Exact and Approximate Median Splitting on Distributed Memory Machines

Matthieu Garrigues, Antoine Manzanera

Exact and Approximate Median Splitting on Distributed Memory Machines

Matthieu Garrigues and Antoine Manzanera
UEI, ENSTA-ParisTech, Paris, France

Abstract— *We present in this paper a new fine grained median split algorithm which places the median on the middle index of an input array of size N. Running on P processors, the randomized version converges in $O\left(\frac{N}{P} \times (log(N) + \mu)\right)$ average time where $0 < P < \frac{N}{4}$ and μ is the time to swap a pair of elements between two nodes. At each iteration, the nodes process only its local elements and exchange part of them with another processor of the network. This makes it a decentralized parallel algorithm and offers the possibility to take advantage of massively parallel computing networks based on distributed memory systems.*

Keywords: median splitting, parallel algorithms, distributed memory machines

1. Introduction

Useful in many fields like statistics or signal processing median selection has been widely used in computer science in the last decades. Given a set of N elements from a totally ordered space, it consists in picking the element that is superior and inferior to the same number of elements. A expensive way of solving the problem would be to sort the whole input so that the median index is $\lceil \frac{N}{2} \rceil$.

Several non parallel algorithms exist. Given a pivot value p, quickSelect [1] splits the input array in two parts, places inferior elements before p and superior elements after, then recursively processes the part that contains the median. In average, it divides the problem size by a factor two after each iteration. Its average run-time is in $O(N)$ but at worst in $O(N^2)$. Median of median [1] provides a method for choosing a pivot that ensures that the run-time stays in $O(N)$ but is less efficient in practice. [2] is a randomized algorithm that successively estimates an interval that contains the median value.

Many works present algorithms that efficiently run on different PRAM models. [3] shows how median selection can map on coarse-grained computers, when the input size is an order of magnitude higher than the number of processors. It presents and benchmarks parallel implementations of [1] and [2] on the connection machine 5 (CM-5). In theses approaches, which reduce the number of elements to process after each iterations, load balancing is needed in order to feed all processors at any iteration. Randomized parallel algorithms in [4] and [5] require $O(loglog(N))$ steps for convergence. In [6], the author presents some fine grained and real-time hardware implementation of median filtering.

In this paper, we present a new fine grained distributed median randomized algorithm. In opposition to the state of the art algorithms, it is efficient on networks (See Fig. 2) containing $O(N)$ (up to $\frac{N}{4} - 1$) processing units running in parallel. Its average observed run-time is $O\left(\frac{N}{P} \times (log(N) + \mu)\right)$ where N is the input size, P the number of processors and μ the time required to swap 2 elements between two nodes. This paper is organized as follows: We first introduce the basic operators used as building blocks. Then, we present three algorithms in their sequential form, from the simpler to the most efficient one. Finally, we discuss about the parallel aspects of the final method in Sect. 5 and its convergence in Sect. 6.

2. Primitives and Notations

Let A be an array of elements drawn from some totally ordered set. $\forall i \in \{0, N-1\}, A[i]$ represents the ith element of the array.

The algorithms presented in this paper make use of the following primitives:

- $reorder(A, i, j)$ (and by extension $reorder(A, i, j, k)$) reorders the ith and jth (resp. the ith, jth, and kth) elements of A, such that $A[i] \leq A[j]$ (resp. $A[i] \leq A[j] \leq A[k]$) after the call.
- $select_{\min}(A, i, j, k)$ (resp. $select_{\max}(A, i, j, k)$) exchanges, if smaller (resp. greater), $A[i]$ with $\min(A[j], A[k])$ (resp. $\max(A[j], A[k])$).

Each instance of these operations is said *effective* and returns true if it actually modifies the input array. If not, it returns false.

3. Swap on Tree

In this section we present the basic version of our algorithm: the principle is to map the input vector A onto a binary tree, reorganizing the data such that the left half subtree is a inf-semilattice, the right half subtree is a sup-semilattice, and the median is on the root. See Fig. 1 for a graphical representation, showing the left (resp. right) part of the tree under (resp. over) the root.

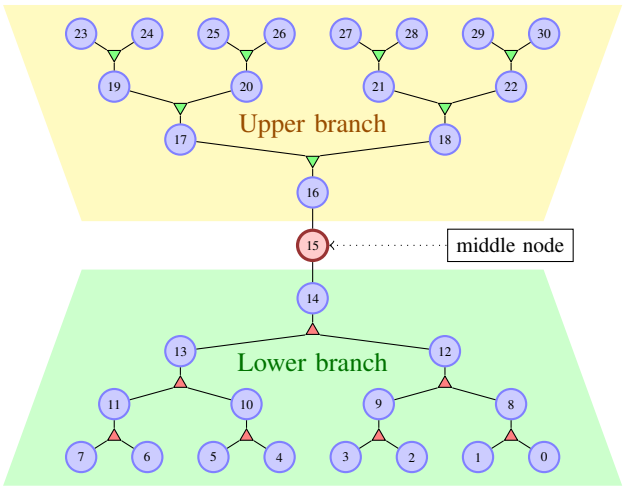

Fig. 1: A vector of 31 elements mapped on a binary tree made of an inf-semilattice (upper branch), a sup-semilattice (lower branch) and a middle node (median). Each node is labelled with its position in the array. The triangles represent the $select_{\min}$ and $select_{\max}$ operators.

The algorithm updates the input A using $select_{\min}$ (in the upper branch), $select_{\max}$ (in the lower branch), and $reorder$ (at the middle node). For simplicity we suppose that dimension $N = 2^k - 1, k > 1$.

Listing 1: Algorithm swap on tree

```
1 middle = (N-1)/2
2 repeat {
3   for (i = (middle - 1)/2; i ≥ 1; i--)
4     //lower branch
5     selectmax(A, middle-i, middle-2i, middle
          -2i-1)
6     //upper branch
7     selectmin(A, middle+i, middle+2i, middle
          +2i+1)
8   endfor
9   //middle node
10  eff = reorder(A, middle-1, middle, middle
        +1)
11 } until not eff
```

a) Complexity Analysis: Every iteration exchanges the greatest element of the lower branch with the smallest of the upper branch. Convergence occurs as soon as the *reorder* instruction (line 10) becomes uneffective, which takes at most $N/2$ iterations.

Considering the binary tree, the inputs of even levels are the output of odd levels (and *vice versa*). Then, half the tree can be processed in parallel, using $N/4$ concurrent tasks. However the number of iterations remains in $O(N)$, then the run time is $O\left(\frac{N^2}{P}\right)$ using P processors. The poor efficiency of this basic version comes from the fact that all the elements that are not placed in the good half of the array (at most $N/2$ in every half) will have to pass through the middle node, because only one element can travel from the upper branch to the lower branch at each iteration. We address this bottleneck issue in the next section.

4. Swap Inter Branches

We describe here an algorithm that improves the *connectivity* between the lower and upper branches by providing shortcuts to travel from one branch to the other.

Keeping the instructions of the *swap on tree* version, we also reorder each node of the lower branch with its counterpart in the upper branch; more precisely, $\forall i \in [0, \lfloor N/2 \rfloor - 1]$, $A[i]$ and $A[N - i - 1]$ are reordered. In a nutshell, the $select$ operators guarantees the convergence, whereas $reorder$ works as a catalyst.

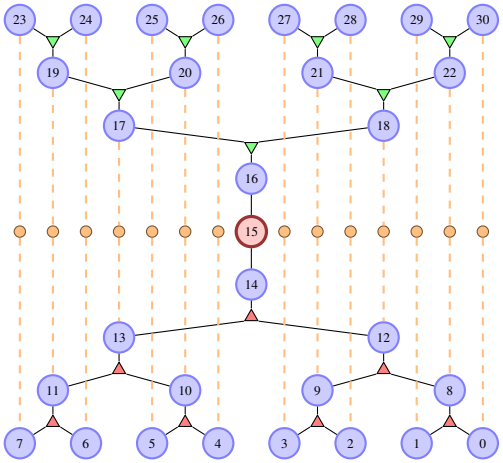

Fig. 2: Binary tree drawn on a vector of 31 elements. The circles and dashed lines show the additional $reorder$ operators of the algorithm *swap inter branches*. Each node is labelled with its position in the array.

Note that elements in the upper branch are only reordered with elements of the same level in the lower branch. The method takes advantage of the fact that at iteration i the probability of an element in the level l to be in the wrong branch is higher if l is close to the middle node. This property becomes stronger when i increases.

a) Complexity Analysis: This version keeps the high potential of parallelism, since all the instructions in the *swap inter branches* loop can be executed concurrently on $N/2$ processors.

After one iteration of *swap inter branches* instructions (line 4), there remain at most $N/4$ misplaced elements in every branch (otherwise it would mean that $k > N/4$

Listing 2: Algorithm swap inter branches

```
middle = (N-1)/2
repeat {
  for (i = 0; i < middle; i++)//swap inter
      branches
    reorder(A,i,N-i-1)
  endfor
  for (i = (middle - 1)/2; i ≥ 1; i--)
    //lower branch
    selectmax(A, middle-i, middle-2i, middle
        -2i-1)

    //upper branch
    selectmin(A, middle+i, middle+2i, middle
        +2i+1)

  endfor
  //middle node
  eff = reorder(A, middle-1, middle, middle
      +1)
} until not eff
```

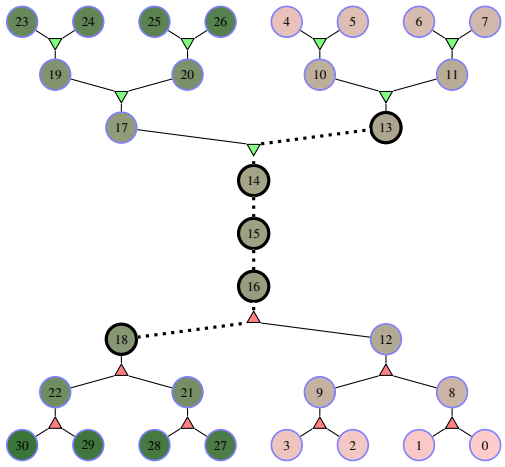

Fig. 3: Worst case for *swap inter branches*. the label and the color of a node represent the ranks of the value that it stores. The bottleneck is highlighted in dashed black.

elements smaller than the median are smaller than k other elements in the other branch and then there would be $2k > N/2$ elements smaller than the median, which is absurd). Unfortunately, the number of iterations is still $N/4$ in the worst case (see Fig. 3), so only two times faster than the *swap on tree* version. It turns out that this version still suffers from a bottleneck: in the *swap inter branches* instructions, every node is always reordered with its vertical counterpart in the other branch. Then, if some elements belonging to two diagonally opposed quarters of the tree need to be compared (like in Fig. 3), they still have to go through the middle node. Furthermore, the $select_{\min}$ and $select_{\max}$ operations (lines 8 and 10) often perform the same comparisons, since many elements do not move during one iteration (this is especially true during the last iterations). Finally, a lot of operations are not effective. In the next section we present the final version of the algorithm, adding horizontal motion at each level to improve the data mobility.

5. Final Algorithm

This section presents our final algorithm, which improves the previous version by changing at every iteration the counterpart of every node, thanks to index shifting at each level of the tree. The purpose is to minimize the bottleneck effect by allowing diagonal motion in the *swap inter branches* instructions, and to maximize the efficiency of the $select_{\min}$ and $select_{\max}$ operations, by a constant renewal of the data caused by the horizontal motion (see Fig. 4). These improvements allow to converge in $O(log(N))$ steps instead of $O(N)$ with the previous algorithm. We present the sequential version (List. 3), where RANDOM(N) is a random integer between 0 and N-1, and MOD(p,n) is the remainder of the Euclidean division of p by n, and then discuss about parallelization.

Listing 3: Final algorithm, sequential form

```
middle = (N-1)/2
repeat {
  eff = false
  //size of the leaf level
  Lsize = (N+1)/4
  //first index of the leaf level (lower
      branch)
  Plow = Lsize-1
  //first index of the leaf level (upper
      branch)
  Pupp = N-Lsize
  //shift offset of the leaf level (lower
      branch)
  Olow_son = RANDOM(Lsize)
  //shift offset of the leaf level (upper
      branch)
  Oupp_son = RANDOM(Lsize)

  repeat {//Processing of one level
    Olow_dad = RANDOM(Lsize/2)
    Oupp_dad = RANDOM(Lsize/2)
    for (k = 0; k < Lsize/2; k++)
      Ilow_dad = Plow + Lsize/2
                 - MOD(k + Olow_dad, Lsize/2)
      Iupp_dad = Pupp - Lsize/2
                 + MOD(k + Oupp_dad, Lsize/2)
      Ilow_child1 = Plow - MOD(2k + Olow_son,
          Lsize)
      Ilow_child2 = Plow
                    - MOD(2k + 1 + Olow_son,
                        Lsize)
      Iupp_child1 = Pupp + MOD(2k + Oupp_son,
          Lsize)
      Iupp_child2 = Pupp
                    + MOD(2k + 1 + Oupp_son,
                        Lsize)
      //lower branch
```

```
30        selectmax(A, Ilow_dad, Ilow_child1,
              Ilow_child2)
31        //upper branch
32        selectmin(A, Iupp_dad, Iupp_child1,
              Iupp_child2)
33        //swap inter branches
34        eff = eff || reorder(A, Ilow_dad,
              Iupp_dad)

35      endfor
36      Plow = Plow + Lsize/2
37      Pupp = Pupp - Lsize/2
38      Olow_son = Olow_dad
39      Oupp_son = Oupp_dad
40      Lsize = Lsize/2
41    } until (Lsize == 1)
42    //middle node
43    reorder(A, middle - 1, middle, middle + 1)
44  } until not eff
```

At each iteration, we scan the nodes from the leaves to the middle node using $reorder$, $select_{\max}$ and $select_{\min}$. A $select$ operation rearranges one element of a pair of nodes at level l with its parent node at level $l-1$. But, unlike *swap inter branches*, relations between levels are shifted with a random offset drawn before each iteration. Figure 4 shows the resulting relations. Arguments of $reorder$ are affected using the same method.

a) Convergence detection: The variable eff tracks whether during one iteration a $reorder$ between branches has been effective (see line 34 of listing 3). If not, we can ensure that the median is on the middle node.

b) Parallelization: At a given time, one iteration reads and writes two opposite children branch levels, and two opposite father branch levels. Since each iteration processes nodes from the leaves to the middle of the tree, we can pipeline the iterations without breaking write/read dependencies. Thus, using a pipeline of $log_2(N)/2$ stages, as many iterations can run in parallel (see Fig. 5).

c) Complexity Analysis: If we suppose that the random number generator used to compute offset always returns 0, the algorithm is equivalent to *swap and shift*. In this case, the number of iterations in the worst case is still $N/4$. Section 6 shows that this algorithm takes in average $2 \times log_2(N)$ pipeline steps to converge. Using P processors, one pipeline step runs in time $O(\frac{N}{P})$. This gives an average time complexity of $O\left(log(N) \times \frac{N}{P}\right)$. Figure 6 compares the convergence of the three presented algorithms.

6. Convergence

Let us consider the input array being re-arranged at a certain step of the algorithm. We define the event e as follows: an element drawn at random from the array is misplaced (i.e. inferior (resp. superior) to the median and located in the upper (resp. lower) branch). In this section we model the evolution of the probability $P(e)$ over the pipeline steps. First, we analyze how $P(e)$ is affected by the operators $select_{\min}$, $select_{\max}$ and $reorder$. Then we observe the average number of pipeline steps needed to place all elements.

Let a and b be two consecutive levels, a carrying the parents of nodes on level b. The event e_x (resp. f_x) occurs

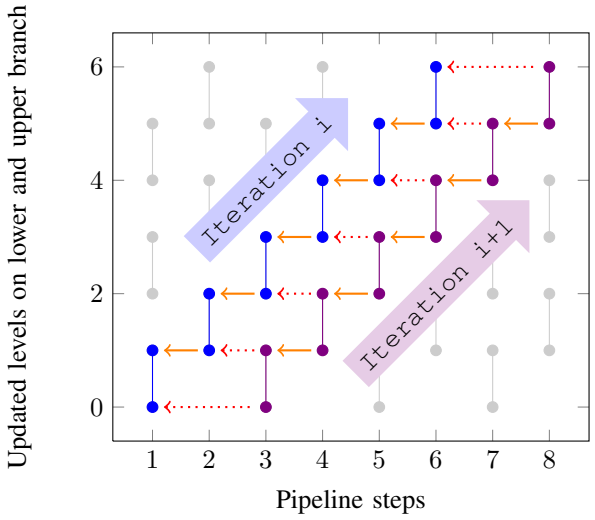

Fig. 5: Dependencies between steps of two iterations. Plain arrows show dependencies between the steps of one iteration, dependencies between iterations are dotted.

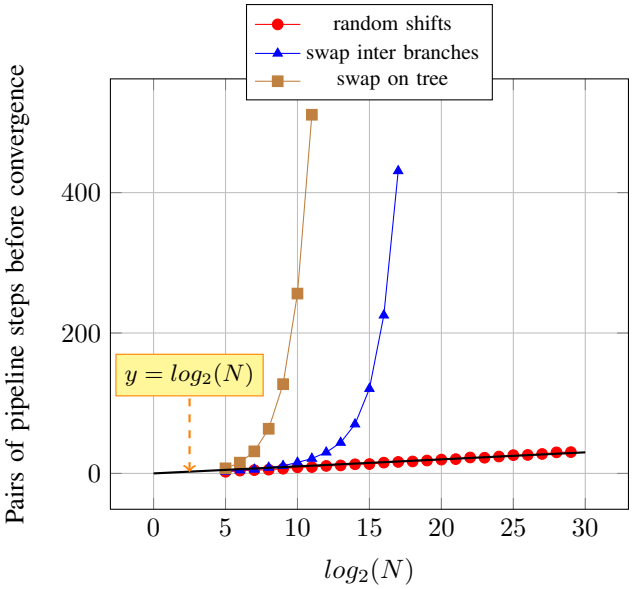

Fig. 6: Average pairs of pipeline steps before convergence on different algorithms. Means of 100 executions on random inputs.

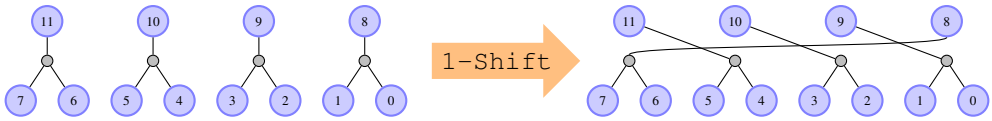

Fig. 4: Example of index shifting using an offset of size 1. Unmodified (left) and shifted (right) relations between two levels. *select* operators are shown in gray. Each nodes is labeled with its position in the array.

when an element is misplaced on level x before (resp. after) the *select* operation. On each branch, level 0 contains the root and level l contains the parents of level $l+1$. Given $p_x = P(e_x)$, we search S_p and S_l such that:

$$\begin{cases} P(f_a) &= S_p(p_a, p_b) \\ P(f_b) &= S_l(p_a, p_b) \end{cases}$$

Figure 7 shows all the possible configurations for the three operators.

$$\begin{aligned} S_p(p_a, p_b) &= \sum_{i=1}^{8} P(f_a|K_i) \times P(K_i) \\ &= 1 - P(\overline{f_a}|K_1) \times (1-p_a) \times (1-p_b)^2 \\ &= 1 - (1-p_a) \times (1-p_b)^2 \end{aligned}$$

$$\begin{aligned} S_l(p_a, p_b) &= \sum_{i=1}^{8} P(f_b|K_i) \times P(K_i) \\ &= P(f_b|K_4) \times (1-p_a) \times p_b^2 + P(f_b|K_6) \times \\ &\quad p_a \times (1-p_b) \times p_b + P(f_b|K_8) \times p_a \times p_b^2 \\ &= \frac{(1-p_a) \times p_b^2}{2} + p_a \times (1-p_b) \times p_b + p_a \times p_b^2 \\ &= p_b \times \left(\frac{p_b \times (1-p_a)}{2} + p_a \right) \end{aligned}$$

Using the same method, we model how *reorder* affects $P(e)$ on level l of lower and upper branches. The event e (resp. f) occurs when an element is misplaced on level l before (resp. after) the operation. The probability of e is noted $p = P(e)$. R defines the relation between p and $P(f)$: $P(f) = R(p)$.

$$\begin{aligned} R(p) &= \sum_{i=1}^{4} P(f|L_i) \times P(L_i) \\ &= P(f|L_1) \times (1-p)^2 + \\ &\quad \big(P(f|L_2) + P(f|L_3)\big) \times p \times (1-p) + P(f|L_4) \times p^2 \\ &= (1-p) \times p \end{aligned}$$

Let e_l^s denote the following event: an element of level l is misplaced after running the step s. It occurs with a probability $P_l^s = P(e_l^s)$. Knowing that a step is a *select* followed by a *reorder* we can build P_l^s by recurrence on s:

$$P_l^s = \begin{cases} R(S_p(P_l^{s-1}, P_{l+1}^{s-1})) & \text{if} \quad l \bmod 2 = s \bmod 2 \\ S_l(P_{l-1}^{s-1}, P_l^{s-1}) & \text{otherwise} \end{cases}$$

P_l^0 is given by the distribution of the input array. In case of all orderings of the input are equally likely, we have: $\forall l,\ P_l^0 = \frac{1}{2}$. On the root and leaf levels of each branch, the previous formula is undefined every two pipeline steps. In this special case, we have $P_l^s = P_l^{s-1}$. Figure 8 draws the convergence of P_l^s and the probability P^s that an element of the input is misplaced after step s:

$$P^s = \sum_{i=1}^{log_2(N+1)-1} \frac{2^i \times P_l^s}{N}$$

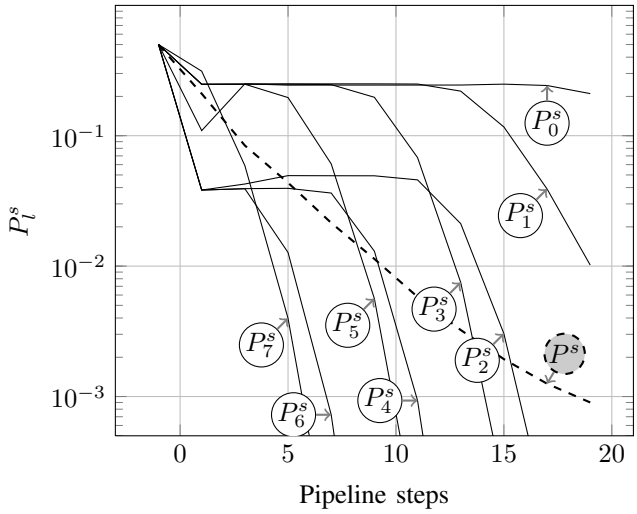

Fig. 8: Evolution of P_l^s and P^s for a input of 2^9-1 elements with $P_l^0 = \frac{1}{2}\ \forall l$. P^s shows that in average, the number of placed elements is divided by two after each pair of pipeline steps. The larger levels are the first to contain placed elements with high probability. For clarity, only values with s odd are displayed.

a) Approximation: Figure 8 shows that the probability for an element to be misplaced is reduced by a factor two every two pipeline steps. If we stop the algorithm after a given step s, we can estimate, for each level, the probability P_l^s that

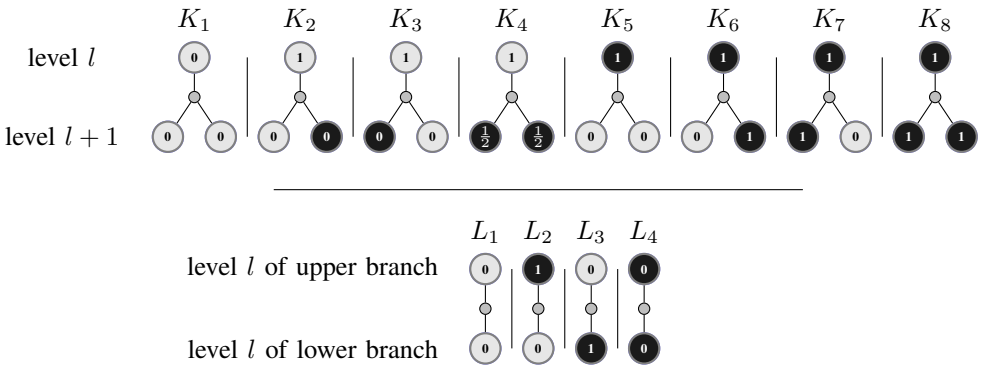

Fig. 7: All possible configurations for *reorder* and *select* operators. Misplaced elements are shown in black and placed elements are shown in gray. Each node is marked with the probability that its contains a misplaced element after the operation.

one of its elements is misplaced. Figure 9 shows the average percentage of misplaced elements after each pair of pipeline steps. For example, whatever the input size, we note that 10 pipeline steps are enough to place 99% of the elements in average.

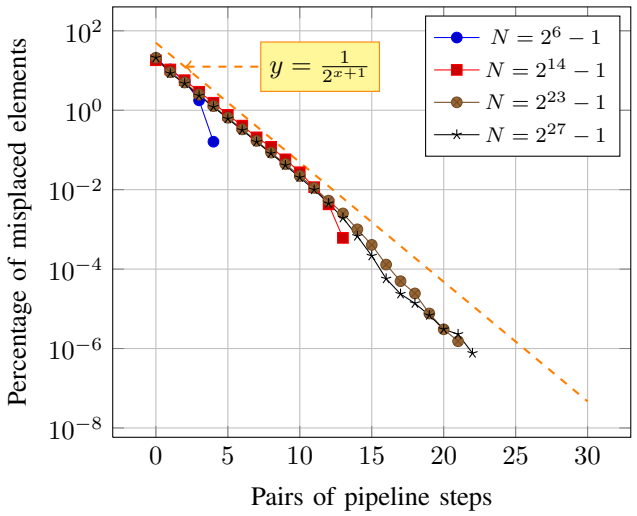

Fig. 9: Evolution of the percentage of misplaced elements over the pipeline steps. Each curve represents the mean of 10 executions of the final algorithm on random arrays.

7. Conclusion

The algorithm we presented in this paper converges to a median split of an array of N elements, placing the median at index $\frac{N}{2}$ in an observed run-time of $O\left(\frac{N}{P} \times (log(N) + \mu)\right)$ average time where μ the time to swap a pair of elements between two processors.

To make the coarse-grained parallel approaches of [3] efficient, we need to provide each processing units (PE) with enough elements, limiting their number. Because the three operators involved process locally only 2 or 3 elements without global knowledge, our approach is expected to bypass this limitation by taking advantage of architectures with a very high number of PEs (in the same order of magnitude of N). We believe that this can lead to efficient implementations on specific networks on chip with thousands of PEs.

References

[1] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, pp. 448–461, August 1973. [Online]. Available: http://dx.doi.org/10.1016/S0022-0000(73)80033-9

[2] R. W. Floyd and R. L. Rivest, "Expected time bounds for selection," *Commun. ACM*, vol. 18, pp. 165–172, March 1975. [Online]. Available: http://doi.acm.org/10.1145/360680.360691

[3] I. Al-furiah, S. Aluru, S. Goil, and S. Ranka, "Practical algorithms for selection on coarse-grained parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, pp. 813–824, 1997.

[4] S. Rajasekaran, "Randomized selection on the hypercube," *J. Parallel Distrib. Comput.*, vol. 37, no. 2, pp. 187–193, Sept. 1996. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1996.0118

[5] D. A. Bader, "An improved, randomized algorithm for parallel selection with an experimental study," *J. Parallel Distrib. Comput.*, vol. 64, no. 9, pp. 1051–1059, Sept. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2004.06.010

[6] R. M. Palenichka, "Parallel median filtering algorithms and their real-time implementation," *Cybernetics and Systems Analysis*, vol. 25, pp. 694–699, 1989, 10.1007/BF01075231. [Online]. Available: http://dx.doi.org/10.1007/BF01075231