Chapter F1: Parallel Sorting

We consider parallel algorithms where the processors have limited capabilities and where there is a definite topology: a processor can communicate with only a few processors. Also there is a global clock. That is, a SIMD (single-instruction multiple-data) version.

We consider the sorting problem. Here each processor maintains one data value, and there is a specific target order which must be established. At each step, a processor can do a bit of computation, and swap data value with one of its neighbors.

This chapter is based on the discussion by Leighton.

F1.1 Odd-Even Transposition Sort on a Linear Array

A linear array has the processors in a row: each processor can communicate with only the one before it and the one after it. We give a linear-array sorting algorithm that takes precisely $N$ steps, where $N$ is the number of processors and the number of data values. Of course, $N - 1$ is a lower bound, because one might need to get a data value from the last cell to the first cell.

Odd-even Transposition Sort

Repeat as needed:

- In parallel, compare cells 1 and 2, cells 3 and 4, etc,
- and swap/transpose values if necessary
- In parallel, compare cells 2 and 3, cells 4 and 5, etc,
- and swap/transpose values if necessary

Example. 46278135.

The sort proceeds 46-27-81-35 → 4-62-71-83-5 → 42-61-73-85 → 2-41-63-75-8

→ 21-43-65-78 → 1-23-45-67-8

To prove that this algorithm works, focus on the smallest element. It is clear that after a while the smallest element ends up in the leftmost cell. At that stage we can look at the second smallest element, and so on. With a bit more work, one can show that the whole process runs in $N$ steps.

F1.2 Odd-Even Merge Sort

The odd-even merge sort is a famous parallel algorithm. It can be implemented on the topology called a hypercube, but we will consider only the PRAM version. PRAM is
**parallel random access machine**: in this model, there is global memory accessible to all processors. In particular all the data is accessible to all processors (though each can only look at one piece in one clock-step). The algorithm runs in $O(\log^2 N)$ steps.

### MergeSort

1. Use MergeSort to sort each half.
2. Merge the two sorted halves.

Obviously we can do the two sorts in parallel. But the merge step looks inherently sequential. The new idea is as follows:

#### Odd-even merge

(Sorted lists $A$ and $B$)

1. Split $A$ into sublists $A_{\text{even}}$ and $A_{\text{odd}}$ containing even- and odd-numbered entries respectively. Similarly split $B$ into $B_{\text{even}}$ and $B_{\text{odd}}$.
2. Merge (recursively) $A_{\text{even}}$ and $B_{\text{odd}}$ to form $C$, and merge $A_{\text{odd}}$ and $B_{\text{even}}$ to form $D$.
3. Merge $C$ and $D$.

At first glance, this appear ridiculous: we have made no progress. But it turns out that merging $C$ and $D$ is trivial. The fact is, the first two elements of the overall sorted list are the top elements in $C$ and $D$ in some order, the third and forth elements in the sorted list are the second elements of $C$ and $D$ in some order, and so on. (See below.) Thus the merge of $C$ and $D$ can be done in parallel, one processor for each position.

**Example.** Consider merging $A=1467$ and $B=2358$. Then $C$ is merge of 47 and 25, $D$ is merge of 16 and 38; so $C$ is 2457 and $D$ is 1368.

### F1.3 Proof and Analysis of Odd-Even Merge Sort

The proof is made simpler by considering only “small” and “big” elements. For any value $T$ (threshold), we define a data element to be $T$-*small* if it less than $T$ and $T$-*big* if it is $T$ or greater. Then it is immediate that:

**Lemma.** A list is sorted if and only if for all choices of $T$, all $T$-small elements come before all $T$-big elements.

Now, notice that $A_{\text{even}}$ and $A_{\text{odd}}$ have almost the same number of $T$-small elements, as do $B_{\text{even}}$ and $B_{\text{odd}}$. So $C$ and $D$ have almost the same number of $T$-small elements.
Indeed, the pairing of $A_{even}$ with $B_{odd}$ ensures that if $A_{even}$ and $A_{odd}$ have different amounts of small elements, and $B_{odd}$ and $B_{even}$ have different amounts too, then $C$ and $D$ have the same number of smalls. Thus in the final list, all the smalls will occur before all the bigs.

It remains to determine how long this process takes. The final merge of $C$ and $D$ takes $O(1)$ time, and the two submerges occur in parallel. So the overall merge takes $O(\log N)$ steps. There are $O(\log N)$ merge-phases, and so the overall algorithm runs in time $O(\log^2 N)$. Provided there are $O(N)$ processors.

### F1.4 Snake Sort

A mesh or grid has $\sqrt{N}$ processors in each of $\sqrt{N}$ rows. Each processor can communicate with the processor above, below, to the left and to the right. Here

```markdown
Snake Sort

Repeat as needed:

- Sort each row (using odd-even transposition sort): but odd rows are sorted left to right, and even rows are sorted right to left.
- Sort each column from top to bottom.
```

The sorted list appears in snake-like order. And surprisingly, this takes at most $\log N + 1$ phases.

#### Example

| 6 10 14 11 | 6 10 11 14 | 1 7 4 2 |
| 5 13 2 7   | 13 7 5 2   | 6 8 5 3 |
| 9 1 16 8   | 1 8 9 16   | 13 10 9 14 |
| 15 3 12 4  | 15 12 4 3  | 15 12 11 16 |

#### Analysis

As above, it is enough to focus on a split of the data into small and big elements. We say a row is **dirty** if it contains both small and big elements; otherwise it is clean.

**Lemma.** By executing the body once (rows then columns), the number of dirty rows is at least halved.
Proof sketch. After the first phase, all the small elements will be on the left in the odd rows, and on the right in the even rows. Now, the first comparison of the column sort will compare pairs of elements in the same column, swapping if necessary. The result is that for each consecutive pair of rows odd then even: at least one of the rows is now clean.

If both rows had been clean before, then both are still clean. Say there are \( x \) all-small rows, and \( y \) all-big rows. At the end of the column sort, there will still be at least \( x \) all-small rows, and at least \( y \) all-big rows. Furthermore, all the dirty rows are now consecutive.

Thus after \( 2 \log \sqrt{N} = \log N \) phases, all but one row is clean. After the final row phase, all the smalls will occur before all the bigs in the dirty row, as required.

Exercises

1. For the supplied data, illustrate the stages of the given sort and count the actual number of steps. (i) odd-even transposition sort (ii) odd-even merge sort.

   9 3 6 1 7 8 2 4

2. An oblivious sorting algorithm is one which consists entirely of a prescribed sequence of compare-and-swap operations: each step is to compare two elements and swap them if necessary so that they are in a particular order.

   (a) Explain why the odd-even transposition sort is oblivious, but neither Quick-sort nor Merge Sort is oblivious.

   (b) Prove that if an oblivious algorithm sorts a list consisting entirely of 0s and 1s, then it sorts a general list.

3. Code up Snake Sort. Determine the average number of passes through the body needed if the grid starts randomly.