Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Sciences 176 (2006) 1321–1337
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Efficient unbalanced merge–sort ☆

Enrico Nardelli [a,b], Guido Proietti [b,c,*]

[a] *Dipartimento di Matematica, Università di Roma "Tor Vergata", Via della Ricerca Scientifica, 00133 Roma, Italy*
[b] *Istituto di Analisi dei Sistemi ed Informatica "A. Ruberti", CNR, Viale Manzoni 30, 00185 Roma, Italy*
[c] *Dipartimento di Informatica, Università dell'Aquila, Via Vetoio, 67010 L'Aquila, Italy*

Received 22 April 2004; received in revised form 24 November 2004; accepted 29 April 2005

Abstract

Sorting algorithms based on successive merging of ordered subsequences are widely used, due to their efficiency and to their intrinsically parallelizable structure. Among them, the merge–sort algorithm emerges indisputably as the most prominent method. In this paper we present a variant of merge–sort that proceeds through arbitrary merges between pairs of quasi-ordered subsequences, no matter which their size may be. We provide a detailed analysis, showing that a set of n elements can be sorted by performing at most $n\lfloor \log n \rfloor$ key comparisons. Our method has the same optimal asymptotic time and space complexity as compared to previous known unbalanced merge–sort algorithms, but experimental results show that it behaves significantly better in practice.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Design of algorithms; Sorting; Experimental analysis; Data structures

☆ This work was partially supported by the Research Project GRID.IT, funded by the Italian Ministry of Education, University and Research.

* Corresponding author. Address: Dipartimento di Informatica, Università dell'Aquila, Via Vetoio, 67010 L'Aquila, Italy. Tel.: +39 0862 433727; fax: +39 0862 433057.
E-mail addresses: nardelli@mat.uniroma2.it (E. Nardelli), proietti@di.univaq.it (G. Proietti).

0020-0255/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2005.04.008

1. Introduction

Let U be a totally ordered universe. Let $S = \{k_1, k_2, \ldots, k_n\}$ be a set of n elements belonging to U, with $k_i \neq k_j$ for $i \neq j$. Sorting S in increasing order asks for finding the unique permutation $\{1, 2, \ldots, n\} \rightarrow \{i_1, i_2, \ldots, i_n\}$ such that

$$k_{i_1} < k_{i_2} < \cdots < k_{i_n}.$$

Arguing about the importance of sorting is somehow redundant, and the interested reader is therefore referred to [6] for a deeper insight on the topic. We here just recall that as far as *internal sorting* is concerned (i.e., where sorting takes place totally in main memory), which is of interest for this paper, the currently fastest deterministic and randomized sorting algorithms run on a word RAM in $O(n \log\log n)$ time [4] and $O(n\sqrt{\log\log n})$ expected time [5], respectively, whereas for a state-of-the-art on *external sorting* we refer the reader to [8].

Among others, the *merge–sort* is a sorting algorithm of primary importance, that goes back to the early history of modern computer science. Merge–sort is based on the *divide-and-conquer* approach: break the original sequence to be sorted into two subsequences of equal size, merge–sort them recursively, and then combine the subsolutions to eventually return the entire sorted sequence. The first merit of merge–sort is clearly its efficiency, since it is well known that its running time is $O(n \log n)$, which is optimal on a classic comparison-based model. Then, it comes its theoretical cleanliness, which allows a natural and straightforward implementation both for linked lists and for arrays. Finally, its structure makes it suitable to parallel implementation.

Merge–sort has its weak points as well, however. First of all, both in the iterative and the recursive implementation, an $O(n)$ additional space is required, i.e., the method is not *in-place*. Moreover, its practical performances are surpassed by other sorting algorithms, whose asymptotic analysis might even be suboptimal (e.g., quick-sort, radix-sort, etc.). Finally, the dividing step requires to be *balanced*, and therefore the intermediate sorted subsequences are obtained as follows: singletons are combined in pairs, then pairs are combined in quadruples, and so on. Merging in an arbitrary order, until only one sorted set remains, is therefore impossible. Actually, this is a severe drawback, since it forbids an unbalanced parallelization of the algorithm, which might be requested, for instance, when input data are scattered among several machines, or when processors having different power are at disposal. In this paper, we exactly address the problem of developing an efficient algorithm implementing such *unbalanced merge–sort*. More precisely, we will present a sorting algorithm which proceeds through arbitrary merges among subsequences, with the relaxation that intermediate subsequence are not totally ordered (although they can be sorted in time proportional to their size). This can be done by maintaining intermediate sets—of arbitrary size—using *binomial search trees*

(BS-trees, for short) [7]. BS-trees enjoy the property of being *quasi-ordered* (i.e., a BS-tree can be sorted in time proportional to its size). Therefore, differently from other possible approaches which might use partially ordered data structures, like those belonging to the huge heap family, BS-trees guarantee the possibility to retrieve the final sorted sequence through successive merging of quasi-ordered subsequences of unrelated size, with the additional potentiality that each subsequence can be sorted in fast linear time, whenever this is needed. Moreover, as we will show, BS-trees are simple and easy-to-implement, and their real performances are close to their expected asymptotic behavior, and this represents a significant benefit in practice.

The initial step towards the implementation of an unbalanced merge–sort algorithm was performed by Brown and Tarjan [2]. More specifically, in [2] the authors developed a fast *merging* algorithm, that is an algorithm which, given two sorted lists P and Q, with $|P| \leqslant |Q|$, returns a list containing the elements of P and Q in sorted linear order. The authors showed that by storing the elements by using *AVL-trees* [1] and by exploiting some ordering properties which are implicit in the structure of AVL-*trees*, it is possible to implement the merging on a pointer machine in $O\left(|P| \log \frac{|Q|}{|P|}\right)$ time, which is optimal [2]. In this way, an $O(n \log n)$ time unbalanced merge–sort algorithm can be obtained starting from n singletons and proceeding through $n - 1$ merging in any arbitrary order.

Although this method is asymptotically optimal, unfortunately it has two drawbacks in real-life applications: first, it makes use of $O(|P| + |Q|)$ additional space at each merging; second, it has quite big cost factors not visible in the asymptotic measure. This is essentially due to the fact that since AVL-trees are used, one needs to maintain at each merging step totally ordered sets, although this is not strictly necessary. Hence, relaxing the total order property of the intermediate sets could result in an easier and cheaper sorting method. In fact, the quasi-ordered structure of BS-trees allows to overcome these drawbacks arising with AVL-trees. Notice that this peculiar feature of the BS-trees have already been used in the past to solve efficiently a variant of the classic set-union problem, named *set-union and intersection problem* [3], in which unions are performed on two identical collections of elements, and an intersection operation among sets from the two collections is additionally managed.

In this paper, we will show that by using BS-trees for maintaining sets, we can execute unbalanced merge–sort optimally in an asymptotic time sense, and better than the method presented in [2] in practice. Besides that, our data structure is simple to implement and, as we will show, the maximum number of key comparisons is kept equal to that of the classic merge–sort, since, as for the traditional method, it is maintained the invariant that exactly 2^i sorted sequences of size $n/2^i,\ i = 1, \ldots, \lfloor \log n \rfloor$, are merged. Thus, as shown by our experiments, when the sorting has to be performed on a multiprocessor system

with a non-uniform distribution of the workload among the processors, through BS-sorting we can still maintain at minimum the total number of comparisons, while for the classic merge–sort, the larger the unbalancedness of the workloads, the higher the additional number of comparisons which are needed at the final stage to merge the sorted subsequences.

The paper is organized as follows: in Section 2 we recall the basic properties of BS-trees. Section 3 contains a detailed worst and average case analysis of the core operation, that is the merging of two BS-trees, while Section 4 is devoted to the analysis of the unbalanced merge–sort algorithm. In Section 5, experimental results showing the effectiveness of our method are given, while finally, Section 6 presents conclusions and possible future developments.

2. The BS-tree data structure

We start by giving the definition of BS-trees. In the following, we assume that with each node of a BS-tree, a unique *key* from a totally ordered universe U is associated.

Definition 2.1. For any $h \geqslant 0$, we define the class of BS-trees of height h, say $\mathcal{Q}_h$, as follows:

(a) if $h = 0$, then any $Q \in \mathcal{Q}_0$ consists of a single node;
(b) if $h > 0$, then any $Q \in \mathcal{Q}_h$ consists of: (i) a particular node, called the root; (ii) a left subtree of the root, consisting of a complete binary search tree of height $h - 1$, whose element keys are smaller than the root key; (iii) a right subtree of the root, that either is empty or consists of a BS-tree of height $k < h$, whose set of element keys has no ordering relation with the root key.

Let $|Q|$ indicate the number of nodes of a BS-tree Q. The following can be proved:

Proposition 2.1. *Let $Q \in \mathcal{Q}_h$. Then, for any $h \geqslant 0$, $2^h \leqslant |Q| \leqslant 2^{h+1} - 1$.*

Proof. By induction on h. If $h = 0$, then by definition $|Q| = 1$ and the proposition is true. Suppose the proposition is true for each $1 \leqslant h \leqslant k - 1$, and let $Q \in \mathcal{Q}_k$. Then, we have to prove that $2^k \leqslant |Q| \leqslant 2^{k+1} - 1$. But the smallest allowed BS-tree in $\mathcal{Q}_k$, by definition, is made up by a root and a complete binary left subtree of height $k - 1$, and then $|Q| \geqslant 1 + (2^k - 1) = 2^k$. On the other hand, again by definition, the greatest allowed BS-tree in $\mathcal{Q}_k$ is made up by a root, a complete binary left subtree of height $k - 1$ and the greatest allowed BS-tree in $\mathcal{Q}_{k-1}$. Then, from the inductive hypothesis, $|Q| \leqslant 2^k + 2^k - 1 = 2^{k+1} - 1$. □

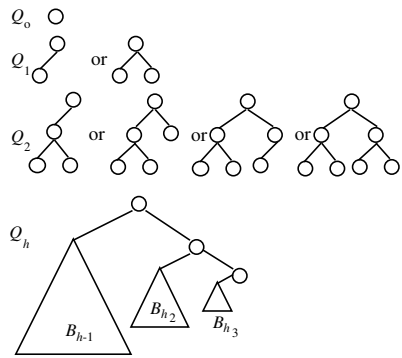

Fig. 1. Some examples of BS-trees.

Proposition 2.2. *Let $Q \in \mathcal{Q}_h$. Then, for any $h \geqslant 0$, the height of Q is* $O(\log |Q|)$.

Proof. It immediately follows from Proposition 2.1 and from the fact that the height of $Q \in \mathcal{Q}_h$ is exactly h by definition. □

We define the *border* of a BS-tree Q as the sequence of nodes on the rightmost path of Q. From Definition 2.1, it is easy to see that if x precedes y on the border, then the height of the left subtree of x is larger than the height of the left subtree of y. In the following, a border node and its left subtree (if any) of height k will be called a $(k+1)$-*component* of the BS-tree (by extension, a border node having an empty left subtree will be named a 0-component). Notice that a k-component contains exactly 2^k nodes. Fig. 1 shows some examples of small BS-trees and the general shape of the data structure (triangles represent complete binary trees).

As for binomial heaps [9], there is a direct relation between the components appearing in a BS-tree storing a set of elements A, and the binary representation of the size of A. More precisely, if $|A| = \sum_{i \geqslant 0} b_i 2^i$, with $b_i \in \{0, 1\}$ then we have that the ith component of the BS-tree exists if and only if $b_i = 1$. Fig. 2 depicts a sample BS-tree for a set A of 13 elements. A detailed description of how a BS-tree is built will be presented in the next section.

3. Merging two BS-trees

In this section we provide a detailed analysis of the merging operation between two BS-trees. This is indeed the fundamental step of the sorting algorithm. As usual, in the following we assume to obtain in constant time the reference to the BS-trees to be merged.

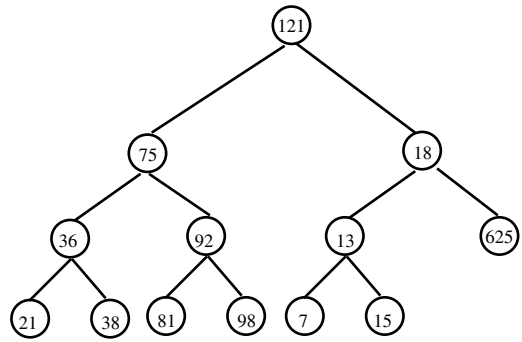

Fig. 2. An example of BS-tree of height 3, storing the set $A = \{21, 36, 38, 75, 81, 92, 98, 121, 7, 13, 15, 18, 625\}$; since $|A| = (13)_{10} = (1101)_2$, to represent A it is needed a BS-tree in $\mathcal{Q}_3$, made up of a 3-, a 2- and a 0-component.

3.1. Worst case analysis

Let P and Q be two BS-trees of height p and q, respectively. Suppose a `merge`(P, Q) operation is required, which asks for returning a single BS-tree containing all the elements in P and Q. Firstly, p and q are compared: if $p \geqslant q$, then Q is *linked* to P, otherwise P is linked to Q. We now describe in detail what it is the meaning of linking two BS-trees.

First, consider the special case in which $|P| = |Q| = 2^k$, that is, both BS-trees consist of a single k-component. In this case, the resulting BS-tree consists of a single $(k+1)$-component, built up through a standard linear merging of P and Q. We refer to this operation as a *coupling* [9] of order k, or, more simply, a k-coupling. In Fig. 3, an example of a 2-coupling, (i.e., with $|P| = |Q| = 4$) is provided.

It turns out the following:

Lemma 3.1. *Two BS-trees, each of which consists of a single k-component can be coupled in $O(2^{k+1})$ time, by performing in the worst and in the average case $2^{k+1} - 1$ and $\frac{2^{2k+1}}{2^{k+1}}$ key comparisons, respectively.*

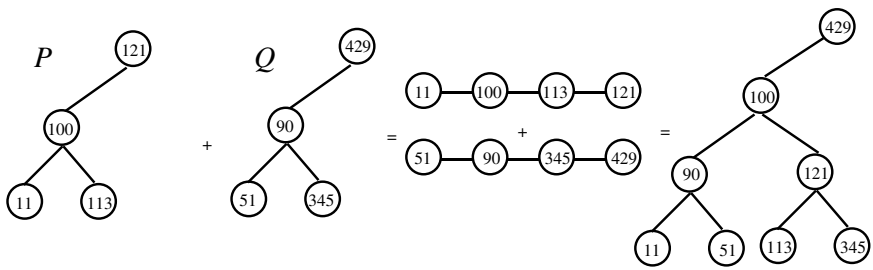

Fig. 3. Performing a coupling on two BS-tree of 4 elements.

Proof. We initially dismantle the two BS-tree, by means of a preorder visit. We thus obtain two lists of 2^k elements sorted in increasing order. Then we merge them with the classical linear merging, which requires $2^{k+1} - 1$ key comparisons in the worst case. As far as the average case is concerned, it is easily seen that the total number of comparisons needed to create the ordered list is $2^{k+1} - \sigma$, where σ is the number of elements remaining in the non-empty list after all the elements of the other list have been exhausted. For the general case in which the two lists to be merged have length n and m, respectively, Knuth [6] have derived $\sigma = n/(m+1) + m/(n+1)$, which, for our special case $n = m = 2^k$ returns an average number of comparisons of

$$2^{k+1} - \frac{2^{k+1}}{2^k + 1} = \frac{2^{2k+1}}{2^k + 1}.$$

Afterwards, we rebuild the BS-tree of height $k + 1$, consisting of a single $(k + 1)$-component. Trivially, all these operations can be performed in $O(2^{k+1})$ time. □

For treating the general case where $|P|$ and $|Q|$ are arbitrary values, it is convenient to use an analogy with the ordinary scheme for the binary addition of $|P|$ and $|Q|$. W.l.o.g., suppose that P and Q have height p and q, respectively, with $p \leqslant q$. The merging proceeds from lower order components of the BS-trees to higher order components. At the ith step of the algorithm (corresponding to a merging of i-components), there are three operands into play: two of the operands are the i-components of P and Q, respectively, while the third operand is a *carry*, namely a (possibly empty) i-component which can be generated by a merging of $(i-1)$-components, in a way similar to the classic binary addition, as explained in more detail in the following.

At the beginning (i.e., for $i = 0$), the carry is empty, and we start by merging the two 0-components of P and Q. If both the components are empty, an empty 0-component for the result is generated, and an empty carry is propagated at the next stage. If exactly one 0-component is nonempty, it constitutes the 0-component of the result, and the carry is empty. Finally, if both the two 0-components are nonempty, they are coupled according to the procedure described earlier, in order to constitute the carry at the next stage, while the 0-component of the result is empty.

At the ith step of the algorithm, the following cases are possible: (i) If all three operands are empty, the i-component of the resulting BS-tree is empty, as it is the carry propagated to the next step; (ii) If exactly one operand is nonempty, it constitutes the i-component of the result, and the carry is empty; (iii) If two operands are nonempty, they are coupled according to the procedure described earlier, in order to constitute the $(i + 1)$th carry; the i-component of the result is empty; (iv) Finally, when all three operands are nonempty, one

of them, for example the one belonging to Q, will constitute the i-component of the resulting BS-tree, and the remaining two operands are coupled in order to form the $(i+1)$th carry.

The procedure stops when P has been exhausted and no carry is propagated any further. In Fig. 4, the algorithm is presented in the case $|P| = 5$ and $|Q| = 7$.

Theorem 3.1. *Given two BS-trees P and Q of height p and q, respectively, with $p \leqslant q$, let t be the order of the last coupling happening during the merging of P and Q. Then, it is possible to execute a* `merge`(P,Q) *operation with at most $2^{t+2} - t - 3$ key comparisons and in $O(p + 2^{t+2})$ time in the worst case.*

Proof. Remember that we perform a `merge`(P, Q) operation by firstly taking into account the size of the involved sets, and then by stopping the procedure as soon as P has been exhausted and no carry is propagated any further. From this, it follows that the border of P must be entirely scanned, thus spending $O(p)$ time. To such a cost, we have to add the time spent in performing the couplings. Since the last coupling has order t, it follows that we have to perform $O(t)$ couplings (of increasing size) of the involved components of the two BS-trees. From Lemma 3.1, a coupling between two i-components requires at most $2^{i+1} - 1$ key comparisons and $O(2^{i+1})$ running time, and therefore we have that `merge`(P, Q) requires at most

$$\sum_{i=0}^{t} (2^{i+1} - 1) = 2^{t+2} - t - 3$$

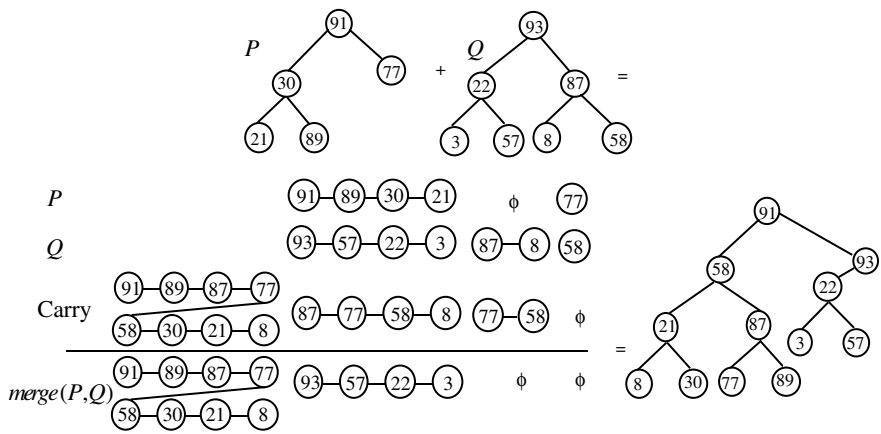

Fig. 4. Performing a `merge`(P, Q) operation.

key comparisons, and in the worst case has a time complexity

$$\mathrm{O}\left(p + \sum_{i=0}^{t}(2^{i+1})\right) = \mathrm{O}(p + 2^{t+2}). \qquad \square$$

3.2. Average case analysis

Theorem 3.1 shows that, in the worst case, merging two BS-trees can be expensive, since t can be as large as q, and therefore the operation can cost as much as the size of Q. However, as we show in this section, the expected cost of a merging operation is optimal. Let $E_{\mathrm{merge}}(k)$, $k \geqslant 0$ integer, denote the probability a $\mathtt{merge}(P, Q)$ operation is executed in $\Theta(p + 2^{k+1})$ time in the worst case. We start by proving the following:

Theorem 3.2. *Given two BS-trees P and Q of height p and q, respectively with $p \leqslant q$, we have that*

$$E_{\mathrm{merge}}(k) = \begin{cases} \frac{1}{2} \cdot \left(\frac{3}{4}\right)^{p} & k = 0 \\ \frac{1}{8} \cdot \left(\frac{3}{4}\right)^{p-k-1} \cdot \frac{2^{k}-1}{2^{k+1}} & 1 \leqslant k \leqslant p-1 \\ 0 & k = p \\ \frac{1}{2} \cdot \frac{2^{p+1}+2^{p}-1}{2^{k+1}} & p+1 \leqslant k \leqslant q-1 \\ 0 & k = q \\ \frac{2^{p+1}+2^{p}-1}{2^{q+1}} & k = q+1 \\ 0 & k > q+1. \end{cases}$$

Proof. Let $E_{\mathrm{carry}}(k)$ denote the probability the leftmost non-empty carry happens in position k, $1 \leqslant k \leqslant q+1$. We extend the notation to the case $k = 0$, to denote the fact that no carry is propagated and no coupling happens. Since when the leftmost non-empty carry appears in position k, then the last coupling takes place between two $(k-1)$-components, from Theorem 3.1 it follows that $E_{\mathrm{carry}}(k) = E_{\mathrm{merge}}(k)$. We therefore focus our attention on $E_{\mathrm{carry}}(k)$.

Looking at the merging as to the binary addition, we have to find the probabilities that the leftmost 1 of the carry either does not appear or it appears in position k, with $k \geqslant 1$. We can represent the merging with the following schema, where the binary representations of $|Q|$ and $|P|$ are shown, along with the carry bits from r_0 to r_{q+1}:

carry :	r_{q+1}	r_q	r_{q-1}	…	r_{p+1}	r_p	$r_{p-1}\ldots$	r_1	0
$\lvert Q\rvert$:	0	1	a_{q-1}	…	a_{p+1}	a_p	$a_{p-1}\ldots$	a_1	a_0
$\lvert P\rvert$:	0	0	0	…	0	1	$b_{p-1}\ldots$	b_1	b_0

The proof is by cases on k.

$\boldsymbol{k = 0}$: To have no carry, that is

$$r_i = 0, \quad 1 \leqslant i \leqslant q+1,$$

it must be

1. $a_i + b_i < 2$ for $i = 0, 1, \ldots, p-1$, so that $r_1 = r_2 = \cdots = r_p = 0$; this happens with probability $(\frac{3}{4})^p$;
2. $r_{p+1} = 0$, since this necessarily implies that $r_{p+2} = r_{p+3} = \cdots = r_{q+1} = 0$; this happens if and only if $a_p = 0$, that is with probability 1/2.

From this it follows that

$$E_{\text{carry}}(0) = \frac{1}{2} \cdot \left(\frac{3}{4}\right)^p.$$

$\boldsymbol{1 \leqslant k \leqslant p-1}$: To have the leftmost 1 of the carry appearing in position k, with $1 \leqslant k \leqslant p-1$, the following has to happen:

$$(r_i = 0, k+1 \leqslant i \leqslant q+1) \wedge (r_k = 1)$$

and then it must be:

1. $(a_{k-1}a_{k-2}\ldots a_0) + (b_{k-1}b_{k-2}\ldots b_0) > 2^k$, so that $r_k = 1$; since

   $$0 \leqslant (a_{k-1}a_{k-2}\ldots a_0) + (b_{k-1}b_{k-2}\ldots b_0) \leqslant 2^{k+1} - 2,$$

   we have that $(2^{k+1} - 1)$ values of the sum are possible. Let $x_i = i - 1$ be the ith feasible value, $1 \leqslant i \leqslant 2^{k+1} - 1$; it is easy to see that x_i and $x_{2^{k+1}-i}, 1 \leqslant i \leqslant 2^k$, can both be produced in i different ways, while x_{2^k} can be produced in 2^k different ways; it follows that the probability that $x_i > 2^k$ is

   $$\frac{\sum_{i=1}^{2^k-1} i}{2\sum_{i=1}^{2^k-1} i + 2^k} = \frac{2^k - 1}{2^{k+1}}.$$

2. $a_k + b_k = 0$, so that $r_{k+1} = 0$, and this happens with probability 1/4;
3. $a_i + b_i < 2$ for $i = k+1, \ldots, p-1$, so that $r_{k+2} = \cdots = r_p = 0$, and this happens with probability $(\frac{3}{4})^{p-k-1}$;
4. $r_{p+1} = 0$, and this happens if and only if $a_p = 0$, that is with probability 1/2.

From the above analysis, it follows that for $1 \leqslant k \leqslant p-1$

$$E_{\text{carry}}(k) = \frac{1}{8} \cdot \left(\frac{3}{4}\right)^{p-k-1} \cdot \frac{2^{k-1}}{2^{k+1}}.$$

$\boldsymbol{k = p}$: To have the leftmost 1 of the carry appearing in position p, the following has to happen:

$$(r_i = 0, p \leqslant i \leqslant q+1) \wedge (r_p = 1),$$

but this is impossible, since being $b_p = 1$, from the fact that $r_p = 1$, it should follow that $r_{p+1} = 1$. Then, $E_{\text{carry}}(p) = 0$.

$\boldsymbol{p+1 \leqslant k \leqslant q-1}$: To have the leftmost 1 of the carry appearing in position k, with $p+1 \leqslant k \leqslant q-1$, the following has to happen:

$$(r_i = 0, k+1 \leqslant i \leqslant q+1) \wedge (r_k = 1)$$

and then it must be:

1. $(a_{k-1}a_{k-2}\ldots a_0) + (1b_{p-1}\ldots b_0) \geqslant 2^k$, so that $r_k = 1$; since

   $$2^p \leqslant (a_{k-1}a_{k-2}\ldots a_0) + (1b_{p-1}\ldots b_0) \leqslant (2^k - 1) + (2^{p+1} - 1),$$

   we have that $(2^k + 2^p - 1)$ values of the sum are possible. Let $x_i = 2^p + (i-1)$ be the ith feasible value, $i = 1, \ldots, (2^k + 2^p - 1)$; then x_i can be produced in i different ways, if $1 \leqslant i \leqslant 2^p - 1$, or in $(2^k + 2^p - i)$ different ways, if $2^k + 1 \leqslant i \leqslant 2^k + 2^p - 1$, while in all the other cases it can be produced in 2^p different ways; after some calculations, it follows that the probability that $x_i \geqslant 2^k$ is

   $$\frac{\sum_{i=1}^{2p-1} i + 2^{2p}}{2\sum_{i=1}^{2p-1} i + 2^p(2^k - 2^p + 1)} = \frac{2^{p+1} + 2^p - 1}{2^{k+1}}.$$

2. $a_k = 0$, so that $r_{k+1} = 0$, and this happens with probability 1/2. From this it follows that for $p+1 \leqslant k \leqslant q-1$

   $$E_{\text{carry}}(k) = \frac{1}{2} \cdot \frac{2^{p+1} + 2^p - 1}{2^{k+1}}.$$

$\boldsymbol{k = q}$: To have the leftmost 1 of the carry appearing in position q, the following has to happen:

$$(r_{q+1} = 0) \wedge (r_q = 1),$$

but this is impossible, since being $a_q = 1$, from the fact that $r_q = 1$, it should follow $r_{q+1} = 1$. Then, $E_{\text{carry}}(q) = 0$.

$\boldsymbol{k = q+1}$: To have the leftmost 1 of the carry appearing in position $q+1$, it must be

$$(1a_{q-1}\ldots a_0) + (1b_{p-1}\ldots b_0) \geqslant 2^{q+1}$$

and since

$$2^p + 2^q \leqslant (1a_{q-1} \dots a_0) + (1b_{q-1} \dots b_0) \leqslant (2^{q+1} - 1) + (2^{p+1} - 1),$$

we have that $(2^q + 2^p - 1)$ values of the sum are possible. Let $x_i = 2^p + (i-1)$ be the ith feasible value, $i = 1, \dots, (2^q + 2^p - 1)$; then x_i can be produced in i different ways, if $1 \leqslant i \leqslant 2^p - 1$, or in $(2^q + 2^p - i)$ different ways if $2^q + 1 \leqslant i \leqslant 2^q + 2^p - 1$, while in all the other cases it can be produced in 2^p different ways; after some calculations, it follows that the probability that $x_i \geqslant 2^{q+1}$ is

$$\frac{\sum_{i=1}^{2^p-1} i + 2^{2p}}{2\sum_{i=1}^{2^p-1} i + 2^p(2^q - 2^p + 1)} = \frac{2^{p+1} + 2^p - 1}{2^{q+1}}.$$

$\boldsymbol{k > q+1}$: Finally, the leftmost carry cannot appear after the $(q+1)$th position, and therefore $E_{\text{carry}}(k) = 0$ for $k > q+1$. □

From the above theorem it immediately descends the following result:

Corollary 3.1. *Given two BS-trees P and Q, with $|P| \leqslant |Q|$, the expected cost $\overline{T}$ (merge(P,Q)) of their merging is $\mathrm{O}\left(|P| \log \frac{|Q|}{|P|}\right)$ time.*

Proof. Let p and q denote the height of P and Q, respectively. Then, we have

$$\begin{aligned}
\bar{T}(\texttt{merge}(P,Q)) &= \mathrm{O}\left(p + \sum_{k=0}^{q+1} E_{\text{merge}}(k) \cdot 2^{k+1}\right) \\
&= \mathrm{O}\left(p + \left(\frac{3}{4}\right)^p + \frac{1}{4}\sum_{k=1}^{p-1}\left(\frac{3}{4}\right)^{p-k-1}(2^k - 1)\right. \\
&\quad \left. + \sum_{k=p+1}^{q-1}(2^{p+1} + 2^p - 1) + 2(2^{p+1} + 2^p - 1)\right) \\
&= \mathrm{O}\left(p + \left(\frac{3}{4}\right)^p + 2^p \cdot \left(\frac{3}{8}\right)^p + 2^{p+1}(q-p)\right) \\
&= \mathrm{O}\left(|P| \log \frac{|Q|}{|P|}\right). \qquad \square
\end{aligned}$$

4. The sorting algorithm

Let $S = \{k_1, k_2, \dots, k_n\}$ be a set of n elements, belonging to a totally ordered universe, with $k_i \neq k_j$ for $i \neq j$. To sort S, we initially associate with each k_i a

BS-tree, say Q_i. Then, the algorithm proceeds by extracting at each step an arbitrary couple of BS-trees, which are then merged into a single BS-tree (which replaces the merged ones). Thus, $n-1$ merging operations are performed (between BS-trees of unrelated sizes), until a single BS-tree Q remains. Such BS-tree consists of a set of k components, whose size depends on the binary representation of n. More precisely, if $n = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_k}$, with $0 \leqslant i_1 < i_2 < \cdots < i_k$, then $Q \in \mathcal{Q}_k$ consists of the union of i_j-components, $j = 1, \ldots, k$. The final step will therefore sort Q, by means of successive (sorted) merges of the constituting components, starting from the smallest ones.

Next theorem provides the running time analysis for the above sketched *BS-sorting* algorithm:

Theorem 4.1. *A set* $S = \{k_1, k_2, \ldots, k_n\}$ *of* $n = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_k}$ *distinct elements, with* $0 \leqslant i_1 < i_2 < \cdots < i_k$, *can be sorted through BS-sorting in* $O(n \log n)$ *time, by performing at most* $1 - 2^{i_1} + \sum_{j=1}^{k} 2^{i_j}(k - j + i_j)$ *key comparisons.*

Proof. Let Q be the BS-tree obtained as a consequence of the $n-1$ successive merging operations described above. We start by analyzing the number of key comparisons required to build Q.

Let us focus our attention on any i_j-component of Q. We now show that, whichever the sequence of merging operations during the sorting may be, such a component gave rise to a set of 2^h couplings of $(i_j - h - 1)$-components, $h = 0, 1, \ldots i_j - 1$. This can be easily proved by induction on h. Indeed, there will be a single coupling of two $(i_j - 1)$-components to produce the i_j-component constituting Q, and thus the claim is true for $h = 0$. Suppose now that there are exactly 2^t couplings of $(i_j - t - 1)$-components. Of course, each of these components takes part in only one $(i_j - t - 1)$-coupling. Moreover, it is created as a consequence of a $(i_j - t - 2)$-coupling. Thus, it follows that to each $(i_j - t - 1)$-coupling, correspond exactly 2 couplings of $(i_j - t - 2)$-components, and from this it follows that we have exactly 2^{t+1} couplings of $(i_j - t - 2)$-components. Therefore, from Lemma 3.1, we have that the number of key comparisons is at most

$$\sum_{h=0}^{i_j-1} 2^h \left(2^{i_j-h} - 1\right) = \sum_{h=0}^{i_j-1} \left(2^{i_j} - 2^h\right) = 2^{i_j}\left(i_j - 1\right) + 1.$$

Then, the number of key comparisons to build Q is at most

$$k + \sum_{j=1}^{k} 2^{i_j}(i_j - 1). \tag{1}$$

We now turn our attention to the number of key comparisons for the last phase of the algorithm, that is the sorting of Q. Let Q^{i_j} denote the i_j-component of Q.

As sketched above, we proceed in the following way: at the first step, we consider the smallest two components, i.e., Q^{i_1} and Q^{i_2}. We visit them in-order, by producing two sorted lists (of size 2^{i_1} and 2^{i_2}, respectively) which are then melded through the standard sorted merging algorithm. This costs, in the worst case, $2^{i_1} + 2^{i_2} - 1$ key comparisons. At the next step, the obtained sorted list is then melded with Q^{i_3}, and so on, up to the last merging, involving Q^{i_k}, which eventually returns the output sequence (i.e., the set S in sorted order).

From the above algorithm, it is therefore clear that the number of key comparisons for the last phase of the BS-sorting is at most

$$\sum_{h=2}^{k}\sum_{j=1}^{h}(2^{i_j}-1) = \sum_{j=1}^{k}[2^{i_j}(k-j+1)] - (2^{i_1}+k-1). \tag{2}$$

Therefore, from (1) and (2), the total number of key comparisons of the BS-sorting is at most

$$1 - 2^{i_1} + \sum_{j=1}^{k} 2^{i_j}(k-j+i_j). \tag{3}$$

It is easy to see that in the worst case, $n = 2^m - 1$, and the number of key comparisons is at most $n\lfloor \log n \rfloor$, while in the best case $n = 2^m$, and the number of key comparisons is at most $n(\log n - 1) + 1$. Therefore, the number of key comparisons is bounded by $n\lfloor \log n \rfloor$. Clearly, key comparison is the dominant operation, from which it follows that the running time of the BS-sorting is $O(n \log n)$. □

5. Experimental results

Our sorting algorithm has been implemented on a PC with a 500 MHz Pentium III processor and 128 MB of RAM. To verify the efficiency of our method, we have tested it into two different scenarios. In the former, we have assumed a classic framework in which a single-processor machine is used, and we have compared our method as opposed to the algorithm provided by Brown and Tarjan [2]. In the latter, we have simulated a multiprocessor machine, and we have compared our method as opposed to the classic merge–sort algorithm.

5.1. Single-processor machine

In the first setting, the experiments have been conducted in the following way: for $n = 10^i$, $i = 1, \ldots, 5$, we randomly generated 100 input sequences, and then we computed both the average number of comparisons and the

Table 1
Number of comparisons and running time for the algorithm by Brown and Tarjan (BT), as opposed to our proposed one (NP)

Input size	Number of comparisons		CPU time	
	BT	NP	BT	NP
10	10.96	26.05	0.0037	0.00001
10^2	444.25	564.03	0.0963	0.0001
10^3	9664.25	9726.52	1.5399	0.0011
10^4	161,304.7	123,685.6	21.098	0.032
10^5	3,179,202	1,566,551	399.256	0.3366

average CPU time (in seconds) for the two approaches. Table 1 shows the results, which confirm the theoretical analysis. Indeed, as far as our approach is concerned, the number of comparisons is dominated by the merging procedure, which in its turn is performed through successive couplings. As shown in Theorem 4.1, the total number of couplings is constant, once n is fixed, and moreover, the expected number of comparisons at each coupling of k-components is just at most 2 units less than in the worst case [6], where $2^{k+1} - 1$ comparisons are required (see Lemma 3.1). Thus, in our approach the worst and the average case tend to coincide, and both admit a unitary coefficient (in terms of number of comparisons). Concerning the algorithm of Brown and Tarjan (*BT algorithm*, for short), we observe that in this case the multiplicative factor tends to 2, since at each merging of a pair of AVL-trees, it turns out that comparisons are performed twice, once when descending and the other one when climbing up the resulting tree. Notice also that from the running time point of view, our algorithm clearly outperforms the other one. This is partially explained from the fact that the BT algorithm is implemented in a recursive fashion. Moreover, apart from the comparisons which are needed to establish the relative order between elements, the BT algorithm also needs several additional expensive operations to correctly manage the underlying AVL-trees.

5.2. Multiprocessor machine

In the second setting, we have adapted our method to perform the so-called *parallel internal sorting* (PIS). In general, PIS works on multi-processors systems, and is composed of two phases: the *local sort* (which runs on subsequences spread among different, parallel processors sharing a common memory), and the *final merge*, in which the sorted subsequences are collected, and a central-possibly more powerful-processor proceeds to the final merging (notice that the merging phase can be pipelined, through a hierarchy of processors, but we do not investigate this more complicated architecture). Following the cost model suggested in the paper by Taniar and Rahayu [8], we assume

that the workload of the various processors is modelled by a non-uniform distribution law, which characterizes the degree of data skewness. Our method is compared to the classic merge–sort, under the assumption that (1) the merge–sort runs the local sort in the classic, recursive way, while as far as the final merge is concerned, this is executed through a sequence of linear merging between pairs of sorted subsequences chosen in a random way (this simulates the asynchronicity of the parallel processors, which complete their job at an unpredictable time); (2) the BS-sorting is executed through arbitrary BS-merging, both during the local sort and the final merge.

Experiments have been executed for different numbers of multiprocessors. More precisely, for an input sequence of size $2^k - 2,\ k = 10, \ldots, 19$, we have assumed to work on a machine with $k - 1$ processors, where the ith processor, $i = 1, \ldots, k - 1$, handled a subsequence of size 2^i. Notice that these sizes have been chosen to make the comparisons between the two methods as fair as possible: indeed, differently from our method, the classic merge–sort takes advantage from the fact that the input size is a power of 2. For each considered number of parallel processors, we randomly generated 100 input sequences, and then we computed the average number of comparisons, along with the corresponding average and maximum percent saving. The results are summarized in Table 2. By looking at the results, it emerges that our method is always more competitive than traditional merge–sort. Moreover, the longer is the size of the input sequence, the larger is the improvement. This depends on the fact that during the final phase, the merge–sort algorithm proceeds by merging pairs of sequences of possibly very different size, and since the average cost of each merging is dominated by the length of the longest sequence, this eventually generates a large number of comparisons. Hence, the more unbalanced are the sequences to merge, the larger is the cost of the final phase. On the

Table 2
Number of comparisons for the merge–sort algorithm (MS), as opposed to our proposed one (NP), along with the corresponding maximum and average percent saving

Input size	Number of comparisons		% Saving	
	MS	NP	Max (%)	Avg (%)
2^{10}–2	11,420	9616	21.9	13.8
2^{11}–2	23,547	20,480	23.1	15.0
2^{12}–2	52,219	45,056	24.1	15.9
2^{13}–2	114,682	98,304	25.0	16.7
2^{14}–2	249,850	212,992	25.7	17.3
2^{15}–2	540,665	458,752	26.3	17.9
2^{16}–2	1,163,257	983,040	26.8	18.3
2^{17}–2	2,490,360	2,097,152	27.3	18.7
2^{18}–2	5,308,408	4,456,448	27.7	19.1
2^{19}–2	11,272,183	9,437,184	28.0	19.4

contrary, our method is basically not influenced by the size of the merged sequences. Observe also that in the extreme case, our method can save up to almost a 30% of comparisons.

6. Conclusions

In this paper we have proposed a new unbalanced merge–sort algorithm, based on BS-trees, whose overall performance is in practice significantly better than the currently best known sorting method of the same type.

Concerning future work, we plan to compare our algorithm with other sorting algorithms. Moreover, we expect that several interesting applications other than sorting can take advantage from using BS-trees (for example, dictionaries in which a low number of searching and deleting operations are requested, or heaps in which searching must be supported).

References

[1] G.M. Adel'son-Vel'skii, E.M. Landis, An algorithm for the organization of the information, Dokl. Akad. Nauk. USSR 146 (1962) 263–266.
[2] M.R. Brown, R.E. Tarjan, A fast merging algorithm, J. ACM 26 (2) (1979) 211–226.
[3] C. Gaibisso, E. Nardelli, G. Proietti, Intersection reporting on two collections of disjoint sets, Inform. Sci. 114 (1–4) (1999) 41–52.
[4] Y. Han, Deterministic sorting in $O(n \log\log n)$ time and linear space, in: Proceedings of the 34th ACM Symposium on Theory of Computing (STOC'02), ACM Press, NY, pp. 602–608.
[5] Y. Han, M. Thorup, Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space, in: Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS'02), IEEE Computer Society, pp. 135–144.
[6] D.E. Knuth, The Art of Computer Programming, Sorting and Searching, vol. 3, Addison-Wesley, Reading, MA, 1973.
[7] E. Nardelli, G. Proietti, Binomial search trees, unpublished manuscript.
[8] D. Taniar, J.W. Rahayu, Parallel database sorting, Inform. Sci. 146 (1–4) (2002) 171–219.
[9] J. Vuillemin, A data structure for manipulating priority queues, Comm. ACM 21 (1978) 309–314.