MPI with Python

Gregor von Laszewski, laszewski@gmail.com
Fidel Leal
Jacques Fleischer

September 3, 2022

Contents

1 MPI Mergesort

This project implements an MPI (Message Passing Interface) mergesort algorithm.

Packages used: Numpy, Pandas, Seaborn, Matplot

1.1 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

1.2 Installation

1.2.1 ENV3 (for macOS)

```
$ python -m venv ~/ENV3
$ source ~/ENV3/bin/activate
```

1.2.2 Clone

To download our code, please follow the instructions:

```
$ mkdir cm
$ cd cm
$ git clone git@github.com:cloudmesh/cloudmesh-mpi.git
```

1.2.3 Verification

Go to the Github to verify that the correct repository has been cloned.

1.2.4 Updating

1. Update local version

   ```
   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
   ```

1.3 Running the Program

Run the program using the command

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

Please read the Input section for specific documentation on each option.

1.4 Overview

This project uses Python to implement an MPI mergesort algorithm (linked here). The algorithm is then run and evaluated in mpi_run.py.

1.4.1 Input

mpi_run.py has several inputs.

user: specify the username of whoever is running the program
node: name of computer node on which program is being run
sort: type of sort that is being run. Use "mpi_mergesort".
size: size of array to sort
repeat: number of times to run sorting algorithm. Final time will be the average of all runs.
id: specifies the merging algorithm that is used to merge the arrays once they've been sent from one process to another, also known as the merge algorithm. Each id number corresponds to a specific algorithm.
0: built-in Python *sorted*
1: sequential merge
2: adaptive merge

For example, if John would like to run the MPI mergesort on his Raspberry Pi 4, and he would like to use an array of size 200, repeating 10 times, and use an adaptive merge to combine the the arrays between processes, he would use the command

```
./mpi_run.py --user=john --node=pi4 --sort=mpi_mergesort --size=200 --repeat=10 --id=2
```

1.4.2 Mergesort

The unsorted array is generated on rank 0. Since the unsorted array must be split into smaller subarrays, *sub_size* is calculated. *sub_size* is the size of each subarray. It is equal to the size of the unsorted array divided by the number of processors, since each processor must be sent a subarry. Note that *n*, the size of the unsorted array, must be evenly divisible by the total number of processors.

Once the subarrays have been distributed using the Scatter command, they are sorted on each processor using the built-in Python. This sort defaults to Timsort.

In order to merge the sorted subarrays, we can visualize the processors as being set up in a binary tree, where each parent has two children. One important note: in this situation, the left child also functions as the parent node. We can then split the tree into two sections: left children and right children. Because we are using a binary tree, we know that the number of left and right children will always be equal. Therefore, we can create a variable *split* that splits the set of processors in half.

If the rank of the processor is in the second half (between *split* and *split* * 2), then it will send its sorted subarray to its “left” partner to be merged. Otherwise, if the rank of the processor is in the first half (between 0 and *split*), it will recieve a sorted subarray from its “right” partner and then merge it with the subarray that it currently contains. When a subarray is sent, it is sent to the processor with rank *rank - split*, ensuring that the processor that it is sent to has a rank between 0 and *split*. This guarantees that each subarray that is sent gets sent to a merging processor. Similarly, when a subarray is received, it is received from a processor with rank *rank + split*, ensuring that the subarray is a sorted array to be merged. This mapping guarantees a unique pairing between left and right child.

Once received, the subarrays are merged together. The merging algorithm can be defined by the user, and will be referred to as the **merge algorithm**. There are currently three merging algorithms that can be used. 1. **sequential_merge_fast**, a “fast” merge that simply combines the two arrays using the built-in Python *sorted* function. Note that sorting algorithms can also be used to merge. 2. **sequential_merge_python**, a merge that uses the well-known technique of appending the smaller of two array elements to a third array 3. **adaptive_merge**, a custom sort algorithm that takes advantage of pre-existing order in the sequence. It is described in more detail here

Then, each individual send/recieve operates as following:

1. Left child sends sorted subarray
2. Right child allocates memory for recieving subarray from left child (*local_tmp*)
3. Right child allocates memory for array to store merged result (*local_result*)
4. Right child receives subarray

5. Right child merges received subarray with its own subarray
6. Right child assigns *local_arr* to point to *local_result*

This loop continues until the tree reaches the height that guarantees us a single sorted list. Each time, the number of nodes is halved (since two have been merged into one).

1.4.3 Output

The output of the program is generated and logged in mpi_experiment.py. This file runs the sort according to user specifications. Three of the most important things it does: 1. The specific algorithm is run *repeat* times. Note that the larger *repeat* is, the more accurate the final mean time will be. The mean time is not calculated here. Rather, all times from each repeat will be outputted. 2. Maps id numbers to merge types. It's important to be able to differentiate between data outputted by each merge type, since we'd like to examine the differences. However, we'd also like to keep the titles/inputs concise, which is why id numbers are used. Each id number is translated to its merge type in the program. 3. Times the runs. Running the sort is done by a bash command, so by starting a timer, running the command, and stopping the timer, we can get the full amount of time it took to run the sort. This is a more accurate measure than placing timers inside of the actual sort program (night.py), since it is able to time the MPI initialization, which timers inside the program wouldn't be able to do.

1.4.4 Final Run Method

This is the outermost run program, i.e. the one that a user will run. See the above Input section for an example of how to run this. mpi_run.py generates a log file and runs mpi_experiment.py.

1.5 adaptive

This sort takes advantage of pre-existing sorted subsequences in two arrays to merge them together. It is designed to merge two sorted arrays together. Given two arrays, called *left* and *right*, we first calculate the min and max values in both.

```
    l_min = left[0]
    r_min = right[0]
    l_max = left[len(left) - 1]
    r_max = right[len(right) - 1]
```

We can also assume that the minimum of left will always be smaller than the minimum of right (if not, we can just swap them).

The sort splits into three scenarios based on *l_min*, *r_min*, *l_max*, and *r_max*:

1.5.1 No overlap

This would be if *l_max* <= *r_min*. In this case, a visual representation of the arrays could look like this.

```
left
------------
                  ----------
                     right
```

where the arrays are entirely disjoint. Then, we can just concatenate the two and return.

1.5.2 Some overlap

This would be if *l_max* were between *r_min* and *r_max*, or *l_max* > *r_min* and *l_max* < *r_max*.

A visual representation:

```
left
--------------
         -----------------
                     right
```

Then, we want to split both arrays into two parts:

```
left
---------|-----
          -----|------------
                     right
```

If we take the divider in *left* and call it *l_idx*, we can see that everything to the left of *l_idx* is completely separate from *right* with no overlap. Similarly, everything to the right of the divider in *right* has no overlap with any of *left*.

Then, how do we define these dividers? Note that the visual is slightly misleading: the arrays are not continuous spans of numbers, rather, they are individual scattered points that span from a min to a max. Then, if we want to find *l_idx* such that everything to the left of *l_idx* doesn't overlap with *right*, we want to find the index of the smallest element in *left* that is greater than *r_min*. Since every element to the left would then be less than or equal to *r_min*, that point would then be the *l_idx* we are looking for.

Similarly, we want to search in *right* for some *r_idx* where everything to the right of *r_idx* has no overlap with *left*. Then, we want to find the index of the largest element in *right* that is less than *l_max*. Since every element to the right would then be greater or equal to *l_max*, that point would be *r_idx*.

From here, binary search is employed to find such points. Read the Binary search section for more information.

Once *l_idx* and *r_idx* are found, we can then separate out the disjoint parts.

```
    left_sorted = left[:l_idx]
    right_sorted = right[r_idx + 1:]
```

The remaining parts can then be concatenated and sorted. This sort uses the Python builtin sort.

```
    unsorted = np.concatenate((left[l_idx:], right[:r_idx+1]))
    unsorted = np.array(sorted(unsorted))
```

Finally, the left sorted portion, the now-sorted middle portion, and the right sorted portion are concatenated and returned.

1.5.3 Complete overlap

This would be if *l_max >= r_max*.

A visual representation:

```
left
-------------------
      ------
      right
```

Then, we want to split *left* into three parts:

```
left
------|------|-------
       ------
      right
```

Let the leftmost divider in *left* be *left_sorted_min*, and the rightmost divider in *left* be *left_sorted_max.* Then, everything to the left of *left_sorted_min* has no overlap with *right*, and everything to the right of *left_sorted_max* has no overlap with *right*.

To find *left_sorted_min*, we want to find the smallest element in *left* that is larger than *r_min*. To find *left_sorted_max*, we want to find the largest element in *left* that is smaller that *r_max*. We can do this using binary search - the specific algorithm is described in the Binary search section.

We can isolate the sorted, separate ends with

```
left_sorted_min = left[:l_idx_min]
left_sorted_max = left[l_idx_max + 1:]
```

Then, we can merge the middle of *left* and all of *right* together and sort the resulting array using the builtin Python sorted.

```
unsorted = np.concatenate((right, left[l_idx_min:l_idx_max + 1]))
unsorted = np.array(sorted(unsorted))
```

Finally, the left end, the sorted middle section, and the right end can be concatenated together and returned.

1.5.4 Binary search

This binary search takes in three parameters: 1. *arr*: the array in which to search 2. *val*: the value with which to compare, and 3. *ineq*: the comparision with which to compare the result to val

Expressed in words: If *ineq* = "<", we are searching for the largest value in *arr* that is less than *val*.

If *ineq* = ">", we are searching for the smallest value in *arr* that is greater than *val*.

The actual algorithm is classic. Calculate a middle value and perform a boolean-evaluating statement. If true, take the lower half of the array. Otherwise, take the upper half of the array. However, note that if the array is increasing, this will only work if *ineq* = ">", since true will always lead to the upper half of the array.

Instead of writing two different binary searches, we make two crucial changes to the function: 1. **Using eval statements.** Instead of hardcoding a < or a > in the if-statement, we can instead construct a statement to check.

```python
to_check = str(arr[mid]) + ineq + str(val)
eval(to_check)
```

Doing this allows us to have both "<" and ">" statements, depending on which one is inputted.

2. **Reversing the array.** If the *ineq* = "<" and the array is increasing, this function will never work, since we always take the upper half of the array when the check statement is true. However, if the array is decreasing, then taking the upper half works: those values are the ones that are less than *val*, and the ones we want to continue to search in. At the end, we can reverse the array, and flip the answer to the reversed index, i.e. 2 would become the 2nd index from the end.

```python
arr = np.flip(arr)
return len(arr) - l - 1
```

1.6 Analyzing the Results

Analysis of the data takes place in mpi.ipynb.

The function *get_data* extracts data from the logfiles. It specifically gets the number of processes, the time, the size of the array, the id, and name, and the tag from each line of output in each log file. This data is then formatted using Pandas DataFrames.

We can then plot the data using *plot_benchmark_by_size*. This function generates the name, title, and graphs the given data by whichever specific attribute the user would like.

name

The above figure shows the relationship between increasing size of array and time of sorting for three different merging algorithms. Recall that 0 is the Python builtin sort, 1 is the iterative merge, and 2 is the adaptive sort. The sizes of the arrays used are [100, 1000, 10000, 100000, 1000000, 5000000, 7000000, 10000000]. We can see that for the first half of sizes, all three sorts appear to perform roughly the same. However, once we pass that point, the builtin Python sort performs much better than the other two. We can also see that while the adaptive merge always performs better than the iterative merge, it is still slower than the builtin Python sort. Lastly, we can observe the very low variability of the Python builtin sort, even as array size increases. Both the other two sorts have a great deal of variation, but the builtin sort stays relatively precise.