Parallel Selection by Regular Sampling

Alexander Tiskin

Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK

Abstract. Bulk-synchronous parallelism (BSP) is a simple and efficient paradigm for parallel algorithm design and analysis. In this paper, we present a new simple deterministic BSP algorithm for the classical problem of selecting the k -th smallest element from an array of size n , for a given k , on a parallel computer with p processors. Our algorithm is based on the technique of regular sampling. It runs in optimal $O(\frac{n}{p})$ local computation and communication, and near-optimal $O(\log \log p)$ synchronisation. The algorithm is of theoretical interest, as it gives an improvement in the asymptotic synchronisation cost over its predecessors. It is also simple enough to be implementable.

1 Introduction

The selection problem is a classical problem in computer science.

Definition 1. *Given an array of size n , and an integer k , $0 \leq k < n$, the selection problem asks for the array's element with rank k (i.e. the k -th smallest element, counting from 0).*

Without loss of generality, we assume that all array elements are distinct. In this paper, we restrict ourselves to comparison-based selection, where the only primitive operations allowed on the elements are pairwise comparisons, with possible outcomes “greater than” or “less than”.

The selection problem is closely related to sorting. Indeed, the naive solution to the selection problem consists in sorting the array, and then indexing the required element. Using an efficient sorting algorithm, such as mergesort, the selection problem can therefore be solved in time $O(n \log n)$. However, it is well-known that, in contrast to sorting, this time bound for the selection problem is not optimal, and linear time is achievable. The first selection algorithm running in time $O(n)$ was given by Blum et al. [4]. Further constant-factor improvements were obtained by Schönhage et al. [17], and by Dor and Zwick [8]; see also a survey by Paterson [15].

In this paper, we consider the selection problem in a coarse-grained parallel computation model, such as BSP or its variant CGM. Once again, the naive solution to the selection problem on a coarse-grained parallel computer consists in sorting the array, and then indexing the required element. Using an efficient sorting algorithm, such as parallel sorting by regular sampling (PSRS) by Shi and Schaeffer [18] (see also [20]), the selection problem can be solved deterministically in local computation $O(\frac{n \log n}{p})$, communication $O(\frac{n}{p})$, and synchronisation $O(1)$, on a parallel computer with p processors. Taking into account the

cost of sharing the input of size $O(n)$ among p processors, the communication cost $O(\frac{n}{p})$ is clearly optimal. The synchronisation cost $O(1)$ is also trivially optimal. However, the local computation cost is not optimal, due to the existence of linear-time sequential selection algorithms. A natural question arises: can the local computation cost of coarse-grained parallel selection be reduced to the optimal $O(\frac{n}{p})$, while keeping the asymptotic optimality in communication and synchronisation?

A step towards this goal was made by Ishimizu et al. [12], who gave a coarse-grained parallel selection algorithm running in optimal local computation and communication $O(\frac{n}{p})$, and in synchronisation that is essentially $O(\log p)$. Fujiwara et al. [9] improved the synchronisation cost to $O(\min(\log p, \log \log n))$. They also proposed another algorithm, with suboptimal local computation cost $O(\frac{n \log p}{p})$ (which is still better than the one achievable by naive sorting), and the optimal synchronisation cost $O(1)$. Gerbessiotis and Siniolakis [10] gave a *randomised* BSP algorithm, running in optimal local computation and communication $O(\frac{n}{p})$, and optimal synchronisation $O(1)$, with high probability.

An optimal selection algorithm in the PRAM model was given by Han [11]. There have also been significant developments in the area of experimental evaluation of selection algorithms. Various practical parallel selection algorithms have been proposed by Al-Furiah et al. [1], Saukas and Song [16], Bader [2], Cafaro et al. [5].

In this paper, we present a new simple deterministic BSP algorithm for selection. Our algorithm is based on the technique of regular sampling. It runs in optimal $O(\frac{n}{p})$ local computation and communication, and in near-optimal $O(\log \log p)$ synchronisation. Our algorithm is of theoretical interest, as it gives an improvement in the asymptotic synchronisation cost over its predecessors. It is also simple enough to be implementable.

Throughout the paper, we ignore small irregularities arising from imperfect matching between integer parameters. This allows us to avoid overloading our notation with floor and ceiling functions, which have to be assumed implicitly wherever necessary.

2 The BSP Model

The model of *bulk-synchronous parallel (BSP) computation* [22,13,3] provides a simple and practical framework for general-purpose parallel computing. Its main goal is to support the creation of architecture-independent and scalable parallel software. Key features of BSP are its treatment of the communication medium as an abstract fully connected network, and strict separation of all interaction between processors into point-to-point asynchronous data communication and barrier synchronisation. This separation allows an explicit and independent cost analysis of local computation, communication and synchronisation.

A *BSP computer* contains

- p processors; each processor has a local memory and is capable of performing an elementary operation or a local memory access every time unit;

- a communication network, capable of accepting a word of data from every processor, and delivering a word of data to every processor, every g time units;
- a barrier synchronisation mechanism, capable of synchronising all processors every l time units.

The processors may follow different threads of computation, and have no means of synchronising with one another between the global barriers.

It will be convenient to consider a version of the BSP model equipped with an external memory, which serves as the source of the input and the destination for the output, and can also be used for intermediate data. Algorithms designed for this model can easily be translated to the traditional distributed-memory setting; see [20] for details.

A BSP computation is a sequence of *supersteps*. The processors are synchronised between supersteps; the computation within a superstep is completely asynchronous. Consider a superstep in which every processor performs up to w local operations, sends up to h_{out} words of data, and receives up to h_{in} words of data. We call w the *local computation cost*, and $h = h_{out} + h_{in}$ the *communication cost* of the superstep. The total superstep cost is defined as $w + h \cdot g + l$, where the *communication gap* g and the *latency* l are parameters of the network defined above. For a computation comprising S supersteps with local computation costs w_s and communication costs h_s , $1 \leq s \leq S$, the total cost is $W + H \cdot g + S \cdot l$, where

- $W = \sum_{s=1}^S w_s$ is the total *local computation cost*;
- $H = \sum_{s=1}^S h_s$ is the total *communication cost*;
- S is the *synchronisation cost*.

The values of W , H and S typically depend on the number of processors p and on the problem size.

The original definition of BSP does not account for memory as a limited resource. However, the model can be easily extended by an extra parameter m , representing the maximum capacity of each processor's local memory. Note that this approach also limits the amount of communication allowed within a superstep: $h \leq m$. One of the early examples of memory-sensitive BSP algorithm design is given in [14].

An alternative approach to reflecting memory cost is given by the model CGM, proposed in [7]. A CGM is essentially a memory-restricted BSP computer, where memory capacity and maximum superstep communication are determined by the size of the input/output: $h \leq m = O\left(\frac{\text{input} + \text{output}}{p}\right)$. A large number of algorithms have been developed for the CGM, see e.g. [19].

In order to utilise the computer resources efficiently, a typical BSP program regards the values p , g and l as configuration parameters. Algorithm design should aim to minimise local computation, communication and synchronisation costs for any realistic values of these parameters.

3 The Algorithm

A typical approach to selection involves partitioning the input array into *subarrays*, *sampling* these subarrays, and then using (a subset of) the obtained samples as *splitters* for the *elimination* of array elements. For sequential selection, it is sufficient to choose suitable constants for the subarray size and the sampling frequency: for example, one can use subarrays of size 5, take subarray medians as samples, and take the median of samples as the single splitter. Each elimination stage reduces the amount of data by a constant factor, and therefore the overall data reduction rate is exponential.

In a coarse-grained parallel setting, a natural subarray size is n/p ; we assume that this value is sufficiently high. As before, we could reduce the amount of data by a constant factor in every stage by taking a suitable constant for the sampling frequency. However, we can do better by varying the sampling frequency. Generally, as the remaining data become more and more sparse, we can afford more and more frequent sampling on the data. This accelerates the data reduction rate to super-exponential.

Algorithm 1 (Selection)

Parameter: integer k , $0 \leq k < n$.

Input: array a of size n in external memory; we assume $n \geq p^{3/2}$.

Output: the value of the element of a with rank k .

Description.

First phase. The array is gradually reduced in size, by eliminating elements in repeated rounds of regular sampling.

Let $N = 2n$. Consider a round in which $m = N/r$ elements remain in the *global array*; for instance, in the first round we have $m = N/2$, and therefore $r = 2$. Each processor reads from the external memory a *local subarray* of size $\frac{m}{p}$. Then, by repeated application of a sequential selection algorithm, each processor selects from its local subarray a set of $2r^{1/2} + 1$ *samples* spaced at regular intervals, inclusive of the two boundary elements of the subarray. Then, all the $(2r^{1/2} + 1)p$ samples are collected in a designated processor. The designated processor sorts the array of samples, and then selects from it a subset of $2r^{1/2} + 1$ *splitters* spaced at regular intervals, inclusive of the two boundary elements (i.e. the minimum and the maximum sample).

The whole set of splitters is then broadcast across the processors, using an efficient broadcasting algorithm, such as e.g. two-phase broadcast [3]. For each splitter, a processor finds its *local rank* within that processor's subarray. These local ranks are then collected in a designated processor, which adds them up to obtain the *global rank* of each splitter within the global array.

Consider two adjacent splitters a^- , a^+ , such that the global rank of a^- (respectively, a^+) is below (respectively, above) k . We call the subset of all array elements that fall (inclusively) between splitters a^- and a^+ *the bucket*. Note that the bucket contains $\frac{(2r^{1/2}+1)p}{2r^{1/2}+1} = p$ samples. Crucially, the bucket also contains the element of global rank k , which is required as the algorithm's output.

The bucket boundary splitters a^- , a^+ are then broadcast across the processors. Each processor eliminates all the elements of its local subarray that fall outside the bucket, i.e. are either below a^- , or above a^+ . Then, a designated processor collects the sizes of the remaining local subarrays, allocates for each processor a range of locations of appropriate size in the external memory, and communicates to each processor the address of its allocated areas. The processors write their remaining local subarrays into the allocated areas.

The bucket, which has now been collected in the external memory, replaces the global array in the subsequent round. To reflect that, we update the value of m by setting it to the size of the bucket, and we let $r = N/m$. We also update the value of k by subtracting from it the global rank of a^- . The round is completed.

Rounds are performed repeatedly, until the size of the array is reduced to $\frac{n}{p}$. *Second phase.* A designated processor reads the remaining array of size $\frac{n}{p}$ from the external memory, and finds the element of rank k by a sequential selection algorithm.

Cost analysis

First phase. Consider a particular round where, as before, there are $m = N/r$ remaining elements. The communication cost of each processor reading its local subarray from the external memory is $O(\frac{m}{p}) = O(\frac{n}{rp})$. The local computation cost of each processor selecting the regular samples is $(2r^{1/2} + 1) \cdot O(\frac{m}{p}) = O(\frac{n}{r^{1/2}p})$.

For each processor, let us call the subset of its local elements that fall between two (locally) adjacent samples a *block* (note that since the local subarray is not sorted, blocks are in general not physically contiguous). We have $2r^{1/2}$ blocks per processor, and therefore $2r^{1/2}p$ blocks overall, each containing $\frac{m}{2r^{1/2}p}$ elements (including or excluding the boundaries as appropriate).

Consider a block defined by a pair of samples b^- , b^+ . We call this block *low* (respectively, *high*) if the block's lower boundary b^- is non-strictly below (respectively, strictly above) the bucket's lower boundary a^- .

A low block has a non-empty intersection with the bucket, if and only if it contains the bucket's lower boundary a^- : $b^- < a^- \leq b^+$. Since for each processor, all its blocks are disjoint, at most one of them can contain the value a^- . Therefore, across all processors, there can be at most p low blocks intersecting the bucket.

A high block has a non-empty intersection with the bucket, if and only if its lower boundary b^- is contained within the bucket: $a^- \leq b^- < a^+$. Each of the p samples contained in the bucket can be a lower boundary sample for at most one high block. Therefore, across all processors, there can be at most p high blocks intersecting the bucket.

In total, there can be at most $p + p = 2p$ blocks having a non-empty intersection with the bucket. Hence, the number of elements in the bucket is at most $2p \cdot \frac{m}{2r^{1/2}p} = \frac{m}{r^{1/2}} = \frac{N}{r^{3/2}}$. Since all the elements outside the bucket are eliminated, the maximum possible fraction (relative to N) of remaining elements gets thus raised

to the power of $3/2$ in every round. After $\log_{3/2} \log_2 p = O(\log \log p)$ rounds, the number of remaining elements is at most $2^{-(3/2)^{\log_{3/2} \log_2 p}} N = \frac{N}{2^{\log_2 p}} = O(\frac{n}{p})$.

The local computation and communication costs decrease superexponentially in every round, and are therefore dominated by the respective costs in the first round, equal to $O(\frac{n}{p})$. The total number of rounds, and therefore the synchronisation cost, is $O(\log \log p)$.

Second phase. The local computation and communication costs are $O(\frac{n}{p})$, and the synchronisation cost is $O(1)$.

Total. The overall resource costs are dominated by the first phase. Therefore, the local computation and communication costs are $O(\frac{n}{p})$, and the synchronisation cost is $O(\log \log p)$. $\square$

It should be noted that the resource costs of Algorithm 1 are controlled mainly by the frequency of initial sampling in every round. In contrast, the frequency of choosing splitters among the samples is relatively less important, and allows for substantial freedom. In particular, we could even choose all the samples as splitters. Instead, we chose the minimum possible number of splitters, thus stressing the similarity with the PSRS algorithm of [18] (see also [20]).

Also note that the assumption of external memory allows us to perform implicit load balancing between the rounds. If the algorithm were expressed in the standard distributed-memory version of BSP, then explicit load balancing within every round would be required.

4 Conclusions

We have given a deterministic BSP algorithm for selection, running in the optimal local computation and communication $O(\frac{n}{p})$, and synchronisation $O(\log \log p)$. It remains an open problem whether the synchronisation cost can be reduced to the optimal $O(1)$ without increasing the asymptotic local computation and communication costs, and without randomisation.

In the context of coarse-grained parallel computation, regular sampling has been used previously for the problems of sorting [18,20] and computing 2D and 3D convex hulls [21]. In the current paper, we have applied this technique to the problem of selection. We expect that further applications of regular sampling are possible, and conclude that this powerful technique should be an essential part of the parallel algorithm design toolkit.

References

1. Al-Furiah, I., Aluru, S., Goil, S., Ranka, S.: Practical algorithms for selection on coarse-grained parallel computers. *IEEE Transactions on Parallel and Distributed Systems* 8(8), 813–824 (1997)
2. Bader, D.A.: An improved, randomized algorithm for parallel selection with an experimental study. *Journal of Parallel and Distributed Computing* 64(9), 1051–1059 (2004)

3. Bisseling, R.H.: *Parallel Scientific Computation: A structured approach using BSP and MPI*. Oxford University Press, Oxford (2004)
4. Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. *Journal of Computer and System Sciences* 7(4), 448–461 (1973)
5. Cafaro, M., De Bene, V., Aloisio, G.: Deterministic parallel selection algorithms on coarse-grained multicomputers. *Concurrency and Computation: Practice and Experience* 21(18), 2336–2354 (2009)
6. Corrêa, R., et al. (eds.): *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques and Applications, Applied Optimization*, vol. 67. Kluwer Academic Publishers, Dordrecht (2002)
7. Dehne, F., Fabri, A., Rau-Chaplin, A.: Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry* 6, 379–400 (1996)
8. Dor, D., Zwick, U.: Selecting the median. *SIAM Journal on Computing* 28(5), 1722–1958 (1999)
9. Fujiwara, A., Inoue, M., Masuzawa, T.: Parallel selection algorithms for CGM and BSP models with application to sorting. *Transactions of Information Processing Society of Japan* 41(5), 1500–1508 (2000)
10. Gerbessiotis, A.V., Siniolakis, C.J.: Architecture independent parallel selection with applications to parallel priority queues. *Theoretical Computer Science* 301(1-3), 119–142 (2003)
11. Han, Y.: Optimal parallel selection. In: *Proceedings of ACM–SIAM SODA*, pp. 1–9 (2003)
12. Ishimizu, T., Fujiwara, A., Inoue, M., Masuzawa, T., Fujiwara, H.: Parallel algorithms for selection on the BSP and BSP* models. *Systems and Computers in Japan* 33(12), 97–107 (2002)
13. McColl, W.F.: Scalable computing. In: van Leeuwen, J. (ed.) *Computer Science Today: Recent Trends and Developments*. LNCS, vol. 1000, pp. 46–61. Springer, Heidelberg (1995)
14. McColl, W.F., Tiskin, A.: Memory-efficient matrix multiplication in the BSP model. *Algorithmica* 24(3/4), 287–297 (1999)
15. Paterson, M.: Progress in selection. In: Karlsson, R., Lingas, A. (eds.) *SWAT 1996*. LNCS, vol. 1097, pp. 368–379. Springer, Heidelberg (1996)
16. Saukas, E.L.G., Song, S.W.: A note on parallel selection on coarse-grained multicomputers. *Algorithmica* 24(3-4), 371–380 (1999)
17. Schoenhage, A., Paterson, M., Pippenger, N.: Finding the median. *Journal of Computer and System Sciences* 13(2), 184–199 (1976)
18. Shi, H., Schaeffer, J.: Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing* 14(4), 361–372 (1992)
19. Song, S.W.: Parallel graph algorithms for coarse-grained multicomputers. In: Corrêa, et al. (eds.) [6], pp. 147–178
20. Tiskin, A.: The bulk-synchronous parallel random access machine. *Theoretical Computer Science* 196(1-2), 109–130 (1998)
21. Tiskin, A.: Parallel convex hull computation by generalised regular sampling. In: Monien, B., Feldmann, R.L. (eds.) *Euro-Par 2002*. LNCS, vol. 2400, pp. 392–399. Springer, Heidelberg (2002)
22. Valiant, L.G.: A bridging model for parallel computation. *Communications of the ACM* 33(8), 103–111 (1990)