1-1-1998

Sorting in Parallel

Ran Libeskind-Hadas
Harvey Mudd College

Sorting in Parallel

Ran Libeskind-Hadas

INTRODUCTION. In 1842, L. F. Menabrea anticipated the benefits of parallel computing in an article [4] that appeared in the Swiss Journal *Bibliotheque universelle de Geneve*:

> When a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes.

Although more than a century passed before Menabrea's vision became a reality, today parallel computers with hundreds and even thousands of processors are used in a broad range of applications.

One of the great challenges of parallel computing is the design of algorithms that make efficient use of the large number of processors in the machine. In this paper we explore parallel algorithms for one of the most fundamental problems in computer science: the problem of sorting a list of numbers. Specifically, given a list $a_1, a_2, \dots, a_n$ of integers, the sorting problem is that of finding a permutation of this list, π, such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

Before considering parallel sorting algorithms, we examine a *sequential* sorting algorithm; an algorithm for a computer with a single processor. One sequential algorithm for sorting is known as *selection sort*. The selection sort algorithm first examines each of the n integers $a_1, \dots, a_n$ to find an integer a_i with minimum value. Elements a_1 and a_i are then exchanged so that the element with minimum value is now in the first position in the list. Next, the algorithm examines each of the $n - 1$ integers $a_2, \dots, a_n$ to find an integer with minimum value in this portion of the list. This integer is exchanged with a_2 so that now the second smallest integer is in the second position in the list. In general, in the k^{th} iteration the algorithm examines integers $a_k, \dots, a_n$, finds a minimum element in this list, and exchanges it with a_k. At the end of n^{th} iteration, the list is sorted. Observe that the first iteration of the algorithm requires n steps to find an element with minimum value and then some constant additional number of steps, c, to exchange this minimum value with a_1. The next iteration requires $n - 1$ steps to locate a minimum value in the remaining list and c steps to perform the exchange. In general, the k^{th} iteration requires $n - k + 1$ steps to locate a minimum value in the remaining list and c steps to perform the exchange. Since there are n iterations, the total number of steps or *running time* of the algorithm is

$$n + (n-1) + (n-2) + \cdots + 1 + cn = \frac{n(n+1)}{2} + cn = \frac{1}{2}n^2 + \frac{2c+1}{2}n.$$

Since the dominant term in this expression is n^2, we say that the *asymptotic running time* of the selection sort algorithm is n^2.

The selection sort algorithm is known as a *comparison algorithm* because the sorted order is determined exclusively by comparisons among the input elements. Several sequential comparison algorithms are known for sorting whose asymptotic running time is only $n \log n$ [1]. These algorithms are, in fact, asymptotically optimal; a fundamental result states that every sequential comparison algorithm for sorting n elements must have an asymptotic running time of at least $n \log n$ [1]. In the next section we show that a parallel computer employing multiple processors can sort n integers in substantially lower asymptotic running time.

Sorting in Parallel. Consider a parallel computer comprising n processors connected in a *linear array*, as shown in Figure 1. The leftmost and rightmost processors of the linear array, labeled p_1 and p_n, respectively, have one *neighbor*; all the remaining processors have two neighbors.

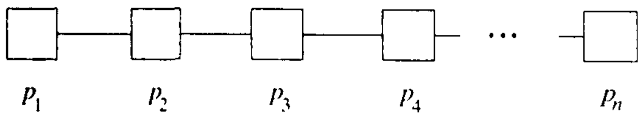

Figure 1. A linear array with n processors.

Each processor is assumed to have a *local memory* that stores some fixed amount of data. In addition, the processors have access to a global counter or "clock" whose initial value is one and is incremented by one at some fixed frequency. At each clock step, every processor performs the following tasks:

1. Send some data from local memory to the neighbor processor(s).
2. Receive data from the neighbor processor(s) and store in local memory.
3. Compute some function of the values in local memory.
4. Store result of computation in local memory.

Suppose we are given a set of n integers distributed among the n processors of a linear array, $p_1, \dots, p_n$. Our objective is to sort this set of integers so that at the end of the computation the integer stored at p_i is less than or equal to the integer stored at p_{i+1}, for $1 \le i < n$. One of the simplest parallel sorting algorithms is known as *odd-even sorting*. On odd numbered clock steps, each odd numbered processor p_i is paired with its right neighbor p_{i+1}. Each processor sends its integer to its partner, both processors compare the two integers, and finally the smaller integer is stored by p_i while the larger integer is stored by p_{i+1}. Thus, each of these pairs of processors effectively compare their integers and swap them if they are out of order. Similarly, on even numbered clock steps each even numbered processor compares its integer with the one stored by the processor to its right and the two numbers are swapped if they are out of order. Somewhat surprisingly, after n clock steps the numbers are sorted. An example of the odd-even sorting algorithm for a linear array with four processors is shown in Figure 2. Figure 2(a) shows the initial configuration and each subsequent row shows the contents of the array after the next clock step.

Several techniques can be used to show that the odd-even algorithm correctly sorts all input sequences. Perhaps the simplest and most elegant proof of this assertion is based on a clever result due to Donald Knuth. Knuth showed that in order to prove that an *oblivious comparison-exchange* sorting algorithm correctly

Is it possible to sort n elements on n processors in asymptotically fewer than n steps if an interconnection network other than a linear array is used? We show that the answer is "yes" by exhibiting a parallel sorting algorithm, known as *Shearsort*, for a two-dimensional array with dimensions $\sqrt{n} \times \sqrt{n}$. For simplicity, we assume that $\sqrt{n}$ is a power of two.

A single phase of the Shearsort algorithm consists of sorting all of the rows of the array and then sorting all of the columns of the array. Odd-indexed rows are

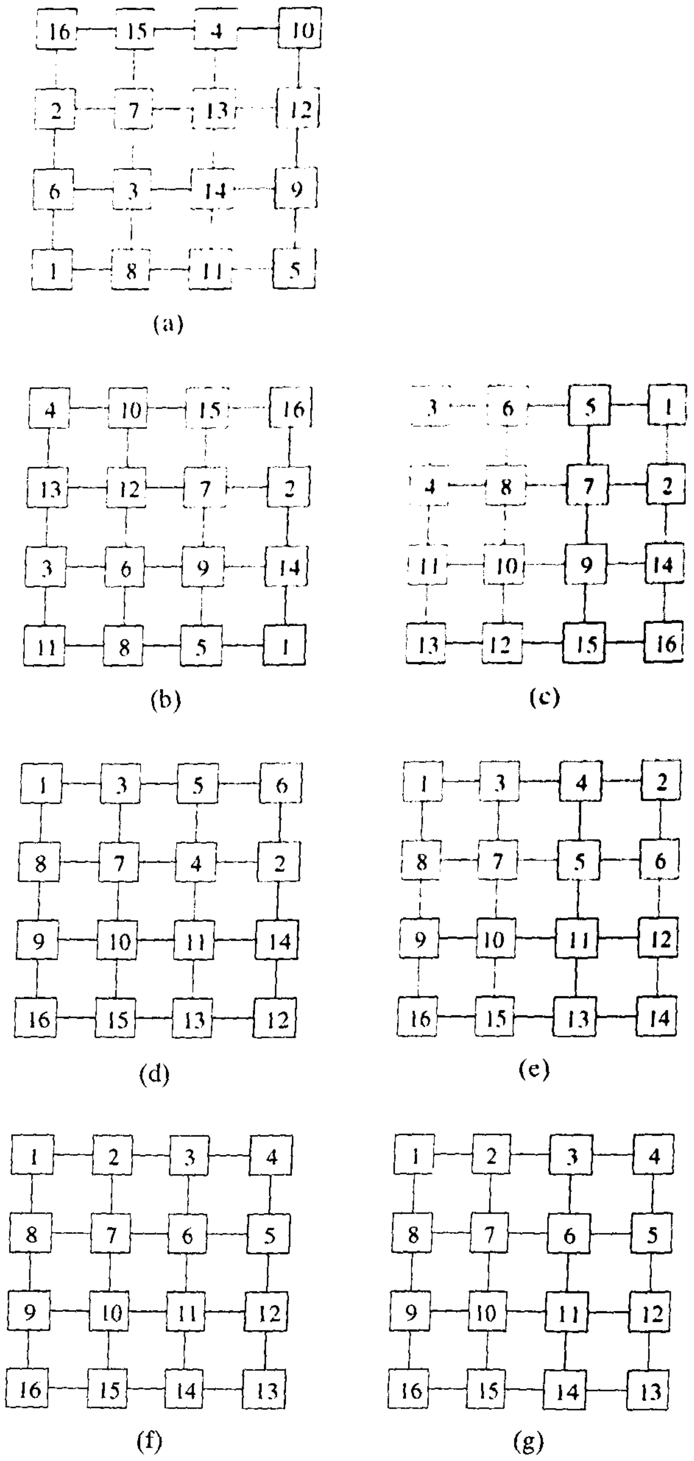

Figure 3. The phases of the Shearsort algorithm in a 4×4 array. (a) Initial configuration. (b) After row sorting in phase 1. (c) After column sorting in phase 1. (d) After row sorting in phase 2. (e) After column sorting in phase 2. (f) After row sorting in phase 3. (g) After column sorting in phase 3. The array is now sorted in a snake-like order.

sorted in increasing order while even-indexed rows are sorted in reverse order. All columns, however, are sorted in increasing order. Remarkably, after $1 + \frac{1}{2}\log_2 n$ phases the n elements are sorted in a snake-like fashion. An example of the phases of the Shearsort algorithm is shown in Figure 3. In this example, $n = 16$ and thus 3 phases of the algorithm are performed.

A direct proof showing that Shearsort correctly sorts its input in $1 + \frac{1}{2}\log_2 n$ phases is quite involved. However, since Shearsort is an oblivious comparison-exchange algorithm, we can again appeal to Knuth's 0-1 Sorting Lemma to prove the correctness of the algorithm.

Consider an arbitrary input comprising n 0's and 1's on a $\sqrt{n} \times \sqrt{n}$ array. We partition the rows of the array into three groups: *solid zero rows* contain only 0's, *solid one rows* contain only 1's, and *mixed rows* contain some 0's and some 1's. Since each phase of the algorithm involves sorting the columns in increasing order, at the end of each phase all solid zero rows must be above all mixed rows. If this were not the case, then a mixed row would be above some solid zero row and thus some column would contain a 1 above a 0, contradicting the fact that the columns have been sorted. A similar argument shows that all mixed rows must be strictly above all solid one rows at the end of each phase.

The key to proving the correctness of Shearsort is to show now that each phase of the algorithm reduces the number of mixed rows by a factor of two or more. To see this, we examine pairs of adjacent mixed rows (with possibly one unpaired row left over). A pair of rows may contain more 0's than 1's, more 1's than 0's, or an equal number of 0's and 1's. Figure 4 depicts these three cases after the rows have been sorted in alternate increasing and decreasing order. In each of the cases depicted in the figure, the rows could be reversed depending on the parities of the row indices.

0 0 0 ... 0 1 1 1 ... 1	0 0 0 ... 0 1 1 1 ... 1	0 0 0 ... 0 1 1 1 ... 1
1 1 1 ... 0 0 0 0 ... 0	1 1 1 ... 1 1 1 1 ... 0	1 1 1 ... 1 0 0 0 ... 0
(a)	(b)	(c)

Figure 4. The three possible cases for pairs of mixed rows. (a) The pair contains more 0's than 1's. (b) The pair contains more 1's than 0's. (c) The pair contains an equal number of 0's and 1's.

Next, the algorithm sorts each column in increasing order. For the moment, assume that the columns are sorted using the following peculiar algorithm: First, in each adjacent pair of mixed rows, if a 1 appears directly above a 0 the two digits are swapped. The result of this procedure for the cases in Figure 4(a), (b), and (c) are shown in Figure 5(a), (b), and (c), respectively. Observe that in each of these cases at least one solid row is created. If a solid zero row is created, as in cases (a) and (c), that row is moved to the region of solid zero rows at the top of the array.

0 0 0 ... 0 0 0 0 ... 0	0 0 0 ... 0 1 1 1 ... 0	0 0 0 ... 0 0 0 0 ... 0
1 1 1 ... 0 1 1 1 ... 1	1 1 1 ... 1 1 1 1 ... 1	1 1 1 ... 1 1 1 1 ... 1
(a)	(b)	(c)

Figure 5. The result of swapping 1's and 0's for the three cases of Figure 4. (a) The case of more 0's than 1's. (b) The case of more 1's than 0's. (c) The case of an equal number of 0's and 1's.

Likewise, if a solid one row is created, as in cases (b) and (c), that row is moved to the region of solid one rows at the bottom of the array. Once these new solid rows have been moved, we complete the column sorting step by sorting the remaining 0's and 1's in each column. By using this peculiar column sorting procedure, each pair of mixed rows is replaced by at most one mixed row at the end of the phase. Initially, each of the $\sqrt{n}$ rows is considered to be mixed and thus at most $\log_2 \sqrt{n}$ phases are needed to reduce the array to at most one mixed row. One additional phase serves to sort the remaining mixed row. Thus, $1 + \frac{1}{2}\log_2 n$ phases suffice to sort the entire array.

In each phase of Shearsort, each odd-indexed row can be sorted in $\sqrt{n}$ clock steps using odd-even sorting and a symmetrical algorithm can be applied to even-indexed rows. Since all rows can be sorted simultaneously, this takes $\sqrt{n}$ clock steps. For the columns, we used the peculiar sorting algorithm previously described. In practice, of course, we would like to use the odd-even sorting algorithm because we know that its running time is asymptotically optimal. Notice though that the net effect of the two column-sorting schemes, the peculiar one and odd-even sorting, is exactly the same. Both algorithms sort the columns! Thus, the result of applying odd-even sorting to the columns must also reduce the number of mixed rows by a factor of two or more. Thus, Shearsort requires a total of $1 + \frac{1}{2}\log_2 n$ phases and each phase requires $2\sqrt{n}$ clock steps, resulting in a total of $2\sqrt{n} + \sqrt{n}\log_2 n$ clock steps to sort n numbers. Thus, the running time of Shearsort is asymptotically $\sqrt{n}\log_2 n$, which is less than the asymptotic running time of n steps incurred by the odd-even sorting algorithm.

Conclusion and Further Reading. Although Shearsort's asymptotic running time is superior to that of the odd-even sorting algorithm, it is possible to do even better. We observed that n is an asymptotic lower bound on the number of clock steps required to sort on a linear array. Similarly, in a two-dimensional array with dimensions $\sqrt{n} \times \sqrt{n}$, an asymptotic lower bound is $\sqrt{n}$ since $2\sqrt{n} - 2$ clock steps are necessary to move an element at one corner of the array to the opposite corner of the array. Several algorithms have been proposed that sort n elements on a $\sqrt{n} \times \sqrt{n}$ two-dimensional array in $\sqrt{n}$ asymptotic running time. Thompson and Kung [5] describe one such algorithm. These algorithms can be generalized to sort n elements on a d-dimensional array with dimensions $\sqrt[d]{n} \times \cdots \times \sqrt[d]{n}$ in $\sqrt[d]{n}$ asymptotic running time.

Do there exist sorting algorithms that attain the theoretical lower bound of $\log n$ asymptotic running time for sorting n integers on n processors? While this is still an open problem, sorting algorithms are known for some topologies that come very close to this lower bound. For example, a sorting algorithm with asymptotic running time of $\log n \log\log n$ is known for hypercubic networks. Moreover, *randomized algorithms* are known for hypercubic networks that sort n elements on n processors with very high probability in $\log n$ asymptotic running time. A beautifully written book by Leighton [3] describes these and a number of related results in detail. Leighton's text also examines parallel algorithms for several classical problems in addition to sorting.

REFERENCES

1. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.
2. D. E. Knuth, *Searching and Sorting*, volume 3 of *The Art of Computer Programming*, Addison-Wesley, 1973.

3. F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, 1992.
4. Ada Lovelace, Sketch of the Analytical Engine by L. F. Menabrea, *Taylor's Scientific Memoirs*, 3 (1843), 352–376 (translation of L. F. Menabrea, Notions sur la machine analytique de M. Charles Babbage, *Bibliotheque universelle de Geneve* 41 (1842), 352–376.)
5. C. D. Thompson and H. T. Kung, Sorting on a Mesh-Connected Parallel Computer, *Communications of the ACM* 20 no. 4 (1977), 263–271.

RAN LIBESKIND-HADAS is the Iris and Howard Critchell Assistant Professor in the Department of Computer Science at Harvey Mudd College. He obtained his Ph.D. in computer science at the University of Illinois at Urbana-Champaign and his bachelor's degree in applied mathematics at Harvard University. His research interests are in algorithms and parallel computer architecture. His father, Shlomo Libeskind (professor of mathematics at the University of Oregon), and his doctoral research advisor, C. L. Liu, have been tremendously influential teachers and mentors.
Harvey Mudd College, Claremont, CA 91711
hadas@cs.hmc.edu

On the Summation of Squared Integers

An easy computation $k^2 = C(k+2,3) - C(k,3)$, where $C(n,m) = \binom{n}{m}$. This equality is more intuitively derived by observing that given a $(k+2)$-element set with distinct fixed elements u and v, $C(k+2,3) - C(k,3)$ gives the number of 3-element subsets containing at least one of u or v. Alternatively, this number can be computed via the Inclusion-Exclusion Principle as

$$C(k+1,2) + C(k+1,2) - k = \frac{(k+1)k}{2} + \frac{(k+1)k}{2} - k = k^2,$$

where $C(k+1,2)$ gives the number of 3-element subsets containing u (respectively, v). Therefore for every positive integer n,

$$\begin{aligned}\sum_{k=1}^{n} k^2 &= \sum_{k=1}^{n} [C(k+2,3) - C(k,3)] \quad \text{(a telescoping sum)}\\ &= C(n+2,3) - C(n+1) \qquad (C(2,3) = C(1,3) = 0)\\ &= \frac{n(n+1)(2n+1)}{6}.\end{aligned}$$

Hidefumi Katsuura, San Jose State University