B6-2

Towards a Realizable Flashsort

Martin Nilsson and Hidehiko Tanaka
Hidehiko Tanaka Lab., Dept. of Electrical Engineering,
The University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, JAPAN

Abstract

Sorting is fundamental operation for parallel computers. For instance, permutation routing is a sorting operation with the destinations as keys. Efficient one-to-many routing depends on sorting as an important subroutine. For high-connectivity networks, such as hypercubes, butterflies, or shuffle-exchange networks, parallel sorting has been shown to have a time complexity lower bound of $O(log^2 n)$, where n is the number of processing elements. However, there is a recent *probabilistic* sorting algorithm, Flashsort, which has complexity $O(log\, n)$. In its original formulation, this algorithm is very complicated. In this paper, we will give a variant description of this algorithm, which we believe is much simpler to understand, although the algorithm as we describe it is not as fast as the original Flashsort. Our description should help in understanding and implementing the original Flashsort algorithm.

Keywords: Probabilistic, parallel, sorting, Flashsort, complexity, routing, algorithm, high-connectivity, networks, hypercube, butterfly, shuffle-exchange.

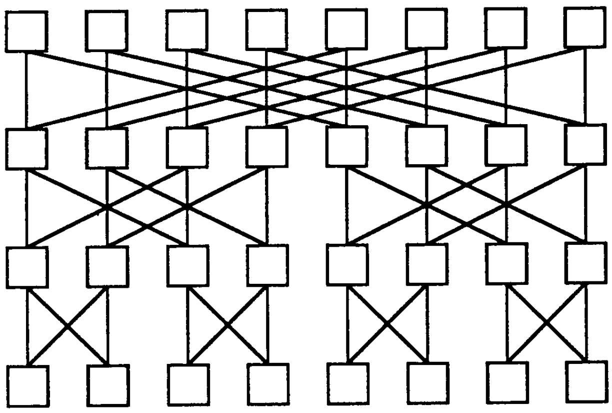
1 Introduction

The word "sorting" is usually associated with application software, not with the fundamental operations, or the primary instruction set, of the computer itself. This is probably correct for sequential computers. However, for parallel computers, sorting becomes a crucial component of communication between processing elements. For instance, permutation routing, i.e. when all processors send messages to other processors, but no two processors send messages to the same destination, can be seen as a sorting operation, where destination tags are used as keys.

A harder problem is one-to-many routing, where any processor may request a message from any other processor. The problem here is that if many processors make requests from the same processor, there will be congestion at this processor. By combining sorting and a *parallel prefix* operations [4], [1], this kind of routing problem can be solved in the same order of time as the sorting operation [4].

Parallel prefix operations for processor interconnection networks with high connectivity, such as hypercube, butterfly, or shuffle-exchange networks, have complexity $O(log\, n)$, where n is the number of processors. The bad news is that fully general sorting can not be done faster than $O(log^2 n)$ [2]. The good news is that if we allow *probablistic* sorting, there is an ingenious algorithm called Flashsort [3], which has probabilistic complexity $O(log\, n)$, and which works with probability arbitrarily close to one. In other words, if the algorithm doesn't work, we re-run it. Since the probability is very high that the algorithm will succeed, we will never have to run the algorithm many times, although we cannot give a strict upper bound.

In the rest of this paper, we will assume that the network is a butterfly network. This makes it easy to map the algorithms onto hypercubes or shuffle-exchange networks [4].



In section 2 and 3, we first describe two important subroutines for permutation routing and parallel prefix operations. In section 4, we will describe the algorithm itself. In section 5, we will compare our description with the original Flashsort algorithm, and discuss the results.

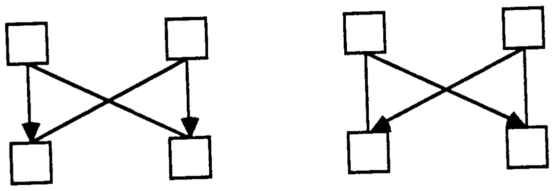
2 Permutation routing

Permutation routing is easy to do in probabilistic $O(\log n)$ time [5]. The following two steps suffice:

- Let each processor send its message to a random destination. (These destinations must all be different.)
- Let these random destinations forward the messages to their final destinations.

In order to implement this algorithm, we need to be able to compute a random permutation. This can be done in the following way:

- Each node in the top row of the butterfly is connected to two nodes in the second row. Some other node in the top row is connected to the same nodes. The two nodes in the top row should randomly agree to either: send their own number to the node directly below themselves, or: mutually send their own number diagonally to the node below the other top row node.



- The nodes in the second row use a similar procedure to pass on the numbers the received from above.
- When the numbers reach the final row, they will be a random permutation.

3 Parallel prefix operations

The *parallel prefix* operation calculates the values x_1, $x_1 + x_2$, $x_1 + x_2 + x_3$, etc. "+" does not have to be addition, but can be any associative operation.

An algorithm for doing the parallel prefix operation is as follows:

- Node i in the first row sends x_i to the node directly below it, and to the node below and to the *right*, if there is one.
- If a node in the second row receives two values, it performs the operation on them, and passes the result down, and if possible, to the right. If it just receives one value, it passes it the way it is.
- Nodes in subsequent rows operate in the same way as those in the second.
- The values in the final row will be the values sought.

4 The Algorithm

We use recursive doubling in a way similar to the famous FFT: A sorting problem of size n is reduced to two sorting problems of size $\sqrt{n}$.

The algorithm goes as follows:

- Randomly permute the elements we want to sort, using the algorithm we described above.
- Choose $\sqrt{n}$ elements, called *splitters*, randomly. This can be done by a slight variation of the random permutation algorithm.
- Sort the splitters. This is done by recursive application of the algorithm on the splitters as a sorting problem of size $\sqrt{n}$.
- Distribute the splitters over the top half of the butterfly, in such a way that the median splitter ($s_{1/2}$) is sent to all nodes in the top row, the middle splitter of the lower half of splitters ($s_{1/4}$) is sent to all nodes in the left half of the second row, the middle splitter of the right half of splitters ($s_{3/4}$) is sent to all nodes in the upper half of the second row, and so on. This distribution can be done easily by passing splitters up the butterfly after sorting them has left them in the bottom row.
- Now we are ready for the real sorting phase: Move elements to be sorted down the top half of the butterfly. Splitters decide which way elements are passed down, in a way similar to quicksort: If an element is less than the splitter, we move down to the left. If it is greater or equal, we move it down to the right.

  When the elements reach the middle of the butterfly, the sorting problem has been reduced to $\sqrt{n}$ parallel sorting subproblems of size $\sqrt{n}$. We can again apply the algorithm recursively, *on all subproblems in parallel.*

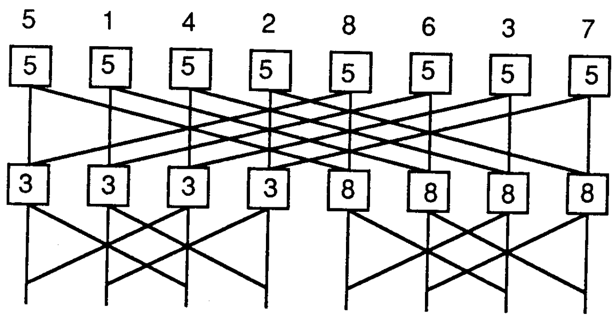
- When we reach the final row, elements will appear sorted. It might happen that there are several elements in consecutive elements at the same node, but probabilistic analysis [3] show that this number will be so small that the elements can be sorted locally using an arbitrary *local* sorting algorithm.

  We can now number all the sorted elements by a parallel prefix operation, and route them to the corresponding processor.

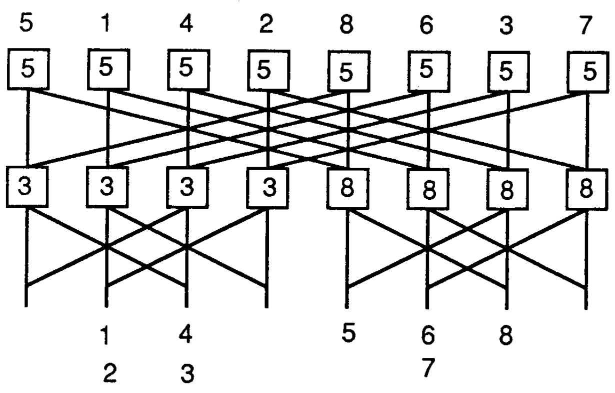
It will normally happen during the execution of this algorithm that several elements want to travel down the same path, and that one of them will have to wait. This will of course slow down the algorithm. However, probabilistic analysis shows that this number will always be so small (with very high probability) that it will not affect the total time complexity [3].

An example will clarify the algorithm.

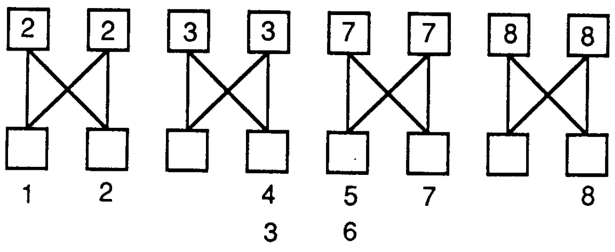
Suppose that $n = 8$, and that we have the following randomized sequence: 5, 1, 4, 2, 8, 6, 3, and 7. Suppose that we select splitters by random and sorting the gives 3, 5, 8. Distributing them over the top half of the butterfly produces the situation:



We now pass the elements down through the top half, which leaves them at the third row, in the following order:



We now have four subproblems of size two. Recursive application of the algorithm tells us to select one splitter for each subproblem. Let's say that 2, 3, 7, and 8 are selected, respectively. After distributing these splitters, and sending down the elements, we have the final picture:



Here, elements 3, 4 and 5, 6 ended up at the same node, so for these local sorting is necessary.

5 Related work, Discussion and Conclusions

The version of Flashsort we have described is a simplified variant of the version described in [3] and [4]. There are two main differences: Our variant selects $\sqrt{n}$ splitters, while [3] selects some number > 6. The advantage we get is that we can apply the idea of recursive doubling, and the the algorithm is considerably simplified. The other main difference is that the primitive operations of our algorithm are synchronized, and are applied to all nodes at the same time, while [3] uses asynchronous operations. This simplifies our algorithm, but in fact slows it down to become $O(log^2 n)$. In order to achieve $O(log\, n)$ complexity, *elements must be passed on as soon as possible.* A real implementation must take this into account.

As the Flashsort algorithm requires some overhead, we don't expect it to be much more efficient than Batcher's sort for small values of n, but it may become quite interesting for massively parallel computers, where $n > 1000$, or ${}^2log\, n > 10$.

6 Acknowledgements

We are grateful for stimulating discussions with members of the Inference Engine Special Interest Group at Tokyo University, and of the Parallel Programming Systems working group at ICOT. We would like to in particular thank Hagihara-sensei of Osaka University for suggesting study of the Flashsort algorithm. This work was supported by a scholarship from the Japanese Ministry of Education, and from the Swedish National Board for Technical Development.

References

[1] Hillis, W.D. and Steele, G.L., Jr.: *Data Parallel Algorithms.* CACM, Vol. 29, No. 12. p 1170-1183.

[2] Knuth, D.E.: *The Art of Computer Programming, Vol. III: Sorting and Searching.* Addison-Wesley, Reading, Mass. 1973.

[3] Reif, J.H. and Valiant, L.G.: *A logarithmic time sort for linear size networks.* In Proc. Fifteenth Annual ACM Symposium on the Theory of Computing, p. 10-16. 1983.

[4] Ullman, J.D.: *Computational Aspects of VLSI.* Computer Science Press, Maryland. 1984.

[5] Valiant, L.G. and Brebner, G.J.: *Universal schemes for parallel communication.* In Proc. Thirteenth Annual ACM Symposium on the Theory of Computing. p. 263-277. 1981.