Scalable and Efficient Parallel Selection

Christian Siebert(✉)

Laboratory for Parallel Programming, Department of Computer Science,
RWTH Aachen University, Aachen, Germany
christian.siebert@rwth-aachen.de

Abstract. Selection algorithms find the k^{th} smallest element from a set of elements. Although there are optimal parallel selection algorithms available for theoretical machines, these algorithms are not only difficult to implement but also inefficient in practice. Consequently, scalable applications can only use few special cases such as minimum and maximum, where efficient implementations exist. To overcome such limitations, we propose a general parallel selection algorithm that scales even on today's largest supercomputers. Our approach is based on an efficient, unbiased median approximation method, recently introduced as *median-of-3 reduction*, and *Hoare*'s sequential *QuickSelect* idea from 1961. The resulting algorithm scales with a time complexity of $\mathcal{O}(\log^2 n)$ for n distributed elements while needing only $\mathcal{O}(1)$ space. Furthermore, we prove it to be a practical solution by explaining implementation details and showing performance results for up to $458,752$ processor cores.

Keywords: Selection · QuickSelect · Median · Parallel algorithms · MPI

1 Introduction

Complex problems such as the search for the nearest neighbor or shortest path require a solution for a subproblem, namely *selection*. We consider this problem of selecting the k^{th} smallest element from a set of n given elements. All elements have a key, and an ordering relation denoted by $\leq$ is defined on the keys. Therefore, we can compare two keys and determine which key is smaller than the other. If all elements would be sorted according to $\leq$ then the k^{th} smallest element would be at position[1] k. As massively parallel supercomputers are becoming widespread, there is a growing need for efficient parallel solutions to the selection problem, where the n elements are distributed over p processors. A common form of the parallel selection problem in practice is the special case with a single element per processor. This forms also the base case where the other two cases with $n < p$ and $n > p$ can be reduced to. Consequently, this paper focuses

[1] In statistics, the k^{th} order statistic of a sample is equal to its k^{th} smallest value, and the position of this value is called *rank*. Unfortunately, *rank* is also used in MPI to identify a process. To disambiguate, we use the terms *position* and *MPI rank*.

R. Wyrzykowski et al. (Eds.): PPAM 2013, Part I, LNCS 8384, pp. 202–213, 2014.
DOI: 10.1007/978-3-642-55224-3_20,

on an efficient, comparison-based solution for the parallel selection problem with $n = p$. A few special cases of selection such as finding the minimum (i.e., $k = 1$) and the maximum (i.e., $k = n$) are properly solved and available in most parallel programming environments (e.g., `MPI_Reduce(...MPI_MIN...)`). Unfortunately, this is not the case for the general selection problem with an arbitrary k. Even the frequent task of finding the *median* element $\widetilde{x}$ (i.e., $k = \lfloor n/2 \rfloor$) efficiently remains unsolved in practice. Instead, a common solution is to sort all elements and then simply extract the desired element. However, this workaround is inefficient as it requires more work than necessary. For example: optimal sequential sorting requires $\mathcal{O}(n \log n)$ time (e.g., with *MergeSort*), but there exist sequential selection algorithms such as *BFPRT* [1] that work in $\mathcal{O}(n)$ time. In parallel, the situation is even worse: many scalable sorting algorithms cannot handle few elements per process (e.g., *SampleSort* [3]) let alone scale to more than a couple of thousand processes. The usual workaround, gathering all distributed elements for sequential processing, is not feasible when we approach *Exascale* [8]. Interestingly, there exist parallel selection algorithms for theoretical machines such as PRAM (Parallel Random Access Machine, see [4]), but from their complexity it can be expected that they are highly inefficient in practice. In fact, we have not encountered any implementation of such parallel selection algorithms.

The rest of this paper is organized as follows: Sect. 2 presents a well-known sequential selection algorithm and an idea for a minor improvement. By transferring both into the parallel world, Sect. 3 proposes a parallel selection algorithm. Its two main ingredients are a median approximation and a parallel partitioning scheme, respectively presented in Sects. 3.1 and 3.2. Both parts achieve a running time of $\mathcal{O}(\log p)$, turning the complete parallel algorithm into a solution that scales well with $\mathcal{O}(\log^2 p)$ while requiring only a constant amount of space. Section 4 provides practical and theoretical evidence that the approximated median indeed leads to suitable partitions. A practical performance evaluation considering different inputs and up to 458,752 processor cores demonstrates the efficiency of our implementation in Sect. 5. Finally, Sect. 6 concludes our findings related to parallel selection.

2 Sequential Selection

In 1961, *C.A.R. Hoare* published a partition-based general selection algorithm [5], also known as *QuickSelect*. Similar to *QuickSort*, *QuickSelect* is a randomized algorithm as it chooses a "pivot" element uniformly at random. Once this pivot is chosen, the algorithm partitions the input according to smaller and larger elements. Based on the size of those partitions, one can infer where the target element must reside. Contrary to *QuickSort*, which proceeds recursively in both partitions, the *QuickSelect* algorithm proceeds recursively only within this target partition. Eventually, recursion stops when the k^{th} smallest element is found. Pseudocode of a sequential *QuickSelect* implementation is shown in Algorithm 1. Although, *QuickSelect* has an expected $\mathcal{O}(n)$ running time, unlucky choices of the pivot make it as slow as $\mathcal{O}(n^2)$ in the worst case. In 1995, Kirschenhofer et al.

Algorithm 1. Seq_QuickSelect($A[1 \dots n], k$) finds the k^{th} smallest element.

```
 1: r ← random(1 ... n)
 2: pivot ← A[r]
 3: {partition A into smaller and larger elements}
 4: for i ← 1 to n do
 5:    if A[i] < pivot then
 6:       append A[i] to A_1
 7:    else if A[i] > pivot then
 8:       append A[i] to A_2
 9:    end if
10: end for
11: if k ≤ length(A_1) then
12:    {target element is among the smaller elements}
13:    return Seq_QuickSelect(A_1, k)
14: else if k > length(A) − length(A_2) then
15:    {target element is among the larger elements}
16:    return Seq_QuickSelect(A_2, k − (length(A) − length(A_2)))
17: else
18:    {target element is the pivot}
19:    return pivot
20: end if
```

improved the odds for a good running time by using a median-of-3 pivot [6]. This approach selects not only one but three sample elements at random and picks the median of these three samples as pivot. Although this improves the probability of selecting a more suitable pivot, the worst case running time still remains $\mathcal{O}(n^2)$. In fact, the authors of a more recent analysis [2] clearly state "*[...] that median-of-three does not yield a significant improvement over the classic rule: the lower bounds for the classic rule carry over to median-of-three*". The problem is that only a constant number of elements are considered in the pivot selection.

Although Kirschenhofer's median-of-3 pivot selection improves the sequential *QuickSelect* performance only insignificantly, we will turn his main idea into an effective parallel pivot selection algorithm. The advantage there is that not a constant number of samples but instead all elements are taken into consideration.

3 Parallel Selection

In summary, the *QuickSelect* algorithm consists of three major steps:

(1) chose a pivot element (preferably close to the median)
(2) partition all elements according to the pivot
(3) recursively proceed in the corresponding partition (until target is found)

The main challenge for a practical parallel selection algorithm is to provide efficient parallel implementation options for all three steps.

3.1 Median Approximation

The actual median of all elements would be optimal for a perfect partitioning into subsets of equal size. Han [4] showed that a parallel median algorithm exists with an optimal running time of $\mathcal{O}(\log p)$. However, this asymptotic complexity hides a constant that is too large for such an algorithm to be applicable in practice. In fact, he also showed that finding the median is just as complex as solving the selection problem in general. Fortunately, any element close to the median is also a suitable pivot. Therefore, we suggest the use of our median approximation approach, which was only briefly introduced in [10]. This approach also achieves the optimal $\mathcal{O}(\log p)$ running time, but in contrast to Han's algorithm is highly efficient in practice. We now explain the details of our approach and subsequently evaluate the quality of the selected median approximation in Sect. 4.

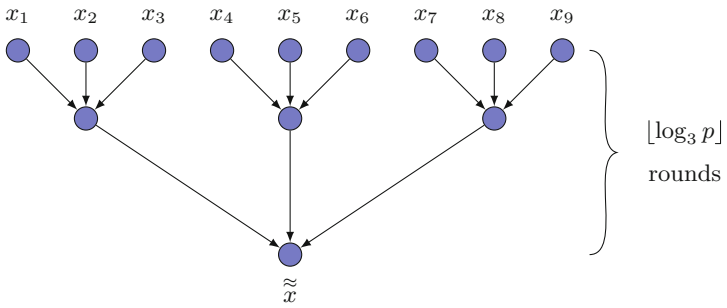

Fig. 1. Median-of-3 reduction scheme to approximate the median of elements x_i as $\approx\approx x$.

Our parallel median approximation approach groups elements into subgroups of three elements each, finds their median and recursively proceeds with the results of all other subgroups until a single result is obtained. Thus, this method is essentially a median-of-3 reduction scheme within a complete ternary tree topology as shown in Fig. 1. All processes start as leaf nodes and send their single input value x_i to a specific process, which corresponds to an inner node. Except for the last (i.e., rightmost) node which can be ignored when it has less than three children, all these inner-node processes receive the values from two other processes, select the median of these values (i.e., drop the smallest and the largest value) and forward the result to the next level. We implemented the necessary data exchange via MPI point-to-point communication, more precisely via `MPI_Isend`, `MPI_Irecv` and `MPI_Waitall`. Although such a communication tree is a recursive data structure, the actual implementation can be done in an iterative fashion, thus requiring only $\mathcal{O}(1)$ space. After $\lfloor \log_3 p \rfloor$ rounds of communication, the process at the root node obtains the final result $\approx\approx x$. This is regarded as approximation to the median of all input values x_i and broadcast as pivot to all participating processes. These processes can then compare their own element with this pivot and proceed with the partitioning step.

Note: As a median of smaller subgroups is not suitable for such a reduction scheme, the *Median-of-3* case represents the base case of a more general *Median-of-k* reduction. At first glance, a general scheme seems beneficial because increasing k (e.g., $k \in \{5, 7, \dots\}$) improves the accuracy of the approximated median until $k \geq n$ where it selects always the exact median. However, though larger values of k yield slightly fewer number of communication rounds according to $\lfloor \log_k p \rfloor$, the optimal time needed to select the median for each subgroup grows linearly with k and therefore easily dominates the overall running time of $\mathcal{O}(k \cdot \log_k p)$. Although occasionally our implementation with $k = 5$ was slightly faster than the base case, we usually observed a performance degradation, especially with higher k. Therefore, we decided to use only the fast *Median-of-3* reduction. Section 4 shows that even this base case suits our approach perfectly.

3.2 Partitioning

Once a pivot is chosen, all elements need to be partitioned into three sets: the elements that are smaller than, equal to, and larger than this pivot. While sequential partitioning is straightforward, doing so in parallel is more difficult. We utilize an auxiliary vector $\vec{v_i}$ consisting of three integers, each representing either $<$, $=$ or $>$. These integers are initialized with zeros. After comparing the process-local element x_i with the globally chosen pivot element, the corresponding vector element is set to one. Using vector notation, this can be written as

$$\vec{v_i} = \begin{cases} (1\ 0\ 0) & \text{if } x_i < \text{pivot}, \\ (0\ 1\ 0) & \text{if } x_i = \text{pivot}, \\ (0\ 0\ 1) & \text{if } x_i > \text{pivot}. \end{cases}$$

Adding these auxiliary vectors element-wise in parallel yields a vector $\vec{s}$ with the total number of elements that are smaller than, equal to or larger than the pivot, respectively. Similarly, using a parallel prefix sum we can determine the actual offsets $\vec{o}$ within each partition, that is, the number of lower-ranked processes owning elements that are smaller than, equal to, or larger than the pivot. With a linear combination of both results, we can compute the destination process d_i for each element. Using MPI terminology[2], this can be expressed as

$$\begin{aligned} \vec{s} &= \text{Allreduce}(\vec{v_i}, +) \\ \vec{o_i} &= \text{Exscan}(\vec{v_i}, +) \\ d_i &= \vec{v_i} \cdot \left(\begin{pmatrix} 0\ 0\ 0 \\ 1\ 0\ 0 \\ 1\ 1\ 0 \end{pmatrix} \cdot \vec{s}^T + \vec{o_i}^T \right). \end{aligned}$$

Both, `Allreduce` and `Exscan` exist as collective operations in the MPI standard and can be implemented efficiently using only $\mathcal{O}(\log p)$ individual communications [7,9]. After sending every element to its destination process, the

[2] We use MPI terminology: assuming x_i is the input at MPI rank i then `Allreduce` computes the sum $\sum_{j=0}^{p-1} x_j$ and `Exscan` computes the prefix sum $\sum_{j=0}^{i-1} x_j$ in parallel.

global partitioning is completed. This can be accomplished in $\mathcal{O}(1)$ time with a single `MPI_Sendrecv()` operation using `MPI_ANY_SOURCE` for the receive part. A pseudocode implementation of the partitioning step is shown in Algorithm 2. Selecting the pivot via our median approximation scheme from Sect. 3.1 ensures that the partitions for $<$ and $>$ have roughly equal size. Therefore, at most close to half of the elements remain to be searched after a partitioning. Consequently, this will lead to a total of $\mathcal{O}(\log p)$ divide-and-conquer rounds.

Algorithm 2. Par_Partition($x, pivot$) partitions all elements according to *pivot*.

```
1: {initialize auxiliary array}
2: int v[3] ← {0, 0, 0}
3: {compare a process' own element with the pivot element}
4: if x < pivot then
5:    v[0] ← 1
6: else if x == pivot then
7:    v[1] ← 1
8: else if x > pivot then
9:    v[2] ← 1
10: end if
11: {determine the sizes of the resulting partitions}
12: s ← Allreduce(v, 3, +)
13: {determine the offsets within each partition}
14: o ← Exscan(v, 3, +)
15: {combine the results to compute the destination of x}
16: if x < pivot then
17:    d ← o[0]
18: else if x == pivot then
19:    d ← s[0] + o[1]
20: else if x > pivot then
21:    d ← s[0] + s[1] + o[2]
22: end if
23: {send the element to its destination process and receive a new element}
24: Sendrecv(x, d, x_new, ANY_SOURCE)
```

3.3 Proceed in the Target Partition

Once all elements are partitioned according to the pivot element, our selection algorithm needs to proceed with the partition where the target element must reside. Similar to the sequential *QuickSelect* algorithm, this target partition is determined by comparing k with the partition sizes in $\vec{s}$. In parallel however, continuing only in the target partition comes with the challenge that only those processes which are responsible for that particular partition can participate in subsequent steps. This is especially problematic for the required collectives `Bcast`, `Allreduce` and `Exscan`, because MPI mandates that all processes

in a communicator have to participate in a collective operation. A possible solution could create one new communicator per target partition (e.g., with `MPI_Comm_split`) including only those processes that are responsible for such a partition. Unfortunately, the existing communicator creation's complexity of $\Omega(p)$ is too expensive for an efficient implementation. Instead, we propose special "range collectives" originally mentioned and briefly introduced in [10]. These range collectives are conceptually identical to their *MPI* counterparts but work only on a continuous sub-range of all processes. As such their interface provides two additional integer arguments `firstproc` and `numprocs` to specify the desired range of participating processes. Only those processes within this range must call these collectives and are actually involved in the operation; all other processes outside the specified range do not need to call the collective, and are ignored by our implementation even if they do. We have implemented all necessary range collectives for our algorithm: they work with constant space and a time complexity of $\mathcal{O}(\log p)$. Therefore, they enable an efficient parallel selection in $\mathcal{O}(\log^2 p)$ time. Moreover, using these range collectives proceeding in the target partition becomes as simple as adjusting the processor range accordingly.

4 Quality of Our Median Approximation

Although our parallel median approximation approach does not necessarily find the exact median, it always finds elements close to it. To support this claim, we quantify now the accuracy of our median approximation approach.

4.1 Simulation

First, we evaluate the accuracy of our median approximation approach in simulation experiments. For this purpose, it is applied numerous times to pseudo-random input. Since we are mainly interested in the position of an element, we use the *Scalable Parallel Random Number Generators Library* to generate input permutations of $\{1, \ldots, n\}$ uniformly at random. Figure 2 depicts the outcome of 244 billion such simulations for $n = 3^7 = 2187$ and plots for each $\approx\!\!\approx\!\!\tilde{x}$ the number of times it occurred. From these resulting frequencies, we derive individual probability estimations for each possible outcome x_i of a median approximation as $p_i = \frac{\text{frequency}(x_i)}{\text{totalruns}}$. For our $2.44 \cdot 10^{11}$ independent simulations, the resulting expected value for the outcome $E[X] = \sum x_i p_i$ is 1092.9989, which is consistent with the actual position of the true median position$(\widetilde{x}) = 1093$. In statistical terms, our median approximation method is therefore called an unbiased estimator for the true median. The resulting variance $\sum p_i \cdot (x_i - E(X))^2$ of this simulation is 2101.27, and thus the standard deviation is only 45.83. In other words, in 50 % of all simulations the position of the approximated median value is at most ± 30 elements or 1.37 % off the actual median. Even when using the most extreme outliers (cf. marked regions of unencountered results in Fig. 2) as pivots, the ratio of partition sizes would never be worse than 34 : 66. We conducted simulations also for 6561 and 19683 elements, confirming the general trend that our median approximation leads to suitable partitions in practice.

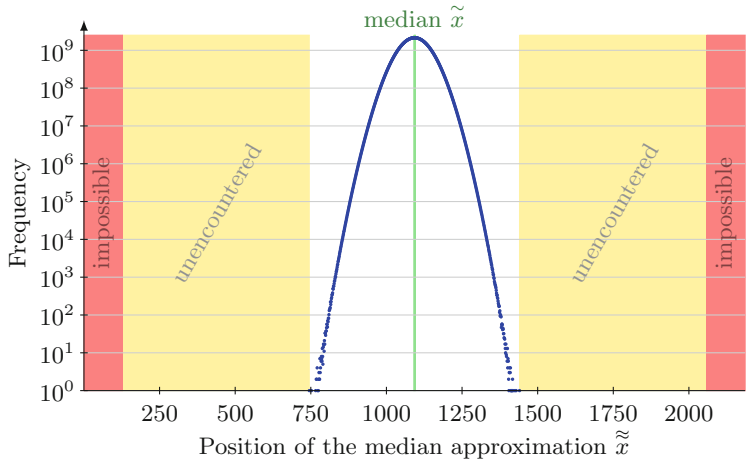

Fig. 2. Simulation of 244 billion median approximations with 2187 random elements.

4.2 Worst Case

While the previous simulation results provide a practical insight into the quality of our median approximation method, we are also interested in the theoretically possible worst case. Our construction of such a worst case is based on two properties. First, observe that the algorithm is insensitive to the order in which the elements enter a node in the ternary reduction tree (cf. Fig. 1); This follows from the commutative property of the median operation. Therefore, we are free to arbitrarily exchange the children (of course with their connected subtrees) of any node without changing the result $\tilde{\tilde{x}}$. For a systematic analysis, we choose a sorted order of the children: the leftmost child is therefore smaller than the middle child, which itself is smaller than the rightmost child. Second, we retrace the median-of-3 reduction from the root node at the bottom towards the leaf nodes at the top. While doing so, we keep track of the relationship between a parent and its children: either a child is identical to the parent or it is smaller or larger than the parent. As comparisons are transitive, this method establishes for many nodes a relationship to the result element $\tilde{\tilde{x}}$. Both, the reordering of children and the relationships with respect to $\tilde{\tilde{x}}$ are illustrated in Fig. 3.

Although the related types of nodes (i.e., $\lhd$, $\square$ and $\rhd$) are settled, we can change the unrelated nodes (i.e., '?'). This enables the construction of a worst case input for our median-of-3 reduction scheme: by inserting elements that are larger than $\tilde{\tilde{x}}$ into the unrelated nodes, we can maximize the bias of the reduction towards selecting a small element. Solving recurrence relations starting at the root node reveals the number of elements for each node type: for $n = 3^k$ ($\forall\, k \in \mathbb{N}$ and $k > 0$) induction shows that there are $2^k - 1$ elements larger than $\tilde{\tilde{x}}$ and $3^k - 2^{k+1} + 1$ elements unrelated to it. Thus in the worst case, only $2^k - 1$ elements

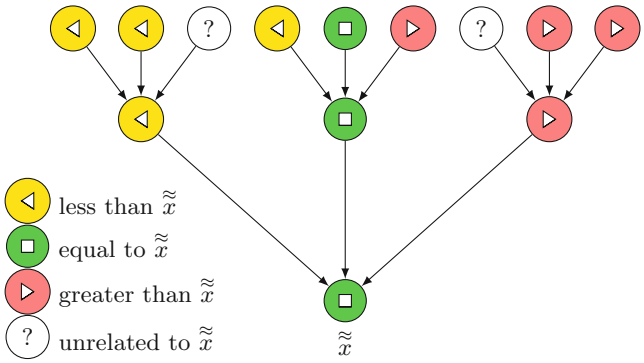

Fig. 3. "Sorted" median-of-3 reduction tree showing the node relationship with $\approx\!\approx\!\approx x$.

are guaranteed to be smaller but up to $3^k - 2^k$ elements can be larger than $\approx\!\approx\!\approx x$. We conclude that the $2^{\lfloor \log_3 n \rfloor} - 1$ largest keys and the $2^{\lfloor \log_3 n \rfloor} - 1$ smallest keys can never be chosen as an approximate median. These two border-zone areas of impossible outcomes of the median approximation are also sketched in Fig. 2. For practical values of p, even such a worst case input results only in a minor degradation of the algorithm's overall running time.

5 Performance Evaluation

This section shows the evaluation results for our parallel selection implementation. All experiments were conducted on *Juqueen*, an IBM BlueGene/Q system consisting of 28,672 compute nodes, each providing 16 IBM PowerPC A2 cores running at 1.6 GHz. Individual nodes communicate within a 5D torus network. Our implementation was built using the IBM XL C/C++ compiler V12.1 and the vendor-supplied MPI library V1.5. All presented measurements utilized 16 MPI processes per node, which corresponds to one process per core.

5.1 Different Inputs

The parallel selection algorithm recursively searches until the target element is found, which can potentially be already after a single round but also after $\mathcal{O}(\log n)$ rounds. As the number of rounds directly influences the performance of our implementation, a time variation can be expected depending on the actual input. Therefore, we measured the individual running time of 10,000 executions with 64 MPI processes for different input arguments k. Figure 4 shows that the minimum, average and maximum (except for few outliers) times over the many measurements are similar for each particular input, indicating reproducible performance for fixed inputs. In contrast, the running time with different input values of `k` can differ a lot: in our measurements it varies from 208 μs for 1 round up to 736 μs for 7 rounds. Similar performance differences can be observed by changing not only k but also the input elements themselves or both.

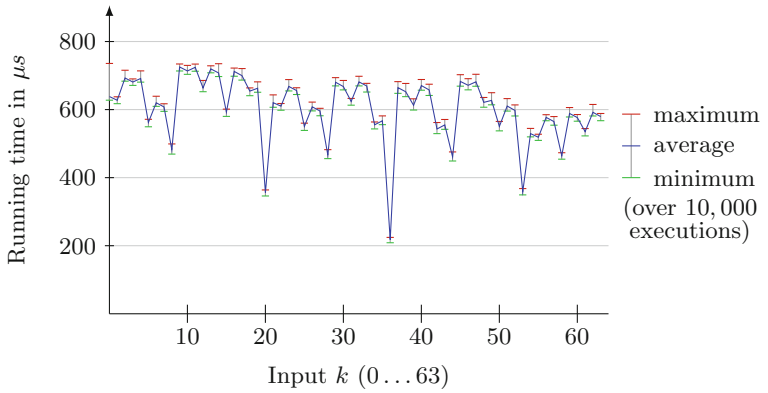

Fig. 4. Selection performance with 64 MPI processes and different values of `k`.

5.2 Scalability

For our scalability evaluation, we measured the running time of our parallel selection implementation for all possible values of k (i.e., $0 \leq k < p$), and recorded five statistical properties: the minimum, lower quartile, median, upper quartile, and maximum. The most extreme outliers and the necessary percentiles (25^{th}, 50^{th} and 75^{th}) are—of course—determined efficiently and with constant space requirements using our scalable selection solution itself. We observed that the quartiles are too close to the median values to be clearly visible. Therefore, Fig. 5 shows only the minimum, the median and the maximum timings. While the minimum values are very sensitive to the input, the maximum and especially the median values show a smooth scalability curve, representing the $\mathcal{O}(\log^2 p)$ complexity of our parallel selection algorithm. Even utilizing 458,752 MPI processes,

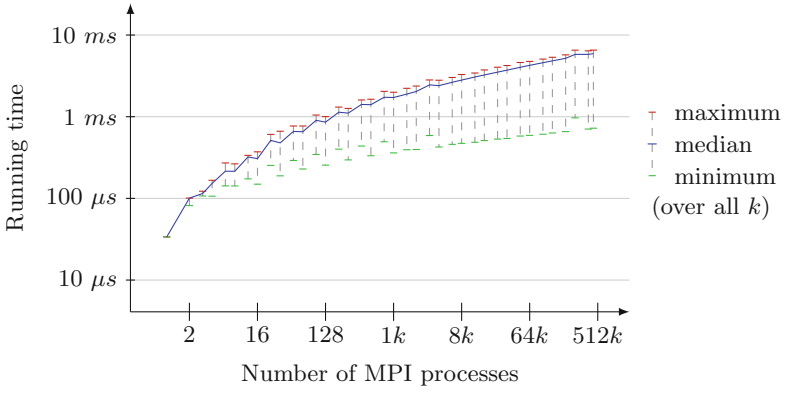

Fig. 5. Selection performance with different number of MPI processes.

representing the full *Juqueen* supercomputer, our implementation needs at most only 6.54 ms in all 458,752 measurements to select an arbitrary element.

In addition, the closely spaced quartiles show that there are only few outliers (e.g., the quick ones that find the target element in early rounds) while most executions are similarly fast. We also noticed that an average timing can be misleading as the few but distant outliers carry a heavy weight to the average and can distort it even beyond the quartile boundaries.

6 Conclusion

This paper presented a scalable and efficient solution for the parallel selection problem. The proposed algorithm is based on the original *QuickSelect* idea with a high-quality median approximation scheme for the pivot selection, an efficient partitioning using parallel prefixes and sums, and an iterative scheme employing range collectives. We provided both practical and theoretical evidence to prove that our pivot selection leads to suitable partitions. The resulting overall time complexity of $\mathcal{O}(\log^2 p)$ and the minimal space requirement makes this parallel selection solution viable for the largest supercomputers. Although it is not asymptotically optimal, it is very efficient in practice: Performance evaluation with up to 458,752 processor cores shows that it can select an arbitrary element in less than 6.6 ms, regardless of the input. As such, it is a scalable solution to the selection problem and therefore applicable to more complex parallel problems. In the future, we want to extend our base algorithm to include the case with multiple elements per process, apply it to some of the many use cases and consider different implementations as well as further optimizations.

References

1. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. J. Comput. Syst. Sci. **7**(4), 448–461 (1973)
2. Fouz, M., Kufleitner, M., Manthey, B., Jahromi, N.Z.: On smoothed analysis of quicksort and Hoare's find. Comput. Comb. **5609**, 158–167 (2009)
3. Frazer, W.D., McKellar, A.C.: Samplesort: a sampling approach to minimal storage tree sorting. J. ACM **17**(3), 496–507 (1970)
4. Han, Y.: Optimal parallel selection. ACM Trans. Algorithms **3**(4) (2007)
5. Hoare, C.A.R.: Algorithm 63 (Partition) and Algorithm 65 (Find). Commun. ACM **4**(7), 321–322 (1961)
6. Kirschenhofer, P., Prodinger, H., Martínez, C.: Analysis of Hoare's FIND algorithm with Median-of-Three partition. Random Struct. Alg. **10**, 143–156 (1997)
7. Rabenseifner, R.: Optimization of collective reduction operations. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3036, pp. 1–9. Springer, Heidelberg (2004)
8. Sack, P., Gropp, W.: A scalable MPI_Comm_split algorithm for exascale computing. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 1–10. Springer, Heidelberg (2010)

9. Sanders, P., Träff, J.L.: Parallel Prefix (Scan) algorithms for MPI. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 49–57. Springer, Heidelberg (2006)
10. Siebert, C., Wolf, F.: Parallel sorting with minimal data. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 170–177. Springer, Heidelberg (2011)