I Came, I Saw, I Divided, I Conquered: On the Parallelization of Merge Sort in Python

Sandra Yang, Gregor von Laszewski

Contents

I Came, I Saw, I Divided, I Conquered: On the Parallelization of Merge Sort in Python

Sandra Yang, Gregor von Laszewski

November 20, 2022

Abstract

The theory of divide-and-conquer parallelization has been well-studied; however, less is known about the implementation and performance of these algorithms in Python. Python's simplicity and extensive selection of libraries make it the most popular scientific programming language in the world today, and the recent shift in high performance computing to highly parallel systems makes understanding divide-and-conquer parallelization in Python uniquely important. Merge sort is a well-known sorting algorithm that is a characteristic example of divide-and-conquer; as such, conclusions can be drawn from merge sort parallelization to improve understanding of divide-and-conquer parallelization. In this paper, we use Python packages multiprocessing and mpi4py to implement several different parallel merge sort algorithms. Experiments are conducted on Indiana University's large-memory computer cluster, and benchmarking and performance analysis is done using Cloudmesh. We find that hybrid multiprocessing merge sort outperforms all other algorithms, achieving a 1.5x speedup compared to the built-in Python sorted() and a 34x speedup compared to sequential merge sort. Our results provide insight into different approaches of implementing parallel merge sort in Python and contribute to the understanding of general divide-and-conquer parallelization in Python on both shared and distributed memory systems.

1 Introduction

Divide-and-conquer algorithms recursively break problems down into smaller subproblems until those become simple enough to solve directly, at which point the recursion terminates. The results are then combined to generate a solution to the original problem. Divide-and-conquer algorithms are widely used for many different purposes, including multiplying very large numbers, finding the closest pair of points, and computing the discrete Fourier transform.

Divide-and-conquer algorithms are uniquely suited to parallelization. Parallelism is usually inherent, as once the division phase is complete, the subproblems are usually independent from one another and can be solved separately. Furthermore, phases often mirror each other, with the combination phase of one subproblem occurring identically, but independently, to the combination phase of another subproblem.

Merge sort is the classic divide-and-conquer algorithm, making it an excellent experimental base upon which to try different approaches to parallelism. In this paper, we explore the parallelization of merge sort using several different Python packages. To parallelize merge sort on Symmetric Multi-Processors (SMPs), we use multiprocessing, and to parallelize it on clustered systems, we use message-passing through the mpi4py module.

We begin by reviewing several single-node sorting algorithms in Section 5. Section 5.3 explores a shared memory multiprocessing implementation of merge sort. We evaluate and analyze its performance in comparison with sequential merge sort (Section 5.1) and the built-in Python sort (Section 5.2) in Section 6.

Next, we describe two merge sorts that are implemented using MPI. Section 7.1 focuses on both message-passing merge sort using mpi4py and a hybrid MPI merge sort that combines both multiprocessing and MPI. These algorithms are evaluated and compared in Section 8. Finally, we offer our conclusions in Section 9.

2 Related Research

The theory of merge sort parallelization has been studied in the past. Cole [1] presents a parallel implementation of the merge sort algorithm with $O(logn)$ time complexity on a CREW PRAM, a shared memory abstract machine which does not provide synchronization and communication, but provides any number of processors. Furthermore, Jeon and Kim [2] explore a load-balanced merge sort that evenly distributes data to all processors in each stage. They achieve a speedup of 9.6 compared to a sequential merge sort on a Cray T3E with their algorithm.

On MPI, Randenski [3] describes three parallel merge sorts implemented in C/C++: shared memory merge sort with OpenMP, message passing merge sort with MPI, and a hybrid merge sort that uses both OpenMP and MPI. They conclude that the shared memory merge sort runs faster than the message-passsing merge sort. The hybrid merge sort, while slower than the shared memory merge sort, is faster than message-passing merge sort. However, they also mention that these relations may not hold for very large arrays that significantly exceed RAM capacity. While this paper provides valuable information on different approaches to parallelization in general, it does not provide Python-specific insight.

3 Installation

The source code is located in GitHub [4]. To install and run it, along with the necessary packages, run the following commands:

```
$ python -m venv ~/ENV3
$ source ~/ENV3/bin/activate
$ mkdir cm
$ cd cm
$ pip install pip -U
$ pip install cloudmesh-installer
$ cloudmesh-installer get mpi
$ cd cloudmesh-mpi
$ pip install mpi4py
$ pip install multiprocessing
```

4 Benchmarking

The code uses the StopWatch from the Cloudmesh library, which easily allows code to be augmented with start and stop timer methods, along with the use of a convenient summary report in the format of a CSV table. [5].

5 Single Machine Sorting

5.1 Sequential Merge Sort

Sequential merge sort is a standard example of a sequential divide-and-conquer algorithm. The idea of merge sort is to divide an unsorted list into smaller, sorted lists, and then merge them back together to produce the sorted list in its entirety (see Figure 1.

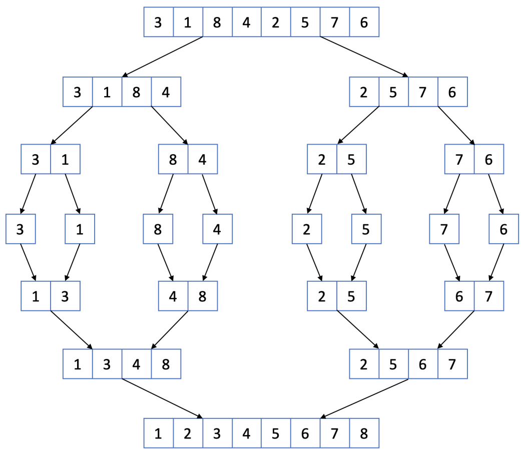

Figure 1: Sequential merge sort example

The steps are as follows:

1. If the input array has less than two elements, return.
2. Split the input array into two halves, left and right.
3. Sort the two halves using the same algorithm.
4. Merge the two sorted halves to form the output array.

The code is presented in Figure 2

A variant on this approach is to stop splitting the array once the size of the subarrays gets small enough. Once the subarrays get small enough, it may become more efficient to use other methods of sorting them, like the built-in Python **sorted**(a) function instead of recursing all the way down to size 1. See Figure 3.

The average time complexity of classic merge sort is $O(n\log(n))$, which is the same as quick sort and heap sort. Additionally, the best and worst case time complexity of merge sort is also $O(n\log(n))$, which is the also same as quicksort and heap sort. As a result,

```
def mergesort(array):
  # check to make sure array has two or
  # more elements
  if len(array) > 1:
    # find midpoint of array
    mid = len(array) // 2

    # split array into two halves
    left = array[:mid]
    right = array[:mid]

    # sort the two halves
    left = mergesort(left)
    right = mergesort(right)

    # merge the two halves
    array = merge(left, right)
  return array
```

Figure 2: Sequential merge sort algorithm.

```
def sequential_mergesort(array):
  n = len(arr)
    if n < SMALLEST_ARRAY_SIZE:
      array = sorted(array)
      return array

    else:
      left = array[:mid]
      right = array[:mid]

      # sort the two halves
      left = sequential_mergesort(left)
      right = sequential_mergesort(right)

      # merge the two halves
      array = merge(left, right)

      return array
```

Figure 3: Hybrid sequential merge sort algorithm.

classical merge sort is generally unaffected by factors in the initial array.

However, classical merge sort uses $O(n)$ space, since additional memory is required when merging. Quicksort also has this space complexity, while heap sort takes $O(1)$ space, since it is an in-place method with no other memory requirements.

A summary of the complexities is shown in Table 1.

5.2 Python Built-in Sort

For the sake of benchmarking our merge sorts, we also run and compare the times of the Python built-in sort. Python uses an algorithm called Timsort, a hybrid sorting algorithm of merge sort and insertion sort. The algorithm finds "runs", or subsequences of the data that are already order, and when these runs fulfill specific merge criteria, they are merged.

The Python builtin sort is run by calling **sorted**(a), where a is the list to be sorted.

5.3 Multiprocessing Merge Sort

Within a Symmetric Multiprocessing Machine (SMP), two or more identical processors are connected to a single mainstream memory, with full accessibility to all the input/output devices. In the case of multiple core machines, the SMP architecture treats each core as a separate processor.

To parallelize Python code on SMP machines, we use multiprocessing, which is a package that supports programming using multiple cores on a given machine. Python's Global Interpreter Lock (GIL) only allows one thread to have exclusive access to the interpreter resources, which means multithreading cannot be used when the Python interpreter is required. However, the multiprocessing package sidesteps this issue by spawning multiple processes instead of different threads. Because each process has its own interpreter with a separate GIL that carries out its given instructions, multiple processes can be run in parallel on one multicore machine.

A common way to parallelize Python code is to employ a pool of processes. multiprocessing provides this with the Pool class, which controls a pool of worker processes to which jobs can be submitted. There are four main steps in the usage of the Pool class.

1. Create the pool using
   pool = multiprocessing.Pool(processes=n)

   Initializes a Pool object pool with n worker processes.

2. Submit tasks.

   (a) Synchronous

   (b) Asynchronous

3. Wait for tasks to complete (dependent upon type of task submission).

4. Shut down the pool.
   pool.close()

5.3.1 Synchronous Tasks

The **map** method in Python maps a function to be applied to every item in the provided iterable. Figure 4 shows two different ways to map the example function to each of the chunks in input and append the returned result to results.

```
# example function
def function(input):
   ...
   return result

# input, divided into chunks
input = [chunk1, chunk2, ...]

results = []
for chunk in input:
   results.append(function(chunk))
```

(a) Mapping function to arrays using loop.

```
# example function
def function(input):
   ...
   return result

# input, divided into chunks
input = [chunk1, chunk2, ...]

results = map(function, input)
```

(b) Using Python map method.

Figure 4: Sequential **map** examples.

The Pool provides a parallel version of the sequential **map** function. Note that **map** blocks, meaning that all worker processes must finish before the program can move on.

```
# example function
def function(input):
   ...
   return result

# input, divided into chunks
input = [chunk1, chunk2, ...]

# create pool with n processes
pool = multiprocessing.Pool(processes=n)

# start worker processes in parallel
# output from each function call
# is appended to results
results = pool.map(function, input)
```

Figure 5: Parallel function of map using Pool.

5.3.2 Asynchronous Tasks

The Pool also provides an asynchronous version of the sequential **map** function called map_async(). It does not block and returns a AsyncResult to access later.

```
...
# start worker processes in parallel
# issue tasks to the worker processes asynchronously
results = pool.map_async(function, input)

# iterate over the results from the tasks
for value in results.get():
   ...
```

Figure 6: Parallel function of map_async() using Pool.

While parallelizing a program reduces the computational time by adding workers and reducing the size of the problem for each process, it also adds unavoidable overhead. Costs include the initialization time, overhead due to external libraries, and the cost of communication between workers.

The official documentation for multiprocessing is located within the Python Standard Library [6].

5.3.3 Overview

A relatively straightforward conversion of sequential merge sort to multiprocessing merge sort can be done. Note that for this implementation, the array to be sorted and the number of processes to be used are passed in as parameters.

To initialize the multiprocessing merge sort, we begin by initializing the Pool and dividing up the array.

```
def multiprocessing_mergesort(arr, processes):
  # create processor pool
  pool = multiprocessing.Pool(
         processes=processes)

  # split array into equally sized chunks
  # one for each process
  # append chunks to arr1
  size = int(math.ceil(
         float(len(arr)) / processes))
  arr1 = []
  for i in range(processes):
    arr1.append(arr[(size*i):(size*(i+1))])
```

Figure 7: Initialization of multiprocessing merge sort.

Next, we spawn the worker processes, which run fast_sort on each chunk in parallel. fast_sort is the builtin Python **sorted**() function. Note that we use synchronous communication here by using **map**().

```
arr1 = pool.map(fast_sort, arr1)
pool.close()
```

Figure 8: Parallel mapping from sort to array chunk.

Finally, we merge the sorted chunks into one whole array.

6 Analysis of Single Machine Sorting

The performance of all sorts was measured on Carbonate, Indiana University's large-memory computer cluster. Sorts were ran on the general-purpose compute nodes, each of which is a Lenovo NeXtScale nx360 M5 server equipped with two 12-core Intel Xeon E5-2680 v3 CPUs. Each node has 256 GB of RAM and runs under Red Hat Enterprise 7.x. The sequential merge sort and builtin sort were run on a single core, while the multiprocessing merge sort was run on varying numbers of cores, from 1 to 24. All sorts were run on arrays of varying sizes of up to 10^7.

c	size	sort	time	speedup	efficiency
1	10^7	mp	7.724	1.000	1.000
4	10^7	mp	3.474	2.223	0.556
8	10^7	mp	3.164	2.441	0.305
12	10^7	mp	2.487	3.106	0.259
16	10^7	mp	2.820	2.739	0.171
20	10^7	mp	2.858	2.703	0.135
24	10^7	mp	2.830	2.730	0.114
1	10^7	seq	85.611	—	—
1	10^7	sorted	3.860	—	—

Figure 9: Performance results on one node from Carbonate partition (all times in seconds, mp = multiprocessing, seq = sequential merge sort).

Figure 9 displays the performance results of the three single-machine sorting algorithms: multiprocessing merge sort, sequential merge sort, and the built-in Python **sorted**(). The multiprocessing merge sort is much slower than the built-in sort at the beginning when running on one core. However, once two cores are reached, the multiprocessing merge sort becomes the fastest sort, and continues to get faster as the number of cores increases. The multiprocessing merge sort achieves a 34x speedup and the built-in Python sort achieves a 22x speedup in comparison to the sequential merge sort. The multiprocessing merge sort typically outperforms the built-in Python sort, achieving a 1.5x speedup when measured using 12 cores.

For large arrays, increased parallelism is a negative factor for time. Figure 10 displays the relationship between the number of cores used and the time of the multiprocessing merge sort algorithm for arrays of size 10^7.

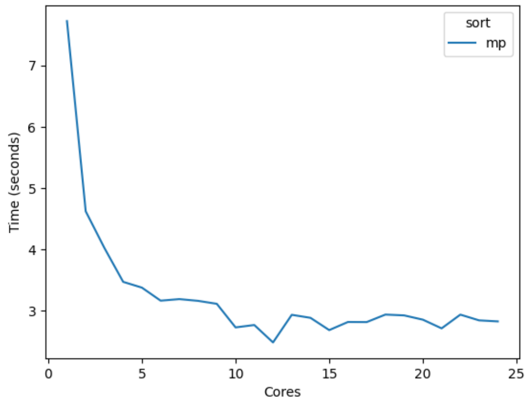

Figure 10: Arrays of size 10^7.

However, this relationship does not hold true for smaller arrays. Figure 11 displays the cores and times of the multiprocessing merge sort algorithm for arrays of size 10^4. In this case, increased parallelism is actually a positive factor for time. Since the array is small, the ratio of the code that is parallelized to the code that must run sequentially is also small. This means that the reductions in speed parallelization are not large enough to offset the parallelization overhead. While costs like initialization and overhead from external libraries remain constant, the cost of communication between processes increases significantly with the number of processes.

In terms of speedup, gains are made extremely quickly when (a) the number of cores being used is small, and (b) the size of the array is small. As the experiments increase from using 1 core to using 4 cores, we achieve substantial increases in speedup, as can be seen in Figure 12. Furthermore, speedup, which initially is less than one for small arrays, increases as

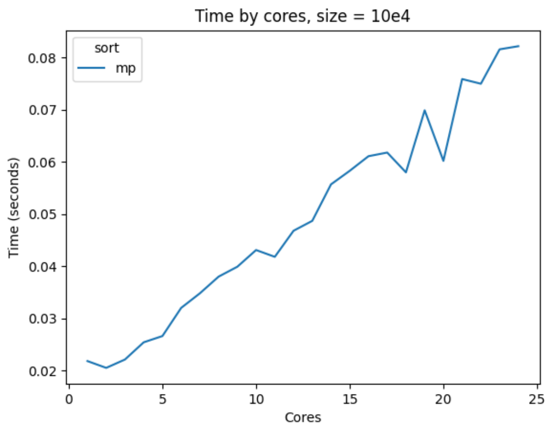

Figure 11: Arrays of size 10^4.

the array sizes increase from 100 to around 2×10^6. However, we can see that as the array sizes continue to increase past that point, speedup begins to level off, increasing at a much slower rate.

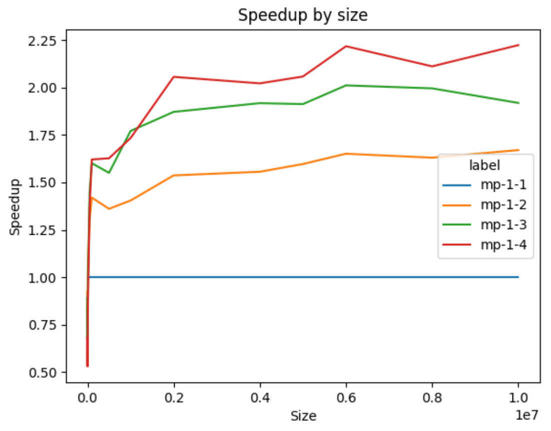

Figure 12: Speedup by size for 1, 2, 3, and 4 cores (mp-1-*c* means multiprocessing merge sort on *c* cores.)

For larger numbers of cores, speedup also begins to level off, and differences in performance between the numbers of cores used become negligible. In Figure 13, multiprocessing merge sort on 6, 18, and 24 cores all display very similar speedup, despite the large difference in the number of cores.

Multiprocessing merge sort on 12 cores outperforms them all, but this can be attributed to system architecture. Since each node on the Carbonate has two 12-core Intel Xeon CPUs, 12 is the optimal number of cores on which to run multiprocessing merge sort. Any more cores requires cross-CPU communication between the two CPUs, a much higher cost than the intra-CPU communication done by 12 or less cores.

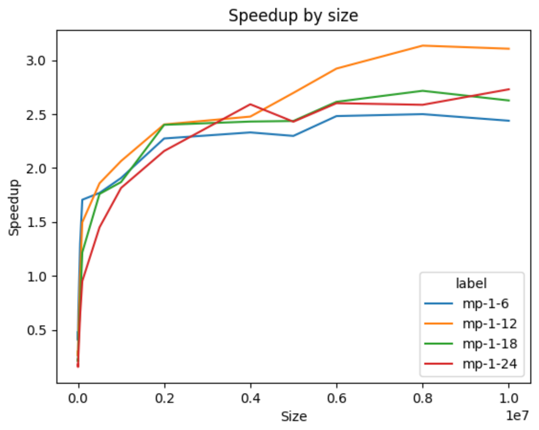

Figure 13: Speedup by size for 6, 12, 18, and 24 cores (mp-1-*c* means multiprocessing merge sort on *c* cores.)

7 Multi Processor Sorting

7.1 MPI Merge Sort

The limitation of multiprocessing is that it does not support parallelism over distributed computing systems. To enable the communication of computing nodes, we use MPI for Python, also known as mpi4py. An object oriented approach to message passing in Python, it closely follows the MPI-2 C++ bindings. Within a distributed computing system, each node has its own local memory, and information can only be exchanged by passing messages through available communication links. The mpi4py module provides both point-to-point and collective communication capabilities for both Python objects and buffer-like objects.

Some usages of mpi4py are introduced: By default, there is a single group that contains all CPUs. MPI.COMM_WORLD is the intra-communicator that handles cross-node communication. comm.Get_size() is used to obtain the number of CPUs available in the group to perform the computation, and comm.Get_rank() is used to obtain the rank of the current CPU. The rank of a CPU is its unique numerical identifier. CPUs can have rank 0 through size - 1, the assignment of which is entirely random. Typically, the root node will be designated as whichever CPU has rank 0, and ranks 1 through size - 1 will be the secondary node.

Point-to-point communication is carried out using comm.send() and comm.recv(). As described in the name, comm.send is used to send data from one CPU, and comm.recv is used to receive data on the recipient CPU. Each envelope contains:

1. The data that is to be sent
2. The datatype of each element of the data
3. The size of the data
4. An identification number for the message
5. The ranks of the sending and receiving nodes

Senders and receivers in a program should be matched, and there should be one receive per send. The sending process exits once the send buffer can be read and written to, and the receiving process exits once it has successfully received the message in the receive buffer. Since completion of communication depends on both processes in the pair, there is a risk that if one fails, the program will be stuck forever.

Collective communication transmits data to every CPU in the group. Important functions for collective communication in mpi4py are:

1. Broadcasting:
   comm.Bcast(data, root=0)

   The root node sends data to all other CPUs in the group.

2. Scattering:
   comm.Scatter(sendbuf, recvbuf, root=0)

   The root node sends equal chunks of the data (sendbuf) to all other CPUs in the group.

3. Gathering:
   comm.Gather(sendbuf, recvbuf, root=0)

   All of the data from all CPUs in the group is collected to the root node.

Note that with mpi4py, the communication of generic Python objects utilizes all-lowercase names like comm.send(), comm.recv(), and comm.bcast(), while the communication of buffer-like objects utilizes upper-case names like comm.Send(), comm.Recv(), and comm.Bcast().

All MPI processes begin at the same time once spawned, and all programs execute the same code in parallel. Therefore, each process must be able to recognize its own position and role within the process tree. MPI processes must use their ranks to map themselves to nodes, forming a virtual tree. The process with rank 0 (hereinafter referred to as process 0 for brevity) is the root of the tree, with the other processes being the nodes of the tree.

The official documentation is hosted by Read the Docs [7].

7.1.1 Overview

All MPI processes run the same code, which must identify the root process and the helper processes. The root process (process 0) generates the unsorted array that is to be sorted and then sends chunks of the unsorted array to each helper process using comm.Scatter(). Each helper process (i) receives its local array from the root process, (ii) invokes the sorting algorithm determined in part 1; and (iii) merges its local array with a paired node.

7.1.1.1 Dividing array into subarrays.

We generate the unsorted array as a NumPy array on process 0. The array is scattered to all the processes so each process has an equal-sized chunk of the list (or subarray). Note that this means the size of the array must be evenly divisible by the number of processes.

Figures 14 and 15 assume that we have 4 processes and an array of size 8.

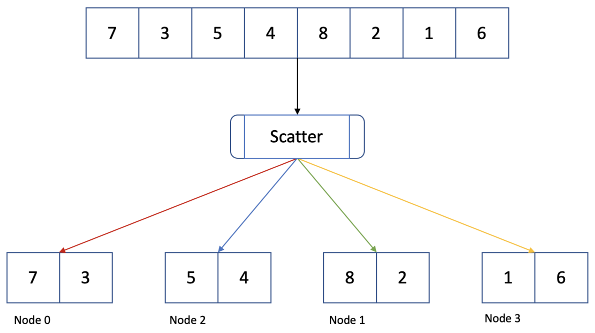

Figure 14: Example of MPI Scatter.

7.1.1.2 Sorting subarrays.

Once the subarrays have been distributed using the Scatter command, they are sorted on each processor. The specific sorting algorithm used to do the

```
# set up MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

# size of each subarray
sub_size = int(n / size)

# allocate local memory on each rank
local = np.zeros(sub_size, dtype="int")

if rank is 0:
  # generate unsorted array
  unsorted = np.random.randint(n, size=n)

comm.Scatter(unsorted, local, root=0)
```

Figure 15: Python code implementation of MPI Scatter.

intra-node sorting will be known as the subsort. The implementation of two different types of MPI merge sorts is carried out, and the type of MPI merge sort dependent upon the subsort algorithm. The options for the subsort algorithm are:

1. Built-in Python sort, where the subarray is sorted using **sorted**() (See Section 5.2). We will refer to this as message-passing merge sort.

2. Multiprocessing merge sort, where the subarray is sorted using multiprocessing_mergesort (See Section 5.3). We will refer to this as hybrid MPI merge sort.

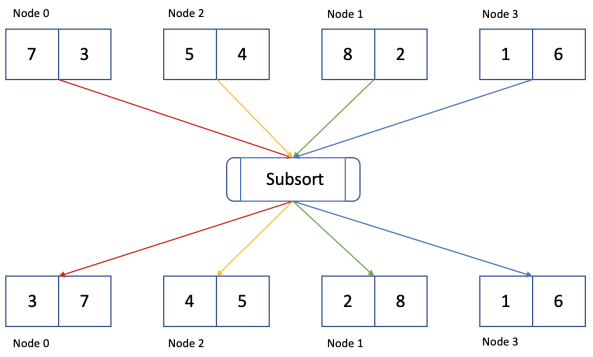

Figure 16: Example of sorting subarrays using subsort.

7.1.1.3 Merging the subarrays.

To merge the sorted subarrays, we enter a loop. Each loop, we will remove the bottom-most layer of the process tree, e.g. the leaves of the tree. The loop will continue until we have a single sorted list.

Within the loop, we determine whether the current process is a left child or a right child of the parent. Note that a left child and its parent node will be the same process. In Figure 17, the right child nodes are outlined in orange. For example, we can see that process 0 is the left child and process 1 is the right child of parent process 0.

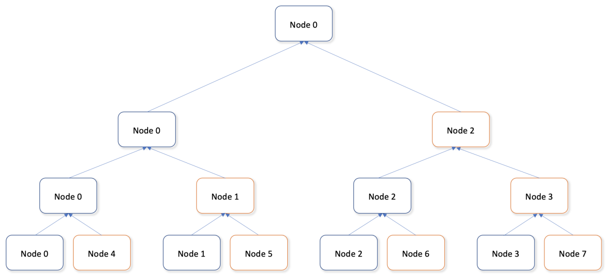

Figure 17: Example process tree.

The way that parent and child nodes are determined in the tree is by process rank. The lower half ranks are the parent processes (and left children) and the upper half ranks are the right children. For example, in the figure above, we can see that ranks 0, 1, 2, and 3 are the parent/left child processes, and ranks 4, 5, 6, and 7 are the right child processes.

Depending on the rank of the process, the node will classify itself as a left child or right child, and it will do the following:

Left child :

1. Allocate memory needed for storing right child's array in tmp

2. Allocate memory needed for result of merging lists in result

3. Receive right child's array from process $rank + split$, where $split$ is the current number of processes divided by two. For example, process 1 knows to receive data from process $1 + 4$, or process 5.

4. Merge local array with tmp in result

Right child:

1. Send local array to process *rank* − *split*, where *split* is the current number of processes divided by two. For example, process 5 knows to send its data to process 5 − 4, or process 1.

By dividing nodes in half and mapping between the halves, unique pairings of parent and child processes are guaranteed for every step of the loop. Figure 18 provides an example of merging sorted subarrays in each step of the process tree.

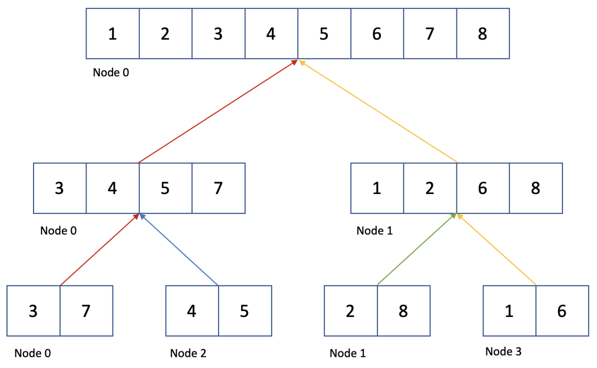

Figure 18: Example of MPI merging.

Figure 19 shows the Python implementation.

```
split = size / 2
while split >= 1:
  # rank is in upper half of processes
  # this process is a right child
  if split <= rank < split * 2:
    # send local array to parent to be merged
    comm.Send(local, rank-split, tag=0)

  # rank is in lower half of processes
  # this process is a left child/parent
  elif rank < split:
    # allocate memory for right child's array
    tmp = np.zeros(local.size, dtype="int")
    # allocate memory for merged result
    result = np.zeros(2*local.size, dtype="int")

    # receive data from right child
    comm.Recv(tmp, rank+split, tag=0)

    # merge arrays
    result = merge(local, tmp)
    local = np.array(result)
  # update split as we have removed
  # bottom layer of process tree
  # or half of the nodes
  split = split / 2
```

Figure 19: Python code example of MPI merging.

8 Analysis of MPI Sorting

The performance of all sorts was measured on Carbonate, Indiana University's large-memory computer cluster. Specific hardware details can be viewed in section 6. Message-passing merge sort (with MPI, Section 7.1.1) was executed on 1, 2, and 4 general processing nodes by using one single core on all nodes with MPI processes. Hybrid MPI merge sort was executed on 1, 2, and 4 nodes by using all available cores on all nodes for distributed MPI processes. All sorts were run on arrays of varying sizes of up to 10^7.

p	c	size	sort	subsort	time
1	1	10^7	mpi	sorted	4.237
1	1	10^7	mpi	mp	80.335
1	4	10^7	mpi	mp	78.716
1	8	10^7	mpi	mp	77.275
1	12	10^7	mpi	mp	76.570
1	16	10^7	mpi	mp	77.588
1	20	10^7	mpi	mp	78.152
1	24	10^7	mpi	mp	71.227

(a) Performance results on 1 node.

p	c	size	sort	subsort	time
2	1	10^7	mpi	mp	38.846
2	4	10^7	mpi	mp	38.508
2	8	10^7	mpi	mp	36.842
2	12	10^7	mpi	mp	37.975
2	16	10^7	mpi	mp	37.800
2	20	10^7	mpi	mp	39.400
2	24	10^7	mpi	mp	37.543

(b) Performance results on 2 nodes.

p	c	size	sort	subsort	time
4	1	10^7	mpi	mp	20.954
4	4	10^7	mpi	mp	19.475
4	8	10^7	mpi	mp	19.282
4	12	10^7	mpi	mp	18.885
4	16	10^7	mpi	mp	19.275
4	20	10^7	mpi	mp	19.463
4	24	10^7	mpi	mp	21.622

(c) Performance results on 4 nodes.

Figure 20: Results from MPI merge sorts on various nodes with array size of 10^7.

Overall, message-passing merge sort runs significantly faster than the hybrid merge sort, and runs slightly slower than multiprocessing merge sort does. Notably, the MPI merge sort using multiprocessing merge sort performs significantly worse than the MPI merge sort using sorted, despite the multiprocessing merge sort outperforming sorted in Section 6. This can be attributed to the usage of fork() system calls by the multiprocessing library to create worker processes, which is not compatible with some MPI implementations, including mpi4py. However, sorted runs completely sequentially and never faces this obstacle, resulting in the difference in time.

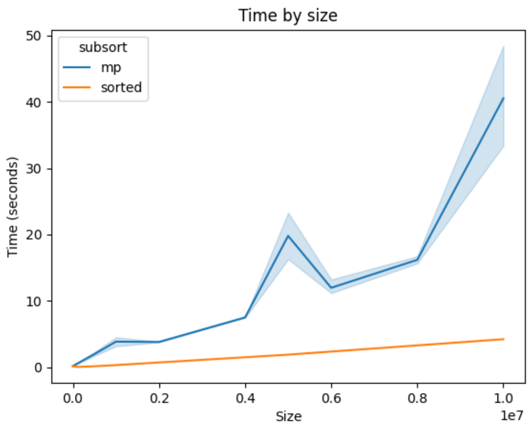

Figure 21: Time by size, average of all runs using subsort = multiprocessing.

Increased parallelism between nodes significantly speeds up run time, as is to be expected. However, due to the limited number of nodes available, there is not enough data to project the potential sort performance with further increased parallelism.

Increased parallelism within each node does appear to correlate with faster run times, but only up to a certain point. For example, in Figure 23, decreases in time are seen up until 14 cores are used, at which point parallelism becomes correlated with increasing run times.

p	c	size	sort	sub	user	node	time
1	24	10^7	mpi	mp	alex	v100	71.227
2	24	10^7	mpi	mp	alex	v100	37.543
4	24	10^7	mpi	mp	alex	v100	21.622

Figure 22: Hybrid MPI merge sort with varying processes (sub = subsort).

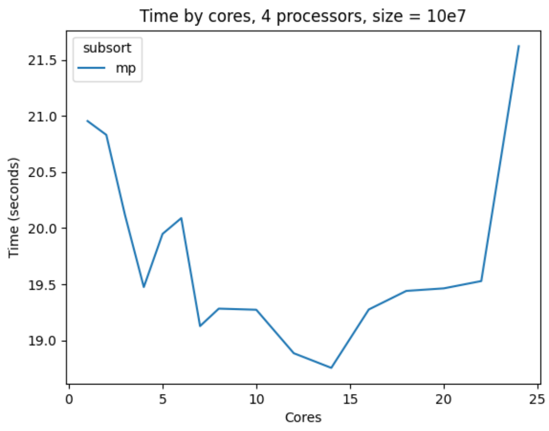

Figure 23: Hybrid MPI merge sort on 4 nodes and array size 10^7.

For arrays of smaller sizes, the relationship between intra-node parallelism and speed is much more variable and cannot be quantified. Figure 24 shows the extreme variation in time for arrays of size 10^4. However, the positive relationship between parallelism and run time after a certain point (around 12 cores for this example) still holds.

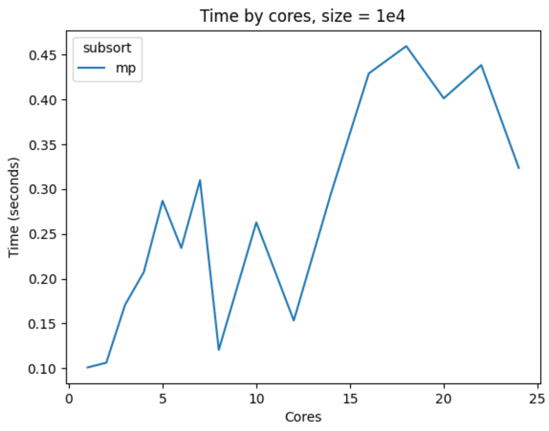

Figure 24: Hybrid MPI merge sort on 4 nodes and array size 10^4.

Finally, in comparison to the multiprocessing merge sort, the message-passing merge sort underperforms. As mpi4py runs on a distributed memory system, it must make a copy of every message that it sends, resulting in a large overhead for every message that must be sent. On the other hand, since the multiprocessing merge sort runs on a SMP, mes-

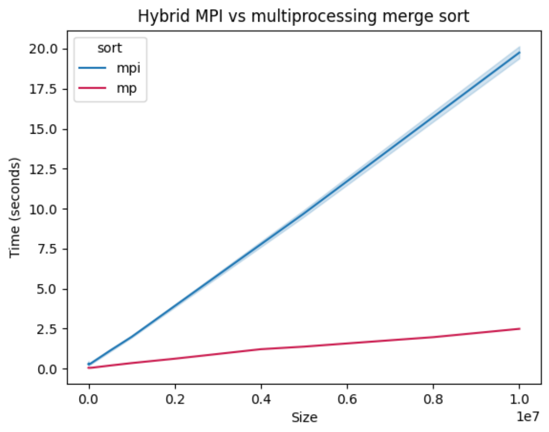

Figure 25: Message-passing vs multiprocessing merge sort.

sage passing between cores is much faster due to the lighter memory requirements.

9 Conclusion

This paper introduces and implements three parallel merge sort algorithms in the Python programming language: multiprocessing merge sort, message-passing merge sort, and hybrid MPI merge sort. We run each sorting algorithm on the Carbonate computing cluster, and using Cloudmesh, we benchmark, compare, and analyze the results of the sorting algorithms. We also measure the performance sequential merge sort and the built-in Python sort to act as a benchmark for the parallel sorts.

We find that multiprocessing merge sort is the fastest sorting algorithm, followed by the message-passing merge sort, and then the hybrid MPI merge sort. We discover that multiprocessing merge sort outperforms even the built-in Python sort, running 1.5x faster in some cases. However, for small array sizes, the multiprocessing merge sort is relatively slow due to the associated overhead. We conclude that while for smaller problems, it may be more beneficial to use a sequential solution, parallelism should be implemented once the problem size is sufficiently large. Note that the best parallel algorithm to use, along with the threshold for "sufficiently large", depends upon the nature of the problem, the network type, and the hardware and software present.

References

[1] R. Cole, “Parallel Merge Sort,” Tech. Rep. [Online]. Available: http://www.inf.fu-berlin.de/lehre/SS10/SP-Par/download/parmerge1.pdf

[2] M. Jeon and D. Kim, “Parallel Merge Sort with Load Balancing,” *International Journal of Parallel Programming*, vol. 31, pp. 21–33, 02 2003.

[3] A. Radenski, “Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs,” *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 367–373, 01 2011. [Online]. Available: https://digitalcommons.chapman.edu/scs_books/19/

[4] “Cloudmesh with MPI,” 2022. [Online]. Available: https://github.com/cloudmesh/cloudmesh-mpi

[5] G. von Laszewski, J. P. Fleischer, and G. C. Fox, “Hybrid reusable computational analytics workflow management with cloudmesh,” 2022. [Online]. Available: https://arxiv.org/abs/2210.16941

[6] “multiprocessing - Process-based parallelism,” 2022. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html

[7] L. Dalcin, “MPI for Python,” 2022. [Online]. Available: https://mpi4py.readthedocs.io/en/stable/

Table 1: Complexity of sort algorithms

Sort	Average Time Complexity	Best Time Complexity	Worst Time Complexity
Bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap sort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$
Quick sort	$O(n\log(n))$	$O(n\log(n))$	$O(n)\ \log(n))$
Merge sort	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$