Appeared in Proc. 4th ACM Symposium on Parallel Algorithms and Architectures, July 1992.

Implementations of Randomized Sorting on Large Parallel Machines

(Preliminary Version)

William L. Hightower
Elon College
Elon College NC 27244

Jan F. Prins†
University of North Carolina
Chapel Hill NC 27599-3175

John H. Reif‡
Duke University
Durham NC 27706

Abstract

Flashsort [RV83,86] and Samplesort [HC83] are related parallel sorting algorithms proposed in the literature. Both utilize a sophisticated randomized sampling technique to form a splitter set, but Samplesort distributes the splitter set to each processor while Flashsort uses splitter-directed routing.

In this paper we present B-Flashsort, a new batched-routing variant of Flashsort designed to sort $N>P$ values using P processors connected in a d-dimensional mesh and using constant space in addition to the input and output. The key advantage of the Flashsort approach over Samplesort is a decrease in memory requirements, by avoiding the broadcast of the splitter set to all processors. The practical advantage of B-Flashsort over Flashsort is that it replaces pipelined splitter-directed routing with a set of synchronous local communications and bounds recursion, while still being demonstrably efficient.

The performance of B-Flashsort and Samplesort is compared using a parameterized analytic model in the style of [BLM+91] to show that on a d-dimensional toroidal mesh B-Flashsort improves on Samplesort when $(N/P) < P/(c_1 \log P + c_2 d P^{1/d} + c_3)$, for machine-dependent parameters c_1, c_2, and c_3. Empirical confirmation of the analytical model is obtained through implementations on a MasPar MP-1 of Samplesort and two B-Flashsort variants.

†Email: prins@cs.unc.edu. Supported in part by ONR contract N00014-89-J-1873 and by DARPA/ISTO contract N00014-91-C-0114.

‡Email: reif@cs.duke.edu. Supported in part by DARPA/ISTO contracts N00014-88-K-0458, N00014-91-J-1985, and N00014-91-C-0114, and by NASA subcontract 550-63 of prime contract NAS5-30428, and by US-Israel Binational NSF Grant 88-00282/2.

1. Introduction

Considerable effort has been made by the theoretical community in the design of parallel algorithms with excellent and sometimes optimal asymptotic efficiency. However, the field has had less success in demonstrating the utility of these algorithms by actual implementations. Ideally, the analysis of parallel algorithms should proceed beyond asymptotic efficiency and be extended to the level of implementations. In practice, this is difficult to do.

For one thing, the relation between the asymptotic complexity of an algorithm and the performance of an implementation can be considerably more complex for implementations of parallel algorithms than for sequential algorithms. For example, the number of processors or the amount of memory available to a processor in existing machines can easily be reached well before the asymptotically superior performance of the algorithm is achieved. Hence several different algorithms and algorithm variants must be considered to determine the ones that give the best performance over combinations of problem and machine sizes and machine architecture. To be able to compare approaches we need analytic models of the performance of algorithms, in terms of measurable quantities; this can sometimes require the modification of algorithms to make them analyzable, without reduction in efficiency if possible. We must also validate the models with actual implementations on parallel machines.

A careful, quantitative, effort of this kind was undertaken by Blelloch et al. [BLM+91] in the comparison of sorting algorithms for the TMC CM-2. In that paper, explicit analytical models for a variety of sorting algorithms were developed in terms of fundamental underlying operations. The analytical models were coupled with measured times for the underlying operations on a CM-2 to accurately predict performance as a function of the number of processors P and the input problem size N. Particular consideration was given to the scaling behavior of the algorithms as N becomes much larger than P, since this reflects the problem sizes for which large parallel machines typically become useful. Samplesort was shown to be faster than Bitonic and Parallel Radix sort for large values of N/P, although its general utility was limited because of large memory requirements and poor performance at lower values of N/P.

Samplesort as described in [HC83], [BLM+91] is a randomized algorithm that works as follows. A sample of the complete set of input keys distributed uniformly across 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. After sending all values to their destinations using general routing, a local sort of the values within 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 set of splitters be available at every processor can be problematic for parallel machines in which the number of processors is large but the amount of local memory is modest. This class includes the recently completed Mosaic machine as well as commercial machines like the MasPar MP-1 and the DAP. A full-size MasPar MP-1, for example, has 16384 processors, but each processor has only 64KB of local storage. To Samplesort a data set consisting of 64 bit values would require each processor to have $64P$ bits or 128KB available for the splitters alone.

In this paper we propose an alternate randomized sorting algorithm related to the algorithm described in [RV83,87] that has become known as Flashsort (see also Ullman[U83], Appendix B for a simplified description of the algorithm). In particular, in [RV83,87] two techniques were used that were not incorporated in the Samplesort of [HC83], [BLM+91]:

(1) splitter-directed routing
(2) recursive application of the sampling, partitioning and routing step

We investigate both of these techniques and give a quantitative analysis of the circumstances under which the first is advantageous. Our algorithm, which we call Batched-routing Flashsort, or B-Flashsort, can be implemented on a wide variety of architectures, including any mesh, CCC, or hypercube network. In this paper we concentrate on its implementation on a low-dimensional toroidal mesh, which admits a simple and efficient approach to splitter-directed routing. The algorithm takes advantage of the toroidal interconnection topology to reduce communication cost by a factor of 2 over the non-toroidal mesh; this is useful in practice since the toroidal topology is actually implemented on many current and proposed machines.

The batched routing technique works as follows. For each dimension j of a d-dimensional mesh, values are moved along dimension j in successively smaller power-of-2 strides. Each processor partitions its values into a set of values to keep and a set of values to send on, based on two splitters it holds. All values to be sent from a processor can be moved contention-free since all movement is the same distance in the same direction. Although we expect that the general routing used in Samplesort, when implemented in hardware, should outperform batched-routing in B-Flashsort, the magnitude of this difference determines the minimum size problem on which

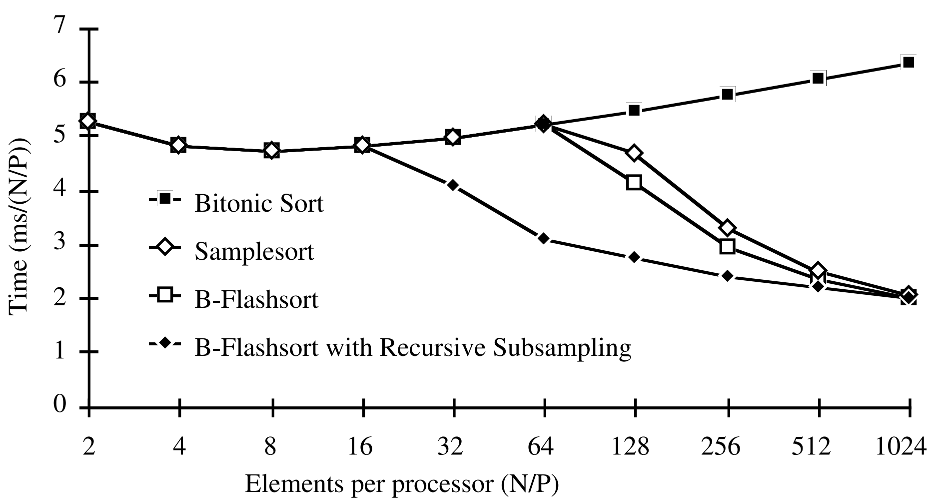

Figure 1. Performance of implemented algorithms on 4096 processor MP-1

Samplesort can be competitive. For machines with large numbers of processors, limited per-processor memory, or software routing, B-Flashsort may always be faster. This may be particularly true if batched routing is implemented at a very low-level.

The virtue of B-Flashsort is that it is simple, has per-processor storage requirements that do not scale with P, and does not require general routing. It is completely analyzable, up to tight bounds on the multiplicative constants. In comparison, in the original Flashsort in [RV83,87] it was difficult to exactly determine analytical constant factors due to the use of pipelining in the splitter-directed routing (with a goal of asymptotic bounds, only upper bounds were determined in [RV83,87]).

With the addition of a recursive sampling strategy we can extend the performance advantage of B-Flashsort over deterministic sorts to much smaller values of N/P. This may improve the utility of B-Flashsort as a general purpose sorting routine since it need not suffer poor behavior at low N/P.

A great deal of previous work has been concerned with sorting on the mesh with P processors in time $O(P^{1/2})$. The currently best bounds (smallest constant factor) known to us are given in [KKNT91]. Their algorithm is derived from Flashsort (with pipelined splitter directed routing) and in addition uses many sophisticated and ingenious techniques, but their algorithm is far too complex to yield an efficient practical implementation. In particular, the time bound has low order terms with large multiplicative constants which actually dominate the performance on existing large parallel machines. This indicates that there is a large gap between a theoretical result on parallel sorting as compared to a parallel algorithm validated by an efficient implementation on an actual parallel machine.

We develop an analytical model for the running time of B-Flashsort, following the approach taken in [BLM+91] and compare it with the model for Samplesort. We derive an expression in terms of machine-dependent parameters characterizing the range of N/P for which B-Flashsort outperforms Samplesort. To validate the analytic model, we implemented four sorting algorithms on a 4096 processor MasPar MP-1. Figure 1 summarizes the performance of the four sorting algorithms: (1) Bitonic sort, the fastest deterministic sort for small N/P on the MP-1 [Prin90], (2) Samplesort, (3) B-Flashsort and (4) B-Flashsort utilizing the recursive sampling strategy.

2. 1-D Algorithm

We will first explain the algorithm in the 1-D case and then extend the description to multi-dimensional meshes. In the 1-D case, P processors are connected in a ring and each processor $0 \le i < P$ starts with a list L_i of N/P values to be sorted. We assume here the values are distinct; at the end of this section we show that this requirement can be relaxed.

```
B-FLASHSORT-1D(L)
  SUBSAMPLE
1.      foreach i ∈ 0 .. P-1
2.            Gi := select k random elements from Li
3.      sort (G) using a deterministic P-processor sort
4.      foreach i ∈ 0 .. P-1
5.            if i = 0 then Si⁻ := −∞
                     else Si⁻ := Gi−1[k-1]
6.            if i = (P-1) then Si⁺ := +∞
                     else Si⁺ := Gi[k-1]
  BATCH-SDR
7.      h := P
8.      while h > 1 do
9.            h := h /2
10.           foreach i ∈ 0 .. P-1
11.                 Li, Mi := [ v ∈ Li | τ(h,i,v)],
                              [ v ∈ Li | ¬τ(h,i,v)]
12.                 Li := Li ++ Mi−h
13.     end
  LOCAL-SORT
14.     foreach i ∈ 0 .. P-1
15.           sort (Li)
```

The algorithm proceeds in three phases, SUBSAMPLE, BATCH-SDR, and LOCAL-SORT. In the first phase, we choose k local samples without replacement at each processor into list G. Next a deterministic sort is used to sort all kP values across P processors. Each processor i defines its portion of the data set as being all values v such that $S_i^- < v \le S_i^+$. Since all values are distinct, every value belongs to exactly one partition. S_i^+ is the largest value of the sorted sample at processor i, and S_i^- is the largest value of the sorted sample in processor i-1. Note that $S_0^- = -\infty$ and $S_{P-1}^+ = +\infty$.

In the BATCH-SDR phase, values are moved toward their destination in log P steps, each using two splitters. Step 11 splits the local list L_i into a new list L_i of values to keep and a list M_i of values to move h steps. The predicate $\tau(h,i,v)$ is true exactly when v belongs on a processor less than h steps away from processor i. Because of the ring topology, the predicate has two cases:

$$\begin{aligned}\tau(h,i,v) = \quad & [(i+h \le P) \wedge (S_i^- < v \le S_{i+h-1}^+)] \\ \vee \quad & [(i+h > P) \wedge ((S_i^- < v) \vee (v \le S_{i+h-1}^+))]\end{aligned} \qquad (2.1)$$

The value of S_{i+h-1}^+ used in $\tau(h,i,v)$ needs to be obtained once for each iteration of the **while** loop. Step 12 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.

LEMMA 1. The predicate

$$\forall(\ i, v:\ 0 \le i < P \text{ and } v \in L_i : \tau(h,i,v)\ ) \qquad (2.2)$$

is an invariant of the iteration at line 8.

PROOF. The assignment $h := P$ establishes (2.2) trivially. In each iteration, as h is halved, L_i is reduced on line 11 to just those values for which (2.2) holds. In line 14, (2.2) holds for all values of M_{i-h}. Thus (2.2) remains invariant.

THEOREM 1. B-FLASHSORT-1D sorts its input values.

PROOF. When the loop terminates after line 13, we have (2.2) and $h = 1$. Taken together this yields $(S_i^- < v \le S_i^+)$ for each value v in each processor i. Thus each processor holds only values destined for it. Since no values are created or destroyed, every input value has been correctly routed. Consequently L is partitioned appropriately and the sort in step 15 completes the proof.

The evaluation of $\tau(h,i,v)$ may be simplified by treating the boolean values as integers (true = 1, false = 0):

$$(S_i^- < v) + (\ v \le S_{i+h-1}^+) \ = \ 1 + ((i+h) \le P)$$

The term on the right hand side is constant within each processor on each iteration, so that two comparisons and two arithmetic operations suffice. If the comparison is carried out using unsigned comparisons, the predicate can be evaluated with a single comparison and a single arithmetic operation.

For simplicity, in the presentation of the algorithm we assumed the values being sorted were distinct. If values may occur several times in the splitter, we must modify predicate $\tau(h,i,v)$ to retain its interpretation in BATCH-SDR:

$$\begin{aligned}\tau(h,i,v) = \\ & [(i+h \le P) \wedge (S_i^- \le v \le S_{i+h-1}^+)] \qquad (2.1)\\ \vee\ & [(i+h > P) \wedge (S_i^- \ne S_{i+h-1}^+) \wedge ((S_i^- < v) \vee (v \le S_{i+h-1}^+))]\\ \vee\ & [(i+h > P) \wedge (S_i^- = S_{i+h-1}^+) \wedge ((S_i^- \le v) \vee (v < S_{i+h-1}^+))]\end{aligned}$$

Now the particular destination of an element that appears multiple times in the splitter is determined by its starting location. Therefore a dataset of non-distinct elements can be sorted using B-Flashsort if the dataset is randomly distributed.

3. Multidimensional Algorithm

The multidimensional version of B-FLASHSORT-MD operating on $P = p^d$ processors arranged in a d-dimensional torus can now be understood as follows.

The SUBSAMPLE step is identical to the 1-D version except that the deterministic sort must operate on a d-dimensional mesh. This step creates a splitter S_I at each processor indexed by a d-vector I, with $S_{(p-1),\ldots,(p-1)} = +\infty$.

The 1D BATCH-SDR is applied in parallel to each column (hyperplane) for each dimension $1 \le j \le d$. The splitters used in the application to dimension j are as follows:

$$S_I^+ = S_{I[1],\ldots,I[j],(p-1),\ldots,(p-1)}$$

$$S_I^- = \begin{cases} S_{I[1],\ldots,I[j-1],I[j]-1,p-1,\ldots,p-1} & \text{if } I[j] > 0 \\ -\infty & \text{otherwise} \end{cases}$$

The splitters for all of the steps can be created using $d-1$ spread operations from the highest indexed-elements in each dimension (in reverse order) starting with the original splitter S_I. BATCH-SDR step j leaves the values totally ordered within dimension j, hence when all steps complete, the input data set is completely partitioned.

For example, on a 2-D mesh, all columns are routed using the splitters in the last column of processors. Then each row is routed using the splitters within each processor, leaving the result partitioned in row-major order.

The LOCAL-SORT step is identical to the 1-D version and establishes the complete ordering.

4. Analysis

Define the *skew* W of the list L over all processors i to be

$$W = \frac{\max_i |L_i|}{\operatorname{avg}_i |L_i|} = \frac{\max_i |L_i|}{N/P}$$

The length of a list L_i can deviate from the average during execution of the algorithm for two reasons: routing-induced skew W_R and splitter-induced skew W_S. The former occurs because of random congestion in the routing phase, while the latter occurs when the choice of the splitters is imperfect so that the list on some processor ends up longer than the average.

THEOREM 2. For any $\alpha > 0$ and $N/P > (\alpha+1) \ln P$, the routing-induced skew W_R satisfies

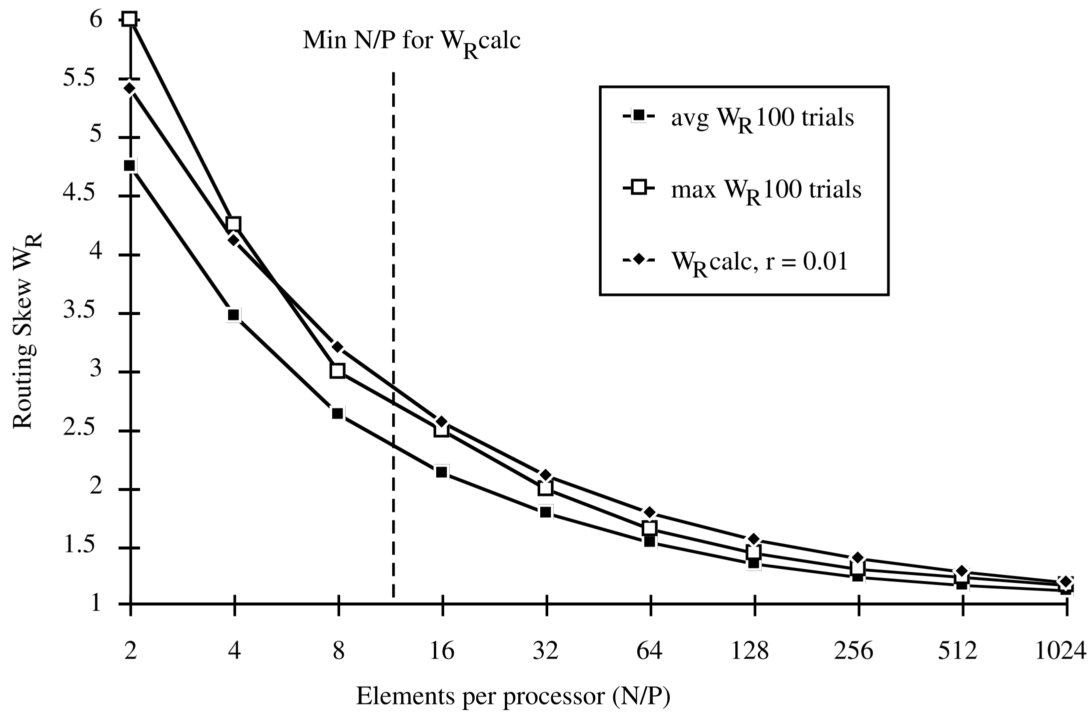

Figure 2. Predicted and observed routing-induced skew W_R for P = 4096.

$$W_R \leq 1+ \sqrt{3(P/N)(\alpha+1) \ln P}$$

with probability $1 - (1/P^{\alpha})$.

PROOF. (Appendix)

The routing-induced skew goes up slowly with increasing P, but decreases much more rapidly with increasing N/P. Given $P > 1$ and arbitrary r we may choose α to obtain a bound on W_R with probability $1-r$ provided N/P is sufficiently large. Figure 2 illustrates predicted and measured routing skew over 100 trials on a 4096 processor 2-D mesh. Figure 3 illustrates the dependence of routing skew on P and r.

Bounds for the expected relation between the per-processor sample size and the maximum size of the sorted lists at completion are given in [BLM+91] and [DNS91]. Given $P>1$ and arbitrary r, the sample size k required to limit splitter-directed skew to W_S with probability $1-r$ is given by:

$$k = \frac{2 \ln (P/r)}{(1\text{-}1/W_S)^2 W_S} \tag{4.1}$$

This bound is conservative since it is independent of the number of elements per processor.

THEOREM 3. Let k be the number of samples chosen per processor. Then for any $\alpha > 0$, the splitter-induced skew W_S satisfies

$$W_S + \frac{1}{W_S} \leq 2 + \frac{2(\alpha+1)}{k} \bullet \ln P$$

with probability $1-(1/P^{\alpha})$.

PROOF. Solve (4.1) for W_S with $r = 1/P^{\alpha}$.

Figure 4 illustrates the dependence of splitter skew on P and the oversampling ratio k. Since the SUBSAMPLE step performs a sort of k elements per processor, we are interested in keeping k small. On the other hand, a large W_S that results from a small k causes slowdown in the BATCH-SDR and LOCAL-SORT stages. Space considerations also encourage us to bound $W_S < 2$ with high probability at large N/P.

A simple compromise is to let $k = 4(\alpha+1) \ln P$, with $\alpha > 0$, chosen to keep the probability of exceeding the skew bounds acceptably low. For this choice, $W_S \leq 2$ for all N and P. Since $N/P \geq k$ (else we degenerate to the deterministic sort), we are guaranteed that $W_R \leq 1.85$ with the same probability. Routing-induced skew disappears in the last stages of SDR while routing-induced skew is introduced in these stages. With the choice of k above we can simplify the analysis by eliminating W_R and bounding skew in all stages of SDR by W_S.

If the skew bounds are exceeded the simplest recourse is to randomize the input and rerun the algorithm. Randomization is required because resampling may not solve a routing congestion problem.

Performance Analysis

We analyze the algorithm in terms of the primitive operations listed below. Each operation is assumed to have a relatively constant per-element time. The time for an arithmetic operation subsumes an indirect reference of its source and destination, as well as the time to update a counter on each processor.

opn	description
A	average time for an indirect memory-to-memory arithmetic operation
X	distance-one nearest-neighbor send time
Z	time to spread values along a dimension
R	average random destination send time

The total running time of B-FLASHSORT-MD is given by the sum of the times of its three phases and is a function of N and P.

$$
\begin{aligned}
T_{\text{B-FLASH-MD}}(N,P) = \; & T_{\text{SUBSAMPLE}}(N,P) \\
& + Z \bullet (d-1) \\
& + W_S \bullet \frac{N}{P} \bullet T_{\text{BATCHSDR}}(P) \\
& + T_{\text{LOCSORT}}(W_S \bullet N/P)
\end{aligned}
$$

The second term is the cost for the d–1 spread operations used to create the splitters for the SDR in each dimension. Of particular interest here is the T_{BATCHSDR} portion of this expression. On each step it performs work proportional to the longest list ($W_S \bullet N/P$). By examination of the B-FLASHSORT-1D algorithm we can express the time per element as follows:

$$
T_{\text{BATCHSDR}}(P) = 4A \log P + \frac{1}{2} \bullet [X \bullet d \bullet (P^{1/d} - 1)]
$$

On each SDR step we must compare a value with the bounding splitters and place it on one of two queues L or M (cost $3A$). We must also move queue M which involves an indirect access of the source and destination queue (cost $2A$), but we only charge 1/2 the per-element cost because on average only half the elements are placed in M. The total distance traveled by elements in queue M is $(P^{1/d} - 1)$ in each of d dimensions. The cost X per step of this distance is scaled by 1/2 because, again, only half the elements move on a given step.

5. Recursive Subsampling

Since each dimension in SDR is treated in turn we can consider an alternative approach to subsampling. At each SDR step we can sample a sufficient portion of the input to

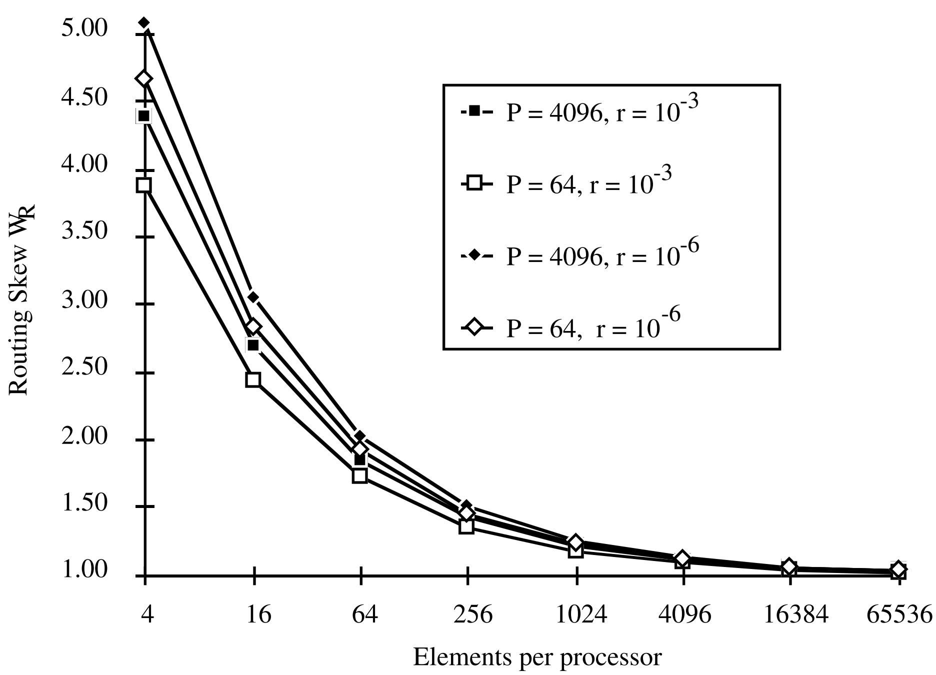

Figure 3. Dependence of W_R on P.

generate splitters for that step only.

For example, in a 2-D mesh we need $\sqrt{P}$ splitters in the first step. Since we can sort using P processors, we can divide the oversampling k obtained from theorem 3 by $\sqrt{P}$ to determine the oversampling ratio using P processors. This permits us to apply the algorithm to much smaller values of N/P without degenerating to the deterministic sort time.

In the second stage we need $\sqrt{P}$ splitters in every row, or one splitter per processor. Note from Figure 4 that the required oversampling ratio to generate $\sqrt{P}$ splitters for $\sqrt{P}$ processors for a skew W_S with probability r is lower than that required to generate P splitters for P processors with the same W_S and r. Furthermore, the sorting of the samples taken in each row using $\sqrt{P}$ processors will be faster than a sort of a sample with the same size per processor using P processors.

Thus recursive subsampling improves the performance of B-Flashsort for small values of N/P relative to P. As a result the B-Flashsort algorithm using recursive subsampling in figure 1 gives better performance at smaller input sizes, but does not alter the performance for large N/P.

The optimum choice for the number of recursive stages and the sampling size at each stage is highly dependent on the dimensionality of the mesh and the performance of the deterministic sorting algorithm at small N/P. The final paper will give details of the analysis.

6. Analytical and Empirical Comparison of Algorithms

Here we compare the performance of Samplesort and Multidimensional non-recursive B-Flashsort. In the non-recursive case the first and last steps of Samplesort and B-Flashsort, subsampling and local-sort respectively, are identical. That is,

$$T_{SAMPLESORT}(N,P) = T_{SUBSAMPLE}(N,P) + T_{SPLIT}(N,P) + T_{LOCSORT}(W_S \bullet N/P)$$

$$T_{B\text{-}FLASH\text{-}MD}(N,P) = T_{SUBSAMPLE}(N,P) + Z \bullet (d-1) + W_S \bullet \frac{N}{P} \bullet T_{BATCHSDR}(P) + T_{LOCSORT}(W_S \bullet N/P)$$

For Samplesort we have

$$T_{SPLIT}(N,P) = 2A \bullet P + 2A \bullet N/P \log P + R \bullet N/P$$

because each of P splitters must be broadcast with cost approximately $2A$, each of N/P elements must be found by binary search in the splitter table ($2A$ per probe), and each element must be sent to the destination queue. For B-Flashsort we have from section 4:

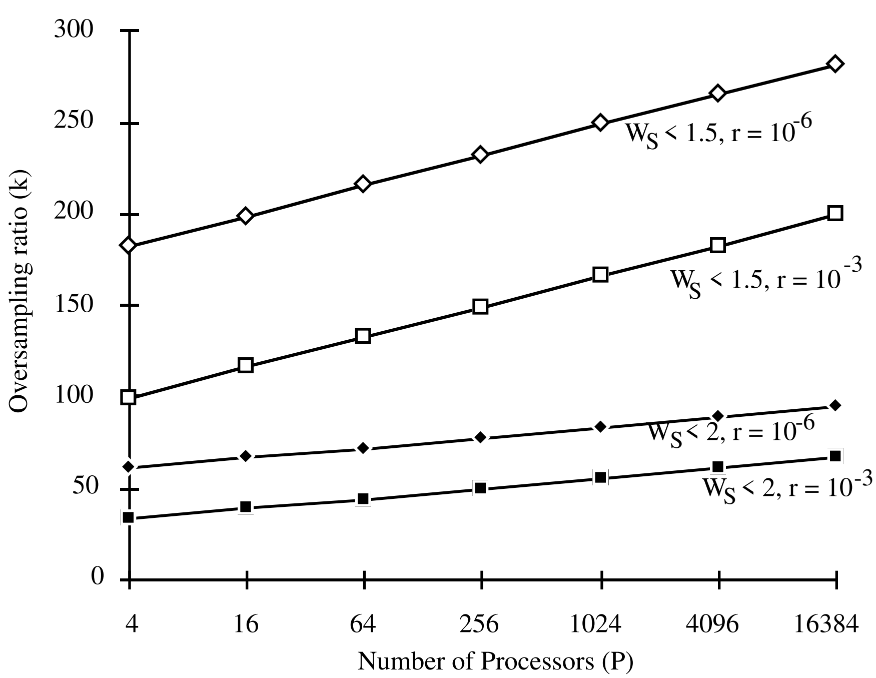

Figure 4. Dependence of oversampling on number of processors.

$$T_{\mathrm{BATCHSDR}}(P) = 4A \bullet \log P + \frac{1}{2} \bullet [X \bullet d \bullet (P^{1/d} - 1)]$$

To assess the predictive utility of the model (at least in one setting) we performed timings of the implementations of SPLIT and BATCHSDR on a 4096 processor MasPar MP-1 and compared the timings with those predicted by the model when times are provided for the various primitive operations of the model. The timings we used for the primitive operations of the model are shown below for 32 bit values on the MP-1. For the MP-1, $d = 2$ and the maximum value of P is 16384.

opn	MP-1 (µS)	description
A	20	average time for an indirect memory-to-memory arithmetic operation
X	3	distance-one nearest-neighbor send time
Z	8	time to spread values along a dimension
R	700	average random destination send time

The measured $T_{\mathrm{BATCHSDR}}(P)$ on the MP-1 was found to 75 log P + 3 (R(P) -1) µS which agrees very well with the predicted time. The measured $T_{\mathrm{SPLIT}}(N,P)$ was found to be $29P + 36\, N/P \log P + 740\, N/P$ which agrees reasonably well with the predicted time, particularly for large N/P.

To find the regime in which B-Flashsort gives superior performance, we examined the inequality

$$Z \bullet (d-1) + W_S \bullet (N/P) \bullet T_{\mathrm{BATCHSDR}}(P) < T_{\mathrm{SPLIT}}(N,P)$$

Combining N/P terms this gives

$$N/P \bullet [(W_S \bullet 4A) - 2A) \bullet \log P + W_S \bullet X \bullet (d/2) \bullet (P^{1/d} - 1) - R] < 2A \bullet P - Z \bullet (d-1)$$

which, taking the average maximum skew $W_S \approx 1.5$, and solving for N/P in terms of P gives

$$\frac{N}{P} < \frac{2A \bullet P - Z \bullet (d-1)}{4A \log P + 0.75X \bullet (P^{1/d} - 1) - R}$$

When we examine this relation for $P = 4096$ and $d = 2$, we predict that for $N/P \leq 300$ the B-Flashsort implementation should be faster than Samplesort, which is indeed borne out experimentally as is demonstrated in Figure 1.

Using the constants for the CM-2 reported with the analytic model of [BLM+91] we obtain values of N/P in the range 320 to 560 as crossover points below which B-Flashsort would be competitive with Samplesort on a full-size CM-2 ($P = 2048$, $d = 11$). The exact value depends on whether the formula for T_{SPLIT} is the one used here or the one defined in [BLM+91]. However, we have not verified this prediction experimentally.

7. Conclusions

Randomization has been demonstrably useful both in simplifying and in improving the efficiency of sorting algorithms on actual parallel machines. The B-Flashsort algorithm developed in this paper combines some of these concepts with an efficient implementation of splitter-directed routing that achieves the SIMD-model communication distance bound of $d \bullet (P^{1/d} - 1)$ for sorting on the d-dimensional torus.

The algorithm presented scales well to large machines, since its memory requirements are independent of the machine size. The analytic model developed suggests that the algorithm will perform well on machines with high-speed local connections on the mesh. Experimentally, B-Flashsort is competitive with Samplesort on the MP-1, a machine with good hardware routing capabilities and fast broadcast (the characteristics that favor Samplesort). A full-size MasPar MP-1 (16,384 processors) sorts approximately 9 million 32-bit integers per second using our implementation.

Appendix

THEOREM 2. For any $\alpha > 0$ and $N/P > (\alpha+1) \ln P$, the routing-induced skew W_R satisfies

$$W_R \leq 1 + \sqrt{3(P/N)(\alpha+1) \ln P}$$

with probability $1-(1/P^{\alpha})$.

PROOF. We use some basic probability theory in the proof of theorem 2. Consider n independently distributed Poisson variables $x_1,\ldots,x_n$, with $\mathrm{Prob}(x_i = 1) = \rho$ and $\mathrm{Prob}(x_i = 0) = 1-\rho$. The random variable

$$X = \sum_{i=1}^{n} x_i$$

has *binomial distribution* described by

$$\mathrm{Prob}(X = k) = \binom{n}{k} \rho^k (1-\rho)^{n-k}$$

The distribution has size n and mean $\mu = n \bullet \rho$. We will be interested in the probability that X has a value larger than some given value $k > \mu$:

$$\mathrm{Prob}(X \geq k) = \sum_{i=k}^{n} \binom{n}{k} \rho^k (1-\rho)^{n-k}$$

which satisfies the following inequality for any $0 \leq \varepsilon \leq 1.8\mu$ [KKNT91] :

$$\text{Prob}(X \geq \mu+\varepsilon) \leq e^{-\varepsilon^2/3\mu} \quad \text{(A.1)}$$

We can now proceed with the theorem. Recall we are sorting N values using P processors. Initially each processor holds N/P values. Consider the values held by an arbitrary processor H at the completion of SDR stage $1 \leq i \leq \log P$. There are a total of $2^i N/P$ values that may reach processor H, each with independent probability 2^{-i}. We model the length of the queue at H at stage i as a binomially distributed variable X with size $2^i N/P$ and mean $\mu = N/P$. Let q = Prob $(X \geq \mu+\varepsilon)$ with $0 \leq \varepsilon \leq 1.8\mu$. Then by (A.1) the probability that $X < \mu+\varepsilon$ at H is

$$1-q \geq 1-e^{-\varepsilon^2/3\mu} \quad \text{(A.2)}$$

Since there are P processors, all of which must satisfy (A.2), the probability r that *all* queues are shorter than $\mu+\varepsilon$ is

$$r = (1-q)^P \leq (1 - e^{-\varepsilon^2/3\mu})^P \quad \text{(A.3)}$$

Since $W_R = (\mu+\varepsilon)/\mu$, we have $\varepsilon = \mu(W_R-1)$. If we rewrite (A.3) in terms of W_R we get

$$\frac{-3}{\mu} \ln (1-r^{1/P}) \geq (W_R - 1)^2 \quad \text{(A.4)}$$

To bound the values of r for which (A.4) holds, note that we must have $0 \leq \varepsilon/\mu \leq 1.8$ to apply (A.1). Since $\varepsilon/\mu = (W_R-1)$ we must have

$$0 \leq \frac{-3}{\mu} \ln (1-r^{1/P}) \leq (W_R - 1)^2 = (\varepsilon/\mu)^2 \leq 3 < 1.8^2$$

which constrains r to satisfy $0 \leq r \leq (1 - e^{-\mu})^P$. By assumption $\mu \geq (\alpha+1) \ln P$, hence

$$\begin{aligned}
&(1-e^{-\mu})^P \\
&\geq (1-e^{-(\alpha+1)\ln P})^P \\
&= \left(1-\frac{1}{P^{\alpha+1}}\right)^{(P^{\alpha+1}/P^{\alpha})} \\
&\approx e^{-1/P^{\alpha}} && \text{(since for } x \to \infty,\ (1-1/x)^x \to e^{-1}) \\
&\approx 1 - \frac{1}{P^{\alpha}} && \text{(since for x} \to \infty,\ e^{-1/x} \to 1-(1/x))
\end{aligned}$$

Therefore the conditions obtain to apply the theorem for all $0 \leq r \leq 1 - (1/P^{\alpha})$. In particular, with $r = 1 - (1/P^{\alpha})$, it may be easily verified using the identities above that

$$W_R \leq 1+ \sqrt{3(P/N)(\alpha+1) \ln P}$$

(End of proof.)

Bibliography

[AH88] A. Aggarwal and M.-D. A. Huang, Network Complexity of Sorting and Graph Problems and Simulating CRCW PRAMs by Interconnection Networks; *Lecture Notes in Computer Science VLSI Algorithms and Architectures (AWOC 88)* (ed. by John Reif), vol. 319, pp. 339-350, Springer-Verlag, 1988.

[AKS83] M.Ajtai, J. Komlos, and E. Szemeredil, Sorting in c log n Parallel Steps, *Combinatorica*, 3:1-19, 1983.

[AV79] D. Angluin and L.G. Valiant, Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings, *Journal of Computer and System Sciences*, 18(2):155-193, April 1979.

[AKl85] S.G. Akl;, *Parallel Sorting Algorithms*, Academic Press, Toronto, 1985.

[BAT68] K. Batcher; Sorting Networks and Their Applications, *Proceedings of the AFIPS Spring Joint Computing Conference,* vol. 32, pp.307-314, 1968.

[BS78] G. Baudet and D. Stevenson, Optimal Sorting Algorithms for Parallel Computers, *IEEE Transactions on Computers*, C-27:84-87, 1978.

[BLE90] G.E. Blelloch, *Vector Models for Data-Parallel Computing*, The MIT Press, 1990.

[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,* July 21-24, 1991, Hilton Head, SC, pp.3-16.

[CHL89] B. Chlebus, Sorting Within Distance Bound on a Mesh-Connected Arrays, *International Symposium on Optimal Algotihms,* vol. 401 of *Lecture Notes in Computer Science*, pp. 232-238, Springer-Verlag, NY, 1989.

[COLE88] R. Cole, Parallel Merge Sort, *SIAM Journal on Computing,* pp. 770-785, 1988.

[CV86] R. Cole and U. Vishkin, Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pp. 206-219, 1986.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, The MIT Press and McGraw Hill, 1990.

[CP90] R.E. Cypher and C.G. Plaxton, Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pp. 193-203, May 1990.

[DNS91] D.J. DeWitt, J.F. Naughton, D.F. Schneider, Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting, Computer Sciences TR#1043, University of Wisconsin - Madison, 1991.

[FKO86] E. Felten, S. Karlin, and S. Otto, Sorting on a Hypercube, Hm 244, Caltech/JPL, 1986.

[FM70] W.D. Frazer and A.C. McKellar, Samplesort: A Sampling Approach to Minimal Storage Tree Sorting, *Journal of the ACM,* 17(3):496-507, 1970.

[HOE56] W. Hoeffding, On the Distribution of the Number of Successes in Independent Trials, *Annals of Mathematical Statistics,* 27:713-721, 1956.

[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.

[KKNT91] C. Kaklamanis, D. Krizanc, L. Narayanan, and T. Tsantilas, Randomized Sorting and Selection on Mesh-Connected Processor Arrays, *3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* July 21-24, 1991, Hilton Head, SC, pp. 17-28.

[KUN88a] M. Kunde, Routing and Sorting on Mesh-Connected Arrays, *Aegan Workshop on Computing: VLSI Algorithms and Architectures,* vol. 319 of *Lecture Notes in Computer Science*, pp. 423-433, Springer-Verlag, NY, 1988.

[KUN88b] M. Kunde, l-selection and Related Problems on Grids of Processors, *Aegan Workshop on Computing: VLSI Algorithms and Architectures,* vol. 319 of *Lecture Notes in Computer Science*, pp. 423-433, Springer-Verlag, NY, 1988.

[LEI865] F.T. Leighton, Tight Bounds on the Complexity of Parallel Sorting, *IEEE Transactions on Computers,* C-34(4):344-354, April 1985.

[LP90] T. Leighton and G. Plaxton, A (Fairly) Simple Circuit That (Usually) Sorts, *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,* pp. 264-274, October 1990.

[NS82] D. Nassimi and S. Sahni, Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network, *Journal of the ACM,* 29(3):642-667, July 1982.

[PAT90] M.S. Paterson, Improved Sorting Networks with $O(\log n)$ Depth, *Algorithmica,* 5:75-92, 1990.

[PLAX89] C.G. Plaxton, Efficient Computation on Sparse Interconnection Networks, Technical Report STAN-CS-89-1283, Stanford University, Department of Computer Science, September 1989.

[PRIN90] J.F. Prins, 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.

[QUI89] M.J. Quinn, Analysis and Benchmarking of Two Parallel Sorting Algorithms: Hypersort and Quickmerge, *BIT,* 29(2):239-250, 1989.

[RR89] S. Rajasekaran and J.H. Reif, Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms, *SIAM Journal on Computing,* 18(3):594-607, June 1989.

[RV83,87] 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.

[REI85] R. Reischuk, Probabilistic Parallel Algorithms for Sorting and Selection, *SIAM Journal of Computing,* 14(2):396-411, May 1985.

[SS86] C. Schnorr and A. Shamir, An Optimal Sorting Algorithm for Mesh Connected Computers, *Symposium on the Theory of Computation,* pp. 255-263, 1986.

[SG88] S.R. Seidel and W.L. George, Binsorting on Hypercubes with d-port Communication, *Proceedings of the Third Conference on Hypercube Concurrent Computers,* pp. 1455-1461, January 1988.

[TK77] C.D. Thompson and H.T. Kung, Sorting on a Mesh-connected Parallel Computer, *Communications of the ACM* 20(4):263-271, 1977.

[U83] J. Ullman, *Computational Aspects of VLSI,* Computer Science Press, 1983

[VD88] P. Varman and K. Doshi, Sorting with Linear Speedup on a Pipelined Hypercube, Technical Report TR-9902, Rice University, Department of Electrical and Computer Engineering, February 1988.

[WAG89] B.A. Wagar, Hyperquicksort: A Fast Sorting Algorithm for Hypercubes, *Hypercube Multiprocessors 1987 (Proceedings of the Second Conference on Hypercube Multiprocessors)* (ed. M.T. Heath), pp. 292-299, Philadelphia, PA, 1987. SIAM.

[WAG90] B.A. Wagar, *Practical Sorting Algorithms for Hypercube Computers,* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, July 1990.

[WS88] Y. Won and S. Sahni, A Balanced Bin Sort for Hypercube Multicomputers, *Journal of Supercomputing,* 2:435-448, 1988.