Random Sampling Techniques in Parallel Computation

Rajeev Raman[1]

Department of Computer Science
King's College London
Strand, London WC2R 2LS, U. K

Abstract. Random sampling is an important tool in the design of parallel algorithms. Using random sampling it is possible to obtain simple parallel algorithms which are efficient in practice. We will focus on the use of random sampling in fundamental problems such as sorting, selection, list ranking and graph connectivity.

1 Introduction

The notion of a random sample of a population is pervasive in many walks of life, from opinion polling to ensuring quality control in factories. The main property of a random sample is that in many cases its properties are representative of a much larger population. In parallel computation this is useful in at least two different ways: firstly, the gathering of the sample is normally highly parallelizable, and secondly, the sample, being small, can be processed and communicated with relatively little regard for efficiency. Not surprisingly, random sampling finds numerous uses in parallel algorithms.

There is no formal definition of what constitutes random sampling—indeed, one could argue that every randomized algorithm uses coin tosses to pick one computation path from among many, and hence performs "random sampling" (on the set of possible computation paths). We will therefore focus on algorithm design paradigms which are underpinned by some intuitive notion of random sampling. In this abstract we will focus on two important paradigms:

Partitioning: Given an instance of a problem, we would like to partition into (essentially) independent sub-problems, which can be solved in parallel.

- ▷ A random sample can be used to provide information about the entire input, which can be used to guide the partitioning.

Pruning: We have a problem which has an inefficient parallel solution, and we wish to find a more efficient solution to this problem.

- ▷ We can use the inefficient algorithm on a small random sample of the input, and use the solution for this sample to discard parts of the input which are irrelevant to the overall solution, thus improving efficiency.

Typically pruning is a four-step process. Given an instance I of the problem, we proceed as follows:

- Obtain a random sample R of I.
- Use the inefficient algorithm to solve the problem on R. If R is not too large compared to I, the cost applying the inefficient algorithm to R may be negligible compared to the overall cost.
- Use the solution to R to discard 'irrelevant' data from I to obtain a new instance I' which is much smaller than I, but from whose solution a solution to I can easily be obtained.
- Use the inefficient algorithm to solve the problem on I'. Since I' is hopefully much smaller than I, once again, the use of an inefficient algorithm may not result in a high overall cost.

We will describe the use of these paradigms in solving important problems such as sorting, selection, list ranking and graph connectivity [12].

2 Preliminaries

2.1 Computation model

We will assume the CRCW PRAM model of computation [12] and augment it with a number of *scan* primitives. In general, given a binary associative operator $\oplus$ and values $a_1, a_2, \ldots, a_n$, the $\oplus$-scan operation computes the n values a_1, $a_1 \oplus a_2, \ldots, a_1 \oplus \ldots \oplus a_n$. (The scan is also known as the *prefix sum* operation [12].) We will assume that a p-processor PRAM can compute the $+$-scan of p values in $O(1)$ time, where $+$ denotes ordinary addition. This is essentially the scan-PRAM model of Blelloch [3, 4]. We choose this powerful-seeming model for two reasons:

- A powerful model allows us to convey the underlying intuition without getting bogged down in details.
- By using languages such as NESL [5], algorithms described on this model can be efficiently and portably executed on a wide variety of common parallel computers.

In order to further facilitate the description of the algorithms we describe them in the *work-time (W-T)* framework [12]. In the W-T framework, a computation is divided into a number of *time steps*, and we merely specify a set of (simple) operations to be performed concurrently in each time step. The complexity of an algorithm specified in the W-T framework is given by two quantities: the number of time steps in the algorithm, which we refer to as the *step complexity* and which is denoted by $T(n)$; and the *operation complexity*, denoted by $W(n)$, which is the number of operations performed by the algorithm, summed over all time steps[1]. By Brent's theorem, the running time of an algorithm described in the W-T framework on a scan-CRCW PRAM with p processors is given by:

$$T_p(n) = O\left(T(n) + \frac{W(n)}{p}\right). \tag{1}$$

[1] n here denotes the input size.

In the typical case $W(n) \geq n \gg p$, and the term $\frac{W(n)}{p}$ dominates the RHS of the above equation. This suggests that the operation complexity is the more important parameter of the two, and it may be beneficial to reduce the operation complexity even at the cost of an increase in the step complexity. In particular, we are interested in obtaining algorithms which have *optimal speedup*: i.e. algorithms where the operation complexity is to within a constant factor of the running time of the best sequential algorithm for that problem.

2.2 Probabilistic analysis

The analysis of many randomized algorithms involves studying random variables which are *binomially* distributed. We say that X is binomially distributed with parameters m and p if X represents the number of 'successes' in m random trials, where the probability of 'success' in any trial is p, independently of the outcomes of the other trials. The distribution of X can be deduced to be:

$$\Pr[X = k] = \binom{n}{k} p^k (1-p)^{n-k}, \quad \text{for } k = 0, 1, \ldots, m. \tag{2}$$

The expected value of X, denoted by $\mathrm{E}(X)$, can easily seen to be mp. The analysis of many randomized algorithms is based on the fact that binomially distributed variables are unlikely to take values which are drastically smaller or greater than their expectation. Roughly, the probability that a binomial random variable X deviates by a fixed percentage from its expected value is at most $e^{-c\mathrm{E}(X)}$ for some constant $c > 0$ (the value of c depends the magnitude of the deviation and other factors). The precise form of these bounds (known as "Chernoff bounds") is given in the appendix.

We also need ways of analyzing the behaviour of the maximum of a collection of random variables. The distribution of the maximum can be quite different from the distribution of the individual random variables. As a simple illustration consider two random variables X and Y, each of which takes on the value 0 or 1 with probability $1/2$. However $\max(X, Y)$ takes value 0 with probability $1/4$ ($\max(X, Y) = 0$ only if $X = Y = 0$) and 1 with probability $3/4$.

A typical task might be: given the distribution of k random variables $Y_1, \ldots, Y_k$, obtain a bound on the probability that $\max\{Y_1, \ldots, Y_k\}$ exceeds a threshold value t. A situation where this task might be useful is if we randomly partition a problem into k sub-problems, and solve each sub-problem independently. The Y_i's might then represent the sizes of the sub-problems and we might be interested in studying the behavior of $\max_i Y_i$, as the overall running time of our algorithm might be dominated by the time to solve the maximum-sized subproblem.

Frequently a relatively crude method suffices to obtain such bounds. Consider the above-mentioned task. We first determine a number p such that for any particular Y_i, $\Pr[Y_i > t] \leq p$. Then:

$$\begin{aligned}\Pr[\max_i Y_i > t] &= \Pr[(Y_1 > t) \vee (Y_2 > t) \vee \ldots \vee (Y_k > t)] \\ &\leq \Pr[Y_1 > t] + \Pr[Y_2 > t] + \ldots + \Pr[Y_k > t] \\ &\leq kp\end{aligned}$$

A special case of the above is when the Y_i's are 'indicator' variables which signal the occurrence of some particular 'bad' event ($Y_i = 1$ means the bad event occurred and $Y_i = 0$ otherwise), and we wish to design an algorithm so that the chance of no bad event at all occurring is quite high. To do this we simply build in some extra tolerance into the algorithm so that individual 'bad' events occur with a sufficiently low probability. Then by the above reasoning, we can obtain an upper bound on the probability that any bad event occurs (e.g. in the above, if $p \leq k^{-2}$ then $kp \leq k^{-1}$, which is still quite small). Since the number of 'bad' events under consideration is often polynomial in the input size n, one normally tries to ensure that bad events occur with probability at most $1/n^c$ for some constant $c > 0$.

3 Partitioning I: Quicksort

Perhaps the simplest example of the partitioning paradigm is the well-known algorithm of Quicksort (see e.g. [7]). The high-level operation of parallel Quicksort is described in Figure 3, where `A` is the input array.

```
Quicksort(A, p, r) // sort the elements A[p], A[p+1], ..., A[r]
begin
1.  if (p >= r) return;
2.  pivot := p + random(r - p);
              // choose a random element of A[p..r] as the pivot
              // a call to random(i) returns a random integer from
              // {0, ... , i - 1}
3.  k := Partition(A, p, r, pivot)
              // move the elements <= pivot to A[p], ..., A[k] and
              // move the elements > pivot to A[k+1], ..., A[r]
4.  In parallel call
      Quicksort(A, p, k) and Quicksort(A, k+1, r)
end
```

Fig. 1. A recursive function `Quicksort`.

This is a straightforward example of the paradigm—ideally, we would like to partition the array about the median value in the array, thereby getting a

partition into two equal-sized sub-problems. However, it is not trivial to find the median element of an unsorted array (indeed, we will study this problem in the next section). Hence we pick a random element from the array, which on average will be "somewhere in the middle".

In [12, p464 ff.] it is described how to use $O(1)$ scans to implement the partitioning step. In our model, therefore, partitioning has $O(1)$ step complexity and $O(n)$ operation complexity. By reasoning as in the sequential case, we can conclude that the average recursion depth of parallel Quicksort is $O(\log n)$. However, we have to bound the maximum depth of the recursion tree, which is trickier. The rough idea is as follows:[2]

1. Fix a particular path from the root to a leaf in the recursion tree. Show that the probability that this path is longer than say $20 \log n$ is at most $1/n^2$.
2. Argue that the number of root-to-leaf paths is $O(n)$ (since there are $O(n)$ leaves in the recursion tree) and so the probability that *none* of the root-to-leaf paths is longer than $20 \log n$ is $O(1/n)$.
3. Conclude that the algorithm terminates in $O(\log n)$ steps and performs $O(n \log n)$ operations with probability $1 - O(1/n)$.

To prove (1), the argument goes as follows. Call a partitioning step *good* if it produces a split which is no worse than 3/4–1/4. The probability that a particular partitioning step is good is about 1/2 (choosing as pivot any of the middle half elements will do). Note that after $\log_{4/3} n < 2.409 \log n$ good partition steps we are guaranteed to reach a leaf. Hence if we go through more than $20 \log n$ partitioning steps and have not yet reached a leaf, we have had fewer than $2.41 \log n$ 'successes' in $20 \log n$ tries, with the probability of success in each try being about 1/2. By the Chernoff bounds, the probability that this happens is roughly $e^{-c \cdot 10 \log n}$. By plugging in the actual values we see that the bound is more or less exactly $1/n^2$. (Note that this analysis is quite crude and the actual constants ought to be much smaller.)

4 Pruning I: Selection

Given an unsorted set of n distinct values and an integer k, $1 \leq k \leq n$, the problem of *selection* is to find the k-th smallest value in the input. An important special case is finding the median, or $n/2$-th smallest element in the input. There are classical sequential algorithms which solve this in $O(n)$ time [7, Chapter 10], so the aim is to solve this problem quickly in parallel with $O(n)$ operations. One may note that just as Quicksort was parallelized, so may be the classical algorithm for selection which is like Quicksort but recurses only on one half of the partitioned array [7, Section 10.2]. As in Section 3, we can analyze such an algorithm and discover that it performs $O(n)$ operations on average, and that with probability $1 - O(1/n)$ its step complexity is $O(\log n)$.

[2] Logarithms are to the base 2 unless otherwise specified.

However, we will discuss a different algorithm, which is based on an algorithm of Floyd and Rivest [8]. This algorithm not only illustrates the pruning paradigm, but also has the advantage that it performs $O(n)$ operations with probability tending to 1 as n tends to infinity. Furthermore, the step complexity can be reduced to $O(1)$ with some more work, but we do not describe this here.

For the pruning paradigm, one needs an inefficient parallel algorithm to begin with. Since selection can be performed by sorting the values, the Quicksort algorithm of Section 3 gives a selection algorithm which performs $O(n \log n)$, and has step complexity $O(\log n)$, with probability $1 - O(1/n)$. We now describe the pruning-based algorithm, specialized to find the median of the input set I (which we denote by μ):

1. Choose $n^{3/4}$ values from the input I uniformly and independently at random, giving the sample R. (Perform the sampling 'with replacement'.)
   - ▷ Since $|R| = n^{3/4}$ we can sort R with $O(n^{3/4} \log n) = O(n)$ operations.
2. Sort R and find the value l with rank $|R|/2 - \sqrt{n}$ in R and the value r with rank $|R|/2 + \sqrt{n}$ in R.
3. Compare all values in I with l and r, discard all values from I except those which are strictly between l and r. Denote the items which remain by I' and let n_l be the number of values $\leq l$ which were discarded.
4. Sort I' and find the value with rank $n/2 - n_l$ in I'. If this exists, return this as μ.
   - ▷ We will show that the probability that $|I'| = O(n^{3/4})$ is very high, and so I' can be sorted in $O(n)$ operations as well.

The intuition behind this algorithm is as follows. The sample R contains $n^{3/4}$ values, and on average $n^{1/4}$ values lie between two consecutive sample values. There are $2n^{1/2}$ values between l and r in the sample R, which means that on average $2 \cdot n^{1/2} \cdot n^{1/4} = 2n^{3/4}$ elements from I lie between l and r, and so the expected size of I' is $2n^{3/4}$. Intuitively, it also seems clear that μ is likely to lie in between l and r, and so the values $\leq l$ and $\geq r$ are not likely to be μ.

We now analyze both the probability that the algorithm correctly computes μ, as well as the running time of the algorithm. Firstly we determine the probability that $l \geq \mu$ (if this happens, μ is discarded in Step 3 and the algorithm will not work correctly). Note that the number of values in R which are less than μ is binomially distributed with parameters $|R|$ and $1/2$ (each sample value has probability $1/2$ of being less than μ, independently of all other sample values—this is one advantage of sampling with replacement). Since l was chosen to be the $(|R|/2 - \sqrt{n})$-th smallest element in R, l can only be larger than μ if $|R|/2 - \sqrt{n}$ or fewer values in R are smaller than μ. Since the expected number of such values is $|R|/2$, this means that we had significantly fewer values than expected. Using the Chernoff bounds we can compute this probability as being smaller than $e^{-cn^{1/4}}$ for some constant $c > 0$. We can similarly bound the probability that $r \leq \mu$ as well as the probability that $|I'| > 4n^{3/4}$ (i.e. more than twice its expected value).

We conclude that the algorithm correctly computes μ and performs only $O(n)$ operations with probability tending to 1 as n tends to infinity.

5 Partitioning II: List Ranking

We are given a singly-linked list L of n nodes. These nodes are numbered $1, \ldots, n$, but the order along the list of these nodes is specified by an array S of size n: for $i \leq i \leq n$, the node after node i is the node $S[i]$, unless i is the last node in the list, in which case $S[i] = 0$. The *list ranking* problem is simply to label each node with its distance from the end of the list. Once ranked, a linked list can be moved into an array, allowing it to be processed more efficiently.

There is a plethora of literature on this problem, and the efforts of the research community have been aimed at obtaining algorithms which are simultaneously work- and time-optimal even on relatively weak models such as the EREW PRAM. Although some of these algorithms are quite simple in absolute terms, in comparision to the trivial sequential solution to list ranking, they are still quite complex—one needs extremely (almost trivially) simple algorithms to obtain competitive performance on parallel computers. The algorithm which we now describe has been empirically demonstrated to clearly outperform other candidates [19] (at least on vector processors like the Cray C-90 and for certain problem sizes), but does not achieve the goal of simultaneous work- and time-optimality.

If it were somehow possible to partition the list into (contiguous) equal-sized sublists of size n/k, we could form a super-list in which each contiguous sublist would be represented by its first element. We could then serially rank this super-list in $O(k)$ steps. Since distances in the super-list are n/k times smaller than distances in the original list, we could easily compute the distance of the representative element of each sub-list. Now, in parallel for each list, we could apply a serial algorithm to compute the distance for each member of the sub-list, using the representative's distance as a starting point. The second phase would have element complexity $O(n)$ and step complexity $O(n/k)$. This algorithm has element complexity $O(n)$, and overall step complexity $O(k + n/k)$, which is $O(\sqrt{n})$ if we choose $k = \sqrt{n}$.

The problem is that partitioning the list into equal-sized components of size n/k is just as hard as ranking the entire list. The first idea would then be to mark $k - 1$ random nodes in the list (choosing with replacement), and use the marked nodes to partition the list into sub-lists whose expected size is n/k, and proceeding as above. One small hitch is that the sub-lists are no longer of equal size. We therefore have to put in an initial step where, in parallel, the length of each sub-list is calculated by a serial algorithm, and modify the distance computation on the super-list to compute weighted distances (each representative being given a weight equal to the size of its sub-list). The step complexity of this algorithm is given by $O(k + L)$ where L is the expected length of the *longest* sub-list. Although the expected size of each sub-list is n/k, it can be shown that the expected value of L is $O((n \log n)/k)$. Choosing

k appropriately gives an algorithm with expected step complexity $O(\sqrt{n \log n})$ and element complexity $O(n)$.

(Running this algorithm on a scan-CRCW PRAM with $\sqrt{n/\log n}$ or fewer processors results in optimal speedup, but may involve frequent load balancing, which is essentially free on the scan-CRCW PRAM model. If needed, a modification to this algorithm can be used to eliminate the need for load balancing.)

6 Pruning II: Graph Connected Components

As with list ranking, there a lot of work has been done on algorithms for computing the connected components of an undirected graph $G = (V, E)$. This problem would be solved sequentially by using depth-first-search (DFS) or breadth-first-search (BFS) [7], both of which are extremely simple algorithms and run in $O(m + n)$ time (where $|V| = n$ and $|E| = m$). However, existing evidence seems to suggest that DFS is probably not parallelizable, and that BFS is not efficiently parallelizable. It seems, therefore, that to solve this fundamental problem efficiently in parallel, new approaches are needed.

Again, the efforts of the research community have been aimed at obtaining algorithms which are simultaneously work- and time-optimal even on relatively weak models such as the EREW PRAM, and again, these algorithms are quite complex (compared to DFS or BFS, at any rate). Empirical research [10] suggests that for moderate problem sizes, the best performance in practice is obtained by using an optimized hybrid algorithm which uses ideas from several previous algorithms [23, 1, 21]. We will refer to this algorithm as HYBRID. HYBRID has logarithmic step complexity but performs $O((m + n) \log n) = O(m \log n)$ operations[3] in the worst case and is therefore non-optimal.

We now describe a simple but clever idea due to Karger et al. [14] which allows us to use the pruning paradigm to suggest an avenue for practical improvements over HYBRID, as follows:

1. Obtain a graph $G' = (V, E')$ by choosing each edge in G independently with probability p (to be determined later).
2. Determine the connected components of G' using HYBRID. Let $F \subseteq E'$ be a spanning forest of G'.
3. Discard from G all edges which have both endpoints in the same connected component of G', except those which belong to F. Call the resulting graph G''.
4. Use HYBRID to determine the connected components of G'', which are the connected components of G.

We now analyze the step and work complexities of this algorithm. Step 1 takes $O(m)$ operations and $O(1)$ steps. Since the expected number of edges in G' is pm, the expected operation count for Step 2 is $O(pm \log n)$. The step complexity is $O(\log n)$ as before. Step 3 can be easily implemented to run in

[3] As we can always remove vertices of degree 0 we may assume that $m = \Omega(n)$.

$O(1)$ steps performing $O(m)$ operations. Adapting an argument from [14] one can show that the expected number of edges in G'' is $O(n/p)$. Hence this Step 4 performs $O((n \log n)/p)$ operations on average and has $O(\log n)$ step complexity.

Summing up the costs of the various steps we get that the step complexity of the resulting algorithm is $O(\log n)$ and the expected operation complexity is $O(m + (pm + n/p) \log n)$. If $m \leq n(\log n)^2$ the optimum choice of p gives us an algorithm which performs $O(\sqrt{mn} \log n)$ operations. As the constant factor in the big-oh for this algorithm is about twice that of HYBRID, we would expect to see improvements when $m/n \geq 4$. (For $m > n(\log n)^2$ the algorithm performs $O(m)$ operations on average, and even more marked improvements over HYBRID may be expected.)

Bibliographic Notes. There are a number of general resources for material on random sampling. One area in which random sampling has proved very fruitful (and which we do not cover here) is in parallel algorithms for computational geometry. Another traditional task for which random sampling is used is to estimate the distribution of values in a set. Further material on these topics may be found in [16, 17, 18]. Karger's recent PhD thesis [13] contains a good collection of results obtained by applying random sampling to graph problems.

Parallel quicksort appears to be folklore. The selection algorithm of [8] was parallelized by [15, 22]. For a summary of research in the 1980s and early 90s on list ranking and connectivity see [2, 9, 12, 20, 24]. For later developments see [6, 11] and the papers cited therein.

Basic and advanced material on randomized algorithms may be found in [16, 17]. In particular, Equations (3) and (4) are proved in [16].

References

1. B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. *IEEE Transactions on Computers* **36** (1987), pp. 1258–1263.
2. S. Baase. Introduction to parallel connectivity, list ranking and connectivity. In J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1993, pp. 61–114.
3. G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers* **C-38** (1989), pp. 1526–1538.
4. G. E. Blelloch. *Vector Models for Data-Parallel Computing*, MIT Press, 1990.
5. G. E. Blelloch. NESL: A nested data-parallel language (version 3.0). Technical Report CS-CMU-94-112, Carnegie-Mellon University, 1994.
6. R. Cole, P. N. Klein and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proc. Sympsosium of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, 1996, pp. 243–250.
7. T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.
8. R. W. Floyd and R. L. Rivest. Expected time bounds for selection. *Communications of the ACM* **18** (1975), pp. 165–172.

9. H. Gazit. Randomized parallel connectivity. In J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1993, pp. 197–214.
10. J. Greiner. A comparison of parallel algorithms for connected components. In *Proc. Sympsosium of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, 1994, pp. 16–23.
11. S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic connectivity problems. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 1996, pp. 438–447.
12. J. JáJá. *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
13. D. Karger. *Random sampling in graph optimization problems.* Ph.D. Thesis, Department of Computer Science, Stanford University, 1995.
14. D. Karger, P. N. Klein and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM* **42** (1995), pp. 321–328.
15. N. Megiddo. Parallel algorithms for finding the maximum and median almost surely in constant time. Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, 1982.
16. R. Motwani and P. Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.
17. K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice-Hall, 1994.
18. S. Rajasekaran and S. Sen. Random sampling techniques and parallel algorithm design. In J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1993, pp. 411–451.
19. M. Reid-Miller. List ranking and list scan on the Cray C-90. In *Proc. Sympsosium of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, 1994, pp. 104–113.
20. M. Reid-Miller, G. Miller and F. Modugno. In J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1993, pp. 115–194.
21. J. H. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR-08-85, Harvard University, 1985.
22. R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, **14** (1985), pp. 396–409.
23. Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms* **3** (1981), pp. 57–67.
24. U. Vishkin. Advanced parallel prefix-sums, list ranking and connectivity. In J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1993, pp. 341–406.

A Chernoff Bounds

Let X be a binomial random variable with parameters m and p, and hence $E(X) = mp$. Then, for any fixed ϵ, $0 \leq \epsilon \leq 1$, we have:

$$\Pr[X \geq (1+\epsilon)\mathrm{E}(X)] \leq e^{-\epsilon^2 \mathrm{E}(X)/3} \tag{3}$$

$$\Pr[X \leq (1-\epsilon)\mathrm{E}(X)] \leq e^{-\epsilon^2 \mathrm{E}(X)/2} \tag{4}$$