Fundamental Algorithms

Chapter 7: Parallel Sorting

Jan Křetínský

Winter 2017/18
Sequential MergeSort

MergeSort(A: Array[1..n]) {
    if n > 1 then {
        m := floor(n/2);
        create array L[1...m];
        for i from 1 to m do { L[i] := A[i]; }

        create array R[1...n-m];
        for i from 1 to n-m do { R[i] := A[m+i]; }

        MergeSort(L);
        MergeSort(R);

        Merge(L,R,A);
    }
}

(How) can we parallelise MergeSort?
MergeSort in Parallel?

MergeSortPar(A: Array[1..n]) {
  if n > 1 then {
    m := floor(n/2);

    do in parallel {
      create array L[1...m];
      for i from 1 to m do { L[i] := A[i]; }
      MergeSort(L); // even better: MergeSortPar(L)

      create array R[1...n-m];
      for i from 1 to n-m do { R[i] := A[m+i]; }
      MergeSort(R); // even better: MergeSortPar(R)
    }

    Merge(L,R,A); // desired: MergePRAM(L,R,A)
  }
}
Parallel MergeSort

**Idea:**

- parallelise “divide-and-conquer”: recursive calls can be done in parallel
- use $p/2$ processors for each of the recursive calls (if $p$ processors are available)
Parallel MergeSort

Idea:
- parallelise “divide-and-conquer”: recursive calls can be done in parallel
- use $p/2$ processors for each of the recursive calls (if $p$ processors are available)

Merging in Parallel?
- can Merge be executed in parallel?
- by how many processors?
Can Merge be Parallelised?

```plaintext
Merge (L: Array[1..p], R: Array[1..q], A: Array[1..n]) {
  // merge the sorted arrays L and R into A (sorted)
  // we presume that n=p+q
  i := 1; j := 1;
  for k from 1 to n do {
    if i > p
      then { A[k] := R[j]; j := j + 1; }
    else if j > q
      then { A[k] := L[i]; i := i + 1; }
    else if L[i] < R[j]
      then { A[k] := L[i]; i := i + 1; }
    else { A[k] := R[j]; j := j + 1; }
  }
}
```
Can Merge be Parallelised?

**Merge** \((L: \text{Array}[1..p], R: \text{Array}[1..q], A: \text{Array}[1..n])\) {
  // merge the sorted arrays \(L\) and \(R\) into \(A\) (sorted)
  // we presume that \(n=p+q\)
  \(i:=1; j:=1:\)
  \[\text{for } k \text{ from } 1 \text{ to } n \text{ do } \{
  \begin{align*}
  &\text{if } i > p \\
  &\quad \text{then } \{ A[k]:=R[j]; j=j+1; \} \\
  &\text{else if } j > q \\
  &\quad \text{then } \{ A[k]:=L[i]; i:=i+1; \} \\
  &\text{else if } L[i] < R[j] \\
  &\quad \text{then } \{ A[k]:=L[i]; i:=i+1; \} \\
  &\text{else } \{ A[k]:=R[j]; j:=j+1; \}
  \end{align*}\]
  \}
}

**Problem:** inherently sequential progress through arrays \(A, L, R\)
Odd-Even Merge

Ideas:

- start with two sorted lists of length $n/2$:

  \[
  \begin{array}{cccccccc}
  2 & 3 & 4 & 7 & 1 & 5 & 6 & 8 \\
  \end{array}
  \]

Observations

- final sequence is nearly sorted (only pairwise exchange required)
- odd- and even-indexed elements can be processed in parallel
Odd-Even Merge

Ideas:

• start with a two sorted lists of length $n/2$:

  2 3 4 7 1 5 6 8

• consider elements with odd and even index:

  2 3 4 7 1 5 6 8

Observations

• final sequence is nearly sorted (only pairwise exchange required)
• odd- and even-indexed elements can be processed in parallel
Odd-Even Merge

Ideas:

- start with a two sorted lists of length $n/2$:
  
  $2 \ 3 \ 4 \ 7 \ 1 \ 5 \ 6 \ 8$

- consider elements with odd and even index:
  
  $2 \ 3 \ 4 \ 7 \ 1 \ 5 \ 6 \ 8$

- sort odd- and even-indexed elements separately:
  
  $1 \ 3 \ 2 \ 5 \ 4 \ 7 \ 6 \ 8$
Odd-Even Merge

Ideas:

- start with a two sorted lists of length $n/2$:

  \[
  \begin{array}{cccccccc}
  2 & 3 & 4 & 7 & 1 & 5 & 6 & 8 \\
  \end{array}
  \]

- consider elements with odd and even index:

  \[
  \begin{array}{cccccccc}
  2 & 3 & 4 & 7 & 1 & 5 & 6 & 8 \\
  \end{array}
  \]

- sort odd- and even-indexed elements separately:

  \[
  \begin{array}{cccccccc}
  1 & 3 & 2 & 5 & 4 & 7 & 6 & 8 \\
  \end{array}
  \]
Odd-Even Merge

Ideas:

- start with a two sorted lists of length \( n/2 \):
  
  \[
  \begin{array}{cccccc}
  2 & 3 & 4 & 7 & 1 & 5 \\
  6 & 8
  \end{array}
  \]

- consider elements with odd and even index:
  
  \[
  \begin{array}{cccccc}
  2 & 3 & 4 & 7 & 1 & 5 \\
  6 & 8
  \end{array}
  \]

- sort odd- and even-indexed elements separately:
  
  \[
  \begin{array}{cccccc}
  1 & 3 & 2 & 5 & 4 & 7 \\
  6 & 8
  \end{array}
  \]

Observations

- final sequence is nearly sorted (only pairwise exchange required)
- odd- and even-indexed elements can be processed in parallel
Correctness of the Final Exchange Step

Claim (after odd/even sort):
- exchanges of $a_{2i}$ and $a_{2i+1}$ are sufficient for sorting

| 1 | 3 | 2 | 5 | 4 | 7 | 6 | 8 |

Proof:
- let $O$ and $E$ be sorted odd and even sequence, respectively; let $A$ be sorted sequence
- add $E_0 = -\infty$ and $O_{n/2 + 1} = \infty$.
- for $i \in 0, \ldots, n/2$

$$A_{2i} = \min\{E_i, O_{i+1}\}$$
$$A_{2i+1} = \max\{E_i, O_{i+1}\}$$

note that $A$ contains elements $A_0 = -\infty$ and $A_{n+1} = \infty$. 
Correctness of the Final Exchange Step

\( i = 0 \) the first two elements in \( A \) are clearly \( A_0 = -\infty \) and \( A_1 = O_1 \);
\( i \geq 1 \) using the induction hypothesis for \( i' = 0, \ldots, i - 1 \) gives that the positions \( A_0, \ldots, A_{2i-1} \) are composed from \( i \) even and \( i \) odd elements; hence, the next element is

\[
A_{2i} = \min\{E_i, O_{i+1}\}
\]

(note that \( E \) is indexed starting from 0 and \( O \) starting from 1)

now, we either have more odd or more even elements; however the number of even/odd elements within a prefix of \( A \) can at most differ by 1; therefore if the last element was odd we now have to choose the smallest even element (and vice versa); this gives

\[
A_{2i+1} = \max\{E_i, O_{i+1}\}
\]
Correctness of the Final Exchange Step

Claim (after odd/even sort):

- exchanges of $a_{2i}$ and $a_{2i+1}$ are sufficient for sorting

![Sorted List]

1 3 2 5 4 7 6 8
Correctness of the Final Exchange Step

Claim (after odd/even sort):
- exchanges of $a_{2i}$ and $a_{2i+1}$ are sufficient for sorting

Counting Argument: $x$ an odd-indexed element: $x = a_{2i+1}$
- exactly $i$ odd-indexed elements are smaller than $x$ (sorted lists)
- $d_l, d_r =$ number of odd-indexed elements $< x$ in left/right half
  \[ i = d_l + d_r \]
- $v_l, v_r =$ number of even-indexed elements $< x$ in left/right half
  - $x$ in left half: $v_l = d_l$, $v_r \in \{d_r, d_r - 1\}$
  - $x$ in right half: $v_l \in \{d_l, d_l - 1\}$, $v_r = d_r$
- consequence: $v_l + v_r \in \{d_l + d_r, d_l + d_r - 1\} = \{i, i - 1\}$
Correctness of the Final Exchange Step (2)

Counting Argument:
- count even- and odd-indexed elements < x in both halves
- \( v_l + v_r \in \{d_l + d_r, d_l + d_r - 1\} = \{i, i - 1\} \)

Possible Scenarios:
- \( v_l + v_r = i \Rightarrow \) exactly \( i \) even elements < \( x \)
  \( \Rightarrow i \)-th even-indexed element \( a_{2i} < x \rightarrow \text{OK} \)
- \( v_l + v_r = i - 1 \Rightarrow \) exactly \( i - 1 \) even elements < \( x \)
  therefore: \( a_{2(i-1)} < x \), but \( a_{2i} > x \rightarrow \text{exchange} \)
- in both cases:
  \( a_{2(i+1)} > x \) (at most \( i \) even elements < \( x \)) \( \rightarrow \text{OK} \)
  \( a_{2(i-1)} < x \) (at least \( i - 1 \) even elements < \( x \)) \( \rightarrow \text{OK} \)

\( \Rightarrow \) only the left even-indexed neighbour of \( x \) can be out of place
OddEvenMerge – A First Try

OddEvenMerge_1 (A: Array[1..n]) {
// merge the sorted arrays A[1..n/2] and A[n/2+1..n]
// into A (sorted); n is a power of 2

OddEvenSplit(A, Odd, Even);
Sort(Odd); Sort(Even);

OddEvenJoin(A, Odd, Even);

for i from 1 to n/2−1 do {
  then exchange A[2i] and A[2i+1]
}
}
OddEvenSplit and OddEvenJoin (in parallel!)

OddEvenSplit (A: Array[1..n],
    Odd: Array[1..n/2], Even: Array[1..n/2]) {
    for i from 1 to n/2 do in parallel {
        Odd[i] := A[2i−1];
        Even[i] := A[2i];
    }
}

OddEvenJoin (A: Array[1..n],
    Odd: Array[1..n/2], Even: Array[1..n/2]) {
    for i from 1 to n/2 do in parallel {
        A[2i−1] := Odd[i];
        A[2i] := Even[i];
    }
}
Towards a Better Implementation of OddEvenMerge

After OddEvenSplit:

- Odd consists of two halves that are already sorted
- Even consists of two halves that are already sorted

⇒ Odd and Even can be sorted using OddEvenMerge
Towards a Better Implementation of OddEvenMerge

After OddEvenSplit:

• Odd consists of two halves that are already sorted
• Even consists of two halves that are already sorted
⇒ Odd and Even can be sorted using OddEvenMerge

OddEvenMerge in Parallel:

• OddEvenSplit and OddEvenJoin are already parallel
• calls to OddEvenMerge can be executed in parallel (recursive calls will again issue parallel calls)
• final exchange loop can be parallelised
Parallel OddEvenMerge

OddEvenMergePRAM (A: Array [1..n]) {
  ! add stopping criterion:
  if n<=2 then { SortTwo(A); return; };

  OddEvenSplit(A, Odd, Even);

  do in parallel{
    OddEvenMergePRAM(Odd);
    OddEvenMergePRAM(Even);
  }

  OddEvenJoin(A, Odd, Even);

  for i from 1 to n/2−1 do in parallel {
      then exchange A[2i] and A[2i+1]
  }
}
Parallelism in OddEvenMerge

\[
\begin{array}{ccccc}
2 & 3 & 7 & 8 & 1 & 4 & 5 & 6 \\
\downarrow & & & & & & & \\
2 & 7 & 1 & 5 & 3 & 8 & 4 & 6 \\
\downarrow & & & & & & & \\
2 & 1 & 7 & 5 & 3 & 4 & 8 & 6 \\
\downarrow & & & & & & & \\
1 & 2 & 5 & 7 & 3 & 4 & 6 & 8 \\
\downarrow & & & & & & & \\
1 & 5 & 2 & 7 & 3 & 6 & 4 & 8 \\
1 & 2 & 5 & 7 & 3 & 4 & 6 & 8 \\
\downarrow & & & & & & & \\
1 & 3 & 2 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

(on 4 processors)
(on 2×2 processors)
(on 4×1 processors)
(on 2×2 processors)
(on 4 processors)
OddEvenMergeSort (in Parallel)

OddEvenMergeSortPRAM(A: Array[1..n]) {
! EREW PRAM with n/2 processors
! n assumed to be 2^k
if n >= 2 then {

do in parallel {
    OddEvenMergeSortPRAM(A[1..n/2]);
    |
    OddEvenMergeSortPRAM(A[n/2+1..n]);
};

OddEvenMergePRAM(A);
}
}
Complexity of Odd-Even MergeSort

Complexity of OddEvenMerge:

- $\Theta(\log n)$ subsequent steps
- each step executed on $\frac{n}{2}$ processors
- total work: $\Theta(n \log n)$

Complexity of Odd-Even MergeSort:

- requires executions of OddEvenMerge on subarrays of lengths $k = 2, 4, \ldots, n$
- each OddEvenMerge step requires $\Theta(\log k)$ steps
- number of subsequent steps:
  \[
  \log 2 + \log 4 + \cdots + \log n = \Theta \left((\log n)^2\right)
  \]
- total work: $\Theta \left(n(\log n)^2\right)$