

PAP

Why do we need parallel computing?

Some of motivations towards exploring parallel computing solutions are:

- Computation
 - There could be large number of iterations
 - The algorithm may take a too long of a time to converge
- Data
 - The memory consumed maybe too large than what can be accommodated by conventional PC RAM.
 - Output can be too big and may take forever to save/transfer.

Moreover, a **big** application can hoard all the resources and cause other applications running on the machine to starve.

Thanks to Moore's law, the conventional wisdom, suggested to wait for newer generation of processors to get more processing power. However, we have reached a point, wherein we can't get more processing power by just increasing frequency, so manufacturers are pushed to increase the number of cores, to derive processing power. Hence, the need to have exposure into parallel computing is more now than ever before.

Parallel computing though is the way forward, it's not a magic solution. It comes with it's own range of caveats and problems. For example,

- How will you divide the data between units?
- How to distribute data to each of the units?
- How are the dependencies between data/results handled?
- How to synchronize various parallel units?
- How reconstruction of the results of individual units occur?
- How will you modify your conventional(serial) algorithms to exploit parallelism? etc.

With the advent of big data and IOT, parallel computing has an ever increasing number of applications in various domains in industry and research. For ex:

- Data analysis
- Numerical simulations
- AI
- Graphics and motion pictures etc.

The people who are working in this domain to provide foundations for much of the applications described above are the unsung heroes who have made all these possible!

Parallelism is not restricted just to multiple processors running on a chip, it can happen at multiple levels:

- Between Instructions/Data on a single core
- Processors on a single chip
- Machines/Cluster of machines on a network
- Custom hardware

There can also be hybrid solutions that can mix and match various architectures mentioned above.

In the following section, we'll see a simple case of sorting done across various paradigms.

Parallel Sorting Algorithms

Any sorting problem, can be represented as:

$$X = \{ x_1, x_2, \dots, x_n \}$$

set of n real numbers stored in an array of:

$$X[] = X[0], X[1], \dots, X[n-1]$$

For n elements, there are $n!$ ways that they can be arranged. The goal is to find 1 permutation out of the $n!$ that abides to our desired order.

Below is the time complexity(Big O) of some of the conventional sequential sorting algorithms.

Name	Complexity(O(..))	Comments
Bubblesort	n^2	Propagates the largest element towards the end of the array in each iteration
Quicksort	$n \log_2(n)$	Recursive algorithm that sorts across a randomly chosen pivot
Mergesort	$n \log_2(n)$	Divide and conquer algorithm that divides the array into smallest terminal units, and merges them back while sorting

In the following sections we introduce various parallel sorting algorithms which are either modifications on the classical ones or unique to parallel paradigm. We theoretically, evaluate the performance gains for each. Some of the assumptions that we make:

- For P processes, all data are already allocated and can be denoted by
- All data are to be sorted in ascending order
- The number of processes P , is in powers of 2.

It's important to note that the best possible for a serial algorithm is:

$$O(n \log_2(n))$$

for a parallelized algorithm it's:

$$O(\log_2(n))$$

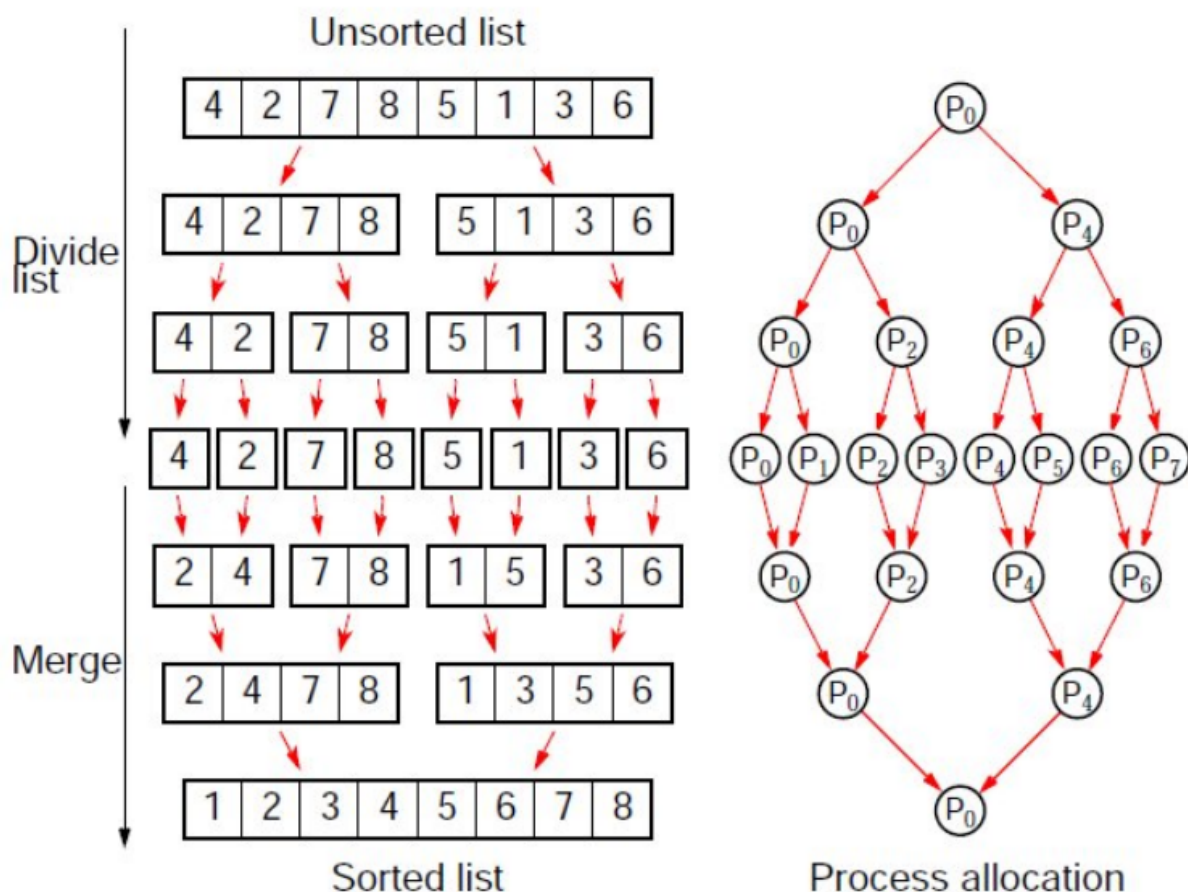
Thanks to parallelization we can reduce the complexity by a factor of n . This holds true no matter how much units of n we use.

In fact, practically, in our lab sessions have seen that we need to have proper balance between n and P . If we put too many processes, then much of time will be lost in synchronization and co-ordination of various processes, in worst cases we are better off using a serial algorithm. If we use too less, then we are not exploiting the full potential of available resources.

Hence, it's all about finding that "sweet" spot.

Parallel Merge Sort

This is quite similar to the sequential merge sort. The tree structure of the algorithm, makes the distribution of the work between processes fairly simple. This is illustrated in the figure below ^[1]:



Complexity analysis

For any merge sort, in the worst case:

- Number of steps to sort a sublist

$$2s - 1$$

where s is the size of the sublist:

$$s = 2^{i-1}$$

where i is the i^{th} iteration.

- At any given step, number of sub-lists:

$$(n/s) = (n/2^{i-1})$$

where n is the total number of elements

- Number of steps in total:

$$\log_2(n)$$

In serial merge sort the time complexity for i^{th} step is:

$$O\left(\left(\frac{n}{2^i}\right)(2^i - 1)\right) = O(n)$$

Whereas in parallel merge sort the time complexity for i^{th} step is:

$$O(2^i - 1) = O(2^i)$$

We can clearly see that without parallelization, the time remains constant but proportional to size of initial array. But, with parallelization, we have reduced the time, now it's in proportion to the subarray size. This is because, at any given step all the $\left(\frac{n}{2s}\right)$ sub-lists are processed in parallel. Hence, complexity of Parallel merge sort:

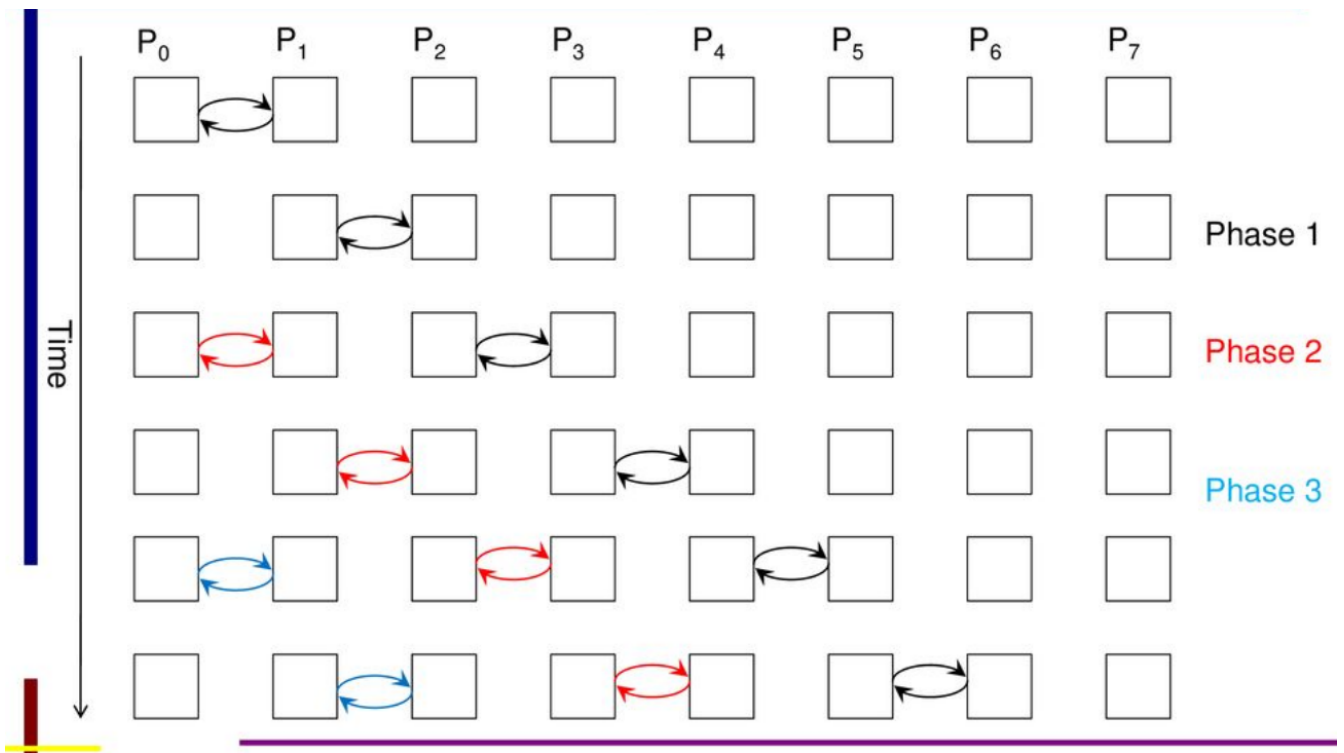
$$O\left(\sum_{i=1}^{\log_2(n)} (2^i)\right) = O(n)$$

Therefore, we see a huge gain, wherein a relatively complex operation such as sorting is reduced to time complexity of a simpler operation such as, for example, say linear search.

Parallel Bubble Sort

A classic serial bubble sort sorts the array by iteratively moving elements to their respective positions in relation to the end of the array in each phase.

In parallel bubble sort the idea is to overlap multiple phases in such a way that the newly started phase doesn't disturb an on-going phase. Considering the sequence of bubble sort, to achieve this, we need to pipeline the sort in the following manner ^[2]:



Complexity analysis

In serial bubble sort, i^{th} phase requires $n - i$ iterations and we'll have $n - 1$ such phases. Hence, we have time complexity:

$$O\left(\sum_{i=1}^{n-1} n - i\right) = O\left(\frac{n(n-1)}{2}\right) = O\left(n^2\right)$$

Now, thanks to parallelization, though each phase will take same time as in serial version, the end time will vary due to pipelining. Consequently, i^{th} phase will end at time:

$$n - 1 + i$$

Hence, the end time of the algorithm and hence complexity is:

$$O\left(n - 1 + n - 1\right) = O\left(2n - 2\right) = O\left(n\right)$$

Hence, we have reduce complexity from n^2 to n .

Parallel Quick Sort

Parallel Sort Regular Sampling

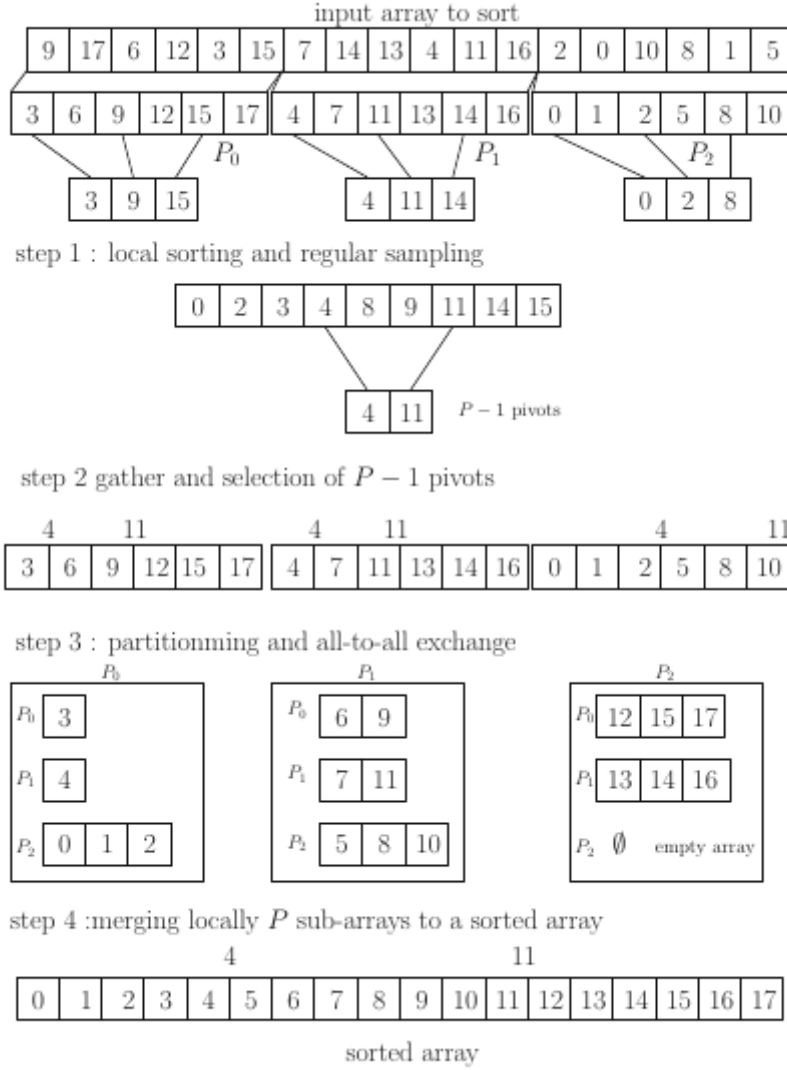
In this algorithm we can relax our last initial assumption. Now P need not be in powers of 2, it can be any arbitrary natural number. The algorithm is as follows:

- Sort local array using any algorithm. If this algorithm is run on distributed network using let's say MPI, by using framework like OpenMP for local sorting higher throughput can be achieved.
- Sample P elements at intervals:

$$0, \frac{n}{P^2}, \frac{2n}{P^2}, \dots, \frac{(P-1)n}{P^2}$$

- Gather all these samples in another process and $P-1$ pivots and broadcast them to all the processes
- All processes partition their local array into P pieces based on these pivots.
- Now each process P_i retains the i^{th} partition and sends its j^{th} partition to process $P_i \forall j \neq i$

To illustrate with an example ^[3]:



In short, the intuition is: Sort locally and strategically share with each other parts of locally sorted array using some reference pivots selected among sorted samples.

Complexity Analysis

- Cost of local computations
 - Local sort (assuming some efficient sequential algorithm was used):

$$O\left(\frac{n}{P} \log_2\left(\frac{n}{P}\right)\right)$$
 - Sorting regular samples in intermediate step:

$$O\left(P^2 \log_2(P^2)\right) = O\left(P^2 \log_2(P)\right)$$

- Merging sublists:

$$O\left(\frac{n}{P} \log_2(P)\right)$$

- Communication cost

- Gathering sample and broadcasting pivots:

$$\approx O(1)$$

- Total exchange in the last step:

$$O\left(\frac{n}{P}\right)$$

If we summate all the costs:

$$O\left(P^2 \log_2(P) + \frac{n}{P} \log_2(P) + \frac{n}{P} \log_2(P) + 1 + \frac{n}{P}\right) = O\left(\frac{n}{P}\left(1 + \frac{1}{P^2} \log_2(nP)\right)\right) = O\left(\frac{n}{P}\right)$$

Now, we have effectively reduced the time-complexity to $t = \frac{n}{P}$ where $\log_2(n) < t < n$.

Core level

SIMD MIMD

Different processor instructions takes different amounts of CPU clock cycles and hence different costs based on the data types that are being manipulated. So, there is scope for optimization of the number of operations. For example, reducing number of floating multiplications by factorization. Nowadays, all of these are directly performed within the compiler. Moreover, if same operation needs to be applied on bunch of data whose results are independent, we can fetch them together and vectorize them on a larger registers and apply the same fetched instruction in parallel on all of them. Hence, SIMD reduces the number of fetches in the pipeline and brings performance benefits.

Chip level

Declarative: OpenMP Imperative: `std::thread`, `pthread`s Library based: Intel TBB/MS PPL

Machine level

MPI

Custom hardware

GPGPU(General Purpose computation on GPU), Hardware accelerators(Intel Xeon Phi), ASIC/FPGA architectures

Performance evaluation

Parallel Sorting: 99 Topology of Interconnection of Networks: 63 Parallel Linear Algebra: 121

[1] source: https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf

[2] source: Clayton S Ferner, UNC Wilmington, Barry Wilkinson, UNC Charlotte

[3] source: Introduction to HPC with MPI for Data Science by Frank Nielsen