INFO-F-404 : Operating Systems 2

2017 – 2018 Project 2 : BITONIC

1 Main goal

Implement the parallel sorting algorithm *Bitonic* using MPI on HYDRA.

2 Introduction

Confronted to the impossibility to persevere the pursuit of Moore's law, the response of the electronic industry to provide always more computing power resides in parallel and multicore/multiprocessor architectures.

Taking advantage of this many cores paradigm requires new (parallels) algorithms.

In the field of sorting, *Bitonic* —which has been seen during the lectures— is one of them.

3 Project details

First, note that the reference[1] from the lectures is annexed to this document, do not hesitate to consult it.

You are asked to implement the simplest form of the *Bitonic* sort algorithm that orders bitonic sequences (with a length being a power of 2) using OpenMPI on the ULB's Hydra cluster. We will consider that, for a sequence of length n, you will have $(n/2) + 1$ nodes at your disposal. One *master node* and $n/2$ *computing nodes*. The *master* will be responsible of transmitting selected parts of the sequence to the *computing nodes* at the beginning, collect results after processing by the *computing nodes* and printing the sorted sequence on screen.

3.1 Implementations details

For the sake of simplicity, we will consider a sequence of size 16 directly encoded in the master's part of your source file. However, your solution shall stay generic. For example, given that your source file starts like the excerpt on the next page, to process a sequence of length 32 rather than 16, only `if (n==16)` and the initialisation of vector `A` shall be modified. Here follows an example of how your project will be launched and the expected results:

```
[cternon@nic50 P2]$ mpirun -np 9 ./bitonic
Sorted sequence : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

[1] Padua, David (Ed.) ; Encyclopedia of Parallel Computing, vol. 1 A–D ; Springer, 2011

```
#include <stdlib.h>
#include <stdio.h>
#include <cmath>
#include "mpi.h"

int main(int argc, char **argv) {

    MPI_Init( &argc, &argv );
    int rank, nb_instance;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_instance);
    MPI_Status status;

    int cnodes=nb_instance-1;
    int n=cnodes*2;

    if (n==16) {

        if (rank == 0) {
        // master's portion

            int A[n] = {14,16,15,11,9,8,7,5,4,2,1,3,6,10,12,13};
```

You are asked to keep initialisation of the bitonic sequence to be sorted in the master's part and give a special attention to which parts shall be transmitted to which nodes and how. The choices you make for this shall be explained in your report.

3.2 Report

You should also write a short report that contains:

1. short description of your code (diagrams) and implementation choices,
2. a description of the protocol that you used to implement the program (messages transiting between processes),
3. a description of the performances (and limitations) of your implementation,
4. a section where you describe difficulties that you met during this project (and solutions that you found).

3.3 Bonus part

This part is not mandatory, but it is worth some bonus points (only if the first part works correctly). The first part proposes the implementation of the *Bitonic sort* algorithm that orders bitonic sequences. However, it is easy to modify an arbitrary sequence into a sequence of bitonic sequences. For the bonus part, you are proposed to develop a second source file that sorts arbitrary sequences (still with a length being a power of 2) using the *Bitonic sort* algorithm.

4 Submission and planning

This project should be done in groups of 3 maximum, you may choose your partner(s).
It has to be submitted before 23:55:00 o'clock on December 21th, 2017.
Your project has to work properly (compile and execute) on HYDRA cluster at ULB.
Please observe that Hydra will be under maintenance from december 11th to 15th.

You have to submit (in a *zip* file) a folder that contains at least the following files:

- your C++ source,
- a short report (pdf format, figures and graphics are welcome).

The zip file with your project has to be submitted to the UV: `http://uv.ulb.ac.be`. The name of the folder and of the zip file will be as follows: if Jean Dupont made his project with Albertine Vanderbeken, they should send a file named *dupont-vanderbeken.zip* (that contains a folder of the same name, that contains all your project files).

For questions on this project, please refer to **cedric.ternon@ulb.ac.be** with subject "[INFO-F-404] project2".

Good work!

which takes temporal overlap of actions into account, and the *history preserving bisimulation* [2], which even keeps track of causal relations between actions. For this purpose, system representations such as *Petri nets* or *event structures* are often used instead of labeled transition systems.

Bibliography

1. Aczel P (1988) Non-well-founded Sets, CSLI Lecture Notes 14. Stanford University, Stanford, CA
2. van Glabbeek RJ (1990) Comparative concurrency semantics and refinement of actions. PhD thesis, Free University, Amsterdam. Second edition available as CWI tract 109, CWI, Amsterdam 1996
3. Hennessy M, Milner R (1985) Algebraic laws for nondeterminism and concurrency. J ACM 32(1):137–161
4. Hollenberg MJ (1995) Hennessy-Milner classes and process algebra. In: Ponse A, de Rijke M, Venema Y (eds) Modal logic and process algebra: a bisimulation perspective, CSLI Lecture Notes 53, CSLI Publications, Stanford, CA, pp 187–216
5. Milner R (1990) Operational and algebraic semantics of concurrent processes. In: van Leeuwen J (ed) Handbook of theoretical computer science, Chapter 19. Elsevier Science Publishers B.V., North-Holland, pp 1201–1242

Further Readings

Baeten JCM, Weijland WP (1990) Process algebra. Cambridge University Press

Milner R (1989) Communication and concurrency. Prentice Hall, New Jersey

Sangiorgi D (2009) on the origins of bisimulation and coinduction. ACM Trans Program Lang Syst 31(4). doi: 10.1145/1516507.1516510

Bisimulation Equivalence

►Bisimulation

Bitonic Sort

Selim G. Akl
Queen's University, Kingston, ON, Canada

Synonyms

Bitonic sorting network; Ditonic sorting

Definition

Bitonic Sort is a sorting algorithm that uses comparison-swap operations to arrange into nondecreasing order an input sequence of elements on which a linear order is defined (for example, numbers, words, and so on).

Discussion

Introduction

Henceforth, all inputs are assumed to be numbers, without loss of generality. For ease of presentation, it is also assumed that the length of the sequence to be sorted is an integer power of 2. The *Bitonic Sort* algorithm has the following properties:

1. *Oblivious:* The indices of all pairs of elements involved in comparison-swaps throughout the execution of the algorithm are predetermined, and do not depend in any way on the values of the input elements. It is important to note here that the elements to be sorted are assumed to be kept into an array and swapped in place. The indices that are referred to here are those in the array.
2. *Recursive:* The algorithm can be expressed as a procedure that calls itself to operate on smaller versions of the input sequence.
3. *Parallel:* It is possible to implement the algorithm using a set of special processors called "comparators" that operate simultaneously, each implementing a comparison-swap. A comparator receives a distinct pair of elements as input and produces that pair in sorted order as output in one time unit.

Bitonic sequence. A sequence $(a_1, a_2, \ldots, a_{2m})$ is said to be *bitonic* if and only if:

(a) Either there is an integer j, $1 \le j \le 2m$, such that

$$a_1 \le a_2 \le \cdots \le a_j \ge a_{j+1} \ge a_{j+2} \ge \cdots \ge a_{2m}$$

(b) Or the sequence does not initially satisfy the condition in **(a)**, but can be shifted cyclically until the condition *is* satisfied.

For example, the sequence (8, 9, 10, 12, 7, 5, 4, 3, 1) is bitonic, as it satisfies condition **(a)**. Similarly, the sequence (3, 2, 1, 5, 8, 9, 4), which does not satisfy condition **(a)**, is also bitonic, as it can be shifted cyclically to obtain (1, 5, 8, 9, 4, 3, 2).

Let $(a_1, a_2, \ldots, a_{2m})$ be a bitonic sequence, and let $d_i = \min(a_i, a_{m+i})$ and $e_i = \max(a_i, a_{m+i})$, for $i = 1, 2, \ldots, m$. The following properties hold:

(a) The sequences $(d_1, d_2, \ldots, d_m)$ and $(e_1, e_2, \ldots, e_m)$ are both bitonic.
(b) $\max(d_1, d_2, \ldots, d_m) \le \min(e_1, e_2, \ldots, e_m)$.

In order to prove the validity of these two properties, it suffices to consider sequences of the form:

$$a_1 \le a_2 \le \ldots \le a_{j-1} \le a_j \ge a_{j+1} \ge \ldots \ge a_{2m},$$

for some $1 \le j \le 2m$, since a cyclic shift of $\{a_1, a_2, \ldots, a_{2m}\}$ affects $\{d_1, d_2, \ldots, d_m\}$ and $\{e_1, e_2, \ldots, e_m\}$ similarly, while affecting neither of the two properties to be established. In addition, there is no loss in generality to assume that $m < j \le 2m$, since $\{a_{2m}, a_{2m-1}, \ldots, a_1\}$ is also bitonic and neither property is affected by such reversal.
There are two cases:

1. If $a_m \le a_{2m}$, then $a_i \le a_{m+i}$. As a result, $d_i = a_i$, and $e_i = a_{m+i}$, for $1 \le i \le m$, and both properties hold.
2. If $a_m > a_{2m}$, then since $a_{j-m} \le a_j$, an index k exists, where $j \le k < 2m$, such that $a_{k-m} \le a_k$ and $a_{k-m+1} > a_{k+1}$. Consequently:

$$d_i = a_i \text{ and } e_i = a_{m+i} \text{ for } 1 \le i \le k - m,$$

and

$$d_i = a_{m+i} \text{ and } e_i = a_i \text{ for } k - m < i \le m.$$

Hence:

$$d_i \le d_{i+1} \text{ for } 1 \le i < k - m,$$

and

$$d_i \ge d_{i+1} \text{ for } k - m \le i < m,$$

implying that $\{d_1, d_2, \ldots, d_m\}$ is bitonic. Similarly, $e_i \le e_{i+1}$, for $k - m \le i < m$, $e_m \le e_1$, $e_i \le e_{i+1}$, for $1 \le i < j - m$, and $e_i \ge e_{i+1}$, for $j - m \le i < k - m$, implying that $\{e_1, e_2, \ldots, e_m\}$ is also bitonic. Also,

$$\begin{aligned}\max(d_1, d_2, \ldots, d_m) &= \max(d_{k-m}, d_{k-m+1}) \\ &= \max(a_{k-m}, a_{k+1}),\end{aligned}$$

and

$$\begin{aligned}\min(e_1, e_2, \ldots, e_m) &= \min(e_{k-m}, e_{k-m+1}) \\ &= \min(a_k, a_{k-m+1}).\end{aligned}$$

Finally, since $a_k \ge a_{k+1}, a_k \ge a_{k-m}, a_{k-m+1} \ge a_{k-m}$, and $a_{k-m+1} \ge a_{k+1}$, it follows that:

$$\max(a_{k-m}, a_{k+1}) \le \min(a_k, a_{k-m+1}).$$

Sorting a Bitonic Sequence

Given a bitonic sequence $(a_1, a_2, \ldots, a_{2m})$, it can be sorted into a sequence $(c_1, c_2, \ldots, c_{2m})$, arranged in nondecreasing order, by the following algorithm $MERGE_{2m}$:

Step 1. The two sequences $(d_1, d_2, \ldots, d_m)$ and $(e_1, e_2, \ldots, e_m)$ are produced.
Step 2. These two bitonic sequences are sorted independently and recursively, each by a call to $MERGE_m$.

It should be noted that in Step 2 the two sequences can be sorted independently (and simultaneously if enough comparators are available), since no element of $(d_1, d_2, \ldots, d_m)$ is larger than any element of $(e_1, e_2, \ldots, e_m)$. The m smallest elements of the final sorted sequence are produced by sorting $(d_1, d_2, \ldots, d_m)$, and the m largest by sorting $(e_1, e_2, \ldots, e_m)$. The recursion terminates when $m = 2$, since $MERGE_2$ is implemented directly by one comparison-swap (or one comparator).

Sorting an Arbitrary Sequence

Algorithm $MERGE_{2m}$ assumes that the input sequence to be sorted is bitonic. However, it is easy to modify an arbitrary input sequence into a sequence of bitonic sequences as follows. Let the input sequence be $(a_1, a_2, \ldots, a_n)$, and recall that, for simplicity, it is assumed that n is a power of 2. Now the following $n/2$ comparisons-swaps are performed: For all odd i, a_i is compared with a_{i+1} and a swap is applied if necessary. These comparison-swaps are numbered from 1 to n/2. Odd-numbered comparison-swaps place the smaller of (a_i, a_{i+1}) first and the larger of the pair second. Even-numbered comparison-swaps place the larger of (a_i, a_{i+1}) first and the smaller of the pair second.

At the end of this first stage, $n/4$ bitonic sequences are obtained. Each of these sequences is of length 4 and can be sorted using $MERGE_4$. These instances of $MERGE_4$ are numbered from 1 to $n/4$. Odd-numbered instances sort their inputs in nondecreasing order while even-numbered instances sort their inputs in nonincreasing order. This yields $n/8$ bitonic sequences each of length 8.

The process continues until a single bitonic sequence of length n is produced and is sorted by giving it as input to $MERGE_n$. If a comparator is used to implement each comparison-swap, and all independent comparison-swaps are allowed to be executed in parallel, then the sequence is sorted in $((1+\log n)\log n)/2$ time units (all logarithms are to the base 2). This is now illustrated.

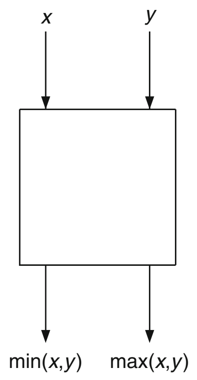
Implementation as a Combinational Circuit

Figure 1 shows a schematic diagram of a comparator. The comparator receives two numbers x and y as input, and produces the smaller of the two on its left output line, and the larger of the two on its right output line.

A combinational circuit for sorting is a device, built entirely of comparators that takes an arbitrary sequence at one end and produces that sequence in sorted order at the other end. The comparators are arranged in rows. Each comparator receives its two inputs from the input sequence, or from comparators in the previous row, and delivers its two outputs to the comparators in the following row, or to the output sequence. A combinational circuit has no feedback: Data traverse the circuit in the same way as water flows from high to low terrain.

Figure 2 shows a schematic diagram of a combinational circuit implementation of algorithm $MERGE_{2m}$, which sorts the bitonic sequence $(a_1, a_2, \ldots, a_{2m})$ into nondecreasing order from smallest, on the leftmost line, to largest, on the rightmost.

Clearly, a bitonic sequence of length 2 is sorted by a single comparator. Combinational circuits for sorting



Bitonic Sort. Fig. 1 A comparator

bitonic sequences of length 4 and 8 are shown in Figs. 3 and 4, respectively.

Finally, a combinational circuit for sorting an arbitrary sequence of numbers, namely, the sequence (5, 3, 2, 6, 1, 4, 7, 5) is shown in Fig. 5. Comparators that reverse their outputs (i.e., those that produce the larger of their two inputs on the left output line, and the smaller on the right output line) are indicated with the letter R.

Analysis

The *depth* of a sorting circuit is the number of rows it contains – that is, the maximum number of comparators on a path from input to output. Depth, in other words, represents the time it takes the circuit to complete the sort. The *size* of a sorting circuit is defined as the number of comparators it uses.

Taking $2m = 2^i$, the depth of the circuit in Fig. 2, which implements algorithm $MERGE_{2m}$ for sorting a bitonic sequence of length $2m$, is given by the recurrence:

$$\begin{aligned} d(2) &= 1 \\ d(2^i) &= 1 + d(2^{i-1}), \end{aligned}$$

whose solution is $d(2^i) = i$. The size of the circuit is given by the recurrence:

$$\begin{aligned} s(2) &= 1 \\ s(2^i) &= 2^{i-1} + 2s(2^{i-1}), \end{aligned}$$

whose solution is $s(2^i) = i2^{i-1}$.

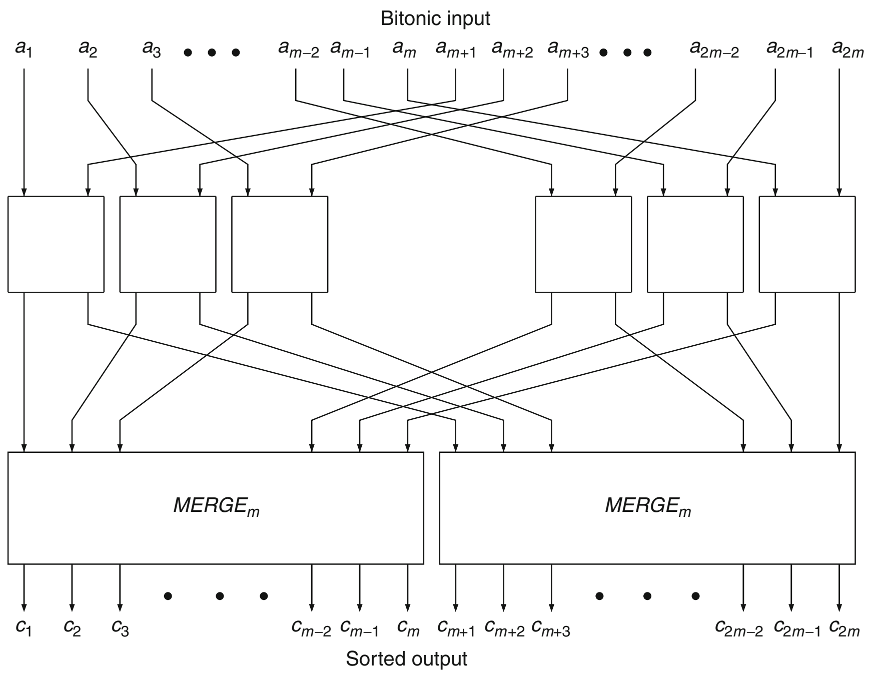
A circuit for sorting an arbitrary sequence of length n, such as the circuit in Fig. 5 where $n = 8$, consists of $\log n$ phases: In the ith phase, $n/2^i$ circuits are required, each implementing $MERGE_{2^i}$, and having a size of $s(2^i)$ and a depth of $d(2^i)$, for $i = 1, 2, \ldots, \log n$. The depth and size of this circuit are

$$\sum_{i=1}^{\log n} d(2^i) = \sum_{i=1}^{\log n} i = \frac{(1+\log n)\log n}{2},$$

and

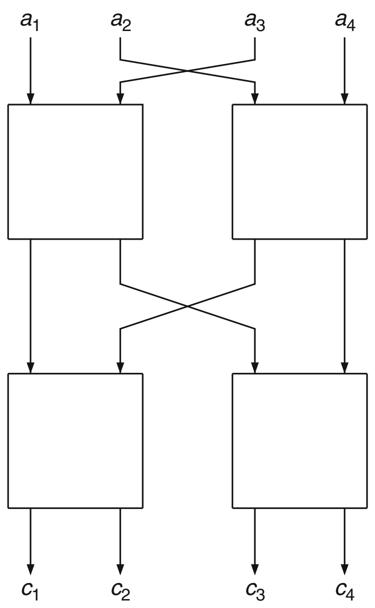
$$\begin{aligned} \sum_{i=1}^{\log n} 2^{(\log n)-i} s(2^i) &= \sum_{i=1}^{\log n} 2^{(\log n)-i} i2^{i-1} \\ &= \frac{n(1+\log n)\log n}{4}, \end{aligned}$$

respectively.



Bitonic input
a_1 a_2 a_3 a_{m-2} a_{m-1} a_m a_{m+1} a_{m+2} a_{m+3} a_{2m-2} a_{2m-1} a_{2m}
$MERGE_m$
$MERGE_m$
c_1 c_2 c_3 c_{m-2} c_{m-1} c_m c_{m+1} c_{m+2} c_{m+3} c_{2m-2} c_{2m-1} c_{2m}
Sorted output

Bitonic Sort. Fig. 2 A circuit for sorting a bitonic sequence of length 2 m



Bitonic Sort. Fig. 3 A circuit for sorting a bitonic sequence of length 4

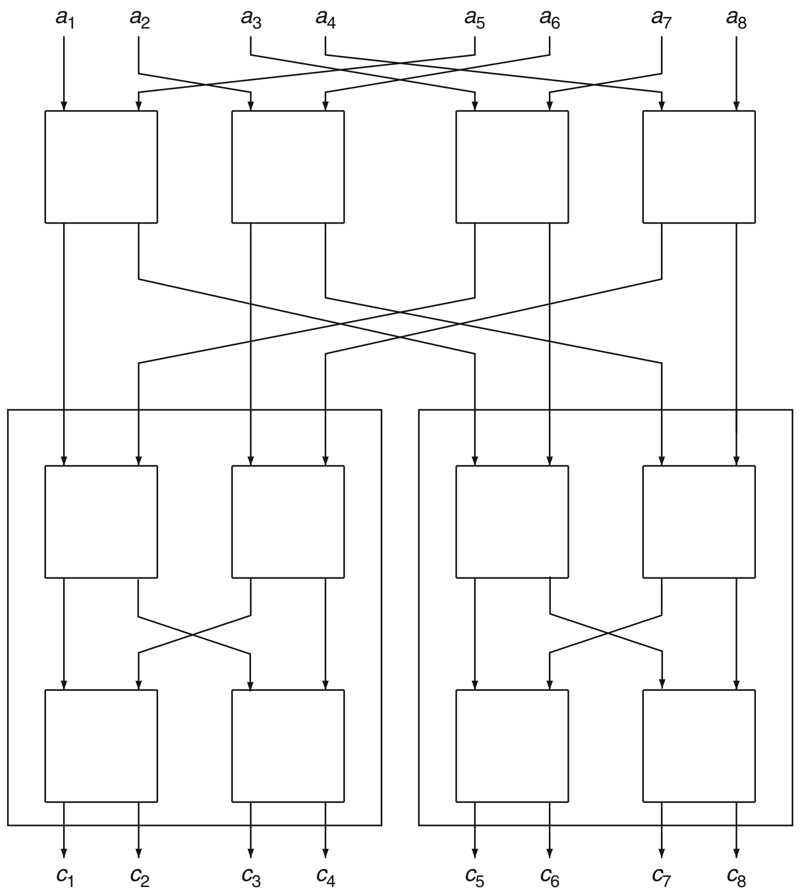
Lower Bounds

In order to evaluate the quality of the *Bitonic Sort* circuit, its depth and size are compared to two types of lower bounds.

A lower bound on the number of comparisons. A lower bound on the number of comparisons required in the worst case by a comparison-based sorting algorithm to sort a sequence of n numbers is derived as follows. All comparison-based sorting algorithms are modeled by a binary tree, each of whose nodes represents a comparison between two input elements. The leaves of the tree represent the $n!$ possible outcomes of the sort. A path from the root of the tree to a leaf is a complete execution of an algorithm on a given input. The length of the longest such path is $\log n!$, which is a quantity on the order of $n \log n$.

Several sequential comparison-based algorithms, such as *Heapsort* and *Mergesort*, for example, achieve this bound, to within a constant multiplicative factor, and are therefore said to be *optimal*. The *Bitonic Sort* circuit always runs in time on the order of $\log^2 n$ and is therefore significantly faster than any sequential algorithm. However, the number of comparators it uses, and therefore the number of comparisons it performs, is on the order of $n \log^2 n$, and consequently, it is not optimal in that sense.

Lower bounds on the depth and size. These lower bounds are specific to circuits. A lower bound on the depth of a sorting circuit for an input sequence of n elements is obtained as follows. Each comparator has



a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8

Bitonic Sort. Fig. 4 A circuit for sorting a bitonic sequence of length 8

two outputs, implying that an input element can reach at most 2^r locations after r rows. Since each element should be able to reach all n output positions, a lower bound on the depth of the circuit is on the order of $\log n$.

A lower bound on the size of a sorting circuit is obtained by observing that the circuit must be able to produce any of the $n!$ possible output sequences for each input sequence. Since each comparator can be in one of two states (swapping or not swapping its two inputs), the total number of configurations in which a circuit with c comparators can be is 2^c, and this needs to be at least equal to $n!$. Thus, a lower bound on the size of a sorting circuit is on the order of $n \log n$.

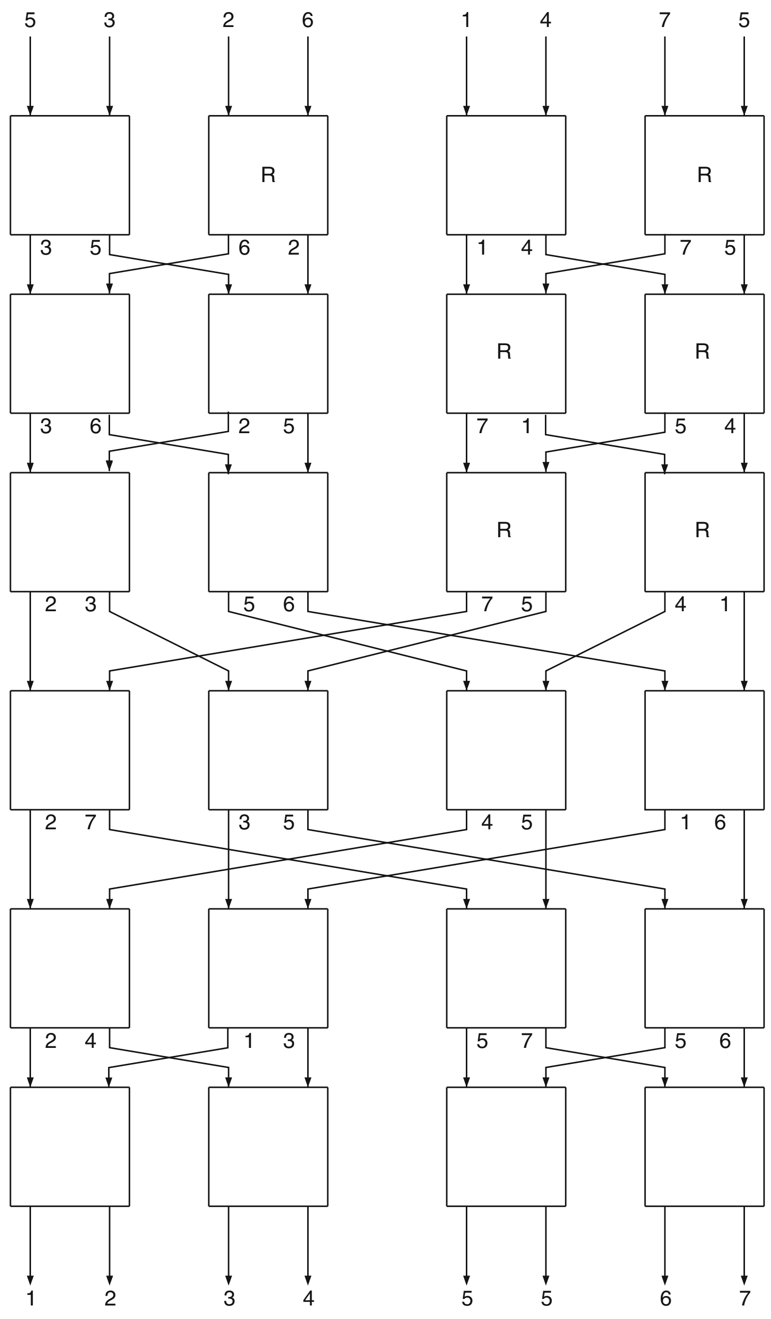
The *Bitonic Sort* circuit exceeds each of the lower bounds on depth and size by a factor of $\log n$, and is therefore not optimal in that sense as well.

There exists a sorting circuit, known as the *AKS* circuit, which has a depth on the order of $\log n$ and a size on the order of $n \log n$. This circuit meets the lower bound on the number of comparisons required to sort, in addition to the lower bounds on depth and size specific to circuits. It is, therefore, theoretically optimal on all counts.

In practice, however, the *AKS* circuit may not be very useful. In addition to its high conceptual complexity, its depth and size expressions are preceded by constants on the order of 10^3. Even if the depth of the *AKS* circuit were as small as $200 \log n$, in order for the latter to be smaller than the depth of the *Bitonic Sort* circuit, namely, $((1+\log n)\log n)/2$, the input sequence will need to have the astronomical length of $n > 2^{399}$, which is greater than the number of atoms that the observable universe is estimated to contain.

Future Directions

An interesting open problem in this area of research is to design a sorting circuit with the elegance, regularity, and simplicity of the *Bitonic Sort* circuit, while at the same



Bitonic Sort. Fig. 5 A circuit for sorting an arbitrary sequence of length 8

time matching the theoretical lower bounds on depth and size as closely as possible.

Related Entries

►AKS Sorting Network
►Bitonic Sorting, Adaptive
►Odd-Even Sorting
►Permutation Circuits
►Sorting

Bibliographic Notes and Further Reading

In 1968, Ken Batcher presented a paper at the AFIPS conference, in which he described two circuits for sorting, namely, the *Odd-Even Sort* and the *Bitonic Sort* circuits [6]. This paper pioneered the study of parallel sorting algorithms and effectively launched the field of parallel algorithm design and analysis. Proofs of correctness of the *Bitonic Sort* algorithm are provided in [3, 6, 8]. Implementations of *Bitonic Sort* on other architectures beside combinational circuits have been proposed, including implementations on a perfect shuffle computer [20], a mesh of processors [16, 21], and on a shared-memory parallel machine [7].

In its simplest form, *Bitonic Sort* assumes that n, the length of the input sequence, is a power of 2. When n is not a power of 2, the sequence can be padded with z zeros such that $n + z$ is the smallest power of 2 larger than n. Alternatively, several variants of *Bitonic Sort* were proposed in the literature that are capable of sorting input sequences of arbitrary length [14, 15, 22].

Combinational circuits for sorting (also known as *sorting networks*) are discussed in [3–5, 8, 11–13, 17]. A lower bound for oblivious merging is derived in [9] and generalized in [23], which demonstrates that the bitonic merger is optimal to within a small constant factor. The *AKS* sorting circuit (whose name derives from the initials of its three inventors) was first described in [1] and then in [2]. Unlike the *Bitonic Sort* circuit that is based on the idea of repeatedly merging increasingly longer subsequences, the *AKS* circuit sorts by repeatedly splitting the original input sequence into increasingly shorter and disjoint subsequences. While theoretically optimal, the circuit suffers from large multiplicative constants in the expressions for its depth and size, making it of little use in practice, as mentioned earlier. A formulation in [18] manages to reduce the constants to a few thousands, still a prohibitive number. Descriptions of the *AKS* circuit appear in [5, 10, 17, 19]. Sequential sorting algorithms, including *Heapsort* and *Mergesort*, are covered in [8, 12].

One property of combinational circuits, not shared by many other parallel models of computation, is their ability to allow several input sequences to be processed simultaneously, in a *pipeline fashion*. This is certainly true of sorting circuits: Once the elements of the first input sequence have traversed the first row and moved on to the second, a new sequence can enter the first row, and so on. If the circuit has depth D, then M sequences can be sorted in $D + M - 1$ time units [5].

Bibliography

1. Ajtai M, Komlós J, Szemerédi E (1983) An $O(n \log n)$ sorting network. In: Proceedings of the ACM symposium on theory of computing, Boston, Massachusetts, pp 1–9
2. Ajtai M, Komlós J, Szemerédi E (1983) Sorting in $c \log n$ parallel steps. Combinatorica 3:1–19
3. Akl SG (1985) Parallel sorting algorithms. Academic Press, Orlando, Florida
4. Akl SG (1989) The design and analysis of parallel algorithms. Prentice-Hall, Englewood Cliffs, New Jersey
5. Akl SG (1997) Parallel computation: models and methods. Prentice Hall, Upper Saddle River, New Jersey
6. Batcher KE (1968) Sorting networks and their applications. In: Proceedings of the AFIPS spring joint computer conference. Atlantic City, New Jersey, pp 307–314. Reprinted in: Wu CL, Feng TS (eds) Interconnection networks for parallel and distributed processing. IEEE Computer Society, 1984, pp 576–583
7. Bilardi G, Nicolau A (1989) Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. SIAM J Comput 18(2):216–228
8. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms, 2nd edn. MIT Press/McGraw-Hill, Cambridge, Massachusetts/New York
9. Floyd RW (1972) Permuting information in idealized two-level storage. In: Miller RE, Thatcher JW (eds) Complexity of computer computations. Plenum Press, New York, pp 105–110
10. Gibbons A, Rytter W (1988) Efficient parallel algorithms. Cambridge University Press, Cambridge, England
11. JáJá J (1992) An introduction to parallel algorithms. Addison-Wesley, Reading, Massachusetts
12. Knuth DE (1998) The art of computer programming, vol 3. Addison-Wesley, Reading, Massachusetts
13. Leighton FT (1992) Introduction to parallel algorithms and architectures. Morgan Kaufmann, San Mateo, California
14. Liszka KJ, Batcher KE (1993) A generalized bitonic sorting network. In: Proceedings of the international conference on parallel processing, vol 1, pp 105–108
15. Nakatani T, Huang ST, Arden BW, Tripathi SK (1989) K-way bitonic sort. IEEE Trans Comput 38(2):283–288
16. Nassimi D, Sahni S (1980) Bitonic sort on a mesh-connected parallel computer. IEEE Trans Comput C-28(1):2–7
17. Parberry I (1987) Parallel complexity theory. Pitman, London
18. Paterson MS (1990) Improved sorting networks with $O(\log N)$ depth. Algorithmica 5:75–92
19. Smith JR (1993) The design and analysis of parallel algorithms. Oxford University Press, Oxford, England
20. Stone HS (1971) Parallel processing with the perfect shuffle. IEEE Trans Comput C-20(2):153–161

21. Thompson CD, Kung HT (1977) Sorting on a mesh-connected parallel computer. Commun ACM 20(4):263–271
22. Wang BF, Chen GH, Hsu CC (1991) Bitonic sort with and arbitrary number of keys. In: Proceedings of the international conference on parallel processing. vol 3, Illinois, pp 58–65
23. Yao AC, Yao FF (1975) Lower bounds for merging networks. J ACM 23(3):566–571

Bitonic Sorting Network

►Bitonic Sort

Bitonic Sorting, Adaptive

Gabriel Zachmann
Clausthal University, Clausthal-Zellerfeld, Germany

Definition

Adaptive bitonic sorting is a sorting algorithm suitable for implementation on EREW parallel architectures. Similar to bitonic sorting, it is based on merging, which is recursively applied to obtain a sorted sequence. In contrast to bitonic sorting, it is data-dependent. Adaptive bitonic merging can be performed in $O\left(\frac{n}{p}\right)$ parallel time, p being the number of processors, and executes only $O(n)$ operations in total. Consequently, adaptive bitonic sorting can be performed in $O\left(\frac{n \log n}{p}\right)$ time, which is optimal. So, one of its advantages is that it executes a factor of $O(\log n)$ less operations than bitonic sorting. Another advantage is that it can be implemented efficiently on modern GPUs.

Discussion

Introduction

This chapter describes a parallel sorting algorithm, *adaptive bitonic sorting* [5], that offers the following benefits:

- It needs only the optimal total number of comparison/exchange operations, $O(n \log n)$.
- The hidden constant in the asymptotic number of operations is less than in other optimal parallel sorting methods.
- It can be implemented in a highly parallel manner on modern architectures, such as a streaming architecture (GPUs), even without any scatter operations, that is, without random access writes.

One of the main differences between "regular" bitonic sorting and adaptive bitonic sorting is that regular bitonic sorting is data-independent, while adaptive bitonic sorting is data-dependent (hence the name).

As a consequence, adaptive bitonic sorting cannot be implemented as a sorting network, but only on architectures that offer some kind of flow control. Nonetheless, it is convenient to derive the method of adaptive bitonic sorting from bitonic sorting.

Sorting networks have a long history in computer science research (see the comprehensive survey [2]). One reason is that sorting networks are a convenient way to describe parallel sorting algorithms on CREW-PRAMs or even EREW-PRAMs (which is also called PRAC for "parallel random access computer").

In the following, let n denote the number of keys to be sorted, and p the number of processors. For the sake of clarity, n will always be assumed to be a power of 2. (In their original paper [5], Bilardi and Nicolau have described how to modify the algorithms such that they can handle arbitrary numbers of keys, but these technical details will be omitted in this article.)

The first to present a sorting network with optimal asymptotic complexity were Ajtai, Komlós, and Szemerédi [1]. Also, Cole [6] presented an optimal parallel merge sort approach for the CREW-PRAM as well as for the EREW-PRAM. However, it has been shown that neither is fast in practice for reasonable numbers of keys [8, 15].

In contrast, adaptive bitonic sorting requires less than $2n \log n$ comparisons in total, independent of the number of processors. On p processors, it can be implemented in $O\left(\frac{n \log n}{p}\right)$ time, for $p \leq \frac{n}{\log n}$.

Even with a small number of processors it is efficient in practice: in its original implementation, the sequential version of the algorithm was at most by a factor 2.5 slower than quicksort (for sequence lengths up to 2^{19}) [5].

Fundamental Properties

One of the fundamental concepts in this context is the notion of a *bitonic sequence*.