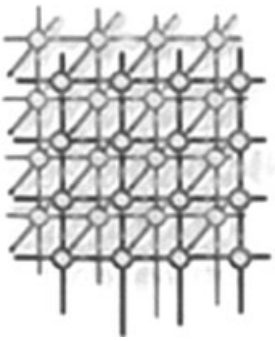
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **21**:2336–2354
Published online 20 July 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1453



Deterministic parallel selection algorithms on coarse-grained multicomputers

M. Cafaro[1,2,3,*,†], Vincenzo De Bene[4] and G. Aloisio[1,2,3]

[1] *University of Salento, Lecce, Italy*
[2] *National Nanotechnology Laboratory/CNR-INFM, Lecce, Italy*
[3] *CMCC—Euro-Mediterranean Centre for Climate Change, Lecce, Italy*
[4] *University of Salento, Lecce, Italy*

SUMMARY

We present two deterministic parallel Selection algorithms for distributed memory machines, under the coarse-grained multicomputer model. Both are based on the use of two weighted 3-medians, that allows discarding at least 1/3 of the elements in each iteration. The first algorithm slightly improves the current experimentally fastest algorithm by Saukas and Song where at least 1/4 of the elements are discarded in each iteration, while the second one is a fast, special purpose algorithm working for a particular class of input, namely an input that can be sorted in linear time using RadixSort. Copyright © 2009 John Wiley & Sons, Ltd.

Received 1 December 2008; Revised 25 February 2009; Accepted 29 March 2009

KEY WORDS: selection; weighted 3-median; RadixSort

1. INTRODUCTION

Given a set A of n (distinct) elements, the rank of an element $x \in A$, denoted as $rank(x, A)$, is the number of elements of A less than or equal to x. The problem of selecting the ith *order statistics* of a set of n elements [1] requires finding the ith smallest element; it can be stated formally as follows.

Definition 1.1. Input: A set A of n (distinct) numbers and a number i, with $1 \leq i \leq n$. Output: The element $x \in A$ such that $rank(x, A) = i$.

*Correspondence to: M. Cafaro, Facoltà di Ingegneria, Università del Salento, Via per Monteroni, 73100 Lecce, Italy.
†E-mail: massimo.cafaro@unisalento.it, massimo.cafaro@unile.it

Copyright © 2009 John Wiley & Sons, Ltd.

The Selection problem can be solved sequentially in time $O(n \log n)$ by sorting the elements using an asymptotically optimal sort algorithm such as Merge Sort or Heap Sort and then indexing directly the ith element. However, better algorithms exist. A deterministic worst-case linear time algorithm has been proposed by Blum *et al*. [2]. Schönhage *et al*. improved the algorithm using fewer comparisons in the worst case [3] introducing the idea of factories; recently, Dor and Zwick [4] proposed a selection algorithm based on their green factories with an upper bound of $2.95n$ comparisons in the worst case. A randomized algorithm, with expected linear time complexity has been proposed by Hoare [5]; thereafter, Floyd and Rivest [6] improved its average-case running time. The Selection problem in parallel has also been extensively studied and a number of parallel algorithms have been devised, as reported in Section 7.

In this paper we present two deterministic parallel Selection algorithms for distributed memory machines, under the coarse-grained multicomputer (CGM) model [7]. CGM fits coarse-grained architectures assuming p processors, each one with $O(n/p)$ local memory and the size of the input n much larger (orders of magnitude apart) than p, i.e. $p \ll n$. Therefore, the model maps much better than others on current architectures; it has been studied extensively in [8–13].

The model is based on the following assumptions: (i) the algorithms execute so-called supersteps, that consist of one phase of local computation and one phase of interprocessor communication with intermediate barrier synchronization, (ii) all of the p processors have access to $O(n/p)$ local memory, (iii) in each superstep, a processor can send and receive at most $O(n/p)$ elements and (iv) the communication network between the processors can be arbitrary. In this model, an algorithm is evaluated w.r.t. its computation time and number of communication rounds.

Although the model is simple, nevertheless it provides a reasonable prediction of the actual performances of parallel algorithms; indeed, parallel algorithms for CGMs usually have a theoretical complexity analysis very close to the actual times determined experimentally when implementing and benchmarking them.

Our first CGM algorithm slightly improves the current experimentally fastest algorithm by Saukas and Song [14], while the second one is a fast, special purpose algorithm working for a particular class of input, namely an input that can be sorted in linear time using *RadixSort* [1].

We recall here that the two current state-of-the-art CGM algorithms for the Selection problem, by Fujiwara *et al*. [15], require, respectively, $O(\min(\log p, \log \log n))$ communication rounds and $O(n/p)$ local computation per round subject to the condition $n/p \geq p^{\varepsilon}$ with $\varepsilon > 0$, and $O(1)$ communication rounds with $O((n/p) \log p)$ local computation per round subject to the same condition. However, their algorithms are mainly of theoretical interest only, and the algorithm by Saukas and Song is in practice much faster when implemented. Therefore, in this paper we compare our algorithms against the one by Saukas and Song, which is subject to the condition $n/p \geq p \log p$, requires $O(\log p)$ communication rounds, and performs $O(n/p)$ local computation per round.

Our algorithms require $O(\log p)$ rounds, and perform $O(n/p)$ local computation per round but are subject to the better condition $n/p \geq p$, which in practice represents all of the cases of interest and is almost always verified in the CGM model given that one of the assumptions of this model is $n \gg p$.

The deterministic worst-case sequential linear time algorithm is based on the use of the median of the medians to partition the elements. Many parallel Selection algorithms, including the first one proposed by Fujiwara *et al.*, were based on the median of the medians and required data

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

redistribution in each iteration to properly balance the workload. The work by Saukas and Song is based instead on the use of the *weighted median* of the medians as the partitioning pivot element; this choice, as shown in their paper, avoids data redistribution allowing, nonetheless, to discard at least 1/4 of the elements in each iteration (using the median of the medians provides the same theoretical result).

Our contribution is based on the similar idea of using two *weighted 3-medians*. The resulting parallel algorithms need five-way partitioning since we work with a couple of pivot elements; however, the use of two weighted 3-medians allows discarding at least 1/3 of the elements in each iteration, as shown in this paper.

While our first algorithm is general purpose (GP), the second is not universal, and requires using *RadixSort*. It is well known that a set of elements can be sorted in worst-case linear time using *RadixSort* if we treat the elements as binary numbers. In practice, we sort n computer words, each having b bits: all of the words are interpreted as having $d = b/r$ base 2^r digits. For example, we can sort 64 bits computer words as follows: setting $b = 64$, r=16, d=b/r=4, we need just 4 passes of *CountingSort* [1] (the auxiliary stable sort algorithm usually exploited in *RadixSort*) acting on base 2^{16} digits.

The remainder of this paper is organized as follows. We present our parallel Selection algorithms in Section 2 and a linear time algorithm for determining the weighted 3-median in Section 3. The main theoretical result of this paper, i.e. the use of two weighted 3-medians allows discarding at least 1/3 of the elements in each iteration, is presented in Section 4. Our parallel algorithms are analyzed in Section 5. The algorithms are then benchmarked in Section 6 and compared with the algorithm of Saukas and Song. Related work is recalled in Section 7 and conclusions are drawn in Section 8.

2. DETERMINISTIC PARALLEL SELECTION ALGORITHMS

Before showing the pseudocode for the proposed algorithms, we recall here the following definitions [1].

Definition 2.1. For n (distinct) elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that that $\sum_{i=1}^{n} w_i = W$ (the weights may be normalized so that $\sum_{i=1}^{n} w_i = 1$), the *lower weighted 3-median* is the element x_k satisfying

$$\sum_{i,x_i<x_k} w_i \leq \frac{W}{3} \quad \left(\text{respectively} \sum_{i,x_i<x_k} w_i \leq \frac{1}{3}\right) \tag{1}$$

$$\sum_{i,x_i>x_k} w_i \leq \frac{2W}{3} \quad \left(\text{respectively} \sum_{i,x_i>x_k} w_i \leq \frac{2}{3}\right) \tag{2}$$

Definition 2.2. For n (distinct) elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = W$ (the weights may be normalized so that $\sum_{i=1}^{n} w_i = 1$), the *upper weighted*

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
x = [ 1 2 3 4 5 6 7 8 9 ]
w = [ 1/9 1/9 1/9 1/9 1/9 1/9 1/9 1/9 1/9 ]
x_k = 5
SUM from 0 to 3 of w_i = 1/9+1/9+1/9+1/9 <= 1/3
SUM from 5 to 8 of w_i = 1/9+1/9+1/9+1/9 <= 2/3
```

3-median is the element x_k satisfying

$$\sum_{i,x_i<x_k} w_i \leq \frac{2W}{3} \quad \left(\text{respectively} \sum_{i,x_i<x_k} w_i \leq \frac{2}{3}\right) \tag{3}$$

$$\sum_{i,x_i>x_k} w_i \leq \frac{W}{3} \quad \left(\text{respectively} \sum_{i,x_i>x_k} w_i \leq \frac{1}{3}\right) \tag{4}$$

Let n be the total number of elements in the array A containing the input, p the number of processors and i the desired order statistics; moreover, at the beginning of each iteration let N and n_i be, respectively, the total number of remaining elements and the number of elements belonging to processor p_i (so that $\sum_{i=0}^{p-1} n_i = N$), and c an adjustable arbitrary parameter taken as a constant, to be determined experimentally. The pseudocode reported in the text as Algorithm 2.1 describes the parallel Selection algorithms in a unified setting, the only distinction being the value of the Boolean variable *sorted*. The initial call is *ParallelSelection*$(A, i, false, c)$ for the GP algorithm, and *ParallelSelection*$(A, i, true, c)$ for the *RadixSort*-based one. *ParallelSelection* uses internally *FiveWayPartition* and *ThreeWayPartition*, which are reported in the text, respectively, as Algorithms 2.2 and 2.3. All of the algorithms are thoroughly analyzed in Section 5. In the next sections we show first how to compute in linear time the weighted 3-medians used by our algorithms, and then proceed to show the main result of this paper, i.e. that the use of two weighted 3-medians allows discarding at least one-third of the elements in each iteration.

3. DETERMINING THE WEIGHTED 3-MEDIANS

It is well known that the weighted median can be computed in linear time [16–20]; here, we show how to compute in linear time the lower weighted 3-median (computing the upper weighted 3-median is symmetric and will not be discussed here). The algorithm reported here is a minor modification of the linear time algorithm for the weighted median. In the following algorithm the input elements appear in the X array, while their weights are stored in the Y array.

Each time the elements in X are moved, the corresponding elements in Y are also moved, in order to keep the correspondence between the elements and their weights. The algorithm is based on the divide and conquer paradigm and works using an interval $[s, e]$. The invariant is that elements with ranks from s to e and the weighted 3-median x_k are contained in $[s, e]$. At runtime the algorithm keeps track of two sums S and E. The former represents the total weight of all of the elements with rank less than s: $S = \sum_{x_j<x_s} w_j$; the latter represents the total weight of all of the elements with rank greater than e: $E = \sum_{x_j>x_e} w_j$.

At the beginning the interval $[s, e]$ is $[1, n]$ and $S = E = 0$, therefore the invariant is verified. The algorithm, reported in the text as Algorithm 3.1, determines the median x_i of the interval $X[s, e]$ and partitions the elements in X using the median x_i as the pivot. Then, it computes A and B as follows: A must be the total weight of all of the elements less than the median x_i in $X[s, e]$ while B must be the total weight of all of the elements greater than the median x_i in $X[s, e]$.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 2.1: ParallelSelection(A, i, *sort*, c)

Input: A, an array; i, the statistics to be determined; *sort*, boolean value indicating whether or not A must be sorted using *RadixSort*; c, a constant influencing the number of iterations, to be determined experimentally

Output: the ith statistics of A

```
/* The n elements of the input array A are distributed to the p
   processors so that each one is responsible for either ⌊n/p⌋ or ⌈n/p⌉
   elements; let left and right be respectively the indices of the
   first and last element of the sub-array handled by the process with
   rank id */
1  left ← ⌊id n/p⌋;
2  right ← ⌊(id + 1) n/p⌋ − 1;
3  if sort then
4      RadixSort(A, left, right);
5  while N > n/(c p) do
6      n_i ← right − left + 1;
7      m_i ← Select(i, left, right);
       /* An ALL-GATHER exchange */
8      Send to the other processes m_i and n_i;
9      Compute x and y, the two weighted 3-medians of the medians;
10     if x ≠ y then
11         m, i, n_i, N ← FiveWayPartition(A, i, x, y, sort);
12     else
13         m, i, n_i, N ← ThreeWayPartition(A, i, x, sort);
14     if m ≠ ∞ then
15         return m;
16     else
           /* the problem has not been solved in parallel */
           /* A GATHER exchange */
17         send to the processor with rank 0 the remaining n_i elements;
           /* The processor with rank 0 solves the remaining problem */
18         if id = 0 then
19             determine new left and right indices;
20             return Select(i, left, right);
21 end
```

Three cases may occur: (i) if $A+S \leq W/3$ and $B+E \leq 2W/3$ then x_i is the weighted 3-median and it is returned; (ii) if $A+S > W/3$ then $E = B+E+w_i$ and the algorithm continues searching the lower partition of $X[s, e]$, i.e. all of the elements in the interval less than x_i; (iii) if $B+E > 2W/3$ then $S = A+S+w_i$ and the algorithm continues searching the upper partition of $X[s, e]$, i.e. all of the elements in the interval greater than x_i.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 2.2: FiveWayPartition($A, i, x, y, sorted$)

Input: A, an array; i, the statistics to be determined; x, a pivot; y, another pivot; *sorted*, boolean value indicating whether or not A has been sorted

Output: the value of the ith statistics of A or ∞, a sentinel value indicating that additional work must be performed, i the new index of the statistics searched for, n_i the new local number of elements and N, the new total number of elements; note that when the ith statistics of A is found, the values of i, n_i, and N are not relevant and we return ∞ as their value

1 **if** $sorted \neq true$ **then**
2 partition the elements around x and y;
/* compute the number of elements respectively $< x$, $= x$, $> x$ and $< y$, $= y$, $> y$ */
3 compute lx_i, ex_i, bxy_i, ey_i and gy_i;
/* compute the total number of elements $< x$, $= x$, $> x$ and $< y$, $= y$, $> y$ */
/* An ALL-REDUCE exchange */
4 compute LX, EX, BXY, EY, GY;
5 **if** $i \leq LX$ **then**
6 discard all of the elements $\geq x$;
7 $n_i \leftarrow lx_i$;
8 $N \leftarrow LX$;
9 **return** ∞, i, n_i, N;
10 **else if** $LX < i \leq LX + EX$ **then**
11 **return** $x, \infty, \infty, \infty$;
12 **else if** $LX + EX < i \leq LX + EX + BXY$ **then**
13 discard all of the elements $\leq x$ and $\geq y$;
14 $i \leftarrow i - (LX + EX)$;
15 $n_i \leftarrow bxy_i$;
16 $N \leftarrow BXY$;
17 **return** ∞, i, n_i, N;
18 **else if** $LX + EX + BXY < i \leq LX + EX + BXY + EY$ **then**
19 **return** $y, \infty, \infty, \infty$;
20 **else**
21 discard all of the elements $\leq y$;
22 $i \leftarrow i - (LX + EX + BXY + EY)$;
23 $n_i \leftarrow gy_i$;
24 $N \leftarrow GY$;
25 **return** ∞, i, n_i, N;

All of the three cases maintain the invariant; the first one is the base case and allows terminating the algorithm when the interval reduces to a single element. The worst-case complexity of the algorithm is $O(n)$. Indeed, in each iteration it performs in time linear in the dimension of the interval under consideration, the operations of selection, partition and the computation of the two

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 2.3: ThreeWayPartition(A, i, x, *sorted*)

Input: A, an array; i, the statistics to be determined; x, a pivot; *sorted*, boolean value indicating whether or not A has been sorted

Output: the value of the ith statistics of A or ∞, a sentinel value indicating that additional work must be performed, i the new index of the statistics searched for, n_i the new local number of elements and N, the new total number of elements; note that when the ith statistics of A is found, the values of i, n_i, and N are not relevant and we return ∞ as their value

```
1  if sorted ≠ true then
2      partition the elements around x;
3  compute l_i, the number of elements < x;
4  compute g_i, the number of elements > x;
   /* An ALL-REDUCE exchange */
5  compute L ← Σ_i l_i and G ← Σ_i g_i;
6  if i ≤ L then
7      discard all of the elements ≥ x;
8      n_i ← l_i;
9      N ← L;
10     return ∞, i, n_i, N;
11 else if L < i ≤ N − G then
12     return x, ∞, ∞, ∞;
13 else
14     discard all of the elements ≤ x;
15     i ← i − (N − G);
16     n_i ← g_i;
17     N ← G;
18     return ∞, i, n_i, N;
```

sums A and B. Since in each iteration the interval's dimension halves, the running time of the algorithm can be described by the recurrence equation $T(n) = T(n/2) + O(n)$ whose solution is $O(n)$. The initial call is *Weighted3Median*$(X, Y, 0, 0)$.

4. MAIN RESULT

Before analyzing the algorithms, we prove the following lemma and theorem, which is the main theoretical result on which the proposed algorithms are based.

Lemma 4.1. *At the beginning of each iteration of PARALLEL-SELECTION, let N and n_i be, respectively, the total number of remaining elements and the number of elements belonging to processor p_i, so that $\sum_{i=0}^{p-1} n_i = N$. For p (distinct) elements $x_1, x_2, \ldots, x_p$ with positive weights*

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 3.1: Weighted3Median(X, Y, S, E)

Input: X, an array of input elements; X, an array of corresponding weights; S and E refers to an interval $[s, e]$, the former represents the total weight of all of the elements with rank less than s, and the latter represents the total weight of all of the elements with rank greater than e

Output: the weighted 3-median of X

```
1  find the median x_i of X;
2  partition X around x_i;
3  for j=0 to length[X] - 1 do
4      if x_j < x_i then
5          A ← A + w_j;
6      else if x_j > x_i then
7          B ← B + w_j;
8  end
9  if A + S ≤ W/3 and B + E ≤ 2W/3 then
10     return x_i;
11 if A + S > W/3 then
12     E ← B + E + w_i;
13     X' ← {x_k ∈ X : x_k < x_i};
14     return Weighted3Median(X', Y, S, E);
15 if B + E > 2W/3 then
16     S ← A + S + w_i;
17     X' ← {x_k ∈ X : x_k > x_i};
18     return Weighted3Median(X', Y, S, E);
```

$w_1, w_2, \ldots, w_p$, such that $\sum_{i=0}^{p-1} w_i = 1$ and $w_i = n_i/N$ the lower weighted 3-median x_k satisfies

$$\sum_{i, x_i \le x_k} n_i \ge \frac{N}{3} \tag{5}$$

$$\sum_{i, x_i \ge x_k} n_i \ge \frac{2N}{3} \tag{6}$$

and the upper weighted 3-median x'_k satisfies

$$\sum_{i, x_i \le x'_k} n_i \ge \frac{2N}{3} \tag{7}$$

$$\sum_{i, x_i \ge x'_k} n_i \ge \frac{N}{3} \tag{8}$$

Proof. Since $\sum_{i=0}^{p-1} w_i = 1$ and $w_i = n_i/N$, then $\sum_{i=0}^{p-1} n_i = N$, which can be rewritten as $\sum_{i,x_i<x_k} n_i + n_k + \sum_{i,x_i>x_k} n_i = N$. It follows that $n_k = N - \sum_{i,x_i<x_k} n_i - \sum_{i,x_i>x_k} n_i$ and $\sum_{i,x_i<x_k} n_i + n_k = N - \sum_{i,x_i>x_k} n_i$. By definition of lower weighted 3-median (see Definition 2.1)

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

$\sum_{i,x_i>x_k} w_i \leq \frac{2}{3}$, therefore $\sum_{i,x_i>x_k} n_i \leq 2N/3$. This in turn implies that $N - \sum_{i,x_i>x_k} n_i \geq N/3$, i.e. $\sum_{i,x_i<x_k} n_i + n_k = \sum_{i,x_i\leq x_k} n_i \geq N/3$.

Similarly, $\sum_{i,x_i>x_k} n_i + n_k = N - \sum_{i,x_i<x_k} n_i$. By definition of lower weighted 3-median (see Definition 2.1) $\sum_{i,x_i<x_k} w_i \leq \frac{1}{3}$, therefore $\sum_{i,x_i<x_k} n_i \leq N/3$; this implies that $N - \sum_{i,x_i<x_k} n_i \geq 2N/3$, i.e. $\sum_{i,x_i>x_k} n_i + n_k = \sum_{i,x_i\geq x_k} n_i \geq 2N/3$.

The proof for the upper weighted 3-median is symmetric. □

Theorem 4.2. *After each iteration, the number of elements is reduced by at least* 1/3.

Proof. Let N be the total number of remaining elements at the beginning of each iteration, and n_i the number of elements belonging to processor p_i so that $\sum_{i=0}^{p-1} n_i = N$. Without loss of generality, assume that the weights are normalized so that $\sum_{i=0}^{p-1} w_i = 1$, i.e. the weights are n_i/N. There are two cases to consider: the two weighted 3-medians are different or equal. Let us assume that the weighted 3-medians differ. By the definition of lower weighted 3-median x_k (see Definition 2.1),

$$\sum_{i,x_i<x_k} w_i \leq \frac{1}{3} \Rightarrow \sum_{i,x_i<x_k} \frac{n_i}{N} \leq \frac{1}{3} \Rightarrow \sum_{i,x_i<x_k} n_i \leq \frac{N}{3}$$
$$\sum_{i,x_i>x_k} w_i \leq \frac{2}{3} \Rightarrow \sum_{i,x_i>x_k} \frac{n_i}{N} \leq \frac{2}{3} \Rightarrow \sum_{i,x_i>x_k} n_i \leq \frac{2N}{3}$$

Taking into account that $N = \sum_i n_i$, then by Lemma 4.1 it follows that $\sum_{i,x_i\leq x_k} n_i \geq N/3$ and $\sum_{i,x_i\geq x_k} n_i \geq 2N/3$. For each median x_i, the number of local elements a_j such that $a_j \leq x_i$ is at least $n_i/2$ by definition of median. It follows that the lower weighted 3-median x_k is greater or equal to at least $\sum_{i,x_i\leq x_k} n_i/2 \geq N/6$ elements and less than or equal to at least $\sum_{i,x_i\geq x_k} n_i/2 \geq N/3$ elements. Applying the same reasoning to the upper weighted 3-median x'_k, it follows that x'_k is greater than or equal to at least $\sum_{i,x_i\leq x_k} n_i/2 \geq N/3$ elements and less than or equal to at least $\sum_{i,x_i\geq x_k} n_i/2 \geq N/6$ elements.

The total number of elements that are strictly less than the lower weighted 3-median is $L' \leq 2N/3$; the total number of elements that are strictly greater than the lower weighted 3-median is $G' \leq 5N/6$. Similarly, the total number of elements that are strictly less than the upper weighted 3-median is $L'' \leq 5N/6$ and the total number of elements that are strictly greater than the upper weighted 3-median is $G'' \leq 2N/3$.

Consider Figure 1, which shows the five-way partitioning induced by the two weighted 3-medians when used as pivot elements. If we fall into the region of the array containing all of the elements less than the first pivot we then discard all of the remaining elements i.e. $N - L'$, which are at least $N/3$. When falling in the region of the array containing all of the elements greater than the first pivot and less than the second one we discard $N - G' + N - L''$ elements, again at least $N/3$. Finally, when falling in the region of the array containing all of the elements greater than the second pivot, we discard $N - G''$, therefore at least $N/3$ of the elements.

Let us assume now that the weighted 3-medians x_k and x'_k are equal. In this case, we have a single pivot and a corresponding three-way partitioning. Reasoning as before, if we consider the pivot as a lower weighted 3-median then it follows that the total number of elements that are strictly less than the pivot is $L' \leq 2N/3$; the total number of elements that are strictly greater is $G' \leq 5N/6$. However, the pivot is also an upper weighted 3-median, which in turn implies that the total number

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

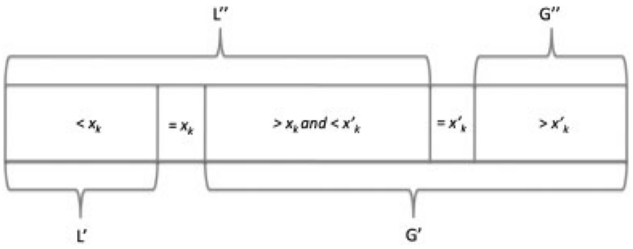


Figure 1. Array partitioning.

of elements that are strictly less than the pivot is $L'' \leq 5N/6$ and the total number of elements that are strictly greater is $G'' \leq 2N/3$. Therefore, the total number of elements that are strictly less than the pivot must be *simultaneously* less than or equal to $2N/3$ and less than or equal to $5N/6$ so that it must be less than or equal to $2N/3$. Moreover, the total number of elements that are strictly greater than the pivot must be simultaneously less than or equal to $2N/3$ and less than or equal to $5N/6$ so that, again, it must be less than or equal to $2N/3$. It follows that, when falling in one of these two regions, we discard at least $N/3$ of the elements. □

5. ALGORITHM'S ANALYSIS

We are now ready to analyze the algorithms. We discuss at first *FiveWayPartition* and *ThreeWay Partition*, since these are called within our parallel algorithms.

5.1. FiveWayPartition

In *FiveWayPartition* (see Algorithm 2.2) each processor starts checking if the elements are already in sorted order (line 1); for the GP algorithm this is not the case, so that each processor partitions its elements around the two weighted 3-medians x and y (line 2). The check requires $O(1)$ time; partitioning can be done in linear $O(n/p)$ time using Bentley's in-place three-way partitioning [21] as follows: begin partitioning the n/p array elements using the first weighted 3-median x, then apply again three-way partitioning on the elements greater than x using the second weighted 3-median y. We have found no faster way of partitioning in-place the array elements in five regions, even though this requires additional investigation.

Each processor then determines lx_i, ex_i, bxy_i, ey_i and gy_i (line 3). These represent the sizes of the local five partitions, i.e. the number of elements, respectively, less than x, equal to x, between x and y, equal to y and greater than y. The time complexity of this step is $O(1)$, since the partitioning step of line 2 has been implemented so that it returns the indices of the two pivot elements and the number of elements equal to each one of the pivots.

However, when using the *RadixSort*-based algorithm we need to determine the sizes of the partitions in line 3 of *FiveWayPartition*; this requires at most $O(n/p)$. Indeed, even though the elements are in sorted order, we cannot in general claim a worst-case complexity of $O(\log(n/p))$ for this step by exploiting a binary search, owing to the fact that there may be repeated pivot

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

elements. We use binary search at first to locate a pivot element, then scan forward and backward the array to locate the first and last position of each pivot. While in practice there are usually $O(1)$ repeated pivot elements, in the worst case these may be up to $O(n/p)$ and scanning the array will consequently be done in $O(n/p)$ time.

The next step is an ALL-REDUCE communication (line 4) in which the processors determine LX, EX, BXY, EY, GY which are the total number of elements, respectively, less than x, equal to x, between x and y, equal to y and greater than y on all of the processors. The complexity of this communication step is $O(\log p)$; indeed, each processor sends just five elements. We then compare i, the input order statistics against the five regions induced by the partition around x and y. The remaining steps either provide the desired order statistics or reduce the number of elements for the next iteration as follows.

If $i \leq LX$ then the ith order statistics lies in the first region: in lines 5–9 each processor discards all of the elements $\geq x$, since these cannot contain the desired order statistics and then updates as needed n_i and N for the next iteration.

If the ith order statistics lies in the second region (lines 10–11), we then return the weighted 3-median x. Lines 12–17 deal with the case of the ith order statistics lying in the third region; in this case each processor discards all of the elements $\leq x$ and $\geq y$ and then updates as needed i, n_i and N for the next iteration. Since we know that there are $LX + EX$ elements smaller than the ith order statistics, i is updated accordingly.

If the ith order statistics lies in the fourth region (lines 18–19), we then return the weighted 3-median y. Finally, if instead the ith order statistics lies in the fifth region (lines 20–25) each processor discards all of the elements $\leq y$ and then updates as needed i, n_i and N for the next iteration. Since in this case we know that there are $LX + EX + BXY + EY$ elements smaller than the ith order statistics, i is updated accordingly.

The steps in which we return directly the result (lines 10–11 and 18–19) require $O(1)$ time; the other steps require $O(1)$ time owing to the fact that, in order to discard the elements, we simply update the corresponding indices as needed. It follows that the parallel complexity of this algorithm is $O((n/p) + \log p)$ for both the GP and *RadixSort*-based parallel selection algorithms.

5.2. ThreeWayPartition

Let us analyze *ThreeWayPartition* (see Algorithm 2.3). Each processor starts checking if the elements are already in sorted order (line 1); for the GP algorithm this is not the case, so that each processor partitions its elements around the weighted 3-median x (line 2). The check requires $O(1)$ time; partitioning can be done in linear $O(n/p)$ time using Bentley's in-place three-way partitioning.

Each processor then computes the total number of its elements, respectively, less than x (line 3) and greater than x (line 4). The time complexity of lines 3–4 is $O(1)$, since the partitioning step has been implemented so that it returns the index of the pivot element and the number of elements equal to the pivot. As discussed in the context of *FiveWayPartition*, when the elements are in sorted order determining the sizes of the partitions requires in the worst-case $O(n/p)$ time.

Line 5 is an ALL-REDUCE communication in which $L \leftarrow \sum_i l_i$ and $G \leftarrow \sum_i g_i$ are computed. The complexity of this communication step is $O(\log p)$; indeed, each processor sends just two elements. The remaining steps either provide the desired order statistics or reduce the number of elements for the next iteration as follows.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

If $i \leq L$ each processor discards all of the elements greater than or equal to x, sets the total number of elements N to L and the number of its elements n_i to l_i (lines 6–10). Otherwise, if $L \leq i \leq N - G$ then the weighted median x is the desired order statistics and it is returned in lines 11–12. The last case to consider is $i > N - G$. Each processor discards all of the elements less than or equal to x and sets the total number of elements N to G, the number of its elements n_i to g_i, and the desired order statistics i to $i - (N - G)$ (lines 13–18).

The time complexity of the steps in which we return the result is $O(1)$; the other steps require $O(1)$ time owing to the fact that, in order to discard the elements, we simply update the corresponding indices as needed. It follows that the parallel complexity of this algorithm is $O((n/p) + \log p)$ for both the GP and *RadixSort*-based parallel selection algorithms.

5.3. GP parallel selection

For the GP parallel selection algorithm (see Algorithm 2.1), the *sorted* parameter is set to *false*. At the beginning, the workload is balanced in lines 1–2. The block distribution is simple: indeed, assuming n elements and p processors, the first and the last elements assigned to processor i are given, respectively, by $\lfloor in/p \rfloor$ and $\lfloor (i+1)n/p \rfloor - 1$; the processor in charge of an element j is given by $\lfloor (p(j+1) - 1)/n \rfloor$. Therefore, block distribution requires $O(1)$ time. Since *sort* is false in line 3, the elements are not sorted in line 4. This check requires $O(1)$ time.

In the while loop starting at line 5, there is no need to balance the workload in each iteration owing to the fact that the usage of the weighted 3-medians of the local medians guarantees that the number of elements discarded is at least $1/3$ without the need for equal block sizes [14]. While this may cause an unbalanced workload, the time saved related to communications (no longer needed for redistribution of data among processors) essentially is considerably more than the computation time due to the unbalanced workload. Indeed, using the weighted 3-medians we take into account not just the values of the medians in each block assigned to a processor, we also consider the remaining elements in each block. This in turn implies that in the next iterations, even though each processor deals with a different number of elements, we are still able to discard at least $1/3$ of the remaining elements per iteration.

Using the previous result we can easily determine the maximum number of iterations of the while loop: N, which initially is n, will become $n/(cp) = O(n/p)$ in at most $\log_{3/2}(cp) = O(\log p)$. During each iteration, line 6 determines the number of local elements n_i, while line 7 computes a processor's local median m_i in time $O(n_i)$, which is at most $O(n/p)$ in the initial iteration. Line 8 is an ALL-GATHER communication in which each processor sends its local m_i and n_i to the other processes. Since each processor sends just two elements, the complexity of the ALL-GATHER exchange communication pattern is $O(\log p)$. At the end of this step each processor computes x and y (line 9), the weighted 3-medians of the local medians, in time $O(p)$. In the algorithm, the weights associated with the local medians m_i are given by $w_i = n_i/N$.

If $x \neq y$ then each processor executes *FiveWayPartition* (lines 10–11), otherwise there is a single weighted 3-median x and each processor executes *ThreeWayPartition* (lines 12–13). Then, we check whether or not the problem has been solved in parallel. This reduces to checking in line 14 if the value returned for the ith statistics differs from the sentinel value ∞. In this case, we have solved the problem in parallel, and return the result in line 15. Otherwise (line 16), at the end of the main loop we are left with at most $n/(cp)$ elements; in line 17 all of the processors perform a GATHER

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

exchange, so that the processor with rank 0 can finally solve sequentially the remaining problem in lines 18–20.

We now analyze the parallel complexity of the algorithm. We recall here that, in the CGM model, the parallel complexity of an algorithm is given w.r.t. the computation time and the number of communication rounds. Our algorithm performs at most $O(\log p)$ rounds; the computational complexity is derived taking into account that, in each round, we determine the local median (in time $O(n/p)$), the weighted 3-medians x and y of the p local medians (in time $O(p)$) and partition the array in time $O(n/p)$ for both *FiveWayPartition* and *ThreeWayPartition*. It follows that our GP algorithm solves the Selection problem in the CGM model with $O(\log p)$ rounds and $O(n/p)$ local computation per round if $n/p \geq p$.

We also give a traditional analysis of the algorithm. For each iteration we do an ALL-GATHER and an ALL-REDUCE (during *FiveWayPartition* or *ThreeWayPartition*). Since both cost $O(\log p)$ and we have a total of $O(\log p)$ iterations, the communication complexity of the main loop is $O((\log p)^2)$. If the problem has not been solved in parallel, then we also have a final GATHER exchange, which accounts for $O(\log p)$, and the communication complexity remains $O((\log p)^2)$.

The computational complexity is, as shown, $O(n/p)$ per iteration and $O(n/cp)=O(n/p)$ at the end of the main loop if the problem has not been solved in parallel. Therefore, the computational complexity is given by $O(\log p(p + n/p) + n/p)=O(\log p(p + n/p))$ and the overall parallel complexity of the algorithm is $O(\log p(\log p + p + n/p))$. For $n/p \geq p$ we can safely ignore in each iteration the terms p and $\log p$, so that the parallel complexity reduces to $O((n/p)\log p)$.

5.4. RadixSort-based parallel selection

Let us now analyze the *RadixSort*-based algorithm (see Algorithm 2.1), highlighting only the differences w.r.t. the GP algorithm. In lines 3–4, each process sorts its elements by using *RadixSort*: this is where we assume something about the input, namely that the input can be sorted in time $O(n/p)$ by each processor. It is worth noting here that we sort the elements belonging to each processor just once throughout the algorithm (a pre-processing step).

During each iteration, in line 7 each processor computes its local median m_i in time $O(1)$, since the elements have been sorted previously using *RadixSort*, so that picking the local median reduces to indexing the element $\lceil n_i/2 \rceil$. The analysis of *FiveWayPartition* and *ThreeWayPartition* is the same, with the following exceptions.

In *FiveWayPartition* we do not partition the elements in lines 1–2 owing to the fact that the elements are already in sorted order. Analogously, in *ThreeWayPartition* we do not partition the elements in lines 1–2. However, when using the *RadixSort*-based algorithm we need to determine the sizes of the partitions in line 3 of *FiveWayPartition* and lines 3–4 of *ThreeWayPartition*; this requires at most $O(n/p)$ in both cases, as already discussed.

We now analyze the parallel complexity of the algorithm. The number of rounds is the same as the GP algorithm, $O(\log p)$; the computational complexity is derived taking into account that initially each process sorts its elements using *RadixSort* (in time $O(n/p)$), and that, in each round, we determine the local median (in time $O(1)$), the weighted 3-medians x and y of the p local medians (in time $O(p)$) and the sizes of the five or three partitions (in time $O(n/p)$ for both *FiveWayPartition* and *ThreeWayPartition*). It follows that, for $n/p \geq p$, the local computation per

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

round is $O(n/p)$. Therefore, the GP and the *RadixSort*-based algorithms share the same parallel complexity.

6. BENCHMARK

In this section we propose experimental results for both our parallel Selection algorithms and the Saukas and Song algorithm, which have been implemented in C using MPI on the HP XC6000 platform. The machine used for the test is equipped with SMP nodes configured with two Intel Itanium 2 single core processors 1.4 Ghz with 1.5 MB level 3 cache and 4 GB of main memory. The interconnection network is the Quadrics *QsNet*^{II} with Elan 4 network processors. The input elements are 64 bits unsigned long integers, and *RadixSort* has been implemented to make 4 passes on base 2^{16} digits.

In the following, we refer to our parallel selection algorithms as GP and RS (*RadixSort*), while we refer to the Saukas and Song algorithm as SS. Although there are many possible input distributions, those used in our experiments are exactly the same as used in the Saukas and Song work i.e. random elements and sorted (ascending order) elements.

The experiments have been carried out on the two input distributions considering 187 million elements owing to the constraints posed by the limited amount of memory available on each node (4 GB). Recalling that in the C programming language ULONG_MAX represents the maximum unsigned long integer and RAND_MAX, the maximum number that can be generated by the random number generator, the former distribution is based on randomly generated unsigned long elements in the range [ULONG_MAX/RAND_MAX, ULONG_MAX]; in the latter the elements appear in ascending sorted order, and the input contains all of the elements in the range [ULONG_MAX - 187 000 000, ULONG_MAX].

For each input distribution generated, the algorithm has been run fifty times on up to 16 processors, and the results have been averaged for each number of processors, over all of the runs. The ith order statistics to be determined is 1000 in the first run for each number of processors; subsequently, i is incremented by adding in each successive run $(187\,000\,000 - 1000)/20$, so as to determine equispaced order statistics in the whole interval.

Finally, we have experimentally found the value of the c constant to be used in the algorithms on the HP XC 6000 in order to minimize the overall running time. This value is 450, it was 100 on both the Parsytec PowerXplorer and the Meiko CS2 platforms, as reported by Saukas and Song. Nevertheless, decreasing or increasing c does not alter considerably the running time: in our tests, the resulting running times consistently exhibited a difference of no more than 0.5 s.

It is worth noting here the following about the implementation of the algorithms:

1. the worst-case linear time selection algorithm is the same for both SS and GP (an implementation of [2], which is still experimentally the fastest);
2. the weighted median algorithm implemented for SS is not their original $O(p\,log\,p)$ algorithm, it is instead the faster $O(p)$ algorithm (therefore, the results reported for SS are even better than they should actually be);
3. the three-way partitioning algorithm is the same for both SS and GP, even though our GP algorithm applies it two times as already discussed to provide five-way partitioning.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

We show the relative speedup (the same parallel code is used for the single processor case) and efficiency achieved by our GP and RS algorithm versus the Saukas and Song algorithm in the experiments related to the case of random elements, respectively, in Figures 2 and 3, whereas Figure 4 presents the running times.

As shown, the GP and SS algorithms exhibit an almost linear speedup and very good efficiency in the case of random elements: for GP it is in the range from 1.02 to 1.07, for SS it is comprised between 0.95 and 0.97. The RS algorithm clearly outperforms the SS algorithm, as shown by the

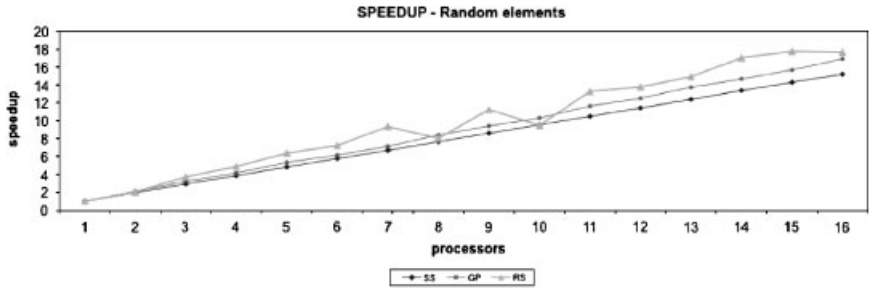


Figure 2. Speedup—Random elements.

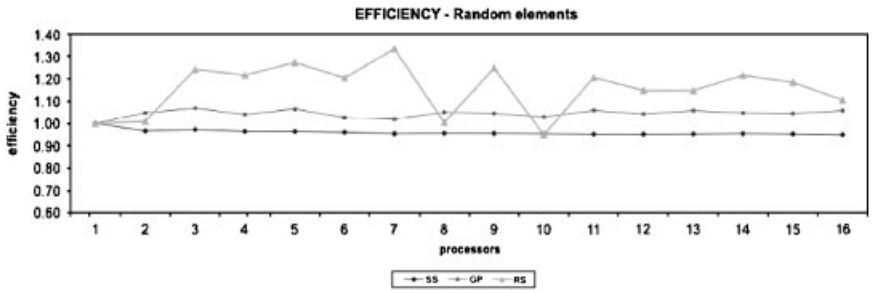


Figure 3. Efficiency—Random elements.

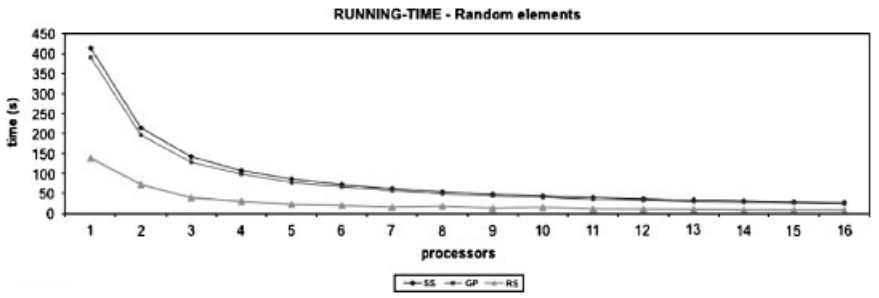


Figure 4. Running time—Random elements.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

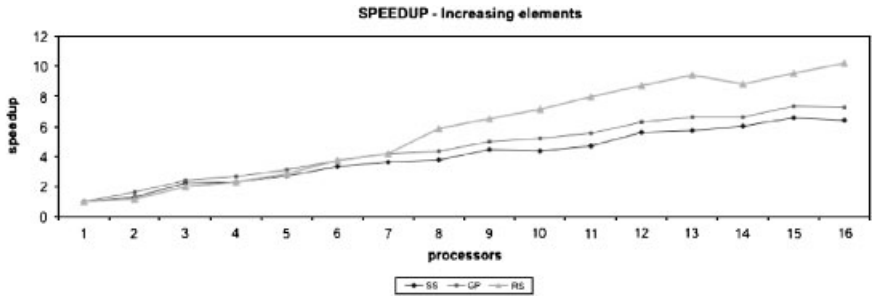


Figure 5. Speedup—Increasing elements.

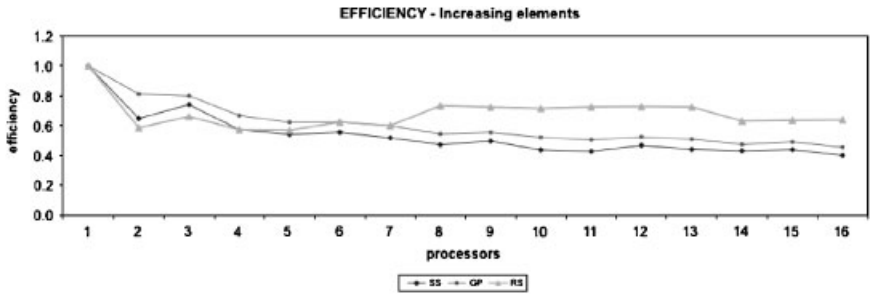


Figure 6. Efficiency—Increasing elements.

speedup, efficiency and running time curves. Indeed, once the elements are sorted, selecting the median in each iteration is immediate. The observed superlinear speedup of both RS and GP is also due to the improved, aggregate cache hit ratio, even though the level 3 cache size on the processors used is only 1.5 MB. Finally, GP is slightly superior w.r.t. the Saukas and Song algorithm regarding the overall running time, as shown in Figure 4, confirming the validity of our approach.

Figures 5 and 6 show the relative speedup and efficiency achieved in the experiments for the ascending sorted order input distribution, whereas Figure 7 presents the running times. The Saukas and Song algorithm exhibits very bad parallel performances, an expected behavior: this input causes the worst, unbalanced distribution since the first processor contains the smallest subset of elements, the second processor the next smallest subset, and so on with the last processor holding the largest elements up to ULONG_MAX. This in turn requires a greater number of iterations. However, as shown in Figure 7, the running time is much better than in the case of random elements. This is because this input distribution represents a very favorable input for the worst-case linear time selection algorithm. The time spent working in parallel is negligible w.r.t. the time spent in the worst-case linear time selection algorithm. Therefore, the running time is much better than in the case of random elements.

SS and GP present almost the same behavior w.r.t. speedup, efficiency and running time; however, GP is slightly better even in this case. Finally, while speedup and efficiency of RS are much better than SS and GP, the running time of RS in this case is the worst. This is expected, when comparing

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

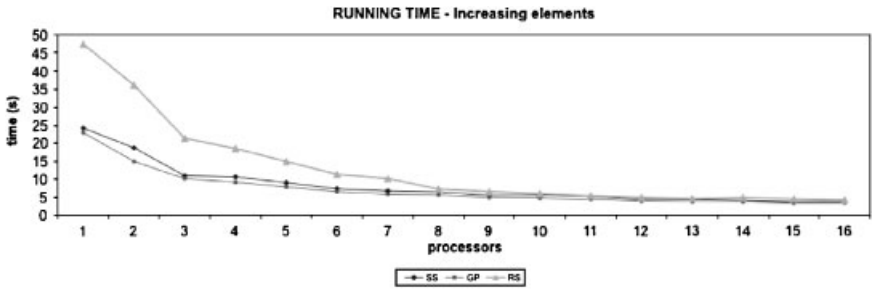


Figure 7. Running time—Increasing elements.

RS to SS and GP: the initial $O(n/p)$ sorting phase is beneficial only when the elements are not in sorted order, so that sorting them allows selecting the median in each iteration in $O(1)$ time. In this case instead, this phase is useless owing to the fact that the elements are already in sorted order.

Finally, it is worth noting that, experimentally, we found in our tests that the case of the two weighted 3-medians being equal is very rare.

7. RELATED WORK

Our algorithms are based on the work of Saukas and Song. Their key contribution is the use of the weighted median of the local medians to avoid redistributing data among processors in each iteration. The main differences between our algorithms and theirs are (i) the use of two weighted 3-medians instead of the weighted median, (ii) the use of *RadixSort* in the RS algorithm, (iii) optimized communications, (iv) an optimal algorithm to compute the weighted 3-medians. In particular, using *RadixSort*, while restricting the applicability of the algorithm allows better performances owing to the fact that the elements are sorted just once. On the contrary, Saukas and Song determine in each iteration the median of the elements assigned to each processor. We also note here that, in practice, the input is subject to a condition that is not overly restrictive and the RS algorithm is broadly utilizable. Anyway, the GP algorithm is fully general and performs slightly better than Saukas and Song algorithm.

Our algorithms exploit optimized communications requiring just two collective communications per iteration; in contrast, for each iteration their algorithm requires two broadcasts and three sends per processor. We determine the weighted 3-medians by using an $O(p)$ algorithm, their version takes time $O(p \log p)$ to compute the weighted median.

The two current state-of-the-art CGM algorithms for the Selection problem, by Fujiwara *et al.* [15], while theoretically important, are actually much slower when implemented. Indeed, the former is based on the use of the median of the medians, so that it requires redistribution of the elements in each iteration to properly balance the workload. The latter is even slower, even though in theory it requires a constant number of communication rounds. The reason is that communications are negligible w.r.t. computations, and for this algorithm the cost of computations far outweighs the benefits of reduced communications.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Among the deterministic parallel algorithms designed to partition the elements around the median of the local medians, therefore requiring equal block sizes and consequently communication to redistribute the data, we also recall here the algorithms proposed by Vishkin [22] and Bader and Jaja [23]. Randomized parallel algorithms, based on particular assumptions about the input, have been designed by Santoro *et al*. [24], Gerbessiotis and Siniolakis [25].

8. CONCLUSIONS

We have presented two deterministic, parallel algorithms for the Selection problem on distributed memory machines, under the coarse-grained multicomputer (CGM) model. Our contribution is based on the use of two weighted 3-medians, that allows discarding at least one-third of the elements in each iteration. To the best of the authors' knowledge, these are the first deterministic parallel CGM algorithms with this property. We proved this feature and compared our algorithms against the current experimentally fastest algorithm by Saukas and Song.

The general purpose algorithm performs slightly better, while the special purpose one outperforms the Saukas and Song algorithm. Our special purpose algorithm works for inputs that can be sorted in linear time using *RadixSort*. It is very fast and scalable; moreover, the input is subject to a condition that is not overly restrictive and the algorithm is therefore broadly utilizable.

Future work is related to the design and implementation of a better five-way partitioning procedure in order to make our algorithms faster, and to the open question of how to choose a different set of pivots in order to discard more than $1/3$ of the elements in each iteration.

The key problem is how to obtain a CGM algorithm with this attractive theoretical property while, at the same time, keeping the running time lower than in the Saukas and Song algorithm, as we did. Indeed, our algorithms require fewer iterations, but do more work per iteration. Extending our approach to a different set of pivots may or may not improve the running time, depending on how fast we are able to determine the pivots and to partition the elements around them.

ACKNOWLEDGEMENTS

The authors are grateful to the anonymous reviewers for their comments and suggestions.

REFERENCES

1. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms* (2nd edn). MIT Press: Cambridge, MA, 2001.
2. Blum M, Floyd RW, Pratt VR, Rivest RL, Tarjan RE. Time bounds for selection. *Journal of Computer and System Sciences* 1973; **7**(4):448–461.
3. Schönhage A, Paterson M, Pippenger N. Finding the median. *Journal of Computer and System Sciences* 1976; **13**(2): 184–199.
4. Dor D, Zwick U. Selecting the median. *Electronic Colloquium on Computational Complexity* (*ECCC*), vol. 2(31), 1995.
5. Hoare CAR. Algorithm 65: Find. *Communications of the ACM* 1961; **4**(7):321–322.
6. Floyd RW, Rivest RL. Expected time bounds for selection. *Communications of the ACM* 1975; **18**(3):165–172.
7. Dehne F, Fabri A, Rau-Chaplin A. Scalable parallel geometric algorithms for coarse grained multicomputers. *SCG '93*: *Proceedings of the Ninth Annual Symposium on Computational Geometry*, San Diego, CA, U.S.A. ACM: New York, NY, U.S.A., 1993; 298–307. DOI: http://doi.acm.org/10.1145/160985.161154, ISBN: 0-89791-582-8.

Copyright © 2009 John Wiley & Sons, Ltd.

8. Dehne F, Deng X, Dymond P, Fabri A, Khokhar AA. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. *SPAA '95*: *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, U.S.A. ACM: New York, NY, U.S.A., 1995; 27–33. DOI: http://doi.acm.org/10.1145/215399.215410, ISBN: 0-89791-717-0.
9. Ferreira A, Schabanel N. A randomized bsp/cgm algorithm for the maximal independent set problem. *Parallel Processing Letters* 1999; **9**(3):411–422.
10. Garcia T, Myoupo JF, Semé D. A work-optimal CGM algorithm for the LIS problem. *SPAA '01*: *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, Crete Island, Greece. ACM: New York, NY, U.S.A., 2001; 330–331. DOI: http://doi.acm.org/10.1145/378580.378756, ISBN: 1-58113-409-6.
11. Garcia T, Myoupo JF, Semé D. A coarse-grained multicomputer algorithm for the longest common subsequence problem. *11th Euromicro Conference on Parallel Distributed and Network based Processing*, Genoa, Italy, 5–7 February, 2003; 349–356.
12. Garcia T, Semé D. A coarse-grained multicomputer algorithm for the longest repeated suffix ending at each point in a word. *Computational Science and its Applications—ICCSA 2003 International Conference*, Montreal, Canada, 18–21 May 2003; 239–248.
13. Garcia T, Semé D. A coarse-grained multicomputer algorithm for the detection of repetitions. *Information Processing Letters* 2005; **93**(6):307–313.
14. Saukas ELG, Song SW. A note on parallel selection on coarse-grained multicomputers. *Algorithmica* 1999; **24**(3–4): 371–380.
15. Fujiwara A, Ishimizu T, Inoue M, Masuzawa T, Fujiwara H. Parallel selection algorithms for cgm and bsp with application to sorting. *IPSJ SIG Notes 19980323*, vol. 98(30), 1999; 129–136.
16. Bleich C, Overton M. A linear-time algorithm for the weighted median problem. *Technical Report 75*, Computer Science Department, Courant Institute of Math Sciences, April 1983.
17. Gurwitz C. Weighted median algorithms for L_1 approximation. *BIT* 1990; **30**(2):301–310.
18. Johnson DB, Mizoguchi T. Selecting the kth element in $X + Y$ and $X_1 + X_2 + \cdots + X_m$. *SIAM Journal on Computing* 1978; **7**(2):147–153.
19. Reiser A. A linear selection algorithm for sets of elements with weights. *Information Processing Letters* 1978; **7**(3): 159–162.
20. Shamos MI. Geometry and statistics: Problems at the interface. *Algorithms and Complexity*: *New Directions and Recent Results*, Traub JF (ed.). Academic Press: New York, 1976; 251–280.
21. Bentley JL, McIlroy MD. Engineering a sort function. *Software*, *Practice and Experience* 1993; **23**(11):1249–1265.
22. Vishkin. An optimal parallel algorithm for selection. *ADVCR*: *Advances in Computing Research* 1987; **4**:79–85.
23. Bader DA, JáJá J. Practical parallel algorithms for dynamic data redistribution, median finding, and selection. *Proceedings of IPPS '96*, *The 10th International Parallel Processing Symposium*, Honolulu, HI, U.S.A., 15–19 April 1996; 292–301.
24. Santoro N, Sidney JB, Sidney SJ. A distributed selection algorithm and its expected communication complexity. *Theoretical Computer Science* 1992; **100**(1):185–204.
25. Gerbessiotis AV, Siniolakis CJ. Deterministic sorting and randomized median finding on the BSP model. *SPAA '96*: *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy. ACM: New York, NY, U.S.A., 1996; 223–232. DOI: http://doi.acm.org/10.1145/237502.237561, ISBN: 0-89791-809-6.

Copyright © 2009 John Wiley & Sons, Ltd.