CS 467/567: Algorithms for Interconnection Networks

Stefan D. Bruda

Winter 2023

PRAM vs. interconnection networks

- The PRAM is a very powerful model, rarely realizable in practive
  - It is however important for the theory of algorithms
  - Lower bounds are particularly strong on the PRAM
  - Surprising equivalences to other, realistic models
- Most massively parallel machines are laid out as networks
- From the point of view of the theory of algorithms interconnection networks typically have fixed topology
  - An interconnection network is therefore a family of graphs with RAM processors (including storage) as nodes and (direct) data links as edges
  - The number of processors (nodes) may vary, but the topology remains the same
  - Possible topology: linear array, mesh, tree, hypercube, fully connected (not realistic), etc.
- Note however that models with variable topology also exist

Linear arrays and on-line sorting

- In a linear array with n procesors, procesor P_i is (bidirectionally) connected to processor P_{i+1} for all $1 \leqslant i < n$
  - The simplest network topology, weakest model
- Problem: Sort in nondecreasing order a sequence $S = \langle S_1, S_2, \ldots, S_n \rangle$ which is available on-line, meaning that each S_i becomes available at time i, $1 \leqslant i \leqslant n$
  - Assume that P_1 is the "input processor" where input data becomes available
- $\Omega(n)$ lower bound for the running time no matter how many processors are available
  - Indeed, this is how much time it takes for all the data to arrive
- Useful basic operation: $\textsc{Compare-exchange}(P_i, P_{i+1})$
  - Compares the designated values held by P_i and P_{i+1} and possibly exchanges them, so that the smaller value is placed in P_i and the largest in P_{i+1}
  - $O(1)$ computation and communication steps

Sorting by comparison-exchange

Algorithm Sort-comparison-exchange:

1. P_1 reads S_1
2. **for** $j = 2$ **to** n **do**
   1. **for** $i = 1$ **to** $j - 1$ **do in parallel** P_i sends its designated value to P_{i+1}
   2. P_1 reads s_j
   3. **for all** odd $i < j$ **do in parallel** $\textsc{Compare-exchange}(P_i, P_{i+1})$
3. **for** $j = 1$ **to** n **do in parallel**
   1. P_1 produces its datum as output
   2. **for** $i = 2$ **to** $n - j + 1$ **do in parallel** P_i sends its datum to P_{i-1}
   3. **for all** odd $i < n - j$ **do in parallel** $\textsc{Compare-exchange}(P_i, P_{i+1})$

- Linear (optimal) running time, but $O(n^2)$ cost

Sorting by merging

Maintain the PRAM idea of several merges overlapping

- Now the merges are pipelined in a real pipeline
- We actually need two pipelines, so conceptually we consider that there are two links (top and bottom) between processors

Mergesort on $r+1$ processors, with $r = \log n$:

- Processor P_1:
  1. Reads s_1 from the input sequence; $i \leftarrow 1$
  2. **for** $i = 2$ **to** n **do**
     1. **if** j is odd **then** place s_{i-1} on the top link
     2. **else** place s_{i-1} on the bottom link
     3. Reads s_i from the input sequence; $i \leftarrow j+1$
  3. Plase s_n on the bottom link

Sorting by merging (cont'd)

- Processor P_i, $2 \leqslant i \leqslant r$:
  1. $j \leftarrow 1$, $k \leftarrow 1$
  2. **while** $k < n$ **do**
     if the top input buffer contains 2^{i-2} values **and** the bottom input buffer contains one value **then**
     1. **for** $m = 1$ **to** 2^{i-1} **do**
        (a) Let x be the largest of the first elements from the top and bottom buffers
        (b) Remove x from its buffer
        (c) **if** j is odd **then** place x on the top link
        (d) **else** place x on the bottom link
     2. $j \leftarrow j+1$, $k \leftarrow k + 2^{i-1}$
- Processor p_{r+1}:
  1. **if** the top input buffer contains 2^{r-1} values **and** the bottom input buffer contains one value **then**
     1. Let x be the largest of the first elements from the top and bottom buffers
     2. Remove x from its buffer and produce it as output
- P_i needs $2^{i-2}+1$ values so it starts at time $2^{i-2}+1$ after P_{i-1}
- P_i produces its first output at time $1 + (2^0+1) + (s^1+1) + \cdots + (2^{i-2}+1) = 2^{i-1} + i - 1$ and its last output $n-1$ time units later
- Running time $2^r + r + (n-1) = 2n + \log n - 1 = O(n)$; cost $O(n \log n)$

Sorting (off-line)

- Lower bound assuming that the input data is distributed to all processors: $\Omega(N)$ time (and so $\Omega(N^2)$ cost)
  - In the worst case one datum must traverse the diameter of the network
  - Diameter: the maximum number of links on the shortest path between two processors
- Therefore a bubble sort variant is optimal

  Algorithm Transposition-sort:
  1. **for** $j = 0$ **to** $N-1$ **do**
     for $i = 0$ **to** $N-1$ **do in parallel**
     1. **if** $i \bmod 2 = j \bmod 2$ **then** Compare-exchange(P_i, P_{i+1})

More complex networks

- Biggest disadvantage of the linear array: largest possible diameter
- The two-dimensional array (or mesh) provides a considerably smaller diameter while maintaining many of the nice properties of the linear array
  - Simple theoretically, appealing in practice
  - Fixed and small maximum degree for nodes (4)
  - Regular and modular topology
- In a mesh of N processors each processor P_{ij} is connected to $P_{(i+1)j}$ and $P_{i(j+1)}$, $1 \leqslant i,j < N^{1/2}$
  - $2N^{1/2} - 2$ diameter, considerably smaller than for the linear array
  - Still the diameter is quite large
- Good compromise between vertex degree and network diameter: the hypercube
  - For some integers i and b, let $i^{(b)}$ if the binary representations of i and $i^{(b)}$ differ only in the b position
  - The processors $P_1, P_2, \ldots, P_N$ for $N = 2^g$, $g \geqslant 1$ are arranged in a g-dimensional hypercube whenever each processor P_i is connected to exactly all the processors $P_{i^{(b)}}$, $0 \leqslant b < g$
  - $O(\log N)$ for both degree and diameter

Matrix multiplication on the hypercube

- Need to compute $c_{jk} = \sum_{i=0}^{n-1} a_{ji} \times b_{ik}$ for $0 \leqslant j,k < n$
  - Straightforward sequential algorithm: $O(n^3)$ running time
  - Best known sequential algorithm: $O(n^{2+\varepsilon})$ running time, $0 < \varepsilon < 0.38$
- For input size $n = 2^g$ we use a hypercube with $N = n^3 = 2^{3g}$ processors
  - Imagine the processors conceptually arranged in an $n \times n \times n$ array such that P_r (or $P_{(i,j,k)}$) occupies position (i,j,k) with $r = in^2 + jn + k$

$$r = \underbrace{r_{3g-1} r_{3g-2} \ldots r_{2q}}_{i} \underbrace{r_{2q-1} r_{2q-2} \ldots r_{q}}_{j} \underbrace{r_{q-1} r_{q-2} \ldots r_{0}}_{k}$$

  - Each set of processors that agrees with each other on one coordinate [two coordinates] form a hypercube with n^2 processors [n processors]
  - Processors $P_{(i,j,k)}$, $0 \leqslant j,k < n$ form a "layer" for n layers overall
  - Designated registers for $P_r/P_{(i,j,k)}$: A_r, B_r, C_r / $A_{(i,j,k)}$, $B_{(i,j,k)}$, $C_{(i,j,k)}$
  - Input available in $A_{(0,j,k)}$ ($A_{(0,j,k)} = a_{jk}$) and $B_{(0,j,k)}$ ($B_{(0,j,k)} = b_{jk}$)
  - Output produced in $C_{(0,j,k)}$ ($C_{(0,j,k)} = c_{jk}$)
- The algorithm performs all the arithmetic calculations in constant time, but still need $O(\log n)$ time for data distribution (not optimal)

Matrix multiplication (cont'd)

Algorithm MATRIX-MULTIPLICATION($A = (a_{ij})_{0\leqslant i,j\leqslant n}$, $B = (b_{ij})_{0\leqslant i,j\leqslant n}$)
returns $C = (c_{ij})_{0\leqslant i,j\leqslant n}$:

1. Data distribution: A and B (layer 0) are distributed to the other processors so that $P_{(i,j,k)}$ stores a_{ji} and b_{ik}
   1. **for** $m = 3g-1$ **down to** $2g$ **do**
      for all $0 \leqslant r < N \wedge r_m = 0$ **do in parallel** $A_{r^{(m)}} \leftarrow A_r$; $B_{r^{(m)}} \leftarrow B_r$
      // result: $A_{(i,j,k)} = a_{jk}$ and $B_{(i,j,k)} = b_{jk}$, $0 \leqslant i < n$
   2. **for** $m = g-1$ **down to** 0 **do**
      for all $0 \leqslant r < N \wedge r_m = r_{2g+m}$ **do in parallel** $A_{r^{(m)}} \leftarrow A_r$
      // $A_{(i,j,i)} \rightarrow A_{(i,j,k)}$; result: $A_{(i,j,k)} = a_{ji}$, $0 \leqslant k < n$
   3. **for** $m = 2g-1$ **down to** g **do**
      for ann $0 \leqslant r < N \wedge r_m = r_{g+m}$ **do in parallel** $B_{r^{(m)}} \leftarrow B_r$
      // $B_{(i,i,k)} \rightarrow B_{(i,j,k)}$; result: $B_{(i,j,k)} = a_{jk}$, $0 \leqslant i < n$
2. Term computation: Each $P_{(i,j,k)}$ computes $C_{(i,j,k)} \leftarrow A_{(i,j,k)} \times B_{(i,j,k)}$
   // result: $C_{(i,j,k)} = a_{ji} \times b_{ik}$
3. Summation: For $0 \leqslant j,k < n$ compute $C_{(0,j,k)} \leftarrow \sum_{i=0}^{n-1} C_{(i,j,k)}$

Connected components in graphs

- Connectivity matrix: given an adjacency matrix $A = (a_{ij})_{0\leqslant i,j<n}$ defining a graph $G = (\{0,1,\ldots,n\},E)$ ($a_{ij} = 1$ if $(i,j) \in E$ and 0 otherwise), the connectivity matrix $C = (c_{ij})_{0\leqslant i,j<n}$ is defined such that $c_{ij} = 1$ if there exists a path from i to j and 0 otherwise
- The connectivity matrix can be computed as follows: $C = A'^n$, where $A' = (a'_{ij})_{0\leqslant i,j<n}$ with $a'_{ii} = 1$ and $a'_{ij} = a_{ij}$ for all $i \neq j$
  - C-style booleans can use plain matrix multiplication; true booleans require multiplication with $\wedge$ instead of $\times$ and $\vee$ instead of $+$
  - Repeat multiplications on the hypercube do not necessitate data redistribution, since the result of the previous multiplication is in the right place for the next multiplication
  - We can compute C using $O(\log n)$ matrix multiplications
    - Indeed, the graph C is the reflexive and transitive closure of the graph A and so $A'^p = A'^n$ for any $p \geqslant n$
  - So C can be computed on the hypercube with n^3 processors and in $O(\log^2 n)$ time

All-pairs shortest paths

- Given a weight matrix W defining a graph $G = (\{0,1,\ldots,n\},E)$, compute the matrix D such that d_{ij} is the cost of the shortest path between i and j
  - We assume no cycles of negative weight (no advantage to visit any vertex more than once)
  - Useful property: Any shortest path between two vertices contain shortest paths between the intermediate vertices
  - So in computing a shortest path we can compute all the combinations of shortest subpaths and then choose the shortest one
  - So the shortest paths d_{ij}^k containing at most $k+1$ vertices can be computed inductively:
    - $d_{ij}^1 = w_{ij}$ whenever there exists a vertex between i and j and ∞ otherwise
    - $d_{ij}^k = \min_{0\leqslant p<n}(d_{ip}^{k/2} + d_{pj}^{k/2})$
  - $D^k = (d_{ij}^k)_{0\leqslant i,j<n}$ computable starting from D^1 using a special form of matrix multiplication with $+$ instead of $\times$ and $\min$ instead of $+$
    - $O(\log^2 n)$ time on the hypercube with n processors
- This can go like this all the way to minimum-weight spanning trees...
- Matrix representation for graphs more advantageous on the hypercube than other representations

- (Binary) tree
  - Degree 3, diameter $O(\log n)$
- Mesh of trees: $n^{1/2}$ identical binary trees of $n^{1/2}$ processors; each set of $n^{1/2}$ “equivalent” processors (in the sense of a preorder traversal) linked to form a binary tree
  - Degree 6, diameter $O(\log n)$
- Star: each processor is labeled with a permitation of $\{1, 2, \ldots, m\}$ (m! processors for a given m); two processors P_u and P_v are connected with each other whenever the index v can be obtained from the index u by exchanging the first symbol with the i-th symbol for some $2 \leqslant i \leqslant m$
  - Degree $m - 1$, diameter $O(m)$