

Lecture 19: Sorting – II

*Professor: S. Lennart Johnsson**TA: Wei Ding*

1 Histogramming and Sorting

Histogramming is common in many scientific and engineering applications. Histogramming is used extensively in image processing, but is also used, for instance, in physics applications where it may be of interest to count the number of particles in certain energy bands. Histogramming can also be used for sorting purposes.

We have already discussed a few sorting algorithms, namely

- odd–even transposition sort
- up–down sort
- shear sort
- heap sort

Odd–even transposition sort is suitable for sorting on linear arrays in that its sorting time is compatible with the worst case lower bound for sorting on linear arrays. However, the running time of the algorithm is the same whether the data is already sorted or not. The up–down sorter is suitable for sorting on a linear array, with data originally and finally external to the array. But, as for the odd–even transposition sort, the sorting time is independent of the data distribution. With data external to the array, a sorting time proportional to the number of elements to be sorted is a lower bound, since the elements must all be inspected (entered into the array), regardless of whether the elements are already sorted or not.

Shear sort is a very elegant sort for two–dimensional meshes, with a sorting time optimal within at most a small constant factor. But again, the algorithm is oblivious, i.e., the running time is independent of whether the data is almost sorted or not.

We discussed heap sort in the context of binary tree networks. But, with data external to the array, linear time was again the best achievable.

We will now first discuss histogramming, then the “classical” parallel sort known as bitonic sort, or Batcher sort after Kenneth Batcher [1]. Then we will discuss Radix sort and Sample sort followed by a discussion of a close to optimal Merge Sort.

1.1 Histogramming

The task of computing a histogram consists in counting the number of items that falls into a certain bin or bucket. An easy way to envision a parallel algorithm is to give each node a copy of each bucket and have the nodes independently sort their items into the buckets as a first step. In order to arrive at a histogram for the total data set, it is necessary to add the counts for each bucket in all nodes. Thus, a reduction must be performed for each bucket. With L buckets and N nodes, L reduction operations are required, with each reduction involving N copies of the same bucket. If all sums are required in one node, then L tree summations can be performed in $L + \log_2 N - 1$ steps, if pipelining is used. Each step corresponds to the communication of one element between a pair of nodes, and the addition of the operands being communicated. In a packet switched communication system, a step can be performed in unit time if the communication is between adjacent nodes, while in a circuit switched or wormhole routing system, a step is unit time if there is no contention for the required communication.

If the result of the histogramming shall be distributed evenly across all nodes instead of entirely assigned to one node, then an all-to-all reduction can be used, and the number of steps for $L \geq N$ is $L - \frac{L}{N}$. For $L < N$, there is not a sufficient number of buckets for an all-to-all reduction. Thus, after $\log_2 L$ steps there is only one “live” bucket per node, but there is still several copies of this bucket, more precisely, there are still $\frac{N}{L}$ copies of each bucket. The summation of the counts for these buckets can be performed in $\log_2 N - \log_2 L$ steps.

Thus, for a total of P elements, P counter updates are required for sequential histogramming. In the parallel histogramming outlined above, with the P elements distributed evenly across all N nodes, $\frac{P}{N}$ counter updates are made in the local phase. Then, an additional $L - \frac{L}{N}$ additions are made for global bucket summations for $L \geq N$, while $L + \log_2 N - \log_2 L - 1$ updates are required for $L < N$. Thus, the speedup of the arithmetic is $\sim N$ for $\frac{P}{N} \gg L$, while for $\frac{P}{N} \ll L$ the speedup is $\sim \frac{P}{L}$.

The storage need is $2P + LN$ for the elements, the rank assignments, and the bucket counters.

For few items per node relative to the number of buckets, or for very uneven distributions, the performance can be improved. The execution time for most systems is dominated by the “all-to-all reduction”. Whenever there are only a few buckets in a node that are non-empty, it may be beneficial with respect to performance only to communicate the counters for those buckets. As the summation of bucket counts from different nodes progresses, more and more buckets are likely to have non-zero counts. Thus, as the reduction process proceeds, the benefit from communicating only non-zero counters is likely to diminish. Moreover, the number of buckets for which a node is responsible to accumulate counts is reduced by a factor of two for each reduction step. Thus, the adaptivity of the reduction process is likely to offer the highest pay-off for the initial stages. A non-adaptive reduction may be ideal for the last several stages. We refer to the algorithm using adaptivity at some stages as the *adaptive* histogramming algorithm.

1.2 Distribution sort (bucket sort)

We will present a distribution sort algorithm making use of the histogramming algorithm above. An algorithm for sorting through distribution count is as follows

1. Assign elements to buckets
2. Perform an accumulation of the counts for all preceding buckets (parallel prefix)
3. Rank assignment
4. Permutation

As an example consider the set of numbers 5, 0, 5, 0, 9, 1, 8, 2, 6, 4, 1, 5, 6, 6, 7, 7. Assume that there is one bucket for each of the decimal numbers 0 – 9. Then the bucket counts and accumulation are as follows;

Bucket	0	1	2	3	4	5	6	7	8	9
Counts	2	2	1	0	1	3	3	2	1	1
Accumulation	0	2	4	5	5	6	9	12	14	15

The rank assignment for each element starts from the accumulation value for each bucket and adds one to the value for each element ranked. The result for our example is

Element	5	0	5	0	9	1	8	2	6	4	1	5	6	6	7	7
Rank	6	0	7	1	15	2	14	4	9	5	3	8	10	11	12	13

For our parallel distribution sort algorithm, the histogramming algorithm yields the global bucket count. Step 2, accumulation, constitutes a parallel +–prefix operation on the global bucket counts. For step 3, the rank assignment, it is necessary to assign a starting index for each copy of each bucket. This can be accomplished by running the reduction algorithm “backwards”. In this backwards phase, the rank of the first (or last) element of a group of copies of a bucket is computed recursively until the group size is a single bucket. In the forward phase, counts for copies of a bucket are added to arrive at a global count for the bucket. In the backward phase, the starting rank for the two counts added together is determined based on the global starting rank, and the number of elements for the two groups of buckets added to form the global count. The net effect is a parallel prefix operation on the bucket counts, with all copies of a bucket ordered before any copy of the next higher ordered bucket.

Thus, the parallel distribution sort algorithm we have outlined is as follows:

1. Local assignment of elements to buckets
2. Global count of the number of elements in a bucket

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

Initial Counts

8 ← ●	40 ← ●	72 ← ●	104 ← ●
● → 10	● → 42	● → 74	● → 106
12 ← ●	44 ← ●	76 ← ●	108 ← ●
● → 14	● → 46	● → 78	● → 110
16 ← ●	48 ← ●	80 ← ●	112 ← ●
● → 18	● → 50	● → 82	● → 114
20 ← ●	52 ← ●	84 ← ●	116 ← ●
● → 22	● → 54	● → 86	● → 118

Global Counting - Step 1

48 ← ●	●	176 ← ●	●				
●	52 ← ●	●	180 ← ●				
●	●	56 ← ●	●	●	184 ← ●	●	
●	●	●	60 ← ●	●	●	188 ← ●	●
64 ← ●	●	192 ← ●	●	●	●	●	●
●	68 ← ●	●	196 ← ●	●	●	●	●
●	●	72 ← ●	●	●	●	200 ← ●	●
●	●	●	76 ← ●	●	●	●	204 ← ●

Global Counting - Step 2

224 ← ●	●	●	●	●	●	●	●
●	232 ← ●	●	●	●	●	●	●
●	●	240 ← ●	●	●	●	●	●
●	●	●	248 ← ●	●	●	●	●
●	●	●	●	256 ← ●	●	●	●
●	●	●	●	●	264 ← ●	●	●
●	●	●	●	●	●	272 ← ●	●
●	●	●	●	●	●	●	280 ← ●

Global Counting - Step 3

Figure 1: Accumulation of bucket counts for distributed buckets (Global Counts).

3. Parallel prefix on the global bucket counts
4. Rank assignment to each copy of each bucket
5. Local rank assignment

Figures 1 and 2 show an example of steps 2 – 4. Together, these steps perform a parallel prefix operation on the bucket counts with all copies of bucket i ordered before any copy of bucket $i + 1$ for $i = \{0, 1, \dots, L - 1\}$.

For the nonadaptive algorithm Steps 2 and 4 each require $\frac{L}{N}(N - 1)$ additions in sequence if $L \geq N$, and $L - 1 + \log_2 N - \log_2 L$ otherwise, assuming an all-to-all reduction based on balanced tree summations. Step 4 can be performed using the same techniques. The

0	•	•	•	•	•	•	•
•	224	•	•	•	•	•	•
•	•	456	•	•	•	•	•
•	•	•	696	•	•	•	•
•	•	•	•	944	•	•	•
•	•	•	•	•	1200	•	•
•	•	•	•	•	•	1464	•
•	•	•	•	•	•	•	1736

Parallel Prefix

0	•	•	•	48	•	•	•
•	224	•	•	•	276	•	•
•	•	456	•	•	•	512	•
•	•	•	696	•	•	•	756
944	•	•	•	1008	•	•	•
•	1200	•	•	•	1268	•	•
•	•	1464	•	•	•	1536	•
•	•	•	1736	•	•	•	1812

Rank Assignment - Step 1

0	•	8	•	48	•	120	•
•	224	•	234	•	276	•	300
456	•	468	•	512	•	588	•
•	696	•	710	•	756	•	834
944	•	960	•	1008	•	1088	•
•	1200	•	1218	•	1268	•	1350
1464	•	1484	•	1536	•	1620	•
•	1736	•	1758	•	1812	•	1898

Rank Assignment - Step 2

0	0	8	24	48	60	120	168
224	225	234	251	276	309	300	349
556	458	468	486	512	546	588	638
696	699	710	729	756	791	834	885
944	948	960	980	1008	1044	1088	1140
1200	1205	1218	1239	1268	1305	1350	1403
1464	1470	1484	1506	1536	1574	1620	1674
1736	1743	1758	1781	1812	1851	1898	1953

Rank Assignment - Step 3

Figure 2: Accumulation of bucket counts for distributed buckets (Parallel Prefix and Rank Assignment).

arithmetic time for Step 3 is $2\frac{L/N}{N}(N - 1)$ for $L/N \geq N$, otherwise it is $2(L/N - 1 + \log_2 N - \log_2 L)$. The factor of two stems from the fact that the parallel prefix operation based on trees require two phases. A sequential algorithm covering all five phases requires $2P + L - 1$ arithmetic operations. Thus, the speedup offered by the parallel distribution count algorithm we have described is

$$\text{Speedup} = \frac{2P + L - 1}{2\frac{P}{N} + 2\frac{L}{N}(N - 1) + 2\frac{L/N}{N}(N - 1)} \approx \frac{2P + L - 1}{2(\frac{P}{N} + L)}$$

for a large number of buckets relative to the number of nodes. Thus, for $\frac{P}{N} \gg L$ the speedup is $\sim N$. For $\frac{P}{N} \ll L$ an adaptive algorithm should be used. For an adaptive algorithm the speedup is in the range $\sim \frac{P}{L}$ to $\sim N$.

For $L = O(\log_2 N)$ the speedup is

$$\text{Speedup} = \frac{2P + O(\log_2 N)}{2\frac{P}{N} + O(\log_2 N)}$$

which yields speedup $\sim N$ for $\frac{P}{N} \gtrsim \log_2 N$.

1.2.1 Mapping onto network architectures

What is the communication time for a linear array?

A complete binary tree?

A butterfly network?

A binary cube?

What would the arithmetic and communication times be for parallel prefix based on butterfly network emulation?

An interesting distribution sort algorithm for shared memory machines has been given by Hirschberg [3]. It performs rank assignment to M numbers sorted into L buckets using M processors and $(L + 2)M$ storage in $O(\log_2 M + \log_2 L)$ time. The algorithm is based on a computational model that allows an arbitrary number of processors to access in parallel any set of distinct storage locations.

1.3 Bitonic sort

Bitonic sort [1], sorts a set of P elements in $P\frac{\log_2 P(\log_2 P+1)}{2}$ steps. This sort parallelizes easily such that P elements can be sorted in $\frac{\log_2 P(\log_2 P+1)}{2}$ steps on P nodes. The algorithm sorts by merging sorted sequences recursively.

A *bitonic* sequence is a sequence with one point of inflection, or a sequence that can be given this property through cyclic rotation.

Definition 1 A *bitonic sequence* is a concatenation of one ascending and one descending sequence. We also refer to any sequence that after a suitable cut and interchange of parts can be made into a bitonic sequence as a bitonic sequence. Formally, let P be the number of elements in the sequence x with indices $\mathcal{I} = \{0, 1, \dots, P-1\}$. Then, there exist an index $k \in \mathcal{I}$ and an integer $K < P$ such that for the index set $\mathcal{A} = \{k, (k+1) \bmod P, \dots, (k+K) \bmod P\}$ $x_i \leq x_{(i+1) \bmod P}$, $i \in \mathcal{A}$. Similarly, there exist an index $l \in \mathcal{I}$ and an integer $L < P$ such that for the index set $\mathcal{D} = \{l, (l+1) \bmod P, \dots, (l+L) \bmod P\}$ $x_l \geq x_{(l+1) \bmod P}$. Moreover, $\mathcal{A} \cup \mathcal{D} = \mathcal{I}$ and $\mathcal{A} \cap \mathcal{D} = \emptyset$.

Lemma 1 If x_i , $i = \{0, 1, 2, \dots, P-1\}$, is a bitonic sequence of even length, then the sequences $y_i = \min(x_i, x_{i+P/2})$ and $z_i = \max(x_i, x_{i+P/2})$, for $i = \{0, 1, 2, \dots, P/2-1\}$ are also bitonic. Furthermore, $\max_i y_i \leq \min_i z_i$.

The last property guarantees that no element in the sequence y_i is greater than any element in the sequence z_i . Conversely, no element in the sequence z_i is smaller than any element in the sequence y_i . Thus, these two sequences are ordered with respect to each other. The first property guarantees that the ordering process can be applied recursively.

Proof: Assume that $x_0 \leq x_1 \leq x_2 \leq \dots \leq x_j \geq x_{j+1} \geq x_{j+2} \geq \dots \geq x_{P-1}$. This sequence is clearly bitonic. Assume $P/2 \leq j \leq P-1$. If not, then the sequence can be reversed to yield this result. Reversing the sequence affects neither the bitonic property, nor the min or max operations.

If $x_{P/2} \leq x_{P-1}$, then $x_i \leq x_{i+P/2}$, $i = \{0, 1, 2, \dots, P/2-1\}$ and $y_i = x_i$ and $z_i = x_{i+P/2}$, so the proposition holds. If $x_{P/2} > x_{P-1}$, then there exist a k , $j - P/2 \leq k \leq P/2$ such that $x_k \leq x_{k+P/2}$ while $x_{k+1} > x_{k+1+P/2}$, since the sequence is ascending for $i \leq j$ and descending for $i > j$. Then, $y_i = x_i$ and $z_i = x_{i+P/2}$ for $0 \leq i \leq k$, while for $k < i < P/2$ $y_i = x_{i+P/2}$ and $z_i = x_i$. Since $k > j$, the sequence y_i is clearly bitonic with

$$y_i \leq y_{i+1} \text{ for } 0 \leq i < k$$

$$y_i \geq y_{i+1} \text{ for } k < i < P/2 - 1$$

and

$$z_i \leq z_{i+1} \text{ for } 0 \leq i < j - P/2$$

$$z_i \geq z_{i+1} \text{ for } j - P/2 \leq i < k$$

$$z_i \leq z_{i+1} \text{ for } k \leq i < P/2 - 1$$

and $z_{P/2-1} < z_0$. Thus, the sequence z_i is also bitonic. Furthermore, $\max_i y_i = \max(x_k, x_{k+1+P/2}) \leq \min(x_{k+P/2}, x_{k+1}) = \min_i z_i$. ■

A bitonic merge can be constructed recursively as shown in Figure 3. Each stage requires 2^{p-1} comparison elements for the comparison of two bitonic sequences of length 2^{p-1} each. The number of stages is p . The first stage consists of

- Shuffle the bitonic input sequence
- Compare even–odd pairs of elements
- Unshuffle the sequence after comparison

After this stage there are two bitonic sequences of half size and ordered with respect to each other. Bitonic mergers of four and eight elements are shown in Figures 4 and 5.

A sorting network can now be constructed recursively using sorting by merging. Thus, first we compare pairs of elements to create sorted sequences of length two, then we merge pairs of such sequences to create sorted sequences of length four, etc. Such a sorting network is shown in Figure 6. Shaded comparison cells outputs the results in opposite order compared to the nonshaded cells. The sorting of 2^p inputs requires $2^{p-1} \frac{p(p+1)}{2}$ comparison elements. Though bitonic sort requires $O(\log_2 P)$ more operations than a good sequential sort, it parallelizes easily and requires $\frac{\log_2 P(\log_2 P+1)}{2}$ time with a sufficient number of comparators. Thus, compared to the lower bound it is nonoptimal with a $O(\log_2 P)$ factor. Compared to shearsort it may however be $O(\frac{\sqrt{P}}{\log_2 P})$ faster.

1.4 Mapping of bitonic sort to networks

1.4.1 Mapping to a mesh

Bitonic sort is a simple and very powerful algorithm. It has been mapped to mesh configured multiprocessors by Thompson and Kung [9] to shuffled row–major order and by Nassimi and Sahni [8] to row–major order. The sorting time is about $7\sqrt{N}$ for a $\sqrt{N} \times \sqrt{N}$ array. Thus its complexity for row–major order is comparable to the mesh sort we described earlier for snake–like order.

Batcher has also proposed a slightly more efficient sort known as odd–even merge sort. It improves upon the bitonic sort with a small constant number of operations, but the leading term in the complexity expression is the same for bitonic sort and odd–even merge sort. The latter is described in [7] and has been adapted to mesh configured processor arrays by Thompson and Kung [9] and Hirschberg and Kumar [6].

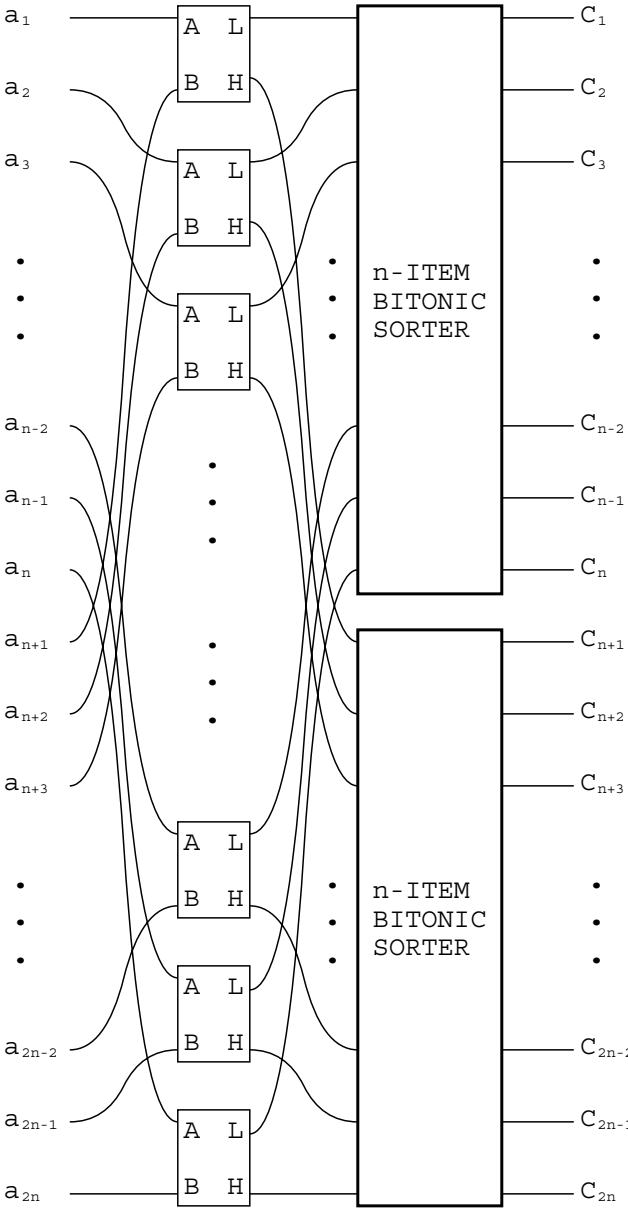


Figure 3: A bitonic merge of two sequences of length 2^{p-1} each.

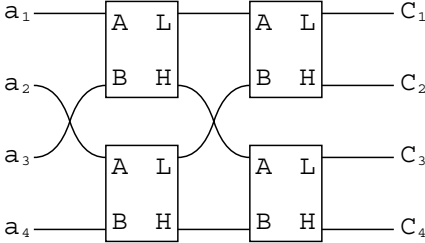


Figure 4: A bitonic merge of two sequences of length two each.

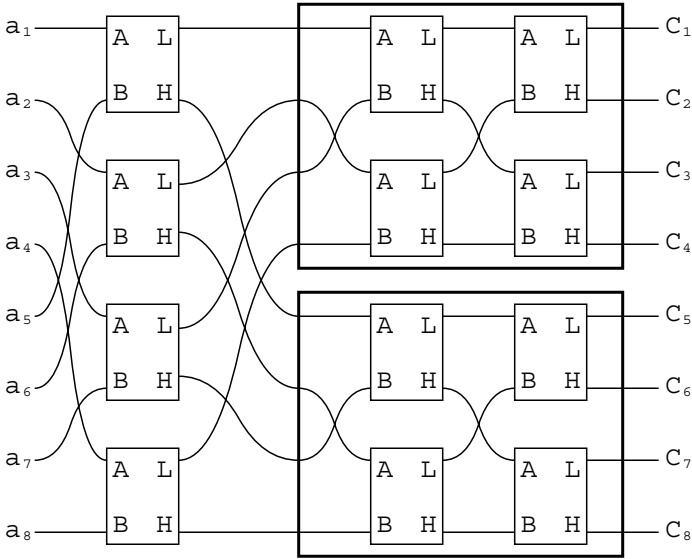


Figure 5: A bitonic merge of two sequences of length four each.

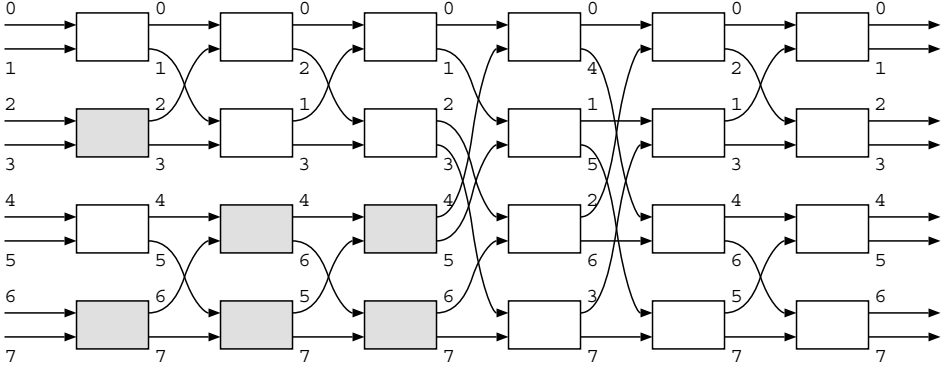


Figure 6: A bitonic sorting network for eight elements.

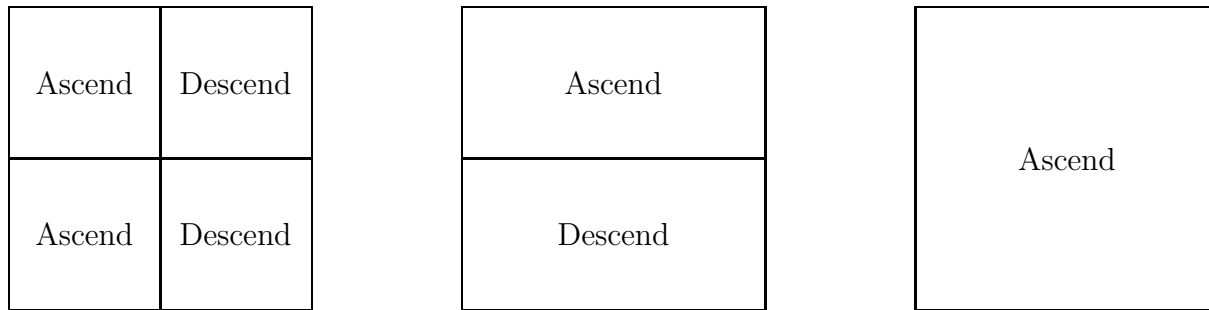


Figure 7: Recursive bitonic merge on a square mesh.

The adaptation of bitonic sort to meshes for sorting in row major order by Nassimi and Sahni can be described as follows:

Recursively merge sorted sequences in quadrants of the array by

1. merging two horizontally adjacent quadrants
2. followed by a vertical merge of the top and bottom half arrays

The idea is illustrated in Figure 7.

We consider the *vertical merge* first. For a final mesh of size $2^i \times 2^i$, the first i steps of the vertical merge imply communication only within columns, and result in 2^i bitonic sequences, one for each row. The sequences are ordered in row major order and it remains to perform a bitonic merge within each of the rows. The bitonic merge within columns can be accomplished by shifting the top and bottom towards the middle, performing the comparison, then shifting the results back, and repeating the process recursively. The number of shift operations with bidirectional communication is $2 \cdot 2^i/4 + 2 \cdot 2^{i-1}/8 + \dots = 2^i - 1$.

The row merge can be carried out in $2^i - 1$ communication steps as well. Thus, the vertical merge using the bitonic merge procedure can be performed in time $2 \cdot 2^i - 2$ communication steps.

Could odd–even transposition sort have been used for the column merge and the row merge of the vertical merge?

The *horizontal merge* is a bit more complex in order to guarantee a row major order. After a horizontal shift of two quadrants towards the middle, the first $i - 1$ bitonic merge steps must result in 2^{i-1} bitonic sequences ordered in row major order, followed by another $i - 1$ merge steps for finalizing the merge within row segments of length 2^{i-1} . Thus, after having shifted the quadrants into position for comparisons, column merge must take place to achieve row major order. The first comparison does not require any communication, but then, the exchange distances are $2^{i-1}/2, 2^{i-1}/4, \dots, 1$. The total

number of exchanges for the column merge phase of the horizontal merge is $2^{i-1} - 1$. At this point, it remains to carry out the merge within sequences corresponding to half rows. This can be made within the center section of the array in consecutive order in $2^{i-1}/2, 2^{i-1}/4, \dots, 1$ exchanges for a total of $2^{i-1} - 1$ exchanges. Then, the consecutively ordered data in the center section must be moved to the correct processor, which can be performed in another $2^{i-1}/2$ communication steps. Thus, for the horizontal merge we have

Shift to center:	$2^{i-1}/2$
Column merge:	$2^{i-1} - 1$
Row merge:	$2^{i-1} - 1$
Move result:	$2^{i-1}/2$
Total :	$3 \cdot 2^{i-1} - 2$

Hence, the total number of communication steps is $2 \cdot 2^i + 3 \cdot 2^{i-1} - 4$ for the merge of four quadrants into a single quadrant.

Thus, for sorting, the communication time is $T_c(2^{2i}) = T_c(2^{2(i-1)}) + \frac{7}{2}2^i - 4$, or $T_c(N) = 7\sqrt{N} - 2\log_2 N = \text{const}$.

The number of comparisons for the vertical merge is $2i - 1$ and for the horizontal merge it is $2(i - 1)$. Hence, the total number of comparisons is $\frac{\log_2 N(\log_2 N + 1)}{2}$.

The mapping described above can be generalized to higher dimensional meshes. A particular case is the binary cube, which we will consider next.

1.5 Mapping to binary cubes

Bitonic sort maps easily to binary cubes. The bitonic merge of a sequence of length 2^p on an 2^p node cube starts by comparing elements in nodes that differ in their most significant bit, then in the next lower order bit, etc. Thus, the merge implies comparisons across dimensions $p - 1, p - 2, \dots, 1, 0$. A complete bitonic sort requires communication in dimensions $0, 1, 0, 2, 1, 0, \dots, p - 1, p - 2, \dots, 1, 0$.

1.5.1 Oversize data sets

If there are $P > N$ elements to be sorted, then bitonic sort can be emulated on the N nodes, with each performing the work for $\frac{P}{2N}$ comparators. With $P = 2^p$, in effect $p - n$ dimensions are local to a node. For each comparison stage, each node performs $\frac{P}{2N}$ comparisons, and the total number of comparisons in sequence is $\frac{P \log_2 P(\log_2 P + 1)}{4N}$. The creation of sorted sequences of length $\frac{P}{N}$ requires $\frac{P}{2N} \frac{\log_2 \frac{P}{N}(\log_2 \frac{P}{N} + 1)}{2}$ operations, which compares unfavorably with the time for a sequential sort, such as for instance merge-sort or radix-sort both of which would sort in time $O(\frac{P}{N} \log_2 \frac{P}{N})$.

Thus, for $P > N$ elements we seek to combine a good sequential sort with bitonic sort [4]. The bitonic sort is used for internode sorting, while sequential sort is used within a

node. The time complexity of the algorithm is $O(P \log_2 P/N)$ for $N \ll P$, and $O(\log_2^2 P)$ for $P \gtrsim N$. We consider sorting both in *cyclic* and *consecutive* order.

Cyclic order sort

The algorithm below sorts two N numbers allocated to N nodes. A mask determines whether sorting is performed in non-descending or non-ascending order, as shown in the algorithm below.

```

For  $i = 1, 2, \dots, n$  do
  If  $i < n$  do
    nodes  $(a_{n-1}, \dots, a_{i+1}, a_i, a_{i-1}, \dots, a_0, a_i = 1$ , set mask = 1.
    nodes  $(a_{n-1}, \dots, a_{i+1}, a_i, a_{i-1}, \dots, a_0, a_i = 0$ , set mask = 0.
  endif
  For  $j = i - 1, i - 2, \dots, 0$  do
    nodes  $(a_{n-1}, \dots, a_{j+1}, 1, a_{j-1}, \dots, a_0$ , send their elements to
    nodes  $(a_{n-1}, \dots, a_{j+1}, 0, a_{j-1}, \dots, a_0)$ , which compare local
    and received elements
    nodes with mask=0 keep the smaller element and
    nodes with mask=1 keeps the larger element
    rejected elements are sent to  $(a_{n-1}, \dots, a_{j+1}, 1, a_{j-1}, \dots, a_0)$ 
  endfor
endfor

```

The address of an element is denoted $(a_{n-1}a_{n-2} \dots a_1a_0)$. This bitonic sort requires a time of $n(n+1)(2t_{co} + t_{ce})/2$.

In sorting a sequence of 2^n numbers on 2^n nodes, only half of the nodes are used for comparison operations. 2^{n-1} nodes suffice to sort 2^n elements. Conversely, two sequences of 2^n elements each can be sorted in time $n(n+1)(2t_{co} + t_{ce})/2$ time on 2^n nodes, with exchanges being the primitive communication operation. The storage need per processor remains the same.

The bitonic merge of two sorted subsequences each of size 2^{k-1+n} requires $k+n$ steps. With cyclically stored sequences, the first k steps are local to each node. With cyclic data allocation the $k-1$ most significant bits of the indices of each sequence are in local memory. The result after those k steps is 2^k bitonic sequences ordered with respect to each other. Each sequence has one element per node. The remaining n steps are performed on pairs of sequences at a time, since two such sequences can be merged concurrently. Figure 8a shows two sorted subsequences, Figure 8 b their distribution across the 2^n nodes, and Figure 8 c the result of the k first steps of bitonic sort.

The first k steps of bitonic merge requires a time proportional to $2^{k-1}k$. But, the operation performed in each node is a merge of two sorted sequences. A good sequential merge algorithm can perform the operation in time of at most $2^k - 1$ comparisons. Thus, the merge of two cyclically stored sequences can be performed as a local sequential merge

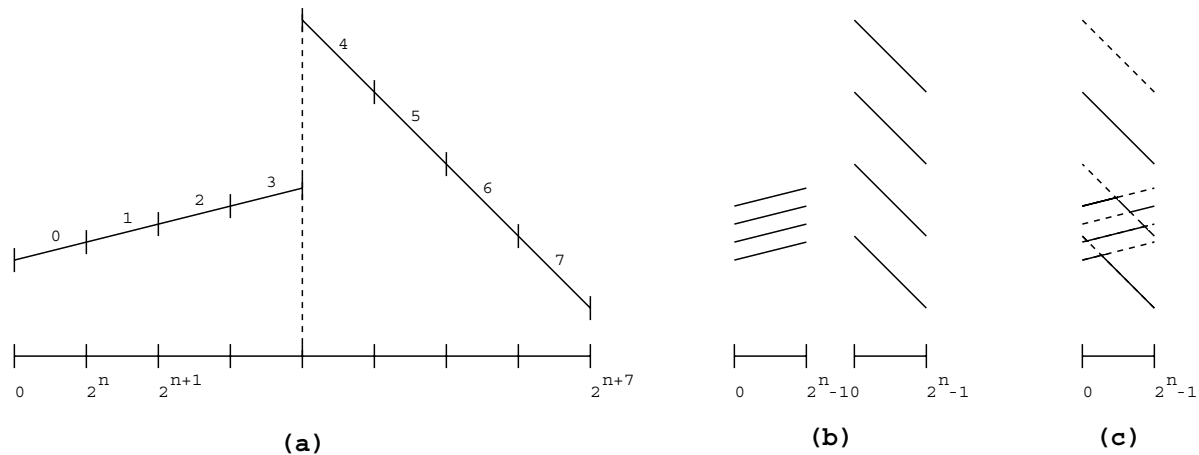


Figure 8: Cyclic storage of bitonic sequences.

followed by a the merge of 2^k independent bitonic sequences of length 2^n each, and with one element per node. The number of comparisons in sequence for the merge of the two sequences is $2^k - 1 + 2^{k-1}n$. The result is one sorted sequence stored cyclically over 2^n nodes.

To build a sorter for 2^{k+n} elements on 2^n nodes we can proceed recursively through recursion on the number of nodes. Assume that one sequence of 2^{k-1+n} elements is sorted cyclically on 2^{n-1} nodes in ascending order, and that another sequence of the same length is sorted cyclically in descending order on another set of 2^{n-1} nodes. Then, the goal is to create a sorted sequence of length 2^{k+n} distributed cyclically on all 2^n nodes. With the sequence allocated cyclically, a set of 2^{n-1} successive elements is allocated to successive nodes in each subset of nodes for each sequence. With the subset of nodes over which the ascending sequence is distributed being subset 0, and the other subset being subset 1, the ascending sequence can be rearranged to be stored cyclically over all 2^n nodes by sending every other data segment of length 2^{n-1} to the corresponding node in the other subset. With segments labeled $0, 1, \dots, 2^k - 1$, sending all odd segments of the ascending sequence in subset 0 to the corresponding nodes in subset 1 results in an ascending sequence stored cyclically over all 2^n nodes. The segment length is now 2^n and the number of segments for the ascending sequence is 2^{k-1} . Similarly, for the descending sequence, all even segments, $0, 2, 4, \dots$ are sent from subset 1 to the corresponding nodes in subset 0 to create a descending sequence stored cyclically over all 2^n nodes. Thus, the total time to sort 2^{k+n} numbers starting with sorted sequences in each node is $\sum_{i=1}^n (2^k - 1 + 2^{k-1}i) = n(2^k - 1 + 2^{k-2}(n + 1))$ and we have a total of $O(2^k k) + n(2^k - 1 + 2^{k-2}(n + 1))$ sequential comparisons for a sort.

Perform a local sort

For $i = 1$ **to** n **do**

Exchange odd subsequences in the 0-nodes w.r.t address bit $i - 1$
with the even subsequences in 1-nodes w.r.t the same bit

Perform a local merge

Perform a bitonic merge on sequences formed from corresponding
 memory locations across subsets of nodes of size 2^i .
endfor

A mask is used to determine whether sorting shall be performed in non-descending or non-ascending order.

Theorem 1 *Sorting of $P = 2^p$ elements into cyclic order on $N = 2^n$ nodes can be performed by a combination of local sort, local merge, and bitonic sort in $O(\frac{P}{N} \log_2 \frac{P}{N}) + (\frac{P}{N} - 1) \log_2 N + \frac{P \log_2 N (\log_2 N + 1)}{4N}$ comparisons.*

Consecutive sort

To sort the elements into a consecutive order we consider a bitonic merge of two sorted sequences of size 2^{k-1+n} with each sequence stored consecutively in 2^{n-1} nodes. In this case, the first n steps require internode communication. If after each comparison the result of the min operation is kept in the lower numbered node of a pair exchanging data, and the result of the max operation kept in the higher numbered node, then the first n steps result in 2^n bitonic sequences stored with one such sequence per node, and ordered in ascending order. The last k steps consist of a local bitonic merge in each node. The result is a sorted sequence stored consecutively. Instead of performing a local bitonic merge in $k2^{k-1}$ comparisons we perform a search for a min or max in k steps followed by a merge in $2^k - 1$ steps.

The first n steps can be viewed as the merge of 2^k bitonic sequences, each with one element per node, by observing that the comparisons in these steps only involves elements with the same local index in each node. Thus, starting with a local sort requiring a time of $O(k2^k)$ we create sorted sequences stored consecutively in subsets of nodes of size 2, 4, etc. The time is $O(k2^k) + \sum_{i=1}^n (2^{k-1}i + 2^k - 1 + k) = O(k2^k) + 2^{k-1} \frac{n(n+1)}{2} + (2^k - 1 + k)n$, or $O(\frac{P}{N} \log_2 \frac{P}{N}) + \frac{P \log_2 N (\log_2 N + 1)}{4N} + (\frac{P}{N} - 1 + \log_2 P - \log_2 N) \log_2 N$.

Perform a local sort in all nodes
For $i = 1$ **to** n **do**
 Perform a comparison–exchange between nodes differing
 in bits $i - 1$ through bit 0, keeping the smaller element
 in the lower numbered node and larger element in the
 higher numbered node
 Find the local maxima (or minima)
 Perform a local merge
endfor

Theorem 2 *Sorting of $P = 2^p$ elements into consecutive order on $N = 2^n$ nodes can be performed by a combination of local sort, local merge, and bitonic sort in $O(\frac{P}{N} \log_2 \frac{P}{N}) + (\frac{P}{N} - 1 + \log_2 \frac{P}{N}) \log_2 N + \frac{P \log_2 N (\log_2 N + 1)}{4N}$ comparisons.*

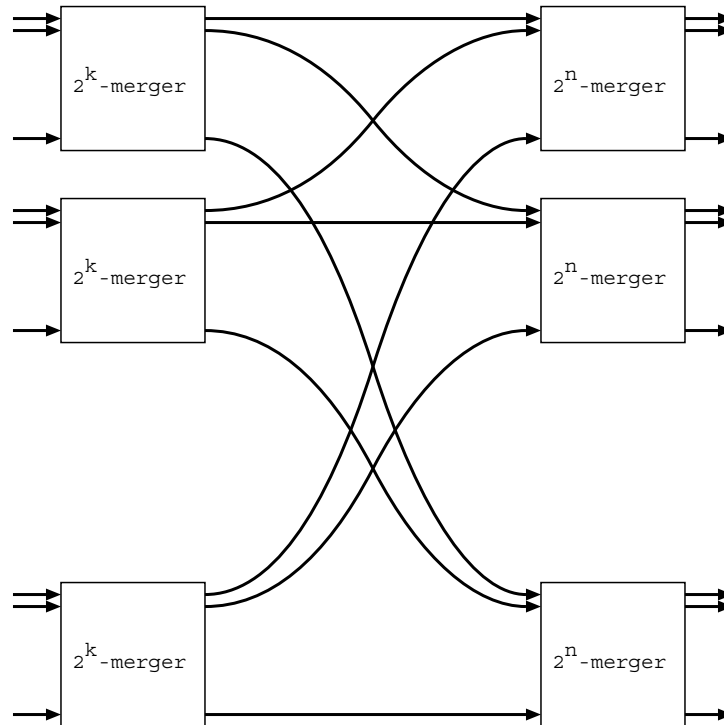


Figure 9: Merge of two sorted sequences for cyclic order.

Note that the consecutive sort is slightly less efficient than the cyclic sort due to the difference in the local merge.

Can the consecutive sort be made as efficient as the cyclic sort?

Discussion

The cyclic and consecutive sorting algorithms above can be viewed as a combination of sorters for sequences of different lengths. Thus, for instance in the case of the sorting in cyclic order, a merge phase of the two sequences with 2^{k-1+n} elements each, consists of 2^n mergers for pairs of sorted sequences of length 2^{k-1} each. After this phase, the view of the data is changed such that the next big phase consists of 2^k bitonic mergers for sequences of length 2^n each. Thus, conceptually the merge for cyclic order can be represented as shown in Figure 9.

We note that each bitonic merge constitutes a *normal* algorithm. The bits are traversed in a fixed order. Thus, in the case of bitonic sort on a binary cube, the communication for successive stages can be pipelined. The communication time for the merge of two sequences with 2^{k-1+n} elements each on 2^n nodes requires a time proportional to $2^{k-1} + n - 1$. In the cyclic order sort we have an additional step involving communication, namely the redistribution from cyclic order over 2^{n-1} nodes to cyclic order over 2^n nodes. This step requires communication time proportional to 2^{k-1} . Thus, consecutive order sort with the above algorithm on a binary cube requires a time proportional to $\frac{P}{2N} \log_2 N + \frac{\log_2 N (\log_2 N - 1)}{2}$, while the cyclic order sort requires communication proportional

Initial order		Step 1		Step 2		Step 3	
Decimal	Binary	Rank	Permute	Rank	Permute	Rank	Permute
5	101	6	110	5	100	4	000
6	110	1	100	1	000	1	001
4	100	2	110	6	101	5	010
7	111	7	010	7	001	2	100
6	110	3	000	2	110	6	101
2	010	4	101	3	110	7	110
1	001	8	111	8	010	3	110
0	000	5	001	4	111	8	111

Table 1: Radix sort applied to eight numbers.

to $2 \frac{P}{2N} \log_2 N + \frac{\log_2 N(\log_2 N - 1)}{2}$.

1.5.2 Mapping to shuffle–exchange networks

Bitonic sort maps well also to shuffle–exchange networks. Since comparison–exchange operations are performed on elements with indices that differ in successive index bits, what is required is to bring them into the least significant bit position. Once there, a comparison corresponds to communication across exchange edges.

1.6 Radix sort

For a description of sequential radix sort see for instance Knuth [5] pp 123 – 129.

A parallel radix–2 sort can be constructed as follows for keys encoded in p bits and the least significant bit being bit 0.

```

For  $i = 0$  to  $p - 1$  do
    Rank the keys with respect to bit  $i$ 
    Permute the keys according to their rank
endfor
    
```

As an example consider sorting the sequence 5, 6, 4, 7, 6, 2, 1, 0. Three bits are required for the encoding of the keys. Thus three iterations are required. Table 1 shows the steps of the algorithm above for this example.

Note that the relative order of already sorted bits is maintained in this algorithm. This radix sort algorithm is *stable*, i.e., subsequent steps do not destroy the partial order already performed.

Node	Initial order		Local count		Count on 1st lb		Count on 2nd lb		Scan		Rank distr. msb		Rank distr. 2nd msb		Rank
	Decimal	Binary	Bucket		Bucket		Bucket		Bucket		Bucket		Bucket		
			1	0	1	0	1	0	1	0	1	0	1	0	
0	5	101	1	1	1	2	1	5		0		0	5	0	5
	6	110													0
1	4	100	1	1	2	1	2	1			5		6	1	1
	7	111													6
2	6	110	0	2	0	3	0	3				2	0	2	2
	2	010													3
3	1	001	1	1	1	1	3	1	5		7		7	4	7
	0	000													4

Table 2: Ranking w.r.t. least significant binary digit.

Would a radix sort progressing from the most significant to the least significant digit be stable?

The radix sort is easily generalized to r -bit digits for a radix- 2^r sort. In such a case we have for b -bit keys

```

For  $i = 0$  to  $b - 1$  step  $r$  do
    Rank the keys with respect to the  $i$ th radix- $2^r$  digit
    Permute the keys according to their rank
endfor
    
```

A parallel implementation of radix sort as described here depends heavily upon the efficiency of parallel rank and permute.

The rank operation can be carried out using the distribution sort described earlier.

```

For  $i = 0$  to  $b - 1$  step  $r$  do
    Perform a bucket sort with respect to the  $i$ th radix- $2^r$  digit
endfor
    
```

Figure 2 gives an example of ranking using the core of the distribution sort algorithm above. In the example there are a total of eight elements distributed evenly over four nodes and the radix is two. Note, that the rank procedure is carried out such that stability is guaranteed.

The rank operation using the distribution sort requires a time of

$$T_{rdist} = c_1 \frac{P}{N} + c_2 \cdot 2(2^r(1 + \frac{1}{N}) + 2(n - r - 1))$$

where we have assumed that each communication is a unit time operation (either requiring adjacency communication or congestion free circuit switching or wormhole routing). For

Radix	Each inst.		All instances		Total
	Rank	Permute	Rank	Permute	
4	7.63	6.71	61.02	53.69	114.71
5	7.63	6.71	53.39	46.98	100.37
6	7.63	6.71	45.76	40.27	86.03
7	7.63	6.71	38.14	33.55	71.69
8	10.91	6.71	43.65	26.84	70.49
9	12.38	6.71	44.75	26.84	71.59
10	13.09	6.71	46.90	26.84	73.74
11	13.39	6.71	39.86	20.13	59.99
12	13.48	6.71	37.87	20.13	58.00
13	13.48	6.71	34.59	20.13	54.72
14	13.48	6.71	34.58	20.13	54.71

Table 3: Cycles per element on a CM-5 for rank assignment and permutations for randomly distributed data and 64k elements per vector unit.

the complete radix- 2^r sort a permutation must be included followed by a repetition $\lceil \frac{b}{r} \rceil$ times.

The total time is

$$T_{rsort} = \lceil \frac{b}{r} \rceil (c_1 \frac{P}{N} + c_2 \cdot 2(2^r(1 + \frac{1}{N}) + 2(n - r - 1)) + Perm(\frac{P}{N}, N))$$

The radix sort described above is faster than the “improved” radix sort in [2], in particular for high radices, and large number of elements per node.

From the expression for T_{rsort} it is clear that an optimum radix exists, and that it depends upon the relative cost of the distribution sort and the permutation. For a high cost of permutations, a high radix is desirable. An example of the tradeoff is given in Table 3. The increase in the time for the rank assignment as a function of the radix is mostly due to DRAM page faults.

1.7 Sample sort

Sample sort [2] has its name from the fact that it first selects a set of splitters defining bucket ranges, then sorts the set of elements into these buckets. Each processor is responsible for one bucket. The splitters are selected such that the bucket sizes are as evenly populated as possible in order to create good load-balance. Thus, the steps of the sample sort are

1. Select a set of $N - 1$ splitters for N processors.

2. Sort the splitters
3. Assign elements to buckets
4. Permute the elements
5. Local sorting

Selecting the splitters must be made with great care in order to achieve a reasonable load–balance for any set of elements. Thus, splitters are selected from the set to be sorted through a sampling process. In fact, instead of selecting $N - 1$ splitters, $sN < P$ candidate splitters are selected from the set of P elements. s is the *oversampling* ratio. The candidate splitters are sorted, and the actual splitters are chosen as the candidate splitters $s, 2s, 3s, \dots, (N - 1)s$. The candidate splitters may be sorted by any of the sorting methods discussed earlier, for instance radix sort. For $\frac{P}{N} \gg 1$, the number of candidate splitters is expected to be much smaller than P and the choice of sorting method for the candidate splitters not very critical for performance.

For small values of $\frac{P}{N}$, the advantage of sample sort over other sorts, such as bitonic sort and radix sort may be small at best.

Selecting the candidate splitters can be made without communication by letting each processor select s candidate splitters randomly from its $\frac{P}{N}$ local data elements, assuming an even initial allocation. In the implementation described in [2], the local data set was divided into s blocks, and a candidate splitter key selected randomly from each of the blocks.

The purpose of the oversampling is to reduce the risk of some buckets receiving a number of keys L much greater than $\frac{P}{N}$, the average number of keys per bucket. The quantity $\frac{L}{P/N}$ is known as the *bucket expansion*. It is a measure of the load–imbalance. We denote the expected value of the bucket expansion $\beta(s, P)$. The likelihood that β is greater than a given value $\alpha > 1$ is

$$Pr(\beta(s, P) > \alpha) \leq Pe^{-(1-\frac{1}{\alpha})^2 \alpha s / 2}$$

This relationship is proved in [2]. As an example, consider $s = 64$ and $P = 10^6$. Then $Pr(\beta > 2.5) \leq 10^{-6}$. Figure 10 shows both estimated bounds of the bucket expansion and observed bounds for a 1000 trials on a 1024 node CM-2 with a total of 10^6 keys.

For Phase 3, assigning elements to buckets, an *all-to-all broadcast* of splitters to processors is performed, followed by a completely local assignment of keys to buckets. The assignment of keys to buckets can be made through a binary search for each key.

Phase 4 is a route operation, since buckets are identified with processors and the processor destination is known from Phase 3.

After Phase 4 sorting within buckets remain, which is an entirely local sort.

The running times for the different phases are

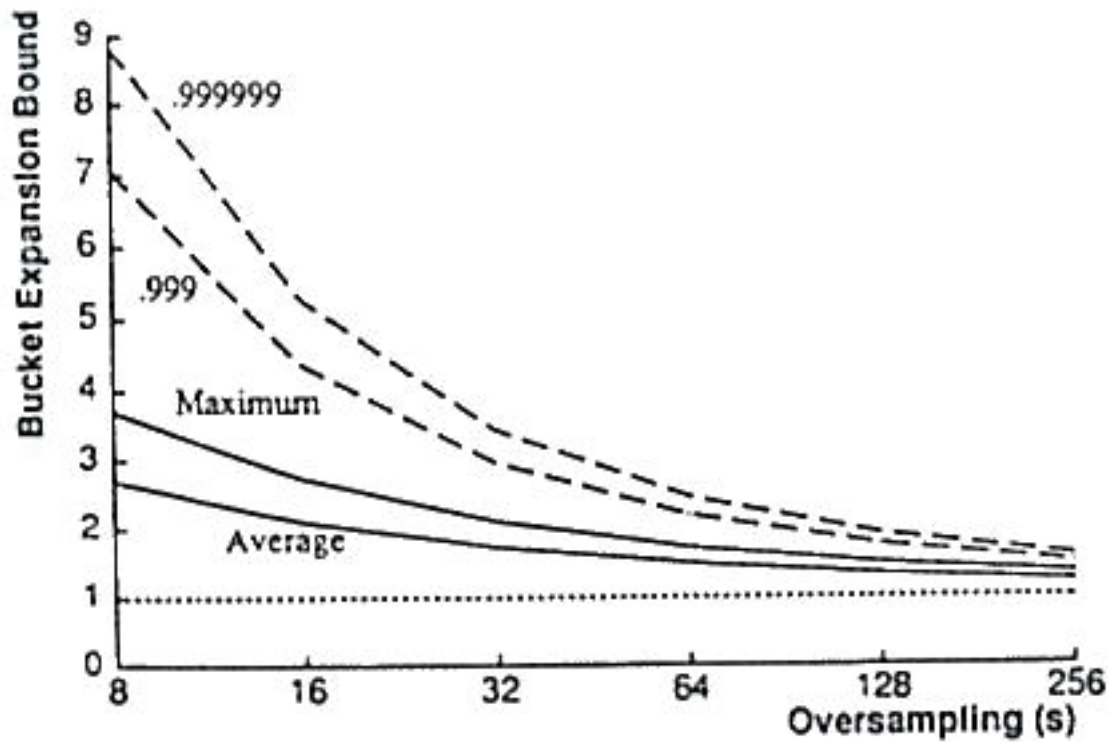


Figure 10: The relationship between bucket expansion and oversampling. The two upper curves shows the bucket expansions that are not exceeded with a probability of of 0.999 and 0.999999 respectively for one million keys and 1024 nodes. The two lower curves shows the observed maximum and average expansions for a 1000 trials.

$$T_1 = c_1 s$$

$$T_2 = c_2 s + c_3 n$$

$$T_3 = c_4 N + c_5 \frac{P}{N}$$

$$T_4 = c_6 \frac{P}{N}$$

$$T_5 = c_7 \frac{p}{N} \beta$$

In the above estimate c_1 accounts for the local selection of candidate splitters. T_2 covers the sort of the candidate splitters. The stated time is for a simple radix sort with $r = b$ requiring only a single permutation, which we assume is dominated by $\log_2 N$. c_4 accounts for the all-to-all broadcast, and effectively assumes a node limited communication model. A node limited communication model is also assumed for the estimate T_4 .

Whenever there may a large number of equal keys, then it may be necessary for load-balance to tag the keys with their addresses for Phase 3. If radix sort or any other stable sort is used for the sorting of candidate splitters, tagging need not be considered for phase 1 or 2. After phase 3 tags are no longer needed.

Figure 11 shows the running time for sample sort on a 1024 node CM-2 as a function of the oversampling ratio s for $\frac{P}{N} = 16384$. Figure 12 shows the time per key per node for sample sort as a function of the number of keys. An oversampling ratio of 32 was used for up to 4096 keys per node, and an oversampling ratio of 64 was used for a larger number of keys per node.

Table 4 gives a comparison of the times for bitonic sort, radix sort and sample sort on a 1024 node CM-2. The column “Memory” gives the relative memory requirements, and the column “Rank” gives the relative times for Rank and sort.

1.8 Tree sort

Tree sort suffers from the same problems as heap sort when parallelized straightforwardly.

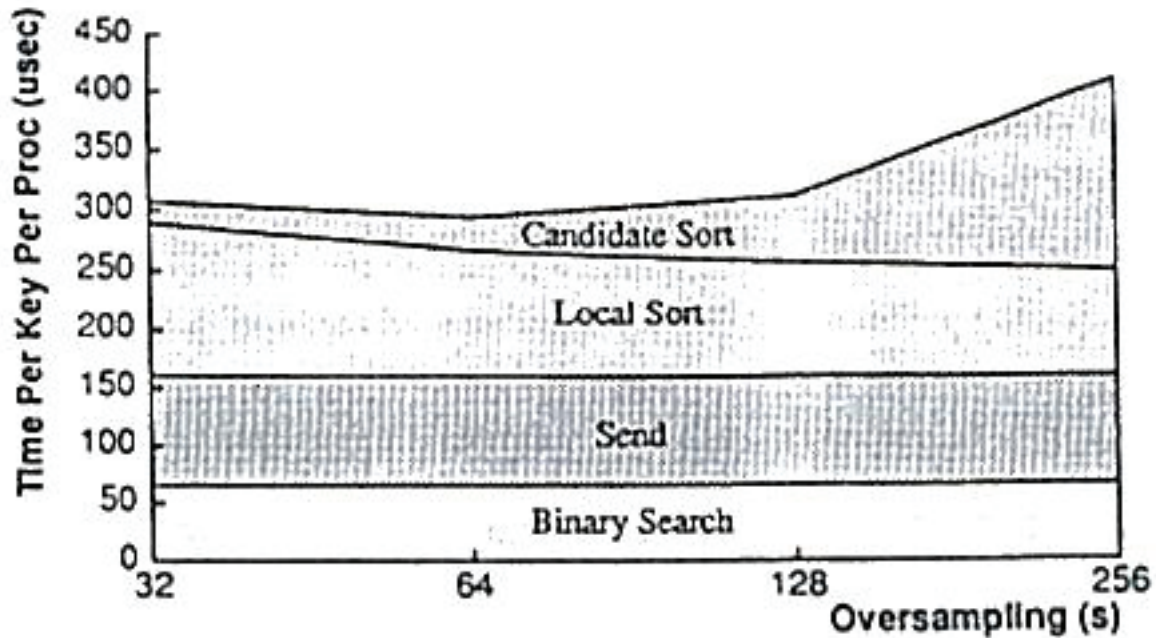


Figure 11: Sample sort time as a function of the oversampling ratio for 16384 keys per node of a 1024 node CM-2.

Method	Stable	Load balance	Time/key/node		Memory	Rank
			$\frac{P}{N} = 64$	$\frac{P}{N} = 16384$		
Bitonic	No	Yes	1600 μ sec	2200 μ sec	1.0	1.5
Radix	Yes	Yes	2400 μ sec	950 μ sec	2.1	1.0
Sample	No	No	5500 μ sec	330 μ sec	3.2	1.5

Table 4: Comparison of sorting times on a 1024 node CM-2.

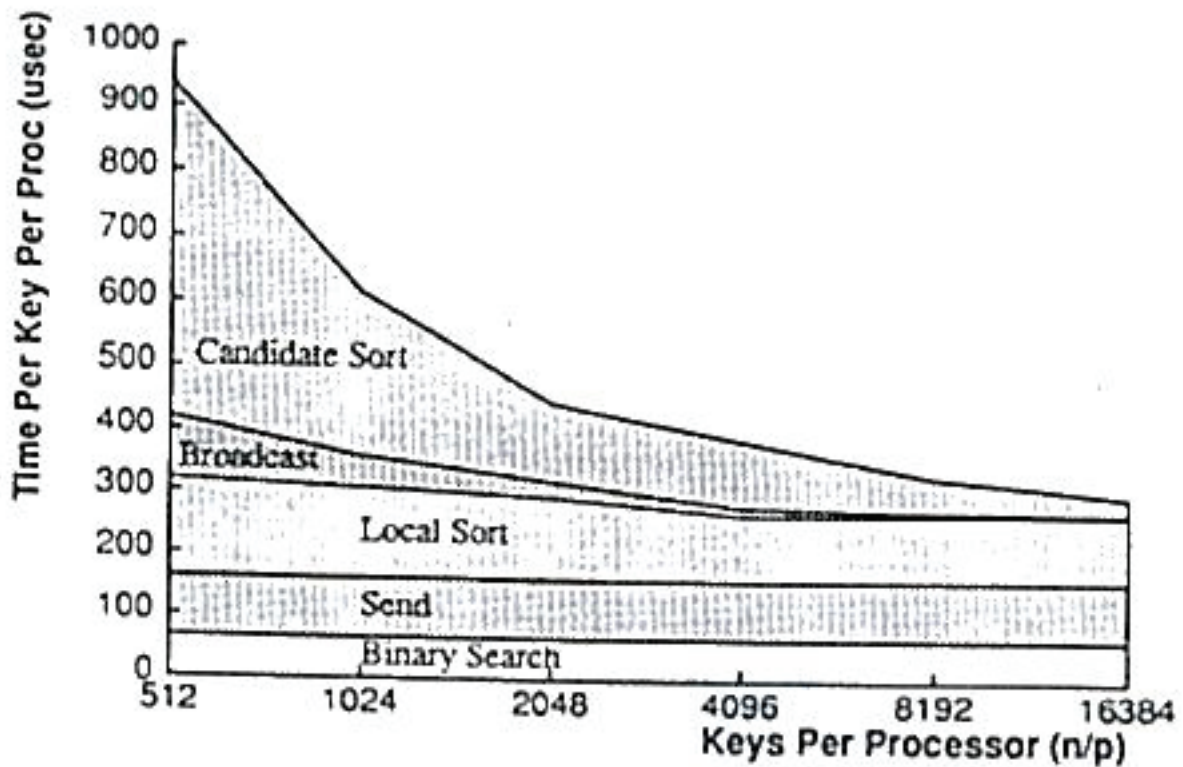


Figure 12: Sample sort time as a function of the number of keys per node for a 1024 node CM-2. Oversampling ratio 32 for up to 4096 keys per node. Oversampling ratio 64 for more than 4096 keys per node.

1.9 Merge sort

In this section we will describe a parallel sort based on merging sorted sequences. For a PRAM model of computation (Parallel Random Access Memory), the execution time for sorting P elements on P processors in time $2 \log_2 P \log_2 \log_2 P + O(\log_2 P)$ [10]. We first describe the merge algorithm on which the sorting is based.

1.9.1 An efficient PRAM merge algorithm

Let x be a sequence of P elements and y a sequence of R elements. The algorithm is based on recursively merging sequences of decreasing length. Each pair of sequences being merged is obtained by sampling the two sequences to be merged, recursively. The first sub-selection for the sequence x consists of the elements with indices $\{\lceil \sqrt{P} \rceil, 2\lceil \sqrt{P} \rceil, \dots, k\lceil \sqrt{P} \rceil\}$, where $k = \lfloor \sqrt{P} \rfloor$ whenever \sqrt{P} is not an integer, else $k = \sqrt{P} - 1$. Similarly, for y the sub-selection consists of the elements $\{\lceil \sqrt{R} \rceil, 2\lceil \sqrt{R} \rceil, \dots, l\lceil \sqrt{R} \rceil\}$, where $l = \lfloor \sqrt{R} \rfloor$, whenever \sqrt{R} is not an integer, else $l = \sqrt{R} - 1$.

1. Merge the subset of elements of x and y .
2. Insert the subset of elements of x into the proper segment of y
3. Repeat the process recursively for each pair of sublists created by the insertion of selected elements of x into y .

Assume that there are $N = \lfloor \sqrt{PR} \rfloor$ processors. Then, steps one and two each can be performed in unit time. The first step requires $kl - 1 \leq \lfloor \sqrt{PR} \rfloor = N$ comparisons. The second step requires at most $k(\lceil \sqrt{R} \rceil - 1) \leq \lfloor \sqrt{P} \rfloor (\lceil \sqrt{R} \rceil - 1) \leq N$ comparisons. If there is enough processors to merge all pairs of sublists created through the insertion, then the time for the merge will amount to two time steps per recursion step. We will now show that indeed N processors suffice for the next recursion step.

After the insertion there are a set of pairs of lists (X_j, Y_j) to be merged. Having inserted the selected elements of x into y , each X_j is a segment of x (between successive selected elements) and thus $|X_j| \leq \lceil \sqrt{P} \rceil$, where $|\cdot|$ denotes the cardinality of the set. Moreover, $\sum_j |X_j| < P$, excluding the subselected elements already merged with subselected elements of y . Similarly, $\sum_j |Y_j| < R$. For the recursion process to proceed we need to assign $\sum_j \lfloor \sqrt{|X_j||Y_j|} \rfloor$ processors to all the pairs of sublists. Thus, we need to show that $\sum_j \lfloor \sqrt{|X_j||Y_j|} \rfloor \leq N$. But, by Cauchy's inequality

$$\sum_j \sqrt{|X_j||Y_j|} \leq \sqrt{\sum_j |X_j| \sum_j |Y_j|} \leq \lfloor \sqrt{PR} \rfloor = N$$

Hence, $\sum_j \lfloor \sqrt{|X_j||Y_j|} \rfloor \leq N$, and we have shown that the recursion can proceed with each recursion step taking two units of time for $N = \lfloor \sqrt{PR} \rfloor$ processors.

The number of recursion steps is at most $\log_2 \log_2 P + 1$ for $P \leq R$. If L_i is the maximum length of the shortest sequence in a pair, then $L_i = L_0^{\frac{1}{2^i}}$, where $L_0 = \lceil \sqrt{P} \rceil$, since $P \leq R$. The recursion terminates when $L_i = 1$, i.e., the last merge/insertion step has a maximum shortest subsequence length of two. The number of merge/insertion steps are $\log_2 \log_2 P$. We now have shown

Theorem 3 *Two sorted sequences of length P and R respectively, $P \leq R$, can be merged on $N = \lfloor \sqrt{PR} \rfloor$ processors in time $2 \log_2 \log_2 P + \text{const}$.*

If there are more processors, say $N = \lfloor \sqrt{rPR} \rfloor$ processors, then the following corollary holds.

Corollary 1 *Two sorted sequences of length P and R respectively, $P \leq R$, can be merged on $N = \lfloor \sqrt{rPR} \rfloor$ processors, $r \geq 2$, in time $2(\log_2 \log_2 P - \log_2 \log_2 r) + \text{const}$.*

Proof: Mark elements $\lceil \sqrt{P/r} \rceil$ and $\lceil \sqrt{R/r} \rceil$ respectively from the two sequences to be merged. It is clear that the merge and insertion operations both require unit time on $\lfloor \sqrt{rPR} \rfloor$ processors. Moreover, the recursion is also still valid with respect to the number of processors required for successive recursion steps. Furthermore, $L_i = \sqrt{L_{i-1}/r}$, and the corollary follows. ■

Corollary 2 *Two sorted sequences of length P and R respectively, $P \leq R$, can be merged on $N \leq P$ processors in time $\frac{P+R}{N} + \log_2\left(\frac{PR \log_2 N}{N}\right) + \text{const}$.*

Proof: We create N pairs of subsequences by marking $N - 1$ elements in each sequence, separated by $\lceil \frac{P}{N} \rceil$ and $\lceil \frac{R}{N} \rceil$ elements respectively. Merging the selected elements on N processors requires at most two units of time, since $2(N - 1) - 1$ comparisons are required and there are N processors. The insertion of all marked elements into the two sequences requires $(N - 1) \log_2 \frac{P}{N} + (N - 1) \log_2 \frac{R}{N} = (N - 1) \log_2 \frac{PR}{N}$ comparisons, which can be done in time $\log_2 \frac{PR}{N}$ on N processors. After the insertion there are $2N$ pairs of lists, where no pair of lists have more than $\frac{P+R}{N}$ elements together. Moreover, by giving the merge of a pair of sublists precisely as many cycles as required, the merge can be completed in $\frac{P+R}{N}$ cycles. It may be required to use the fact that a pair of sublists can be merged by starting from both ends concurrently to realize this bound. ■

These merge algorithms by Valiant [10] clearly offers a linear speedup over the sequential algorithms for $N \ll P + R$, and close to perfect speedup and efficiency for $N \approx \sqrt{PR}$.

1.9.2 An efficient PRAM Merge-sort

Theorem 4 *Sorting a sequence of P elements can be made on $N = P/2$ processors in time $2 \log_2 P \log_2 \log_2 P + O(\log_2 P)$ time using the merge algorithm in theorem 3*

Proof: Using the merge algorithm, after stage i there are $P/2^i$ lists, each of length 2^i . The merge of a pair of such lists requires time $2 \log_2 \log_2 i + \text{const}$ on 2^i processors, i.e., a total of $\frac{P}{2^i} 2^i = \frac{P}{2}$ processors. Hence, the total time for the sort is $\sum_{i=1}^{\lceil \log_2 P \rceil} (2 \log_2 \log_2 i + \text{const}) \leq 2 \log_2 P \log_2 \log_2 P + O(\log_2 P)$. ■

For $1 < N < P$ processors we have the following corollary.

Corollary 3 *Sorting a sequence of P elements can be made on $1 < N < P$ processors in time $\frac{P \log_2 P}{N} + 2 \log_2 N \log_2 \left(\frac{P \log_2 N}{N} \right)$ time using the merge algorithm in corollary 2.*

Proof: First sort locally $\frac{P}{N}$ elements in each node in time $O\left(\frac{P}{N} \log_2 \frac{P}{N}\right)$. Then, by using the merge algorithm in Corollary 2, we first merge $N/2$ pair of sequences of length $\frac{P}{N}$ on N processors. Merging a pair of sequences on two processors result in a leading term of $\frac{P/N + P/N}{2} = \frac{P}{N}$ time steps. In the next step, pairs of sequences of length $\frac{2P}{N}$ are merged on four processors, since there is only half as many such sequences as sequences of length $\frac{P}{N}$. Hence, the dominating term for the $\log_2 N$ merge steps is $\frac{P}{N} \log_2 N$. We have now shown that the leading term resulting from the local sort and the leading term for the parallel merge is $\frac{P}{N} \log_2 P$. The lower order term follows from the lower order term of the parallel merge. ■

References

- [1] Kenneth E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, pages 307–314. IEEE, 1968.
- [2] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zaglia. A comparison of sorting algorithms for the connection machine cm-2. In *SPAA '91*, pages 3–16. ACM Press, 1991.
- [3] Daniel S. Hirschberg. Fast parallel sorting algorithms. *Communications of the ACM*, 21(8):657–661, 1978.
- [4] S. Lennart Johnsson. Combining parallel and sequential sorting on a Boolean n-cube. In *1984 International Conference on Parallel Processing*, pages 444–448. IEEE Computer Society, 1984.
- [5] Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, 1973.
- [6] M. Kumar and Daniel S. Hirschberg. An efficient implementation of batcher’s bitonic odd-even merge algorithm and its application in parallel sorting schemes. *IEEE Trans. Computers*, 32(3):254–264, 1983.
- [7] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.
- [8] David Nassimi and Sartaj Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Computers*, C-28(1):2 – 8, January 1979.
- [9] C.D. Thompson and H.T. Kung. Sorting on a mesh-connected parallel computer. *CACM*, 20(4):263–271, 1977.
- [10] Leslie Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348–355, September 1975.