Assignment #4 (HPC . ICS632 Nov. 2015) ---------- Ehsan Kourkchi

To generate all executable files use Makefile:

```
> make all
smpicc -O4  hw04.skeleton.c -D MODE=0 -lm -o matmul_init
smpicc -O4  hw04.skeleton.c -D MODE=1 -lm -o matmul_outerproduct_v1
smpicc -O4  hw04.skeleton.c -D MODE=1 -lm -o matmul_outerproduct_v1_omp
-fopenmp -D OMP=1
smpicc -O4  hw04.skeleton.c -D MODE=2 -lm -o matmul_outerproduct_v2
smpicc -O4  hw04.skeleton.c -D MODE=3 -lm -o matmul_outerproduct_v3
```

To find the value to pass to `–cfg=smpi/running_power`, we use `calibrate_flops.py` and the resulting calibration factor turns out to be: `0.047`, on my desktop computer.

Question1: Creating the matrices

How to run the code for 4 processors and matrix size of 8*8:

```
> smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:0.047 -np 4 -platform
cluster_1600.xml -hostfile hostfile_1600.bin ./matmul_init -n 8
```

```
Matrix size: 8x8
# of processors: 4

Block of A on rank 0 at coordinates (0,0)
0.000000 0.000000 0.000000 0.000000
1.000000 1.000000 1.000000 1.000000
2.000000 2.000000 2.000000 2.000000
3.000000 3.000000 3.000000 3.000000
Block of A on rank 1 at coordinates (0,1)
0.000000 0.000000 0.000000 0.000000
1.000000 1.000000 1.000000 1.000000
2.000000 2.000000 2.000000 2.000000
3.000000 3.000000 3.000000 3.000000
Block of A on rank 2 at coordinates (1,0)
4.000000 4.000000 4.000000 4.000000
5.000000 5.000000 5.000000 5.000000
6.000000 6.000000 6.000000 6.000000
7.000000 7.000000 7.000000 7.000000
Block of A on rank 3 at coordinates (1,1)
4.000000 4.000000 4.000000 4.000000
5.000000 5.000000 5.000000 5.000000
6.000000 6.000000 6.000000 6.000000
7.000000 7.000000 7.000000 7.000000
Block of B on rank 0 at coordinates (0,0)
0.000000 1.000000 2.000000 3.000000
1.000000 2.000000 3.000000 4.000000
```

```
2.000000 3.000000 4.000000 5.000000
3.000000 4.000000 5.000000 6.000000
Block of B on rank 1 at coordinates (0,1)
4.000000 5.000000 6.000000 7.000000
5.000000 6.000000 7.000000 8.000000
6.000000 7.000000 8.000000 9.000000
7.000000 8.000000 9.000000 10.000000
Block of B on rank 2 at coordinates (1,0)
4.000000 5.000000 6.000000 7.000000
5.000000 6.000000 7.000000 8.000000
6.000000 7.000000 8.000000 9.000000
7.000000 8.000000 9.000000 10.000000
Block of B on rank 3 at coordinates (1,1)
8.000000 9.000000 10.000000 11.000000
9.000000 10.000000 11.000000 12.000000
10.000000 11.000000 12.000000 13.000000
11.000000 12.000000 13.000000 14.000000
```

Question 2: Multiplying the matrices

In this implementation, we use MPI_Bcast for broadcasting the chunk of matrices across rows/columns.

```
> smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:0.047 -np 4 -platform
cluster_1600.xml -hostfile hostfile_1600.bin ./matmul_outerproduct_v1 -n 8
--verbose


 Matrix size: 8x8
 # of processors: 4

 ---- Block of C on rank 0 at coordinates (0,0) ----
0.000000 0.000000 0.000000 0.000000
28.000000 36.000000 44.000000 52.000000
56.000000 72.000000 88.000000 104.000000
84.000000 108.000000 132.000000 156.000000
**************
rank: 0  checksum: 960.000000
**************
 ---- Block of C on rank 2 at coordinates (1,0) ----
112.000000 144.000000 176.000000 208.000000
140.000000 180.000000 220.000000 260.000000
168.000000 216.000000 264.000000 312.000000
196.000000 252.000000 308.000000 364.000000
**************
rank: 2  checksum: 3520.000000
```

```
 ---- Block of C on rank 1 at coordinates (0,1) ----
0.000000 0.000000 0.000000 0.000000
60.000000 68.000000 76.000000 84.000000
120.000000 136.000000 152.000000 168.000000
180.000000 204.000000 228.000000 252.000000
**************
rank: 1  checksum: 1728.000000
**************
 ---- Block of C on rank 3 at coordinates (1,1) ----
240.000000 272.000000 304.000000 336.000000
300.000000 340.000000 380.000000 420.000000
360.000000 408.000000 456.000000 504.000000
420.000000 476.000000 532.000000 588.000000
**************
rank: 3  checksum: 6336.000000
**************

 checksum:                12544.000000
 Analytical Sum:          12544.000000
 ** OK ** The Multiplication sounds to be fine.

 Computation + Communication time (sec): 0.001
 Computation time (sec): 0.000
 Communication time (sec): 0.001
```

Question3: Impact of the network speed

Matrix size: N=1600

As seen in Image 1, for the ideal network configurations, speed-up increases as the number of processes increases. However, for the real network, increasing the number of processors does not help that much and the maximum speed-up is around 10 when using 16 processors.

The reason is that, in our implementation, we use **`MPI-Bcast`** functions which block the computation until all processes receive their needed piece of the matrix.
Increasing the number of processors increases the number of send and receive and hence the dead time when process is blocked. Therefor, in this

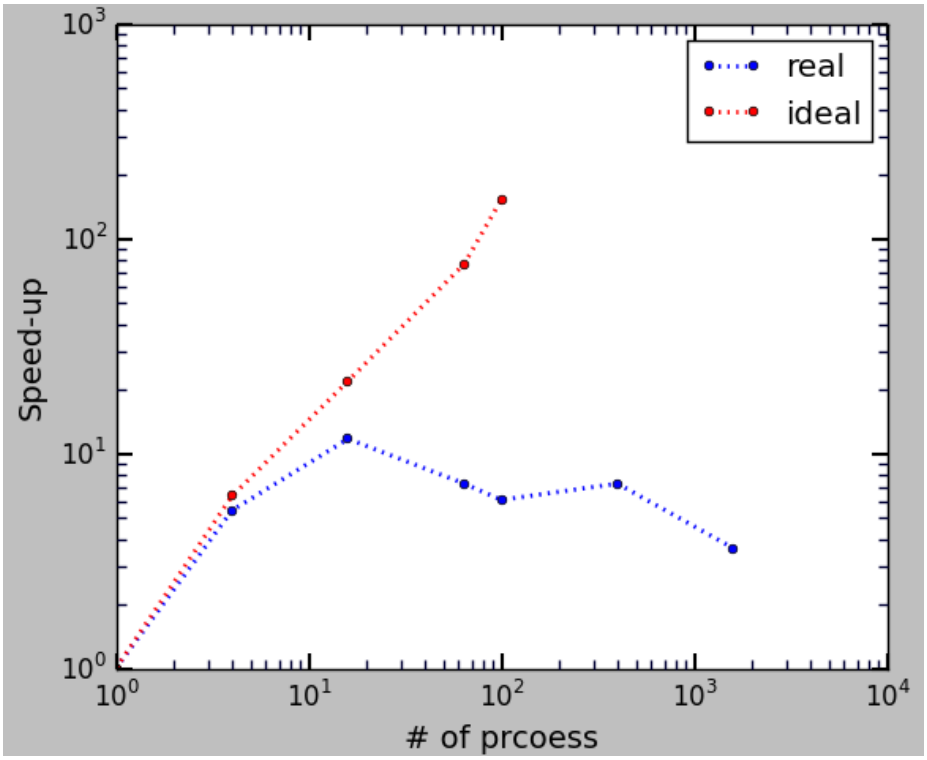

Image 1: Red: for the ideal network, (i.e. infinite bandwidth, no latency). Blue: Realistic network (bw="100Gbps" lat="20us")

case increasing the number of processors does not help at al. In the ideal world, dead time is zero, since and send and receive are done at the same time.

In Image 1, for the ideal case, the speed-up is ideal. Using 1600 processors, we expect to have a running time which is 1600 faster than a single processor. We did not continue the red curve for 400 and 1600 processors. The expectation is a linear increase.

In the realistic case,
1600 possessors → `T1600 = 0.042 (sec)`
1 processors → `T0 = 0.152 (sec)`

speed-up = T0/T1600 = 3.62

expected speed-up in ideal case: 1600

Parallel efficiency of the realistic case when using 1600 processors: `3.62/1600 =` **`0.2%`**

The conclusion is that in real machines increasing the number of processors does not necessarily result in a better efficiency and shorter running time. Due to the nature of MPI implementation, the parallel efficiency might have an upper bound due to the network communication, dead waiting time and so.

Question 4: Communication share

p = 16 processors
N: 800:3200

As seen in Image 2, computation time increases with the size of matrix, which is trivial since the complexity of matrix multiplication is O(N^3).

On the other hand, compared to computation time, the communication time is fairly constant, and increases a little bit when the matrix size increases.

Therefore, for larger matrices, it makes more sense to use a parallel code. For small matrices, the communication time would be dominant and one is better to run the code in a serial fashion.

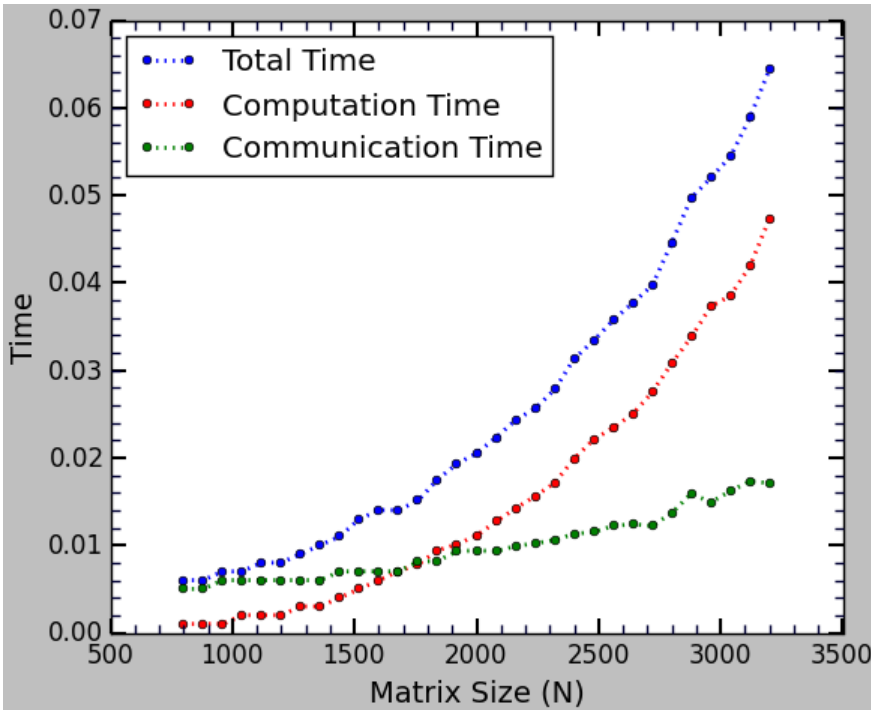

Image 2: number of processors: p=16 and time is in second.

Image 3, is the same as image 2 but for the normalized time. Since the computation complexity is O(N^3), we normalize the running time by the total number of matrix elements. Trivially, the red curves, i.e. the computation time becomes flat. Communication time per matrix element decreases and interestingly the total running time per matrix element decreases. This is a good news for multiplying large matrices.

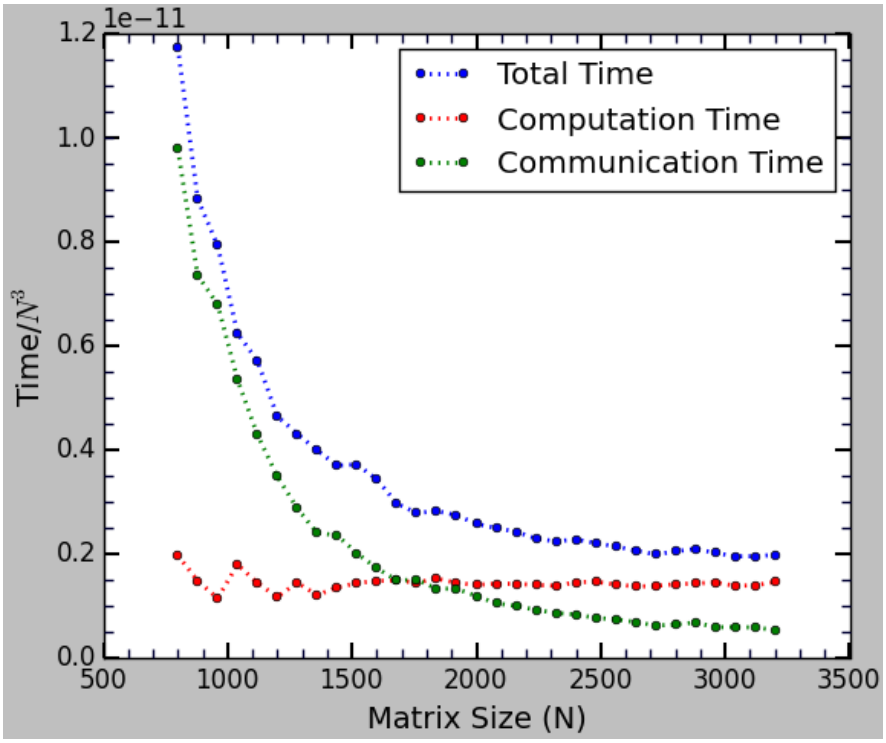

Image 3: Normalized time. Number of processors is 16 and time is in second.

Question5: "Cheating" the simulation

matmul_outerproduct_v2

In this version, only the calculation part is simulated.

matmul_outerproduct_v3

In this version, we also use `SMPI_SHARED_MALLOC` and `SMPOI_SHARED_FREE`.

To simulate the calculation part, we set:
flops = pow(n,3) * FLOP_CALIBRATION_FACTOR/1.E10

First we run `matmul_outerproduct_v1`, which is the actual code that actually does the matrix multiplication.

```
i.e. > smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:0.047 -np 16
-platform cluster_1600.xml -hostfile hostfile_1600.bin ./matmul_outerproduct_v1 -n
2000

Matrix size: 2000x2000
 # of processors: 16
 Computation + Communication time (sec): 0.022
 Computation time (sec): 0.011
 Communication time (sec): 0.011
```

Then we run `matmul_outerproduct_v3` with different flop factors until we get the same running time:

```
i.e. > smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:0.047 -np 16
-platform cluster_1600.xml -hostfile hostfile_1600.bin ./matmul_outerproduct_v3 -n
2000 -f 0.22

Matrix size: 2000x2000
 # of processors: 16
Computation + Communication time (sec): 0.022
Computation time (sec): 0.011
Communication time (sec): 0.011
```

---

1) Re-calibrating the simulation

The best value based on several try and error runs:
FLOP_CALIBRATION_FACTOR = **0.22**

Re-testing with 100 processors:

```
> smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:0.047 -np 100 -platform
cluster_1600.xml -hostfile hostfile_1600.bin ./matmul_outerproduct_v1 -n 2000

> smpirun --cfg=smpi/bcast:mpich --cfg=smpi/running_power:0.047 -np 100 -platform
cluster_1600.xml -hostfile hostfile_1600.bin ./matmul_outerproduct_v3 -n 2000 -f
0.22
```

Both of them rerun the same result:

```
Computation + Communication time (sec): 0.050
Computation time (sec): 0.002
Communication time (sec): 0.048
```

2) Running new experiments:

Image 4, shows our results for 100 processors and different matrix sizes, when using the totally simulated calculations … Flop factor is 0.22.
Matrix dimensions are set to be in the range of 800 to 30,000 with the steps of 800.

As seen on the left diagram, increasing the matrix size, does not increases the communication time substantially and therefore as concluded in the answer to Question 5, large matrices are find.

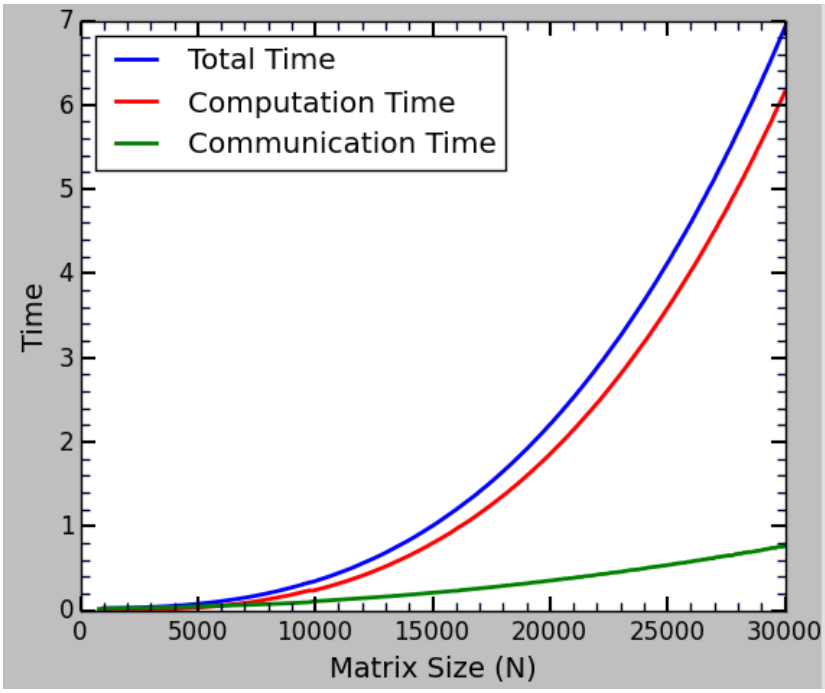

Image 4: No. of processors is 100, time is second.

Image 5 shows the normalized time. It has the same meaning as Image 3. The normalized computation time is flat as expected. The communication time per matrix element decreases as the size of matrices increase. As the size of the matrices increases, the total cost of computation asymptotically reaches to the cost of computation, and one can ignore the cost of communication.

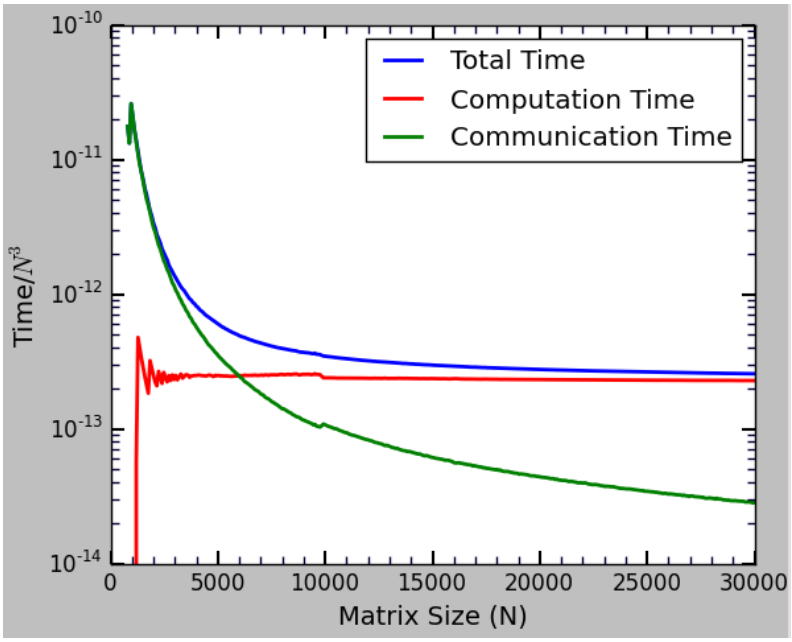

Image 5: The same as Image 4 but for the normalized time and in logarithmic scale.

Question 6: Running on real clusters ..

This part has been done in two different clusters.

1. **University of Hawaii HPC cluster**

each node: 2 Intel Xeon E5-2680v2 "Ivy Bridge" 10-core, (25 MB cache 2.8GHz processors)
→ each node has 20 cores

2. **University of Wyoming Mt. Moran cluster**

each node: two 8-core Intel E5-2670 (Sandy Bridge), (20 MB cache, 2.6 GHz processors)
→ each node has 16 cores

We have run our code with two different configuration on each cluster. In each mode we allocate 4 nodes.

- MPI: Using 16 sequential processor on each node, for a total 4 * 16 = 64 MPI tasks. See below for the slurm script.

```
#!/bin/bash
#SBATCH -o size-<matrix_size>-iter-%a-%N.txt
#SBATCH --time=00:02:00
#SBATCH -p ics632.q
#SBATCH --ntasks-per-node 16
#SBATCH --nodes 4

mpirun  ./matmul_outerproduct_v1 -n <matrix_size>
```

- MPI+OMP: Using one 16-way multi-threaded (with OpenMP) MPI process per node, for a total 4 MPI task (one per node). See below for the slurm script.

```
#!/bin/bash

#SBATCH -o size-<matrix_size>-iter-%a-%N.omp.txt
#SBATCH --time=00:02:00
#SBATCH --ntasks-per-node 1
#SBATCH --nodes 4
#SBATCH --cpus-per-task 16
#SBATCH -p ics632.q

export OMP_NUM_THREADS=16

mpirun  ./matmul_outerproduct_v1_omp -n <matrix_size>
```

On each cluster, we ran our code in both MPI and MPI+OMP modes for different matrix sizes. Sizes are in the range (800:10,000) with the step size of 80. In each case, we have done 10 trials to get our results.

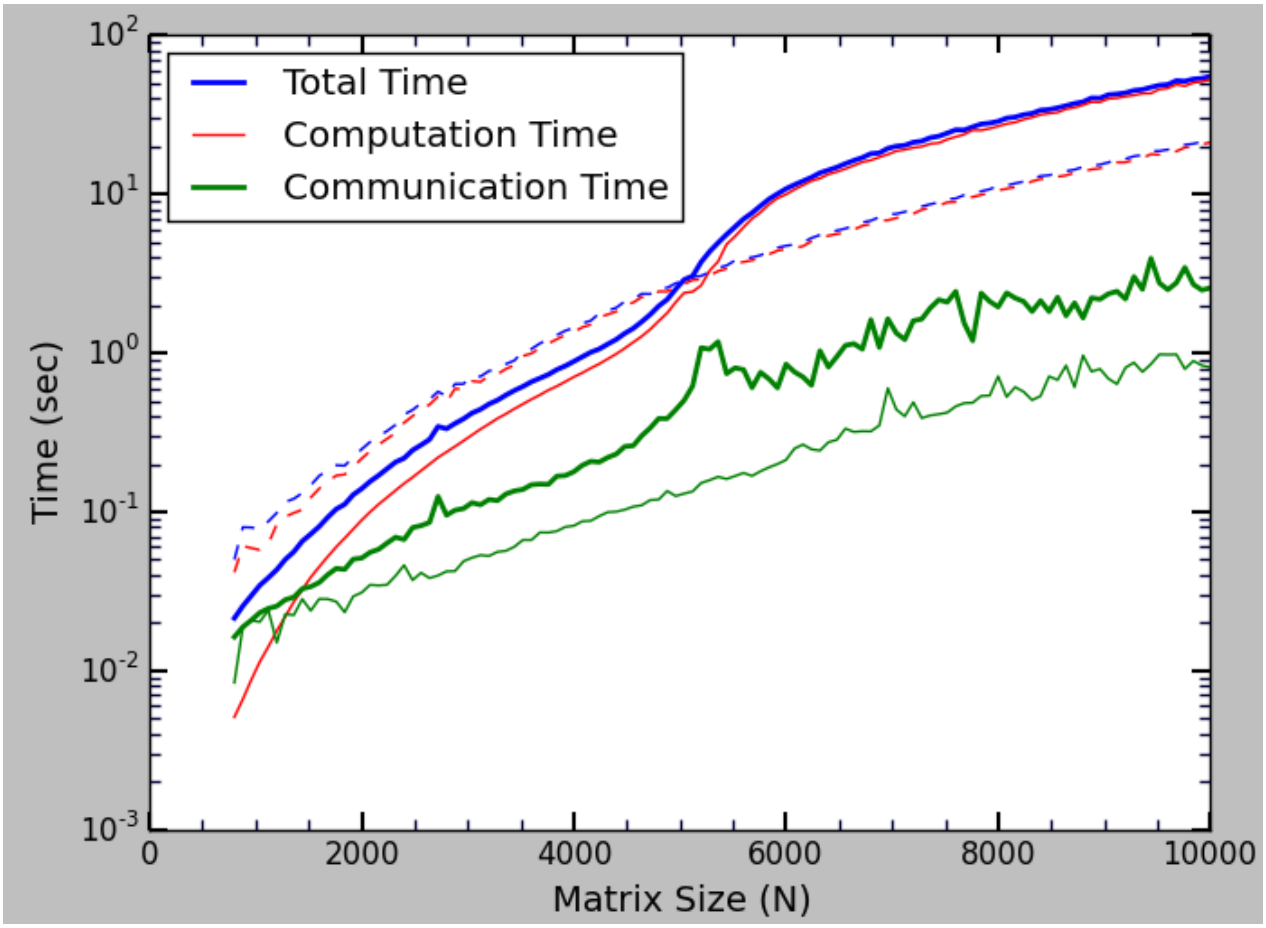

Image 6: Different components of the running time, for different matrix size. This is the results for the UH-hpc cluster. Bold solid lines represent the results for MPI-mode and dashed and thin lines represent MPI+OMP mode. - UH-cray cluster

As seen in Image 6, for UH cluster, the communication time is longer when running in MPI mode. As expected, for large matrix sizes, the communication time can be ignored since the total running time and the computation time are almost the same.

In general, MPI+OMP mode is more efficient compared to the MPI mode. The reason is not the difference between the communication time, since this time is fairly small, ~2 seconds. The computation time itself is much smaller when we use multi-threaded code. The reason might be the efficient usage of the memory in this case, because we are required to distribute different pieces of matrices over and over amongst those CPUs (on a same node) which have access to the same memory.

So the conclusion is that, when it is possible, use OMP over MPI which enables CPU to seethe same chunk of memory. Sending data is time-expensive and sharing the data is more efficient.

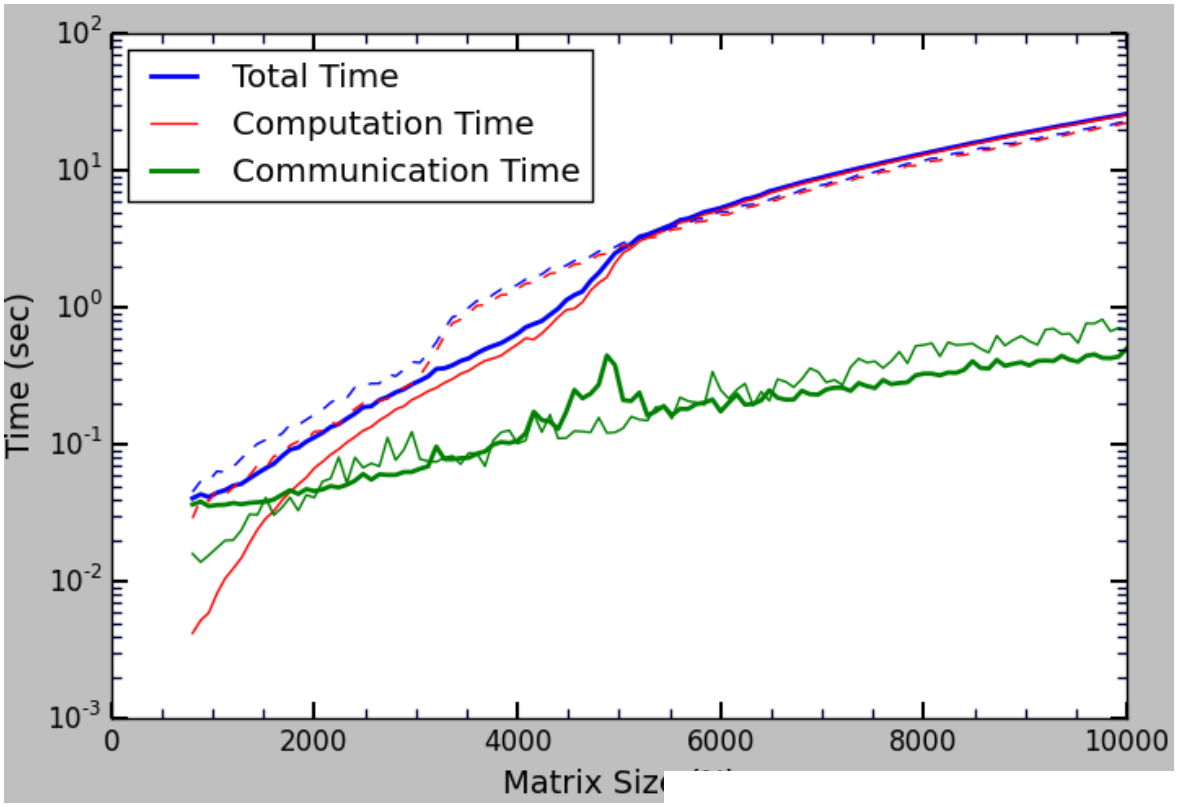

Image 7: Same as Image 6, for Mt. Moran cluster.

Images 7 and 8 show the same results for Mt. Moran cluster. It is again obvious that MPI+OMP is more efficient (see Image 8). Compared to Image 6, the communication time for MPI+OMP is shorter in this cluster. This can be due to different traffic of both cluster, and different network topologies.

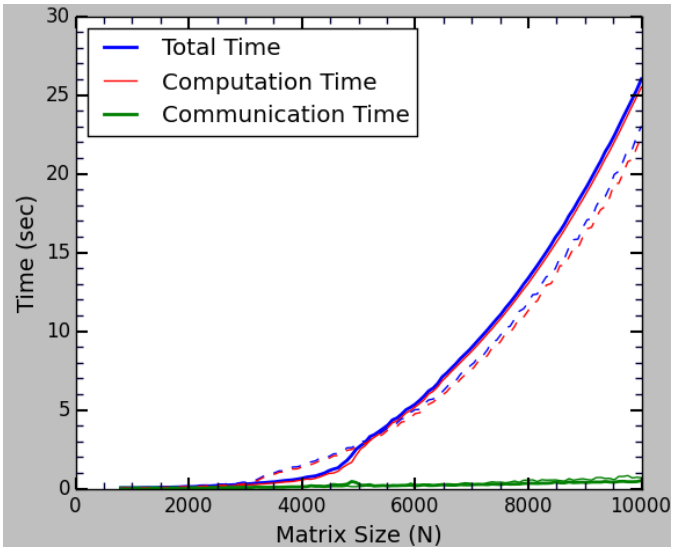

Image 8: Same as Image 7, in linear scale. - Mt. Moran

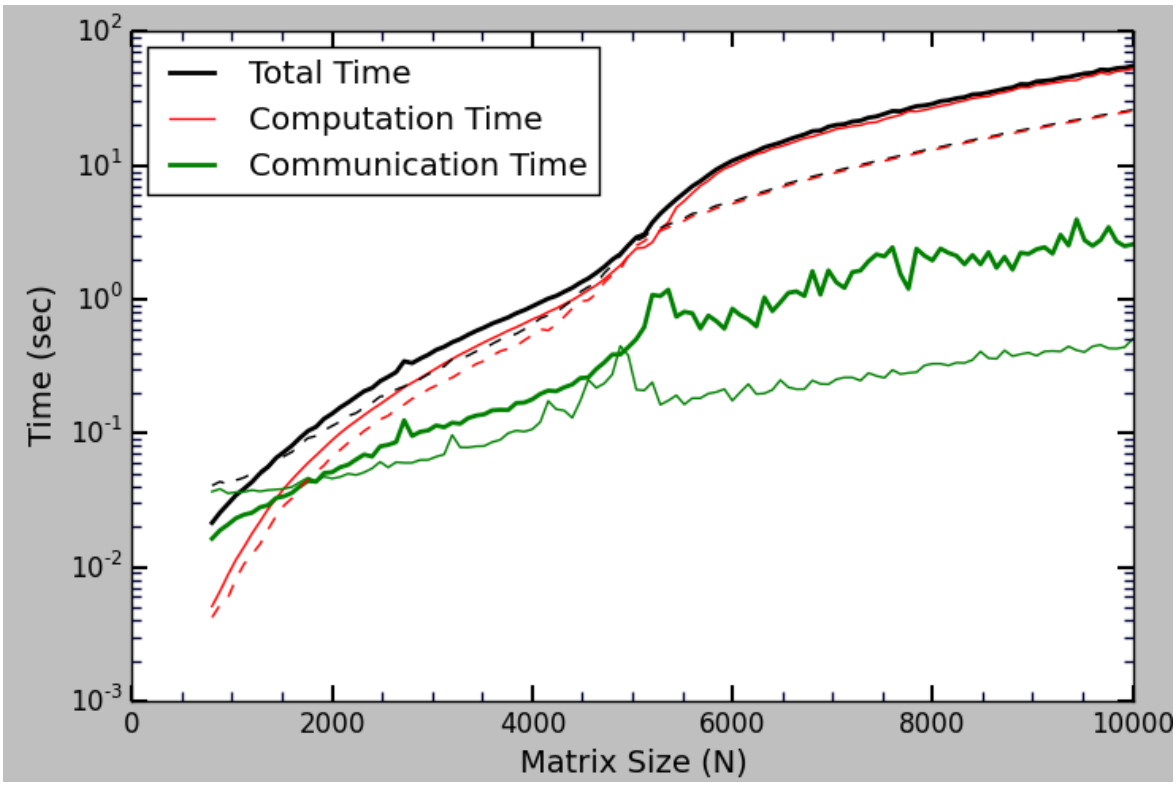

Image 9: Comparing UH-cray cluster with UW Mt. Moran cluster, when running in MPI mode. Bold solid lines represent the results for UH-cray and dashed and thin lines represent Mt. Moran clusters.

In image 9, we compare both UH-cray and Mt. Moran clusters, when running in MPI only mode.
Communication seems to be faster in Mt. Moran cluster, and overall efficiency seems to be better.

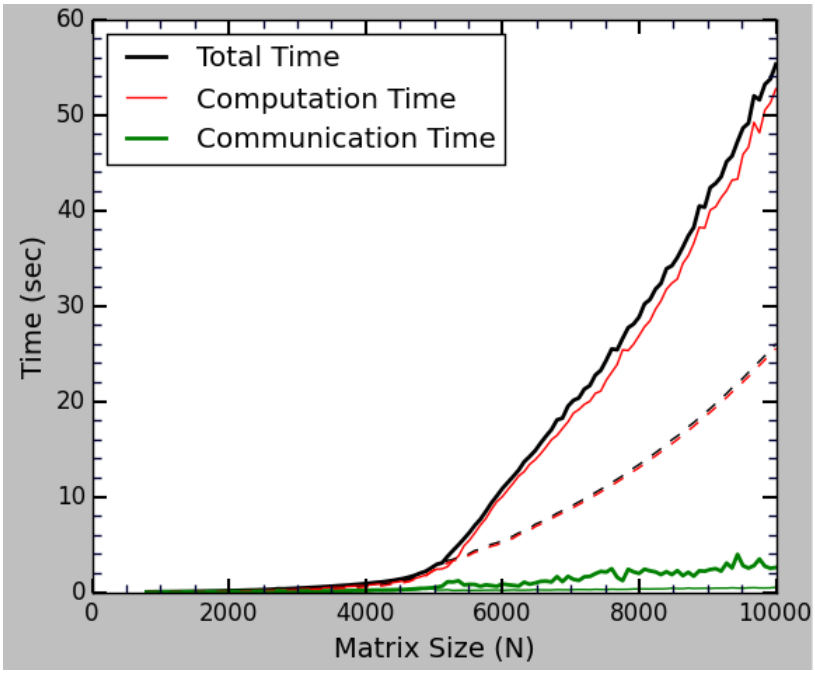

Image 10: Same as Image 9, in linear scale.

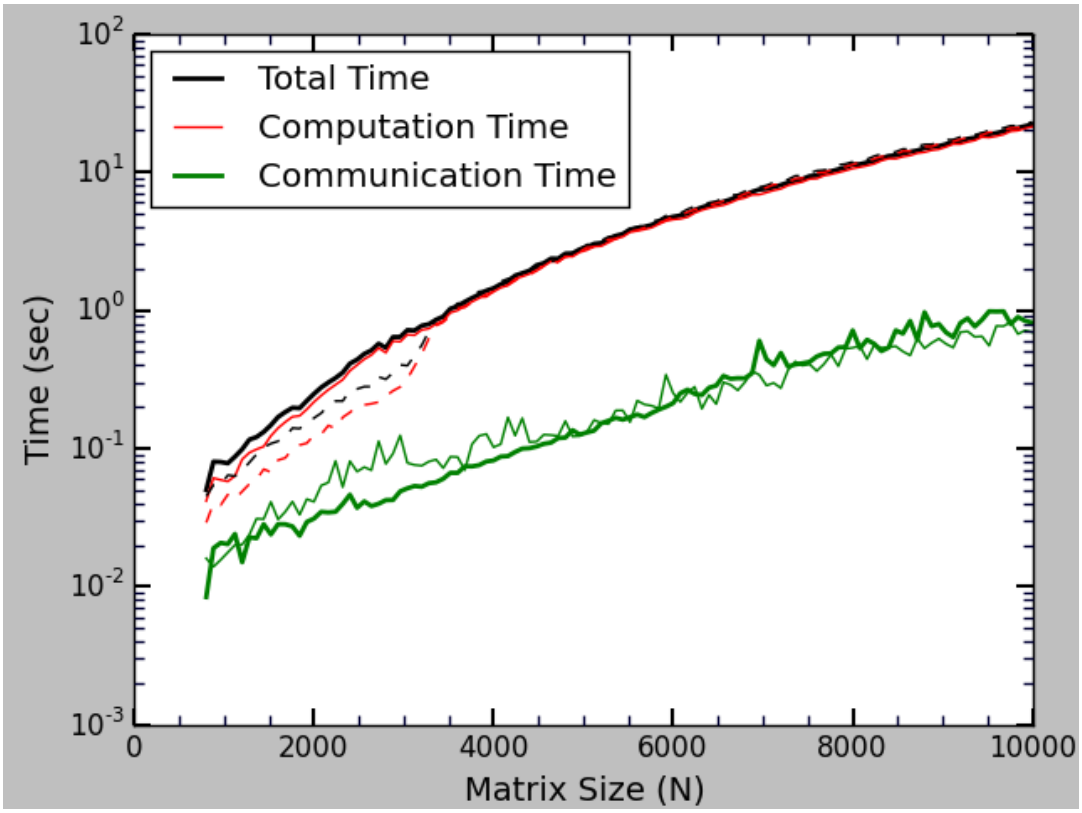

Image 11: Same as Image 9, when running in MPI+OMP mode.

In images 11 and 12, we compare the efficiency of both cluster when we run in MPI+OMP mode. Surprisingly, both cluster show the same efficiency and even UH-cray cluster performs slightly better.

The reason might be that, when we run in Multi-threaded mode, we less frequently halt the computations for communication and we miss less cycles when computing. This is better explained when we observe that the computational powers of each core on both clusters are relatively the same. Therefore, the computation time seems to be less affected by other factors.

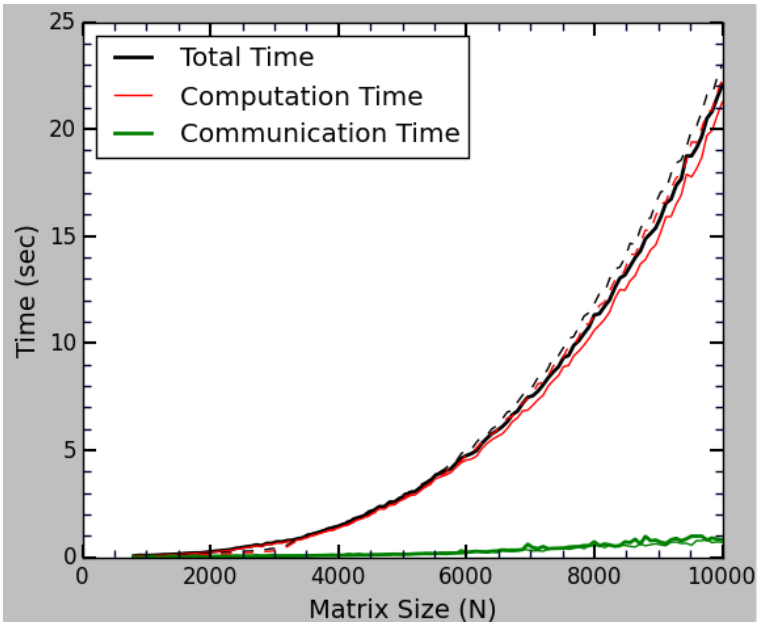

Image 12: Same as Image 11, in linear scale.

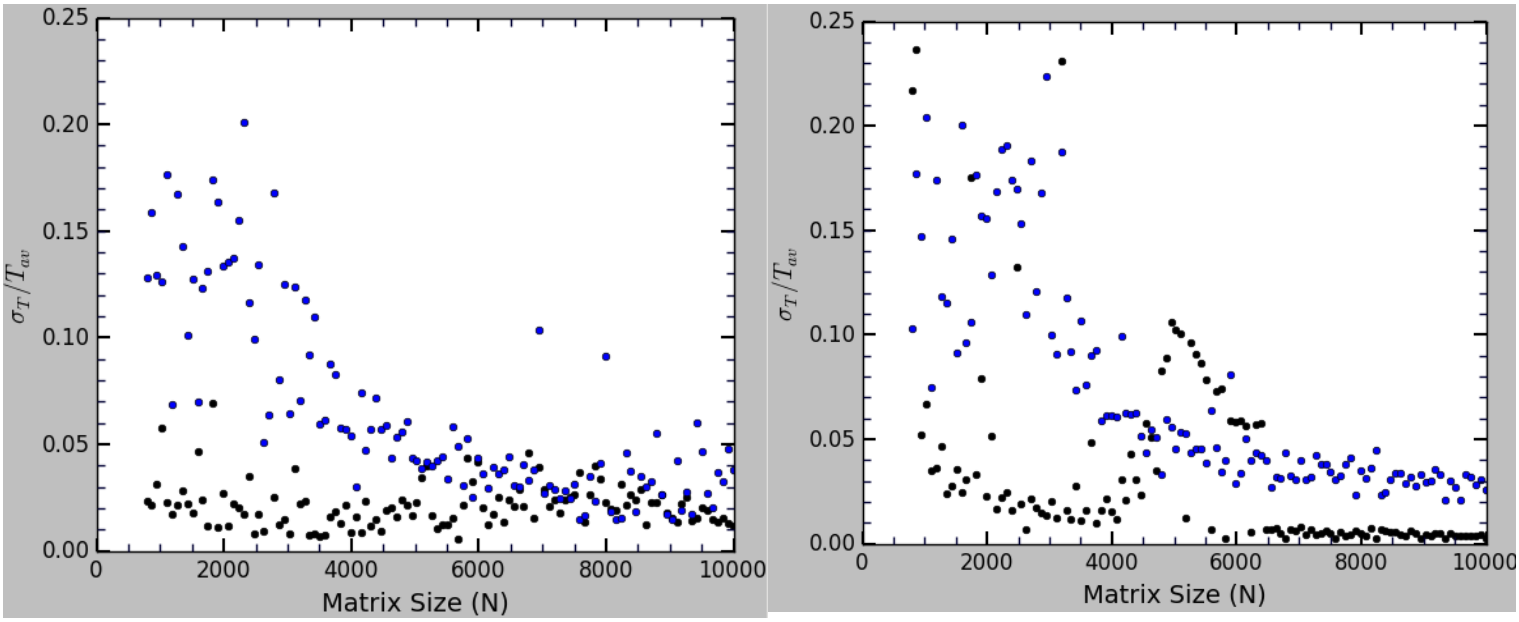

Image 13: relative fluctuation in running time, for different matrix sizes. Left: UH-cray cluster, Right: Mt. Moran cluster. Black: MPI, Blue: MPI+OMP

We compare how the running time fluctuates across trials for each matrix size. To do that, we calculate the standard deviation of the running time for each set of 10-trials and divide it to the average time. This ways we have an estimation of the relative run-time fluctuation that can be compared for different matrix sizes, different platforms and different modes (MPI or MPI+OMP). Image 13 shows our results for the total running time.

As seen, in MPI mode, fluctuations are smaller than MPI+OMP. This might be due to the fact that smaller size matrices are distributed among nodes when we only use MPI. In MPI mode, each matrix is divided by 64, but for MPI+OMP, it's divided by 4.

For MPI+OMP mode (blue dots), both clusters show the same trend and almost the same level of fluctuations. For MPI mode, Mt. Moran is more stable for larger matrix sizes, however it behaves statistically for mediocre sizes (N=4000—6000). These diagrams might be changed when running the code in different times, though.

The best way to make a strong conclusion is to running the same analysis at different times. Also another shortcoming of the current analysis is that we have run all the iterations is array mode, i.e. for each iteration different nodes might have been allocated, which changes the communication pattern the underlying network topology. The best way to perform this analysis is to target the same nodes for the entire analysis.