Parallel Sorting by Regular Sampling: A Comparison of Implementations

University of Cape Town

CSC4028Z

24 May 2023

Abstract

This report investigates 3 implementations — OpenMP, MPI, and hybrid (OpenMP + MPI) — of the Parallel Sorting by Regular Sampling algorithm. All implementations achieve an effective speedup. It was found that OpenMP performs better with smaller number of processors, and with fixed number of processors with larger problem size plateaus at the greatest speedup. The MPI implementation appears more efficient with a larger number of processors. The hybrid implementation was found to be ineffective in comparison to the OpenMP and MPI implementations.

Authors:
Ahmed Ghoor (GHRAHM004)
Taariq Mowzer (MWZMOH012)

1 Introduction

Sorting has received much attention from the computer science community and many algorithms have been constructed to sort a list of n items. Bubble sort and selection sort are algorithms which work well for small lists ($n \leq 50$) but scale $\mathcal{O}(n^2)$ time-wise. Merge sort and quicksort are the go to algorithms for large lists. Merge sort runs in guaranteed $\mathcal{O}(n \log n)$ time, whereas quicksort is worst case $\mathcal{O}(n^2)$ but on average is $\mathcal{O}(n \log n)$.

A sequential comparison-based sorting algorithm is $\Omega(n \log n)$ (Sedgewick and Wayne [2011]). Meaning no sequential sorting algorithm exists which scales better than merge sort and quicksort. To sort large quantities of data in a practical amount of time, scientists must turn to sorting algorithms making use of parallelisation and High Performance Computing (HPC) resources.

Shi and Schaeffer [1992] developed Parallel Sorting by Regular Sampling (PSRS). PSRS is an asymptotically optimal parallel sorting algorithm intended for Multiple Instruction - Multiple Data (MIMD) processes.

Modern HPC resources have two main paradigms of parallelisation. First, Message Passing Interfaces (MPI) where processes perform computations on separate memory. Processes communicate via message passing. Second, shared memory processing where threads perform computations on a shared memory. A common API for shared memory processing is OpenMP (OMP).

This report investigates implementations of PSRS on UCT's HPC cluster Curie. Three implementations of PSRS are compared. A pure MPI approach, a pure OMP approach and a hybrid MPI + OMP approach. These approaches are compared to a conventional sequential quicksort.

2 Parallel Sorting by Regular Sampling (PSRS)

2.1 Description of algorithm

Let A be the list of n elements to be sorted. Let p be the number of processors. For simplicity, we assume that p is even and n is a multiple of p^2, and so $w = n/p$, w/p and $p/2$ are integers.

1. Divide the Array A into p contiguous subarrays of size n/p.
2. Each processor sorts a contiguous subarray.
3. Regular Sampling:
   - 3.1. Let S^j be the jth contiguous subarray. The regular samples of S^j are the $p-1$ elements $S_{w/p}, \ldots S_{(p-1)w/p}$.
   - 3.2. Collect all regular samples of each contiguous subarray to a single processor. Sort all the regular samples. Call this sorted list R.

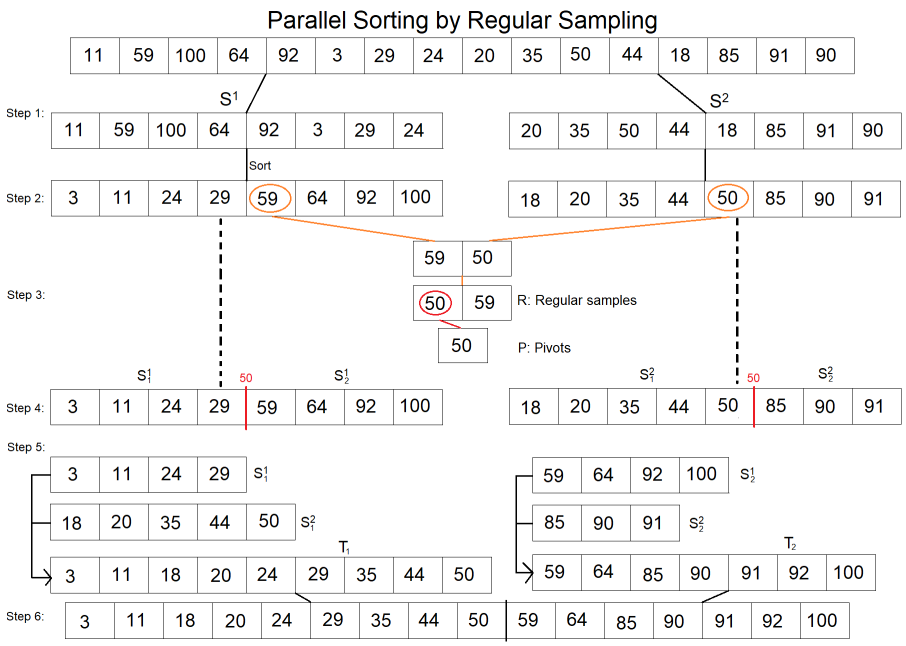

Figure 1: An example of PSRS for $n = 16$ and $p = 2$. When $p = 2$, there is only one pivot.

3.3. Construct the pivots P. The pivots is a list size p consisting of the elements $R_{p/2}, R_{p+p/2}, \ldots, R_{(p-1)p+p/2}$.

4. Each processor receives the list of pivots P. Each subarray S^j is partitioned into further sub-subarrays $S_1^j, \ldots, S_p^j$, so that the elements of S_i^j are greater than P_{i-1} but less than or equal to P_i. Since S^j is already sorted, the sub-subarrays can be found with a binary search.

5. Processor q $(1 \leq q \leq p)$ receives the sub-subarrays $S_q^1, \ldots S_q^p$. Then p-merge sort the sub-subarrays $S_q^1, \ldots S_q^p$ forming a sorted list T_q.

6. The concatenated list $T = T_1 + \ldots T_p$ is the final sorted list.

2.2 Time complexity and scaling

Shi and Schaeffer [1992] showed that PSRS takes $\mathcal{O}(n/p \log n)$ time when $n \geq p^3$. If we assume that the $\mathcal{O}$ notation gives accurate predictions for the runtime of PSRS and sequential sort, we obtain the following theoretical predictions.

- **Speedup:** A list of size n takes $\mathcal{O}(nlogn)$ time to sort using a sequential algorithm and $\mathcal{O}(n/p \log n)$ time with PSRS using p processors. Hence,

$$\text{Theoretical speedup} = \frac{\text{Sequential time}}{\text{Parallel time}} = \frac{n \log n}{n/p \log n} = p. \tag{1}$$

  PSRS is a theoretically asymptotically optimal parallel sorting algorithm.

- **Strong scaling:** For sufficiently large n, the run time of PSRS as a function of the

number of processors p is $T(p) = n/p \log n$. Hence

$$T(p) \propto 1/p. \tag{2}$$

An interpretation of this is doubling the number of processes should halve the solving time when the problem size is held fixed. Theoretically, PSRS is strongly scalable.

- **Weak scaling:** According to HPC-Wiki [2022] the complexity of the task is important in determining the *size* of the problem. An optimal sequential algorithm sorts a list of size n in $\mathcal{O}(n \log n)$ complexity. Hence, we use $n \log n$ — and not n — as a measure of size.

  The problem size per processor is then $\omega = (n \log n)/p$. Holding w fixed, we see that the time to solve a problem of size ω with p processors is $T(p) = n/p \log n = (n \log n)/p = \omega$. Hence, the efficeincy for p processors is

$$\text{Efficiency}(p) = \frac{T(1)}{T(p)} = \frac{\omega}{\omega} = 1. \tag{3}$$

  If the problem size increases at the same rate as the number of processors, the time taken to solve the problem remains the same. Theoretically PSRS is weakly scalable.

3 Implementations

An important step in the PSRS is the p-merge sort done in step 5. In Shi and Schaeffer [1992] it is stated that the p-merge sort is expected to take less than $\mathcal{O}(w \log p)$ time. This suggests that the authors intended for the p-merge sort to be done with a a priority queue (or heap structure). Since increased performance from heap structures only occurs for large p and p is small in our experiments ($p \leq 32$), we opted for the naive method of performing a p-merge sort which takes $\mathcal{O}(pw) = \mathcal{O}(n)$ time. This does not effect the theoretical time complexity of the algorithm.

The MPI and Hybrid implementations use a Dihedral pairing (See Appendix A) to perform the message passing required in step 5. An alternative to this approach is to use gather commands. No performance difference was found between these approaches. We use the Dihedral approach since it is versatile, simple, and foremost it is interesting.

All implementations were coded in C and take an array of integers as input.

3.1 Sequential

The `qsort` function availables in C's standard library was used as the sequential sorting algorithm.

3.2 OpenMP

Shane Fitzpatrick's implemention of PSRS in OpenMP is used. The repo is available here. Fitzpatrick's implementation of PSRS did a quicksort on the sub-subarrays $S_1^j, \ldots S_p^j$. The

code was adapted to do a p-merge sort of the sub-subarrays as suggested in Shi and Schaeffer [1992]. This yielded a 15% performance increase during preliminary testing on a home PC.

To avoid overhead, the directive `#pragma omp parallel` was called once at the start of the code. If only a single thread is required to perform a task, `#pragma omp single` is used. Shared memory refers to memory shared between all threads. Threads have both read and write access to shared memory.

Between each step there is a barrier to ensure synchronisation. The directive `#pragma omp barrier` was used.

1. Each thread creates a local copy of the subarray called `local_a`. For synchronization, all threads have access to the memory location of each of the local copies.
2. Each thread sorts its local copy `local_a`.
3. Each thread computes the regular samples of `local_a`, placing the regular samples in shared memory. A single thread then sorts the regular samples and places the pivots in shared memory.
4. Each thread reads from the pivot array and computes the sub-subarrays $S_1, \ldots, S_p$. The data is not copied, instead the location and length of each of the sub-subarrays is recorded in the sorted local copies `local_a`.
5. Since the memory addresses of the local copies `local_a` are in shared memory. Each thread has access to all other sub-subarrays. In practice, it only reads the arrays which are relevant to it. Each thread performs a p-merge sort.
6. The results of the p-merge sort are written directly into an array in shared memory. This implicitly concatenates the lists.

3.3 MPI

MPI's functions are well suited to implement PSRS.

1. The master process distributes the size of the array subsegment to each process using `MPI_Scatter`. The master process uses `MPI_Scatter` again to distribute the array subsegments.
2. Each process sorts their subarray.
3. Each process computes their regular samples and sends them the master process using `MPI_Gather`. The master process computes the pivots, and sends the pivots to every process using `MPI_Bcast`.
4. Using the pivots, each process computes the location of their sub-subarrays $S_1^j, \ldots S_p^j$.
5. Each process sends the corresponding sub-subarray S_q^j to process q using `MPI_Send`. Process q receives the subarray using `MPI_Recv`. To prevent deadlocks and ensure efficient communication a dihedral pairing (See Appendix A) is used.

6. Each process performs a p-merge resulting in the sorted list T_j. The length of the sorted lists T_j are sent to the master process using `MPI_Gather`. Subsequently, `MPI_Gatherv` is used to gather the variable length arrays at the master process into a single sorted array T.

3.4 Hybrid (MPI + OpenMP)

The Hybrid implementation of PSRS is the most complicated. Within the hybrid implementation there are u processes each having v threads. This results in $p = u \times v$ processors. At each step of the PSRS algorithm, processes communicate with each other using the MPI communications. Threads within each process communicate via shared memory.

This implementation uses `MPI_FUNNELED`. Meaning, MPI calls are only performed by the master thread in each process. `MPI_FUNNELED` is preferred over `MPI_THREAD_MULTIPLE` for performance. A master process is used to synchronize the processes. A master thread — accessed using `#pragma omp master` — is used to synchronise threads within a process.

1. The master process distributes n the size of the array to each process using `MPI_Scatter`. The subarrays of size n/u are then distributed using `MPI_Scatter`. Within each process, each thread makes a local copy of the subarray of size $n/(uv) = n/p$ called `local_a`. Like the OMP implementation, within each process, each thread has access to the memory location of each of the other threads' subarrays.

2. Each thread sorts its local copy `local_a` of the subarray.

3. Each thread computes its regular samples, placing them in shared memory. The master thread in each process sends the regular samples of the process to the master process using `MPI_Gather`. The master thread in the master process sorts the regular samples and determines the pivots. Pivots are communicated back to each process via `MPI_Bcast`. Pivots are written in shared memory within each process so that each thread can read the pivots.

4. Each thread computes the sub-subarrays $S_1^j, ...S_p^j$. Like the OMP implementation, the data is not copied, instead the location and length of each of the sub-subarrays is recorded in the sorted local copies `local_a`.

5. Since `MPI_Funnel` is used, only the master thread can be used for communication. Due to memory sharing, the master thread has access to the local copies of all the other threads.

   A naive implementation would require v^2 communications between two processes (each thread communicates with every other thread). A more sophisticated approach is taken where all v^2 communications are sent in two big sends. The v^2 communications can be considered to be a single communication where Instead of sending v^2 1D arrays we send a 2D array whose first dimension is of length v^2. MPI does not support sending 2D arrays, so the 2D array is serialised into a 1D array. An additional array is sent containing the information required to rebuild the 2D array.

To prevent deadlock and ensure efficient communication between processes, a dihedral pairing is used (See Appendix A)

6. The previous messages were written into shared memory. So like the OMP implementation, each thread performs a p-merge sort. The results of the p-merge sort is written directly into shared memory. Once the p-merge sort is complete, the master thread of each process can gather the sorted T_j's to the master process. This is done with a `MPI_Gather` and `MPI_Gatherv` similar to the MPI implementation.

3.5 Correctness of implementation

To confirm correctness, each implementation was run on sample tests. The output was then compared to the expected output. An error is raised if the output and the expected output differ by a single element. These sample tests and expected output were generated using the same methods as the benchmarking test data.

To confirm that no issues occurred during benchmarking, the same check for correctness takes place. Checking for correctness is not included in the benchmarking time.

4 Experiments

We investigate the following three questions:

1. Do the PSRS implementations exhibit strong scaling?
2. Do the PSRS implementations exhibit weak scaling?
3. Which implementation achieves the greatest speedup?

4.1 Test cases

Test cases were generated using Python. Each test case consists of test input and expected output. Once generated test input and expected output are written to separate files to be used in benchmarking.

Test cases were generated using the following procedure. An increasing list is constructed by generating random numbers from 1 to 10 and taking the cumulative sum. This increasing list is the expected output. The test input is obtained by shuffling the cumulative sum list with Python's `random.shuffle`.

At no point during test case generation was a sorting algorithm used. This guarantees that the expected output is independent of any sorting algorithm implementation.

The size of test cases ranged from lists of size $4^5 = 1024$ to $4^{12} \approx 16$ million.

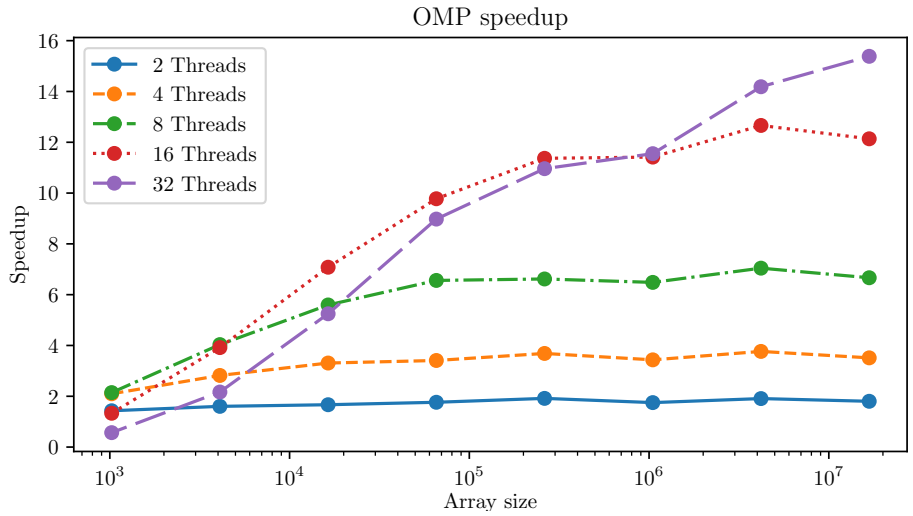

Figure 2: Results of running OMP implementation of PSRS on randomly shuffled arrays.

4.2 Benchmarking

All 4 implementations were run on all the test cases in different scenarios. The following scenarios were investigated:

- OpenMP with $2, 4, 8, 16, 32$ threads.
- MPI with on 1 or 2 nodes. $2, 4, 8, 16, 32$ processes
- Hybrid on 2 nodes with $2, 4, 8$ processes with $2, 4$ threads.

To obtain the time for each test case, the PSRS algorithm was run 20 times, and only the minimum time was recorded.

All implementations were run on The University of Cape Town's HPC. The Curie partition was used which has 10 nodes consisting of 64 DELL C6145 cores each.

4.3 Results and Discussion

See Figure 2. Observing the largest array of size roughly 10^7, we see that OMP scales strongly. For a fixed number of threads that the speedup increases as the array size increases until it plateaus. Specifically, using p threads speedup usually plateaus just below a speedup of p. It is expected due to overhead that we do not reach a speedup of p, however the speedup obtained is remarkably close.

With 32 threads only a speedup of 16 was achieved. However, 32 thread OMP has not shown signs yet of plateauing. It is expected that the speedup of the 32 thread OMP will still increase for even larger array sizes.

Observe Figure 3. The first observation to be made about MPI is that no drop of efficiency occurs when the algorithm is run on on 2 nodes instead of 1. PSRS was designed to have very

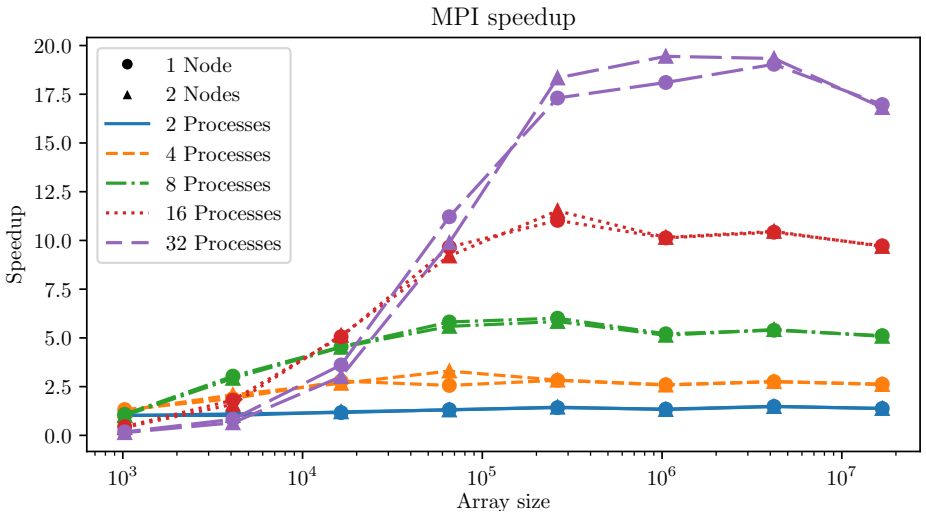

Figure 3: Results of running MPI implementation of PSRS on randomly shuffled arrays.

few communications, so this is expected. With this in mind, to simplify the next experiment the Hybrid implementation will be run on only 2 nodes.

Observing the largest array of size roughly 10^7, we see that MPI scales strongly. The same plateauing behaviour which occurred for OMP happens here. Interestingly, MPI with 32 processes plateaus at a speedup of about 20.

To further simplify further tests, an investigation is done to find the optimal balance of threads to processes for the hybrid implementation. Figure 4 shows that the hybrid implementation performs slightly better with more processes than threads. Notably when the number of processes is 2, the hybrid implementation performs catastrophically. There is no reason the PSRS algorithm should perform so badly. Presumably there is a mistake in the implementation that is only reflected when 2 processes are used.

From Figure 5, note for large arrays, the hybrid implementation scales strongly. The 32 processor scenario plateaus at a speedup of 16. Note that the scenario using 4 processors (2 processes and 2 threads) achieves a speedup under 2. This confirms the previously seen catastrophic performance of this implementation when 2 processes are used.

The following observations can be made from Figure 6. For large data sizes OMP eventually achieves the greatest speedup. OMP's speed plateaus at a higher level than MPI and hybrid implementations. For the exception of 32 processors, OMP has not yet plateaued and will likely achieve a greater speedup.

The MPI and hybrid implementations perform relatively similarly. In the case of 4, 16 and 32 processors the MPI version performs better than hybrid implementation. The hybrid implementation performs marginally better with 8 processors.

For very small lists all implementations perform badly. Regardless of implementation using 32 processors to sort an array of 1000 elements does not yield much speedup. Across all implementations speedup only occurs for larger size arrays with the tipping point being

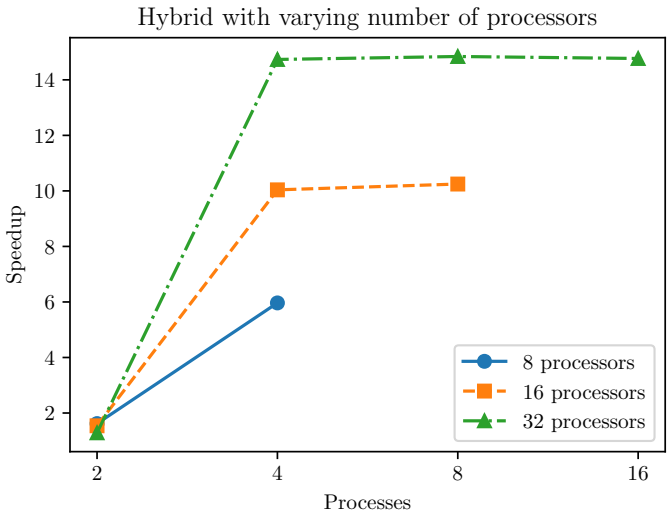

Figure 4: Running hybrid implementation on 2 Nodes on array of size $4^{12} \approx 16$ million. Note # processors = # processes × # threads.

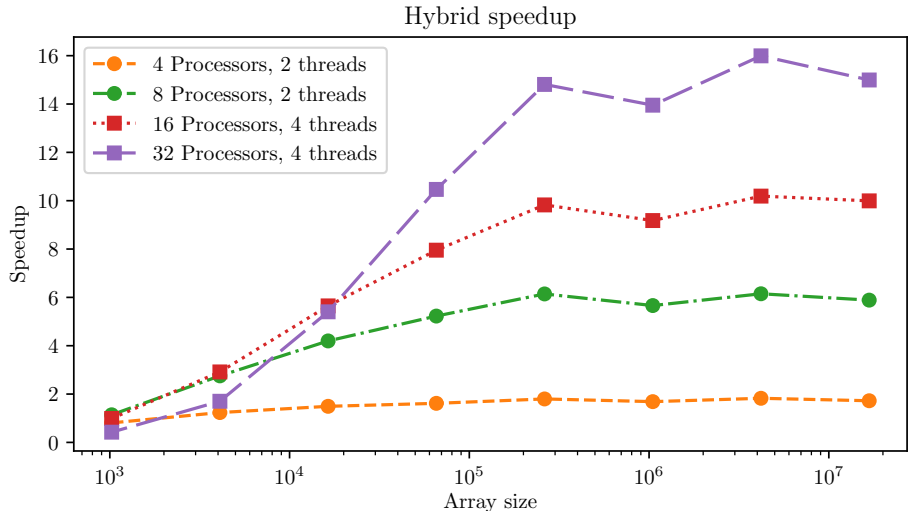

Figure 5: Running hybrid implementation of PSRS on 2 Nodes on randomly shuffled arrays.

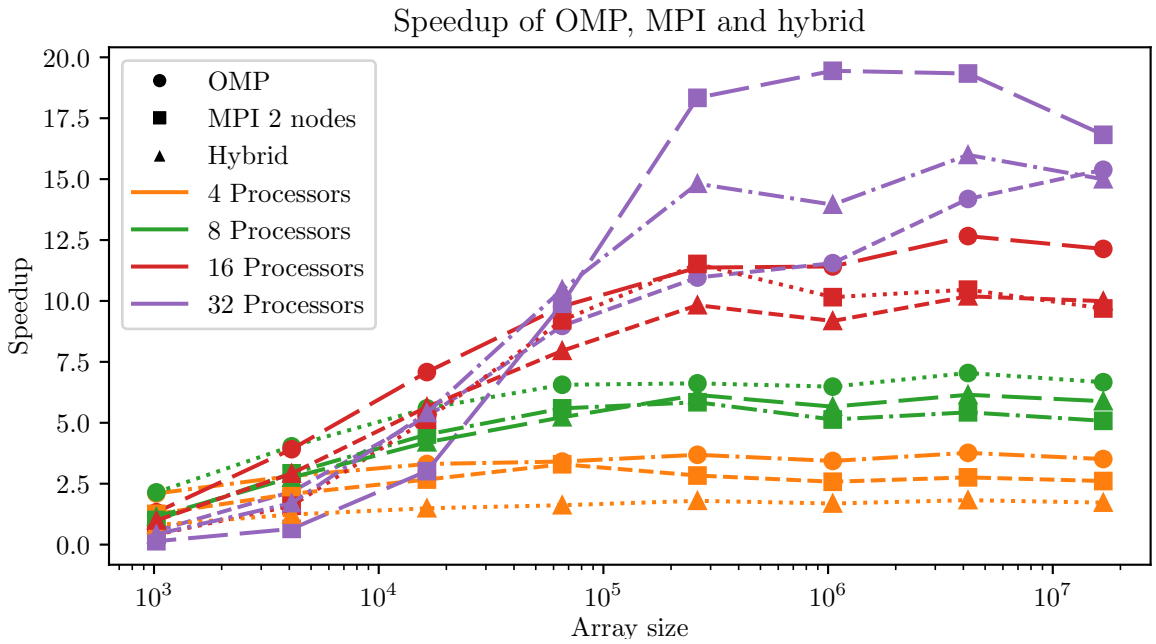

Figure 6: Speedup comparison of each implementation

arrays greater than $10^5 = 100$ thousand. However, out of all the poor performances, OMP consistently achieves the greatest speedup for small arrays.

Overall though, the performance of the implementations are quite close. No implementation is significantly superior to any other implementation, and differences may have been caused minor differences in code. These results do not suggest any parallelisation paradigm (OMP, MPI, or hybrid) is much superior to any other paradigm.

Figure 7 shows the efficeincy of each implementation (See section 2.2 for definition of Efficeincy). For all implementations efficiency decrease to about 50% as the number of processors increase. All implementations hence do not weakly scale well.

Surprisingly, OMP starts off with an efficeincy of 120% when less than 4 processors are used. Presumably, the implementation lucked out on very good pivot selection during it's quicksort routines resulting in a quick run time.

The MPI implementation's efficiency is the least effected by the increase in processors. This seems to suggest that MPI may scale better for large problems with large number of processors.

The hybrid implementation's efficiency starts off the worst. This is likely caused from the previously observed fact that when 2 processes are used the hybrid implementation performs very poorly.

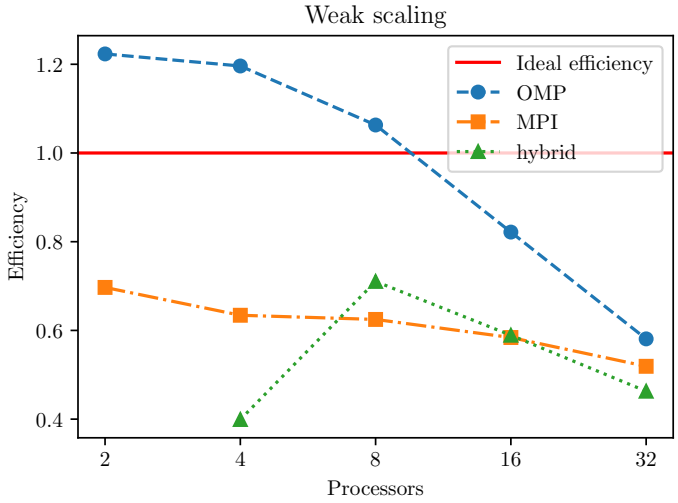

Figure 7: Weak scaling of each implementation. The unit of work was computed for $n = 4^{12}, p = 32$, meaning $\omega = \frac{n \log n}{p} \approx 8.7 \times 10^6$.

5 Conclusions

In short, the following has been demonstrated.

1. All PSRS implementations exhibit strong scaling.
2. All PSRS implementations do not weakly scale.
3. MPI achieved the greatest speedup of just under 20 using 32 processors.

PSRS does not strongly scale effectively for arrays smaller than 10^4 =10 thousand. For small arrays, the greatest speedup achieved was 7.5 with the OMP implementation which used 16 threads. For small arrays the greater the number of processors, the worse the speedup. This makes sense as PSRS is supposed to only be effective when the array size is much larger than the number of processes ($n \geq p^3$).

For arrays larger than $10^6 = 1$ million, PSRS becomes quite effective and increasing the number of processors does increase speedup. The greatest speedup was 20 achieved by the MPI implementation using 32 processes split between 2 nodes. However, out of all the implementations it appears that the OMP implementation achieves the greatest speedup for a fixed number of processes on large arrays. It is expected that the OMP implementation would yield the greatest speedup using 32 threads if larger arrays (of size 10^9) were benchmarked. The authors approximate that OMP with 32 threads is capable of a speedup of 25.

In terms of work per node (weak scaling), OMP is the most efficient. However, MPI's efficiency seems the least effected by an increase in the number of processors. This is by virtue of the PSRS algorithm which requires only $\mathcal{O}(n)$ communication. Similarly, MPI's performance is independent of the number of nodes it is run on. It is expected that the MPI implementation would be more efficient if many processors are used. This is good as many processors must be run on multiple nodes where use of MPI is required.

In all respects, the hybrid implementation performed worse than either the OMP or MPI implementations. It also performs terribly when 2 processors are used, but this may be a fault in the authors implementation. Considering the hybrid implementation was by far the most complicated, it does to seem worth it to implement PSRS in a hybrid MPI + OMP paradigm.

In contrast, the MPI implementation was easy to code, performs well on large datasets where PSRS is effective, appears to weakly scale the best, and is capable of working on multiple nodes. For this reason, in the authors preferred implementation is MPI.

References

HPC-Wiki, June 2022. URL `https://hpc-wiki.info/hpc/Scaling#Weak_Scaling`. Accessed: 2013-05-23.

R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 2011.

H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

A Dihedral pairing

The purpose of the dihedral pairing is to solve the following problem.

Handshake problem. Suppose there are $n > 2$ politicians labelled from 1 to n. A politician is happy when they have shaken hands with every other politicians, and also shaken their own hand. A politicians can only be involved in at most one handshake per turn. How can the politicians be organised so that in n turns all n politicians are happy.

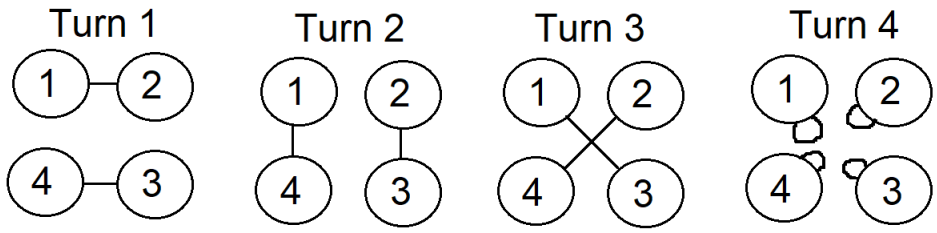

Figure 8: Solution to Handshake problem when $n = 4$. At Turn 4 all the politicians shake their own hand.

The handshake problem is equivalent to a communication problems we face in MPI. Namely, that N processes need to send unique information to the N other processes. If communication safety is required, and blocking sends are used, then communication is similar to a handshake in that the receiving and sending processes both need to agree to communicate (technically this is implementation specific). Hence, a solution to the Handshake problem would give an efficient way to pair processes for communication and prevent deadlock.

We give a solution that makes use of the Dihedral group D_{2n}. The group D_{2n} encodes the symmetries of a polygon with n points. The Dihedral group is generated from two symmetries: rotation r, and reflection s.

A group action of D_{2n} on the vertices of a polygon $\{0, \ldots, n-1\}$ can be defined as follows:

$$r \cdot x = x + 1 \text{ (mod n) and } s \cdot x = (n-1) - x$$

for $x \in \{0, \ldots, n-1\}$.

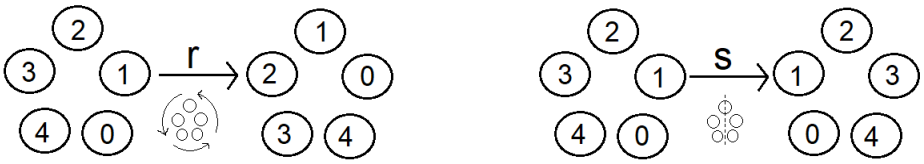

Figure 9: Visualisation of the group actions r and s on a pentagon.

We make the observation that doing a reflection followed by a reflection leads is the same as doing nothing i.e. $s^2 = 1$, and that the rotation actions $1, r, \ldots r^{n-1}$ takes a vertex to every other vertex. We combine these facts and the well known property that $rs = sr^{-1}$ to construct the following solution to the Handshake problem.

On turn k, politician x should shake hands with person $(sr^k) \cdot x$.

Since $(sr^k)(sr^k) \cdot x = (sr^k)^2 \cdot x = 1 \cdot x = x$, this map does in fact lead to a handshake. It is similarly easy to show that x does not shake the same hand twice, from which it follows that x shakes every hand (including their own).

This can easily be implemented with the following function.

```
int shake_hands(int p_x, int turn, int n){
        int p_y = p_x + turn;
        p_y %= n;
        p_y = n - 1 - p_y;
        return p_y;
}
```

Given politician x, turn number and the number of politicians, `shake_hands` returns the politician whom x should shake hands with.