Sorting

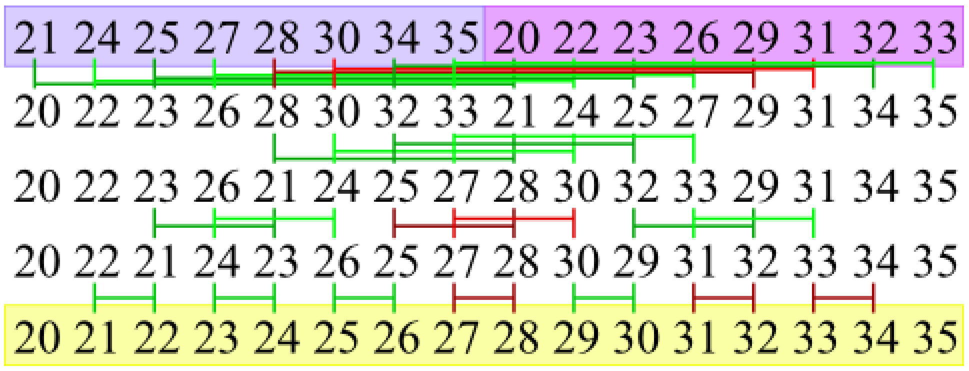
So far we have been working with very elementary problems — problems where the single-processor solution is so straightforward that in classical algorithms we rarely discuss the problems at all. Now we'll turn to one of the most heavily studied classical problems of all: sorting. How can we sort an array of numbers quickly when we have many processors?

We know we can sort an n-element array on a single processor in $O(n \log n)$ time. With p processors, then, we would hope to find an algorithm that takes $O((n \log n) / p)$ time. We won't quite achieve that here, but instead we'll look at an algorithm that takes $O\big((n ((\log p)^2 + \log n) / p)\big)$ time. Our algorithm will be based on the mergesort algorithm. It's natural to look at mergesort, because it's a simple divide-and-conquer algorithm. Divide-and-conquer algorithms are particularly attractive for multiprocessor systems: After splitting the problem into subproblems, we split our processors to process each subproblem simultaneously, and then we need a way to use all our processors combine the subproblems' solutions together.

(Incidentally, there *is* a $O((n \log n) / p)$ algorithm for parallel sorting, developed in 1983 by Ajtai, Komlós, and Szemerédi and subsequently simplified by M. S. Paterson in 1990. Even the simplified version, though, is more complex than what we want to study here. What's more, the multiplicative constant hidden by the big-O notation turns out to be quite large — large enough to make it less efficient in the real world than the algorithm we'll study, which was developed by Ken Batcher in 1968.)

4.1. Merging

Our first problem is determining how to merge two arrays of similar length. For the moment, we'll imagine that we have as many processors as we have array elements. Below is a diagram of how to merge two segments, each with 8 elements.



The first round here consists of comparing the elements of each sorted half with the element at the same index in the other sorted half; and for each pair, we will swap the numbers if the second number is less than the first. Thus, the 21 and 20 are compared, and since 20 is less, it is moved into the processor 0, while the 21 moves up to processor 8. At the same time, processors 1 and 9 compare 24 and 22, and since they are out of order, these numbers are swapped. Note, though, that the 28 and 31 are compared but not moved, since the lesser is already at the smaller-index processor.

The second round of the above diagram involves several comparisons between processors that are 4 apart. And the third round involves comparisons between processors that are 2 apart, and finally there are comparisons between adjacent processors.

We've illustrated the process with a single number for each processor. In fact, each processor will actually have a whole segment of data. Where our above diagram indicates that two processors should compare numbers, what will actually happen is that the two processors will communicate their respective segments to each other and each will merge the two segments together. The lower-index processor keeps the lower half of the merged result, while the upper-index processor keeps the upper half.

Below is some pseudocode showing how this might be implemented. In order to facilitate its usage in mergesort later, this pseudocode is written imagining that only a subset of the processors contain the two arrays to be merged.

```
void merge(int firstPid, int numProcs) {
    // Note: numProcs must be a power of 2, and firstPid must be a multiple o
f
    // numProcs. The numProcs / 2 processors starting from firstPid should ho
ld one
    // sorted array, while the next numProcs / 2 processors hold the other.

    int d = numProcs / 2;
    if(pid < firstPid + d) mergeFirst(pid + d);
    else                    mergeSecond(pid - d);
    while(d >= 2) {
        d /= 2;
        if((pid & d) != 0) {
            if(pid + d < firstPid + numProcs) mergeFirst(pid + d);
        } else {
            if(pid - d >= firstPid)           mergeSecond(pid - d);
        }
    }
}

void mergeFirst(int otherPid) {
    send(otherPid, segment);           // send my segment to partner
    otherSegment = receive(otherPid);  // receive partner's whole segment
    merge segment and otherSegment, keeping the first half in my segment vari
able;
}

void mergeSecond(int otherPid) {
    otherSegment = receive(otherPid);  // receive partner's whole segment
    send(otherPid, segment);           // send my segment to partner
    merge segment and otherSegment, keeping the second half in my segment var
iable;
}
```

Though the algorithm may make sense enough, it isn't at all obvious that it actually is guaranteed always to merge the two halves into one sorted array. The argument that it works is fairly involved, and we won't go into it here.

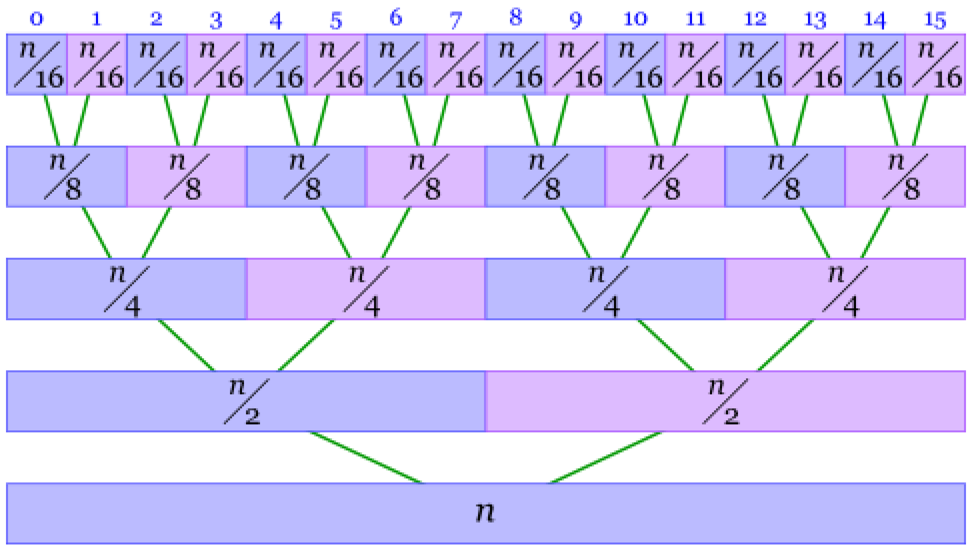
We will, though, examine how much time the algorithm takes. Since with each round the involved processors will send, receive, and merge segments of length $\lceil n / p \rceil$, each round of our algorithm will take $O(n / p)$ time. Because there are $\log_2 p$ rounds, the total time taken to merge both arrays is $O\big((n / p) \log p\big)$.

4.2. Mergesort

Now we know how to merge two $(n / 2)$-length arrays on p processors in $O\big((n / p) \log p\big)$ time. How can we use this merging algorithm to sort an array with n elements?

Before we can perform any sorting, we must first sort each individual processor's segment. You've studied single-processor sorting thoroughly before; we might as well use the quicksort algorithm here. Each processor has $\lceil n / p \rceil$ elements in its segment, so each processor will take $O\big((n / p) \log (n / p)\big) = O\big((n / p) \log n\big)$ time. All processors perform this quicksort simultaneously.

Now we will progressively merge sorted segments together, until the entire array is one large sorted segment, as diagrammed below.



This is accomplished using the following pseudocode, which uses the merge subroutine we developed in Section 4.1.

```
void mergeSort() {
    perform quicksort on my segment;
    int sortedProcs = 1;
    while(sortedProcs < procs) {
        sortedProcs *= 2;
        merge(pid & ~(sortedProcs - 1), sortedProcs);
    }
}
```

Let's continue our speed analysis by skipping to thinking about the final level of our diagram. For this last level, we have two halves of the array to merge, with n elements between them. We've already seen that merging these halves takes $O((n / p) \log p)$ time. Removing the big-O notation, this means that there is some constant c for which the time taken is at most $c\,((n / p) \log p)$.

Now we'll consider the next-to-last level of our diagram. Here, we have two merges happening simultaneously, each taking two sorted arrays with a total of $n / 2$ elements. Half of our processors are working on each merge, so each merge takes at most $c\,(((n / 2) / (p / 2)) \log (p / 2)) = c\,((n / p) \log (p / 2)) \leq c\,((n / p) \log p)$ time — the same bound we arrived at for the final level. That is the time taken for each of the two merges at this next-to-last level; but since each merge is occurring simultaneously on a different set of processors, it is also the total time taken for both merges.

Similarly, at the third-from-last level, we have four simultaneous merges, each performed by $p / 4$ processors merging two sorted arrays with a total of $n / 4$ elements. The time taken for each such merge is $c\,\big(((n / 4) / (p / 4)) \log (p / 4)\big) = c\,\big((n / p) \log (p / 4)\big) \le c\,\big((n / p) \log p\big)$. Again, because the merges occur simultaneously on different sets of processors, this is also the total time taken for this level of our diagram.

What we've seen is that for each of the last three levels of the diagram, the time taken is at most $c\big((n / p) \log p\big)$. Identical arguments will work for all other levels of the diagram except the first. There are $\log_2 p$ such levels, each being performed after the previous one is complete. Thus the total time taken for all levels but the first is at most $\big(c\,((n / p) \log p)\big) \log_2 p = O\big((n / p)\,(\log p)^2\big)$.

We've already seen that the first level — where each processor independently sorts its own segment — takes $O\big((n / p) \log n\big)$ time. Adding this in, we arrive at our total bound for mergesort of $O\big((n / p)\,((\log p)^2 + \log n)\big)$.

Source: http://www.toves.org/books/distalg/index.html#5