Parallel Sorting

Piotr Luszczek
Sequential Sorting: Two Examples

- **Quicksort**
  - $\Theta(N \log N)$
  - Fast in practice
  - Unstable
    - Data with identical keys might end up in a different order
      - Many applications require those data to retain their order
    - Sensitive to median selection
      - Worst case complexity is quadratic
      - Using median of medians is complicated and costly

- **Heap sort**
  - $\Theta(N \log N)$
  - Slower in practice
    - Building and maintaining virtual tree of data: heap
  - Stable
  - Worst case complexity is the same as the average case
Naive Parallel Sort (Don’t Use!)

- Partitioning is simple:
  - Each process “p” gets N/P elements

- Repeat for each of N elements

- Complexity
  - \((N/P + \log P) \times N = N^2/P + N \log P\)
  - Very simple implementation:
    for (\(e = 0; e < N; ++e\))
      MPI_Reduce(..., MPI_MAX)
Improved Naive Parallel Sort

- Complexity
  - $\frac{N}{P} \log \frac{N}{P} + (\log P) \times N = \frac{N}{P} \log \frac{N}{P} + N \log P$
  - Very simple implementation:
    - `quicksort();`
    - `for (e = 0; e < N; ++e)`
    - `MPI_Reduce(..., MPI_MAX)`
Main Problem with Naive Implementations

- We must keep track of location of the largest element: 
  ```
  MPI_Reduce(..., MPI_MAXLOC)
  ```
- We must keep track of number of local elements: 
  ```
  MPI_Reduce(..., local[lastEl])
  ```
- We must keep track of where the value should go: 
  ```
  MPI_Reduce(currentRoot, ...)
  ```
- All processes need to know the location: 
  ```
  MPI_Bcast(currentRoot, &maxloc)
  ```
Towards Better Parallel Sort

2 processes

smallest values

largest values

4 processes

smallest values

no invariant guaranteed

largest values
Parallel Sort Using a Median: Hyperquicksort

- How to select median?
  1. Pick a process and value at random
  2. Sort values locally and pick a local median
  3. Global communication required for better median

- Keep the local values sorted
  - Initial cost: $\Theta(N/P \log N/P)$
  - Merge local old values with global new values: $\Theta(N/P)$
Divisibility, Network, and Median Selection

- Ideally
  - N is power of 2
    - Good load balancing
  - P is power of 2
    - Easy to find partner processor at each recursion level
  - Network is a hypercube
    - Easy to translate logical processor numbers to physical addresses
    - Bandwidth of the network grows with the network size
    - Latency to send a message increases slowly with network size

- Median selection
  - Local median is easy to find
    - Local values are kept sorted
  - Local median is usually not a global one
    - Imagine data that is already sorted
  - Bad median will create a load imbalance
    - Local data is no longer power of 2
    - It is costly to rebalance the load after every median