Using a grid platform for solving large sparse linear systems over GF(2)

Thorsten Kleinjung, Lucas Nussbaum, Emmanuel Thomé

Using a grid platform for solving large sparse linear systems over $\mathrm{GF}(2)$

Thorsten Kleinjung
ÉPFL/IC/Lacal
ÉPFL
Lausanne, Switzerland
thorsten.kleinjung@epfl.ch

Lucas Nussbaum
Projet Algorille
LORIA – Université Nancy 2
Nancy, France
lucas.nussbaum@loria.fr

Emmanuel Thomé
Projet Caramel
INRIA Nancy
Nancy, France
Emmanuel.Thome@inria.fr

***Abstract*—In Fall 2009, the final step of the factorization of `rsa768` was carried out on several clusters of the Grid'5000 platform, leading to a new record in integer factorization. This step involves solving a huge sparse linear system defined over the binary field $\mathrm{GF}(2)$. This article aims at describing the algorithm used, the difficulties encountered, and the methodology which led to success. In particular, we illustrate how our use of the block Wiedemann algorithm led to a method which is suitable for use on a grid platform, with both adaptability to various clusters, and error detection and recovery procedures. While this was not obvious at first, it eventually turned out that the contribution of the Grid'5000 clusters to this computation was major.**

I. Introduction

The work described in the present article originates from the `rsa768` project, which is an *integer factorization* challenge. This algorithmic task is of prime interest in cryptographic context, since it is the keystone of the RSA cryptosystem. The *cryptographer* builds a *public key* which is an integer N, the *private key* being two prime numbers p and q of roughly equal size satisfying $N = pq$. Factoring N unveils the private key from the public key. It is therefore of prime interest to assess constantly the state of the art in terms of integer factoring, so that deployed cryptographic solutions (e-commerce, EMV credit cards, and many, many more) can be designed with proper parameters. Here, `rsa768` denotes a particular 768-bit (232 decimal digits) composite integer published by RSA laboratories as a *challenge*[1]. This challenged has been solved in December 2009 as a joint effort between several research teams [1].

Factoring a 768-bit number reveals a large number of stumbling blocks. The preferred, and most efficient algorithm for this task, is the *Number Field Sieve* (NFS) factoring algorithm [2]. Describing this algorithm is out of scope of the present article. It suffices to know that there are two computationally intensive steps. The first step, called *sieving*, is a typical resource harvesting task. The second step, which draws our attention in the present article, is radically different. This step is a *linear algebra* step.

Several solutions to a homogeneous linear system defined over the binary field $\mathrm{GF}(2)$ are to be found. While linear algebra is ubiquitous, the linear system encountered here differs in many fundamental aspects from the systems typically encountered in other contexts – these differences are outlined in Section II. The question of *how* this linear algebra computation is performed has several aspects. The main contribution of this work is the presentation of an approach which makes it possible to solve a large linear system of this kind (about 192,000,000 equations and unknowns) on a grid platform, with the capability of using not only one, but *several* unconnected clusters.

Indeed, while part of the `rsa768` computation used computer time provided by partners ÉPFL (Switzerland), and NTT (Japan), we describe here how a significant part of the computation has been carried out on the Grid'5000 platform, which is an experimental platform dedicated to research in large-scale and distributed systems, composed of a dozen of clusters and a total of 6000 CPU cores.

Our approach illustrates that the needs in terms of computer resources for a computation such as the `rsa768` linear system are quite different from a "supercomputer", and that even an exclusive access to a dedicated "in-house" cluster is not nec-

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

[1]Money rewards used to be offered for solving such challenges, prior to 2007. Since then, RSA laboratories have discontinued this practice.

essary. As an outcome, the amount of distribution achieved for the `rsa768` linear system goes quite a bit beyond what has been previously done.

A second characteristic of linear algebra computations, especially those relying on black box methods (described in Section II) is the crucial importance of the error detection and recovery procedures. We describe how our approach is capable to achieve this fault management in a satisfactory way.

Section II gives details on the linear system to be solved, and lists the relevant algorithms. Section III gives specifically a bird's eye view on the block Wiedemann algorithm, which has been used in the computation. Section IV discusses how the computation is split into elementary subtasks, while Section V details how these subtasks are scheduled, together with the issues relative to the management of the computation at a central level. Section VI describes how the results were checked against errors. Some experimental results and comparisons are summarized in Table II.

A common topic occuring in linear algebra calculations is not detailed in this present article. Of course, it is important to know how linear algebra operations (in our case, matrix-times-vector multiplications) are performed on a cluster using e.g. the MPI interface, and taking advantage of high-speed networks, topologies, and so on. This is obviously one of the ingredients of our work, but rather orthogonal to our primary focus which is on the use of several unconnected clusters.

II. Statement of the problem

A. The matrix

Throughout the article, we assume that an $N \times N$ matrix M defined over the binary field $\mathrm{GF}(2)$ is given, and that we seek elements of the *right nullspace*[2] of this matrix. More specifically, since we instantiate our work with the `rsa768` matrix, we provide the exact characteristics of the matrix under consideration:

- $N = 192{,}796{,}550$ rows and columns; dimension of the kernel at least $1,000$.
- 27,797,115,920 non-zero coefficients, i.e. approximately 144 non-zero coefficients per matrix column;
- 105 GB to store the matrix in a straightforward format.

[2]Most presentations of the Number Field Sieve call for a row dependency, hence an element of the left nullspace. For consistency of the presentation, we give here the transposed view.

Of course, as can be judged from the figures above, the matrix M is extremely *sparse*. Sparse matrices occur in various contexts, but are probably best known in the context of numerical computations. It is important to stress that the problem we consider here is radically different, notably with respect to the following aspects.

- The field of definition being a finite field (not $\mathbb{R}$ or $\mathbb{C}$), there is *no* relevant notion of *convergence*, *fixed point*, or *dominant eigenvalue*. Many algorithms rely on such key concepts for solving numerical linear systems, and this is unfortunately of no use in our context.
- While it occurs fairly often that large sparse matrices are symmetric, or perhaps almost symmetric, this is not the case here. The matrix to be solved does have some distinguishing shape, however. The density of the columns is evenly distributed, while the density of rows is not: a small number of rows are extremely heavy, almost dense, while the vast majority of rows are considerably sparser than average. 90% of the rows have less than 110 non-zero coefficients, 66% have less than 36. Thus one clearly has "dense" and "sparse" areas, the sparse parts accounting unfortunately for the largest part of the computation time.
- Numerical stability is not an issue, since all computations are *exact*. There is a counterpart though. In numerical calculations errors due to bad memory are usually not a problem. A flipped bit in the mantissa does not change the value much and even if a bit in the exponent flips, its effect might be unnoticeable after a few iterations. In contrast, a flipped bit in the calculation over $\mathrm{GF}(2)$ will be a disaster since its effect will spread and, after a few iteration, all components of the vector will be affected, thus invalidating the whole computation.

B. Algorithms

The problem of solving large sparse linear systems over finite fields has emerged a few decades ago in the context of integer factorization and computation of discrete logarithms over finite fields. A classical survey of the early methods is given in [3].

Sparseness being the key characteristic here, the most efficient methods are *black box* algorithms. The matrix M defining the linear system to be solved is considered as a black box, to which no internal access is allowed, and which is capable

of performing only one operation, namely the multiplication of the matrix by a vector.

When the linear system to be solved is defined over the finite field $\mathrm{GF}(2)$, the algorithms of choice are so-called "block" methods, alternatively viewed as "single-instruction, multiple data" algorithms: vectors are replaced by blocks of vectors. A block of vectors is a sequence of N tuples of n bits, where the "blocking factor" n is typically 64 or a multiple thereof. This effectively represents n vectors of N coordinates. Operations such as the addition of two vector blocks use the machine-word wide XOR operation when $n = 64$. In return, the number of required black box applications is reduced almost proportionally (up to a certain limit).

Two algorithms are commonly used, known as the block Lanczos and the block Wiedemann algorithm. The block Lanczos algorithm is appealing because it requires slightly less computations than the block Wiedemann algorithm. However it does not scale as well, as outlined in [4], [1]. The block Wiedemann algorithm offers the possibility of splitting most of the computation into several independent computations, thereby allowing a very desirable coarse-grain parallelism. This advent of the block Wiedemann algorithm as an alternative to the block Lanczos algorithm appeared first on a significant scale in [5] and [4].

III. The block Wiedemann algorithm

The block Wiedemann algorithm [6] is a block extension of the Wiedemann algorithm [7]. This method is *a priori* specific to finite fields. We give here an outline of the algorithm. A complete exposition of the block Wiedemann algorithm can be found in several places [6], [8], [9] and some of the references therein.

The starting point of the block Wiedemann algorithm is a vector block y, consisting of n linearly independent N-bit vectors. One typically takes $n = 64n'$, so that n' sequences of N 64-bit words are needed to store the vector y. A second input is a vector block x, similarly consisting of m linearly independent N-bit vectors. Here m is a second blocking parameter, not necessarily equal to n. Likewise, we typically set $m = 64m'$. The analysis of the algorithm requires the vectors in the block x be random vectors. However for efficiency we content ourselves with x being a set of m unit vectors, while y is indeed random.

A. First step: scalar products

The first step of the block Wiedemann algorithm computes the first $L = \left\lceil \frac{N}{m} \right\rceil + \left\lceil \frac{N}{n} \right\rceil$ terms of the following power series.

$$A(X) = \sum_{i=0}^{L} a_i X^i = \sum_{i=0}^{L} {}^t x M^i y X^i.$$

The computation of this series can be done by repeating the following iteration:

- Set $y_0 \leftarrow y$, and $i = 0$;
- Output $a_i = {}^t x y_i$. Set $y_{i+1} = M y_i$, and $i \leftarrow i + 1$;
- Repeat previous step until $i = L$.

The key observation here is that column j of the matrices $(a_i)_i$ does not depend on the columns j' of the starting vector y, as long as $j \neq j'$. Therefore, the terms of the series $A(X)$ can be computed piecewise, e.g. 64 columns by 64 columns. This is exactly how the computation is carried out. Using the superscript notation $S^{(j)}$ to denote the submatrix extracted from a matrix S by selecting the 64 columns of indices $64j$ to $64j + 63$, we thus have:

$$a_i^{(j)} = {}^t x M^i y^{(j)},$$
$$a_i = \left(a_i^{(0)} \,\middle|\, a_i^{(1)} \,\middle|\, \ldots \,\middle|\, a_i^{(n'-1)} \right)$$

Producing the terms of $A(X)$ therefore essentially involves repeated application of the black box. Each submatrix $A(X)^{(j)}$ gives rise to a sequence of $m \times 64$ matrices over $\mathrm{GF}(2)$; we have n' such sequences of L terms. A sequence of ℓ consecutive terms of such a sequence is thus made of $\ell \times m \times 64$ bits. Note that a sequence $(a_i^{(j)})_{i=i_0 \ldots i_0+\ell-1}$ requires only prior knowledge of the vector $y_{i_0}^{(j)} = M^{i_0} y^{(j)}$.

B. Second step: linear generator

The next (central) step of the block Wiedemann algorithm computes a *linear generator* for $A(X)$. Description of how this step works is out of scope here, the reader is referred to [9] for an algorithm of complexity $O((m+n)^2 N \log N)$ for this task. This linear generator is an $n \times n$ matrix $F(X)$ of polynomials of degree $\left\lceil \frac{N}{n} \right\rceil$ such that

$$w = \sum_{i=0}^{\left\lceil \frac{N}{n} \right\rceil} M^i y \, {}^t F_i$$

is with high probability an element of the kernel of the matrix M. This step is the one whose complexity *grows* with the blocking factors m and n, and furthermore it does not lend itself well to

running on a grid platform. However the flexibility in the choice of parameters made it possible to arrange so that this step is not a stumbling block.

C. Third step: evaluation

The final step uses the linear generator, and proceeds in a way which is similar to the first step. The operations performed are:

- Set $y_0 \leftarrow y$, $S = 0$, and $i = 0$;
- Do $S \leftarrow S + y_i\, {}^tF_i$. Set $y_{i+1} = My_i$, and $i \leftarrow i + 1$;
- Repeat previous step until $i = \lceil \frac{N}{n} \rceil$. Output $S = w$.

As was already pointed out for the first step of the computation, it is possible to split the computation of S into several independent tasks. Indeed, we have:

$$\begin{aligned} S &= \sum_{i=0}^{\lceil \frac{N}{n} \rceil} M^i y\, {}^tF_i, \\ &= \sum_{j=0}^{n'-1} \sum_{i=0}^{\lceil \frac{N}{n} \rceil} M^i y^{(j)}\, {}^tF_i^{(j)} \end{aligned}$$

where the superscript notation $^{(j)}$ is as in Section III-A. It follows that $S = w$ can be computed as a sum of several partial sums. The formula above explicits already n' clearly identified partial sums, but in fact, assuming a few vectors $y_{i_0}^{(j)}$ have been kept from the first step of the computation, it is possible to compute the partial sum:

$$S_{(i_0 \ldots i_0+\ell-1)}^{(j)} = \sum_{i=i_0}^{i_0+\ell-1} M^{i-i_0} y_{i_0}^{(j)}\, {}^tF_i^{(j)}.$$

Such a sum is a vector block of Nn bits. Note the considerable difference with the size of the results from the first step. Here, partial sums are large and do not depend on the length ℓ of the partial computation.

IV. Splitting into small tasks

A. Steps 1 and 3 into pieces

Having developed the outline of the block Wiedemann algorithm, we now wish to identify a decomposition of steps 1 and 3 into subtasks. The intent is that one such subtask is run on one cluster, and several subtasks can run *concurrently* on different, *unconnected* clusters.

Computations of steps 1 and 3 relate to n' different recurring sequences $y^{(0)}, \ldots, y^{(n'-1)}$. Focusing on sequence number j, the data computed from the sequence elements $y_{i_0}^{(j)}$ to $y_{i_0+\ell-1}^{(j)}$ is thus derived from the knowledge of the starting iteration $y_{i_0}^{(j)}$, as well as some possible auxiliary data.

More specifically, we can instantiate the data sizes at stake here with the `rsa768` matrix sizes. The *blocking factors* chosen for the `rsa768` computation were $m = 1024$ (hence $m' = 16$) and $n = 512$ (hence $n' = 8$).

First, iterations i_0 to $i_0 + \ell - 1$ of step 1 on sequence j require the input vector $y_{i_0}^{(j)}$. This represents N machine words of 64 bits, which is 1.43 GB. The output of this subtask is the collection of tiny matrices $\left\{ {}^txy_i^{(j)},\ i_0 \le i < i_0 + \ell \right\}$. Each such matrix is an $m \times 64$ matrix, which occupies 8 kB of memory. The output of the subtask is therefore 8ℓ kB, plus the ending iteration $y_{i_0+\ell}^{(j)}$. Step 1 has to run until iteration number $L \approx 565,000$.

Iterations i_0 to $i_0 + \ell - 1$ of step 3 on sequence j also require the input vector $y_{i_0}^{(j)}$ (which may have been kept from the step 1 runs). Furthermore, the linear generator coefficients $F_i^{(j)}$ are required to compute the result $\sum_{i=i_0}^{i_0+\ell-1} y_i^{(j)}\, {}^tF_i^{(j)}$. The amount of linear generator data to be read is ℓ coefficient matrices of size $n \times 64$, which corresponds to 4ℓ kB. The output of the subtask is the partial sum, which is n' times as large as a vector, hence about 12 GB. The ending iteration $y_{i_0+\ell}^{(j)}$, as before, is also needed. Step 3 has to run until iteration number $\lceil \frac{N}{n} \rceil \approx 375,000$.

B. Concurrent computations

We have already mentioned that computations relative to different recurring sequences are completely unrelated, so that they can run on different unconnected computing resources. Also, note that step 3 uses iterates $y_{i_0}^{(j)}$ which have already been computed by step 1. While storing *all* iterates would obviously be a bad idea, it appears that storing a few of them increases the potential number of concurrent subtasks that can run during step 3.

C. Constraints for choosing clusters

The choice of an appropriate cluster for running a subtask is driven by several criteria. Most of the time spent in the computation is related to the matrix-times-vector multiplications. Therefore it is natural, among the available clusters, to favour those with a high-speed network available (Myrinet, Infiniband for example). We also require obviously that the matrix, as well as all required temporary buffers, fit in RAM on the cluster nodes. As a rule of thumb, for the `rsa768` matrix, this requires a total of 200 GB of RAM. As briefly

argued in the introduction, the specifics of how the matrix-times-vector multiplication is performed on a cluster are not developed here.

For a cluster to be used, the number of CPU cores used must also match some constraints. It must be decomposable in the form $p \times q$, where p and q are preferably of comparable size so that the cost of communication is not too unbalanced in the matrix-times-vector multiplications. It is also necessary to prepare the matrix for a computation on such a grid of pq jobs (in the MPI sense). This is a relatively expensive precomputation, whose output is a collection of files of approximate total size 75 GB.

D. Roadmap of a subtask

A subtask, when specified as an abstract description such as "work with iterations i_0 to $i_0 + \ell - 1$ of step 1", and when run on a specific (sub-) cluster, goes through several steps. When a job starts on the computer cluster, no persistent storage exists, so that the complete required data set must be imported from central storage before computations can actually begin. This includes the precomputed matrix data mentioned above, the starting iteration $y_{i_0}^{(j)}$, as well as for step 3 the linear generator data $\left\{F_i^{(j)},\ i_0 \leq i < i_0 + \ell\right\}$.

Likewise, all results must be saved to central storage, because all local data is lost when the job terminates.

V. Several clusters for several subtasks

The crux of our approach is the capability to adapt to a changing set of available clusters. The clusters which, as per the criteria developed in the previous section, are eligible for running a subtask at a given moment, are not always the same. Table II at the end of this article gives an indication on the number of different configurations which have been used. This dynamic set of available resources contrasts with the situation of "in-house" clusters which in favourable cases can be tailored specifically to the needs of the computation.

The management of the computation at the "central" level (above the "subtask" level) has two facets. Of course, this includes the dispatching of subtasks on the different computer clusters. This dispatching must be efficient, and must avoid exhausting the computing resources (this concern appears because Grid'5000 is an experimentation grid). Another facet is related to one of the key differences between a dedicated cluster and a shared computing resource. As alluded to in Section IV, we cannot rely on persistent local storage on the nodes. This implies that storage is an issue which must be handled at the central level.

A. Scheduling grid jobs for subtasks

The complete `rsa768` linear algebra computation lasted about four months, but each subtask is limited to the maximum job length allowed in the platform (about 60 hours for Grid'5000). Therefore, organizing subtasks is a matter of scheduling grid jobs via the job submission systems. Since the `rsa768` linear system was solved as a joint effort with other groups from ÉPFL (Switzerland) and NTT (Japan), only 4 of the 8 sequences were available for use on Grid'5000. This means that at a given time, up to 4 jobs (subtasks) could be running on the platform. A job has to be bound to a particular cluster, so as to make best use of the resources. While the block Wiedemann algorithm as a whole has some asynchronism capability, subtasks want to work as synchronously as possible.

Of course, as Grid'5000 is a shared resource, it was not reasonable to use some of its clusters exclusively during the whole computation. Instead, after discussion with the steering committee, we compromised on using two types of jobs:

- reservations a few hours beforehand on some clusters, or submission bound to start immediately – in that case, the resources were assigned to us for the whole duration of the job. This was done in general at most a couple of hours beforehand so as to let other users get a chance of using the clusters, and only during nights and week-ends ;
- "*best-effort*" jobs – which are automatically killed if another user requests the resources with a higher priority. This allowed us to use the platform when no other user was asking for the resources, and to release the resources as soon as they are requested by another job.

The characteristics we considered for submitting jobs were mostly speed, and sometimes reliability. We built a table of the speeds achieved by given clusters for our application. Also, some clusters were large enough to be able to fit several jobs, e.g. three 24-node jobs. Given the clusters available at a given time, this chart indicates the preferred choice, maximizing the cumulated number of iterations computed per second. This selection step evolved from manual choice at the beginning of the `rsa768` computation, towards a largely automated procedure.

As remarked in paragraph IV-B, step 3 is less constrained in terms of the number of concurrent

data	size	output from	input to
Input matrix	105 GB	none	balancing
Balancing	75 GB	balancing	1+3
Vectors $y_i^{(j)}$	1.43 GB	1+3	1+3
Matrices $a_i^{(j)}$	8 kB	1	2
Matrices $F_i^{(j)}$	4 kB	2	3
$S^{(j)}_{i_0 \dots i_0+\ell-1}$	12 GB	3	none

Table I
FILES USED WITHIN THE COMPUTATION. STEPS 1, 2, 3 REFER TO THE BLOCK WIEDEMANN STEPS. STEP 2 IS NOT COMPUTED ON THE GRID PLATFORM.

jobs running, therefore the optimization is relaxed. This implies that clusters which are *a priori* much slower than others, notably by lack of a fast interconnect, can become of some use.

Including "best-effort" jobs in the computation is by no means obvious. It relies on the possibility of achieving quick startup times and efficient periodic checkpoints through the use of adequate storage. This is explained in the following paragraphs.

B. Storage points, data sets, and transfers

Terabytes are cheap. However, terabytes of efficient, fast storage, near computing resources, are considerably more expensive to install and to maintain. We elected to aim at carrying out the `rsa768` computation with a limited amount of storage close to the computing resources, while offloading to servers outside of Grid'5000 most of the data which was not meant to be reused shortly. We thus consider three classes of storage points. The first class is the set of compute nodes, which have no persistent local storage. They do however have access to Grid'5000 storage servers, which form the second class. Storage on these servers is persistent and available, but there is no reason to overuse it – we have been allowed a reasonable quota of 500 GB on three storage points, partially redundant. These servers are accessible (with reasonable throughput) from all compute nodes, but not from outside of Grid'5000 without tunnels. Storage servers are not meant to run computations. The last class is storage outside of Grid'5000. Sky is the limit here, as we have typically used a handful of large disks on desktop machines. These machines can access Grid'5000 storage nodes with SSH tunnels, and have CPU power for small tasks (checks, assembling files).

The data sets have been described in Section IV and are listed also in Table I. An important remark is that the precomputed matrix data adapted to one cluster size weight 75 GB. Therefore, for the amount of required data to remain reasonable, we chose to restrict to splittings 8×8, 12×12, 14×14, and 16×16; this already requires 300 GB of storage, which need to be available on the storage nodes because at any job startup, one of these data sets must be imported. This restricts the available cluster sizes, while still offering a reasonable variety of possible configurations.

Once started, a job must import its input data, begin the computation, optionally save its results periodically, and if possible trap a notification of approaching termination so as to get time to save a last checkpoint and the corresponding data.

The input data to be imported is the 75 GB precomputed balancing data, the starting vector $y_i^{(j)}$, and for step 3 the relevant linear generator file $F^{(j)}$. How to import these files ? The question of how *not* to do it is easily settled. Compute nodes have access to site storage via the NFS protocol, but the data throughputs are bad. After having tried several options, we chose the following solution:

- On each of the storage nodes on which we had a storage allowance, an `rsync` daemon listens on a custom port. This daemon, in order to limit the load on the machine, honors only a limited number of concurrent requests using the `max connections` parameter. The storage node from which a compute node tries to import a given file is selected at random. The compute node keeps trying servers at random until one accepts its request.
- An optimization is useful in the case where several jobs of identical size are expected to run at a given site (quite frequent within step 3). Each node holding a local copy of some submatrix file announces it with a phony symbolic link on the NFS partition (the link target being the node name), and listens with an `rsync` daemon on the same usual custom port. Compute nodes willing to retrieve the same file have therefore a way to first check for the availability of the corresponding file. This optimization helped reducing the data import times dramatically, when applicable of course.

Once the data file has been imported, computations can start. The key question is when and how to trigger saving of intermediate checkpoints and results. Triggering one checkpoint implies the production of an amount of data equal to:

- 1.43 GB + 8ℓ kB within step 1, where ℓ is the distance to the previous checkpoint.
- 1.43 GB + 12 GB within step 3, irrespective

of the distance to the previous checkpoint.

The choice of the checkpoint frequency is related to the lifetime of a job. Bar possible errors, this lifetime is decided at submission time, typically several hours, e.g. 12h. The lifetime of a best-effort job is unpredictable (and can be arbitrarily small). We used the following rules:

- Always trigger a checkpoint at least at multiples of 4096 iterations.
- For best-effort jobs, trigger periodic checkpoints so as to avoid losing the output of long-running jobs. We chose an 80 minutes period.
- Trigger a checkpoint 10 minutes (for step 1) to 20 minutes (for step 3) before job end.

Saving of checkpoint data and results by compute nodes to storage servers was done with the same `rsync` scheme that was used for importing input data. However, since saving results has higher priority than importing data, we used a second `rsync` port, with a larger number of allowed connections. Each storage server had thus typically 4 to 8 maximum outbound connections, and 8 to 12 maximum inbound connections.

It must be noted that the aggregated bandwidth consumed by saving checkpoints in step 3 can be quite considerable if the grid platform is used at the peak of its possibilities. For the `rsa768` computation, up to 15 concurrent subtasks could run during step 3. Assuming all are run by best-effort jobs, the total amount of data produced reaches in such a case about 40 MB/s on the whole grid. Depending on which kind of data path has to absorb this traffic (e.g. by offloading the data to remote servers), this can be an issue. This "peak" problem has not occurred in the course of the `rsa768` computation. As a partial solution, assembling the partial sums of step 3 (replacing $S_{a\ldots b-1}$ and $S_{b\ldots c-1}$ by their sum $S_{a\ldots c-1}$) can be done from within the grid platform, which avoids the possibly slower link from the grid platform to the outside. To do so, we first need to gain confidence in the partial sum files, which is the purpose of the following section.

VI. Living with errors

The amount of data computed requires that some checks for data integrity be performed. It turns out that it is possible to do such checks with the block Wiedemann algorithm quite easily.

Step 1 checks boil down to verifying that $(a_i^{(j)})_{i=i_0\ldots i_0+\ell-1}$ and $y_{i_0+\ell}^{(j)}$ are both consistent with the vector $y_{i_0}^{(j)}$. We do this check for constant length ℓ, such that all the iterates whose index is a multiple of ℓ are saved (given the checkpoint frequencies indicated above, we choose $\ell = 4096$). For this check, we fix a dimension μ arbitrarily (e.g. $\mu = m$), and precompute random $\mu \times m$ matrices $r_0, \ldots, r_{\ell-1}$, as well as the vectors ${}^tV_r = \sum_{k=0}^{\ell-1} r_k\, {}^txM^k$, and ${}^tC_\ell = {}^txM^\ell$. Computing vectors V_r and C_ℓ has a mild computational cost. The check is whether ${}^tV_r y_{i_0}^{(j)} = \sum_{k=0}^{\ell-1} r_k a_{i_0+k}^{(j)}$, and also whether ${}^tC_\ell y_{i_0}^{(j)} = {}^txy_{i_0+\ell}^{(j)}$. This check is only a dot product, and can be done with very limited resources. Errors are detected with satisfactory probability.

For step 3, a similar procedure can be used to check that a partial sum $S_{(i_0\ldots i_0+\ell-1)}^{(j)}$ is valid. This check does not require that the length ℓ be fixed beforehand. This is particularly handy because the partial sum files turn out to be very large, and the need to assemble them into files of equal sizes, but representing a larger work length ℓ, is desired. This can be done only if the different partial files are validated. The check is done as follows. Pick an arbitrary small integer δ, such that ${}^txM^\delta$ has no zero coordinate. The check is the following:

$$ {}^txM^\delta S_{(i_0\ldots i_\ell-1)}^{(j)} = \sum_{i=i_0}^{i_0+\ell-1} a_{i+\delta}\, {}^tF_i^{(j)}. $$

This check is fast and accurate enough for our purposes. As for the previous check, performing the verification is not expensive.

During the computation, some true errors have been detected using these checks. Running out of storage space was the principal source of errors. Data files were truncated in such occasions, or corrupted with blocks of zeroes. On a computation of this scale, the error-checking procedures described above were very efficient at detecting such events.

VII. Conclusion

The main unknown associated with the `rsa768` computation was whether the linear algebra would be doable with reasonable expenses and resources. While it was not obvious at first, and triggered many obstacles, we succeeded in proving that a distributed platform such as Grid'5000, despite being composed of a variety of clusters with different CPU and network interconnects, provided an effective answer. We were able to use 6 different clusters on Grid'5000, with 16 different configurations, as is illustrated by the timing results in Table II. Most unexpectedly, we were also able to adapt to the constraints of

(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)
Lausanne	56	2×AMD 2427	2.2	12	16	ib20g	12	144	4.3	4.8	40%
Tokyo	110	2×Pentium-D	3.0	2	5	eth1g	110	220	5.8	7.8	%
Grenoble	34	2×Xeon E5420	2.5	8	8	ib20g	24	144	3.7		30%
Lille	46	2×Xeon E5440	2.8	8	8	mx10g	36	144	3.1	3.3	31%
							32	256	3.8		38%
							24	144	4.4		33%
Nancy	92	2×Xeon L5420	2.5	8	16	ib20g	64	256	2.2	2.4	41%
							36	144	3.0	3.2	31%
							24	144	3.5	4.2	30%
							18	144		5.0	31%
							16	64		6.5	19%
Orsay	120	2×AMD 250	2.4	2	2	mx10g	98	196	2.8	3.9	32%
Rennes	96	2×Xeon 5148	2.3	4	4	mx10g	64	256	2.5	2.7	37%
							49	196	2.9	3.5	33%
Rennes	64	2×Xeon L5420	2.5	8	32	eth1g	49	196	6.2		67%
							24	144	8.4		67%
							18	144	10.0		68%
							8	64		18.0	56%

Table II
DIFFERENT PER-ITERATION TIMINGS ON VARIOUS CLUSTERS. (A) CLUSTER LOCATION; (B) TOTAL CLUSTER SIZE (NUMBER OF NODES); (C) CLUSTER CPU TYPE; (D) NODE CPU FREQUENCY; (E) CORES PER NODE; (F) RAM PER NODE (GB); (G) CLUSTER INTERCONNECT (ETH1G: GIGABIT ETHERNET, MX10G: 10GBPS MYRINET, IB20G: 20GBPS INFINIBAND); (H) JOB SIZE (NUMBER OF NODES); (I) NUMBER OF CORES USED PER JOB; (J) TIME PER ITERATION IN SECONDS (STAGE 1); (K) TIME PER ITERATION IN SECONDS (STAGE 3); (L) PERCENTAGE USED FOR COMMUNICATION.

best-effort jobs, and use them effectively through appropriate use of checkpoints and distributed data storage.

Timings given in Table II indicate the performance obtained in Fall 2009 for the computation as it has been actually run, during a period of approximately three months. The total CPU time spent for this computation was slightly above 100 CPU core-years. The computation described in this article was carried out as a compromise between the time for developing the infrastructure of programs for managing the individual tasks, and the goal of eventually obtaining the solution to the linear system. Some aspects of the approach have been identified as candidates for possible improvements, notably the "peak bandwidth" problem mentioned at the end of Section V. Further experiments of this kind would have to take this problem into account, possibly in a different way. Overall though, little doubt is left: moderately larger matrices are clearly within reach using this approach.

REFERENCES

[1] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, "Factorization of a 768-bit rsa modulus," in *CRYPTO 2010*, ser. Lecture Notes in Comput. Sci. Springer–Verlag, 2010, accepted for publication.

[2] A. K. Lenstra and H. W. Lenstra, Jr., Eds., ser. Lecture Notes in Math., vol. 1554. Springer–Verlag, 1993.

[3] B. A. LaMacchia and A. M. Odlyzko, "Solving large sparse linear systems over finite fields," in *CRYPTO '90*, ser. Lecture Notes in Comput. Sci., A. J. Menezes and S. A. Vanstone, Eds., vol. 537. Springer–Verlag, 1990, pp. 109–133.

[4] K. Aoki, J. Franke, T. Kleinjung, A. K. Lenstra, and D. Osvik, "A kilobit special number field sieve factorization," in *ASIACRYPT'2007*, ser. Lecture Notes in Comput. Sci., vol. 4833. Springer–Verlag, 2007, pp. 1–12.

[5] E. Thomé, "Computation of discrete logarithms in $\mathbb{F}_{2^{607}}$," in *ASIACRYPT 2001*, ser. Lecture Notes in Comput. Sci., C. Boyd and E. Dawson, Eds., vol. 2248. Springer–Verlag, 2001, pp. 107–124.

[6] D. Coppersmith, "Solving linear equations over $\mathrm{GF}(2)$ via block Wiedemann algorithm," *Math. Comp.*, vol. 62, no. 205, pp. 333–350, Jan. 1994.

[7] D. H. Wiedemann, "Solving sparse linear equations over finite fields," *IEEE Trans. Inform. Theory*, vol. IT–32, no. 1, pp. 54–62, Jan. 1986.

[8] E. Kaltofen and A. Lobo, "Distributed matrix-free solution of large sparse linear systems over finite fields," *Algorithmica*, vol. 24, no. 4, pp. 331–348, 1999.

[9] E. Thomé, "Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm," *J. Symbolic Comput.*, vol. 33, no. 5, pp. 757–775, Jul. 2002.