FORTRAN and MPI

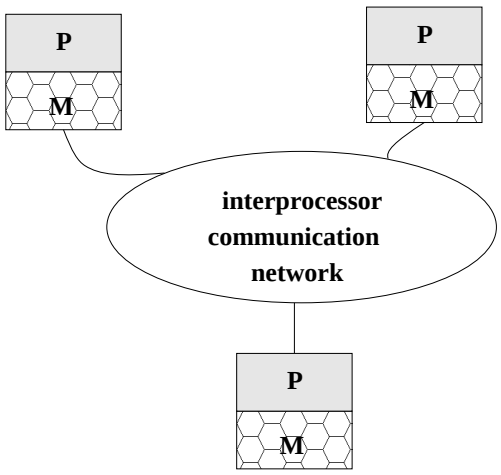
Message Passing Interface (MPI)

Day 4

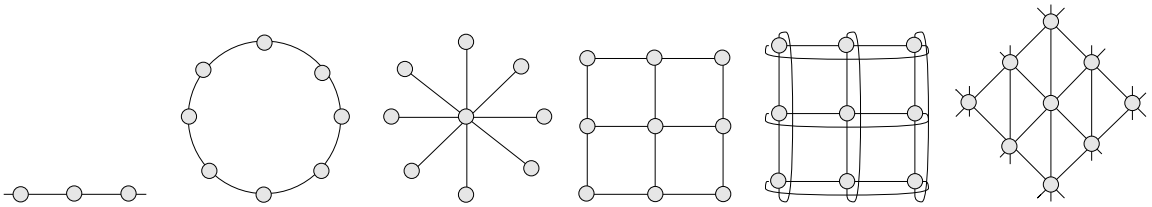
Course plan:

- MPI - General concepts
- Communications in MPI
  – Point-to-point communications
  – Collective communications
- Parallel debugging
- Advanced MPI: user-defined data types, functions
  – Linear Algebra operations
- Advanced MPI: communicators, virtual topologies
  – Parallel sort algorithms
- Parallel performance. Summary. Tendencies

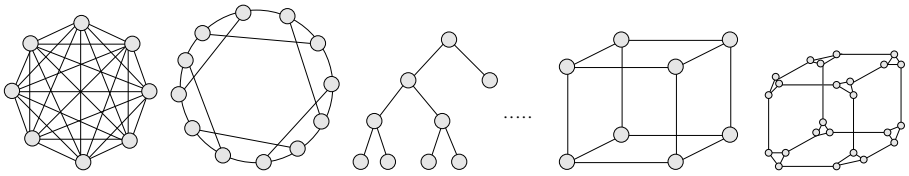
Recall: Distributed memory machines



Recall: Interconnection network topologies



(a) linear array (b) ring (c) star (d) 2D mesh (e) 2D toroidal mesh (f) systolic array



(g) completely connected (h) chordal ring (i) binary tree (j) 3D cube (k) 3D cube

Communicators

Communicators

Creating communicators enables us to split the processors into groups.

Benefits:

- The different groups can perform independent tasks.
- Collective operations can be done on a subset of processors
- The subsets are logically the same as the initial (complete) group of PEs, i.e., there exists P0 in each subgroup. In this way, for example, recursive algorithms can be implemented.
- Safety - isolating messages, avoiding conflicts between modules etc.

First we need to introduce the notion of `group`
A group is an ordered set of process identifiers (henceforth processes).

Each process in a group is associated with an integer rank. Ranks are contiguous and start from zero.

Groups cannot be directly transferred from one process to another.

A group is used within a communicator to describe the participants in a communication "universe" and to rank such participants (thus giving them unique names within that "universe" of communication).

There is a special pre-defined group: `MPI_GROUP_EMPTY`, which is a group with no members. The predefined constant `MPI_GROUP_NULL` is the value used for invalid group handles.


```
MPI_COMM_GROUP(comm,group)

input: comm - communicator
output: group - group corresponding to the communicator

MPI_COMM_CREATE(COMM, GROUP, IERROR)

INTEGER COMM, GROUP, NEWCOMM, IERROR
```

Groups

Various functions available to manipulate with communicators in MPI:

MPI_GROUP_UNION(group1, group2, newgroup)

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

MPI_GROUP_INCL(group, n, ranks, newgroup)

MPI_GROUP_EXCL(group, n, ranks, newgroup)

MPI_GROUP_FREE(group)

Communicators

```
 MPI_COMM_CREATE(comm, group, newcomm)
[ IN comm] communicator (handle)
[ IN group] Group, which is a subset of the group of comm (handle)
[ OUT newcomm] new communicator (handle)
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
INTEGER COMM, GROUP, NEWCOMM, IERROR
```

The function creates a new communicator.

Various functions available to manipulate with communicators in MPI:

MPI_COMM_SIZE(comm, size)

MPI_COMM_COMPARE(comm1, comm2, result)

MPI_COMM_SPLIT(comm, color, key, newcomm)

MPI_COMM_FREE(comm)


```
 MPI_COMM_SPLIT(comm, color, key, newcomm)
 IN   comm   communicator (handle)
 IN   color  control of subset assignment (integer)
 IN   key    control of rank assigment (integer)
 OUT newcomm new communicator (handle)

int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm *newcomm)

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

What are those: `color` and `key`?

Partitions the group associated with comm into disjoint subgroups, one for each value of color.
Each subgroup contains all processes of the same color.
Within each subgroup, the processes are ranked in the order defined by the value of the argument key.
A new communicator is created for each subgroup and returned in `newcomm`.

A process may supply the color value `MPI_UNDEFINED`, in which case newcomm returns `MPI_COMM_NULL`. This is a collective call, but each process is permitted to provide different values for color and key.


```
program Comm
implicit none
include "mpif.h"

integer ierror, rank, size, rankh, sizeh, key
integer ALL_GROUP, color, HALF_COMM
integer N, M
integer, dimension(MPI_STATUS_SIZE) :: status

call MPI_Init(ierror)

call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)

N = rank
M = rank+10
call MPI_COMM_GROUP(MPI_COMM_WORLD, ALL_GROUP, ierror)
```

```
color=rank/2
key=0

call MPI_COMM_SPLIT(MPI_COMM_WORLD,color,key,HALF_COMM, ierror)

call MPI_Comm_rank(HALF_COMM, rankh, ierror)
call MPI_Comm_size(HALF_COMM, sizeh, ierror)
write(*,*) 'Global ', rank,' is now local rank ',rankh,' ',color
write(*,*) 'Global ',rank,' local ',rankh, ' has ',N,' and ',M
```



```
if (rankh.eq.0) then
call MPI_SENDRECV(N,1,MPI_INTEGER,1,1,
.                 N,1,MPI_INTEGER,1,2,HALF_COMM,status,ierror)
else
call MPI_SENDRECV(N,1,MPI_INTEGER,0,2,
.                 N,1,MPI_INTEGER,0,1,HALF_COMM,status,ierror)
endif

write(*,*) 'Global ',rank,' local ',rankh, ' new ',N,' and ',M

call MPI_COMM_FREE(HALF_COMM,  ierror)
call MPI_GROUP_FREE(ALL_GROUP, ierror)

call MPI_Finalize(ierror)

stop
end
```

Possible output of the above code (sorted afterwards)

```
 Global   3 local   1 has   3 and   13
 Global   2 local   0 has   2 and   12
----------------------------------------
 Global   1 local   1 new   0 and   11
 Global   0 local   0 new   0 and   11

 Global   2 local   0 new   2 and   13
 Global   3 local   1 new   2 and   13
----------------------------------------
----------------------------------------
 Global   2 local   0 has   2 and   12
 Global   3 local   1 has   3 and   13
----------------------------------------
 Global   0 local   0 new   1 and   10
 Global   1 local   1 new   0 and   11

 Global   2 local   0 new   3 and   12
 Global   3 local   1 new   2 and   13
```


Virtual interconnection topology

Algorithm $\longrightarrow$ communication pattern $\longrightarrow$ graph

The processes represent the nodes in that graph,
the edges connect processes that communicate with each other.
MPI provides message-passing between any pair of processes in a group.

However, it turns out to be convenient to describe the virtual communication topology utilized by an algorithm.

The provided MPI functions for that are:

`MPI_GRAPH_CREATE` and `MPI_CART_CREATE`

which are used to create general (graph) virtual topologies and Cartesian topologies, respectively. These topology creation functions are collective.

The tool provided to describe Cartesian grids of processors is

`MPI_CART_CREATE`

Cartesian structures of arbitrary dimension are allowed.
In addition, for each coordinate direction one specifies whether the process structure is periodic or not.

Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary.

Cartesian topology functions

MPI_CART_CREATE(comm_old,ndims,dims,periods,reorder,comm_cart)

IN	comm_old	input communicator
IN	ndims	number of dimensions in a Cartesian grid
IN	dims	integer array of size `ndims` specifying the number of procs in each dimension
IN	periods	logical array of size `ndims` specifying whether the grid is periodic ('true') or not ('false') in each dimension
IN	reorder	ranks may be reordered ('true') or not ('false')
OUT	comm_cars	communicator with new Cartesian topology

Returns a handle to a new communicator. If `reorder='false'`, the rank of each processor in the group is identical to its rank in the new group.

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

```
...
dims(1) = 4
dims(2) = 4
periods(1) = 'true'
periods(2) = 'true'
reprder    = 'false'
MPI_CART_CREATE(MPI_COMM_WORLD,2,dims,
.                  periods,reorder,GRID_COMM)
```

MPI_CART_SHIFT(comm,direction,disp,rank_src,rank_dest)

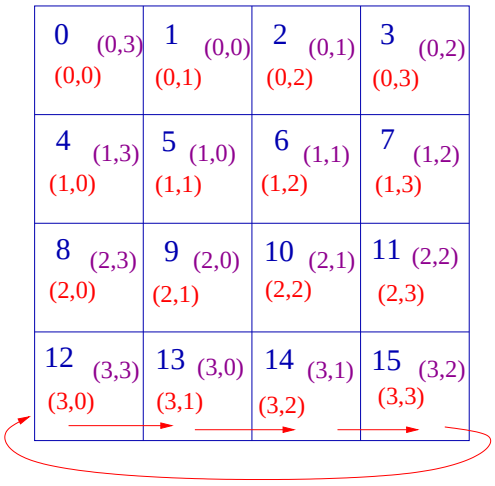
IN	comm	communicator with Cartesian structure
IN	direction	coordinate dimension of shift
IN	disp	displacement (> 0: upwards shift, < 0: downwards shift)
OUT	rank_src	rank of source process
OUT	rank_dest	rank of destination process

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                   int *rank_source, int *rank_dest)

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)

INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
```


Virtual topologies




```
MPI_CART_SHIFT(GRID_COMM,1,1,src,dest,ierror)
```

```
....
C find process rank
      CALL MPI_COMM_RANK(comm, rank, ierr))
C find cartesian coordinates
      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
      CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
      CALL MPI_SENDRECV_REPLACE(A,1,MPI_REAL,dest,0,source,0,comm,
     +                          status, ierr)
```

OBS!:

In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1) nodes,

where DIMS is the array that was used to create the grid.

In C, the dimension indicated by direction = i is the dimension specified by dims[i].

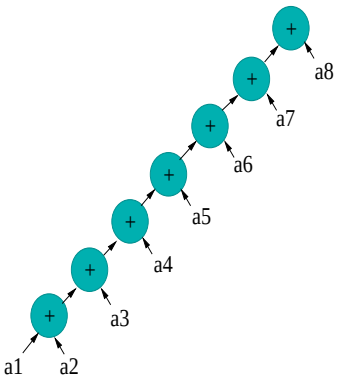
Parallel *Divide-and-Conquer* techniques

- Partitionings:
  - data partitioning
  - functional partitioning
- Examples of Divide-and-Conquer approaches:
  - $\sum_{i=1}^{n} a_i$, $\sum_{i=1}^{n} a_i b_i$
  - Numerical integration
  - Solution of tridiagonal systems
  - Bucket sort algorithm

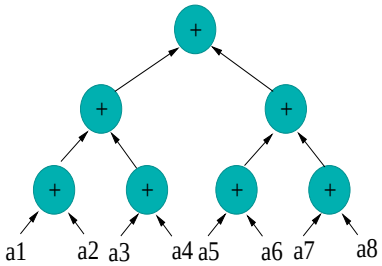
More examples of Divide-and-Conquer algorithms:

matrix manipulations	block splitting of matrices, multifrontal methods
the 'knapsack' problem	the problem to find a set of items each with a weight w and a value v, in order to maximize the total value while not exceeding a fixed weight limit
'Mergehull'	an algorithm for determining the convex hull of n points in a plane.
integer factorization	an algorithm for decomposing positive integer numbers into prime factors
set-covering	the problem is to find a minimal set of subsets of a set$S$, which covers the properties of all elements of the set S.

((((((a1+a2)+a3)+a4)+a5)+a6)+a7)+a8

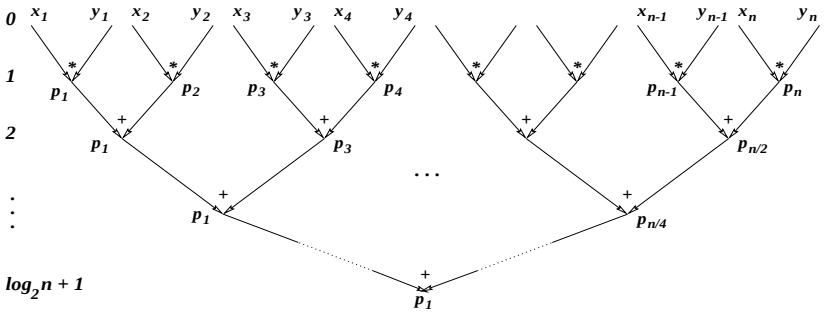


(a1+a2)+(a3+a4))+((a5+a6)+(a7+a8))



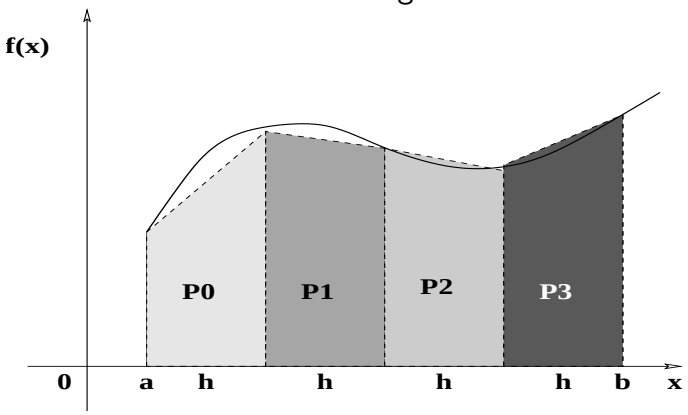
A trivial example how to gain parallelism

Scalar product, summing-up n numbers
FAN-IN, FAN-OUT algorithms



Binary tree algorithm represented by a cascade graph to compute $\sum_{i=1}^{n} x_i y_i$, where $n = 2^s (s = 3)$

Numerical integration



Proc$_0$: broadcast a and b
Proc$_i$: $h = (b - a)/p$
$a_i = a + (i - 1) * h$, $b_I = a + Ih$
$area\_local = F(a_i) + f(b_i) * h * 0.5$
Proc$_0$: gather $area\_local$ into $area\_local_i$
$area = \sum_{i=0}^{p-1} area\_local_i$

Parallel Sort Algorithms

Parallel Sort Algorithms

A general-purpose parallel sorting algorithm must be able to sort a large sequence with a relatively small number of processors.

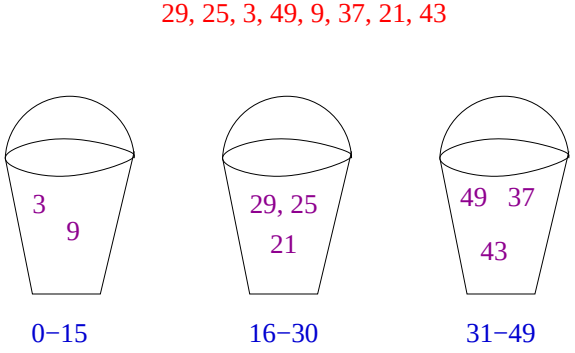
Let p be the number of processors and n be the number of elements to be sorted. Each processor is assigned a block of size n/p elements. Let $A_0, A_1, \cdots, A_{p-1}$ be the blocks assigned to processors $P_0, P_1, \cdots, P_{p-1}$, respectively. We say that $A_i < A_j$ if every element of A_i is smaller than every element in A_j. When the sorting finishes, each processor P_i holds a set $A_i^{'}$ such that for $i \leq j$ and $\bigcup_{i=0}^{p-1} A_i = \bigcup_{i=0}^{p-1} A_i^{'}$.

In a typical sorting process, each processor sorts locally the block it owns, then selects a pivot, split the block into two according to the pivot, exchange half of the block with its neighbors, merge the received block with the block it retained.

- Sequential sort
  - Selection sort, Insertion sort, Bubble sort, each has a complexity of $O(n^2)$, where n is the number of elements
  - Quick sort, Merge sort, Heap sort
    - O(n log n)
    - Quick sort best on the average
- Different approach to design a parallel sort
  - Use a sequential sort and adapt
    - How well can it be done in parallel? Not all sequential algorithms can be parallelized easily.
    - Sometimes a poor sequential algorithm can develop into a reasonable parallel algorithm (e.g. bubble sort).
  - Develop a different approach
    - More difficult, but may lead to better solutions.

Name	Description	Complexity	Modifications
bubble	sort by comparing each adjacent pair of items in a list in turn, swapping the items if necessary, and repeating the pass through the list until no swaps are done	$O(n^2)$	bidirectional bubble sort, exchange sort, sink sort
insertion	Sort by repeatedly taking the next item and inserting it into the final data structure in its proper order with respect to items already inserted. Run time is $O(n^2)$ because of moves.	$O(n^2)$	binary insertion sort

Name	Description	Complexity	Modifications
bucket	A distribution sort where input elements are initially distributed to several (but relatively few) buckets based on a certain predefined *value-brakets*. Each bucket is sorted if necessary, and the buckets' contents are concatenated.	$O(n \log \log n)$	bin sort, range sort



Name	Description	Complexity	Modifications
quicksort	One element, x of the list to be sorted is chosen and the other elements are split into those elements less than x and those greater than or equal to x. These two lists are then sorted recursively using the same algorithm until there is only one element in each list, at which point the sublists are recursively recombined in order yielding the sorted list.	$O(n \log n)$	balanced, external, hybrid
merge sort	A sort algorithm which splits the items to be sorted into two groups, recursively sorts each group, and merges them into a final, sorted sequence	$O(n \log n)$	(non)balanced, balanced two-way, k-way

Name	Description	Complexity	Modifications
heap sort	A sort algorithm which builds a heap, then repeatedly extracts the maximum item. Heap data structure: A tree where every node has a key more extreme (greater or less) than the key of its parent.	$O(n \log n)$	weak heep sort, adaptive heap sort
radix sort	A multiple pass sort algorithm that distributes each item to a bucket according to part of the item's key beginning with the least significant part of the key. After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant part of the key (c depends on the size of the key and number of buckets.	$O(cn)$	bottom-up radix sort, radix quicksort

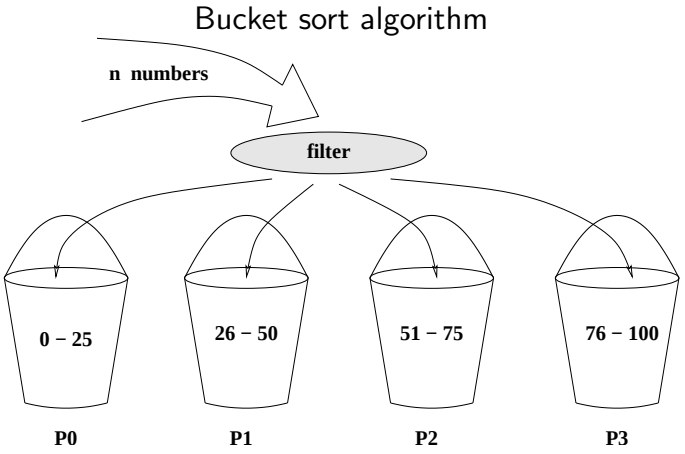
Name	Description
diminishing increment sort	**Counting Sort**
shell sort	

Parallel algorithms on sequences and strings:

Matching Parentheses: This is an interesting algorithm since one might think that matching parentheses seems very sequential. For each location the algorithm returns the index of the matching parenthesis. The algorithm is based on a scan and an integer sort (rank). The scan returns the depth of each parenthesis and the sort groups them into groups of equal depth. At this point we can simply switch the indices of neighbors. Assuming a work-efficient radix sort, this algorithm does O(n) work and has the depth is bounded by the sort.

```
function parentheses_match(string) =
let
    depth = plus_scan({if c==‘( then 1 else -1 : c in string});
    depth = {d + (if c==‘( then 1 else 0): c in string; d in depth};
    rnk = permute([0:#string], rank(depth));
    ret = interleave(odd_elts(rnk), even_elts(rnk))
in permute(ret, rnk);

parentheses_match("()(()())((()))");
```

Quick sort in each bucket - serial complexity to sort k numbers: $O(k \log k)$

$T_1 = n + p\frac{n}{p}\log(\frac{n}{p}) = n(1 + \log(\frac{n}{p})) = O(n)$

$T_p = n + \frac{n}{p}\log(\frac{n}{p})$

Divide-and-Conquer approaches for sorting algorithms:

- Merge sort
  - collects sorted list onto one processor, merging as items come together
  - maps well to tree structure, sorting locally on leaves, then merging up the tree
  - as items approach root of tree, processors drop out of the process, limiting parallelism
- Quick sort
  - maps well to hypercube
  - divide list across dimensions of the hypercube, then sort locally
  - selection of partition values is even more critical than for sequential version since it affects load balancing
  - hypercube version leaves different numbers of items on different nodes

- Basic idea of parallel quick sort algorithm on a hypercube
  1: Select global partition value (pivot), split values across highest cube dimension, high values going to upper side and low values to lower side
  2: Repeat on each of the lower dimensional cubes forming the upper and lower halves of the original (divide)
  3: Continue this process until the remaining cube is a single processor, then sort locally
  4: Each node contains a sorted list, and the lists from node to node are in order, using *Grey* code numbering of nodes.

- How to implement it? Hyper quick sort;
  - I. Divide data equally among nodes
  - II. Sort locally on each node first
  - III. Broadcast median value from node 0 as pivot
  - IV. Each list splits locally, then trades halves across highest dimension
  - V. Apply the above two steps successively (and in parallel) to lower dimensional cube forming the two halves, and so on until dimension reaches 0 (a single node)

- Run time for quicksort on hypercube: following components contribute to run time.
  1. Local sort (e.g. sequential quicksort) in $O(\frac{n}{p}\log\frac{n}{p})$
  2. Broadcasting a pivot during i-th iteration takes $O(d-(i-1))$ where $d-(i-1)$ is the dimension of the sub-hypercube.
     In a d-dimension hypercube, one-to-all broadcast can be done in d steps. Thus the total time spent in broadcasting pivots (it needs $d = \log p$ iterations)

$$\sum_{i}^{d} i = \frac{d(d+1)}{2} = \frac{(\log p)(\log p + 1)}{2} = O(\log^2 p)$$

  3. Partitioning n/p elements in $O(\frac{n}{p}\log\frac{n}{p})$, but we need to do it $\log p$ times
  4. Exchange local list with neighbors in $O(\frac{n}{p}\log\frac{n}{p})$, but we need to do it $\log p$ times.
  5. Merge local list with the one received in $O(\frac{n}{p}\log\frac{n}{p})$, but we need to do it $\log p$ times

So the total run time is

$$T_p = O\left(\frac{n}{p}\log\frac{n}{p}\right) + O\left(\frac{n}{p}\log p\right) + O(\log^2 p).$$

If the number of processors p is equal to $n/\log n$,
then its best run time is

$$O(\log n * \log(\log n)) + O(\log^2 n) = O(\log^2 n).$$