Expanding the Knowledge Economy: Issues, Applications, Case Studies
Paul Cunningham and Miriam Cunningham (Eds)
IOS Press, 2007 Amsterdam
ISBN 978–1–58603–801-4

Benchmarking the Gridmosi Library

Rodica POTOLEA, Alin SUCIU, Alexandru MASCASAN
Technical University of Cluj-Napoca, 28 Baritiu, Cluj-Napoca, 400027, Romania
Tel: +40 264 401221, Fax: + 40 264 594491,
Email: rodica.potolea@cs.utcluj.ro, alin.suciu@cs.utcluj.ro

Abstract: The goal of the grid-computing model was to make better use of distributed resources, to put them together to achieve higher throughput and tackle large scale computation problems. Performance gain is pursued at each and every level of an application. We focus our present study on the "gridification" of several classes of computationally and/or data intensive algorithms, included in the Gridmosi Library. Various benchmarks were carried out, using two selected grid nodes from the SEE-GRID and the results show the important performance gains of our approach. Our system represents a proof of concept for a scenario in which an industrial user can access large computational power and storage capacity. This does not require the user to possess neither a performant computing system, nor extensive expertise in the application domain.

1. Introduction

The birth of grid computing was very often associated with the introduction of the electrical power grid due to certain similarities. Back in the beginning of the 20th century, electric power generation was possible, but the real problem was making it available worldwide without the necessity of each home consumer to possess an electric generator. What once was the electrical power is now the computational power, and what once were the electrical generators are now high performance computing systems with large storage capabilities.

Switching now to our point of interest – computing is currently in the same “dilemma” – the revolutionary thing to do would be to introduce a grid infrastructure to make computing power and resources available to a greater extent. This analogy was pinpointed by Leonard Kleinrock in 1969 [6]. An early definition of a computational grid was introduced in 1998 by Ian Foster and Carl Kesselman [3].

Nowadays scientists are more and more concerned with how many floating point operations per month or per year they can extract from a computing environment, rather than considering floating point operations per second [4, 5]. With the introduction of the grid concept, more attention has been devoted to such computing environments known as High Throughput Computing environments.

Parallelization has emerged as a need for improving the response time – the time needed to generate the solution to a problem. In order to achieve parallelization of a problem we focused on two approaches [7]:

- Data parallelism – refers basically to running the same piece of code on different pieces of input data. The major task here is splitting the input into chunks of data on which the same algorithm may be applied. This may be done only when the particularity of the problem allows it, when working on a subset of the initial input does not alter the final solution. Constraints may occur that can limit the overall potential for parallelism.
- Control parallelism – refers to running different sequences of code on the same data in order to obtain the result. The task here is parallelizing the problem at the algorithm level, i.e. create a parallel approach to the solution by splitting the computing responsibilities among several processes. This is a much more challenging approach, as

it involves handling communication among processes which might increase the overall running time. In order to obtain a better performance, communication among processes must be kept to a minimum.

We propose here a model for a service provided to industrial nonexpert users, offering computing power and storage capability, based on a library of algorithms (the Gridmosi Library). The user must select from the library the desired functionality, without having knowledge neither about the implementation nor about the optimal number of processors for the particular input chosen. The system, in a transparent fashion, selects the optimal number of processors and executes the selected functions accordingly.

The next section presents an overview of the Gridmosi Library, and then in Section 3 we present the core problems which occur when implementing Grid solutions to different problems. Section 4 contains a description of the main algorithms we implemented in our Gridmosi Library, focusing on the particularities of each of the decisions taken, and the corresponding experimental results. Conclusions of the finalized work are presented in Section 5, together with the ongoing development and the goals for the near future.

2. The Gridmosi Library

Our local grid is a development infrastructure which is integrated within a national homogeneous grid (GRID MOSI – eInfrastructure for Modelling, Optimization and SImulation on Grid), with 4 different locations in 3 geographically different cities (Cluj-Napoca, Bucharest and Timisoara). Moreover, the national Grid is connected to SEE-GRID (South Eastern Europe Grid – a large European grid dedicated to the scientific community). The local infrastructure contains 1 SE (Storage Element 1.2 TB HDD), 1 CE (Computing Element), 12 WN (Work Nodes), Gigabit LAN and 155 Mbps Internet connections, with the aim of upgrading to 1Gbps in near future.

We have started with the idea of implementing a library of algorithms to cover the classes of the most frequent basic problems requested to be solved. The first aim was to move from the sequential implementation to a parallel one. We also intended to have a parallel implementation that exploits in the best possible way the advantages of running parallel processes and makes better use of distributed resources.

We have started from the idea that the running time of an algorithm is not decreasing indefinitely while the number of processing elements is increasing. As the theoretic shape is a parabolic one, our goal was to prove it in practice. Moreover, we were interested whether there are exceptions to this rule. Finding the minimum number of processors to be used while running a particular algorithm on a particular dimension of the problem is very important. The ultimate goal of a grid could be to simply provide computational power to users and our library can be seen as a service provided to customers. They would not be interested in how many processors their task is running on. The library provides such an environment in which the user simply sends – through an interface – the problem to be solved and he/she receives the answer.

One general problem which could dramatically affect the performance of any algorithm implemented refers to the data access. Since it is a frequent operation which occurs in all algorithms, it has to be optimal. One can imagine a straightforward method for data access on a grid: placing the data files on a storage element and retrieving them during the execution of the job. The second way to access data is locally: this implies that along with the job we send the necessary input data. All these are packed together and sent as a unit called input sandbox. We developed an application that reads various chunks of data from a file. In order to determine an optimal buffer size for grid applications we considered both methods for data access. We analyze these two methods and search for the optimal buffer size. The results obtained were used for implementation of the algorithms listed below.

The Gridmosi Library contains a collection of algorithms from the following classes: dense matrix algorithms, graph algorithms, sorting, numerical algorithms, cryptographic algorithms and statistical tests for randomness [8].

For each algorithm we present the benchmarking results obtained after running a number of tests in which we focused on determining the optimal number of work nodes the algorithm needs to run, in order to have the smallest execution time. To achieve this goal, the tests performed were based on varying the number of working nodes assigned to solving an algorithm for a fixed problem size.

After obtaining the expected parabolic shape of the graphical representation of running time, we repeated the tests but on a modified problem size. On all algorithms that we tackled we performed tests for a wide range of problem sizes. The difference between the largest problem size and the smallest problem size tested is one order of magnitude. This approach was considered in order to be able to draw a meaningful conclusion from interpretation of test results.

The following dense matrix algorithms were implemented: Matrix Multiplication (the Fox Algorithm), Matrix Multiplication and Gauss Elimination. The following graph algorithms were implemented: Minimum Spanning Tree, Shortest Path, Transitive Closure and Graph Isomorphism.

From the class of sorting algorithms we implemented the Enumeration Sort algorithm and the Quicksort algorithm. The class of numerical algorithms is represented in our library by the Fast Fourier Transform algorithm, more precisely the Cooley – Tukey algorithm for one-dimensional, unordered, radix-2 FFT. The following encryption algorithms were implemented: AES, Blowfish and RC6. The following statistical tests for randomness were implemented: Monobit Test and Runs Test.

3. Basic Processing Issues

One basic step in obtaining an implementation on a grid is represented by the parallelization. As mentioned in the introductory section 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.

Another basic issue which has to be solved refers to the optimal buffer size for grid applications. In order to get to the most efficient way to handle this problem, we have developed and run tests, considering two alternatives.

In reading data from SE, 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. The restriction has imposed the introduction of a transfer protocol (gridftp) based on the general file transfer protocol and also checks the needed credentials.

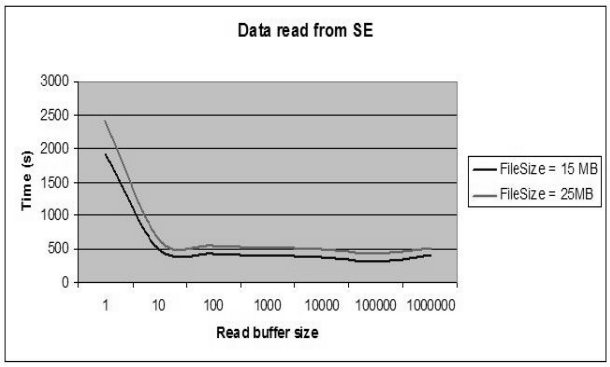

Figure 1: Variation of Time Required to Read a File from SE

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. A relatively minimum can be spotted in the vicinity of value 100,000 bytes. After this value (Figure 1), the read time increases but not at all dramatically to approximately 500 milliseconds.

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 rage [10,000; 500,000], and we also performed our test on larger files for a better view of the actual read process.

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 the executable files to the WNs.

When specifying in a ".jdl" file that the reading is to be performed from the SE, the input data is not copied locally on each WN (along with the executable files), so when performing a read operation, the WN sends a read request to the SE. The SE identifies the file to be read, reads the amount of information needed and sends it back to the WN that issued the request. All this data exchange is performed on the local network, so having a read performance graph influenced by the network traffic is obvious and inevitable. Even though the results are influenced by network traffic, the general trend is represented by a parabolic shape, with the minimum situated in the interval [150,000; 300,000].

The second method for data access in a grid environment that we tested is local read. The input files are not remotely placed on the SE, they are copied locally, on each WN local disk. This implies the existence of several replicas in case the job is intended to be run on multiple WNs (as in the case of MPI jobs). In case the input data is of significant amount it is not advisable to store multiple replicas; a lot of storage space would be wasted.

The concept of data grid, described earlier, works on huge data and it is not uncommon for a job to need access to hundreds of GB of data. Not even the distribution of such data would be extremely expensive, but WNs are simple computers, workstations with simple architecture where the storage of such an amount of data may be impossible. As a consequence, it has been imposed a limit on the amount of data that can be transferred in one input sandbox, and this is 10 MB. Due to this limitation our tests had to be done on files smaller than this maximum value.

It can be noted 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 relevant here. In its essence the local

read mechanism on the grid, 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 behaviour observed when using small 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].

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 (to be easy to remember).

4. Implementation and Experimental Results

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. 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 2 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.

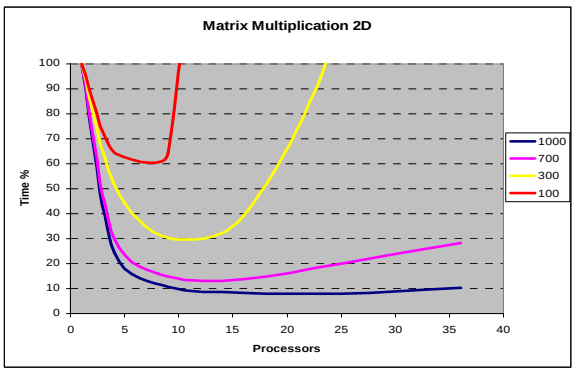

Figure 2 Performance Improvements on GRID for for Matrix Multiplication

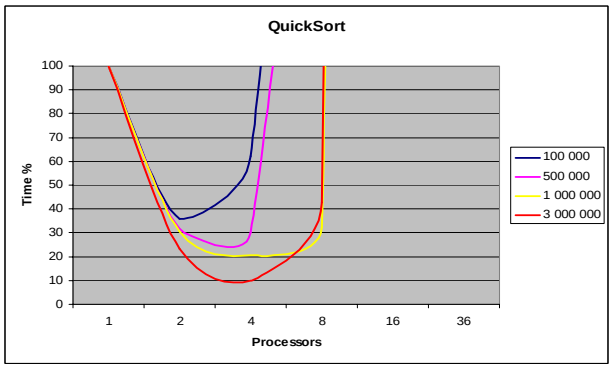

Figure 3 Performance Improvement on GRID QuickSort

The second graphic shows that while the dimension of input data increases, the optimal number of processors to run the algorithm shifts to the right. We could observe those outcomes for almost all implemented algorithms.

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 $3*10^6$ 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 ($5*10^5$).

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 Figure 4 below, 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.

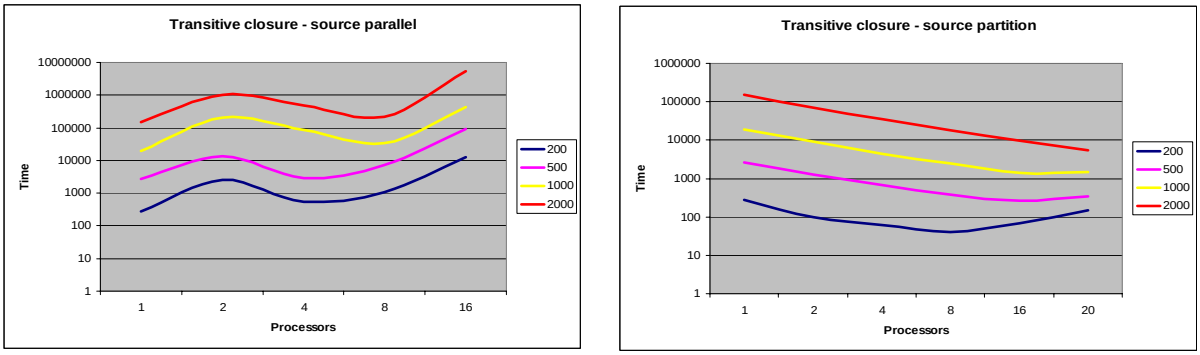

Figure 4: Atypical Shapes for Non-Communicating Algorithms Implemented on GRID

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 possessors available.

With the FFT Based on Cooley-Tukey algorithm (which requires inter-process interaction only during the first d = log p of the log n iterations) we return to the same evolution of the running time. An improvement of the running time of the parallel solution, compared to the sequential one, with almost 2 orders of magnitude (parallel running time yields to 2% of the mono processor running time), and a smooth transition to the right of the optimal number of processors as the problem size increases could be noticed.

By interpreting the results obtained for the cryptographic algorithms we have implemented and tested (Rijndael, AES, Blowfish, RC6), we may observe several interesting aspects: speed increase is not linear (especially for small input sizes), as input

size increases, a constancy can be detected (for instance the 8 MB decryption takes half as long for 2 processors than for one, and approximately a quarter for 4 processors), which leads to the conclusion that the time remains almost constant as while doubling the dimension, we double the number of processors.

Table 1: Decryption Time for Rijndael

Decrypt	1	2	3	4	5	6	7	8
1MB	**6.40**	4.42	2.89	2.39	1.66	1.64	1.24	1.23
2MB	12.80	**8.64**	5.74	4.11	3.64	2.90	2.49	2.35
4MB	25.54	17.25	11.51	**8.58**	6.98	5.77	4.94	4.51
8MB	66.16	33.45	22.47	17.25	13.78	11.50	9.88	**8.63**

On Figure 5 we have the diagrams of Rijndael and RC6 algorithms for different dimensions of input (expressed on MB) the time in seconds (ordinate) and a function of the number of processors (abscissa).

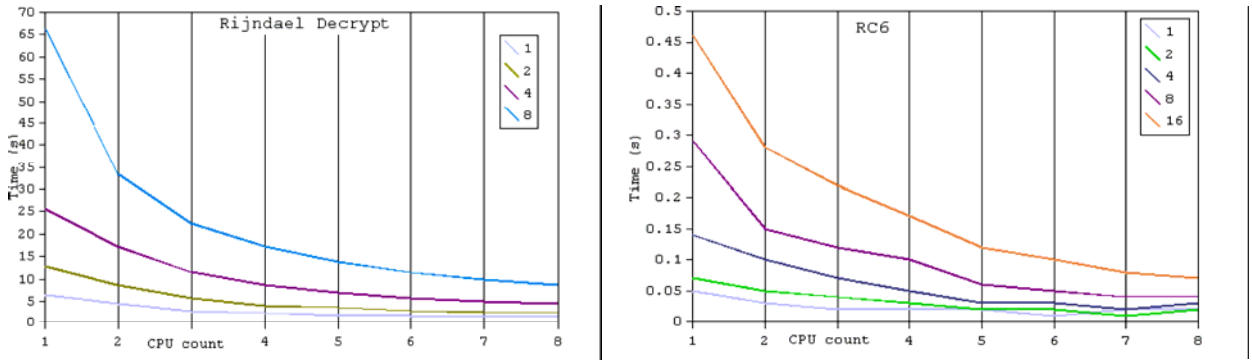

Figure 5: Time (in seconds) for Cryptographic Algorithms

5. Conclusions and Further Work

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 environment provides an interface which allows users to select the problem to be solved, and the dimension of input data. The system chooses the optimum number of processors to run the problem, and executes it on the grid.

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:

- Optimal number of WNs is strongly depending on the size of the input. As the size increases, the optimal value shifts to the right.
- After reaching the optimal value, the running time tends to increase, due to the communication component which becomes larger. Hence, most of the shapes 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 (hence the parallel running time is 10% from the sequential one, one order of magnitude improvement), 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 intend to refine this study, taking into consideration two different approaches. First, more input samples in order to establish the optimal number of WNs for input sizes within a larger range (i.e. run the same algorithms on increased sizes, 10 times larger at every evaluation step). Second, more algorithms could be added to the Library, according to the actual needs of the industrial users.

The experiments ran on the one of the grid nodes, show that the Monobit Test and the Runs Test algorithms perform better as the number of work nodes increases. It also shows that the same tests need to be run on another grid node which has more WNs, in order to be able to find the optimal number of nodes suitable for each type of algorithm. We are currently in the process of testing and collecting data for these tests. Moreover, we have evaluated several potential algorithms to be considered for further implementation. Based on this analysis we are considering the factorization problem, a battery of randomness tests considering NIST, DIEHARD (Marsaglia) and TESTU01 (Simard & L'Ecuyer), finalists for AES competition encryption algorithms selected by NIST, and public key encryption algorithms (RSA, ECC).

References

[1] M. Baker, R. Buyya and D. Laforenza, "Grids and Grid Technologies for Wide Area Distributed Computing", Software Practice and Experience, John Wiley & Sons, 2002.

[2] P. T. Bulhoes, C. Byun, R. Castrapel and O. Hassaine, "N1 Grid Engine 6 Features and Capabilities", SUPerG, Phoenix, Arizona, May 2004.

[3] I. Foster and C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, 1st edition, 1998.

[4] I. Foster, C. Kesselman and S. Tuecke, "The Anatomy of the Grid", International J. Supercomputer Applications, 15(3), 2001.

[5] Ian Foster, "The Grid: A New Infrastructure for 21st Century Science", Physics Today, 55(2): 42-47, February 2002.

[6] L. Kleinrock, "UCLA to be first station in nation wide computer network", UCLA – Office of Public Information, July 3, 1969.

[7] L. Ferreira and V. Berstis, "Fundamentals of Grid Computing", IBM Redpaper, 12 November 2002, IBM Form Number REDP-3613-00.

[8] A. Mascasan, R. Potolea and A. Suciu, Optimal Buffer Size for Grid Applications, Automation Computers Applied Mathematics, pages 203-210, 2006.

[9] A. Rukhin et al, "A statistical test suite for random and pseudorandom number generators for cryptographic applications", NIST Special Publication 800-22, 2000.