Homework 3: Analysis of Parallel Sorts

Geoffrey Ulman

March 4, 2010

1 Merge Sort

The merge sort implementation detailed in Table 1 is a straightforward adaption of the algorithm described in the homework wiki. The only item of note is step 7, which adds communication to indicate that a processor has been filled, and that low values should now be sent to the next lowest numbered processor. Table 1 describes the algorithm's steps along with a message count and size for each step. The total data sent at each step is the product of these two values.

1.1 Message Count

Summing the message counts from Table 1, remembering to multiply steps 5 and 6 by n, we arrive at a total message count of

$$2m^2 + mn - m \propto O(m^2 + mn)$$

Of particular note is the fact that the number of messages required depends on the size of the data array (because in the worst case a message might have to be sent for each data value). Also, the m^2 term is unsurprising because step 3 requires each node to broadcast to each other node.

1.2 Message Size

Multiplying the message counts and sizes in Table 1 gives an estimate of the total data transmitted by the algorithm of

$$2m^2 - 2m + mn + n \propto O(m^2 + mn)$$

Because almost all the messages sent by the parallel merge sort algorithm are size 1, it makes sense that the big-oh bound for the total message size is the same as the big-oh bound for the number of messages required.

*Worst case message size occurring when values are interleaved so that no processor has more than one smallest value at a time.

Table 1: Merge Sort Algorithm Detail

		Message	
	Description	Count	Size
1.	Send chunks of data to each processor	m	$\frac{n}{m}$
2.	Perform a local sort on each processor		
3.	Each processor sends its lowest value to each other processor	$m(m-1)$	1
4.	Repeat step 5 and 6 (worst case n times)		
5.	Processor with the smallest value sends all values smaller than second lowest value to the next non-full node	1	1*
6.	Processor from step 5 sends its new lowest value to each other processor	$m-1$	1
7.	Additionally, once during step 4 each processor must inform each other processor when it has $\frac{n}{m}$ values so data can be sent to the next lowest node	$m(m-1)$	1

Table 2: Bin Sort Algorithm Detail

		Message	
	Description	Count	Size
1.	Send chunks of data to each processor	m	$\frac{n}{m}$
2.	Each processor randomly picks a pivot value and sends that value to each other processor	$m(m-1)$	1
3.	Processors send their data to other processors based on the chosen pivot values	$m(m-1)$	$\frac{n}{m(m-1)}$
4.	Each processor sends how many values it has to each other processor	$m(m-1)$	1
5.	Each processor sends its excess data to nearby processors	$m-1$	$\frac{n}{m(m-1)}$[†]
6.	Perform a local sort on each processor		

1.3 Performance

For pathologically large number of processors m, parallel merge sort will perform poorly for two reasons. First, because step 3 requires that each processor broadcast to every other processor, the message count will grow quickly as $O(m^2)$. Second, because step 4 loops over all n data elements and requires that each processor receive a message for each data element, the message count also grows as $O(mn)$. Therefore, large numbers of processors will have increasingly more overhead with large data sets. This is a serious problem because we ideally want to be able to tackle large data sets by increasing the number of processors we use.

As $\frac{n}{m}$ approaches 1 (i.e. one data element for each processor and thus $n = m$), the theoretic message count and message size becomes $O(n^2)$. While the sort on each processor is trivial, messages equivalent to the square of the number of data elements is clearly not feasible for large data sets (that is more messages sent than the number of comparisons, $O(n \log(n))$, that it would take to simply do the sort on a single processor!)

As $\frac{n}{m}$ becomes large, the m terms drop out and the theoretic count and size becomes $O(n)$. This is much better than the last case, however it means that the sort will essentially have to be conducted on a single processor. Thus, this case reduces to a single processor sort with $O(n)$ messaging overhead.

2 Bin Sort

The stages of my bin sort implementation can be summarized as *Bin Balance Sort.* Each processor first chooses an upper bound (at worst, by choosing a data value from those that it was assigned, or alternatively, by using prior knowledge about the data values to choose better bounds). The processors then communicate their upper bounds and exchange data so that each processor contains data between its upper and lower bounds (a processor's lower bound being the next smallest upper bound). Most likely, this will result in some processors ending up with more data than others. To fix this, each processor publishes the number of data values it ended up with. Processors with excess data values then pass their lowest data values to nearby processors with smaller bounds and their highest data values to nearby processors with higher bounds. How many data values each processor must send and which processor they must be sent to can be determined from the published per-processor data totals with no additional communication. After this transfer, each processor will have $\frac{n}{m}$ unsorted data values. Each processor then performs a local sort and the algorithm is complete.

The algorithm used in Table 2 step 5 requires some additional explanation, although space constraints prevent a thorough description. It is performed locally on each processor and when it completes each processor knows how many data values it must send (and to which other processors) to ensure that each processor ends up with $\frac{n}{m}$ data values. The basic idea is that processors with more than $\frac{n}{m}$ data values pass some of their lowest values to the processor with the next lowest pivot and some of their highest values to the processor with the next highest pivot. If those adjacent processors now have too much data, they in turn must pass off their lowest/highest values to the processors adjacent to them.

2.1 Message Count

Summing just the message counts from the above table, we arrive at a total message count of

$$3m^2 - m - 1 \propto O(m^2)$$

[†] Worst case assumes each data value must be moved to another processor and the n data values are evenly distributed among the $m(m-1)$ messages.

The total message count is dominated by the communication from each processor to every other and unaffected, unlike the parallel merge sort, by the size of the data array.

2.2 Message Size

Multiplying the message counts and sizes in the above table gives an estimate of the total data transmitted by the algorithm of

$$2m^2 - 2m + 2n + \frac{n}{m} \propto O(m^2 + n)$$

Bin sort, like merge sort, uses communication from each processor to every other, explaining the m^2 term. However, unlike merge sort, the total data transmitted depends only on the size of the data array n, not on nm. This is because when data values are exchanged they are sent to a specific processor (in steps 3 and 5) instead of to every processor, like in steps 5 and 6 of the merge sort algorithm.

2.3 Performance

For pathologically large number of processors m, parallel bin sort may perform poorly because both message size and message count increase as $O(m^2)$. However, the number of exchanges from each processor to every other processor is fixed (it happens three times in steps 2, 3, and 4). The algorithm is further helped by the fact that for non-pathological data sets, the total size of the messages transfered does not depend on mn, so the total amount of data transfered may be reasonable even for large data sets and processor counts.

As $\frac{n}{m}$ approaches 1, the theoretic message count and message size becomes $O(n^2)$. In the limit, both the bin and merge sort become very similar in this case. Each processor has only a single data value, and binning them is equivalent to the rearranging that merge sort performs.

As $\frac{n}{m}$ becomes large, the m terms drop out and the theoretic message count approaches $O(1)$ and the theoretic message size approaches $O(n)$. Intuitively, a constant number of messages are required because the data essentially resides on a single processor. In the limit, a single size n message is required to load the data onto the processor, then a local sort is performed.

3 Comparison

Simply comparing message counts and total message sizes, it appears that bin sort is the more efficient parallel algorithm. Merge sort eventually sends each data element to each processor, accounting for the $O(mn)$ terms in its message count and size bounds. Bin sort sends data to specific processors, never needing to broadcast data from each processor to all others (although bin sort does broadcast counts in this way, it does this a fixed number of times, as opposed to once for each data value). Because bin sort sends each piece of data a fixed number of times to a single processor, its message size bounds have $O(n)$ terms where merge sort has $O(mn)$. This is a critical difference between the algorithms, because it means tackling large data sets by adding processors results in significantly higher total message size for merge sort than for bin sort. Both algorithms suffer equally from the need to broadcast messages between all processors (accounting for the $O(m^2)$ terms in the message size and message count bounds of both algorithms). However, the point of developing parallel algorithms is to tackle large data sets (large n) with large numbers of processors (large m). Therefore, an algorithm like parallel merge sort, whose message size bound is a product of m and n, is severely disadvantaged.

The worst case performance of the two algorithms are quite similar. As stated in Section 2.3, as $\frac{n}{m}$ approaches 1 the algorithms perform very similarly. The algorithms are also similar in the other extreme. As $\frac{n}{m}$ grows large both algorithms eventually reduce to a single processor sort with some slight messaging overhead.

With prior information about the range and distribution of the data values, the bin sort algorithm can greatly reduce its communication by choosing good pivot values. Merge sort cannot take advantage of such information as effectively. Both algorithms handle pre-sorted data sets relatively well. With bin sort, after data is distributed to the processors, each processor will publish its pivot, recognize that it has no data to send to other processors, and proceed to the local sort. With merge sort, after each processor has sent its lowest value, one will recognize that all its values are less than the second lowest and send them in one chunk.

In general, parallel bin sort beats parallel merge sort both in situations where bandwidth is constrained (making minimizing total message size is critical) and where messaging overhead is high (making low total message count important). However, the merge sort is conceptually simpler, and can perform competitively for nearly sorted data sets (which often means that multiple smallest data values can be sent together in step 5).