Divide & Conquer

Solve a problem P by solving a hierarchy of subproblems.

- The hierarchy is represented by a tree.
  - P is the root of the tree.
  - A node can be solved, if its children are solved.
  - Leaves are directly solvable.
- Crucial observation: often children of a node can be solved independently.
- A sequential divide & conquer algorithm can be parallelized, provided the process of solving a node from its children can be parallelized.

Sorting

- Odd-even transposition sort
  - sorts n keys with p processes in time $O(\frac{n}{p} \cdot \log \frac{n}{p} + n)$.
  - However poor compute/communicate ratio: communication dominates merging in each phase.
  - Efficient only for few, i.e., $p = O(\log_2 n)$ processes.
- Parallelizations of quicksort turn out to be superior.
  - Quicksort determines a splitter M, recursively sorts all keys smaller than M and then all keys larger than M.
  - Parallelize the task of determining all keys smaller, resp. larger than M.
  - The two subproblems should have roughly same size: M should be an approximate median.

Parallel Quicksort

(1) An approximate median M is determined.

(2) Each process i partitions its keys according to M and determines smaller_i, resp. larger_i, the number of keys smaller, resp. larger than M. // Time $O(\frac{n}{p})$.

(3) Apply MPI_Scan to determine the two prefix sums $\sum_{j=1}^{i} \text{smaller}_j$ and $\sum_{j=1}^{i} \text{larger}_j$ // Time $O(\log_2 p)$.

(4) Broadcast the number k of keys smaller than M.

- Recursively: the first $p \cdot \frac{k}{n-1}$ processes sort the smaller keys, the remaining processes sort the larger keys.
- Each process determines the new positions of keys and initiates a send. // Time $O(\frac{n}{p})$.

// The sequential quicksort is applied whenever only one process
// is assigned. Otherwise recursively sort the first k keys and the last
// $n - k - 1$ keys in parallel.

Parallel Quicksort: The Analysis

- If we do not charge for computing an approximate median:
  - $O(\log_2 p)$ recursive steps are performed until the sorting problem is reduced to size $O(\frac{n}{p})$.
  - There are at most $O(\frac{n}{p} + \log_2 p)$ compute and communication steps for one recursive step.
  - Including the final sorting step, the run time is bounded by $O((\frac{n}{p} + \log_2 p) \cdot \log_2 p + \frac{n}{p} \log_2 \frac{n}{p}) = O(\frac{n}{p} \cdot \log_2 n + \log_2^2 p)$.
  - Constant efficiency for $n = \Omega(p \cdot \log_2 p)$.
- To compute an approximate median:
  - randomly select a key M and broadcast M in time $O(\log_2 p)$
  - or let all processes determine a local median. A distinguished process gathers all local medians in time $O(p)$ and broadcasts an approximate median. The new running time is $O(\frac{n}{p} \cdot \log_2 n + p \cdot \log_2 p)$ with constant efficiency for $n = \Omega(p^2)$.

Parallel Quicksort: Discussion

- Computation versus communication:
  - The compute time per recursive step is dominated by the time $O(\frac{n}{p})$ to partition the keys.
  - Communication, when rearranging keys, also requires time $O(\frac{n}{p})$.
  - Communication dominates computation.
- The run time is dominated by the slowest recursive call.
  Determine local medians carefully:
  - For instance, let all processes sort their keys immediately and keep their keys sorted by merging when rearranging keys.
  - Each process chooses the exact median as local median without any delay.
  - The final sorting step is not necessary any more.

Sample Sort

- The idea: compress the $O(\log_2 p)$ iterations of quicksort into essentially one phase:
  - select a sorted sample of $p - 1$ splitters
  - and partition keys according to the sample.
- Questions:
  - when to sort and
  - how to determine the sample?

Sample Sort: The Algorithm

(1) Each process sorts its $\frac{n}{p}$ keys sequentially and
- afterwards determines a sample of size s .
- Process 1 gathers all samples,
- sorts the sample keys sequentially,
- determines the final sample S of size $p-1$ and broadcasts S.

// $O(\frac{n}{p}\log_2\frac{n}{p} + ps\log_2 ps)$ compute steps and
// communication $O(p \cdot s + p\log_2 p)$.

(2) Each process partitions its keys according to sample S.
Per process: the p sorted subsequences are distributed by an all-to-all personalized broadcast.
// Compute time $O(p \cdot \log_2\frac{n}{p})$ and communication $O(\frac{n}{p})$.

(3) Each process merges the p sorted sequences in time $O(\frac{n}{p} \cdot \log_2 p)$.

Sample Sort: The Analysis

- Assume that $p \cdot s \leq n/p$ and equivalently that $s \leq n/p^2$ holds.
  - Sorting all $\frac{n}{p}$ keys dominates over sorting the gathered samples.
  - The computing time is bounded by $O(\frac{n}{p} \cdot \log_2 \frac{n}{p} + p \cdot \log_2 \frac{n}{p} + \frac{n}{p} \cdot \log_2 p) = O(\frac{n}{p} \log_2 n + p \cdot \log_2 \frac{n}{p})$.
  - Communication is bounded by $O(p \cdot s + p \cdot \log_2 p + \frac{n}{p}) = O(p \cdot \log_2 p + \frac{n}{p})$, again since $p \cdot s \leq n/p$.
- Choice of parameters:
  - If $s = p$, then a good sample can be computed.
  - If $n = \Omega(p^3)$, then sample sort runs in time $O(\frac{n}{p} \cdot \log_2 n)$ with communication $O(\frac{n}{p})$.
  - Computation dominates over communication, if n is large.

Dynamic Programming

Solve a problem P by solving a hierarchy of subproblems.

- The hierarchy is represented by a directed graph without cycles.
  - P is the sink, i.e., the only node with fanout zero.
  - A node can be solved, if its immediate predecessors are solved.
  - Sources (nodes with fanin zero) are directly solvable.
- Typically subproblems have smaller complexity and all subproblems of same complexity can be solved independently.
- A sequential dynamic programming algorithm can be parallelized, provided all subproblems of "same complexity" can be solved in parallel.
- Dynamic programming has many applications for instance in bioinformatics.

Transitive Closure

We are given a directed graph $G = (V, E)$ with node set $V = \{1, \ldots, n\}$. Determine the transitive closure graph $\overline{G} = (V, \overline{E})$: the edge (i, j) belongs to $\overline{E}$ iff there is a path in G from i to j.

- $A[i,j] = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$ is the adjacency matrix of G.
- Warshall's Algorithm:

```
for k=1 to n do
    for i=1 to n do
        for j=1 to n do
            A[i,j] = A[i,j] or ( A[i,k] and A[k,j]);
```

- Run time $= O(n^3)$.
- Correctness:
  - Invariant: After the outer loop for $k-1$ completes, $A[i,j] = 1$ iff there is a path from i to j with intermediate nodes in $\{1, \ldots k-1\}$.
  - Invariant holds after the outer loop for k completes.

Parallelizing Warshall's Algorithm

- We parallelize the inner i, j-loops, but keep the outer k loop.
- For rowwise decomposition of the adjacency matrix A:
  - If we have reached k, then all updates $A[i,j] = A[i,j] \vee (A[i,k] \wedge A[k,j])$ have to be performed in parallel.
  - The process knowing row k has to broadcast its row.
  - Compute time per k iteration $= O(\frac{n^2}{p})$ and broadcast time $O(n \cdot \log_2 p)$. Total run time $O(\frac{n^3}{p} + n^2 \cdot \log_2 p)$.
- The checkerboard decomposition is better:
  - The process holding $A[i,k]$, resp. $A[k,j]$, has to broadcast its values in its row, resp. column, of the mesh of processes.
  - The communication time per k-iteration is bounded by $O(\frac{n}{\sqrt{p}} \log_2 \sqrt{p})$ and the total run time is $O(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} \log_2 \sqrt{p}))$.
- Many "small" broadcasts for the checkerboard decomposition replace the "big" broadcast for the rowwise decomposition.

The All-Pairs-Shortest-Path Problem

For a directed graph $G = (V, E)$ with nodes $V = \{1, \ldots, n\}$ and edge weights $w(e)$, determine the length of a shortest path from i to j for any pair (i, j) of nodes.

- $B[i,j] = \begin{cases} w(i,j) & (i,j) \in E, \\ \infty & \text{otherwise.} \end{cases}$ is the weighted distance matrix.
- Floyd's Algorithm:

```
for k=1 to n do
    for i=1 to n do
        for j=1 to n do
            B[i,j] = min ( B[i,j], B[i,k] + B[k,j] );
```

- Run time $= O(n^3)$.
- Correctness: After completing the iteration for k,

  $B[i,j]$ is the length of a shortest path from i to j with intermediate nodes in $\{1, \ldots, k\}$.

Parallelizing Floyd's Algorithm

- Proceed as for Warshall's Algorithm:
  - keep the outer for-loop and parallelize the inner i, j-loops.
  - The checkerboard decomposition is again better.
- Drawback: Warshall's and Flody's algorithm are adequate only for dense graphs (i.e., graphs with many edges).
  - Otherwise repeated applications of depth-first search are faster than Warshall
  - and repeated applications of Dijkstra are faster then Floyd.

Similarity of Two Strings

Often similarities of DNA or RNA sequences imply functional similarity. Determine the similarity of two sequences assuming unknown point mutations such as insertions, deletions and substitutions.

- View a DNA sequence as a word over the alphabet $\Sigma = \{\text{adenine, cytosine, guanine, thymine}\}$.
- How many insertions, deletions or substitutions of letters are necessary to obtain sequence v from sequence u?
- A slightly different perspective:
  - Imagine a blank symbol "$-$" inserted in several positions of u as well as v.
  - The new strings u^* and v^* are called an alignment iff they have identical length and $u_i^* \neq -$ or $v_i^* \neq -$ for all positions i.

Global Pairwise Alignment

Assume that u^*, v^* is an alignment of strings u and v.

- How to get from u to v?
  - If $u_i^* = -$, then insert v_i^* into u,
  - if $v_i^* = -$, then delete u_i^*
  - and if both $u_i^* \neq v_i^*$ are different from the blank symbol, then replace u_i^* for v_i^*.
- We want an alignment which verifies maximal similarity between u and v.
  - The function $d : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$ penalizes a disagreement with a low or negative score.
  - The similarity of an alignment u^*, v^* of u and v is defined by $s(u^*, v^*) = \sum_i d(u_i^*, v_i^*)$.

Determine an alignment u^*, v^* of u, v with maximal score $s(u^*, v^*)$.

The Idea

Let $u^i = u_1 \cdots u_i$ be a prefix of u and let $v^j = v_1 \cdots v_j$ be a prefix of v. Define $D(i,j)$ as the maximal score of an alignment of u^i and v^j.

- How does an optimal alignment between u^i and v^j look like?
  - Either it aligns the last letter of v^j with the blank symbol
  - or it aligns the last letter of u^i with the blank symbol
  - or it aligns the last letters of u^i and v^j.
- In either case the alignment of the remaining symbols has to be optimal.

$$D(i,j) = \max\{D(i,j-1)+d(-,v_j), D(i-1,j)+d(u_i,-), D(i-1,j-1)+d(u_i,v_j)\}.$$

The Algorithm of Needleman-Wunsch

(1) // Initialization

$D(0,0) = 0$;

for $(i = 1; i <= |u|; i++)$

  $D(i,0) = \sum_{k=1}^{i} d(u_k, -)$;

for $(j = 1; j <= |v|; j++)$

  $D(0,j) = \sum_{k=1}^{j} d(-, v_k)$;

(2) // Computation

for $(i = 1; i <= |u|; i++)$

  for $(j = 1; j <= |v|; j++)$

$$D(i,j) = \max \quad \begin{matrix} \{D(i,j-1) + d(-,v_j), D(i-1,j) + d(u_i,-), \\ D(i-1,j-1) + d(u_i,v_j)\} \end{matrix}$$

The run time is bounded by $O(|u| \cdot |v|)$.

A Parallelization of Needleman-Wunsch

- To determine $D(i, j)$ we only need to know $D(i', j')$ for $i' + j' < i + j$.
- Reorganize step (2) of Needleman-Wunsch:
  - we work with an outer k-loop and compute $D(i, k - i)$ with an inner i-loop.
  - Parallelize the inner loop.
- Choose the rowwise decomposition of the score matrix D. We have $|u| + |v|$ phases and compute all entries $D(i, k - i)$ in phase k.
  - If a process becomes active, it evaluates its portion of the score matrix D according to increasing component sum.
  - The process begins with boundary pairs $(i, k - i)$ whose solution it immediately communicates to the respective neighbor process.
- The run time for $|u| = |v| = n$:
  - per phase communicate two boundary pairs and compute in time $O(\frac{n}{p})$.
  - The total run time is bounded by $O(\frac{n^2}{p} + n)$.