020 |
| 15 | 8 | 5.051 |
| 16 | 8 | 5.008 |
| 17 | 16 | 6.412 |
| 18 | 16 | 6.606 |
| 19 | 16 | 6.523 |
| 20 | 16 | 6.507 |
| 21 | 32 | 6.163 |
| 22 | 32 | 5.834 |
| 23 | 32 | 6.044 |
| 24 | 32 | 6.052 |

And subsequently the following graph:

**Discussion**

The maximum speedup was found to be around 4-8 processors. This matches well with the system that we tested the application on which was around 6 processors. Additional processors add unnecessary overhead without leading to quicker execution. The improvement between 16 and 32 processes may be explained by the limitations of the time tool.