Parallel Algorithms: Sorting
Sorting

• Input:
  • Each processor has some records, for a total of N records
  • Let us assume each record has a 64 bit integer key, K
    • Plus some data, of B bytes

• Output:
  • At the end, we want processor 0 to hold M = N/P records with the smallest keys, and so on.
    • Relaxation: approximately N/P keys on each processor at the end i.e. allow a few percent over or under N/P
  • Operations allowed:
    • Comparison, or also breaking down the key
    • Generalization: keys are strings, with a large length limit (100?)
Uses of sorting

• Where can sorting be useful?
• Cosmology: N particles spread across processors
  • Partitioning: combine the x, y and z coordinates of the particles by some trickery and sort by this number
  • Piano-Hilbert space filling curves
• Distributed storage of data: simple lookup to find the home processor
Algorithm 1: Radix sort

- Use 16 bit buckets (arbitrary choice), and 4 phases
  - Why 4 phases?
- In the first phase, sort data based on *least significant* 16 bits: which 16 bits are least significant?
- Re-distribute data to processors, keeping the load balanced
- Repeat for next 3 phases using the next 16 bits
- We may return to this algorithm later, but notice:
  - In each phase, each processor has to partition its data and send some portions to each other processor
Example using 10 bins (Radix 10)

23  18  45  92  15  12  48

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9
Algorithm 2: Histogramming

• Comparison based (doesn’t assume keys are 64 bits)
• Let us consider a 1 phase algorithm first
• Processor 0 creates an initial separator keys (lets say P-1) keys
  • How?
    • Let us say uniformly dividing the name space in P parts
• Broadcasts to everyone
• Everyone calculates how many keys they own are in each partition
• Reduction brings totals to PE 0
• Adjust keys and Repeat until correct separators found.
• Then:
  • Each processor partitions its data based on the final separators, and sends each partition to its correct destination processor (keeping one set for itself).
Algorithm 3: Sorting by Regular Sampling

• n keys on p processors
  • So, each processor has n/p items

• Phase I: each processor sorts its own data
  • And identifies elements at p regular intervals (i.e. at index 0, n/p, ..)
    • I.e. finds p-1 equally spaced “pivots” that partition its data into p equal parts
    • This is the “regular sample” of its data

• Phase II:
  • All (p^2) regular samples are sent to one process, which sorts them
  • Selects p-1 equally spaced pivots again, and broadcasts to everyone

• Phase III:
  • Each processor partitions its data in p buckets based on the pivots
  • Sends j’th partition to processor j.

• Phase IV: Each processor “merges” the data it received.

m = n/p;
sort(A, m);
for (i=0; i<m; i*m/p <m)
    msg[i] = A[i*m/p];
send(msg, 8*p, BYTE, 0, TAG, ..)
if (I am 0)
    { for (i = 0; i<p; i++)
        recv(& (buf[i*p], 8*p, BYTE, i, TAG, ..)
        sort(buf, p*p);
        copy sample of buf int “sample”
        broadcast sample
    }
else call broadcast to recv sample.
For (dest = 0; dest < p; dest++)
    if dest is not me {
        copy data from A between two samples
        meant for dest
        send to dest }
Recv P messages and “merge” them.
Regular Sampling: Attributes

• What are some of the issues?
  • Does it work? (What does that mean)
    • Is the data sorted at the end?
    • n/p keys on each processor at the end?
      • Not quite. But bounded by 2n/p
    • Does it fit within memory at other times? (Transient memory)?
      • Sending messages while receiving them
      • Memory at the “root”
  • How efficient it is?
  • Scalable? Isoefficiency?
Quicksort: Sequential algorithm

• Choose a pivot (say, the first value)
• Partition the data into two sub-arrays
  • One contains values smaller than the pivot
  • The other contains value larger than or equal to pivot
  • The sizes of those may be unequal
• Sort each sub-array separately, using the same method
Quicksort: Parallelization

• Data is distributed
  • So each processor has $M$ keys
    • Let's assume equal distribution
    • But if not, spend some time equalizing it: one scan and one all-to-all

• Recursive formulation:
  • A group of processes, with ranks $[min, ..max]$ will sort the data that’s available on the same set of processes
  • We have to make sure this is true in each recursive call

• A good pivot is useful
  • Let's assume its selected somehow by one processor (say min)
Parallel quicksort

• Broadcast pivot to everyone in the set [min..max]
  • Make a communicator? Or do point-to-point sends?

• Partitioning wrt the pivot: every processor $i$ partitions its data into 2 sets: $\text{Smaller}_i$, $\text{Larger}_i$

• We have to find a processor $\text{mid}$, such that [min..mid] and [mid+1..Max] will do the recursive work at the next level.
  • Calculate Sum of sizes of $\text{smaller}_i$ and $\text{larger}_i$, and calculate mid to divide processors in the same ratio
  • Use prefix-sum (twice) to decide where to sent all my data