PARALLEL PROCESSING SYSTEMS

Chapter 6: More Shared-Memory Algorithms

Introduction

- we develop PRAM algorithms for several additional problems
  - Sequential rank-based selection
  - A parallel selection algorithm
  - A selection-based sorting algorithm
  - Alternative sorting algorithms
  - Convex hull of a 2D point set
  - Some implementation aspects

Sequential rank-based selection

- the problem is finding a (the) k^{th} smallest element in a sequence $S = x0 , x 1, . . . , x n -1$ whose elements belong to a linear order
  - Median, maximum, and minimum finding are special cases
  - Clearly, can be solved through sorting
    - Sort the sequence in nondescending order
    - Output the kth element of the sorted list
    - is wasteful
      - requires Ω(n log n) time
  - O(n)-time selection algorithms are available

Sequential rank-based selection

- a recursive linear-time selection algorithm
  - Step 1
    - If body requires constant time, say c0
    - Else body requires linear time in |S|, say c1 |S|
  - Step 2 constitutes a |S|/q selection problem
  - Step 3 takes linear time in |S|, say c3 |S|
  - Step 4 is a selection problem of the size 3|S|/4 in worst case

Sequential rank-based selection algorithm *select*(S, k)

1. if $|S| < q$ {q is a small constant}
   then sort S and return the kth smallest element of S
   else divide S into $|S|/q$ subsequences of size q
   Sort each subsequence and find its median
   Let the $|S|/q$ medians form the sequence T
   endif
2. $m = select(T, |T|/2)$ {find the median m of the $|S|/q$ medians}
3. Create 3 subsequences
   L: Elements of S that are $< m$
   E: Elements of S that are $= m$
   G: Elements of S that are $> m$
4. if $|L| \geq k$
   then return $select(L, k)$
   else if $|L| + |E| \geq k$
   then return m
   else return $select(G, k - |L| - |E|)$
   endif

Sequential rank-based selection

- a recursive linear-time selection algorithm
  - Total running time
    - $T(n) = T(n/q)+T(3n/4)+cn$
    - has a linear solution for any $q > 4$
    - E.g., $q=5$
      - $T(n) = 20cn$

Sequential rank-based selection algorithm *select*(S, k)

1. if $|S| < q$ $\{q$ is a small constant$\}$
   then sort S and return the kth smallest element of S
   else divide S into $|S|/q$ subsequences of size q
   Sort each subsequence and find its median
   Let the $|S|/q$ medians form the sequence T
   endif
2. $m = select(T, |T|/2)$ {find the median m of the $|S|/q$ medians}
3. Create 3 subsequences
   L: Elements of S that are $< m$
   E: Elements of S that are $= m$
   G: Elements of S that are $> m$
4. if $|L| \geq k$
   then return $select(L, k)$
   else if $|L| + |E| \geq k$
   then return m
   else return $select(G, k - |L| - |E|)$
   endif

Sequential rank-based selection

- a recursive linear-time selection algorithm
  - analysis to justify the term T(3n/4)
    - The median m of the n/q medians is no larger than at least half, or (n/q)/2, of the medians
    - each of which is in turn no larger than q/2 elements of the original input list S.
    - Thus, m is guaranteed to be no larger than at least $((n/q)/2) \times q/2 = n/4$ elements of the input list S

Sequential rank-based selection

- a recursive linear-time selection algorithm
  - example with n = 25 and q = 5

n/q sublists of q elements

S	6 4 5 6 7	1 5 3 8 2	1 0 3 4 5	6 2 1 7 1	4 5 4 9 5
T	6	3	3	2	5
m			3		

1 2 1 0 2 1 1	3 3	6 4 5 6 7 5 8 4 5 6 7 4 5 4 9 5
L	E	G
$\|L\| = 7$	$\|E\| = 2$	$\|G\| = 16$

To find the 5th smallest element in S, select the 5th smallest element in L ($|L| \geq 5$) as follows

S	1 2 1 0 2	1 1
T	1	1
m		1

0	1 1 1 1	2 2
L	E	G

Sequential rank-based selection

- A parallel selection algorithm
  - If parallel computation model supports fast sorting
    - the problem can be solved through sorting
    - This is the case, e.g., for the CRCW-S or “summation” submodel with $p = n^2$ processors

Sequential rank-based selection

- A parallel algorithm for CRCW-S with $p = n^2$
  - Processor (i, j) compares inputs S[i] and S[j]
    - writes a 1 into rank[j] if $S[i] < S[j]$ or if $S[i] = S[j]$ and $i < j$.
  - rank[j] will hold the rank of S[j] in the sorted list
  - In the second cycle
    - Processor (0, j), $0 \leq j < n$, reads S[j] and writes it into S [rank[j]]
  - The selection process is completed in a third cycle when all processors read S[k – 1]
    - the kth smallest element in S

Sequential rank-based selection

- A parallel algorithm for CRCW-S with $p = n^2$
  - It is difficult to imagine a faster selection algorithm.
  - However, it is quite impractical
    - It uses both
      - many processors
      - a very strong PRAM submodel

Sequential rank-based selection

- Parallel version of the sequential algorithm
  - Assume $p=n^{1-x}$
    - x is a parameter that is known a priori
    - x = 1/2 corresponds to $p=\sqrt{n}$
    - P is sublinear in n
  - Step 1 involves
    - Broadcasting
      - needs O(log p) = O(log n) time
    - dividing into sublists
      - is done in constant time
      - each processor independently computing the beginning and end of its associated sublist based on |S| and x
    - Sequential selection on each sublist of length n/p
      - needs $O(n/p) = O(n^x)$ time

<u>Parallel rank-based selection algorithm *PRAMselect(S, k, p)*</u>

1. if $|S| < 4$
   then sort S and return the kth smallest element of S
   else broadcast $|S|$ to all p processors
   divide S into p subsequences $S^{(j)}$ of size $|S|/p$
   Processor j, $0 \le j < p$, compute the median $T_j := select(S^{(j)}, |S^{(j)}|/2)$
   endif
2. $m = PRAMselect(T, |T|/2, p)$ {find the median of the medians in parallel}
3. Broadcast m to all processors and create 3 subsequences
   L: Elements of S that are $< m$
   E: Elements of S that are $= m$
   G: Elements of S that are $> m$
4. if $|L| \ge k$
   then return $PRAMselect(L, k, p)$
   else if $|L| + |E| \ge k$
   then return m
   else return $PRAMselect(G, k - |L| - |E|, p)$
   endif

Sequential rank-based selection

- Parallel version of the sequential algorithm
  - Step 3 can be done as follows
    - each processor counts the number of elements that it should place in each of the lists L , E, and G
      - in $O(n/p) = O(n^x)$ time
    - three diminished parallel prefix computations are performed
      - to determine the number of elements to be placed on each list by all processors with indices that are smaller than i.
    - the actual placement takes $O(n^x)$ time
      - each processor independently writing into the lists L, E , and G
        - using the diminished prefix computation result as the starting address

<u>Parallel rank-based selection algorithm *PRAMselect(S, k, p)*</u>

```
1.  if |S| < 4
    then  sort S and return the kth smallest element of S
    else  broadcast |S| to all p processors
          divide S into p subsequences S^(j) of size |S|/p
          Processor j, 0 ≤ j < p, compute the median T_j := select(S^(j), |S^(j)|/2)
    endif
2.  m = PRAMselect(T, |T|/2, p)      {find the median of the medians in parallel}
3.  Broadcast m to all processors and create 3 subsequences
    L:  Elements of S that are < m
    E:  Elements of S that are = m
    G:  Elements of S that are > m
4.  if |L| ≥ k
    then return PRAMselect(L, k, p)
    else  if |L| + |E| ≥ k
          then return m
          else return PRAMselect(G, k − |L| − |E|, p)
    endif
```

Sequential rank-based selection

- Parallel version of the sequential algorithm
  - logarithmic terms are negligible compared with $O(n^x)$
  - Step 4 will have no more than 3n/4 inputs
  - for $p = n^{1-x}$ we have
    - $T(n, p) = T(n^{1-x}, p) + T(3n/4, p) + cn^x$
    - $T(n, p) = O(n^x)$

<u>Parallel rank-based selection algorithm *PRAMselect(S, k, p)*</u>

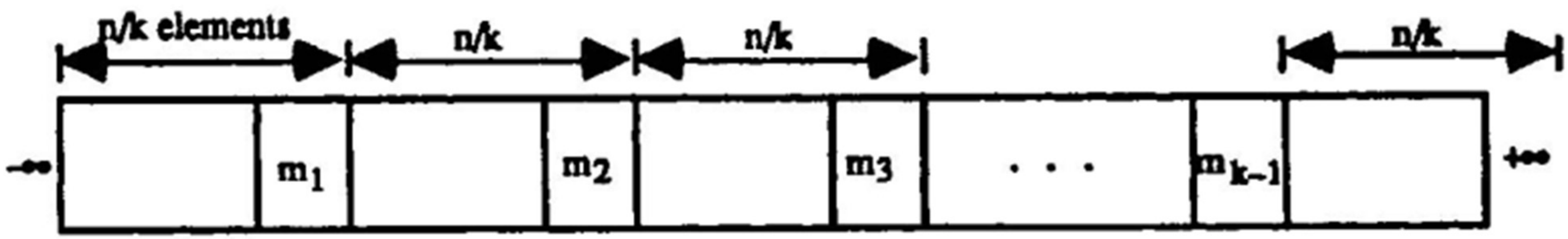
1. if $|S| < 4$
   then sort S and return the kth smallest element of S
   else broadcast $|S|$ to all p processors
   divide S into p subsequences $S^{(j)}$ of size $|S|/p$
   Processor j, $0 \le j < p$, compute the median $T_j := select(S^{(j)}, |S^{(j)}|/2)$
   endif
2. $m = PRAMselect(T, |T|/2, p)$ {find the median of the medians in parallel}
3. Broadcast m to all processors and create 3 subsequences
   L: Elements of S that are $< m$
   E: Elements of S that are $= m$
   G: Elements of S that are $> m$
4. if $|L| \ge k$
   then return $PRAMselect(L, k, p)$
   else if $|L| + |E| \ge k$
   then return m
   else return $PRAMselect(G, k - |L| - |E|, p)$
   endif

Sequential rank-based selection

- Parallel version of the sequential algorithm
  - Speed-up (n, p) = Θ (n)/O(n^x) = $\Omega(n^{1-x}) = \Omega(p)$
  - Efficiency = Speed-up / p = Ω (1)
  - Work (n, p) = pT(n, p) = $\Theta(n^{1-x})O(n^x) = O(n)$
  - One positive property
    - it is adaptable to any number of processors
    - yields linear speed-up in each case.
    - we do not have to adjust the algorithm for running it on different hardware configurations.
      - It is self-adjusting

Sequential rank-based selection

- A selection-based sorting algorithm
  - Choose a small constant k
  - identify the k – 1 elements in positions n/k, 2n/k, 3n/k, . . . , (k – 1)n/k in the sorted list
    - Call them $m_1, m_2, m_3, \ldots, m_{k-1}$
    - define $m_0 = -\infty$ and $m_k = +\infty$
  - Put the above k–1 elements in their proper places in the sorted list
  - Move all other elements so that
    - any element that is physically located between m_i and m_{i+1} in the list has a value in the interval $[m_i, m_{i+1}]$
  - independently sort each of the k sublists



Sequential rank-based selection

- A selection-based sorting algorithm
  - assumptions
    - $p < n$ processors with $p = n^{1-x}$.
    - x is known a priori
    - we can choose $k=2^{1/x}$

Sequential rank-based selection

- A selection-based sorting algorithm
  - Step 1 takes constant time
  - Step 2 consists of
    - k parallel selection problems
      - n inputs & n^{1-x} processors.
      - k is a constant
      - total time is $O(n^x)$

Parallel selection-based sorting algorithm *PRAMselectionsort(S, p)*

1. if $|S| < k$ then return *quicksort* (S)
2. for $i = 1$ to $k - 1$ do
   $m_i := PRAMselect(S, i|S|/k, p)$
   {for notational convenience, let $m_o := -\infty$; $m_k := +\infty$}
   endfor
3. for $i = 0$ to $k - 1$ do
   make the sublist $T^{(i)}$ from elements of S that are between m_i and m_{i+1}
   endfor
4. for $i = 1$ to $k/2$ do in parallel
   PRAMselectionsort $(T^{(i)}, 2p/k)$
   {$p/(k/2)$ processors are used for each of the $k/2$ subproblems}
   endfor
5. for $i = k/2 + 1$ to k do in parallel
   PRAMselectionsort $(T^{(i)}, 2p/k)$
   endfor

Sequential rank-based selection

- A selection-based sorting algorithm
  - Step 3
    - each processor compares its n^x values with the k – 1 thresholds
    - counts the elements for each of the k partitions.
    - k diminished parallel prefix computations performed
      - each taking O(log p) =O(log n) time
    - each processor writes its n^x elements to the various partitions.
    - Step 3 takes a total of $O(n^x)$ time

Parallel selection-based sorting algorithm *PRAMselectionsort(S, p)*

1. if $|S| < k$ then return *quicksort* (S)
2. for $i = 1$ to $k - 1$ do
   m_i := *PRAMselect* (S, $i|S|/k$, p)
   {for notational convenience, let $m_o := -\infty$; $m_k := +\infty$}
   endfor
3. for $i = 0$ to $k - 1$ do
   make the sublist $T^{(i)}$ from elements of S that are between m_i and m_{i+1}
   endfor
4. for $i = 1$ to $k/2$ do in parallel
   PRAMselectionsort ($T^{(i)}$, $2p/k$)
   {$p/(k/2)$ processors are used for each of the $k/2$ subproblems}
   endfor
5. for $i = k/2 + 1$ to k do in parallel
   PRAMselectionsort($T^{(i)}$, $2p/k$)
   endfor

Sequential rank-based selection

- A selection-based sorting algorithm
  - Step 4 cannot handle all the k subproblems
    - Needed processors to solve each subproblem
      - (number of inputs)$^{1-x}$ = $(n/k)^{1-x} = (n/2^{1/x})^{1-x} = n^{1-x}/2^{1/x-1} = p/(k/2)$
    - Total needed processors
      - $k * p/(k/2) = 2*p$

Parallel selection-based sorting algorithm *PRAMselectionsort(S, p)*

```
1.  if |S| < k then return quicksort(S)
2.  for i = 1 to k – 1 do
       m_i := PRAMselect(S, i|S|/k, p)
       {for notational convenience, let m_o := –∞; m_k := +∞}
    endfor
3.  for i = 0 to k – 1 do
       make the sublist T^(i) from elements of S that are between m_i and m_i+1
    endfor
4.  for i = 1 to k/2 do in parallel
       PRAMselectionsort(T^(i), 2p/k)
       {p/(k/2) processors are used for each of the k/2 subproblems}
    endfor
5.  for i = k/2 + 1 to k do in parallel
       PRAMselectionsort(T^(i), 2p/k)
    endfor
```

Sequential rank-based selection

- A selection-based sorting algorithm
  - Steps 4 and 5 recursively call the algorithm
  - Total running time
    - $T(n, p) = 2T(n/k, 2p/k) + cn^x$
    - $T(n, p) = O(n^x \log n)$

Parallel selection-based sorting algorithm *PRAMselectionsort(S, p)*

1. if $|S| < k$ then return *quicksort* (S)
2. for $i = 1$ to $k - 1$ do
   m_i := *PRAMselect* (S, $i|S|/k$, p)
   {for notational convenience, let $m_o := -\infty$; $m_k := +\infty$}
   endfor
3. for $i = 0$ to $k - 1$ do
   make the sublist $T^{(i)}$ from elements of S that are between m_i and m_{i+1}
   endfor
4. for $i = 1$ to $k/2$ do in parallel
   PRAMselectionsort ($T^{(i)}$, $2p/k$)
   {$p/(k/2)$ processors are used for each of the $k/2$ subproblems}
   endfor
5. for $i = k/2 + 1$ to k do in parallel
   PRAMselectionsort ($T^{(i)}$, $2p/k$)
   endfor

Sequential rank-based selection

- A selection-based sorting algorithm
  - Speed-up(n, p) = $\Omega(n \log n) / O(n^x \log n) = \Omega(n^{1-x}) = \Omega(p)$
  - Efficiency = Speed-up / p = $\Omega(1)$
  - Work(n, p) = pT(n, p) = $\Theta(n^{1-x})\, O(n^x \log n) = O(n \log n)$

Sequential rank-based selection

- A selection-based sorting algorithm
  - Example

$|S| = 25$ elements, using $p = 5$ processors (thus, $x = 1/2$ and $k = 2^{1/x} = 4$)

$$S: 6\ 4\ 5\ 6\ 7\ 1\ 5\ 3\ 8\ 2\ 1\ 0\ 3\ 4\ 5\ 6\ 2\ 1\ 7\ 0\ 4\ 5\ 4\ 9\ 5$$

$$m_0 = -\infty$$

$$n/k = 25/4 \approx 6 \qquad m_1 = PRAMselect(S, 6, 5) = 2$$

$$2n/k = 50/4 \approx 13 \qquad m_2 = PRAMselect(S, 13, 5) = 4$$

$$3n/k = 75/4 \approx 19 \qquad m_3 = PRAMselect(S, 19, 5) = 6$$

$$m_4 = +\infty$$

T: - - - - - 2|- - - - - - 4|- - - - - 6|- - - - - -

T: 0 0 1 1 1 2|2 3 3 4 4 4 4|5 5 5 5 5 6|6 6 7 7 8 9

Alternative sorting algorithms

- Alternative sorting algorithms
  - previous algorithm results in k subproblems of the same size
    - allows us to establish an optimal upper bound on the worst-case running time
    - Brings up complexity
  - There exist many useful algorithms that
    - are quite efficient on the average
    - but exhibit poor worst-case behavior
  - Sequential quicksort is a prime example
    - runs in order n log n time in most cases
    - but can take on the order of n^2 time for worst-case input patterns.

Alternative sorting algorithms

- Alternative sorting algorithms
  - In the case of previous sorting algorithm
    - We can choose thresholds approximately equal to in/k
      - the rest of the algorithm dose not change
      - the only difference we get
        - k subproblems will be of roughly the same size

Alternative sorting algorithms

- Alternative sorting algorithms
  - Given a large list S of inputs
    - Use a random sample of the elements to establish the k thresholds.
    - easier if we pick k = p
      - A single processor handles each subproblem
  - assumption: p$\ll\sqrt{n}$

Parallel randomized sorting algorithm PRAMrandomsort(S, p)

1. Processor j, $0 \le j < p$, pick $|S|/p^2$ random samples of its $|S|/p$ elements and store them in its corresponding section of a list T of length $|S|/p$
2. Processor 0 sort the list T
   {the comparison threshold m_i is the $(i|S|/p^2)$th element of T}
3. Processor j, $0 \le j < p$, store its elements that are between m_i and m_{i+1} into the sublist $T^{(i)}$
4. Processor j, $0 \le j < p$, sort the sublist $T^{(j)}$

Alternative sorting algorithms

- Alternative sorting algorithms
  - Binary radixsort
    - we examine every bit of the k-bit keys in turn
    - starting from the least-significant bit (LSB)
    - In Step I
      - Examine bit i, $0 \le i < k$
      - Shift records with keys having
        - a 0 in bit i toward the beginning of the list
        - a 1 in bit i toward the end of the list
      - keep the relative order of records with the same bit
        - sometimes referred to as stable sorting

Input list	Sort by LSB	Sort by middle bit	Sort by MSB
5 (101)	4 (100)	4 (100)	1 (001)
7 (111)	2 (010)	5 (101)	2 (010)
3 (011)	<u>2 (010)</u>	<u>1 (001)</u>	2 (010)
1 (001)	5 (101)	2 (010)	<u>3 (011)</u>
4 (100)	7 (111)	2 (010)	4 (100)
2 (010)	3 (011)	7 (111)	5 (101)
7 (111)	1 (001)	3 (011)	7 (111)
2 (010)	7 (111)	7 (111)	7 (111)

Alternative sorting algorithms

- Alternative sorting algorithms
  - Binary parallel radixsort
    - upward and downward shifting step can be done efficiently in parallel
    - For Bit 0, new position of each record can be established by two prefix sum computations:
      - a diminished prefix sum computation on the complement of Bit 0
        - for records with 0 in bit position 0
      - a normal prefix sum computation on Bit 0
        - for each record with 1 in bit position 0 relative to the last record of the first category
    - running time
      - mainly consists of the time to perform 2k parallel prefix computations
        - k is the key length in bits
      - For k a constant
        - the running time is asymptotically O(log p) for sorting a list of size p using p processors

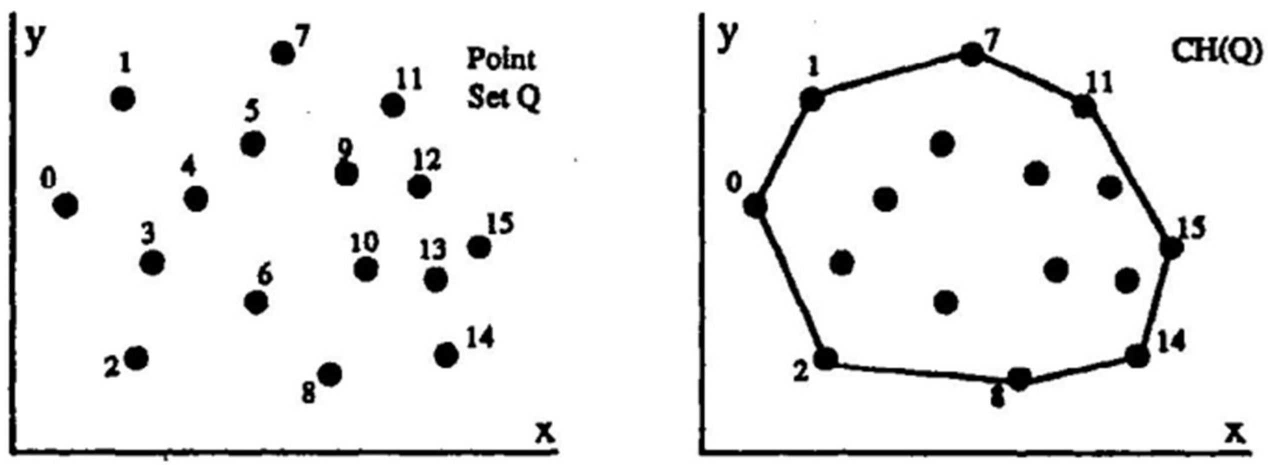
Input list	Compl't of Bit 0	Diminished prefix sums	Bit 0	Prefix sums plus 2	Shifted list
5 (101)	0	—	1	1 + 2 = 3	4 (100)
7 (111)	0	—	1	2 + 2 = 4	2 (010)
3 (011)	0	—	1	3 + 2 = 5	2 (010)
1 (001)	0	—	1	4 + 2 = 6	5 (101)
4 (100)	1	0	0	—	7 (111)
2 (010)	1	1	0	—	3 (011)
7 (111)	0	—	1	5 + 2 = 7	1 (001)
2 (010)	1	2	0	—	7 (111)

Convex hull of a 2D point set

- The convex hull problem for a 2D point set
  - Given a point set Q of size n
    - points specified by their (x, y) on the Euclidean plane
    - find the smallest convex polygon that encloses all n points
  - It is an example of geometric problems
    - encountered in image processing and computer vision.
  - The algorithm is also an excellent case study of multiway divide and conquer

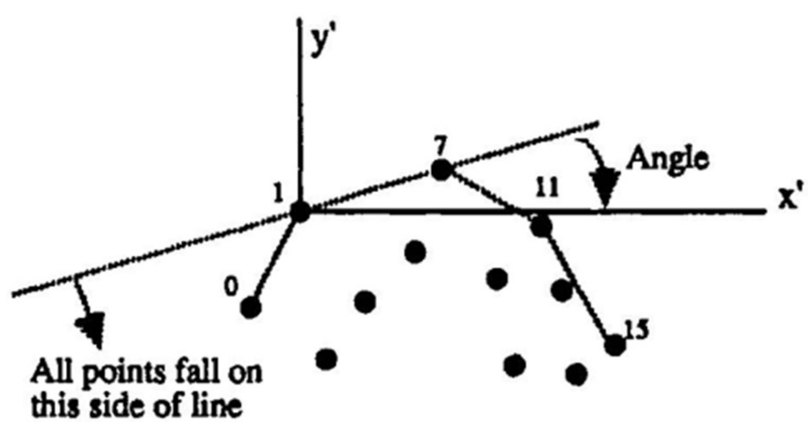
Convex hull of a 2D point set

- The convex hull problem for a 2D point set
  - The inputs can be assumed to be
    - in the form of two n-vectors X and Y
  - The desired output is a list of points
    - belonging to the convex hall
    - starting from an arbitrary point
    - proceeding, say, in clockwise order
    - has a size of at most n
  - convex hull can be divided into
    - the upper hull
      - goes from the point with the smallest x to the one with the largest x
    - the lower hull
      - returns from the latter to the former



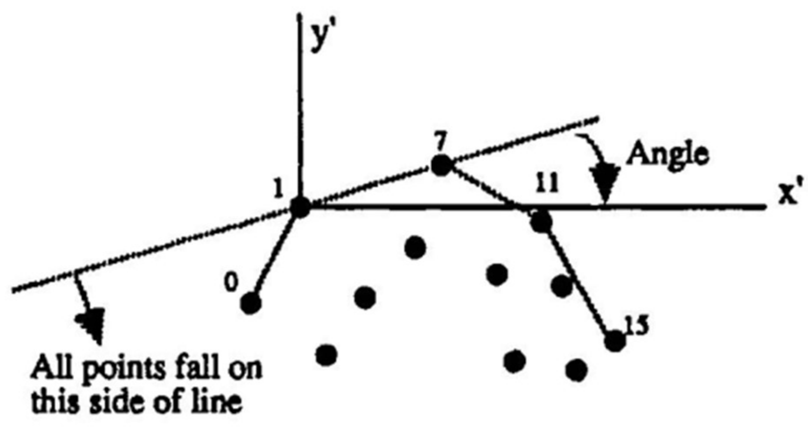
Convex hull of a 2D point set

- properties that allow us to construct an efficient PRAM algorithm
  - Property 1.
    - Let q_i and q_j be consecutive points of CH(Q).
    - View q_i as the origin of coordinates.
    - The line from q_j to q_i forms a smaller angle with x axis than the line from q_j to any other q_k in Q.



Convex hull of a 2D point set

- properties that allow us to construct an efficient PRAM algorithm
  - Property 2
    - A segment (q_i, q_j) is an edge of CH(Q)
      - iff all of the remaining n–2 points fall to the same side of it

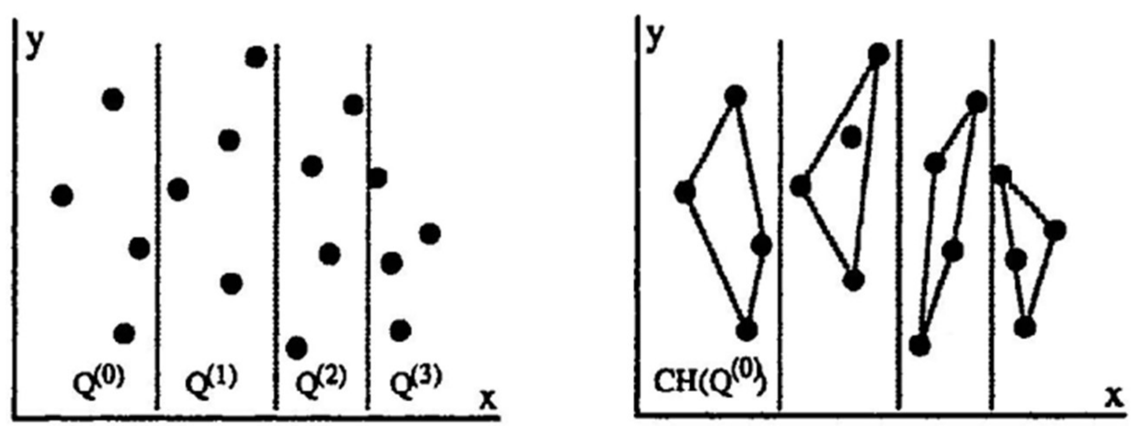


Convex hull of a 2D point set

- Convex hull algorithm
  - for a 2D point set of size p
  - on a p-processor CRCW PRAM.

Parallel convex hull algorithm PRAMconvexhull(S, p)

1. Sort the point set by the x coordinates
2. Divide the sorted list into $\sqrt{p}$ subsets $Q^{(i)}$ of size $\sqrt{p}$, $0 \le i < \sqrt{p}$
3. Find the convex hull of each subset $Q^{(i)}$ by assigning $\sqrt{p}$ processors to it
4. Merge the $\sqrt{p}$ convex hulls $CH(Q^{(i)})$ into the overall hull $CH(Q)$

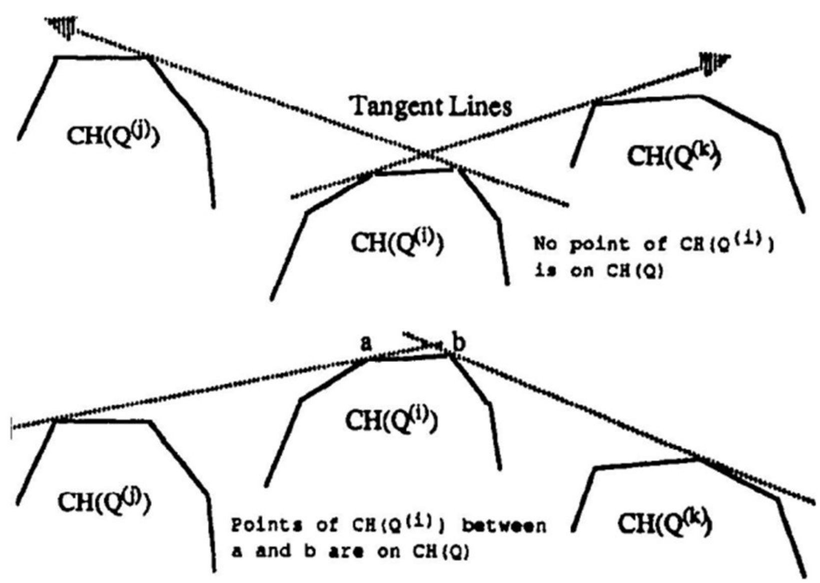


Convex hull of a 2D point set

- Convex hull algorithm
  - Step 4 is the heart of the algorithm
    - Each subset of size $\sqrt{p}$ is assigned $\sqrt{p}$ processors
      - to determine the upper tangent line between its hull and each of the other $\sqrt{p}$ -1 hulls.
      - One processor finds each tangent in O(log p) steps using Overmars algorithm
        - Based on binary search.
        - To determine the upper tangent from $CH(Q^{(i)})$ to $CH(Q^{(k)})$
          - The midpoint of the upper part of $CH(Q^{(k)})$ is taken
          - the slopes for its adjacent points compared with its own slope
          - If the slope is minimum
            - then we have found the tangent point
          - Otherwise
            - the search is restricted to one or the other half
          - CREW model must be assumed
            - Because multiple processors read data from all hulls

Convex hull of a 2D point set

- Convex hull algorithm
  - Step 4 is the heart of the algorithm
    - Once all the upper tangents from each hull to all other hulls are known
      - a pair of candidates are selected
        - By finding the min/max slopes
    - If the angle between the two candidates is less than 180
      - no point from $CH(Q^{(i)})$ belongs to $CH(Q)$
    - Else
      - a subset of points from $CH(Q^{(i)})$ belongs to $CH(Q)$



Convex hull of a 2D point set

- Convex hull algorithm
  - The final step is to renumber the points in proper order
    - to obtain rank or index of each node on CH(Q)
      - Use a parallel prefix on the list of the number of points from each CH($Q^{(i)}$) that are belong to the combined hull
  - The complexity excluding the initial sorting
    - $T(p, p) = T(p^{1/2}, p^{1/2}) + c \log p \approx 2c \log p$
  - sorting can also be performed in O(log p)
    - overall time complexity is O(log p)
  - the above algorithm is asymptotically optimal
    - Because best sequential algorithm requires $\Omega(p \log p)$

Some implementation aspects

- In any physical implementation of shared memory
  - the m memory locations are in B memory banks (modules)
  - each bank holding m/B addresses
- in each memory cycle
  - a memory bank can provide access to a single memory word.
- multiport memories exist
  - can allow access to a few independently addressed words in a single cycle
  - are quite expensive
  - if the number of memory ports is less than m/B
    - which is certainly the case in practice
    - Multiport memories do not allow us the same type of permitted access even in the weakest PRAM submodel.

Some implementation aspects

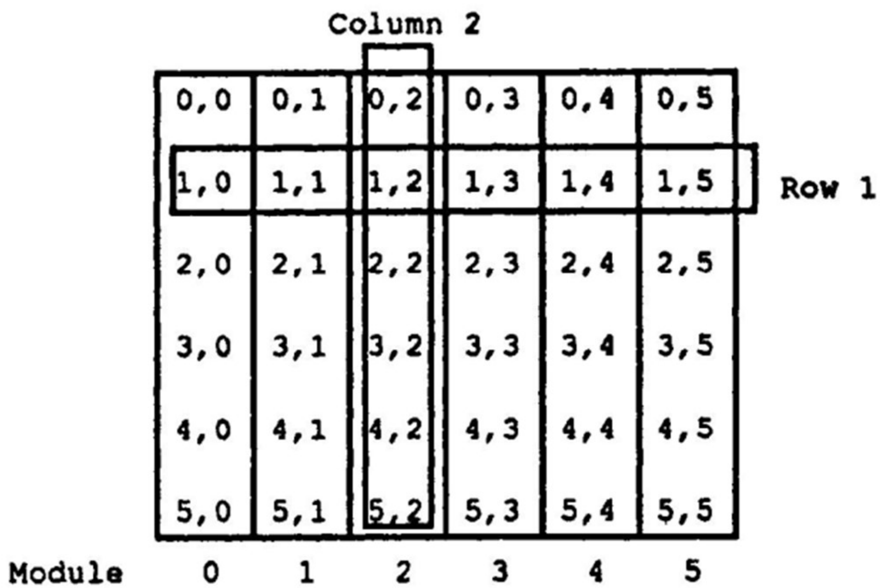
- even if the PRAM algorithm assumes the EREW
  - memory bank conflicts may still arise
- moderate to serious loss of performance may result
  - Depending on how bank conflicts are resolved
- An obvious solution: prevent conflicts by
  - try to lay out the data in the shared memory
  - organize the computational steps
    - so that a memory bank is accessed at most once in each cycle.
  - quite a challenging problem
    - has received significant attention from the research community

Some implementation aspects

- Consider an $m \times m$ matrix multiplication with $p=m^2$ processors
  - each processor has an index pair (i, j).
  - P_{ij} is responsible for computing the element c_{ij}
  - P_{iy}, $0 \leq y < m$ need to read Row i of A
  - we can skew the accesses
    - P_{iy} reads the elements of Row i beginning with A_{iy}.
      - the entire Row i of A is read out in every cycle
      - albeit with the elements distributed differently to the processors in each cycle.

Some implementation aspects

- Consider an m×m matrix multiplication with $p=m^2$ processors
  - To remove conflicts for all elements of each row
    - we must assign different columns of A to different memory banks.
    - It is possible if
      - we have at least m memory banks
      - We store the matrix in column-major order
        - the element (i, j) is found in location i of memory bank j
    - If fewer than m memory modules are available
      - the element (i, j) can be stored in location $i + m\lfloor j/B \rfloor$ of memory bank j mod B
      - This ensures maximum parallelism in reading the row elements

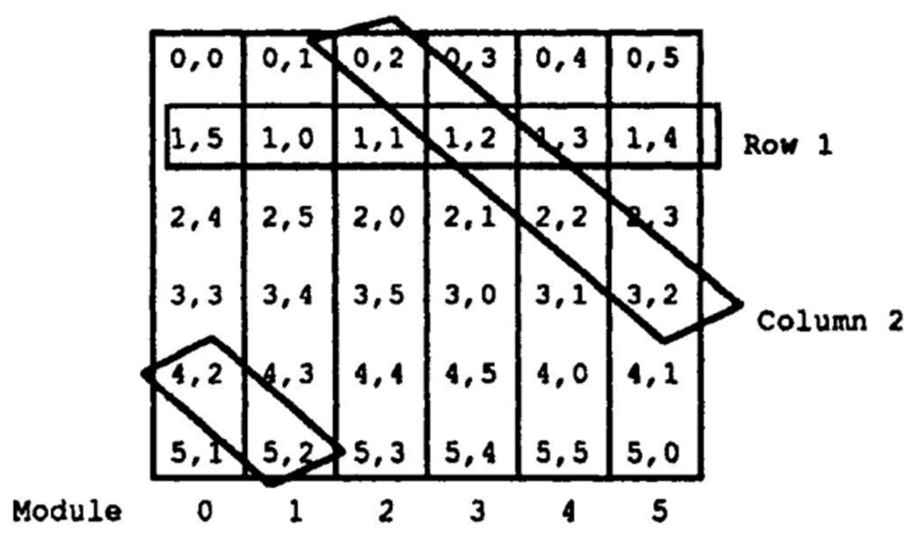


Some implementation aspects

- Consider an $m \times m$ matrix multiplication with $p=m^2$ processors
  - Processors P_{xj}, $0 \leq x < m$, all access the j^{th} column of B.
    - column-major storage leads to memory bank conflicts for all columns of B.
    - We can store B in row-major order to avoid such conflicts.
    - if B is later to be used in a different matrix multiplication, say $B \times D$
      - the layout of B must be changed
        - by physically rearranging it in memory
      - or the algorithm must be modified

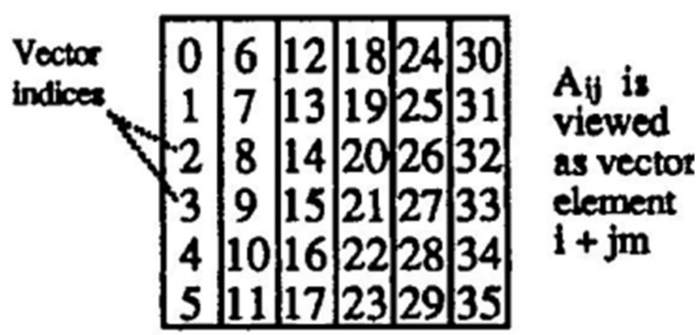
Some implementation aspects

- Consider an m×m matrix multiplication with $p=m^2$ processors
  - skewed storage can be used
    - both columns and rows are accessible in parallel without memory bank conflicts.
    - the element (i ,j) is found in location i of module (i + j) mod B
      - If $B \geq m$
        - all elements (i, y), $0 \leq y < m$ are in different modules
        - all elements (x, j), $0 \leq x < m$ are in different modules
      - conflicts could arise for diagonal elements (x, x) unless
        - $B \geq 2m$
        - or else B is an odd number in the range $m \leq B < 2$ m.



Some implementation aspects

- Generalized conflict-free parallel matrix access
  - View the m×m matrix as an m^2-element vector
    - column-major or row-major order does not matter
      - only interchanges the first two strides



Column:	$k, k+1, k+2, k+3, k+4, k+5$	Stride of 1
Row:	$k, k+m, k+2m, k+3m, k+4m, k+5m$	Stride of m
Diagonal:	$k, k+m+1, k+2(m+1), k+3(m+1), k+4(m+1), k+5(m+1)$	Stride of $m+1$
Antidiagonal:	$k, k+m-1, k+2(m-1), k+3(m-1), k+4(m-1), k+5(m-1)$	Stride of $m-1$

Some implementation aspects

- Generalized conflict-free parallel matrix access
  - The problem is reduced to
    - Given a vector of length *l*
      - store it in B memory banks in such a way that
        - accesses with strides $s_0, s_1, \ldots, s_{h-1}$ are
          - conflict-free (ideal)
          - involve the minimum possible amount of conflict.

Some implementation aspects

- Generalized conflict-free parallel matrix access
  - linear skewing scheme
    - stores the k^{th} vector element in the bank a+kb mod B
    - The address within the bank
      - is irrelevant to conflict-free parallel access
      - does affect the ease with which memory addresses are computed by the processors
    - The constant a is also irrelevant and can be safely ignored.
    - Thus, we can limit our attention to assigning V_k to memory module $M_{kb \bmod B}$.

Some implementation aspects

- Generalized conflict-free parallel matrix access
  - linear skewing scheme
    - the elements k, k+s, k+2s, . . . , k+(B– 1)s are in different memory modules
      - iff sb is relatively prime with respect to the number B of memory banks.

Some implementation aspects

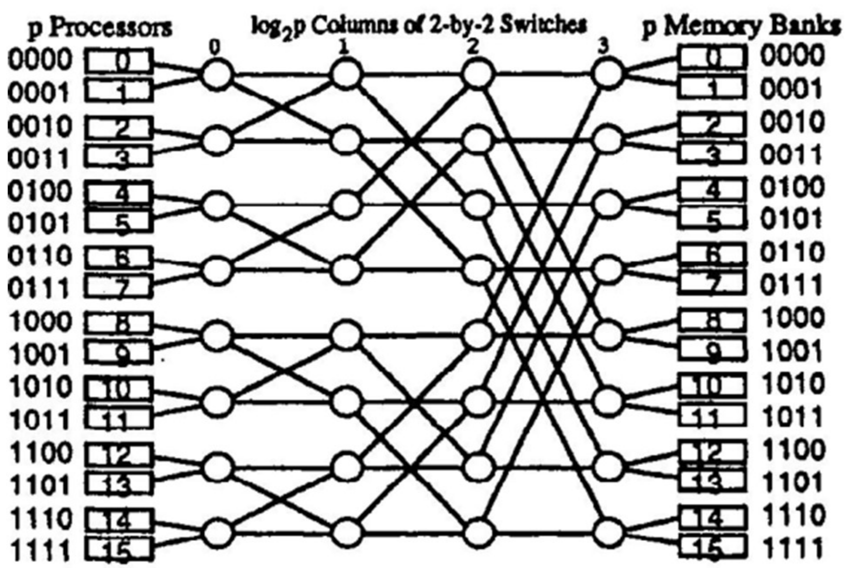
- Generalized conflict-free parallel matrix access
  - linear skewing scheme
    - the elements k, k+s, k+2s, . . . , k+(B– 1)s are in different memory modules
      - iff sb is relatively prime with respect to the number B of memory banks.
    - If we choose B to be a prime number
      - conflict-free parallel access for all strides is guaranteed for b=1
    - But having a prime number of banks is inconvenient for other reasons
      - Thus, many alternative methods have been proposed

Some implementation aspects

- Even assuming conflict-free access to memory banks
  - Still, multiple memory access must be directed from the processors to the memory banks
  - this is a nontrivial problem
    - If we have many processors and memory banks
  - Ideally
    - the memory access network should be a permutation network
      - Can connect each processor to any memory bank
        - as long as the connection is a permutation.
  - However, permutation networks are
    - quite expensive to implement
    - difficult to control (set up).
  - Therefore
    - we usually settle for networks that do not possess full permutation capability.

Some implementation aspects

- Multistage interconnection network
  - an example of a compromise solution.
  - It is a butterfly network
    - we will encounter again in the next chapters
  - For now
    - only note that memory accesses can be self-routed through this network
      - by letting the i^{th} bit of the memory bank address determine the switch setting in Column $i-1$ ($1 \leq i \leq 3$)
        - 0 indicating the upper path
        - 1 the lower path.
    - E.g., any request to memory bank 3 (0011)
      - will be routed to the “lower,” “upper,” “upper,” “lower” output line
        - by the switches that forward it in Columns 0–3.
        - independent of the source processor



Some implementation aspects

- Multistage interconnection network
  - switches can be designed to deal with access conflicts by
    - simply dropping duplicate requests
      - memory acknowledgment is required
    - buffering one of the two conflicting requests
      - introduces nondeterminacy in the memory access time
      - determining the buffer size is a challenging problem
    - combining access requests to the same memory location.