Algorithmic Foundation for Benchmarking of Computational Platforms Running Asymmetric Cryptosystems

Mikhail A. Kupriyashin[1] and Georgii I. Borzunov[1,2]

[1] National Research Nuclear University "MEPhI" (Moscow Engineering Physics Institute), Moscow, Russia
kmickle@yandex.ru
[2] Russian State University of A. N. Kosygin (Technology. Design. Art), Moscow, Russia
borzunov_g@mail.ru

Abstract
General purpose benchmarks do not yield accurate performance estimates for special tasks. In this paper we consider implementation of exact algorithms for the 0-1 Knapsack Problem in order to determine the performance of parallel computation platforms intended for running or performing analysis on asymmetric ciphersystems. We study some features of exact parallel algorithms for the Knapsack Problem, as well as load balancing techniques for them. We propose an algorithmic foundation for computational platform benchmarking aimed at getting accurate performance estimates for these platforms.

Keywords: parallel computation, benchmarking, asymmetric ciphersystems, Knapsack Problem

1 Introduction

The 0-1 Knapsack Problem is a well known NP-complete problem having a wide range of applications in real life. In particular, the Knapsack Problem is used as a core component in some asymmetric cipher systems[7]. In this case, the highly developed base of approximate algorithms for the Knapsack Problem is inapplicable, as only the exact solutions make sense. Performance of both implementation and analysis of such systems heavily depend on knapsack packing performance. Therefore executing exact algorithms for the Knapsack Problem on a chosen computational platform yields a good estimate of the platform's performance.

This estimate is ought to be more accurate than the estimates obtained through the established general purpose tests, such as High Performance Linpack (HPL)[3], HPCG[2] and Graph500 Benchmark[1]. In particular, HPL and HPCG are built around floating-point computation, whilst Graph500 is bandwidth-oriented. The Knapsack Problem, in turn, involves a

lot of large positive integer computation with low intensity of interprocessor communication. Thus, a new benchmarking technique is required to obtain reliable performance estimates. These estimates apply whenever the evaluated computational platform is used to solve tasks involving Knapsack cipher systems implementations.

The technique in question requires an algorithmic foundation composed of a number of exact algorithms for the Knapsack problem. The following approaches to the Knapsack Problem are known: exhaustive search, packing tree traversal, list merging and dynamic programming.

In this paper, the Knapsack problem is defined as follows: given a numbered set of n items with positive integer weights $(a_1; \ldots; a_n)$ find subsets of these items with net weight exactly equal to a predefined number w. A given instance of the Knapsack Problem may have one or several solutions, as well as no solution at all. Depending on the situation it may be necessary to determine all the solutions or just one random solution.

2 Known Knapsack Problem Solution Methods

Using exhaustive search is a straightforward approach to the problem [6]. All the possible subsets of items for the given Knapsack Problem instance are enumerated and net weight is calculated. Whenever the net weight value becomes equal to the predefined weight, the corresponding subset becomes known to be a solution of the current instance. There exists a total of 2^n feasible subsets of n items, so it takes 2^n steps to enumerate. Besides, for each of the vectors the net weight is calculated. That takes n binary comparison operations and in $\frac{n}{2}$ additions in average. Additionally, the net weight is then compared to the predefined value. So, the complexity is $2^n \cdot (1 + \frac{n}{2})$ comparison operations to find all the solution of the given instance. This corresponds to the $O(n \cdot 2^n)$ complexity class. High computational complexity is offset by low memory requirements: at any given moment of time only the current subset of items (n bits) and its weight (up to $log_2(n \cdot a_{max})$ bits) is to be stored, along with the instance of the problem itself (n values up to $log_2(a_{max})$ bits each) and the target weight w value up to $log_2(n \cdot a_{max})$ bits. The sum of the aforementioned corresponds to the $O(n)$ complexity class (here a_{max} is the maximal weight across the item set. It is considered constant). In fact, problem instances with task sizes over 200 may be solved without any memory allocation problems even on memory-constrained platforms. One more advantage of this method is near-perfect scalability. The set of all possible item sets may be split into even parts and distributed between the worker nodes. The subset enumeration algorithm means, though. For instance, it has been shown that splitting the lexicographic sequence of subsets into equal parts does not ensure uniform workload distribution[11].

The Knapsack Problem may also be solved by means of building and travesing the packing search tree[10]. All the possible subsets of items are arranged to form a tree, so that the weight of each node is less than the weight of all its child nodes. As a result, whenever the weight of the noe is greater than the target value w, further search in the branch is guaranteed to yield no result. Thus, the amount of subsets to be checked may be greatly reduced. Experiments show that the number of sets to check depends on the average item weight, as well as on target weight w. Regardless of the average value, when the value of w is $\left(\frac{n \cdot a_{max}}{2}\right)$, it is only required to check around 50% of all the subsets. When $a_i = 2^i$, the dependence between w and complexity reduction is close to linear. Experiment graphs on Fig. 1 illustrate the amount of reduction under different conditions[5]. Despite considerable complexity reduction in comparison to the exhaustve search, the time complexity fot the algorithm stays exponential. The complexity class is also $O(n \cdot 2^n)$ as the amount of subsets still grows exponentially and the subset weighing procedure stays the same. The memory complexity is higher, as the node stack

must be maintained during the traversal. The stack must support storing $\left(\frac{n(n-1)}{2}\right)$ subsets, each n bits long. Therefore, space complexity class escalates to $O(n^3)$ this is still feasible for memory-constrained computational platforms at large task sizes.

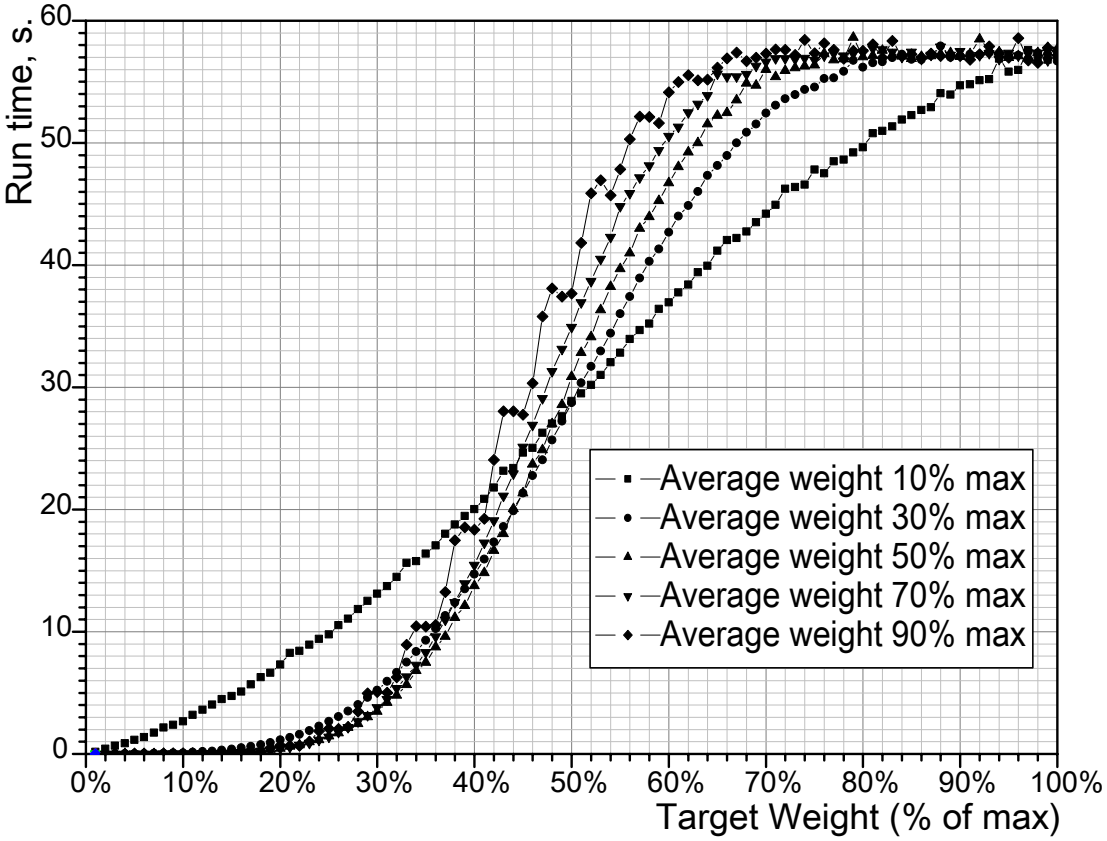

Figure 1: Dependence between exhaustive search run time and some parameters of the Knapsack Problem instance

Structure of the packing tree, built as described in [10], makes payload distribution difficult. Splitting the tree close to the root yields different branches, every other branch is 2 times as large as the previous. If the tree is split near the leaves, complexity reduction is lost, as the nodes farther from the root are supposed to be ignored as having the largest net weights. A linearization method has been proposed that allows to split the search tree into (initially) equal parts while maintaining the traversal order. The method was described in [8]. Using this method provides for decent quality workload distribution (experiments never showed parallel computation efficiency fall below 50%. Also, the method allows to construct the tree dynamically, without the need for node stack. This additionally reduces space complexity to $O(n)$. Splitting the tree into parts causes a decay in subset number reduction (starting node for some subtask may be located in the area that should have been ignored during sequential execution), but this decay is negligible.

The list merge method, first described by E. Horowitz and S. Sahni in [4], is based on splitting the original knapsack into two sub-knapsacks, each containing half of the original item set.

Two lists are built for these sub-knapsacks, each containing all the possible subsets and the corresponding net weights. The resulting lists are sorted by net weight. Then, net weights for each pair of list items (one from the first list, one from the second) have their total weight compared with the target weight w. First element of the first list is being compared with all the items of the second list, until total weight of the list items exceeds the target weight. The next element from the lirst list is then taken and compared with all the items in the second list again. As with the packing tree search, some of the combinations are omitted, as they are guaranteed to yield no solutions. Building the lists takes $O\left(n \cdot 2^{\frac{n}{2}}\right)$ comparisons for each of the lists. Merginig takse from $O(2^{\frac{n}{2}})$ comparisons in the best case (when all the items on the second list turn to be too large compared to the target weight) up to $O(2^n)$ comparisons in the worst case (when no reduction occurs). Thus, the Horowitz-Sahni algorithm is a chance to reduce the computational complexity of the solution to $O(2^{\frac{n}{2}})$. Fig. 2 shows the time compexity for the algorithm under various conditions (compare with Fig. 1). The benefit in computational complexity comes at the cost of space complexity. Both lists are to be stored and then sorted. Each list contains $(2^{\frac{n}{2}})$ entries, $(n + log_2(n \cdot a_{max}))$ bits each; this corresponds to the $O\left(2^{\frac{n}{2}}\right)$ complexity class. This severely limits the algorithm applicability at task sizes over 50.

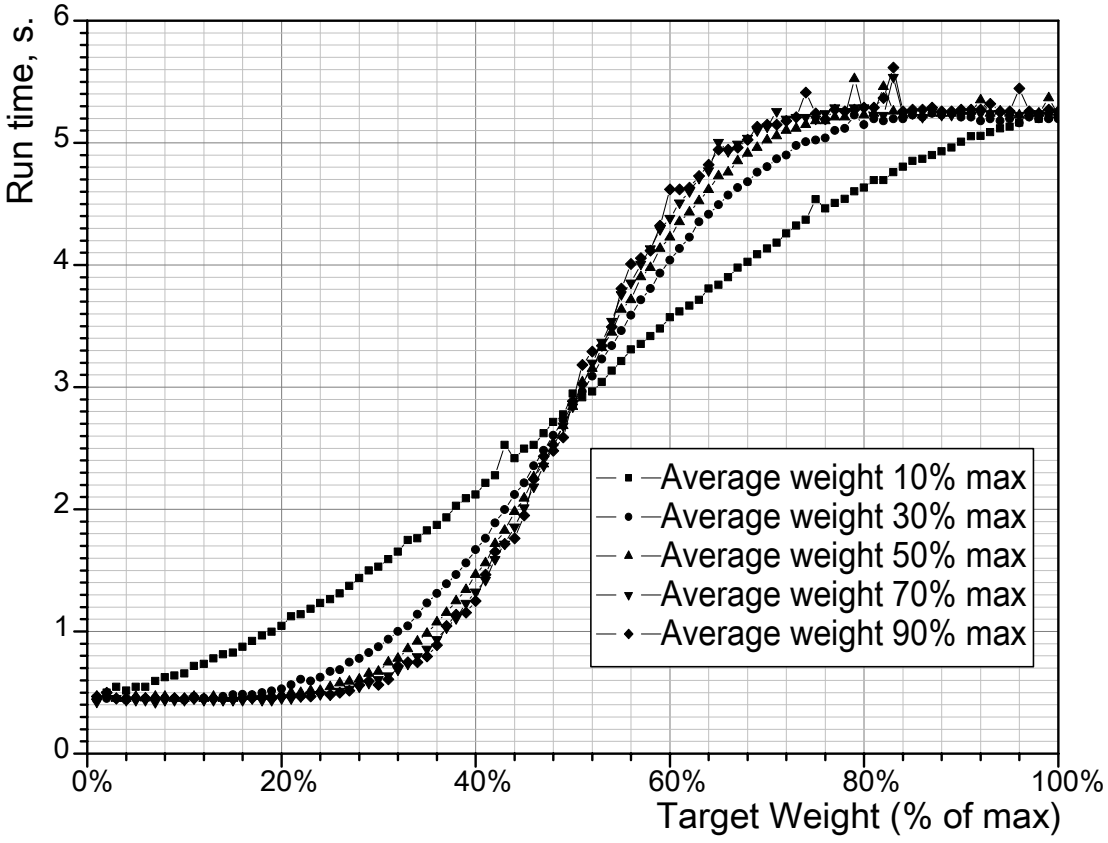

Figure 2: Dependence between packing tree traversal run time and some parameters of the Knapsack Problem instance

The dynamic programming approach[9] is quite different from the three methods described above. A table with n rows and w columns is filled, each cell $(i; j)$ showing the maximal sum not

exceeding j that may be obtained using the first i items of the subset. Every next row may be filled based upon the values in the previous row. When the table is filled up, n reverse steps are needed to construct the solution subset based upon the table values. Filling the table takes $n \cdot w$ steps, also n reverse steps are needed. This corresponds to the $O(n \cdot w)$ complexity class. Despite being linear in n, this value tends to be exceptionally large, as w is typically a large integer. Still, if w is small, dynamic programming allows to solve an instance of the Knapsack Problem in linear time. Space complexity class is the same as computational complexity. Although fast, this method is rarely applicable. In crptographic applications, knapsacks tend to have low density $D = \left(\frac{n}{log_2(a_{max})}\right)$ (around 1.0; lower values are insecure due to the polynomial-time lattice reduction attacks, however). This rockets the space complexity to $O(n^2 \cdot 2^n)$. It is still important to keep the dynamic programming approach in mind when dense knapsacks are in question. It should also be noted that only one solution may be obtained via dynamic programming.

3 Forming the Algorithmic Foundation

In the previous section, several exact algorithms for the Knapsack Problem were considered. Here we discuss whether or not these algorithms should compose the foundation for the computational platfom benchmarking.

First of all, the exhaustive search is important. Almost 100% scalability may make it a good choice for parallel computational platforms with lots of processors. 50% subset number reduction offered by the packing tree search algorithms may be offset by poor parallel computation efficieny. Still, the latter algorithm is expected to be faster in most cases. The second important reason to include the exhaustive search algorithm is that it gives the best estimate on how long it takes to weigh a single subset. As weighing subsets is what knapsack cipher systems implementations do in order to encrypt a block of data, the performance of exhaustive search is tightly connected, whereas the other algorithms are only useful in analysis of cipher systems.

With the linearization method included, wh expect the packing search tree traversal algorithm to be the most efficient in general case. While maintaining low space complexity, it has a computational complexity advantage over the exhaustive search in most cases.

The Horowitz-Sahni list merging algorithm is less interesting. Despite the lower complexity class, its space complexity makes it unfeasible under harsh conditions set forth by asymmetric knapsack cipher systems. At task sizes around 100 the memory requirements are almost sure to exceed the capablities of existing computational platforms, but this task size is still quite small for cryptographic needs. Still, it is likey faster than the algorithms discussed above, and extensively tests memory performance as well as integer computation performance, which makes its performance an interesting indicator.

The dynamic programming method is the least promising of the studied algorithms. Applicable only at unique conditions it is unlikely to be useful unless exceptionally dense knapsacks are in question. An implementation of this method is memory-intensive, but the list merge algorithm considered above shares this feature, too.

4 Summary

In this paper we considered several exact algorithms for the Knapsack Problem and assessed their applicability to benchmarking computational platfoms running asymmetric ciher systems-

related tasks. We believe that the exhaustive search algorithm should be implemented to measure the potential knapsack cipher systems performance. The packing tree traversal algorithm with linearization is good to assess the knapsack cipher systems analysis performance. Horowitz-Sahni algorithm may serve this purpose too, evaluating memory performance as well as integer computation performance. The dynamic programming approach in not recommended for the algorithmic foundation, as its applicability is too low.

5 Acknowledgements

This work was supported by Competitiveness Growth Program of the Federal Autonomous Educational Institution of Higher Professional Education National Research Nuclear University MEPhI (Moscow Engineering Physics Institute).

References

[1] Brief introduction | graph 500. url: http://www.graph500.org/.

[2] HPCG Benchmark. URL: http://hpcg-benchmark.org/.

[3] HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. url: http://www.netlib.org/benchmark/hpl/.

[4] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. 21(2):277–292.

[5] Mikhail A. Kupriyashin. Research on using trees to solve the knapsack problem by means of parallel computation. In *Supercomputing Days in Russia: Conference Proceedings*, pages 626–631. MSU Publishing.

[6] Mikhail A. Kupriyashin and Georgii I. Borzunov. Analiz i sravneniye algoritmov nahozhdeniya tochnogo reshenia zadachi o ruykzake (analysis and comparison of exact algorithms for the knapsack problem; in russian).

[7] Mikhail A. Kupriyashin and Georgii I. Borzunov. Analiz sostoyania rabot, napravlennykh na povysheniye sty=oykosti shifrosistem na osnov ezadachi o ruykzake (analysis of papers on increasing the knapsack cryptosystems security; in russian). (1):111–112.

[8] Mikhail A. Kupriyashin and Georgii I. Borzunov. Computational load balancing algorithm for parallel knapsack packing tree traversal. 88:330–335.

[9] Mikhail A. Kupriyashin and Georgii I. Borzunov. Issledovanie algoritma tochnogo resheniya zadachi o ruykzake metodom dinamicheskogo programmirovaniya (finding the exact solutions of the knapsack problem using dynamic programming; in russian). (17):121–130.

[10] Mikhail A. Kupriyashin and Georgii I. Borzunov. Sokrasheniye vremennoy slozhnosti bazovogo algoritma resheniya zadachi o ruykzake (reducing the computational complexity of solving the knapsack problem; in russian). (1):66–67.

[11] Mikhail Andreevych Kupriyashin and Georgii Ivanovich Borzunov. Visualization and analysis of the exact algorithm for knapsack problem based on exhaustive search. 7(4):87–100.