

17 Sorting

One of the few problems that seems absolutely to require global shared memory is sorting of data.

- Consider distributing the data to be sorted among several processors, as in a beowulf cluster.
- Sort the data locally, and then attempt to merge it.
- We can arrange for any possible permutation of elements to be stored across the multiple processors, so that a max number of messages would need to be sent ($n \lg n$ messages) in order to sort n things.

17.1 Bubble Sort

```
for(i = 1; i < n; i++)
{
    for(j = i+1; j <= n; j++)
    {
        if(a[i] > a[j])
        {
            SWAP(a[i],a[j])
        }
    } /* end j loop */
} /* end i loop */
```

- Order n^2 running time

17.2 Quicksort

CHOOSE PIVOT ELEMENT $a[j]$

SPLIT $a[.]$ into the set A of elements $< a[j]$ and the set B $> a[j]$

RECURSIVELY CALL quicksort on A and on B

- SPLIT can be done in linear time
- If our pivot divides $a[.]$ into a fixed fraction on each side with each call, then order $n \lg n$ on average
- Usually we determine the pivot by median of three elements, or such

17.3 Quicksort: Claim: SPLIT can be done in linear time

Walk a pointer from the left (“small”) end of the array. As long as the elements are less than the pivot value, keep bumping the pointer. When you encounter an element that is larger than the pivot value, pull it off to the side.

Now walk a pointer from the right (“large”) end of the array. When you encounter an element smaller than the pivot value, swap that element with the one found in the walk from the left edge, and return to the left-edge walk.

When you hit the midpoint, adjust as needed.

This takes only as many comparisons as there are elements in the array, and only as many data moves as are necessary to move the small values to the left and the large values to the right of the pivot point.

17.4 Quicksort: Claim: If our pivot divides $a[.]$ into a fixed fraction on each side with each call, then order $n \lg n$ on average

If the pivot selection breaks the array into a fraction of size $\alpha < 1/2$ and a fraction of size $\beta = 1 - \alpha > 1/2$ at every step, then the cost is no worse than:

$$\begin{aligned} T(n) &= T(\alpha n) + T(\beta n) + n - 1 \\ &= T(\alpha^2 n) + T(\alpha\beta n) + T(\beta\alpha n) + T(\beta^2 n) + 2(n - 1) \\ &= T(\alpha^3 n) + T(\alpha^2\beta n) + T(\alpha\beta^2 n) + T(\beta^3 n) + 3(n - 1) \\ &\leq 2^k T(\beta^k n) + k(n - 1) \end{aligned}$$

We carry the recursion out until $\beta^k n = 1$, that is, until $k = \log_{1/\beta} n$. The main term is the $k(n - 1)$, which is now $O(n \log n)$.

17.5 Heapsort

- A **min-heap** is a binary tree, with data stored at every node, such that the value stored in the parent node is **smaller** than the value stored in either child node.
- An array converted to a heap has $a[i] > a[2*i]$ and $a[i] > a[2*i+1]$.

CREATE A HEAP from an array $a[.]$

```
for(i = 1; i < n; i++)
```

```
{
```

```
    PULL OFF element  $a[1]$  (the smallest element in the heap)
```

```
    RE-CREATE the heap
```

```
} /* end i loop */
```

- $\lg n$ each time to re-build the heap
- $n \lg n$ worst case to create the heap initially (can in fact show that this is order n)
- So $n \lg n$ to do heap sort

17.6 Merge Sort

- If A and B are two sorted lists $n/2$ long, we can merge them with $n - 1$ comparisons

SORT all pairs of elements ($n/2$ times 1 comparisons)

MERGE SORT pairs to produce sorted lists that are 4 long ($n/2$ times 7 comparisons)

MERGE SORT 4-long lists to produce 8-long sorted lists ($n/4$ times 15 comparisons)

...

- Overall, $n \lg n$ comparisons

17.7 Theoretical Lower Bounds

- Theorem: One cannot sort n items by comparison in fewer than $O(n \lg n)$ comparisons worst case.
- Proof: With n items, we have $n!$ possible permutations, and thus by Stirling's formula we have $n! \approx (n/e)^n$ possible sorted lists. This is

$$2^{n \lg(n/e)} > 2^{(n \lg n)/2}$$

leaf nodes. If we build a binary decision tree with that many leaf nodes, it will take at least $(n \lg n)/2$ levels in the tree. It thus takes $\Theta(n \lg n)$ decisions to determine which permutation you have, if you are presented with an arbitrary array.

17.8 PRAM Sorting

- We have $n \lg n$ sequential sorting algorithms, and we have $n \lg n$ best possible sorting time.
- It is trivial to use n processors to get linear sorting time: Let n processors each handle one element and run through the entire array. Each processor determines the rank order of its element, and then they all write their element into the correct location in the array.
- With global memory, and n processors, we would like to achieve $\lg n$ running time. We can, but it's a difficult theorem to prove. We can get order n or better sorting time in several ways.
- Parallel Bubble Sort: With a $2n$ -long array, we can sort in $2n$ steps.
- Parallel Quick Sort: Choose a pivot element. Compare all elements against the pivot (n processors, 1 time step) Store into two different arrays. Call recursively $\lg n$ times, for $\lg^2 n$ running time.
- Parallel Heap Sort: With a processor at each node of the heap, we can compute in one tick how to rebuild the heap, and thus achieve sorting in linear time.
- None of these is really practical. With k processors, we could get a speedup of almost k on quicksort, but this *does* require shared memory, and therefore does not scale.

17.9 Lower Bounds on Distributed Parallel Sort

- Assume two distributed blocks of n elements each. Even go so far as to assume that the two blocks are each sorted. What's the cost of merging them into one sorted block of length $2n$?
- Best case insertion sort: binary search to put the first element of Block A into position in Block B. It is at least one data communication and then $\lg n$ probes to do this. And then one more data comm and $\lg(n - 1)$ probes for the second element. And thus

$$\lg n + \lg(n - 1) + \dots + 1 = \lg(n!) \approx n \lg(n/e)$$

probes to merge, and n data communications.

- Best case merge sort: pass all n elements of A to the processor with Block B, and do a merge in n comparisons.
- Bottom line, part A: We *must* send an element from one block to the other in order to put it into its proper location (i.e., we cannot sort what we have not looked at).
- A parallel merge sort would take $\lg n$ merge steps with at least $n/2$ elements sent each time. Can we do better?
- Yes, *probabilistically*, we should be able to do better.

17.10 Scalable Parallel Quicksort by Random Sampling

- Begin by distributing the data in blocks to all processors. Hence n items to p processors and each processor has n/p items.
- Each processor sorts locally using its favorite algorithm.
- Now comes the statistical part. If we assume that we have randomly distributed data, then each processor's data will more or less interleave with every other processor's data. Let's count on this being true.
- Every processor samples locally to be able to break up its data into p sub-blocks, so each processor generates $p - 1$ samples. Every processor sends the samples to the head node, which sorts the $p(p - 1)$ samples and thus produces a global set of $p - 1$ breakpoints.
- If in fact the data are randomly distributed, then these global samples can be used to break up the global data set into p sub-blocks.
- Broadcast the $p - 1$ breakpoints to all processors.
- Each processor sends the data smaller than the first breakpoint to processor zero. And the data between the first and the second breakpoint to processor one. And the data between the second and third to processor two. And so forth.

- If the data are randomly distributed, and our sample reflects the data, then all processors now have the same amount of data, and the data for processor i are all smaller than the data for processor $i + 1$, for all i .
- Each processor sorts locally, and we can read off the sorted list processor by processor.
- And we only have to move n items twice, once to distribute initially, and once after we have distributed the global samples.

17.11 Analysis: Scalable Parallel Quicksort by Random Sampling

- The first sort is locally of n/p elements, and thus takes $\Theta((n/p) \lg(n/p))$ time
- We assume that $n \gg p$ and thus that the number of samples is small compared to the amount of data, so gathering the samples doesn't cost us in data transmission time but *does* cost us the $\lg p$ message passing time for a gather.
- We charge zero for sampling locally.
- Sorting the p^2 samples on the head node costs $\Theta(p^2 \lg(p^2)) = \Theta(p^2 \lg(p))$ time
- We charge zero for sampling on the head node.
- Broadcasting the p samples back is again no cost for data but $\lg p$ for the broadcast
- Each process now sends $\frac{p-1}{p} \frac{n}{p} \approx \frac{n}{p}$ data items in an all-to-all communication. This *does* cost in data time, either n/p if we have the full bandwidth for such a communication, or else n if we don't.
- The local merge of p blocks each of length n/p costs $\Theta((n/p) \lg p)$ time
- Overall, then, the cost is

$$\frac{n}{p} \lg \frac{n}{p} + p^2 \lg p + \frac{n}{p} \lg p$$

- Cost is

$$\frac{n}{p} \lg \frac{n}{p} + p^2 \lg p + \frac{n}{p} \lg p$$

- Assume $n \gg p$ and then the $p^2 \lg p$ disappears
- So $\Theta(\frac{n}{p} \lg n + \lg p)$ compute and $\Theta(\frac{n}{p} + \lg p) = \Theta(n/p)$ communication
- Isoefficiency: We need

$$n \lg n > Cp \left(\frac{n}{p} \right) \lg p = Cn \lg p$$

$$\lg n > C \lg p$$

$$n > p^C$$

- Memory per processor must grow like p^{C-1}
- This is not great, but there is no way to sort in a different way. The data must be stored somewhere.