Communication-optimal sorting

The best complexity achievable by a parallel comparison-based sort is

\[ T_{\text{sort}}(n, P, H) = \Omega \left( \frac{n \log_2(n)}{P} \cdot \gamma + \frac{n \log_2(n)}{P \log_2(\sqrt{n/P})} \cdot \beta + \frac{\log_2(n)}{\log_2(\sqrt{n/P})} \cdot \alpha \right) \]

\[ \text{recall } \log_2(n)/\log_2(\sqrt{n/P}) = \log_{\sqrt{n/P}}(n) \]

- R. Cole’s parallel mergesort (1988) achieves this for \( P = \Theta(n) \)
- M.T. Goodrich (1999) provides a mixed mergesort/samplesort that attains the cost for arbitrary \( P \)
- Cole’s construction uses a pipelined binary tree, Goodrich extends it to an \( s \)-ary tree with \( s = \sqrt{n/P} \)
Sampling and merging samples

The parallel mergesort algorithm is defined in a bottom-up way:

- we first place and sort subsequences of size $s^2 = n/P$ on $P$ leaves of the $s$-ary tree.
- for each tree node $u$, we will merge the $s$ subsequences assigned to its children $\pi(u)$
  \[
  L(u) = \text{sort}\left( \bigcup_{v \in \pi(u)} L(v) \right)
  \]
- each tree node will distribute its subsequence over $s$ more processors than those of its children.
- since the tree is of height $\log_s(P)$, the root node will end with $n$ sorted elements distributed over $P$ processors.
- at tree node with height $h$ will merge $s$ subsequences of size $s^h$, using a regular sample of total size $s^h$.
- Q: sounds simple enough... so why do we need to pipeline?
- A: merging the regular subsamples is almost as hard as the merge itself, so we will construct them gradually.
The short life of an exponentially-growing tree node

Nodes in the $s$-ary tree are either \textit{waiting}, \textit{growing}, or \textit{full}:

- node $u$ at the start of iteration $t$:
  - has a subset of its target subsequence $L_t(u) \subseteq L(u)$
  - is \textit{waiting} if $L_t(u) = \emptyset$
  - is \textit{growing} if $L_t(u) \neq \emptyset$ and $L_t(u) \neq L(u)$
  - is \textit{full} if it has its target subsequence $L_t(u) = L(u)$

- leaves are full at iteration 0
- each tree node $u$ waits up to iteration $t$, the first iteration at which its children have a subsequence of size $s^2$, then collects a sample of size $s$ from each and merges sequentially, so $|L_t(u)| = s^2$
- if $u$ is growing at iteration $t$, then it uses $L_t(u)$ as a sample to merge

$$L_{t+1}(u) = \text{sort}\left( \bigcup_{v \in \pi(u)} R(|L_t(u)|, L_t(v)) \right)$$

where $R(k, U)$ extracts a regular sample of size $k$ from $U$

- growing nodes grow by a factor of $s$, i.e. $|L_{t+1}(u)| = s|L_t(u)|$
Communication-optimal sort [Cole 1988], [Goodrich 1999]

1. Pass samples up and merge
Communication-optimal sort [Cole 1988], [Goodrich 1999]

3. Merge subpartitions
Communication-optimal sort [Cole 1988], [Goodrich 1999]

4. Pass up new samples
Communication-optimal sort [Cole 1988], [Goodrich 1999]

5. Pass up larger samples, use old sample to split
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Communication-optimal sort [Cole 1988], [Goodrich 1999]

zoom out horizontally
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Communication-optimal sort [Cole 1988], [Goodrich 1999]

Zoom out again!
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Communication-optimal sort [Cole 1988], [Goodrich 1999]

Zoom out again! Now sorting with $3^6 = 729$ processors!
Communication-optimal sort [Cole 1988], [Goodrich 1999]

Zoom out again! Now sorting with $3^6 = 729$ processors!
Communication-optimal sort [Cole 1988], [Goodrich 1999]

Zoom out again! Now sorting with $3^6 = 729$ processors!
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Communication-optimal sort [Cole 1988], [Goodrich 1999]
Sampling guarantees

At every step of the algorithm, we merge $s$ sequences of size $s^t$ using a sample of size $s^t$, with $s^{t-1}$ processors

- we show can partition the sequences such that each processor gets no more than $2s^2 = 2n/P$ elements
- there are at most $s$ elements in a sequence between two consecutive sample elements selected from that sequence
  - this is nontrivial, but holds as a consequence of a regular sample of $s^t$ elements being a regular sample of a regular sample of $s^{t+1}$ elements
- we select $s^{t-1} - 1$ splitters from sorted sample of size $s^t$
- if there are $k_j$ elements from subsequence $j$ between splitter $i$ and $i + 1$, it has at most $(k_j + 1)s$ elements between these splitters
- we can bound total $\sum_{j=1}^{s} k_j = s$
- the total number of elements in each interval is bounded by $\sum_{j=1}^{s} (k_j + 1)s \leq 2s^2$
Cost analysis of Cole/Goodrich parallel mergesort

There are $\log_s(P)$ levels in the tree

- A parent becomes full 3 iterations after its children become full, so the algorithm terminates in $3 \log_s(P)$ iterations
- The amount of work done at each step for growing parent is a factor of $s$ less than the child, and there are $s$ less nodes at the tree level
- Therefore, the work and memory usage decrease geometrically up from the highest level that is full
- Processing every node requires $O(s^2 \cdot \beta)$ and as much memory from every processor involved
- Therefore, each iteration can be executed with $O(s^2 \cdot \beta)$ communication and as much memory

$$W_{\text{sort}}(n, P) = O(s^2 \log_s(n) \cdot \beta) = O(n \log \sqrt[n/P]{P(n) \cdot \beta})$$
Parallel sample/mergesort

Analyzing the algorithm

Short pause
The Parallel Random Access Machine (PRAM)

The PRAM model is perhaps the most popular traditional parallelism model and is still in use today

- use a maximal number of processors suitable to compute the algorithm in a minimal number of parallel steps (depth)
- e.g. Cole’s mergesort uses $O(n)$ processors to sort in $O(\log(n))$ steps
- all processors can access a global memory
  - EREW (exclusive read exclusive write) - processors cannot access the same memory locations in a parallel step
  - CREW (concurrent read exclusive write) - processors can read but not write to the same memory locations in a parallel step
  - CRCW (concurrent read concurrent write) - processors can read and write to the same memory locations in a parallel step
  - for CRCW model need to define how writes are arbitrated (common, random, priority)

Brent’s Lemma: an EREW PRAM algorithm that uses $P$ processors and $s$ steps can be computed with in $P/k$ processors in $sk$ steps
Tree contraction

Consider an algebraic or boolean expression, e.g.

\[ T = ((a + b) \cdot (c + d)) \cdot ((e + f + g) \cdot (h + i)) \cdot (j + k) \]

- the problem can be represented as a rooted tree
- computing \( T \) corresponds to tree contraction
- we can use tree contraction for important tasks such as tree isomorphism and subexpression elimination in dataflow analysis
- a naive algorithm would recursively evaluate the \( n \) nodes in the tree, starting with the leaves
- Q: how many parallel steps would such an algorithm require in the worst case?
- A: \( O(n) \) if the tree has height \( O(n) \)
Parallel tree contraction

Miller and Reif (1989) provide a famous PRAM algorithm for the problem.

- Their algorithm consists of rake and compress steps.
- **Rake** evaluates all the leaves of the tree.
- **Compress** contracts all chains in the tree.
  - A chain is any connected subtree where each node has only one child.
  - Compress removes every other node in the chain.
- Q: Can you see why $O(\log(n))$ rake and compress steps would contract the tree?
- A: Any new tree branch increases the number of leaves by one.
How to compress in parallel?

It is nontrivial to identify which nodes in a chain are odd or even

- Miller and Reif provide deterministic and randomized solutions
- the deterministic solution splits chains using pointer chasing
  - we start with every child pointing to its parent
  - if parent has one child, point to grandparent (if exists)
  - this splits each chain into two, in one of the chains, a node that participated in compress now has no child
  - we proceed ensuring that this node is never evaluated
  - PRAM requires $O(n)$ processors and $O(\log(n))$ steps
- the randomized solution deletes a subset of nodes in any chain
  - randomly assign 1 or 0 to each node in the chain
  - pointer chase from every node marked with 0 whose parent and child are marked with 1
  - if $s$ nodes are part of chains, we delete $\Omega(s)$ nodes without breaking chains
  - PRAM requires $O(n/\log(n))$ processors and $O(\log(n))$ steps
Randomized Miller and Reif algorithm in BSP

So, how do we do tree contraction in the BSP model?

- perform $O(s)$ accesses and pointer chases needed in a PRAM step using $O(1)$ BSP supersteps and $O(s/P)$ communication
- with each step of rake/compress we decrease the number of nodes ($s$) geometrically
- need to assume the accesses/nodes are load balanced (can randomly permute initially)
- the communication cost then goes down geometrically
- after $O(\log(P))$ steps, the size of the tree is $O(n/P)$, so we can collect all nodes on one processor and contract the tree locally
- the total cost is then

\[ O(n/P \cdot \beta + \log(P) \cdot \alpha) \]