Parallel Algorithms & the PRAM Model

Advanced Topics Spring 2009
Prof. Robert van Engelen
Overview

- The PRAM model of parallel computation
- Simulations between PRAM models
- Work-time presentation framework of parallel algorithms
- Design and analysis of parallel algorithms
The PRAM Model of Parallel Computation

- Parallel Random Access Machine (PRAM)
- Natural extension of RAM: each processor is a RAM
- Processors operate synchronously
- Earliest and best-known model of parallel computation

Shared Memory

$P_1 \quad P_2 \quad P_3 \quad \cdots \quad P_p$

$p$ processors, each with private memory

All processors operate synchronously, by executing load, store, and operations on data

Shared memory with $m$ locations
Synchronous PRAM versus Asynchronous PRAM

- The **synchronous PRAM** model has a similarity with data-parallel execution on a SIMD machine
  - All processors execute the same program
  - All processors execute the same PRAM step instruction stream in "lock-step"
  - Effect of operation depends on local data
  - Instructions can be selectively disabled (for if-then-else flow)

- The **asynchronous PRAM** model
  - Several competing models
  - No lock-step
Classification of PRAM Model

- A PRAM step ("clock cycle") consists of three phases
  1. *Read*: each processor may read a value from shared memory
  2. *Compute*: each processor may perform operations on local data
  3. *Write*: each processor may write a value to shared memory

- Model is refined for concurrent read/write capability
  - Exclusive Read Exclusive Write (EREW)
  - Concurrent Read Exclusive Write (CREW)
  - Concurrent Read Concurrent Write (CRCW)

- CRCW PRAM: what to do with concurrent writes?
  - Common CRCW: all processors must write the same value
  - Arbitrary CRCW: one of the processors succeeds in writing
  - Priority CRCW: processor with highest priority succeeds in writing
Comparison of PRAM Models

- A model $A$ is less powerful compared to model $B$ if either
  - The time complexity is asymptotically less in model $B$ for solving a problem compared to $A$
  - Or the time complexity is the same and the work complexity is asymptotically less in model $B$ compared to $A$

- From weakest to strongest:
  - EREW
  - CREW
  - Common CRCW
  - Arbitrary CRCW
  - Priority CRCW
Simulations Between PRAM Models

- An algorithm designed for a weaker model can be executed within the same time complexity and work complexity on a stronger model.

- An algorithm designed for a stronger model can be simulated on a weaker model, either with
  - Asymptotically more processors (or more work by the same number of processors)
  - Or asymptotically more time
Simulating a Priority CRCW on an EREW PRAM

Theorem: An algorithm that runs in $T$ time on the $p$-processor priority CRCW PRAM can be simulated by EREW PRAM to run in $O(T \log p)$ time

- A concurrent read or write of an $p$-processor CRCW PRAM can be implemented on a $p$-processor EREW PRAM to execute in $O(\log p)$ time
- $Q_1, \ldots, Q_p$ CRCW processors, such that $Q_i$ has to read (write) $M[j_i]$
- $P_1, \ldots, P_p$ EREW processors
- $M_1, \ldots, M_p$ denote shared memory locations for special use
- $P_i$ stores $<j_i, i>$ in $M_i$
- Sort pairs in lexicographically non-decreasing order in $O(\log p)$ time using EREW merge sort algorithm
- Pick representative from each block of pairs that have same first component in $O(1)$ time
- Representative $P_i$ reads (writes) from $M[k]$ with $<k, \_>$ in $M_i$ and copies data to each $M$ in the block in $O(\log p)$ time using EREW segmented parallel prefix algorithm
- $P_i$ reads data from $M_i$
Example 1: Reduction on the EREW PRAM

- Reduce (sum) $p$ values on the $p$-processor EREW PRAM in $O(\log p)$ time
- Reduction algorithm uses exclusive reads and writes
- Algorithm is the basis of other EREW algorithms
Example 1

Sum of \( n \) values using \( n \) processors (i)
Each processor \( i, 1 \leq i \leq n \), executes:

**Input:** \( A[1,\ldots,n], \ n = 2^k \)

**Output:** sum \( S = \sum_{j=1..n} A[j] \)

begin

\( B[i] := A[i] \)

for \( h = 1 \) to \( \log n \) do

if \( i \leq n/2^h \) then


if \( i = 1 \) then

\( S := B[i] \)

end

How much time?

How many operations?
Example 2: Matrix Multiply on the CREW PRAM

- Consider $n \times n$ matrix multiplication $C = AB$ using $n^3$ processors

- Each element of $C$

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

can be computed on the CREW PRAM in parallel using $n$ processors in $O(\log n)$ time

- All $c_{ij}$ can be computed using $n^3$ processors in $O(\log n)$ time
Example 2

Matrix multiply with $n^3$ processors $(i,j,l)$
Each processor $(i,j,l)$ executes:

**Input:** $n \times n$ matrices $A$ and $B$, $n = 2^k$

**Output:** $C = A B$

begin


for $h = 1$ to $\log n$ do

if $i \leq n/2^h$ then

$C'[i,j,l] := C'[i,j,2l-1] + C'[i,j,2l]$

if $l = 1$ then

$C[i,j] := C'[i,j,1]$

end

end

$O(\log n)$ time

How many operations?
Example 2: CREW versus EREW PRAM

- Algorithm on the CREW PRAM requires $O(\log n)$ time and $O(n^3)$ operations ($n^2$ processors perform $O(n)$ ops)

- On the EREW PRAM, the exclusive reads of $a_{ij}$ and $b_{ij}$ values can be satisfied by making $n$ copies of $a$ and $b$, which takes $O(\log n)$ time with $n$ processors (broadcast tree)

- Total time is still $O(\log n)$

- But requires more work and total memory requirement is huge!
The WT Scheduling Principle

- The work-time (WT) scheduling principle schedules $p$ processors to execute an algorithm
  - Algorithm has $T(n)$ time steps and $W(n)$ total operations
  - A time step can be parallel, i.e. pardo
- We can adapt the algorithm to run on the $p$-processor PRAM in $\leq \lceil W(n)/p \rceil + T(n)$ steps

Proof
- Let $W_i(n)$ be the number of operations (work) performed in time unit $i$, $1 \leq i \leq T(n)$
- Simulate each set of $W_i(n)$ operations in $\lceil W_i(n)/p \rceil$ parallel steps, for each $1 \leq i \leq T(n)$
- The number of steps on the $p$-processor PRAM takes
  $\Sigma_i \lceil W_i(n)/p \rceil \leq \Sigma_i (\lceil W_i(n)/p \rceil + 1) \leq \lceil W(n)/p \rceil + T(n)$
Work-Time Presentation

- The WT presentation can be used to determine the time and operation requirements of an algorithm

- The upper-level WT presentation framework describes the algorithm in terms of a sequence of time units
  - From which we can determine $T(n)$ and $W(n)$

- The lower-level follows the WT scheduling principle
  - $p$-processor PRAM requires $\leq \lceil W(n)/p \rceil + T(n)$ steps
Example 1 Revisited: WT Presentation

Input: \( A[1,\ldots,n] \), \( n = 2^k \)
Output: \( \text{sum } S = \sum_{j=1}^{n} A[j] \)

\[
\text{begin}
\text{for } 1 \leq i \leq n \text{ pardo}
B[i] := A[i]
\text{for } h = 1 \text{ to } \log n \text{ do}
\text{for } 1 \leq i \leq n/2^h \text{ pardo}
\text{if } i = 1 \text{ then}
S := B[1]
\text{end}
\]

Do you spot any concurrent reads?
concurrent writes?

\( T(n) = O(\log n) \)
\( W(n) = O(n) \)

WT scheduling principle:
total time \( \leq O(n/p + \log n) \)
Example 2 Revisited: WT-Presentation

Input: \( n \times n \) matrices \( A \) and \( B \), \( n = 2^k \)

Output: \( C = A B \)

begin

\[
\text{for } 1 \leq i, j, l \leq n \text{ pardo} \\
\]

\[
\text{for } h = 1 \text{ to } \log n \text{ do} \\
\text{for } 1 \leq i, j \leq n, 1 \leq l \leq n/2^h \text{ pardo} \\
C'[i,j,l] := C'[i,j,2l-1] + C'[i,j,2l]
\]

\[
\text{for } 1 \leq i, j \leq n \text{ pardo} \\
C[i,j] := C'[i,j,1]
\]

end

\[
T(n) = O(\log n) \\
W(n) = n^3
\]

WT scheduling principle:
total time \( \leq O(n^3/p + \log n) \)
Example 3: PRAM Recursive Prefix Sum Algorithm

Input: Array of \((x_1, x_2, \ldots, x_n)\) elements, \(n = 2^k\)

Output: Prefix sums \(s_i, 1 \leq i \leq n\)

begin
  if \(n = 1\) then \(s_1 = x_1\); exit
  for \(1 \leq i \leq n/2\) pardo
    \[y_i := x_{2i-1} + x_{2i}\]
    Recursively compute prefix sums of \(y\) and store in \(z\)
  for \(1 \leq i \leq n\) pardo
    if \(i\) is even then \(s_i := z_i/2\)
    if \(i > 1\) is odd then \(s_i := z_{(i-1)/2} + x_i\)
    if \(i = 1\) then \(s_1 := x_1\)
end
Proof of Work Optimality

- **Theorem**: The PRAM prefix sum algorithm correctly computes the prefix sum and takes \( T(n) = O(\log n) \) time using a total of \( W(n) = O(n) \) operations.

- **Proof** by induction on \( k \), where input size \( n = 2^k \)
  - Base case \( k = 0 \): \( s_1 = x_1 \)
  - Assume correct for \( n = 2^k \)
  - For \( n = 2^{k+1} \)
    - For all \( 1 \leq j \leq n/2 \) we have
      \[ z_j = y_1 + y_2 + \ldots + y_j = (x_1 + x_2) + (x_3 + x_4) + \ldots + (x_{2j-1} + x_{2j}) \]
    - Hence, for \( i = 2j \leq n \) we have \( s_i = s_{2j} = z_j = z_{i/2} \)
    - And \( i = 2j+1 \leq n \) we have \( s_i = s_{2j+1} = s_{2j} + x_{2j+1} = z_j + x_{2j+1} = z_{(i-1)/2} + x_i \)
  - \( T(n) = T(n/2) + a \) \( \Rightarrow T(n) = O(\log n) \)
  - \( W(n) = W(n/2) + bn \) \( \Rightarrow W(n) = O(n) \)
PRAM Nonrecursive Prefix Sum

**Input:** Array $A$ of size $n = 2^k$

**Output:** Prefix sums in $C[0,j]$, $1 \leq j \leq n$

begin

for $1 \leq j \leq n$ pardo

$B[0,j] := A[j]$

for $h = 1$ to $\log n$ do

for $1 \leq j \leq n/2^h$ pardo

$B[h,j] := B[h-1,2j-1] + B[h-1,2j]$

end

for $h = \log n$ to 0 do

for $1 \leq j \leq n/2^h$ pardo

if $j$ is even then $C[h,j] := C[h+1,j/2]$

else if $i = 1$ then $C[h,1] := B[h,1]$

else $C[h,j] := C[h+1,(j-1)/2] + B[h,j]$

end
First Pass: Bottom-Up

\[
B[3,j] = 
\begin{bmatrix}
5 & 3 & -6 & 2 & 7 & 10 & -2 & 8
\end{bmatrix}
\]

\[
B[2,j] = 
\begin{bmatrix}
8 & -4 & 17 & 6
\end{bmatrix}
\]

\[
B[1,j] = 
\begin{bmatrix}
4 & 23 & 27 & 1
\end{bmatrix}
\]

\[
B[0,j] = 
\begin{bmatrix}
27
\end{bmatrix}
\]
Second Pass: Top-Down

\[
\begin{align*}
B[3,j] &= 27 \\
C[3,j] &= 27 \\
B[2,j] &= 4 & 23 \\
C[2,j] &= 4 & 27 \\
B[1,j] &= 8 & -4 & 17 & 6 \\
C[1,j] &= 8 & 4 & 21 & 27 \\
B[0,j] &= 5 & 3 & -6 & 2 & 7 & 10 & -2 & 8 \\
C[0,j] &= 5 & 8 & 2 & 4 & 11 & 21 & 19 & 27 \\
A[j] &= 5 & 3 & -6 & 2 & 7 & 10 & -2 & 8
\end{align*}
\]
Example 4: Pointer Jumping

- Finding the roots of a forest using pointer-jumping
**Pointer Jumping on the CREW PRAM**

**Input:** A forest of trees, each with a self-loop at its root, consisting of arcs \((i, P(i))\) and nodes \(i\), where \(1 \leq i \leq n\)

**Output:** For each node \(i\), the root \(S[i]\)

```plaintext
begin
  for \(1 \leq i \leq n\) pardo
    \(S[i] := P[i]\)
    while \(S[i] \neq S[S[i]]\) do
      \(S[i] := S[S[i]]\)
end
```

\(T(n) = O(\log h)\) with \(h\) the maximum height of trees

\(W(n) = O(n \log h)\)
PRAM Model Summary

- PRAM removes algorithmic details concerning synchronization and communication, allowing the algorithm designer to focus on problem properties.
- A PRAM algorithm includes an explicit understanding of the operations performed at each time unit and an explicit allocation of processors to jobs at each time unit.
- PRAM design paradigms have turned out to be robust and have been mapped efficiently onto many other parallel models and even network models.
  - A SIMD network model considers communication diameter, bisection width, and scalability properties of the network topology of a parallel machine such as a mesh or hypercube.
Design and Analysis of Parallel Algorithms

- Arithmetic problems:
  - Polynomial evaluation: first-order linear recurrence
  - Polynomial multiplication: FFT
  - Lagrange interpolation

- Planar geometry:
  - The convex hull problem revisited: constant-time computation of the upper common tangent
First-Order Linear Recurrences

- Consider the first-order linear recurrence:
  \[ y_1 = b_1 \]
  \[ y_i = a_i y_{i-1} + b_i \quad \text{for } 2 \leq i \leq n \]

- At first sight this seems impossible to parallelize, at least in its current form.

- However, note that the prefix sum
  \[ y_i = \sum_{j=1}^{i} b_j \]
  is a special case of a first-order linear recurrence where
  \[ a_i = 1 \] (the multiplicative unit element)

- We know how to parallelize the prefix sum.
Divide and Conquer
Parallelization

- Rewrite $y_i = a_i y_{i-1} + b_i$ into $y_i = a_i (a_{i-1} y_{i-2} + b_{i-1}) + b_i$
- This equation defines a linear recurrence of size $n/2$ for even index $i$
  
  \[
  z_1 = b_1', \\
  z_i = a_i' z_{i-1} + b_i', \quad 2 \leq i \leq n/2
  \]

1. Let

\[
  a_i' = a_{2i} a_{2i-1} \\
  b_i' = a_{2i} b_{2i-1} + b_{2i}
\]

2. Solve $z_i$ recursively

3. For $1 \leq i \leq n$ set

\[
  y_i = z_{i/2} \quad \text{if } i \text{ is even} \\
  y_i = a_i z_{(i-1)/2} + b_i \quad \text{if } i \text{ is odd} > 1 \\
  y_i = b_1 \quad \text{if } i = 1
\]
First-Order Linear Recurrence

Input: Arrays $B = (b_1, b_2, \ldots, b_n)$ and $A = (a_1 = 0, a_2, \ldots, a_n)$, $n = 2^k$

Output: The $y_i$ values such that $y_i = a_i y_{i-1} + b_i$

begin

if $n = 1$ then $y_1 := b_1$; exit

for $1 \leq i \leq n/2$ pardo

$a_i' := a_{2i} a_{2i-1}$

$b_i' := a_{2i} b_{2i-1} + b_{2i}$

Recursively solve the recurrence $z_i$ defined by

$z_1 = b_1'$ and $z_i = a_i' z_{i-1} + b_i'$ for $2 \leq i \leq n/2$

for $1 \leq i \leq n$ pardo

if $i$ is even then $y_i := z_{i/2}$

if $i > 1$ is odd then $y_i := a_i z_{(i-1)/2} + b_i$

if $i = 1$ then $y_1 := b_1$

end
Parallel Time and Work

- From the algorithm we observe
  - $T(n) = T(n/2) + O(1)$ therefore total parallel time $T(n) = O(\log n)$
  - $W(n) = W(n/2) + O(n)$ therefore total operations $W(n) = O(n)$
Polynomial Evaluation

- We wish to evaluate the polynomial

\[ p(x) = b_1x^{n-1} + b_2x^{n-2} + b_3x^{n-3} + \ldots + b_n \]

- Two steps:
  1. Use prefix sum
     - Compute the \( x^{n-i} = [1, x, x^2, x^3, \ldots, x^{n-1}] \) concurrently for all \( i \), which takes \( O(\log n) \) time and \( O(n) \) work
  2. Use a tree reduction to compute the sum
     - Parallel sum \( b_i x^{n-i} \) takes \( O(\log n) \) parallel time and \( O(n) \) work
Polynomial Evaluation (cont’d)

- We wish to evaluate the polynomial

\[ p(x) = b_1 x^{n-1} + b_2 x^{n-2} + b_3 x^{n-3} + \ldots + b_n \]

- Horner’s rule

\[ p(x) = (((b_1 x + b_2) x + b_3) x + \ldots + b_{n-1}) x + b_n \]

gives a first-order linear recurrence with \( a_i = x \)

- Takes \( O(\log n) \) total parallel time with \( O(n) \) total operations
Polynomial Multiplication

Consider the polynomials

\[ p(x) = a_0x^{n-1} + a_1x^{n-2} + a_2x^{n-3} + \ldots + a_{n-1} \]
\[ q(x) = b_0x^{m-1} + b_1x^{m-2} + b_2x^{m-3} + \ldots + b_{m-1} \]

We wish to compute the product

\[ r(x) = p(x)q(x) = c_0x^{n+m-2} + c_1x^{n+m-3} + c_2x^{n+m-4} + \ldots + c_{n+m-2} \]

where

\[ c_k = \sum_{j=0}^{k} a_j b_{k-j} \]

(we take \( a_j = 0 \) for \( j \geq n \) and \( b_{k-j} = 0 \) for \( k-j \geq m \))
Polynomial Multiplication (cont’d)

- We can compute all $c_k = \sum_{j=0..k} a_j b_{k-j}$ in $O(\log (n+m))$ parallel time
  - Takes $O((n+m-1)^2/2) = O(nm)$ operations
Polynomial Multiplication & FFT

- Convolution theorem: polynomial multiplication with FFT
  - $O(\log(n+m))$ parallel time, with a simple use of the FFT algorithm reduces the total number of operations to $O((n+m) \log(n+m))$
  - The FFT of the coefficients $a_i$ of $p$ and $a_j$ of $q$ gives the values of the product polynomial $r(\omega^j) = p(\omega^j)q(\omega^j)$ at the distinct roots of unity $\omega^j$

\[
\begin{align*}
[a_0, a_1, \ldots, a_{n-1}] & \quad \text{DFT} \quad \begin{array}{c}
[b_0, b_1, \ldots, b_{n-1}] \quad \text{DFT} \\
[p(1), p(\omega), \ldots, p(\omega^{n-1})] & \quad \text{point-wise} \quad [q(1), q(\omega), \ldots, q(\omega^{m-1})] \\
& \quad \text{DFT}^{-1} \quad \\
& \quad [c_0, c_1, \ldots, c_{n+m-2}] 
\end{array}
\end{align*}
\]
Polynomial Multiplication & FFT

**Input:** Polynomial coeff. \( a = (a_0, a_1, \ldots, a_{n-1}) \) and \( b = (b_0, b_1, \ldots, b_{m-1}) \)

**Output:** \( c = (c_0, c_1, \ldots, c_{n+m-2}) \) such that \( c_k = \sum_{j=0}^{k} a_j b_{k-j} \)

begin
1. Find integer \( l = 2^s \) such that \( n + m - 2 < l \leq 2(n + m - 2) \)
2. Use FFT to compute \( y = \text{DFT}_l(a) \) and \( z = \text{DFT}_l(b) \)
3. Compute \( u_j = y_j z_j \) for all \( j = 0, \ldots, l-1 \)
4. Use FFT\(^{-1} \) to compute \( c = \text{DFT}_{l}^{-1}(u) \) giving \( c = (c_0, c_1, \ldots, c_{l-1}) \)

end

Steps 2, 4 take \( O(\log (n + m)) \) parallel time and \( O((n + m) \log (n + m)) \) operations

Step 3 takes \( O(1) \) parallel time and \( O(n + m) \) total operations

Note: \( a, b, \) and \( c \) vectors are implicitly padded with 0s, e.g. \( a_i = 0 \) for all \( i \geq n \)
Parallel FFT

- The FFT is easily parallelizable, since the fast sequential algorithm reduces the $O(n^2)$ problem into a $O(n \log n)$ problem using a divide-and-conquer strategy.
Parallel FFT

Input: \( x = (x_0, x_1, \ldots, x_{n-1}) \), \( n = 2^k \), \( \omega = e^{i2\pi/n} \), where \( i = \sqrt{-1} \)
Output: \( y = \text{DFT}_n(x) \)
begin
1. if \( n = 2 \) then
   \( y_1 := x_1 + x_2 \); \( y_2 := x_1 - x_2 \); exit
2. for \( 0 \leq j \leq n/2 - 1 \) pardo
   \( u_j := x_j + x_{n/2+j} \)
   \( v_j := \omega^j (x_j - x_{n/2+j}) \)
3. Recursively compute \( z := \text{DFT}_{n/2}(u) \) and \( z' := \text{DFT}_{n/2}(v) \)
4. for \( 0 \leq j \leq n - 1 \) pardo
   if \( j \) is even then \( y_j := z_{j/2} \)
   if \( j \) is odd then \( y_j := z'_{(j-1)/2} \)
end
Lagrange Interpolation

- Given a set of \( n \) points \( \{(\alpha_j, \beta_j)\}_{j=0..n-1} \) determine the polynomial \( p \) of degree \( n - 1 \) such that for all \( j = 0, \ldots, n-1 \)

\[
p(\alpha_j) = \beta_j
\]

- Lagrange interpolation specifies \( p \) as follows

\[
p(x) = \sum_{j=0}^{n-1} \beta_j \frac{\prod_{l=0, l\neq j}^{n-1} (x - \alpha_j)}{\prod_{l=0, l\neq j}^{n-1} (\alpha_j - \alpha_l)}
\]
Lagrange Interpolation (cont’d)

- Divide-and-conquer strategy: rearrange terms
- Define
  \[ q_l = x - \alpha_l \]
  and
  \[ Q(x) = \prod_{l=0}^{n-1} q_l = \prod_{l=0}^{n-1} (x - \alpha_j) \]
  then the derivative of \( Q \) at point \( \alpha_j \) is
  \[ Q'(\alpha_j) = \prod_{l=0, l \neq j}^{n-1} (\alpha_j - \alpha_l) \]
  which can be evaluated \( \gamma_j = Q'(\alpha_j) \), \( c_j = \beta_j/\gamma_j \) giving
  \[
p(x) = \sum_{j=0}^{n-1} \beta_j \frac{Q(x)/(x - \alpha_j)}{Q'(\alpha_j)} = Q(x) \sum_{j=0}^{n-1} \frac{c_j}{x - \alpha_j}
\]
Lagrange Interpolation (cont’d)

- A balanced tree can be used to compute the sum in

\[ p(x) = Q(x) \sum_{j=0}^{n-1} \frac{c_j}{x - \alpha_j} \]

and use FFT-based polynomial multiplication

- There is another way: note that

\[
\frac{p(x)}{Q(x)} = \frac{p_{k-1,0}(x)}{Q_{k-1,0}(x)} + \frac{p_{k-1,1}(x)}{Q_{k-1,1}(x)} = \frac{p_{k-1,0}(x)Q_{k-1,1}(x) + p_{k-1,1}(x)Q_{k-1,0}(x)}{Q(x)}
\]

where

\[
p_{k-1,0}(x) = Q_{k-1,0} \sum_{j=0}^{n/2-1} \frac{c_j}{x - \alpha_j} \quad Q_{k-1,0}(x) = \prod_{j=0}^{n/2-1} q_l(x)
\]

\[
p_{k-1,1}(x) = Q_{k-1,1} \sum_{j=n/2}^{n-1} \frac{c_j}{x - \alpha_j} \quad Q_{k-1,1}(x) = \prod_{j=n/2}^{n-1} q_l(x)
\]
Lagrange Interpolation (cont’d)

**Input:** Set of pairs \((\alpha_j, \beta_j)\) for \(j = 0, \ldots, n - 1, n = 2^k\)

**Output:** The \(n\) coefficients of \(p(x) = p_{k,0}(x)\) such that \(p(\alpha_j) = \beta_j\)

begin

1. **for** \(0 \leq j \leq n - 1\) **pardo**
   
   \[ Q_{0,j}(x) := x - \alpha_j \]

2. **for** \(h = 1\) **to** \(\log n\) **do**
   
   **for** \(0 \leq j \leq n/2^h - 1\) **pardo**
   
   \[ Q_{h,j}(x) := Q_{h-1,2j}(x) \times Q_{h-1,2j+1}(x) \]

3. Compute \(Q'_{0,j}(x)\) and \(\gamma_j := Q'_{0,j}(\alpha_j)\) for all \(j = 0, \ldots, n - 1\)

4. **for** \(0 \leq j \leq n - 1\) **pardo**
   
   \[ p_{0,j}(x) := \beta_j/\gamma_j \]

5. **for** \(h = 1\) **to** \(\log n\) **do**
   
   **for** \(0 \leq j \leq n/2^h - 1\) **pardo**
   
   \[ p_{h,j}(x) := p_{h-1,2j}(x) \times Q_{h-1,2j+1}(x) + p_{h-1,2j+1}(x) \times Q_{h-1,2j}(x) \]

end

\[ \begin{array}{c|c|c}
\text{T(n)} & \text{W(n)} \\
\hline
O(1) & O(n) \\
\hline
O(\log^2 n) & O(n \log^2 n) \\
\hline
O(\log n) & O(n^2) \\
\hline
O(1) & O(n) \\
\hline
O(\log^2 n) & O(n \log^2 n) \\
\hline
\end{array} \]
Convex Hull Problem Revisited

- The planar convex hull of a set of points $S = \{p_1, p_2, \ldots, p_n\}$ of $p_i = (x, y)$ coordinates is the smallest convex polygon that encompasses all points $S$ on the $x$-$y$ plane.
Convex Hull Problem Revisited

- The upper convex hull spans points \( \{q_1, \ldots, q_s\} \subseteq S \) from point \( q_1 \) with minimum \( x \) to \( q_s \) with maximum \( x \).
- The convex hull = upper convex hull + lower convex hull
- Problem:
  - Given points \( S = \{p_1, \ldots, p_n\} \) such that \( x(p_1) < x(p_2) < \ldots < x(p_n) \), construct the upper convex hull in parallel.
Convex Hull Problem Revisited

- Points $S = \{p_1, \ldots, p_n\}$ may have duplicate x-coordinate values.
- Sort the points $x(p_1) \leq x(p_2) \leq \ldots \leq x(p_n)$ in $O(\log n)$ parallel time and $O(n \log n)$ operations (pipelined merge sort).
- Then, if two or more points have the same $x$ coordinate:
  - Keep the point with the largest $y$ coordinate for the UCH.
  - Keep the point with the smallest $y$ coordinate for the LCH.
- We can now assume that $x(p_1) < x(p_2) < \ldots < x(p_n)$ to compute the UHS (and similarly the LHS).
Convex Hull Problem Revisited

- Parallel convex hull:
  1. Divide the $x$-sorted points $S$ into sets $S_1$ and $S_2$ of equal size
  2. Compute upper convex hull recursively on $S_1$ and $S_2$
  3. Combine $UCH(S_1)$ and $UCH(S_2)$ by computing the upper common tangent to form $UCH(S)$

Upper common tangent $(a,b)$
Convex Hull Problem Revisited

- Base case of recursion: two points, which are returned as \( \text{UCH}(S) \)
- *Revisit the common tangent computation:*
  - The line segment \((a,b)\) can be computed sequentially in \(O(\log n)\) time with \(n = |\text{UCH}(S_1) + \text{UCH}(S_2)|\) using a binary search method
- *And replace with parallel computation:*
  - The line segment \((a,b)\) can be computed in \(O(1)\) parallel time
- Line segments can be implemented as linked list of points, thus \(\text{UCH}(S_1)\) and \(\text{UCH}(S_2)\) can be connected using one pointer change of \(a\) to point to \(b\) in \(O(1)\) time
Intermezzo: Parallel Search

- Let \( X = (x_1, x_2, \ldots, x_n) \) be \( n \) distinct elements from a set \( S \) such that \( x_1 < x_2 < \ldots < x_n \)

- Given \( y \in S \), find the index \( i \) for which \( x_i \leq y < x_{i+1} \) where we added \( x_0 = -\infty \) and \( x_{n+1} = +\infty \)

- Parallel search with \( p \) processors:
  - Split \( X \) in \( p \) segments of (almost) equal length
  - Each processor verifies if \( y \) is in its segment
  - If so, restrict search to the segment containing \( y \) and repeat
Intermezzo: Parallel Search

Objective: search index of 27

Parallel time = $O\left(\frac{\log(n + 1)}{\log(p + 1)}\right)$

$p_2$ found $i=13$
Convex Hull Problem Revisited: Using Parallel Search

- Let $UCH(S_1) = (r_1, \ldots, r_s)$ and $UCH(S_2) = (q_1, \ldots, q_t)$
- We need to determine points $a = r_i$ and $b = q_{j(i)}$ such that all points in $S$ are below the line through points $a$ and $b$
Convex Hull Problem Revisited: Using Parallel Search

- Given a point \( r_i \in \text{UCH}(S_1) \) then for any \( q_k \in \text{UCH}(S_2) \) we can determine in \( O(1) \) sequential time if \( q_k = q_{j(i)} \), or \( q_{j(i)} \) is to the left of \( q_k \) or \( q_{j(i)} \) is to the right of \( q_k \).

- Thus, using parallel search, we can determine for point \( r_i \) the tangent \((r_i, q_{j(i)})\) in \( O(\log t / \log p) \) parallel time using \( p \) processors, where \( t = |\text{UCH}(S_2)| \).
Convex Hull Problem Revisited: Using Parallel Search

- Given a point \( r_i \in \text{UCH}(S_1) \) and \( q_{j(i)} \in \text{UCH}(S_2) \), then we can determine in \( O(1) \) sequential time if \( r_i = a \), or \( a \) is to the left of \( r_i \) or \( a \) is to the right of \( r_i \).

- Thus, using parallel search, we can determine the tangent \((a, b)\) in \( O(\log(st) / \log p) \) parallel time using \( p \) processors, where \( s = |\text{UCH}(S_1)| \) and \( t = |\text{UCH}(S_2)| \).
Convex Hull Problem Revisited: Using Parallel Search

- Take $p = \sqrt{s}\sqrt{t}$ then

$$O\left(\frac{\log (st)}{\log (\sqrt{s}\sqrt{t})}\right) = O\left(\frac{\log (st)}{\frac{1}{2}\log (st)}\right) = O(1)$$

parallel time and

$$O(\sqrt{s}\sqrt{t}) = O(n)$$

1. Choose $\sqrt{s}$ points from $UCH(S_1)$ thereby dividing the set $UCH(S_1)$ into (almost) equal blocks of size $\sqrt{s}$ each

2. Find the $q_{j(k\sqrt{s})}$ for each $r_{k\sqrt{s}}$, $k = 1, \ldots, \sqrt{s}$, using $p = \sqrt{s}\sqrt{t}$ processors in $O(1)$ parallel time

3. Deduce the block $B_k = (r_{k\sqrt{s}+1}, \ldots, r_{(k+1)\sqrt{s}-1})$ that contains $a$

4. For each $r_i$ in block $B_k$, determine $q_{j(i)}$ and search $a = r_i$ using $p = \sqrt{s}\sqrt{t}$ processors in $O(1)$ parallel time

5. Set $b = q_{j(i)}$
Convex Hull Problem Revisited: Putting it Together

- Preprocess the points by sorting in $O(\log n)$ parallel time (pipelined merge sort), such that $x(p_1) \leq x(p_2) \leq \ldots \leq x(p_n)$
- Remove duplicates $x(p_i) = x(p_j)$ (for UCH and LCH)
- Divide-and-conquer:
  1. Split $S$ into $S_1, S_2$ and recursively compute the UCH of $S_1$ and $S_2$
  2. Combine $UCH(S_1)$ and $UCH(S_2)$ by computing the upper common tangent in $O(1)$ time to form $UCH(S)$
- Repeat to compute the LCH
- Parallel time (assuming $p = O(n)$ processors)
  \[ T(n) = T(n/2) + O(1) \]
  gives
  \[ T(n) = O(\log n) \]
Further Reading

- An Introduction to Parallel Algorithms, by J. JaJa, 1992