Table 1: This is an example.

Figure 1: Times for different sorts

Contents

1 Sorting on GPU

We measure the performance of our multiprocessing and MPI mergesort algorithms on the Carbonate GPU partition. Each node is equipped with two Intel 6248 2.5 GHz 20-core CPUs, four NVIDIA Tesla V100 PCIe 32 GB GPUs, one 1.92 TB solid-state that memory constraints allowed us to experiment with arrays of up to BOGO integer elements.

Multiprocessing merge sort was executed on 1 to 24 processes by using all available processes on the Carbonate partition. Message-passing merge sort was executed on 1, 2, 4, and 8 nodes by using one core on each node for MPI processes.

2 Performance Comparison

All performance comparison done here will be on data from the Carbonate partition.

2.1 Time

Using our results from the Carbonate partition, we analyze and graph the resulting times from selected runs.

We can analyze time in two ways. First, we analyze algorithm performance based on increasing size of arrays. Some algorithms differ within themselves (for example, two multiprocessing merge sorts might use different numbers of processes). However, we can account for this variation by aggregating all the values and showing error bands around the lines that are plotted.

The figure illustrates the average run time for arrays of up to size 10^7 for each different sort type (sequential merge sort, multiprocessing merge sort, and MPI merge sort.)

The general behavior displayed in the figure can be summarized in the following points:

1. For array sizes of up to around 2 million, the multiprocessing mergesort, on average, is quicker than the MPI mergesort. The overhead of MPI contributes significantly more to the overall sort time for smaller arrays, since the creation of processes takes longer than the creation of threads. However, the MPI mergesort is consistently the fastest sort type when arrays are sufficiently large enough that the time reduction from parallelism on individual nodes is enough to offset the additional costs of communication and thread creation.

2. The MPI sort displays the lowest average increase in time relative for array size, with a mean value of 1.708 seconds/million numbers. In other words, when the array size increases by an additional one million elements, the MPI sort time increases by an average of 1.708 seconds. The multiprocessing sort underperforms this significantly, with a mean increase of 2.706 seconds per additional million numbers. The sequential sort has a mean increase of 8.754 seconds per additional million numbers.

Second, we analyze algorithm performance based on processes used. We will keep the size of the array at a constant value and, for each number of processes, compare the times for each sorting algorithm.

![Table x: Time by process for size=10000, mp](https://github.com/cloudmesh/cloudmesh-mpi/blob/main/examples/sort/time-mp-10000.png)

![Table x: Time by process for size=10000, mpi](https://github.com/cloudmesh/cloudmesh-mpi/blob/main/examples/sort/time-mpi-10000.png)

![Fig 2: Time by process for size=10000](https://github.com/cloudmesh/cloudmesh-mpi/blob/main/examples/sort/image
by-p-sort-10000.png)width="50

For arrays of size 10000, increasing MPI parallelism is actually a positive factor for time. This aligns with what we see above, where MPI mergesort initally performs worse than multiprocessing mergesort on small arrays.

![Table x: Time by process for size=1e7, mp](https://github.com/cloudmesh/cloudmesh-mpi/blob/main/examples/sort/in
time-mp-1e7.png)

![Table x: Time by process for size=1e7, mp](https://github.com/cloudmesh/cloudmesh-mpi/blob/main/examples/sort/in
time-mpi-1e7.png)

![Fig 3: Time by process for size=10000000](https://github.com/cloudmesh/cloudmesh-mpi/blob/main/examples/sort/im
by-p-sort-10000000.png)

For arrays of size 10^7, we can see in figure 3 that increasing paralellism reduces runtimes for both algorithms.

2.2 Speedup

Speedup is defined as $\frac{T_s}{T_p}$, where T_s is the compute time of the sequential algorithm and T_p is the compute time of the parallel algorithm.

2.3 Efficiency

3 Conclusion

- MPI is advantageous to use when data is large enough - further speedup can be obtained by using MPI I/O operations.

3.1 Source Code

The source code is located in GitHub at the following location:

- <https://github.com/cloudmesh/cloudmesh-mpi/tree/main/examples/sort>

We distinguish the following important files:

- [night.py](https://github.com/cloudmesh/cloudmesh-mpi/blob/main/examples/sort/night.py)

3.2 Installation

3.2.1 ENV3 (for macOS)

bash $python - mvenv\ /ENV3$ source /ENV3/bin/activate

3.2.2 Clone

To download our code, please follow the instructions:

bash $mkdircm$ cd cm $gitclonegit@github.com : cloudmesh/cloudmesh - mpi.git$

3.2.3 Verification

Go to the [Github](https://github.com/cloudmesh/cloudmesh-mpi) to verify that the correct repository has been cloned.

3.2.4 Updating

1. Update local version

```
bash
  git pull
```

2. To add file (only do once)

```
git add filename
```

3. Once file is changed, do

```
git commit -m "this is my comment" filename
```

4. Remote upload

```
git push
```

3.3 Running the Program

Run the program using the command

```
./mpi_run.py --user={username} --node={node} --sort={mpi_mergesort}--size={size} --repeat={repeat} --id
```

Please read the Input sectionfor specific documentation on each option.

3.4 Overview

This project uses Python to implement an MPI mergesort algorithm (linked [here](https://github.com/cloudmesh/cloudmesh-mpi/blob/main/examples/sort/night.py)). The algorithm is then run and evaluated in [mpi$_r un.py$](*https* : *//github.com/cloudmesh/cloudmesh − mpi/blob/main/examples/sort/mpi$_r$un.py*).