Distributed Algorithms and Optimization – Parallel Algorithms

Qin Liu

The Chinese University of Hong Kong

Background

- Parallel and distributed computing is necessary to handle big data

Background

- Parallel and distributed computing is necessary to handle big data
  - Parallel: pthread, thread pool pattern, OpenMP

Background

- Parallel and distributed computing is necessary to handle big data
  - Parallel: pthread, thread pool pattern, OpenMP
  - Distributed: MPI, Hadoop, GraphLab, Spark

Background

- Parallel and distributed computing is necessary to handle big data
  - Parallel: pthread, thread pool pattern, OpenMP
  - Distributed: MPI, Hadoop, GraphLab, Spark
- How to design efficient algorithms in parallel and distributed computing?

Background

- Parallel and distributed computing is necessary to handle big data
  - Parallel: pthread, thread pool pattern, OpenMP
  - Distributed: MPI, Hadoop, GraphLab, Spark
- How to design efficient algorithms in parallel and distributed computing?
- Today, start with **parallel computing** on a single machine with multiple processors and shared Random Access Memory

Background

- Parallel and distributed computing is necessary to handle big data
  - Parallel: pthread, thread pool pattern, OpenMP
  - Distributed: MPI, Hadoop, GraphLab, Spark
- How to design efficient algorithms in parallel and distributed computing?
- Today, start with **parallel computing** on a single machine with multiple processors and shared Random Access Memory
  - Distributed computing involves network communication overhead

Why focus on parallel algorithms?

- The growth CPU clock-speed slowed down after it has reached around 3 Gigahertz

Why focus on parallel algorithms?

- The growth CPU clock-speed slowed down after it has reached around 3 Gigahertz
- Now, machines have many cores

Why focus on parallel algorithms?

- The growth CPU clock-speed slowed down after it has reached around 3 Gigahertz
- Now, machines have many cores
- Because computation has shifted from sequential to parallel, our algorithms have to change as well...

Outline

Fundamentals of Parallel Algorithm Analysis

The Master Theorem

Matrix Multiplication: Strassen's algorithm

Mergesort

Fundamentals of Parallel Algorithm Analysis

Efficiency of Parallel Algorithms

- Sequential RAM Model: one processor and one memory module

Efficiency of Parallel Algorithms

- Sequential RAM Model: one processor and one memory module
  - use the number of operations to estimate the wall-clock compute time: e.g., quick sort takes $O(n \log n)$, bubble sort takes $O(n^2)$

Efficiency of Parallel Algorithms

- Sequential RAM Model: one processor and one memory module
  - use the number of operations to estimate the wall-clock compute time: e.g., quick sort takes $O(n \log n)$, bubble sort takes $O(n^2)$
- Parallel RAM Model: multiple processors interact with the memory module(s)

Efficiency of Parallel Algorithms

- Sequential RAM Model: one processor and one memory module
  - use the number of operations to estimate the wall-clock compute time: e.g., quick sort takes $O(n \log n)$, bubble sort takes $O(n^2)$
- Parallel RAM Model: multiple processors interact with the memory module(s)
  - usually suppose concurrent read and exclusive write

Caveats of PRAM

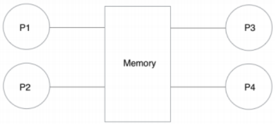
Many variants for an algorithm designer to consider:

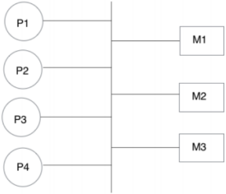
- Resolving concurrent writes: Garbage, Arbitrary, Priority, Combination

Caveats of PRAM

Many variants for an algorithm designer to consider:

- Resolving concurrent writes: Garbage, Arbitrary, Priority, Combination
- Memory speed

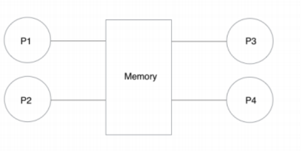


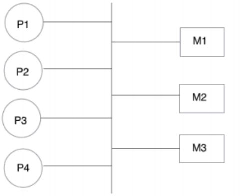


Caveats of PRAM

Many variants for an algorithm designer to consider:

- Resolving concurrent writes: Garbage, Arbitrary, Priority, Combination
- Memory speed





  - layers of cache in between each processor and memory module which all have different read-write speeds

Notations

Let

$T_1 =$ amount of (wall-clock) time algorithm takes on one processor

$T_p =$ amount of (wall-clock) time algorithm takes on p processors

Notations

Let

$T_1 =$ amount of (wall-clock) time algorithm takes on one processor

$T_p =$ amount of (wall-clock) time algorithm takes on p processors

Lower bound of T_p:

$$\frac{T_1}{p} \leq T_p$$

Work and Depth

Work:

- Defined as the number of operations required

Work and Depth

Work:

- Defined as the number of operations required
- Enough for sequential models, but no longer holds in a parallel model

Work and Depth

Work:

- Defined as the number of operations required
- Enough for sequential models, but no longer holds in a parallel model

Depth:

- What if we have infinitude of processors? Would the compute time for an algorithm then be zero?

Work and Depth

Work:

- Defined as the number of operations required
- Enough for sequential models, but no longer holds in a parallel model

Depth:

- What if we have infinitude of processors? Would the compute time for an algorithm then be zero?
- No, because algorithms usually have an inherently sequential component to them

Work and Depth

Work:

- Defined as the number of operations required
- Enough for sequential models, but no longer holds in a parallel model

Depth:

- What if we have infinitude of processors? Would the compute time for an algorithm then be zero?
- No, because algorithms usually have an inherently sequential component to them
- Known as depth

Representing Algorithms as DAG's

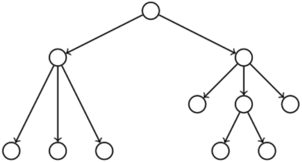


Figure: Example DAG

Represent the dependencies between operations in an algorithm using a directed acyclic graph (DAG)

- each fundamental unit of computation is represented by a node

Representing Algorithms as DAG's

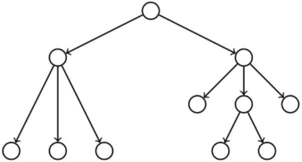


Figure: Example DAG

Represent the dependencies between operations in an algorithm using a directed acyclic graph (DAG)

- each fundamental unit of computation is represented by a node
- an edge from node u to node v means if computation of v is required as an input to computation u

Representing Algorithms as DAG's

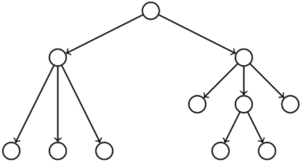


Figure: Example DAG

Represent the dependencies between operations in an algorithm using a directed acyclic graph (DAG)

- each fundamental unit of computation is represented by a node
- an edge from node u to node v means if computation of v is required as an input to computation u
- it's actually a tree

Representing Algorithms as DAG's

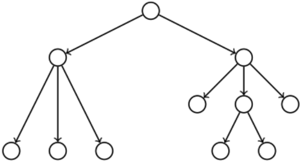


Figure: Example DAG

Represent the dependencies between operations in an algorithm using a directed acyclic graph (DAG)

- each fundamental unit of computation is represented by a node
- an edge from node u to node v means if computation of v is required as an input to computation u
- it's actually a tree
- the root represents the output of our algorithm

Representing Algorithms as DAG's (cont'd)

Define work to be

$$T_1 = \text{number of nodes in DAG}$$

Representing Algorithms as DAG's (cont'd)

Define work to be

$$T_1 = \text{number of nodes in DAG}$$

With an infinitude of processors, the compute time is given by the depth of the tree

$$T_\infty = \text{depth of computation DAG}$$

Brent's theorem

Theorem (Brent's theorem)

We claim

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

The depth T_∞ measures how far off our algorithm performs relative to the best possible version or how parallel an algorithm is.

Proof of Brent's theorem

Proof: On level i of our DAG, there are m_i operations. Hence by that definition, since T_1 is the total work of our algorithm,

$$T_1 = \sum_{i=1}^{n} m_i$$

where we denote $T_\infty = n$. For each level i of the DAG, the time taken by p processors is given as

$$T_p^i = \left\lceil \frac{m_i}{p} \right\rceil \leq \frac{m_i}{p} + 1.$$

This equality follows from the fact that there are m_i constant-time operations to be performed at m_i, and once all lower levels have been completed, these operations share no inter-dependencies. Thus, we may distribute or assign operations uniformly to our processors. The ceiling follows from the fact that if the number of processors not divisible by p, we require exactly one wall-clock cycle where some but not all processors are used in parallel. Then,

$$T_p = \sum_{i=1}^{n} T_p^i \leq \sum_{i=1}^{n} \left(\frac{m_i}{p} + 1 \right) = \frac{T_1}{p} + T_\infty$$

■

Speed-up

$T_{p,n}$: the run-time on p processors given an input of size n.
The speed-up of a parallel algorithm: $\mathrm{SpeedUp}(p, n) = \frac{T_{1,n}}{T_{p,n}}$

Definition

If $SpeedUp(p, n) = \Theta(p)$, we say the algorithm is <u>strongly scalable</u>.

Definition

If $SpeedUp(p, np) = \frac{T_{1,n}}{T_{p,np}} = \Omega(1)$, we say the algorithm is <u>weakly scalable</u>.

Embarrassingly Parallel

Figure: An Embarrassingly Parallel DAG

- No dependency between operations
- Scalable in the most trivial scence

Example: Summation

```
s ← 0 for i ← 1, 2, ..., n do
    s+ = a[i]
end
return s
```

Figure: Sequential Summation

$$T_1 = T_2 = \cdots = T_\infty = n$$

How to redesign the algorithm?

Parallel Summation

Instead of

$$((a_1 + a_2) + a_3) + a_4$$

We assign each processor a pair of elements

$$(a_1 + a_2) + (a_3 + a_4)$$

Parallel Summation

Instead of

$$((a_1 + a_2) + a_3) + a_4$$

We assign each processor a pair of elements

$$(a_1 + a_2) + (a_3 + a_4)$$

Results in

$$T_\infty = \log_2 n$$

Hence by Brent's theorem

$$T_p \leq \frac{n}{p} + log_2 n$$

Example: Matrix Multiplies

Given two $n \times n$ matrices, A and B, output one $n \times n$ matrix $C = AB$, where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Example: Matrix Multiplies

Given two $n \times n$ matrices, A and B, output one $n \times n$ matrix $C = AB$, where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Work is

$$T_1 = O(n^3)$$

Example: Matrix Multiplies

Given two $n \times n$ matrices, A and B, output one $n \times n$ matrix $C = AB$, where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Work is

$$T_1 = O(n^3)$$

Parallel algorithm:

- Each c_{ij} takes $O(n)$ independently
- $T_p \leq O(\frac{n^3}{p} + n)$
- Parallelize each c_{ij} summation
  - $T_p \leq O(\frac{n^3}{p} + \log n)$

The Master Theorem

Recursive Algorithms

- Easy to parallelize
- The recursive calls don't depend on each other, hence we may assign each to a processor
- How to analyze the time complexity?

Unrolling the Simplest Case

Suppose the recurrence relation is of the form:

$$T(n) = bT(n/b) + n,$$

where b is an integer greater than 1 (e.g., $b = 2$ for quick sort).

$$\begin{aligned}
T(n) &= bT\left(\frac{n}{b}\right) + n && \text{by our definition of } T(n) \\
&= b\left[bT\left(\frac{n}{b^2}\right) + \frac{n}{b}\right] + n && \text{unrolling to the second-level of our tree} \\
&= b^2T\left(\frac{n}{b^2}\right) + b \cdot \frac{n}{b} + n && \text{re-arranging} \\
&= b^2\left[bT\left(\frac{n}{b^3}\right) + \frac{n}{b^2}\right] + b \cdot \frac{n}{b} + n && \text{unrolling to the third level of our tree} \\
&= b^3T\left(\frac{n}{b^3}\right) + b^2 \cdot \frac{n}{b^2} + b \cdot \frac{n}{b} + n && \text{re-arranging} \\
&= \ldots \\
&= n\log_b(n)
\end{aligned}$$

General Form of the Master Theorem

Suppose $T(n) = aT(\frac{n}{b}) + f(n)$

1. If $f(n) \in O(n^c)$ where $c < \log_b a$, then

$$T(n) \in \Theta(n^{\log_b a})$$

2. If for some constant $k \geq 0$, $f(n) \in \Theta(n^c \log^k n)$ where $c = \log_b a$, then

$$T(n) \in \Theta(n^c \log_b^{k+1} n)$$

3. If $f(n) \in O(n^c)$ where $c > \log_b a$ and also

$$af(\frac{n}{b}) \leq kf(n)$$

   for some constant $k < 1$, then

$$T(n) \in \Theta(f(n))$$

Examples

- $T(n) = T(n/b) + 1.$

Examples

- $T(n) = T(n/b) + 1.$
  - Applying case 2, $c = 0, k = 0$

Examples

- $T(n) = T(n/b) + 1$.
  - Applying case 2, $c = 0, k = 0$
  - $T(n) = \log_b n$

Examples

- $T(n) = T(n/b) + 1$.
  - Applying case 2, $c = 0, k = 0$
  - $T(n) = \log_b n$
  - E.g., binary search $O(\log_2 n)$

Examples

- $T(n) = T(n/b) + 1.$
  - Applying case 2, $c = 0, k = 0$
  - $T(n) = \log_b n$
  - E.g., binary search $O(\log_2 n)$
- $T(n) = T(n/b) + n.$

Examples

- $T(n) = T(n/b) + 1.$
  - Applying case 2, $c = 0, k = 0$
  - $T(n) = \log_b n$
  - E.g., binary search $O(\log_2 n)$
- $T(n) = T(n/b) + n.$
  - Applying case 3, $c = 1, k \in (1/b, 1)$

Examples

- $T(n) = T(n/b) + 1.$
  - Applying case 2, $c = 0, k = 0$
  - $T(n) = \log_b n$
  - E.g., binary search $O(\log_2 n)$
- $T(n) = T(n/b) + n.$
  - Applying case 3, $c = 1, k \in (1/b, 1)$
  - $T(n) = \Theta(n)$

Examples

- $T(n) = T(n/b) + 1.$
  - Applying case 2, $c = 0, k = 0$
  - $T(n) = \log_b n$
  - E.g., binary search $O(\log_2 n)$
- $T(n) = T(n/b) + n.$
  - Applying case 3, $c = 1, k \in (1/b, 1)$
  - $T(n) = \Theta(n)$
  - E.g., quick select $O(n)$

General Recursive Relations

Example: $T(n) = T(\sqrt{n}) + 1$, with a base case $T(2) = 1$.
We have to unroll the recurrence by hand to find a solution.

$$\begin{aligned}
T(n) &= T(n^{1/2}) + 1 && \text{our definition of } T(n) \\
&= \Big(T(n^{1/4}) + 1\Big) + 1 && \text{unrolling to second level} \\
&= \Big(\Big(T(n^{1/8}) + 1\Big) + 1\Big) + 1 && \text{unrolling to third level...}
\end{aligned}$$

General Recursive Relations

Example: $T(n) = T(\sqrt{n}) + 1$, with a base case $T(2) = 1$.
We have to unroll the recurrence by hand to find a solution.

$$\begin{aligned}
T(n) &= T(n^{1/2}) + 1 && \text{our definition of } T(n) \\
&= \Big(T(n^{1/4}) + 1\Big) + 1 && \text{unrolling to second level} \\
&= \Big(\Big(T(n^{1/8}) + 1\Big) + 1\Big) + 1 && \text{unrolling to third level...}
\end{aligned}$$

Let k denote the number of recursive calls to reach base case:

$$n^{1/2^k} = 2$$

$$\frac{1}{2^k} \log_2 n = \log_2 2$$

$$k = \log_2 \log_2 n$$

Example: All Prefix Sum

Given

$$A = [a_1, a_2, \ldots, a_n],$$

output

$$R = [r_0, r_1, r_2, \ldots, r_n],$$

where $r_k = \sum_{i=1}^{k} a_i$ and $r_0 = 0$.

Algorithm 2: Prefix Sum

```
Input: All prefix sum for an array A
if size of A is 1 then
    return only element of A
end
Let A' be the sum of adjacent pairs
Compute R' = AllPrefixSum(A') // Note: R' has every other element of R
Fill in missing entries of R' using another n/2 processors
```

Example: All Prefix Sum (cont'd)

Given

$$A = [a_1, a_2, \ldots, a_n],$$

output

$$R = [r_0, r_1, r_2, \ldots, r_n],$$

where $r_k = \sum_{i=1}^{k} a_i$.

Example: All Prefix Sum (cont'd)

Given

$$A = [a_1, a_2, \ldots, a_n],$$

output

$$R = [r_0, r_1, r_2, \ldots, r_n],$$

where $r_k = \sum_{i=1}^{k} a_i$.
Let

$$A' = [a_1 + a_2, a_3 + a_4, \ldots, a_{n-1} + a_n]$$

$$R' = [r_2, r_4, \ldots, r_n]$$

For i is odd

$$r_i = r_{i-1} + a_i$$

Algorithm Analysis

Let $T_1 = W(n)$ and $T_\infty = D(n)$, we have

$$W(n) = W(n/2) + O(n) = O(n)$$

$$D(n) = D(n/2) + O(1) = O(\log n)$$

So

$$T_p \leq O(n/p + \log n)$$

Matrix Multiplication: Strassen's algorithm

Matrix Multiplies

- Naive way: $O(n^3)$
- Strassen's sequential algorithm: $O(n^{\log_2 7}) = O(n^{2.81})$
- Idea – Block Matrix Multiplication

$$\mathbf{C} = \mathbf{AB} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

Block Matrix Multiplication

$$
\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\
\mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}
\end{aligned}
$$

Parallelizing the Algorithm

8 matrix multiplies between matrices of size $n/2 \times n/2$ and 4 matrix additions:

$$W(n) = 8W(n/2) + O(n^2)$$

By the Master Theorem,

$$W(n) = O(n^3)$$

No progress compared to the naive way

Strassen's Algorithm

Reduce the number of sub-calls to matrix-multiplies to 7:

$\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$
$\mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$
$\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2})$
$\mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$
$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2}$
$\mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$
$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})$

Strassen's Algorithm (cont'd)

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$
$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

Strassen's Algorithm (cont'd)

Work:

$$W(n) = 7W(n/2) + O(n^2)$$

By the Master Theorem,

$$W(n) = O(n^{\log_2 7})$$

Strassen's Algorithm (cont'd)

Work:

$$W(n) = 7W(n/2) + O(n^2)$$

By the Master Theorem,

$$W(n) = O(n^{\log_2 7})$$

Since the sub-calls can be parallelized and the depth of matrix-additions is $O(1)$

$$D(n) = D(n/2) + O(1) = O(\log n)$$

Strassen's Algorithm (cont'd)

Work:

$$W(n) = 7W(n/2) + O(n^2)$$

By the Master Theorem,

$$W(n) = O(n^{\log_2 7})$$

Since the sub-calls can be parallelized and the depth of matrix-additions is $O(1)$

$$D(n) = D(n/2) + O(1) = O(\log n)$$

By Brent's theorem

$$T_p \leq O(\frac{n^{2.81}}{p} + \log n)$$

Drawbacks of Divide and Conquer

Communication cost:

- Parallel RAM model assumes zero communication costs between processors
- Realistically, we never have efficient communication
- In clusters of computers, Strassen's algorithm is impratical
- Need to chop up matrices which incurs lots of shuffle cost

Drawbacks of Divide and Conquer (cont'd)

Big $\mathcal{O}$ and big constants:

- In Strassen's algorithm, the $O(n^2)$ term requires $20 \cdot n$ operations
- When the data is distributed across machines, we can only afford to pass it one time
- Big $\mathcal{O}$ notion is good to get started and throw away super bad algorithms
- Sometimes we need to look closely

Mergesort

Mergesort

Algorithm 1: Merge Sort

Input : Array A with n elements

Output: Sorted A

$n \leftarrow |A|$

if n *is 1* **then**

  return A

end

else

  `// (In Parallel)`

  $L \leftarrow$ `MERGESORT(A[0,...,n/2))` `// Indices` $0, 1, \ldots, \frac{n}{2} - 1$

  $R \leftarrow$ `MERGESORT(A[n/2,...,n))` `// Indices` $\frac{n}{2}, \frac{n}{2} + 1, \ldots, n - 1$

  return *MERGE(L,R)*

end

Mergesort (cont'd)

Algorithm 2: Merge

Input : Two sorted arrays A, B each of length n
Output: Merged array C, consisting of elements of A and B in sorted order

```
a ← pointer to head of array A (i.e. pointer to smallest element in A)
b ← pointer to head of array B (i.e. pointer to smallest element in B)
while a, b are not null do
    Compare the value of the element at a with the value of the element at b
    if value(a) < value(b) then
        add value of a to output C
        increment pointer a to next element in A
    end
    else
        add value of b to output C
        increment pointer b to next element in B
    end
end
if elements remaining in either a or (exclusive) b then
    Append these sorted elements to our sorted output C
end
return C
```

Naive Parallelization

We have

$$W(n) = 2W(n/2) + O(n) = O(n \log n)$$

$$D(n) = D(n/2) + O(n) = O(n)$$

Naive Parallelization

We have

$$W(n) = 2W(n/2) + O(n) = O(n \log n)$$

$$D(n) = D(n/2) + O(n) = O(n)$$

Using Brent's theorem

$$T_p \leq O(\frac{n \log n}{p} + n)$$

Naive Parallelization

We have

$$W(n) = 2W(n/2) + O(n) = O(n \log n)$$

$$D(n) = D(n/2) + O(n) = O(n)$$

Using Brent's theorem

$$T_p \leq O(\frac{n \log n}{p} + n)$$

Not much speed-up compared to sequential version.
Bottleneck lies in `merge`.

Improved Parallelization

Merge two sorted array A and B in parallel. Suppose the output is C, for any element x in A or B

$$\mathrm{rank}_C(x) = \mathrm{rank}_A(x) + \mathrm{rank}_B(x)$$

Algorithm 3: Parallel Merge

Input : Two sorted arrays A, B each of length n

Output: Merged array C, consisting of elements of A and B in sorted order

for *each* $a \in A$ **do**

  Do a binary search to find where a would be added into B,

  The final rank of a given by $\mathrm{rank}_C(a) = \mathrm{rank}_A(a) + \mathrm{rank}_B(a)$.

end

n parallel binary searches, each takes $O(\log n/2) = O(\log n)$ time.

Improved Parallelization (cont'd)

We have

$$W(n) = 2W(n/2) + O(n \log n) = O(n \log^2 n)$$

$$D(n) = D(n/2) + O(\log n) = O(\log^2 n)$$

By Brent's theorem

$$T_p \leq O(\frac{n \log^2 n}{p} + \log^2 n)$$

Best known algorithm by Richard Cole:

$$T_p \leq O(\frac{n \log n}{p} + \log n)$$

References

- CME 323: Distributed Algorithms and Optimization by Reza Zadeh (Stanford)
- Parallel Algorithms by Guy E. Blelloch and Bruce M. Maggs
- Introduction to Algorithms by Cormen, Leiserson, Rivest, Stein