Parallel Sorting

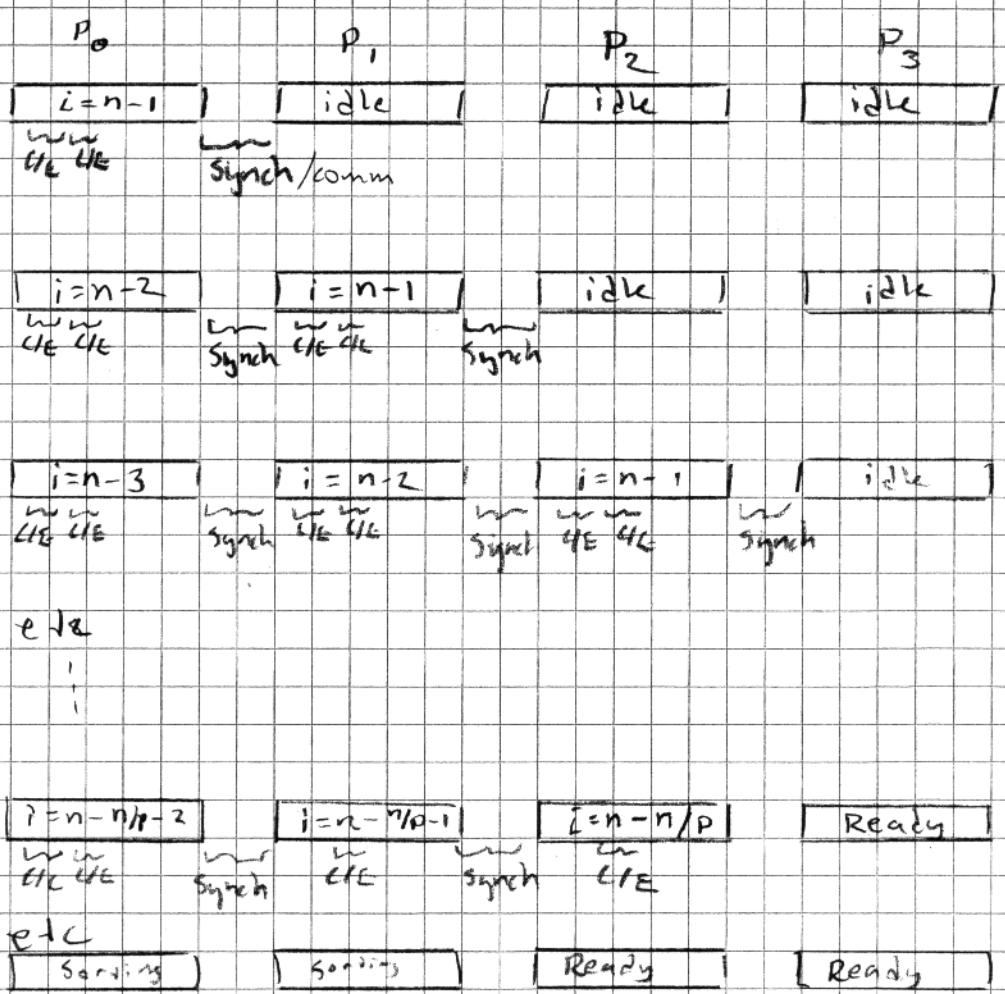
Bubbel Sort:

```
for i = n-1 downto 1
    for j = 1 to i
        compare-exchange (a_j, a_{j+1})      a_{j+1} = max(a_j, a_{j+1})
    end for
end for
```

In j-loop, move the largest element to the rightmost position (i+1) } Repeat: loop

Both loops "ordered", use pipelining

Do several outer iterations in parallel

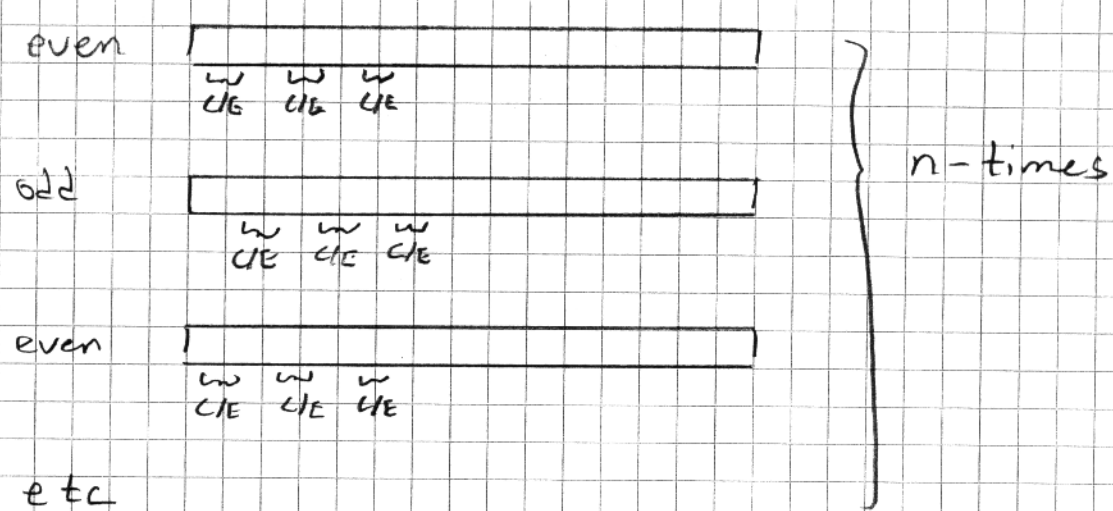


Problems:

- Load imbalance [Can improve with smaller partitions but increase synch/comm]
  (start-up time, last processors ready first)
- Communication/Synchronization
  (In each iteration, n-times)

Change algorithm: (odd-even sort)

Introduce two phases, odd and even.
In odd, C/E only elements with odd indices with their right neighbor.
In even, C/E only elements with even indices with their right neighor.



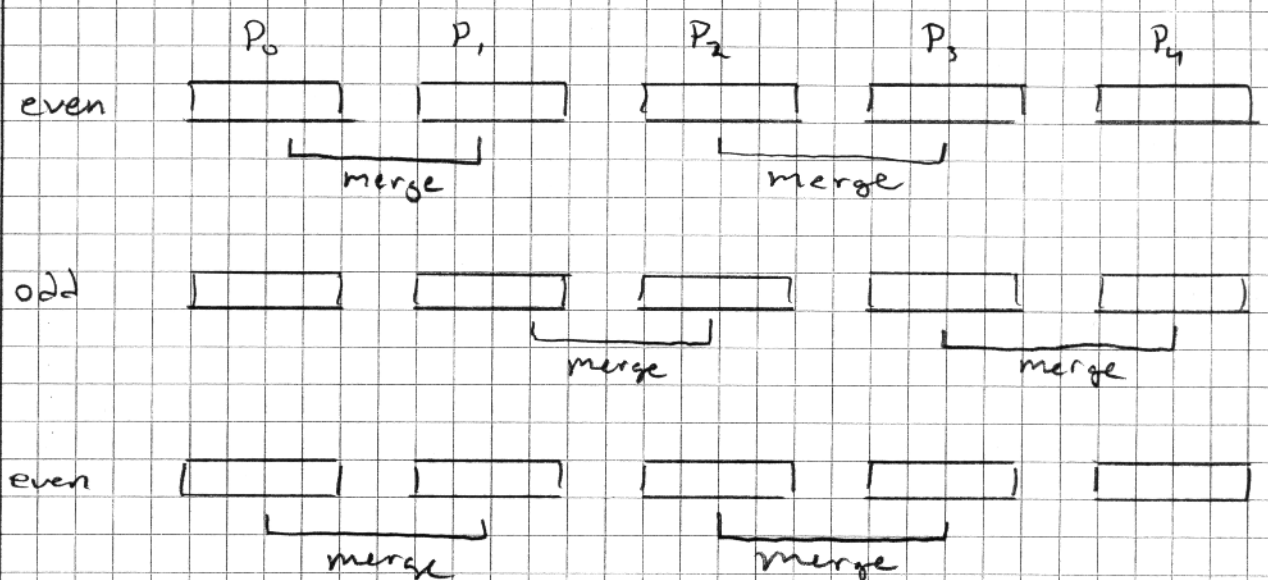
⇒ Work within each phase is perfectly parallel!

[No load imbalance, comm/synch in every (second) iter]

Parallel Odd-even Sort

1] Divide data equally and sort locally (in parallel) with the fastest serial algorithm

2] Introduce odd and even phases.
In even, processors with even id gives its data to next processor (on right)
This processors merge the sorted sequences into one sequence and gives back half of the data
In odd, repeat with processors with odd indices.



etc. P-steps. [or abort when no changes]

Problems:

- Load imbalance
  (Half of processors working in each merging step)
- Unnecessary communication
  (Data moved back and forth)
- Slow (long time to move data from one side to the other)

(*)

Quicksort

Sequential algorithm: (Recursive)

1) Select pivot (any element)
2) Divide data into two list according to the pivot element (smaller/larger)
3) Sort the lists independently with Quicksort (Perfectly parallel tasks)

Parallel implemention (naive):

Start with one processor and all data, In each split employ a new processor for the other part. After $\log_2 P$ steps sort locally within the processors.

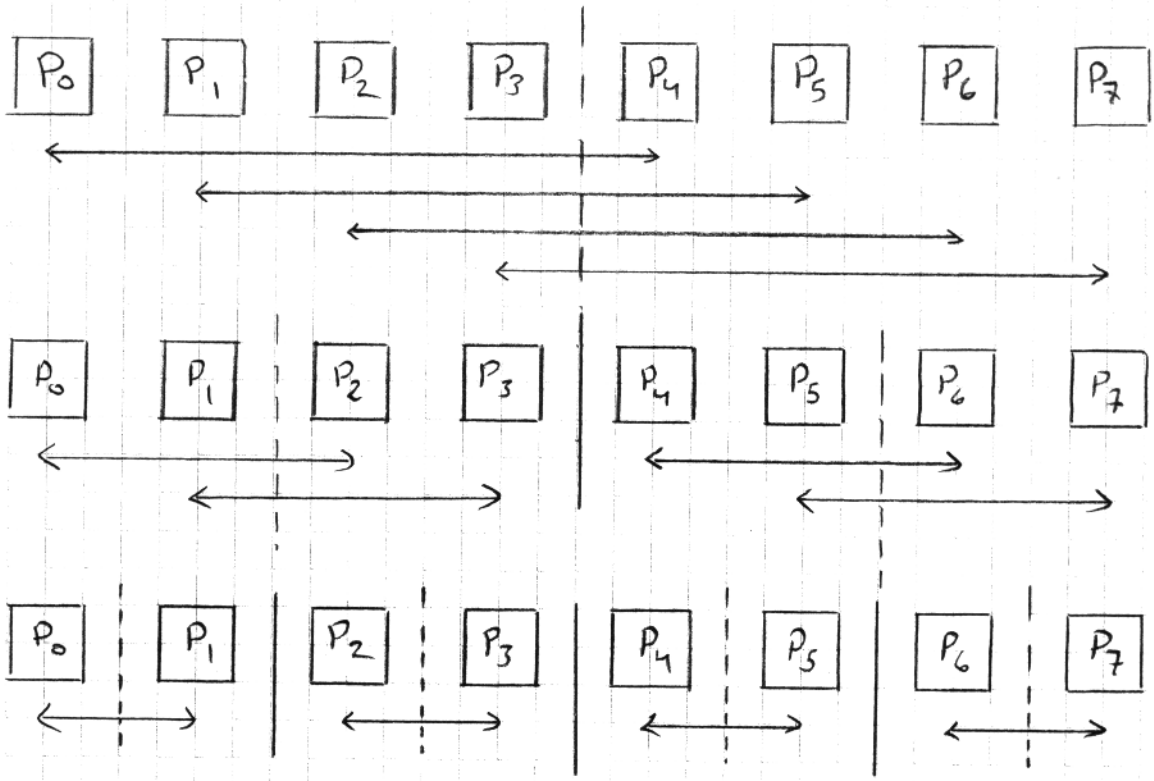
[Compare pivot strategy 0 on lab, worst possible!]

Parallel Quicksort

1) Divide data equally and sort locally (parallel)
2) Select pivot (median) and broadcast within processor set
3) In each processor divide data according to pivot
4) Divide the processors into two sets and exchange data pairwise between processors in the two sets such that the processors in one set gets data smaller than pivot and the other set gets data larger than pivot
5) In merge data in each processor, keeping data sorted

Repeat 2-5 recursively $\log_2 P$ steps.

Illustration P=8



Problems:

(- Complex algorithm, there is support in MPI)

- An unfortunate pivot selection will give a bad load balance between the two processor sets (in the remainder of the algorithm)

Pivot selection

Strategy 1: Select the median in one processor in each processor set.

Ex) Step 1 - select median in P_0

Step 2 " P_0, P_4

Step 3 " P_0, P_2, P_4, P_6

(Very bad if data almost sorted)

Strategy 2: Select the median of all medians in respective processor set.

(Bad if all medians bad, either high or low)

AH: Take mean of the middlemost medians, strategy 3 in Lab

Strategy 3: Select the mean value of all medians in respective processor set.

(Can be bad if data is not uniform, eg. many small numbers but only few large, extreme medians get too much weight.)

Strategy 4: Select all pivots at once

$$\left(\sum_{i=1}^{\log_2 P} 2^{i-1} = P-1 \text{ pivots} \right)$$

a) Each processor selects l evenly distributed elements within its data.

In one processor:

b) Sort all selected elements ($l*p$) globally

c) Choose $P-1$ evenly distributed elements as pivots and broadcast.

With l large enough (or P large) $\Rightarrow$ the pivots will give a good representation of the data $\Rightarrow$ good load balance.

Expensive strategy!

Strategy 5: Compute statistical expectation values for the medians if the distribution is known, eg. exp dist, normal distr. uniform
cf: strategy 2.2

Note: The book describes a variant of Quicksort where the lists are sorted locally in the end (not first).

$\Rightarrow$ Difficult to select pivots (median?) and in the end you may have different amount of data in each processor (bad load balance)

Avoided with our algorithm, better to sort locally first!

Bucket Sort

Algorithm:

1) Define k-buckets in the interval [min, max] and filter the elements into the buckets

2) Assign the buckets to the processors

3) Sort the buckets locally (parallel)

Problems: - Large serial section, filtering part
- Load imbalance, difficult to create buckets with equal number of elements

[See Lab]

What is the complexity? How do we implement the filtering part.

Assume equal sized buckets in the interval [min, max].

$$\Rightarrow \text{bucket } b = \left\lceil \frac{a[i]-min}{(max-min)} * nbuck \right\rceil - 1$$

(special case $a[i] = min$)

Linear time, independent of nbuck!

Hard to parallelize, ie get $Sp > 1$ (*)

Non equal sized buckets?

```
for (i=0; i<n; i++)
    for (b=0; b<nbuck; b++)
        if ("a[i] in buck[b]")
            insert(a[i], buck[b]);
```

Change loop order and parallelize over b

⇒ Linear time! (Insert is critical-section
Avoided with loop-order change)