ALGORITHMES PARALLELES : tris

Daniel Etiemble
LRI, Université Paris Sud
de@lri.fr

Références

- Parallel sorting algorithms, ***http://web.mst.edu/~ercal/387***
- Parallel Programming in C with MPI and OpenMP, Chapter 14: Sorting, Michael J. Quinn

Tris séquentiels

- Tris simples
  - Tri bulle
    - Simple
    - Pair impair
  - Tri insertion
  - Etc
- Tris plus efficaces
  - Tri rapide (quick sort)
    - Choix d’un pivot
    - Liste de tous les éléments inférieurs au pivot
    - Liste de tous les éléments supérieurs au pivot
    - Appel récursifs sur les listes créées
  - Algorithmes utiles
    - Fusion de deux listes triées

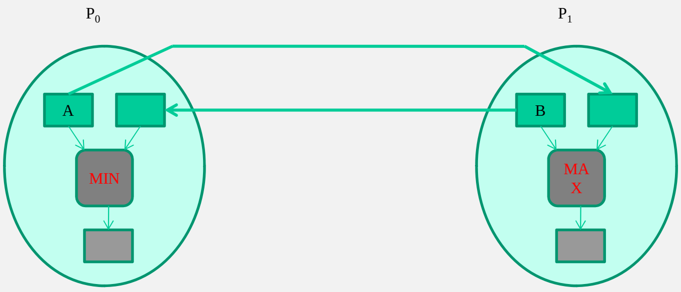
Algorithmes parallèles

- Hypothèses
  - Mémoire distribuée
    - La liste des N éléments est distribuée sur p processeurs avec N/p éléments par processseur
  - Mémoire partagée
    - Les N éléments sont dans la mémoire partagée
    - Accès par les p processeurs à la mémoire partagée
      - Caches

Compare-and-Exchange Sorting Algorithms

Form the basis of several, if not most, classical sequential sorting algorithms.

Two numbers, say *A* and *B*, are compared between P_0 and P_1.



Odd-Even Transposition Sort - example

Step	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	2	7	8	5	1	3	6
1	2	4	7	8	1	5	3	6
2	2	4	7	1	8	3	5	6
3	2	4	1	7	3	8	5	6
4	2	1	4	3	7	5	8	6
5	1	2	3	4	5	7	6	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

Time ↓

Parallel time complexity: $T_{par} = O(n)$ (for P=n)

Odd-Even Transposition Sort – Example (N >> P)

Each PE gets n/p numbers. First, PEs sort n/p locally, then they run odd-even trans. algorithm each time doing a merge-split for 2n/p numbers.

P_0 P_1 P_2 P_3
13 7 12 | 8 5 4 | 6 1 3 | 9 2 10

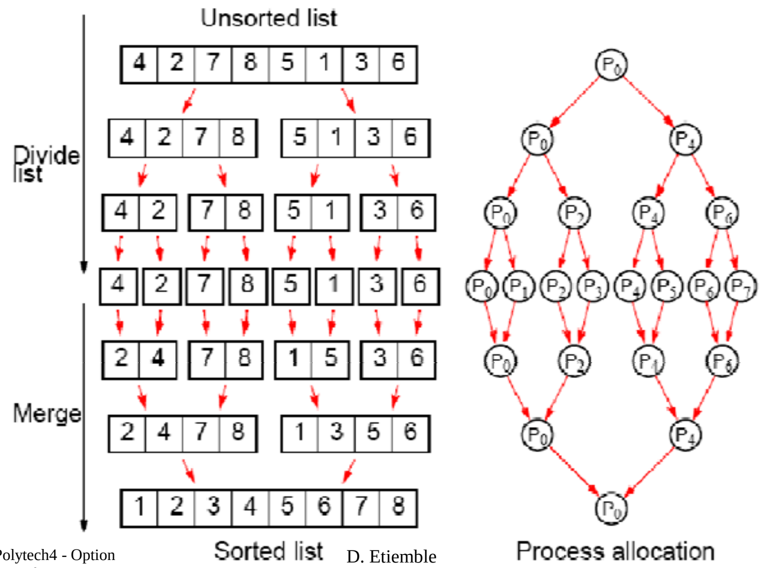
Local sort
7 12 13 | 4 5 8 | 1 3 6 | 2 9 10
O-E
4 5 7 | 8 12 13 | 1 2 3 | 6 9 10
E-O
4 5 7 | 1 2 3 | 8 12 13 | 6 9 10
O-E
1 2 3 | 4 5 7 | 6 8 9 | 10 12 13
E-O

SORTED: 1 2 3 4 5 6 7 8 9 10 12 13

Time complexity: T_{par} = (Local Sort) + (p merge-splits) +(p exchanges)

T_{par} = (n/p)log(n/p) + p*(n/p) + p*(n/p) = (n/p)log(n/p) + 2n

Parallelizing Mergesort



Mergesort - Time complexity

Sequential :

$$T_{seq} = 1 * n + 2 * \frac{n}{2} + 2^2 * \frac{n}{2^2} + \ldots + 2^{\log n} * \frac{n}{2^{\log n}}$$

$$T_{seq} = O(n \log n)$$

Parallel :

$$T_{par} = 2\left(\frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} + \ldots + \frac{n}{2^k} \ldots + 2 + 1\right)$$

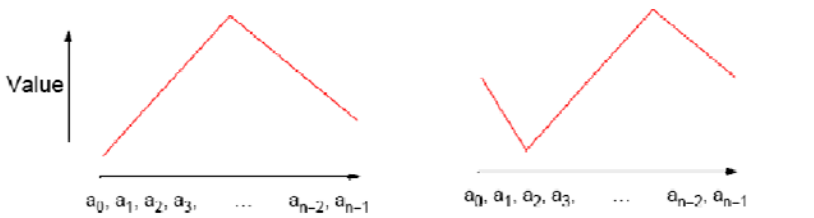
$$= 2n\left(2^0 + 2^{-1} + 2^{-2} + \ldots + 2^{-\log n}\right)$$

$$T_{par} = O(4n)$$

Bitonic Mergesort

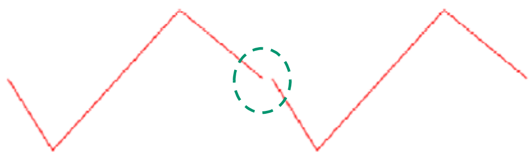
Bitonic Sequence

A *bitonic sequence* is defined as a list with no more than one **LOCAL MAXIMUM** and no more than one **LOCAL MINIMUM**.
(Endpoints must be considered - wraparound)



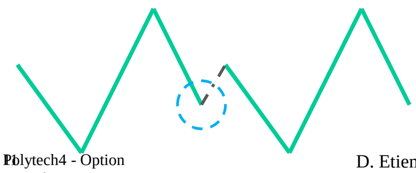
(a) Single maximum (b) Single maximum and single minimum

A *bitonic sequence* is a list with no more than one **LOCAL MAXIMUM** and no more than one **LOCAL MINIMUM**.
(**Endpoints must be considered - wraparound**)



This is ok!
1 Local MAX; 1 Local MIN
The list is bitonic!

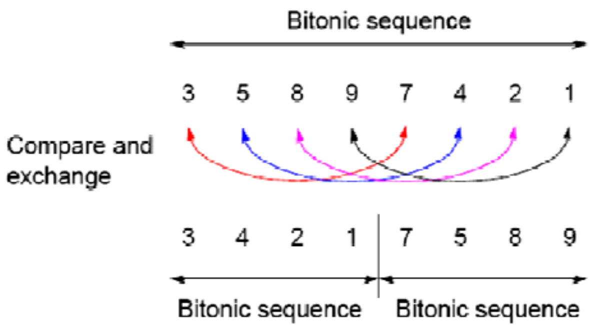
This is **NOT** bitonic! Why?



1 Local MAX; 2 Local MINs

Binary Split

1. Divide the **bitonic list** into two equal halves.
2. Compare-Exchange each item on the first half with the corresponding item in the second half.



Result:
Two bitonic sequences where the numbers in one sequence are all less than the numbers in the other sequence.

Repeated application of binary split

Bitonic list:

24 20 15 9 4 2 5 8 | 10 11 12 13 22 30 32 45

Result after Binary-split:

10 11 12 9 4 2 5 8 | 24 20 15 13 22 30 32 45

If you keep applying the BINARY-SPLIT to each half repeatedly, you will get a SORTED LIST !

10 11 12 9 . 4 2 5 8 | 24 20 15 13 . 22 30 32 45
4 2 . 5 8 10 11 . 12 9 | 22 20 . 15 13 24 30 . 32 45
4 . 2 5 . 8 10 . 9 12 .11 15 . 13 22 . 20 24 . 30 32 . 45
2 4 5 8 9 10 11 12 13 15 20 22 24 30 32 45

Q: How many parallel steps does it take to sort ?

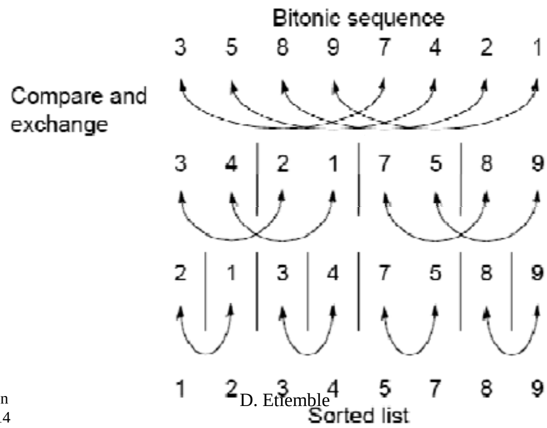
A: **log *n***

Sorting a bitonic sequence

Compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right.

Given a bitonic sequence,

recursively performing '*binary split*' will sort the list.



Bitonic Sort

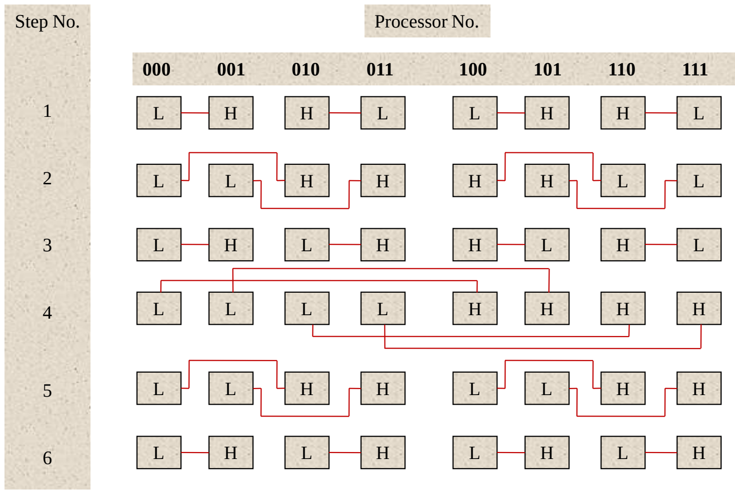
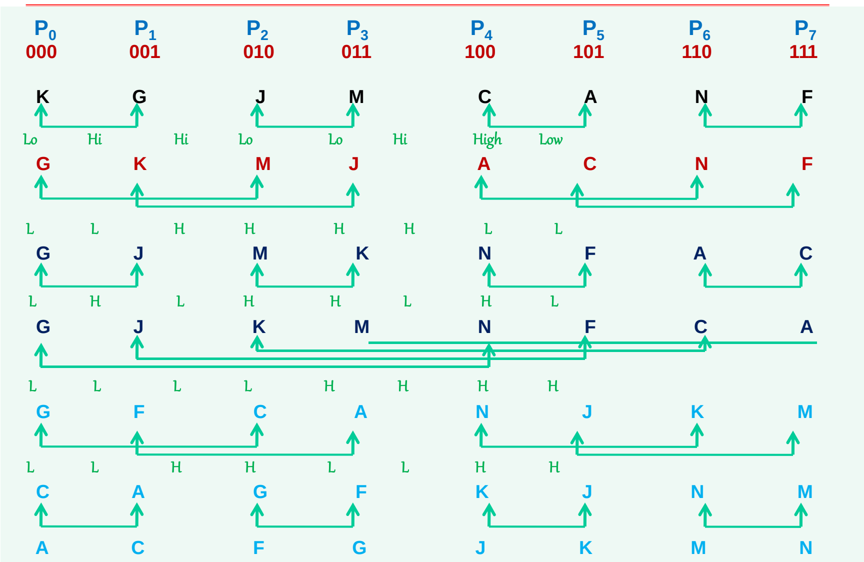


Figure 2: Six phases of Bitonic Sort on a hypercube of dimension 3

Bitonic sort (for N = P)

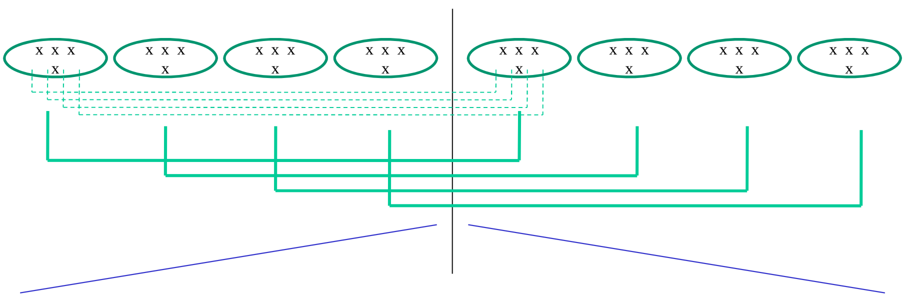


Number of steps (P=n)

In general, with $n = 2^k$, there are k phases, each of 1, 2, 3, …, k steps. Hence the total number of steps is:

$$T_{par}^{bitonic} = \sum_{i=1}^{i=\log n} i = \frac{\log n(\log n + 1)}{2} = O(\log^2 n)$$

Bitonic sort (for N >> P)



Bitonic sort (for N >> P)

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
000	001	010	011	100	101	110	111
2 7 4	13 6 9	4 18 5	12 1 7	6 3 14	11 6 8	4 10 5	2 15 17
Local Sort (ascending):							
2 4 7	6 9 13	4 5 18	1 7 12	3 6 14	6 8 11	4 5 10	2 15 17
L	H	H	L	L	H	High	Low
2 4 6	7 9 13	7 12 18	1 4 5	3 6 6	8 11 14	10 15 17	2 4 5
L	L	H	H	H	H	L	L
2 4 6	1 4 5	7 12 18	7 9 13	10 15 17	8 11 14	3 6 6	2 4 5
L	H	L	H	H	L	H	L
1 2 4	4 5 6	7 7 9	12 13 18	14 15 17	8 10 11	5 6 6	2 3 4
L	L	L	L	H	H	H	H
1 2 4	4 5 6	5 6 6	2 3 4	14 15 17	8 10 11	7 7 9	12 13 18
L	L	H	H	L	L	H	H
1 2 4	2 3 4	5 6 6	4 5 6	7 7 9	8 10 11	14 15 17	12 13 18
L	H	L	H	L	H	L	H
1 2 2	3 4 4	4 5 5	6 6 6	7 7 8	9 10 11	12 13 14	15 17 18

Number of steps (for N >> P)

$$T_{par}^{bitonic} = \text{Local Sort} + \text{Parallel Bitonic Merge}$$

$$= \frac{N}{P}\log\frac{N}{P} + 2\frac{N}{P}(1+2+3+\ldots+\log P)$$

$$= \frac{N}{P}\{\log\frac{N}{P} + 2(\frac{\log P(1+\log P)}{2})\}$$

$$= \frac{N}{P}(\log N - \log P + \log P + \log^2 P)$$

$$T_{par}^{bitonic} = \frac{N}{P}(\log N + \log^2 P)$$

Sequential Quicksort

17	14	65	4	22	63	11

Unordered list of values

17	14	65	4	22	63	11

Choose pivot value

Sequential Quicksort

14	4	11	17	65	22	63

Low list
($\leq$ 17)

High list
(> 17)

Sequential Quicksort

4	11	14	17	65	22	63

Recursively apply quicksort
to low list

4	11	14	17	22	63	65

Recursively apply quicksort
to high list

Sequential Quicksort

4	11	14	17	22	63	65

Sorted list of values

Attributes of Sequential Quicksort

- Average-case time complexity: $\Theta(n \log n)$
- Worst-case time complexity: $\Theta(n^2)$
  - Occurs when low, high lists maximally unbalanced at every partitioning step
- Can make worst-case less probable by using sampling to choose pivot value
  - Example: "Median of 3" technique

Polytech4 - Option parallélisme - 2014 | D. Etiemble | 25

Quicksort Good Starting Point for Parallel Algorithm

- Speed
  - Generally recognized as fastest sort in average case
  - Preferable to base parallel algorithm on fastest sequential algorithm
- Natural concurrency
  - Recursive sorts of low, high lists can be done in parallel

Polytech4 - Option parallélisme - 2014 | D. Etiemble | 26

Definitions of "Sorted"

- Definition 1: Sorted list held in memory of a single processor
- Definition 2:
  - Portion of list in every processor's memory is sorted
  - Value of last element on P_i's list is less than or equal to value of first element on P_{i+1}'s list
- We adopt Definition 2: Allows problem size to scale with number of processors

Parallel Quicksort

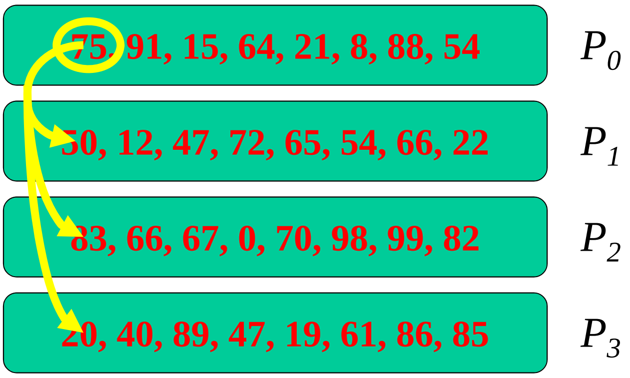
75, 91, 15, 64, 21, 8, 88, 54 P_0

50, 12, 47, 72, 65, 54, 66, 22 P_1

83, 66, 67, 0, 70, 98, 99, 82 P_2

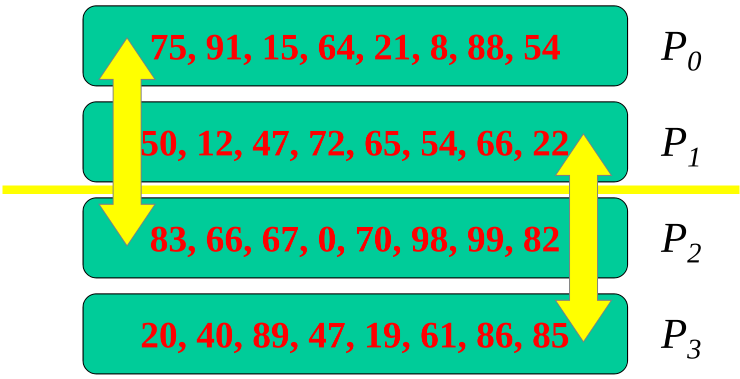
20, 40, 89, 47, 19, 61, 86, 85 P_3

Parallel Quicksort



Process P_0 chooses and broadcasts randomly chosen pivot value

Parallel Quicksort



Exchange "lower half" and "upper half" values"

Parallel Quicksort

Lower "half"	75, 15, 64, 21, 8, 54, 66, 67, 0, 70	P_0
	50, 12, 47, 72, 65, 54, 66, 22, 20, 40, 47, 19, 61	P_1
Upper "half"	83, 98, 99, 82, 91, 88	P_2
	89, 86, 85	P_3

After exchange step

Parallel Quicksort

Lower "half"	75, 15, 64, 21, 8, 54, 66, 67, 0, 70	P_0
	50, 12, 47, 72, 65, 54, 66, 22, 20, 40, 47, 19, 61	P_1
Upper "half"	83, 98, 99, 82, 91, 88	P_2
	89, 86, 85	P_3

Processes P0 and P2 choose and broadcast randomly chosen pivots

Parallel Quicksort

Lower "half"

75, 15, 64, 21, 8, 54, 66, 67, 0, 70 — P_0

50, 12, 47, 72, 65, 54, 66, 22, 20, 40, 47, 19, 61 — P_1

Upper "half"

83, 98, 99, 82, 91, 88 — P_2

89, 86, 85 — P_3

Exchange values

Parallel Quicksort

Lower "half" of lower "half" — 15, 21, 8, 0, 12, 20, 19 — P_0

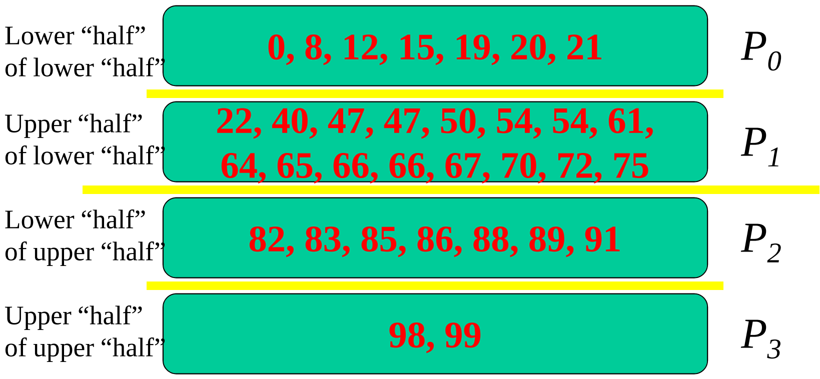
Upper "half" of lower "half" — 50, 47, 72, 65, 54, 66, 22, 40, 47, 61, 75, 64, 54, 66, 67, 70 — P_1

Lower "half" of upper "half" — 83, 82, 91, 88, 89, 86, 85 — P_2

Upper "half" of upper "half" — 98, 99 — P_3

Exchange values

Parallel Quicksort



Each processor sorts values it controls

Hyperquicksort

- Start where parallel quicksort ends: each process sorts its sublist
- First "sortedness" condition is met
- To meet second, processes must still exchange values
- Process can use median of its sorted list as the pivot value
- This is much more likely to be close to the true median

Hyperquicksort

75, 91, 15, 64, 21, 8, 88, 54	P_0
50, 12, 47, 72, 65, 54, 66, 22	P_1
83, 66, 67, 0, 70, 98, 99, 82	P_2
20, 40, 89, 47, 19, 61, 86, 85	P_3

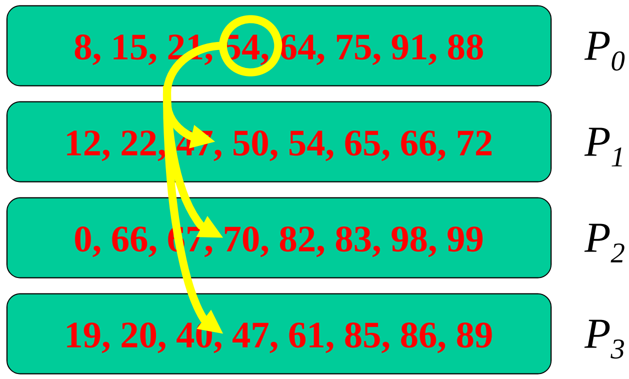
Number of processors is a power of 2

Hyperquicksort

8, 15, 21, 54, 64, 75, 88, 91	P_0
12, 22, 47, 50, 54, 65, 66, 72	P_1
0, 66, 67, 70, 82, 83, 98, 99	P_2
19, 20, 40, 47, 61, 85, 86, 89	P_3

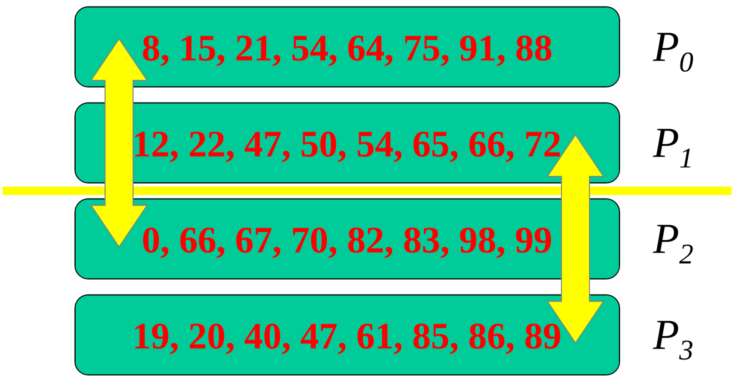
Each process sorts values it controls

Hyperquicksort



Process P_0 broadcasts its median value

Hyperquicksort



Processes will exchange “low”, “high” lists

Hyperquicksort

0, 8, 15, 21, 54 — P_0

12, 19, 20, 22, 40, 47, 47, 50, 54 — P_1

64, 66, 67, 70, 75, 82, 83, 88, 91, 98, 99 — P_2

61, 65, 66, 72, 85, 86, 89 — P_3

Processes merge kept and received values.

Hyperquicksort

0, 8, 15, 21, 54 — P_0

12, 19, 20, 22, 40, 47, 47, 50, 54 — P_1

64, 66, 67, 70, 75, 82, 83, 88, 91, 98, 99 — P_2

61, 65, 66, 72, 85, 86, 89 — P_3

Processes P_0 and P_2 broadcast median values.

Hyperquicksort

- 0, 8, 15, 21, 54 — P_0
- 12, 19, 20, 22, 40, 47, 47, 50, 54 — P_1
- 64, 66, 67, 70, 75, 82, 83, 88, 91, 98, 99 — P_2
- 61, 65, 66, 72, 85, 86, 89 — P_3

Communication pattern for second exchange

Hyperquicksort

- 0, 8, 12, 15 — P_0
- 19, 20, 21, 22, 40, 47, 47, 50, 54, 54 — P_1
- 61, 64, 65, 66, 66, 67, 70, 72, 75, 82 — P_2
- 83, 85, 86, 88, 89, 91, 98, 99 — P_3

After exchange-and-merge step

Complexity Analysis Assumptions

- Average-case analysis
- Lists stay reasonably balanced
- Communication time dominated by message transmission time, rather than message latency

Complexity Analysis

- Initial quicksort step has time complexity $\Theta((n/p) \log (n/p))$
- Total comparisons needed for log p merge steps: $\Theta((n/p) \log p)$
- Total communication time for log p exchange steps: $\Theta((n/p) \log p)$

Another Scalability Concern

- Our analysis assumes lists remain balanced
- As *p* increases, each processor's share of list decreases
- Hence as *p* increases, likelihood of lists becoming unbalanced increases
- Unbalanced lists lower efficiency
- Would be better to get sample values from all processes before choosing median

Parallel Sorting by Regular Sampling (PSRS Algorithm)

- Each process sorts its share of elements
- Each process selects regular sample of sorted list
- One process gathers and sorts samples, chooses pivot values from sorted sample list, and broadcasts these pivot values
- Each process partitions its list into p pieces, using pivot values
- Each process sends partitions to other processes
- Each process merges its partitions

PSRS Algorithm

75, 91, 15, 64, 21, 8, 88, 54	P_0
50, 12, 47, 72, 65, 54, 66, 22	P_1
83, 66, 67, 0, 70, 98, 99, 82	P_2

Number of processors does not have to be a power of 2.

PSRS Algorithm

8, 15, 21, 54, 64, 75, 88, 91	P_0
12, 22, 47, 50, 54, 65, 66, 72	P_1
0, 66, 67, 70, 82, 83, 98, 99	P_2

Each process sorts its list using quicksort.

PSRS Algorithm

8, (15), 21, (54), 64, (75), 88, 91	P_0
12, (22), 47, (50), 54, (65), 66, 72	P_1
0, (66), 67, (70), 82, (83), 98, 99	P_2

Each process chooses *p* regular samples.

PSRS Algorithm

8, (15), 21, (54), 64, (75), 88, 91	P_0
12, (22), 47, (50), 54, (65), 66, 72	P_1
0, (66), 67, (70), 82, (83), 98, 99	P_2
15, 54, 75, 22, 50, 65, 66, 70, 83	

One process collects p^2 regular samples.

PSRS Algorithm

8, 15, 21, 54, 64, 75, 88, 91 — P_0

12, 22, 47, 50, 54, 65, 66, 72 — P_1

0, 66, 67, 70, 82, 83, 98, 99 — P_2

15, 22, 50, 54, 65, 66, 70, 75, 83

One process sorts p^2 regular samples.

PSRS Algorithm

8, 15, 21, 54, 64, 75, 88, 91 — P_0

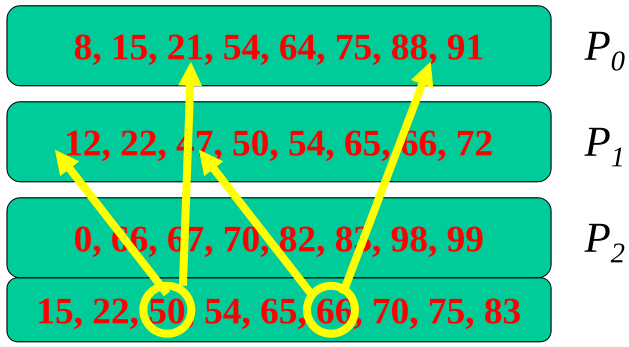
12, 22, 47, 50, 54, 65, 66, 72 — P_1

0, 66, 67, 70, 82, 83, 98, 99 — P_2

15, 22, 50, 54, 65, 66, 70, 75, 83

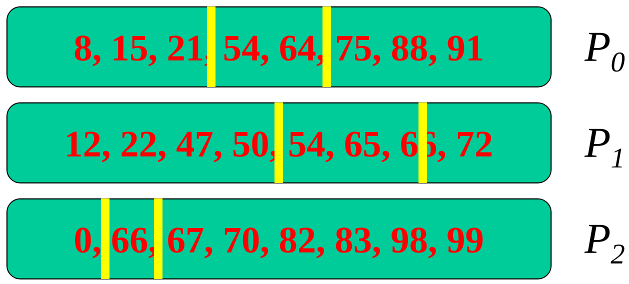
One process chooses p-1 pivot values.

PSRS Algorithm



One process broadcasts *p*-1 pivot values.

PSRS Algorithm



Each process divides list, based on pivot values.

PSRS Algorithm

8, 15, 21	12, 22, 47, 50	0	P_0
54, 64	54, 65, 66	66	P_1
75, 88, 91	72	67, 70, 82, 83, 98, 99	P_2

Each process sends partitions to correct destination process.

Polytech4 - Option parallélisme - 2014 D. Etiemble 57

PSRS Algorithm

0, 8, 12, 15, 21, 22, 47, 50	P_0
54, 54, 64, 65, 66, 66	P_1
67, 70, 72, 75, 82, 83, 88, 91, 98, 99	P_2

Each process merges *p* partitions.

Polytech4 - Option parallélisme - 2014 D. Etiemble 58

Assumptions

- Each process ends up merging close to n/p elements
- Experimental results show this is a valid assumption
- Processor interconnection network supports *p* simultaneous message transmissions at full speed
- 4-ary hypertree is an example of such a network

Time Complexity Analysis

- Computations
  - Initial quicksort: $\Theta((n/p)\log(n/p))$
  - Sorting regular samples: $\Theta(p^2 \log p)$
  - Merging sorted sublists: $\Theta((n/p)\log p)$
  - Overall: $\Theta((n/p)(\log(n/p) + \log p) + p^2\log p)$
- Communications
  - Gather samples pivots: $\Theta(p^2)$
  - Broadcast p-1 pivots: $\Theta(p\log p)$
  - All-to-all exchange: $\Theta(n/p)$
  - Overall: $\Theta(n/p + p^2)$

Summary

- Three parallel algorithms based on quicksort
- Keeping list sizes balanced
  - Parallel quicksort: poor
  - Hyperquicksort: better
  - PSRS algorithm: excellent
- Average number of times each key moved:
  - Parallel quicksort and hyperquicksort: log p / 2
  - PSRS algorithm: $(p-1)/p$

Analysis of Parallel Quicksort

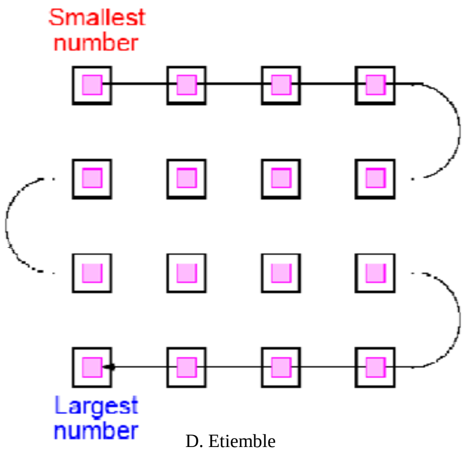
- Execution time dictated by when last process completes
- Algorithm likely to do a poor job balancing number of elements sorted by each process
- Cannot expect pivot value to be true median
- Can choose a better pivot value

Sorting on Specific Networks

- Two network structures have received special attention:
  mesh and hypercube
  Parallel computers have been built with these networks.
- However, it is of less interest nowadays because networks got faster and clusters became a viable option.
- Besides, network architecture is often hidden from the user.
- MPI provides libraries for mapping algorithms onto meshes, and one can always use a mesh or hypercube algorithm even if the underlying architecture is not one of them.

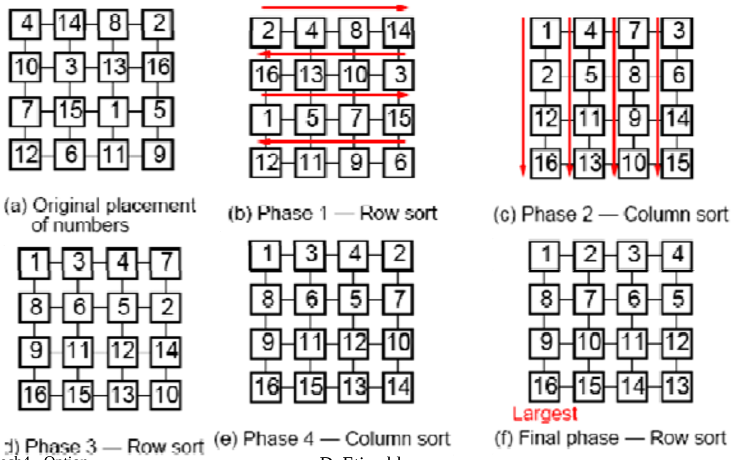
Two-Dimensional Sorting on a Mesh

The layout of a sorted sequence on a mesh could be row by row or *snakelike*:



Shearsort

Alternate row and column sorting until list is fully sorted.
Alternate row directions to get snake-like sorting:



Shearsort – Time complexity

On a **n** x **n** Mesh, it takes **2log n** phases to sort $\mathbf{n^2}$ numbers.
Therefore:

$$T_{par}^{shearsort} = O(n \log n) \quad \textbf{on a n x n mesh}$$

Since sorting n^2 numbers sequentially takes $T_{seq} = O(n^2 \log n)$;

$$Speedup_{shearsort} = \frac{T_{seq}}{T_{par}} = O(n) \quad \textbf{(for } \mathbf{P = n^2)}$$

$$\textbf{However, } efficiency = \frac{1}{n}$$

Rank Sort

Number of elements that are smaller than each selected *element* is counted. This count provides the position of the selected number, its "rank" in the sorted list.

- First **a[0]** is read and compared with each of the other numbers, **a[1]** … **a[n-1]**, recording the number of elements less than **a[0]**.

  Suppose this number is ***x***. This is the index of **a[0]** in the final sorted list.

- The number **a[0]** is copied into the final sorted list **b[0]** … **b[n-1]**, at location **b[x]**. Actions repeated with the other numbers.

Overall sequential time complexity of rank sort: $T_{seq} = O(n^2)$
(not a good sequential sorting algorithm!)

Sequential code

```
for (i = 0; i < n; i++) {          /* for each number */
    x = 0;
    for (j = 0; j < n; j++)         /* count number less than it */
        if (a[i] > a[j]) x++;

    b[x] = a[i];          /* copy number into correct place */
}
```

*This code needs to be fixed if duplicates exist in the sequence.

sequential time complexity of rank sort: $T_{seq} = O(n^2)$

Parallel Rank Sort (P=n)

One number is assigned to each processor.
P_i finds the final index of a[i] in O(n) steps.

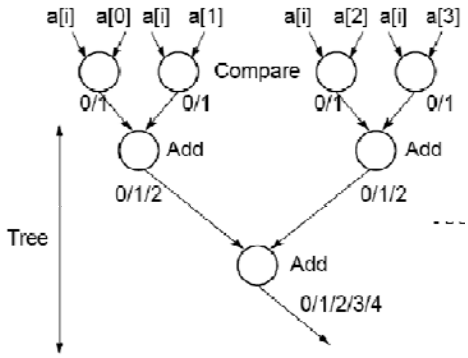
```
forall (i = 0; i < n; i++) {        /* for each no. in parallel*/
   x = 0;
   for (j = 0; j < n; j++)      /* count number less than it */
       if (a[i] > a[j])   x++;
   b[x] = a[i];                    /* copy no. into correct place */
}
```

Parallel time complexity, O(n), as good as any sorting algorithm so far. Can do even better if we have more processors.

Parallel time complexity: $T_{par} = \mathbf{O}(n)$ (for P=n)

Parallel Rank Sort with P = n^2

Use n processors to find the rank of one element. The final count, i.e. rank of a[i] can be obtained using a binary addition operation (global sum ➔ MPI_Reduce())

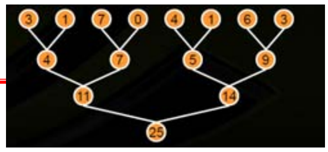


Time complexity (for P=n^2):

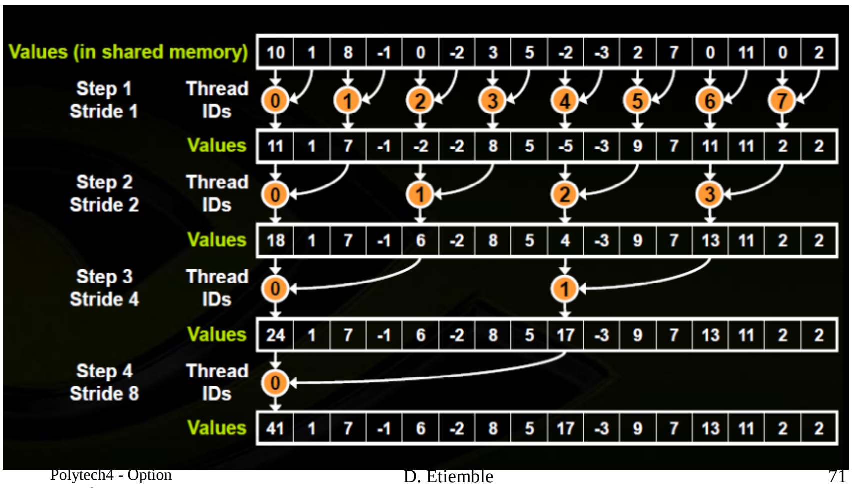
$T_{par} = \mathbf{O}(\log n)$

Can we do it in O(1) ?

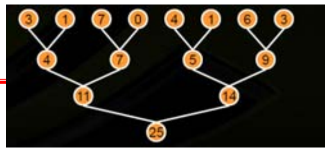
Réduction sur GPU



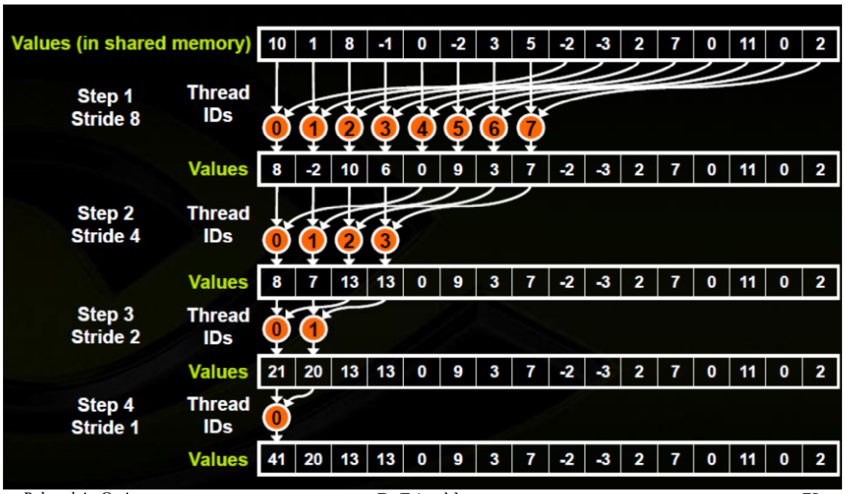
Entrelacement



Réduction sur GPU



Continu



Algorithme Scan (GPU)

$$\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$$