B-Flashsort: A High-performance Sort for the MasPar MP-1 and MP-2

Implementation Description

Jan F. Prins
Department of Computer Science
University of North Carolina at Chapel Hill

October 1992

1. Introduction

The sorting of large data sets is a key step in a great variety of applications such as query processing, polygon rendering, statistical analysis and optimization problems. While efficient sorting techniques and implementations have long been available for sequential computers, the problem remains a challenge for parallel computers. One reason is that efficient sequential algorithms take into account the outcome of all previous comparisons in determining the next comparison to make, which is a fundamentally serializing notion at odds with parallel execution. Another reason is that sorting is highly communication intensive, and communication performance is often a limiting factor in parallel machines.

Extensive cooperative efforts between the research community and supercomputer manufacturers have recently resulted in new, very high-performance, parallel implementations of sorting. The highest sorting performance available at any price can now be found on large multiprocessor supercomputers such as the 16-processor Cray C90 and the 1024 processor CM-5. Recognition of the importance of sorting performance, and the challenges that it poses for parallel machines, has also led to its incorporation into supercomputer benchmarks such as NAS [TS92].

I have been interested in rapid sorting using the MasPar family of computers since 1990, and have developed a variety of sorting implementations [PS90]. Recently this work led to the development of B-Flashsort, a new algorithm for parallel sorting using processors arranged in a torus with an SIMD model of communication [HPR92]. An implementation of this algorithm for the MasPar MP-1 and MP-2 has been constructed and is described in this document.

The implementation performs the following: given n values initially distributed over a p processor MP-1 or MP-2 as n/p values at each processor, the values are rearranged into increasing order over the processors (in the standard processor enumeration order). Although each processor may hold fewer or more than n values on completion, with overwhelming probability, the maximum number of values in any processor will be less than $1.8n$. For a given size MasPar machine, the sort time scales *linearly* with dataset size. Using this implementation, it is estimated that a full-size MP-2 can attain over 1/3 of the sorting performance recently reported for a 1,024 processor CM5 [TS92].

2. Algorithm Description

The collection of parallel sorting algorithms in the literature is extensive, but two observations limit the class of algorithms suitable for a high-performance sort for MasPar machines considerably. First, we must be very careful about transferring algorithms from abstract parallel computational models such as PRAM since the overhead introduced in implementing the basic operations of such models may be as expensive as the sort operation itself. This restricts us primarily to vector-model algorithms and a variety of network-model algorithms including mesh algorithms in particular (if we are to use MasPar *xnet*communication). Second, we are concerned with sorting datasets of size $n >> p$ since p is always small ($p \leq 16{,}384$ for MasPar machines) compared to the size of problems that demand supercomputer efforts. This restriction requires us to "slow-down" element-per-processor algorithms by using fewer processors and to increase the efficiency of the sequential computations introduced by that process.

An example of such an approach is the adaptation of bitonic sort to the MP-1 [PS90]to yield the *vbsort* (virtualized bitonic sort) algorithm that (mostly) uses the xnet on the MP-1 and runs in-place for $n = 2^k p$ for $k \geq 1$. Another example is radix-sort [BLM+91], which was adapted from the vector model for execution on the CM-2.

For large values of n we can exploit an important observation: it is possible to predict the distribution of values in a large dataset from a small sample. Moreover the sample size required to predict the distribution of n values is a function of the accuracy required of the prediction and not a function of n. This observation led to the development of randomized parallel algorithms for sorting [HC83, RV87].

The *samplesort* algorithm of [HC83] as adapted for a p node (slicewise) CM-2 in [BLM+91] works as follows. A sample of the complete set of n input keys distributed uniformly across p processors is extracted and sorted using some efficient deterministic algorithm. From the sorted sample a subset consisting of $p-1$ values is extracted that partitions the input into p bins that are similarly-sized with high likelihood. These $p-1$ *splitters* are then broadcast to all processors, after which binary search can be used in parallel at all processors to locate the destination processor for each of their values. Subsequently all values are sent to their destination processor using general routing, and a local sort of the values received at each processor completes the algorithm.

Although samplesort is very fast for large n/p, it does have some important limitations. The requirement that the complete splitter set be available at every processor is difficult to meet on large MasPar machines. A full-size MasPar MP-1, for example, has 16,384 processors, but each processor has only 64KB of local storage. To samplesort a data set in which the values are 64-bit integers would require each processor to have $64p$ bits or 128KB available for the splitters alone. In addition, the broadcast of such a large set of values can be costly.

We have developed an alternate randomized parallel sorting algorithm that does not broadcast the splitters, but rather performs classification of the data in the course of routing: the destination of the data is determined with increasing precision as it is moved through the machine. This technique is related to *splitter-directed routing* as used in [RV87] for the Flashsort sorting algorithm for the hypercube. In that approach, the splitters are organized into a binary tree mapped into the nodes of a hypercube. A value is classified by routing it through the tree, as directed by comparison with the splitters. Since all values must move through the root node, the resultant bottleneck requires the use of a sophisticated pipelining strategy to achieve good communication efficiency, and is not compatible with an SIMD communication model. By constructing a different routing strategy the bottleneck can be eliminated, and batched-routing can be used instead of pipelined routing. Consequently our algorithm is called *B-Flashsort* (for Batched-routing Flashsort).

The batched-routing technique works as follows on the torus. Values are moved first vertically and then horizontally on the torus in successively smaller power-of-2 strides. Each processor partitions its values into a set of values to send on and a set of values to keep, based on two splitters it holds. The batches of values to be sent on are moved from all processors simultaneously via xnet. On a $p_1 \times p_2$ torus each value moves over at most p_1+p_2-2 *xnet* steps. This is optimal for the torus under the SIMD communication model; no other sorting algorithm using the MasPar *xnet* can perform fewer communication steps.

The high-level B-Flashsort algorithm for the $p = p_1 \times p_2$ torus is given in Figure 1. Each processor (i,j) starts with a list $L_{i,j}$ of n/p values to be sorted. The algorithm proceeds in three phases, SUBSAMPLE, BATCH-SDR, and LOCAL-SORT.

In the SUBSAMPLE phase, we choose k local samples without replacement at each processor (i,j) into list $G_{i,j}$. The value of k is a function of p and the maximum ratio of maximum to average queue length (at a given likelihood); see [HPR92]. Next a fast deterministic sort (in this case vbsort) is used to sort all kp selected values across p processors. Each processor $0 \le l < p$ (in the standard row-major enumeration order) defines its portion of the data set as all values v such that $max\ G_{l-1} < v \le max\ G_l$ where $max\ G_{-1} = -\infty$ and $max\ G_p = +\infty$. From this definition we derive the values for vertical routing phase splitters SV^- and SV^+ and horizontal routing phase splitters SH^- and SH^+.

In the BATCH-SDR phase, values are first moved vertically toward their destination row in $\log p_1$ steps, each using two splitters. Step 13 splits the local list $L_{i,j}$ into a new list $L_{i,j}$ of values to keep and a list $M_{i,j}$ of values to move h steps. The predicate $\tau_V(h,v,(i,j))$ is true exactly when v belongs on a row of processors less than h steps down from processor i. Because of the torus topology, the predicate has two cases:

$$\begin{aligned}\tau_V(h,v,(i,j)) = \\ &[(i+h \le p_1) \wedge (SV^-_{i,j} < v \le SV^+_{i+h-1,j})] \\ \vee\ &[(i+h > p_1) \wedge ((SV^-_{i,j} < v) \vee (v \le SV^+_{i+h-1,j}))]\end{aligned}$$

```
B-FLASHSORT–2D(L)

   SUBSAMPLE
1.     foreach (i,j)∈ (0,0) .. (p1–1,p2–1)
2.         G[i,j] := select k random elements from L[i,j]
3.     sort (G) using vbsort
4.     foreach (i,j)∈ (0,0) .. (p1–1,p2–1)
5.         SH+[i,j] := if (i,j) ≠ (p1–1,p2–1) then G[i,j][k-1] else +∞
6.         SH–[i,j] := if j ≠ 0 then G[i,j–1][k-1] else –∞
7.         SV+[i,j] := if i ≠ p1–1 then G[i,p2–1][k-1] else +∞
8.         SV–[i,j] := if i ≠ 0 then G[i–1,p2–1][k-1] else –∞

   BATCH-SDR
9.     h := p1
10.    while h > 1 do
11.        h := h /2
12.        foreach (i,j)∈ (0,0) .. (p1–1,p2–1)
13.            L[i,j], M[i,j] := [ v ∈ L[i,j] | τV(h,i,v)], [ v ∈ L[i,j] | ¬τV(h,i,v)]
14.            L[i,j] := L[i,j] ++ M[i–h,j]
       end

15.    h := p2
16.    while h > 1 do
17.        h := h /2
18.        foreach (i,j)∈ (0,0) .. (p1–1,p2–1)
20.            L[i,j], M[i,j] := [ v ∈ L[i,j] | τH(h,j,v)], [ v ∈ L[i,j] | ¬τH(h,j,v)]
21.            L[i,j] := L[i,j] ++ M[i,j-h]
       end

   LOCAL-SORT
22.    foreach (i,j)∈ (0,0) .. (p1–1,p2–1)
23.        radix-sort (L[i,j])
```

Figure 1. B-Flashsort Algorithm

The value of $SV^{+}_{i+h-1,j}$ needs to be obtained once for each iteration of the **while** loop. Step 14 corresponds to a batch routing of the values to be moved. All values move the same distance h in the same direction, so that synchronous nearest-neighbor communication can be employed. Following the vertical phase a similar horizontal phase locates the values at the correct processors within each row. The predicate $\tau_H(h,v,(i,j))$ is defined analogously to $\tau_V(h,v,(i,j))$ above.

Finally, in the LOCAL-SORT phase, all values have arrived at the correct processor and must be placed into the correct order locally. We use a radix sort since this gives performance linear in the length of the longest local list.

For a detailed analysis of the correctness, running time, and parameters to control routing congestion and splitter estimation error see [HPR92]. Here we summarize the time and space complexity of the algorithm.

2.1 Time complexity

For a fixed machine size, the SUBSAMPLE phase of B-FLASHSORT–2D has constant time and the remaining phases have time linear in n. Hence for a given machine, the sort time scales linearly with the problem size. In fact the sorting rate increases toward an asymptote with increasing data set size as the constant cost of the SUBSAMPLE phase is amortized over a larger number of elements.

For a fixed problem size, doubling the number of processors adds one more iteration in the BATCH-SDR phase, but halves n/p, hence the sorting rate can come close to doubling.

2.2 Space complexity

The expansion of the local lists through inaccuracies in the approximation of the splitter set requiresat most 1.8 times the size of the input data set with overwhelming probability. In order to apply radix-sort, we must actually have double this space, since radix sort can not be performed in-place. However, in the algorithm as implemented, provisions are made to use in-place local sorting algorithms (*heapsort* or *merge-exchange sort* depending on the value of n) in case insufficient space is available. However, these algorithms degrade performance compared to radix sort.

3. Results

We report on the sorting rate for 32-bit and 64-bit integers on various size MP-1 machines and a 4K MP-2 and give comparative performance of other sorting algorithms on MasPar and other machines.

3.1 Performance on MP-1 and MP-2

Tables 1 and 2 give the *bflash* sorting rate for 32-bit and 64-bit integers for various problem sizes on several different sizes of MP-1 and a 4K MP-2. The 8K MP-1 is the machine at UNC, and 4K MP-1 timings were made on this machine using the *mpswopt* facility. Access to the 16K MP-1 and the 4K MP-2 are courtesy of Ames Lab in Iowa. The 16K MP-1 is a small-memory machine, and hence the figures in parentheses are extrapolated rather than measured.

B-Flashsort Sorting Rate (millions of 32-bit integers per second)				
n/p	4K MP-1	8K MP-1	16K MP-1	4K MP-2
65	0.7	1.2	2.0	1.1
125	1.1	1.8	3.0	1.6
250	1.5	2.6	4.5	2.5
500	2.0	3.5	6.0	3.6
1000	2.4	4.1	7.4	4.5
2000	2.6	4.6	(8.3)	5.2
4000	2.7	4.9	(8.8)	5.6
5000	2.8	5.1	(9.0)	5.8

Table 1. B-Flashsort 32-bit integer sorting performance

B-Flashsort Sorting Rate (millions of 64-bit integers per second)				
n/p	4K MP-1	8K MP-1	16K MP-1	4K MP-2
65	0.4	0.7	1.1	0.6
125	0.6	1.0	1.7	1.0
250	0.8	1.4	2.4	1.4
500	1.1	1.9	3.3	2.0
1000	1.2	2.2	(3.7)	2.4
2000	1.3	2.4	(4.0)	2.8

Table 2. B-Flashsort 64-bit integer sorting performance

3.2 Comparison with other sorting algorithms on MP-1 and MP-2

The only large dataset sorting algorithms implemented for the MP-1 and MP-2 of which I'm aware are those that I have programmed at UNC. These are: *vbsort* (bitonic sort [PS90]), *radix* (parallel radix sort [BLM+91]), and *sample* (randomized samplesort from [BLM+91]).

Bitonic sort is very effective at low n/p which is why it is the sort of choice for the SUBSAMPLE phase of B-Flashsort. However, even with an efficient communication–cost embedding of the algorithm in the mesh, *vbsort* performs 4 to 7 times as many *xnet* shift operations as *bflash*., hence it is not competitive, particularly for large machines.

Radix sort uses a fixed number of scan operations independent of n, and hence this approach only becomes competitive with bitonic sort for large values of n. Since radix sort also uses four to eight general routing steps (depending on key size and radix size), it quickly becomes uncompetitive with samplesort which has a single routing step.

Finally samplesort comes close to the performance of B-Flashsort, but recall that samplesort can not really be implemented in the current MasPar architecture for large key sizes and/or large numbers of processors. Sorting 32-bit values on a 4K large-memory machine is about the maximum configuration in which samplesort can operate. It is interesting that samplesort comes closer in performance to B-Flashsort on an MP-1 than it does on an MP-2. This is due to the increased relative cost of the broadcast of the splitter set from the ACU to the PE's in samplesort on the MP-2.

Table 3 gives the maximum performance obtained with each of the sorting algorithms on a 4K MP-1 and MP-2 and the value of *n*/*p* at which it was reached.

	4K MP-1		4K MP-2	
algorithm	best perf.	n/p	best perf.	n/p
vbsort	0.9	8	1.2	16
radix sort	1.0	4096	n/a	n/a
samplesort	2.6	5000	4.0	5000
B-flashsort	2.8	5000	5.8	5000

Table 3. 32-bit integer performance of various sorting algorithms

3.3 Comparison with supercomputer sorting algorithms

The Cray Y-MP and the TMC CM-2 have been the subject of aggressive efforts to develop faster sorting implementations. In the case of TMC a full team of programmers provided microcoding and measurement support for the development of samplesort. In both cases the resultant algorithms were at least a factor of five faster than the previous system sorting algorithms on these machines.

Since the hardware on both of these machines directly support 64-bit arithmetic operations, the performance figures for sorting on these machines are most strongly stated as 64-bit value sorting rates.

Table 4 lists the 64-bit performance of B-Flashsort in comparison with published sorting rates of other machines. Using B-Flashsort, a full-size MP-1 can outperform a single head of the Y-MP. If we extrapolate the performance of a 16K MP-2, it seems clear that it will outperform a full-size CM-2 and will likely match the performance of the full-size CM-200 (42% faster than a CM-2).

I am not aware of published figures for 64-bit sorting performance of the TMC CM5 and the Intel Paragon. However, a 64-bit samplesort, when implemented using the CM5 floating-point processors, is likely to yield a 32-processor CM5 performance similar to B-Flashsort on a 4K MP-1. That level of performance would extrapolate to a sorting rate of about 90 million elements per second for a 1,024 processor CM-5.

Algorithm	Machine	config.	elt type	dataset size	sorting rate ($x10^6$ elt/sec)	reference
radix	Cray Y-MP	8 procs	64b int	$0.5x10^6$	16.7	[ZB91]
sample	CM-2	64K PE	64b int	$32x10^6$	6.3	[BLM91]
B-Flash	MP-1	16K PE	64b int	$32x10^6$	(4.0)	
radix	Cray Y-MP	1 proc	64b int	$0.5x10^6$	3.3	[ZB91]
B-Flash	MP-2	4K	64b int	$8.2x10^6$	2.8	
bitonic	iWarp	8 x 8	64b int	$1x10^6$	2.3	[S92]

Table 4. B-flashsort performance relative to published 64-bit sorting figures.

I have available two published 32-bit sorting rates that have appeared in the literature. A 1-billion 32-bit integer sort was performed on an Intel Delta at the rate of 16.7 million elements per second, but this algorithm can not sort anything other than a uniformly distributed input data set, hence is not included. Thearling and Smith [TS92] report a sorting rate of over 52 million elements per second on a 1,024 processor CM5 using a radix sort. A full-size MP-2 could likely achieve over 1/3 of this performance at a small fraction of the cost.

It is worth noting that the proposed NAS sorting benchmark calls for the sorting of 19-bit quantities, for which the 32-bit performance of the MP-1 and MP-2 give a good lower bound.

Algorithm	Machine	config.	elt type	dataset size	sorting rate ($x10^6$ elt/sec)	reference
radix	CM5	1K PE	32b int	$1000x10^6$	52.6	[TS92]
B-Flash	MP-1	16K PE	32b int	$82x10^6$	(9.0)	
B-Flash	MP-2	4K PE	32b int	$20x10^6$	5.8	

Table 5. B-flashsort performance relative to published 32-bit sorting figures.

4. References

[BLM+91] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha; A Comparison of Sorting Algorithms for the Connection Machine CM-2; *3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* ACM, 1991.

[HC83] J.S. Huang and Y.C. Chow, Parallel Sorting and Data Partitioning by Sampling, *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference,* pp. 627-631, November 1983.

[HPR92] W. Hightower, J.F. Prins, J. Reif, Implementations of Randomized Sorting on Large Paralllel Machines; *4th Annual ACM Symposium on Parallel Algorithms and Architectures,* ACM, 1992.

[PS90] J.F. Prins, J.A Smith, Efficient Bitonic Sorting of Large Arrays on the MasPar MP-1, *3rd Symposium on Frontiers of Massively Parallel Processing*, 1990; expanded version Technical Report 91-041, Univ. of North Carolina, 1991.

[RV87] J.H. Reif and L.G. Valiant, A Logarithmic Time Sort for Linear Size Networks, *15th Annual ACM Symposium on Theory of Computing,* Boston, MA, pp. 10-16, 1983; also in *Journal of the ACM,* 34(1):60-76, January 1987.

[S92] T. Stricker, Supporting the Hypercube programming model on mesh architectures (A fast sorter for iWarp tori) , *4th Annual ACM Symposium on Parallel Algorithms and Architectures,* ACM, 1992.

[TS92] K. Thearling, S. Smith, An Improved Supercomputing Sorting Benchmark; *Supercomputing 92,* ACM 1992.

[ZB91] M. Zagha, G. E. Blelloch, Radix Sort for Vector Multiprocessors, *Supercomputing 91,* ACM 1991.