SF2568: Parallel Computations for Large-Scale Problems

Lecture 8: Parallel Sorting

February 5, 2024

Acknowledgements

These slides are an extension of slides by Michael Hanke and Niclas Jansson.

Outline

1. Introduction
2. Review of Sorting Algorithms
3. Algorithms Based on Compare and Exchange
4. Other Algorithms

The Problem

Definition

Let $r_1, \ldots, r_N$ be a list of records where each record is identified by a unique key $k_1, \ldots, k_N$. A **sorting algorithm** is an algorithm which computes a permutation π such that, for the reordered list $r_{\pi(1)}, \ldots, r_{\pi(N)}$, it holds $k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(N)}$ where $\leq$ denotes an (reflexive) order relation in the set of keys

- The most often used order relations are numerical order and lexicographical order
- The general definition includes also the case of sorting in decreasing order

Why Sorting

- Information retrieval in large data sets
- Optimization of many algorithms
  - Graph algorithms
  - Sparse matrix computations
  - Irregular domains and grids
- Sorting is one of the basic tasks in computer science

Parallel Sorting

Sorting is usually considered extensively in introductory computer science classes

Why do it here again?

So what is the challenge?

Issues in Parallel Sorting

1. Where are input and output sequences stored?
    - Stored only in the memory of one processor
    - Distributed over all processors
2. How are comparisons performed?
    - One element per process
    - More than one element per process

Distributed Data

- We need a generalization of the definition of sorting
  - Assume the permutation π according to the sorting criteria be given
  - Let $p = 0, \ldots, P-1$ denote the processors. The data distribution μ after the sorting algorithm fulfils:
    1. μ is a linear data distribution of $1, \ldots, N$,
    2. $r_{\pi(n)}$ resides on processor p with $(p, i) = \mu(n)$

In other words: After sorting, all records residing on processor p are less than or equal to all records residing on processor $p + 1$

Challenge

Obtain optimal (time) complexity with a minimal number of processing elements!

We will only consider internal sorting, that is, all data reside in the main memory

Review – Rank Sort

```
r = zeros(N,1);
for i = 1:N
  for j = 1:N
    if a(j) < a(i)
      r(i) = r(i)+1;
    end
  end
end
```

Properties:

- On a **shared memory** machine with $P = N$ processes, the speedup is $\mathcal{O}(\log N)$.
- On a **distributed memory** machine, there will be either a huge memory overhead or a large amount of communication.

Classification

- **Comparison-based algorithms**: sorts a list by repeatedly **comparing** pairs of elements and, if they are out of order, exchanging them (sorting by compare-and-exchange)
- **Non comparison-based algorithms**: Use a special **apriori** known **properties** of the keys. (ex: The keys are integers from a fixed interval)

Time Complexity

- The optimal complexity of a comparison-based algorithm is

$$T_1^* = \mathcal{O}(N \log N)$$

- The best theoretical parallel complexity for $P = N$ processes is, therefore,

$$T_1^* = \mathcal{O}(\log N)$$

  - Neglecting any communication
- There is an algorithm of this complexity, but the constant hidden in the Big-O notation is extremely large

Selected Sorting Algorithms

Method	Best	Average	Worst
Rank Sort	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
Bubble Sort	$\mathcal{O}(N)$		$\mathcal{O}(N^2)$
Merge Sort	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$
Insertion Sort	$\mathcal{O}(N)$	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
Quicksort	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(N^2)$
Heap Sort	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$

Other Selected Sorting Algorithms

Method	Best	Average	Worst
Bucket Sort	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N^2)$
Radix Sort	$\mathcal{O}(N\,d/s)$	$\mathcal{O}(N\,d/s)$	$\mathcal{O}(N\,d/s)$

- d - key length
- s - chunk size

Basic Assumptions

- The records consist only of a key value
- The keys are a set of numbers
- The order relation is the natural order by magnitude
- Most often, the list is mapped to an array (vector), say a.
- For a parallel version, the vector is linearly distributed over the processors $0, \ldots, P-1$

Compare and Exchange

- The basic operation for two data items `A` and `B` is

```
if A > B
  temp = A;
  A = B;
  B = temp;
end
```

- *What may be the strategy if A and B are stored on different processors (assuming a distributed memory machine)*

Distributed Compare and Exchange

Processor 0

```
send(A, 1);
receive(A, 1);
```

Processor 1

```
receive(A, 0);
if A > B
  send(B, 0);
  B = A;
else
  send(A, 0);
end
```

Symmetric Distributed Compare and Exchange

- In the previous code snippet, one processor is idle while the other is doing the comparison
- In the spirit of SPMD programming, a more symmetric compare-and-exchange implementation would be desirable

Processor 0

```
send(A, 1);
receive(B, 1);
if A > B
  A = B;
end
```

Processor 1

```
receive(A, 0);
send(B, 0);
if A > B
  B = A;
else
  send(A, 0);
end
```

- Even if the number of comparisons is doubled, the execution time is identical to the first version
- In MPI, the send/receives can conveniently be combined into one `MPI_Sendrecv` call

Compare and Exchange for Linear Data Distributions

- We will generalize the symmetric strategy to the case that N is a multiple of P
- Linear data distribution places N/P consecutive elements on each processor
- Since finally the elements on each processor are sorted, we can assume that the first step of each sorting algorithm is local sorting
- Therefore, comparing elements on different processors amounts to a merging step (which has complexity $(2(N/P) - 1)t_a$) followed by a split step

Compare and Exchange – Algorithm

Processor 0

```
send(list0, 1);
receive(list1, 1);
list  = merge(list0, list1);
list0 = list(1:N/P);
```

Processor 1

```
receive(list0, 0);
send(list1, 0);
list  = merge(list0, list1);
list1 = list(N/P+1:end);
```

Note: This algorithm assumes implicitly that the results of the merge operations are **identical on both processors**.

Assumption

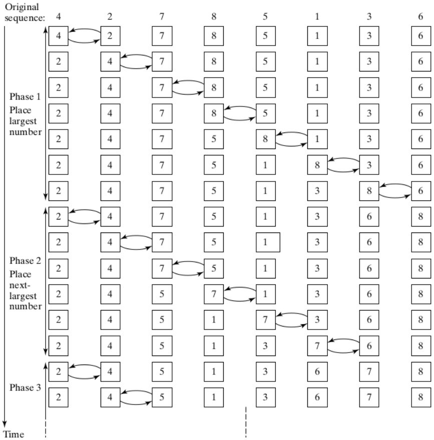
In the following we will assume that each processor holds (at most) one element

Bubble Sort

- **Idea**: The largest number is moved to the end of the list by a number of compare and exchanges. Then this step is repeated for the remaining list, and so on.
- Sequential algorithm (non-optimized!):

```
for i = N:-1:2
  for j = 1:i-1
    if a(j) > a(j+1)
      temp   = a(j);
      a(j)   = a(j+1);
      a(j+1) = temp;
    end
  end
end
```

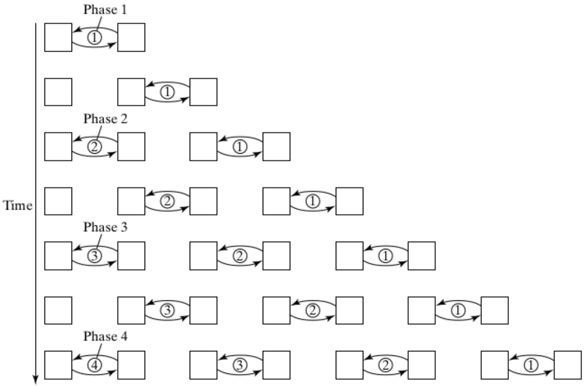
Bubble Sort


Original sequence: 4 2 7 8 5 1 3 6
Phase 1
Place largest number
Phase 2
Place next-largest number
Phase 3
Time

Bubble Sort – Properties

- Complexity is independent of the order of the elements
- Time complexity $\mathcal{O}(N^2)$
- The algorithm is purely sequential
- **Straightforward parallelization idea**: The bubbling action of the next iteration of the inner loop could start before the preceding iteration has finished, so long as it does not overtake the proceeding bubbling action
- This suggests that a pipeline implementation might be beneficial

Bubble Sort – Pipelined Computation



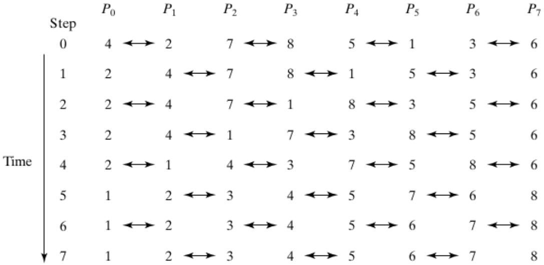
Parallel Bubble Sort – Odd-Even Sort

- **Idea**: Rearrange the comparisons in as many independent comparisons as possible!
- **Observation**: Comparisons can be carried out in parallel if every processor is involved in at most one compare-and-exchange.
- This can be achieved if the processors are grouped into even/odd pairs or odd/even pairs

Odd-Even Sort – Algorithm

- **odd-even phase**
  - The odd processes p compare and exchange their elements with the even processors $p+1$
- **even-odd phase**
  - The even processes compare and exchange their elements with the odd processors $p+1$

Odd-Even Sort



Fact

The algorithm is guaranteed to terminate after $N/2$ odd-even and even-odd steps

Odd-Even Sort – Implementation

$p = 1, 3, \ldots, N-1$

```
% Even Phase
send(A, p-1);
receive(B, p-1);
if A < B
  A = B;
end
if p <= N-3
% Odd Phase
  send(A, p+1);
  receive(B, p+1);
  if A > B
    A = B;
  end
end
```

$p = 0, 2, \ldots, N-2$

```
% Even Phase
receive(A, p+1);
send(B, p+1);
if A < B
  B = A;
end
if p >= 2
% Odd Phase
  receive(A, p-1);
  send(B, p-1);
  if A > B
    A = B;
  end
end
```

Odd-Even Sort – Complexity

Assume linear data distribution

1. Initial local sort: $\mathcal{O}(N/P \log N/P)$
2. During the global sorting phase, there are $P/2$ iterations
3. Each iteration contains
   - 4 send/receives of length N/P
   - 2 merges of length $2N/P$

$$T_P = \frac{P}{2}\left(4\left(t_{startup} + N/P\, t_{data}\right) + 2\left(N/P - 1\right) t_a\right) + \mathcal{O}(N/P \log N/P)$$

Neglecting communication costs, for $P = N$ we obtain

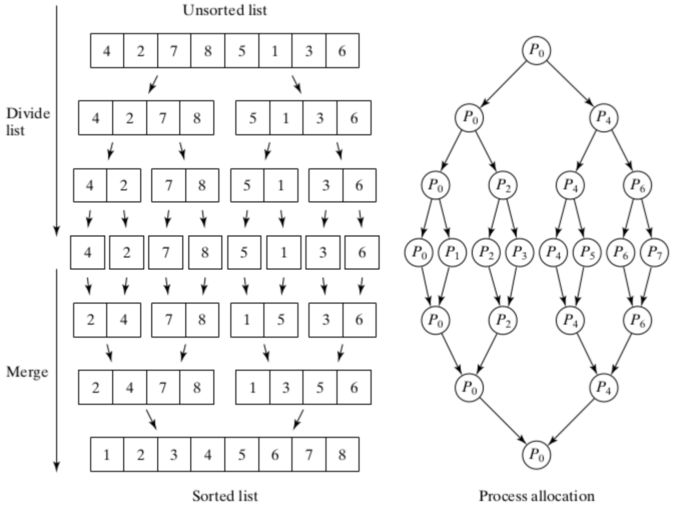
$$T_P = \mathcal{O}(P), \qquad S_P = \mathcal{O}(\log P)$$

Merge Sort – Idea

- **Fact**: Two ordered lists of length n can be merged into one ordered list of length $2n$ with $2n-1$ comparisons
- A list of length $n=1$ is ordered by definition
- This suggests a divide and conquer strategy:
  - Apply the following algorithm `mergesort` recursively

  1. If $n>1$, divide the list in two halves and call `mergesort` for both sub lists
  2. Merge the two sub lists into one ordered list and return that list
  3. If $n=1$, return the list

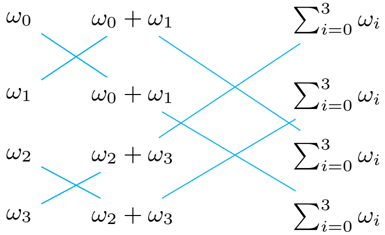
Parallel Merge Sort



Merge Sort – Properties

- Complexity is independent of the order of the elements
- Time complexity is $\mathcal{O}(N \log N)$, which is optimal for comparison-based sorting algorithms
- The divide-and-conquer approach is immediately parallelizable:
  - The data (list) distribution step amounts to a scatter operation (if the data are not already distributed in this way)
  - The merge step can be done along the lines of **recursive doubling**

```
s = omega_p;
for d = 0:D-1
  q = bitflip(p,d);
  send(s,q);
  receive(h,q);
  s = s+h;
end
```



Merge Sort – Implementation

- Assume the data is already distributed
- Assume for simplicity $N = P = 2^D$ (being a power of 2)
- Implementation

```
list = A;
for d = 0:D-1
  q = bitflip(p,d);
  send(list, q);
  receive(listq,q);
  list = merge(list, listq);
end
```

- **Note:** After completion, every processor holds the complete list!

Merge Sort – Performance Analysis

- For each d, the number of data exchanged is 2^d
- Communication time

$$
\begin{aligned}
t_{comm} &= \sum_{d=0}^{D-1} (t_{startup} + 2^d t_{data}) \\
&\approx D t_{startup} + 2^D t_{data} = \log N t_{startup} + N t_{data}
\end{aligned}
$$

- Computation time

$$t_{comp} = \sum_{d=0}^{D-1} (2 \cdot 2^d - 1) t_a \approx N t_a$$

- Hence,

$$T_P = \mathcal{O}(P), \qquad S_P = \mathcal{O}(P)$$

Quicksort

- In average, quicksort has a sequential time complexity of $\mathcal{O}(N \log N)$
- Quicksort is another example of a divide-and-conquer algorithm

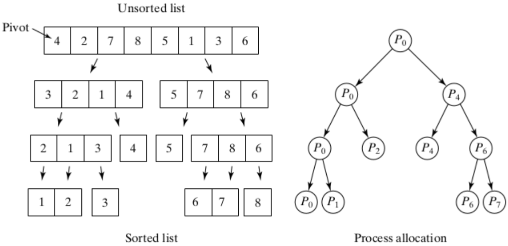
Question

Can quicksort be a basis for a good parallel algorithm?

Quicksort – Algorithm

1. Choose a pivot element from the list
2. Partition the list into two sub lists such that one list contains all elements less than the pivot, while the remaining elements form another list
3. Apply `quicksort` recursively to these two sub lists

The divide-and conquer strategy is similar to mergesort



Quicksort – Problems

- In contrast to `mergesort`, the data distribution is **not known in advance**
- Fundamental problem with all tree constructions - initial partition must be done on one processor, only, **which will seriously limit speed**.
- The tree with subproblems may become **heavily unbalanced**, i.e., the sub list may vary considerably in length
- The **worst case behaviour is** $\mathcal{O}(N^2)$ even in the parallel case

Bitonic Mergesort

Question

Can we do better?

Definition

A sequence $a_1, a_2, \ldots, a_n$ is called bitonic if either

1. there exists an index i such that $a_1 < \cdots < a_i$ and $a_i > \cdots > a_n$, or
2. there is a cyclic shift of indices such that (1) holds

Bitonic Split

Let $a_1, \ldots, a_N$ be a bitonic sequence with $N = 2n$. Consider:

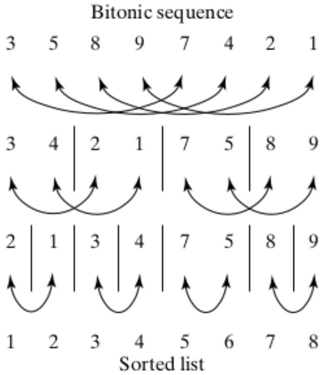
```
function [al,ar] = splitincr(a)
n = length(a)/2;
for i = 0:n-1
  if a(i) > a(i+n)
    al(i) = a(i+n);
    ar(i) = a(i);
  else
    al(i) = a(i);
    ar(i) = a(i+n);
  end
end
```

```
function [al,ar] = splitidecr(a)
n = length(a)/2;
for i = 0:n-1
  if a(i) < a(i+n)
    al(i) = a(i+n);
    ar(i) = a(i);
  else
    al(i) = a(i);
    ar(i) = a(i+n);
  end
end
```

Properties of the transformed sequence

- `al` and `ar` are two bitonic sequences
- all elements of the first sequence are smaller than those of the second sequence

Sorting a Bitonic Sequence

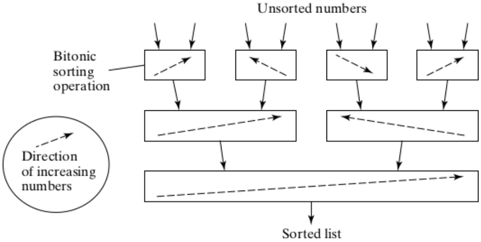


Sorting a Bitonic Sequence

```
function a = bisort(a,dir)
  if length(a) == 1
    return
  end
  switch dir
    case incr:
      [al,ar] = splitincr(a);
    case decr:
      [al,ar] = splitdecr(a);
    otherwise:
    error(' Oh no! :( ')
  end
  a1 = bisort(al,dir);
  a2 = bisort(ar,dir);
  a = [a1,a2];
```

Bitonic Sort

- A sequence containing **two** elements is **bitonic**
- **Every bitonic sequence can be sorted** by the algorithm given before in increasing or decreasing order
- **Joining two neighbouring sequences** where the first is increasing while the second is decreasing **provides a bitonic sequence**



Bitonic Sort

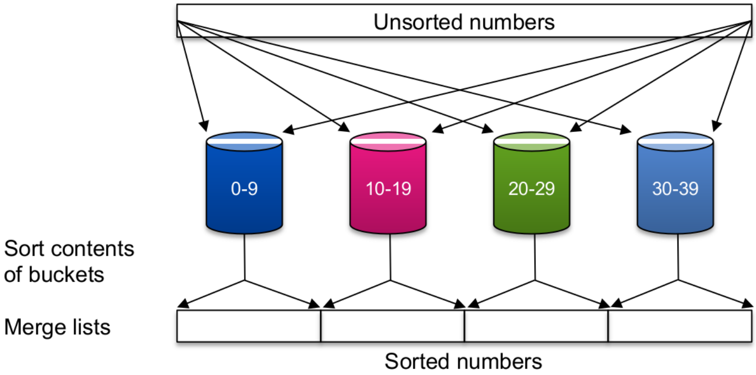
```
function a = bitonicsort(a,dir)
  n = length(a);
  if n == 1
    return
  end
  a1 = bitonicsort(a(0:n/2-1),incr);
  a2 = bitonicsort(a(n/2:n-1),decr);
  a = bisort([a1,a2],dir);
```

Fact

Bitonic Sort implemented on $P = N = 2^D$ processors has a time complexity of $\mathcal{O}(\log^2 N)$!

- Bitonic sort can be mapped efficiently to mesh and hypercube processor topologies

Bucket Sort


Unsorted numbers
0-9
10-19
20-29
30-39
Sort contents of buckets
Merge lists
Sorted numbers

Bucket Sort

Assumption

The keys are uniformly distributed on the interval $[a, b]$

- **Idea**: For P processors, subdivide the interval $[a, b]$ into P equal chunks, $I_p = [x_p, x_{p+1})$ where
  $x_p = a + ph, h = (b - a)/P$
  i.e. one (big) bucket per processor
- Each processor scans its local list and determines where to send the elements
- After having received all elements, sort the local lists

Bucket Sort – Performance Analysis

- Computation: Each processor needs to perform N/P comparisons
- The local sort has $\mathcal{O}(N/P \log(N/P))$ comparisons. Hence,

$$t_{comp} = \mathcal{O}(N/P \log(N/P))t_a$$

- Invoking the uniform distribution assumption, $P - 1$ communication steps (all-to-all) with a message length $\mathcal{O}(N/P^2)$

$$t_{comm} = (P-1)(t_{startup} + N/P^2 t_{data})$$

Sample Sort

- Bucket sort does not work efficiently if the interval is unknown or if the assumption of uniform distribution is not fulfilled
  - Same problem as for quicksort: unevenly divided list
  - All the numbers fall into one bucket
- **Can one find a subdivision of the real axis** $x_0 < x_1 < \cdots < x_P$ **such that the number of elements falling in each subinterval is roughly constant**
- This subdivision is data-dependent and must be generated at each sorting

Sample Sort – Idea

- A sample of size s is selected from the sequence and sorted
- Select $P-1$ elements $x_1 < \cdots < x_{P-1}$ (called splitters)
- Set $x_0 = -\infty$ and $x_P = +\infty$
- Define the buckets by $I_P = [x_p, x_{p+1})$

Sample Sort – Complexity

$$\begin{aligned} T_P = {} & \mathcal{O}(N/P \, \log N/P) && \text{local sort} \\ & + \mathcal{O}(P^2 \, \log P) && \text{sample sort} \\ & + \mathcal{O}(P \, \log N/P) && \text{block partition} \\ & + \mathcal{O}(N/P) + \mathcal{O}(P \, \log P) && \text{communication} \end{aligned}$$

Here, we used a sample of $P - 1$ elements per process, $s = P(P - 1)$

Radix Sort

Assumption

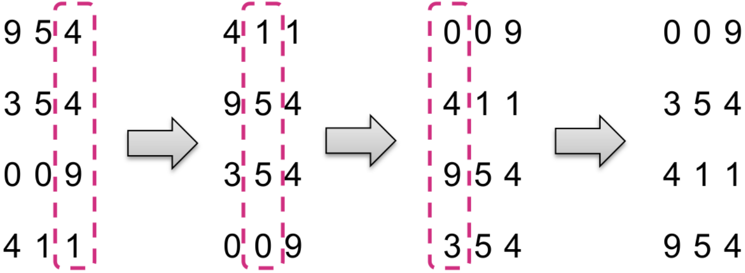
The keys are positive integers

- Then every key k has a unique representation in a β-adic position system,

$$k = a_0\beta^0 + a_1\beta^1 + \cdots + a_n\beta^n$$

- Assume that n is the maximal exponent appearing
- **Idea**: Sort the sequence by first sorting the least significant digit (a_0), then a_1, and so forth until the most significant digit (a_n)

Radix Sort



Radix Sort – Complexity

- Let $\beta = 2^r$

$$T_P = \beta n(\mathcal{O}(\log N) + \mathcal{O}(N))$$

  The $\mathcal{O}(N)$-term belongs to the communication part
- The challenge is to keep β small
  - A parallel implementation needs to perform communication
  - A bitwise radix sort will quickly become quite expensive (too many passes)