Quicksort

The serial implementation of quicksort algorithm involves *two steps*:

1. *Rearrange* the elements in the list by comparing with the chosen pivot element, the list is *partitioned* into **two** such that the elements less than or equal to the pivot element are part of the *first list* and the elements greater than the pivot element form the *second list*
2. *Recursively sort* these lists using quicksort to obtain the sorted list

$\leq \qquad pv$	$>$

Note:

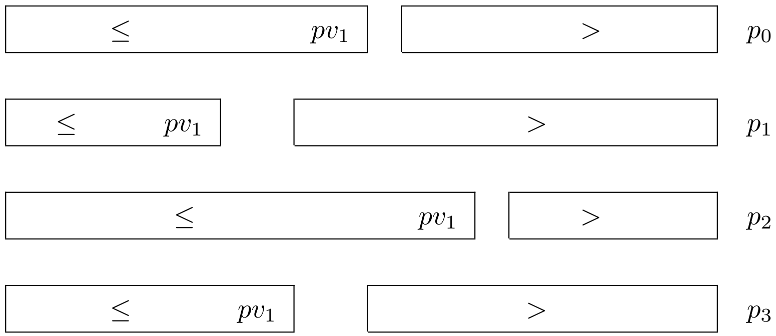
1. The choice of the pivot element can be the first element or the median but this would not affect the sorting procedure
2. It is necessary to ensure that the list is not empty in the recursive implementation of quick sort

The **parallel implementation** would be implemented using the partition procedure, step(1) of the serial implementation as **driver** on *different processes*.

Parallel quicksort

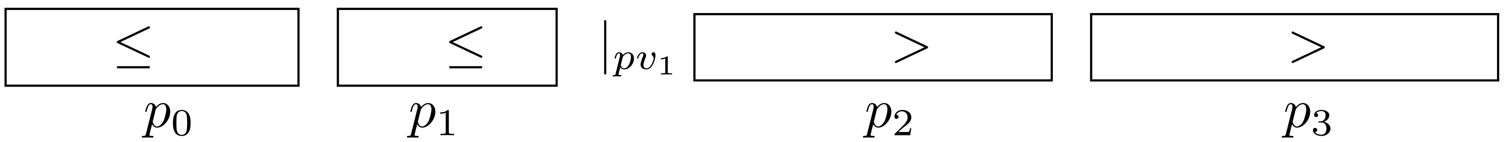
To begin with the list of numbers are *distribute equally* across **four** processes. The implementation primarily involves two stages:

1. Choose a pivot element (say pv_1) from process p_0 and use this to **partition the individual lists** on each of the processes p_0, p_1, p_2, and p_3.



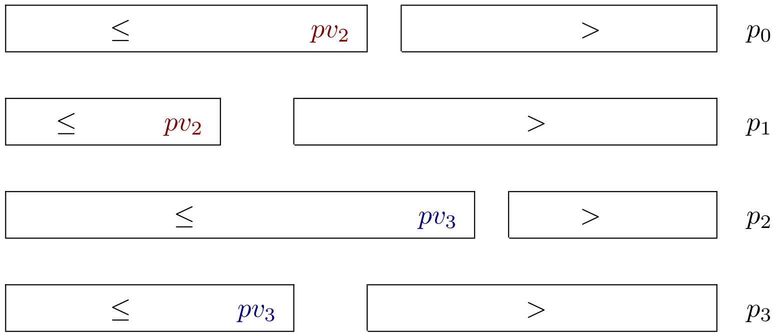
   a. Collect the $\leq$ list on proc. p_2 to proc. p_0 and $\leq$ list on proc. p_3 on proc. p_1
   b. Collect the $>$ list on proc. p_0 to proc. p_2 and $>$ list on proc. p_1 on proc. p_3

Now, the numbers on processess p_0 and p_1 are less than or equal to pv_1 and the ones on processess p_2 and p_3 are greater than pv_1. [$\{n_{p_0}\} \,\&\, \{n_{p_1}\} \leq pv_1 < \{n_{p_2}\} \,\&\, \{n_{p_3}\}$]



Parallel quicksort contd.

2. Choose **two** pivot elements, say pv_2 from proc. p_0 to be used for partitioning on processes 0 & 1 and pv_3 from proc. p_2 to be used for partitioning on processes 2 & 3.



   a. Collect the $\leq$ list on proc. p_1 to proc. p_0 and $\leq$ list on proc. p_3 on proc. p_2

   b. Collect the $>$ list on proc. p_0 to proc. p_1 and $>$ list on proc. p_2 on proc. p_3

Now, the numbers are such that $\{n_{p_0}\} \leq pv_2 < \{n_{p_1}\} \leq pv_1 < \{n_{p_2}\} \leq pv_3 < \{n_{p_3}\}$

$$\underbrace{\boxed{\leq pv_2}}_{p_0} \;\Big|_{pv_2} \underbrace{\boxed{pv_2 < n \leq pv_1}}_{p_1} \;\Big|_{pv_1} \underbrace{\boxed{pv_1 < n \leq pv_3}}_{p_2} \;\Big|_{pv_3} \underbrace{\boxed{> pv_3}}_{p_3}$$

Note: The list on individual process is still not sorted. The final sorted list is obtained by concatenating the sorted list from different process (in a particular order).

Parallel quicksort: Implementation pointers

1. There are **four transactions** while collecting the data across pairs of processess in **both the stages**. There should be a pair of **MPI_Send** and **MPI_Recv** calls for each of them.
2. **Stage 1**: Data exchange between processes $0 \Leftrightarrow 2$ & processes $1 \Leftrightarrow 3$
3. **Stage 2**: Data exchange between processes $0 \Leftrightarrow 1$ & processes $2 \Leftrightarrow 3$
4. The size of the arrays to be exchanged is not known, so explicitly needed to be communicated among the processes involved in data exchange.
5. **MPI_Gatherv** is used to collect the sorted list (of different sizes) from different processes.
6. The arguments of **MPI_Gatherv**, namely **sizes of data to be reveived (recvcount)** from different processes and **displacement** (to be used while concatenating the list) must be available on each of the process.
7. The size of the arrays, **recvcount** and **displacement** is equal to the number of processess in the communicator.

Note: This implementation **does not maintain load balance** among different processes, since the number of elements to be sorted changes based on the choice of the pivot as well as the order of the numbers in the list.