

# Towards a GridMOSI Library

Alin Suci, and Rodica Potolea, *Member, IEEE*

**Abstract** — In the last decade, problems that were computationally unfeasible, due to lack of computing power and storage capabilities, are becoming solvable, through an expansion of the problem solving architectures to much larger scales. Applications that are expensive in resources (either memory or time consuming, or both) are far beyond the capabilities of a regular computer. They are being approached again, from a different point of view: their study is oriented towards resource sharing technologies – grid and cluster computing. While cluster computing is using very expensive multi-processor machines, grid computing is establishing itself as the 'de facto' standard for solving computationally or data intensive problems. Grids use a large structure of computing resources, connected by a network (the internet), in order to solve large-scale computation problems. The grid computing model is making a better use of distributed resources, put them together in order to achieve higher throughput and be able to tackle large scale computation problems. It also offers better means of sharing research results. This paper summarizes the results obtained so far on several sets of benchmarks: optimal interval for the size of the read buffer, optimal grid implementation of various algorithms, and the particularities of implementing cryptographic and cryptanalytic algorithms for the grid\*.

**Keywords** —grid, benchmarks, optimality, cryptography.

## I. INTRODUCTION

From a physical point of view, a grid may be considered as a series of interconnected nodes called computing elements or simply CEs. These nodes, in the mesh topology that they form, communicate among themselves at high data rates. The communication lines form the backbone of the grid. Each CE has under its administration a number of computers which play the role of the worker (according to the master-worker model). These computers are called worker nodes (WNs). They represent the raw computing power resource behind the grid concept. Intuitively the greater the number of WNs a CE has, the more computing power it provides to the participating grid.

These WNs are not designed to be accessed publicly by any user, nor they are individually accessible from a remote site. They are meant to be administrated solely by the CE to which they subscribe. Even though physically it is possible to assign these WNs other duties (such as laboratory equipment in an university) it is not recommended to do so. Their only role is defined in the context of a grid.

Each WN of our grid (GridMOSI) has a Pentium IV class processor, at 3GHz, with 1GB of RAM and 160 GB of secondary storage, running Scientific Linux 3.0.8, gcc 3.2.3 and g-lite 3.0.2.

Here we propose a study for the implementation of various algorithms on the grid. Section II is dedicated in finding the optimal size of the read buffer for grid applications, and we performed a series of benchmarks using our local grid (GridMOSI). There is no similar study to our knowledge, although the buffer size was extensively studied in the context of network communication, especially for the TCP/IP protocol.

Section III contains a description of several algorithms we implemented in our Gridmosi Library, focusing on the particularities of each of the decisions taken, and the corresponding experimental results. In section IV we describe the adapting the cryptographic and cryptanalytic algorithms for grid-based execution, and the identification of those algorithms for which performance increases are possible by exploiting data parallelism. Several benchmarks were carried out on our local grid (GridMOSI) and the results are presented here.

## II. OPTIMAL READ BUFFER SIZE FOR GRID APPLICATIONS

We developed a test application that reads chunks of data having various sizes from a file. In order to determine an optimal read buffer size for grid applications we considered both methods for data access [8].

### A. Reading from the Storage Element

Initially the test file had to be copied on the SE. This action can be done only by a certificated user as in the case of job submission, as authentication is required for file transfer. This restriction has imposed the introduction of a transfer protocol (gridftp) that is based on the general file transfer protocol and in addition checks the needed credentials.

For the first phase of our tests, we chose medium size files and accessed them by using the storage element. Initially the buffer's size was considered to be measured in bytes, in steps of one size order, just to draw a rough conclusion.

As it can be seen from Fig. 1, a minimum can be spotted in the vicinity of value 100,000. After this value, the read time increases in the proximity of 500 milliseconds.

\* This work was funded by the Romanian Ministry for Education, Research and Youth, through project 95/03.10.2005 - GridMOSI

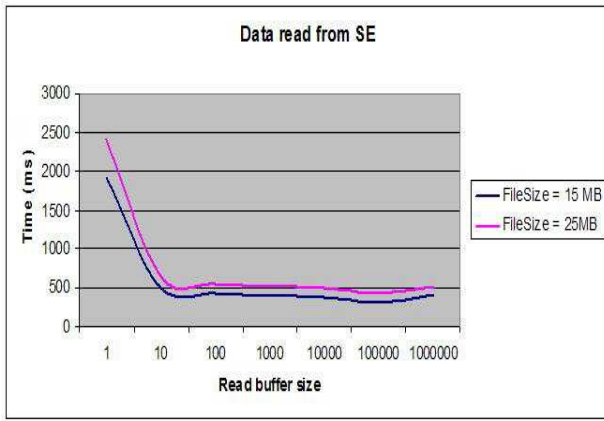


Figure 1. Variation of time needed to read a file depending on the read buffer size

The next step in our attempt to determine the buffer size for which the optimal performance is achieved was to refine the previous test. For this second test we made two changes: the buffer size was considered in increments of 10,000 and in the range [10,000; 500,000], and also we performed our test on larger files to have a better view over the actual read process.

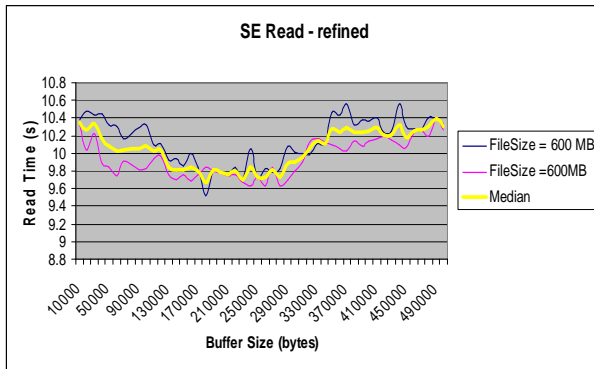


Figure 2. SE read test for larger files

The test results on larger files are influenced by the network traffic at the moment of the test. This interference is due to the nature of the actual read process when this is performed on the SE. The general process when submitting a job to the grid scheduler is: the manager identifies a CE that can satisfy the job requirements, assigns the job to the CE manager. The CE then identifies the free WNs which may start executing the job and delivers them the executable files.

Even though the results are influenced by network traffic, a general trend may be observed from the graphical representation (Fig. 2), which may be roughly viewed in a parabolic shape, with the minimum situated in the interval [150,000; 300,000].

### B. Local read

This is the second method for data access in a grid environment. The input files are not remotely placed on the SE, they are copied locally, on each WN local disk. In case the input data is of significant amount (hundreds of MB, or several GB) it is not advisable to store multiple replicas, a

lot of storage space would be wasted. From the tests we run we obtained (as can be seen from Figure 3) that the best response times are obtained with a buffer size of approximately 200,000 bytes. Due to the fact that file size is limited, the refinement process, applied to the read from SE method, is not applicable here. In its essence the local read mechanism on the grid WNs, is the same as any local read on a standalone station. This analogy has allowed us to extend the refinement process for the local read to a standalone computer. The general behavior observed when using smaller files is maintained when using larger files. There is a massive improvement until buffer reaches size 50,000 and then the read time remains at a fairly constant value in the interval [130,000; 290,000].

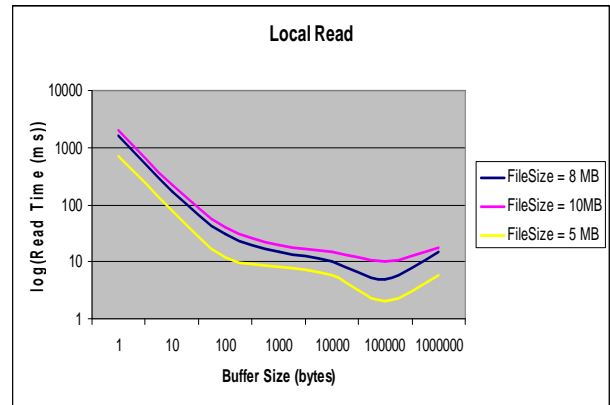


Figure 3 Time variation for local read

## III. GRIDMOSI LIBRARY

### A. Basic processing issues

One basic step in obtaining an implementation on a grid is represented by the parallelization. There are two different approaches. Data parallel is the approach in which input data should be split to be processed by different CPUs. For the control parallel approach different sequences are processed by different CPUs, and for this the code should be parallelized. This is not a trivial task as conflicts should be avoided.

The approach requires observing the constraints within the data and sequential code. Constraints refer to precedence relations of computation to be satisfied in order to solve the problem correctly. Read/Write conflicts define dependencies that limit the parallelism potential. As some types of constraints can be removed (antidependencies and output dependencies), we analyzed the sequential code and transformed it such that to increase the parallelism. On the other hand data flow dependence indicating write before read ordering of operations which must be satisfied can not be avoided. We had to find different approaches for cases where too many data flow constrains does not allow us to move to the parallel implementation from the sequential one.

### B. Experimental results on benchmarks performed

We have started by benchmarking several algorithms from various classes (from linear to exponential) in order to determine the relation between the dimension of input

data and the optimum number of WNs to solve the given problem [6]. The dense matrix algorithms, graph algorithms, sorting and numerical algorithms were tested on several grid nodes with up to 36 WNs, while the cryptographic algorithms and statistical tests for randomness were tested on a grid node with 10 WNs. This approach was used in order to study the behaviour of not just the algorithms but the behaviour of the grid nodes as well. Each of the working nodes are Pentium 4 computers, at 3 GHz, with 1 GB RAM, running Scientific Linux 3.0.6 CERN and the LCG2 middleware.

For matrix algorithms we have implemented and tested matrix multiplication with data parallel approach (for which 1-D Partitioning and 2-D Partitioning has been carried out), matrix power and Gaussian elimination (with 1D implementation and cyclic 1-D mapping). For all of them the theoretical shape proved to be correct: as the number of processors involved in computation increases, the running time decreases, until the minimum is obtained. If the number of processors is increased more, the running time is increasing as well. The explanation is that after reaching minimum, the communication component of the running time tends to increase, thus altering the overall running time.

However, after the first set of algorithms has been tested, we observed two important outcomes. The overall running time improves (if compared to the sequential implementation) to up to one order of magnitude when the input size increases, as can be seen from the relative analysis depicted in the figure below. Here, on the abscissa, we have represented the number of processors, while on the ordinate the time chosen as the ratio between the parallel and sequential implementation of the same algorithm.

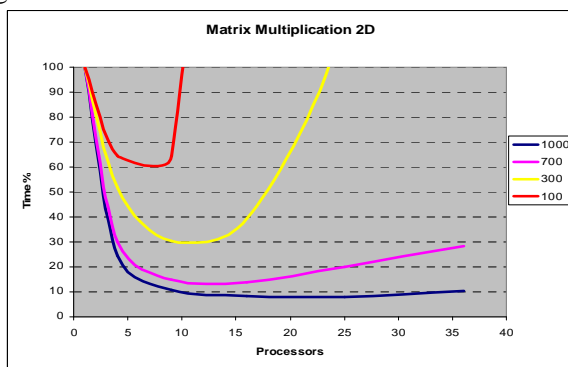


Figure 4 Performance improvements on GRID for Matrix Multiplication

For the QuickSort algorithm we have chosen a data parallel approach (data split evenly among the processors), and we have obtained basically the same outcomes. Moreover, it becomes obvious that larger the data, better the improvement comparing to the sequential implementation. While for a dimension of  $10^5$  the time is 40% from the sequential one, for  $10^6$  the time is 20%, and for  $3 \cdot 10^6$  the time is 10%, ensuring an increase of the performance with one order of magnitude. For the enumeration sort this becomes even more obvious, as the

performance increase is almost two orders of magnitudes (parallel running time represents 1.7% of the mono processor running time) for the optimum number of processors (32) and large data ( $5 \cdot 10^5$ ).

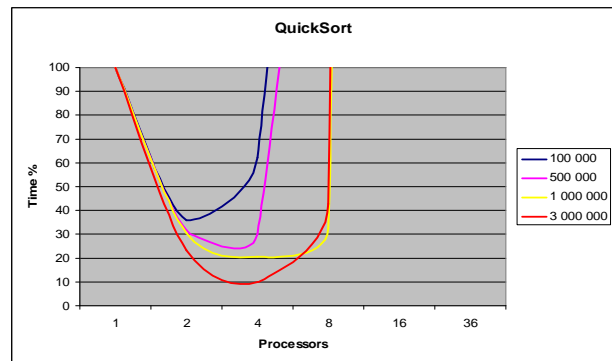


Figure 5 Performance improvements on GRID for QuickSort

For graph algorithms we have implemented and tested the Minimum Spanning Trees problem (Prim's algorithm, data parallel approach) and Transitive Closure problem. For the later one, we have tested two approaches. The source-partitioned approach requires no inter process communication, and is using data partitioning (the set of vertices is partitioned and each vertex belongs to a set which is assigned to a process. Each process computes shortest path from each vertex in its set to any other vertex in the graph, by using the sequential algorithm.). The source-parallel approach requires inter process communication and uses the parallel formulation of the single-source algorithm.

As can be seen from the Fig. 6, the shapes of the overall running time have particular forms and do not follow the theoretical one. The explanation is that besides the initial distribution of data, during the execution, the WN are not interchanging data.

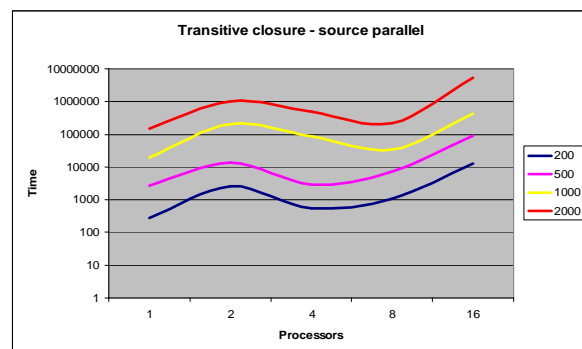


Figure 6 A typical shape for non-communicating algorithms implemented on GRID

Graph Isomorphism is an exponential running time problem. The trivial solution of the problem requires  $n!$  matches to be performed for which the parallel approach seems to be a very good choice. In the parallel approach each process is responsible for computing  $(n-1)!$ . Due to the fact that there is no communication among processes at all, the response time of the tests decreases as the number of processors increases, with the best time obtained for the algorithm being run on the maximum processors available.

#### IV. CRYPTOGRAPHIC AND CRYPTANALITIC ALGORITHMS FOR THE GRID

##### A. Execution modes

The classical execution scenario is that the user submits the job for grid execution, waits until the job is done (by checking the job status repeatedly) and retrieves the results of execution (the result files).

In the context of our work [7], we discovered that this simple execution mode can be extended in several ways in order to make it more suitable for our cryptographic algorithms, and also permit the increase in performance by parallelization.

By adapting Flynn's taxonomy to the particularities of the grid execution model, we were able to identify the following execution modes:

**1. SPSD/L** – [Single Program Single Data / Local] – is the "classical" execution mode of a program, by simply calling the executable, in the UI node environment. This mode is obviously very useful for testing a program locally, before sending it to the grid for execution on a larger scale.

**2. SPSD/G** – [Single Program Single Data / Grid] – is the "classical" execution mode for the grid, mentioned above. It is based on the JDL and on several commands provided by the grid system, which allow the submission, tracking and management of jobs.

**3. SPMD/G** – [Single Program Multiple Data / Grid] – allows the execution of the same program on multiple input data sets (usually one or several files). It repeatedly invokes the specified program for each of the input data set, saving a lot of time for the user. Example: encrypting several files with the same encryption algorithm.

**4. SPMD/G/DP** – [Single Program Multiple Data / Grid / Data Parallel] – allows the execution of the same program in data parallel mode on the grid, by applying the same program on independent data chunks from the input data. By this type of parallelization important performance gains can be obtained. Example: AES encryption of a large file is performed in parallel on several equal parts of the file and the results are appended forming the final result.

**5. MPSD/G** – [Multiple Program Single Data / Grid] – allows the execution of several different programs on the same input data. Each program executes independently of the others, by distribution on the grid nodes. Example: encrypting the same file with several encryption algorithms.

**6. MPSD/G/DP** – [Multiple Program Single Data / Grid / Data Parallel] – is similar to the previous execution mode, but in this case each program exploits data parallelism, and executes in SPMD/G/DP mode.

**7. MPMD/G** – [Multiple Program Multiple Data / Grid] – allows the execution of several programs on several input data sets, by creating a job description file for each pair of <program, input data>. Example: encrypting several files with several encryption algorithms for each of them.

**8. MPMD/G/DP** – [Multiple Program Single Data / Grid / Data Parallel] – same as previous mode but each

program executes in SPMD/G/DP mode, exploiting the data parallelism.

##### B. Experimental results

From the category of cryptographic algorithms (block ciphers) we have implemented the five finalists of the AES contest [14]: Mars, RC6, Rijndael, Serpent, Twofish. Fig. 7 shows the execution time (encryption) for each of these algorithms in the SPSD/L mode. As expected, Rijndael (the winner) is fastest.

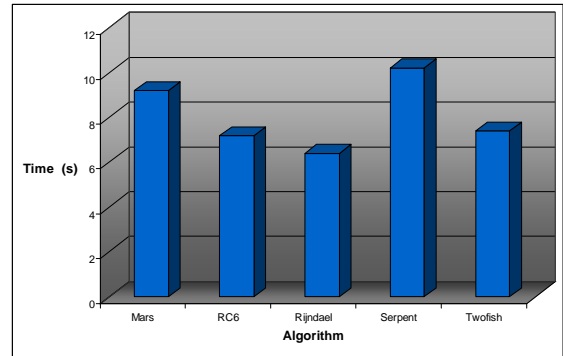


Figure 7. Block ciphers, compared

As we expect, the performance increases by using several nodes for execution. Using the data parallelization mode SPMD/G/DP, we run Rijndael on the same input file on 1, 2, 4, 8, 16, 32, 64 and 128 working nodes and the results are shown in Fig. 8. To be noticed that the theoretical parabolic shape is obtained, as in most of the tests presented in section III.

Stream ciphers (of which we implemented RC4) do not exhibit any data parallelism, being sequential in nature. This makes them less suitable for grid execution, due to the impossibility of parallelization.

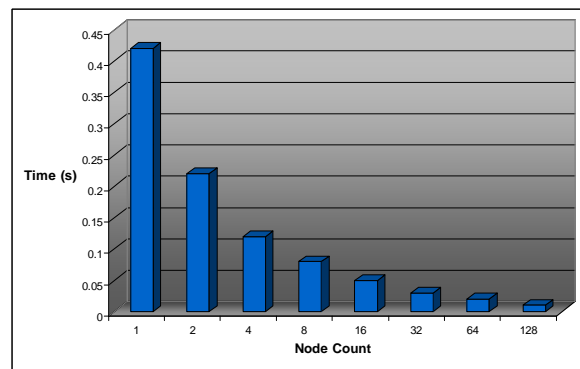


Figure 8. Rijndael execution in data parallel mode

A time consuming operation for public key algorithms (RSA in our case [15]) is the key generation. The longer the key, the longer the time needed to generate it. The time grows exponentially with the key size. Fortunately the key generation procedure is suitable for parallelization.

Hash functions are sequential in nature and therefore not suitable for parallelization. However they are very useful in cryptographic applications and we implemented both SHA-1 and SHA-2 (256, 512).

We also implemented several random number generators and randomness tests as part of our cryptographic algorithm suite [17]. Having good random numbers is a crucial part of any cryptographic system.

Factorization aims to find the prime factors of a large integer and is a well known hard problem. The security of the RSA algorithm is based on this [15]. The time needed for factoring, even using the best known algorithms increase very steeply, as Figure 9 shows.

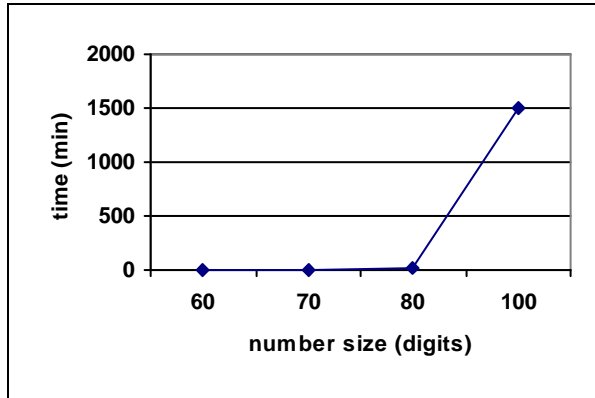


Figure 9. Factoring large integers

For factorization we choose two of the best factoring algorithms: Quadratic Sieve (QS) and General Number Field Sieve (GNFS). Fortunately we found ways of parallelizing them on the grid, reducing by 60% the factoring time of a 100 digits integer.

## V. CONCLUSION

In this paper we have presented the benefits of grid computing for several classes of algorithms, the increased performance it brings to the computing community, with focus on the GridMOSI Library. For the moment our library contains a set of algorithms implemented which are available to be run by the users. The goal of our study was to determine the optimal number of working nodes, for each of the algorithms tested and for different input sizes, in order to achieve optimal performance for the grid-based execution. Regarding that the optimal number of WNs to be considered when running a given algorithm, the conclusions split the algorithms set into two. For implementations which require inter-process communication is as follows:

- ✓ The optimal number of nodes depends on the input size. While the size increases, the optimal value shifts to the right.
- ✓ Because of the communication component, after reaching the optimal value, the running time tends to increase.
- ✓ Except for cases that do not require inter process communication most of the shapes (corresponding for the total processing time) follow the theoretical aspect.
- ✓ The speedup depends on the input size of the problem. For most algorithms, for the optimal

number of WNs, it is about 10, the best being achieved for large data. The best improvement we got was of almost two orders of magnitude (about 2% for a particular problem), which represents a speedup of 50.

For implementations which do not require inter-process communication the remarks are different. As there is no (or very small) communication component, the overall running time depends exclusively on the computation time. This makes the shape of the running time curve a decreasing one as the number of WNs is increasing. Hence it looks like increasing the WNs available will lead to a continuous decrease of the time. This is not completely true. A saturation was observed (for most algorithms), or an uneven behaviour (as for Transitive Closure source parallel approach).

We have also analyzed the two methods for data access in a grid environment. Based on the results obtained we have reached the conclusion that the optimal read buffer size is situated in the interval [130,000; 290,000] for local read method, and [150,000; 300,000] for the SE read method. In order to make the design of an application uniformly, to have the same access parameters in both cases, we propose a buffer size of 200,000 bytes. This value was chosen so that to be at a relative equal distance from the limits of both intervals, and also to have a greater recall factor.

As for cryptographic class of algorithms, seven classes of algorithms were studied, and eight execution modes for the grid were identified. Algorithms in each class were implemented and their execution times were measured. The experimental results show an important increase in performance for several classes of algorithms (most notably block ciphers and factorization), by using the data parallel execution modes. Although useful for grid based cryptographic applications, algorithms like stream ciphers and hash functions are inherently sequential and cannot benefit from the parallel potential of the grid.

## REFERENCES

- [1] I. Foster, C. Kesselman and S. Tuecke, "The Anatomy of the Grid", *International J. Supercomputer Applications*, 15(3), 2001.
- [2] I. Foster, "The Grid: A New Infrastructure for 21st Century Science", *Physics Today*, 55(2): 42-47, February 2002.
- [3] I. Foster and C. Kesselman, "The Grid: Blueprint for a New Computing Infrastructure", Morgan Kaufmann Publishers, 1st edition, 1998.
- [4] L. Ferreira and V. Berstis, "Fundamentals of Grid Computing", IBM Redpaper, 12 November 2002, IBM Form Number REDP-3613-00.
- [5] B. W. Kernighan, D. M. Ritchie. "The C Programming Language, Second Edition". Prentice Hall, Inc., 1988, ISBN 0-13-110362-8.
- [6] R. Potolea, A. Suci, A. Mascasan. "Benchmarking the GridMOSI Library". *Echallenges Conference*, 2007, The Hague, Netherlands, (accepted).
- [7] A. Suci, R. Potolea. *Cryptographic and Cryptanalytic Algorithms for Grid Applications*, ICCP 2007, Workshop on Grid Computing, Cluj-Napoca, 2007.
- [8] R. Potolea, A. Suci. *Finding the Optimal Read Buffer Size for Grid Applications*, SYNASC 2007, GridCAD workshop, Rimisoara, 2007.
- [9] G. Marsaglia, *Seeds for random number generators*, May 2003, *Communications of the ACM*, Volume 46 Issue 5.

- [10] P. L'Ecuyer, Software for uniform random number generation: distinguishing the good and the bad, December 2001, Proceedings of the 33rd conference on Winter simulation.
- [11] F. Panneton, P. L'ecuyer, On the xorshift random number generators, October 2005, ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 15 Issue 4.
- [12] C. Jermaine, A. Pol, S. Arumugam, Online maintenance of very large random samples, June 2004, Proceedings of the 2004 ACM SIGMOD international conference on Management of data.
- [13] S.-Ju Kim, K. Umeno, and A. Hasegawa, Corrections of the NIST Statistical Test Suite for Randomness, Cryptology ePrint Archive, Report 2004/018, 2004.
- [14] Advanced Encryption Standard homepage, <http://csrc.nist.gov/CryptoToolkit/aes/rijndael/>
- [15] RSA Laboratories homepage, <http://www.rsa.com/rsalabs/>
- [16] ENT A Pseudorandom Number Sequence Test Program homepage - <http://www.fourmilab.ch/random/>.
- [17] NIST Statistical Tests for Randomness homepage, <http://csrc.nist.gov/rng/>.
- [18] Diehard Battery of Tests of Randomness homepage, <http://stat.fsu.edu/pub/diehard/>.