Parallel BucketSort Algorithm in MPI

COMP2444: Networks and Scalable Architecture

University Of Leeds

"Why did the functions stop calling each other?

Because they had constant arguments..."

March 6, 2016

Question 6

Time Analysis

We will perform a time analysis of the parallel algorithm between the start and end of timer. Firstly, the algorithm is broken down into five parts:

1. Scatter the global array into big buckets

The `MPI_Scatter` method is used to distribute the global array, so the communication will be assumed to be in parallel hence,

$$t_{comm1} = t_{startup} + nt_{data}$$

2. Pour each rank's big bucket into a small bucket

This stage is done in each rank independently. There are five operations per loop (1×addition, 1×subtraction, 3×assignment) ignoring the if statement which on a relative scale executes a negligible amount of times. The loop ranges from 0 to `dataPerProc` which is equal to n/p. Thus,

$$t_{comp1} = 5n/p$$

3. Pour each rank's small bucket back into the correct big bucket

At the start, since the same big bucket is used for storage, it is re-initialized to all 0s including the variable keeping track of it's size hence,

$$t_{comp2} = n + 1 \approx n$$

Afterwards, each rank sends the contents of it's small buckets to the appropriate rank's big bucket. To simplify the analysis, we will consider that when a rank needs to send values back to itself, it will do so through MPI communication rather than a C routine.

Also, despite involving non-blocking communication, the sending, receiving and waiting all happen consecutively which means that communication is sequential (The use of non-blocking methods is purely to prevent deadlocking when sending data larger than the internal buffer).

Thus the time taken taken for communication would be,

$$t_{comm2} = n_{smallbuckets} \times t_{startup} + \sum_{n} t_{data} = pt_{startup} + nt_{data}$$

Once data is received, it is then copied from the big bucket to the small bucket hence the total computation spent on copying data is,

$$t_{comp3} = nt_{data}$$

Finally, before performing the serial sort, `MPI_Barrier` is called to ensure that all of the small buckets have successfully transferred their contents into big buckets. However, this isn't really needed (perhaps it is more suited for mental comfort) due to the previous `MPI_Wait`. Thus, the time delay due to the `MPI_Barrier` is 0.

4. Serial sort each big bucket

Each rank performs a serial sort for it's big bucket using a quicksort algorithm so in the worst case we have,

$$t_{comp4} = O(nlog(n))$$

5. Join all big buckets

Each rank sends the contents of it's big bucket back to rank 0 which means that all the data is transferred back to rank 0. Thus, the total amount of communication needed is,

$$t_{comm3} = n_{bigbuckets} \times t_{startup} + \sum_{n} t_{data} = (n/p)t_{startup} + nt_{data}$$

Finally, summing all the parts to yield a final answer for the overall time analysis:

$$\begin{aligned} t_{comp} &= t_{comp1} + t_{comp2} + t_{comp3} + t_{comp4} \\ &= (5n/p) + (n) + (nt_d) + (O(nlog(n)) \\ &= n(1 + 1/p) + nt_d + O(nlog(n)) \end{aligned}$$

$$\begin{aligned} t_{comm} &= t_{comm1} + t_{comm2} + t_{comm3} \\ &= (t_s + nt_d) + (pt_s + nt_d) + ((n/p)t_s + nt_d) \\ &= (1 + p + n/p)t_s + 3nt_d \end{aligned}$$

$$\begin{aligned} t_{total} &= t_{comm} + t_{comp} \\ &= n(1 + 1/p) + (1 + p + n/p)t_s + 4nt_d + O(nlog(n)) \end{aligned}$$

Measurements

The graphs below show the time complexity of the implemented bucketsort algorithm under certain constraints (the raw not averaged data of the graphs are included in the appendix).

To start with, figure 1 (time vs n) shows a linear relationship when **p is fixed**. This can hold true for our derived formula under certain conditions as well:

$$\begin{aligned} t_{total} &= n(1+1/p) + (1+p+n/p)t_s + 4nt_d + O(nlog(n)) \\ &\approx (n+An) + (B+Cn) + Dn + O(nlog(n)) \\ &\approx K + Pn + O(nlog(n)) \end{aligned}$$

However, the obtained graph is rather linear so it seems we require much larger values of n before the graph show some significant $nlog(n)$ behaviour.

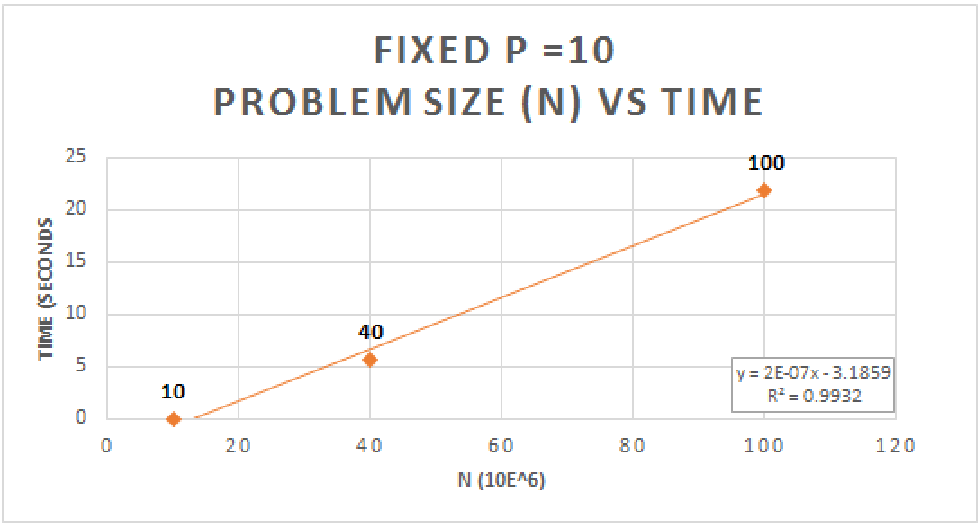


Figure 1: Time vs problem size , with the number of processors being a constant 10 CPUs.

As for figure 2 (time vs p), it shows non-linear behaviour when **n is fixed**. This can also hold true for our derived formula, again under certain conditions too:

$$\begin{aligned} t_{total} &= n(1 + 1/p) + (1 + p + n/p)t_s + 4nt_d + O(nlog(n)) \\ &\approx (A/p) + (Bp + Cp) + D \\ &\approx K/p + Qp \end{aligned}$$

The equation above can be approximated to a quadratic curve using a Taylor expansion hence implying the approximate quadratic behaviour on the graph, at least over the given range.

The time taken between 4 and 5 processes increases by 700% compared with between 8 and 10 process which increases by 25% is because with 4 process, the algorithm is ran solely on a single machine hence very fast communication between it's local cores.

Also, the small local fluctuations between each data reading in all graphs is due to a slightly different state of the hardware and software (e.g. variations in CPU usage, the randomly generated input, ram writing to different locations).

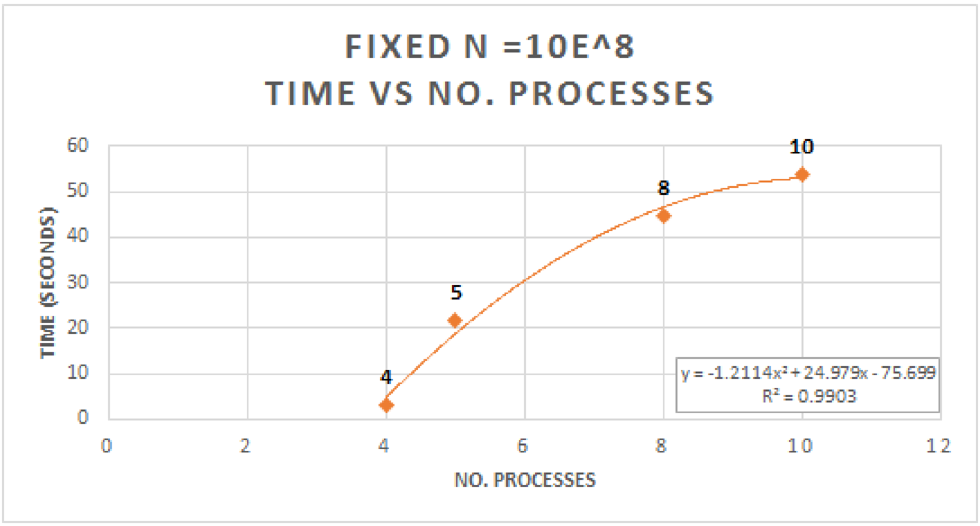
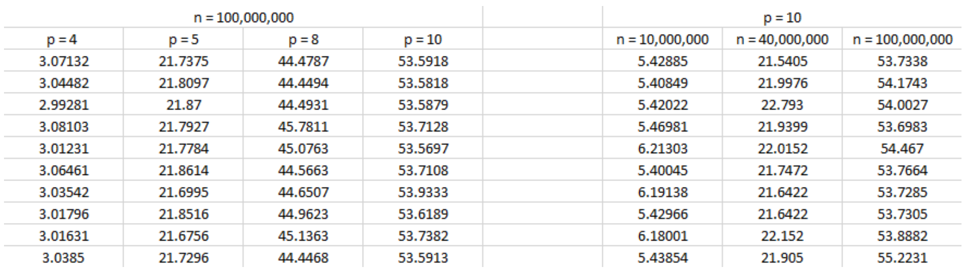


Figure 2: Time vs number of processes , with the problem size being held a constant 100,000,000 floating point numbers.

Lastly, the fluctuations between each data reading and the next is largely due to



n = 100,000,000					p = 10		
p = 4	p = 5	p = 8	p = 10		n = 10,000,000	n = 40,000,000	n = 100,000,000
3.07132	21.7375	44.4787	53.5918		5.42885	21.5405	53.7338
3.04482	21.8097	44.4494	53.5818		5.40849	21.9976	54.1743
2.99281	21.87	44.4931	53.5879		5.42022	22.793	54.0027
3.08103	21.7927	45.7811	53.7128		5.46981	21.9399	53.6983
3.01231	21.7784	45.0763	53.5697		6.21303	22.0152	54.467
3.06461	21.8614	44.5663	53.7108		5.40045	21.7472	53.7664
3.03542	21.6995	44.6507	53.9333		6.19138	21.6422	53.7285
3.01796	21.8516	44.9623	53.6189		5.42966	21.6422	53.7305
3.01631	21.6756	45.1363	53.7382		6.18001	22.152	53.8882
3.0385	21.7296	44.4468	53.5913		5.43854	21.905	55.2231