Noname manuscript No.
(will be inserted by the editor)

A Heterogeneous Computing Environment to Solve the 768-bit RSA Challenge

Thorsten Kleinjung[1] · **Joppe W. Bos**[1] · **Arjen K. Lenstra**[1] · **Dag Arne Osvik**[1] · **Kazumaro Aoki**[2] · **Scott Contini**[3] · **Jens Franke**[4] · **Emmanuel Thomé**[5] · **Pascal Jermini**[6] · **Michela Thiémard**[6] · **Paul Leyland**[7] · **Peter L. Montgomery**[8,9] · **Andrey Timofeev**[9] · **Heinz Stockinger**[10]

the date of receipt and acceptance should be inserted later

Abstract On December 12, 2009, the 768-bit, 232-digit number RSA-768 was factored using the number field sieve. Overall, the computational challenge would take more than 1700 years on a single, standard core. In the article we present the heterogeneous computing approach, involving different compute clusters and Grid computing environments, used to solve this problem.

Keywords RSA · HPC · integer factorization

1 Introduction

Scientific calculations in domains such as physics (fluid dynamics, high-energy physics etc.), chemistry (quantum chemistry, molecular modeling, etc.), biology (large scale genomic or proteomics projects), and climate modeling often require massive amounts of computing power. If the kind of problem calls for tightly coupled massively parallel computing, it is often well suited for supercomputers and stands a good chance to attract funding both to get access to the proper hardware and to develop suitable software. In contrast, embarrassingly parallel applications, where any number of stand-alone devices will do, are often not welcome on classical supercomputers but are instead relegated to high-performance computing clusters, Grid infrastructures, or desktop computing environments such as BOINC or Condor.

The latter category includes cryptographic applications. Although they are not less CPU-demanding than other scientific applications, they are not considered to be of much interest by the HPC community. Despite this disregard, many cryptographic problems are computationally, mathematically and algorithmically challenging and non-trivial, and practically relevant for users of cryptography (i.e., almost everyone these days). One of these problems is *integer factorization*, the subject of this paper: we present the heterogeneous computational infrastructure that was used to set a new integer factorization record by factoring the challenge number RSA-768, a 768-bit, 232-digit composite integer [11].

We used the approach commonly used to factor large integers since the late 1980s. Thus, we did not aim for a single, homogeneous computing environment or single supercomputer, but used a heterogeneous compute environment consisting of several cluster and Grid environments (in Australia, Japan, and many European countries), different operating systems, job execution environments and technical personnel. Several technical as well as non-technical reasons have led us to a such a heterogeneous approach:

– Single Grid infrastructures such as the European Grid infrastructure: only a limited number of resources (for a limited time) in the European Grid

[1]EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland
· [2]NTT, 3-9-11 Midori-cho, Musashino-shi, Tokyo, 180-8585 Japan
· [3]Research performed while at Macquarie University, Sydney, Australia
· [4]University of Bonn, Department of Mathematics, Beringstraße 1, D-53115 Bonn, Germany
· [5]INRIA CNRS LORIA, Équipe CARAMEL - bâtiment A, 615 rue du jardin botanique, F-54602 Villers-lès- Nancy Cedex, France
· [6]EPFL AI DIT, CP 121, CH-1015 Lausanne, Switzerland
· [7]Brnikat Ltd, 19a Hauxton Rd, Little Shelford, Cambridge, CB22 5HJ, United Kingdom
· [8]Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
· [9]CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
· [10]Swiss Institute of Bioinformatics, Vital-IT Group, CH-1015 Lausanne, Switzerland

environment could be made available to the RSA-factorization problem. Therefore, even though using a Grid-only environment is promising and convenient, we had to look into alternative approaches.
- Individual clusters at different sites are available but are managed in different ways (i.e., different local resource management systems, different file systems, different hardware, different site expertise with respect to the RSA application, etc.). Again, standardized Grid protocols and middleware would have helped to address the problem in a more homogeneous way but the clusters we used are not part of a single Grid environment.
- A diversified, heterogeneous environment better represents the Internet as a whole than any organized computer infrastructure. It thus allows a more reliable estimate of the havoc miscreants could cause by large surreptitious cryptanalytic efforts.

In brief, the RSA-768 challenge has the following features and computational stages. The first main stage is the most CPU intensive and embarrassingly parallel part of the computation and also created most data. Compressed about 5 TB had to be transferred over the Internet, at a rate of about 10 GB per day, to a central location where, once enough data had been collected, the second main stage was prepared. This stage is not embarrassingly parallel and was traditionally done on a single supercomputer or, later, on a single tightly coupled cluster. Lacking access to sufficiently large clusters for a long enough period of time, we had to adopt a somewhat more complicated approach that allows usage of a limited number of tightly coupled clusters at different locations. As a result, roughly 100GB of data had to be distributed, using the Internet again, to the contributing clusters (restricting to those in France, Japan, and Switzerland). They worked independently, with the exception of an intermediate step that had to be done on a single tightly coupled cluster with 1 TB of RAM. This implied, yet again, a substantial data exchange. In summary, the RSA-768 challenge was solved using a truly heterogeneous infrastructure with no communication among the contributors nor computational cores except to break the computation into independent tasks.

Given the embarrassingly parallel nature of the first main stage, the most CPU intensive part of the computation, we could have used one of the tools that allow contributions by volunteers on the Internet. That is how it was done in the late 1980s and early 1990s, using crude email and FTP-based precursors of currently popular tools. Indeed, an ongoing integer factorization BOINC project [14] does just that. Having managed several such proof-of-concept projects from 1988 until about 1995, we found that a lot of time goes into addressing simple concerns of the contributors, in particular if considerable RAM resources are required (as was the case for RSA-768 with at least one but preferably two GB RAM per core). We chose to limit our project to a limited set of dedicated and knowledgeable researchers who could be expected to resolve occasional issues themselves.

Section 2 gives the cryptographic motivation, Section 3 presents the computational steps in more detail, and Section 4 describes the heterogeneous compute infrastructure used.

2 Cryptographic Background and Motivation

Cryptography is ubiquitous on the Internet. Authentication methods used by browsers, Grid computing applications, and websites accessed through HTTPS commonly rely on X.509 certificates based on the RSA cryptosystem. This means that the security depends on the hardness of factoring an appropriately chosen integer, typically of 1024 or 2048 bits: factoring that integer could undermine the security of that particular instance of the HTTPS protocol.

A 1024-bit integer has more than 300 decimal digits, a 2048-bit one twice as many. Dealing with such large numbers makes protocols unwieldy. Smaller numbers would be more efficient – but they are easier to factor and give less security. One would like to use the smallest key size that gives an acceptable level of security. Once a certain size has been picked it is hard to modify the choice. But integer factoring constantly gets easier, not just because computers keep getting faster but also because factoring methods keep getting better. So far this goes at a fairly steady pace. However, discovery of an efficient factoring method cannot be ruled out: it is conceivable that from one moment to the next the entire information security infrastructure collapses[1]. This would affect much more than just web-security, since most of the public and private sectors use the same cryptographic methods as HTTPS does. A "disaster" of this sort has not occurred yet – at least, not that we are aware of – and most of us simply hope that it will not happen either.

The steady progress is taken into account in cryptographic standards that prescribe key sizes for RSA. At this point in time we are on the verge of an important transition: the USA National Institute of Standards and Technology (NIST) recommends phasing out 1024-bit

[1] Integer factorization is easy on a quantum computer [16]. Quantum computers have not been realized yet. Estimates and opinions on this subject vary.

RSA by the end of the year 2010 and to adopt 2048-bit RSA or other systems of comparable security. This does not mean that by the end of 2010 integers of 1024 bits can suddenly be factored. It means that the security of 1024-bit RSA is perceived to become too low and that, indeed, several years down the road, factoring 1024-bit integers may become feasible.

How does our effort, factoring a 768-bit RSA challenge key, fit in this picture? We now know what effort sufficed to factor a 768-bit integer. Interestingly, it turned out to be an order of magnitude easier than predicted by some in the field [15]. Combined with a theoretical analysis we can now more accurately predict what would be required to factor a single 1024-bit RSA challenge. Unless an integer factoring breakthrough occurs, we are convinced that an effort on a scale similar to ours will have no chance of success within the next five years. After that, all bets are off [11]. Thus, NIST rightly encourages phasing out 1024-bit RSA but there is no need to rush into a costly, overhasty security upgrade. There is almost certainly no risk in a more economical gradual adoption of the new standards, as long as the transition is complete by the year 2014.

Furthermore, our result shows that an effort of this sort can be pulled off even if there is no uniformity in the platforms used or in the way they communicate. This requires an extra degree of prudence when selecting cryptographic key sizes, as the enormous computational power of Internet as a whole, or a substantial fraction thereof, can in principle be harnessed for similar cryptanalytic calculations.

A project of this sort is scientifically interesting in its own right as well. When trying to solve larger problems there are always new challenges that must be dealt with. For instance, for a previous large scale effort [1], when dividing the second main step over a number of independent tightly coupled clusters, we had not realized that the faster clusters would finish their task much earlier than the slower ones, quite simply because we had never ran this step for such a long time in such a heterogeneous environment. Faced with the "threat" of long idle times on some clusters, a new algorithmic twist was developed allowing total flexibility in task sizes, eliminating all idle times. It considerably facilitated management of the present project where the new approach was used for the first time. This project, in turn, triggered algorithmic advances for new types of processors, adapting not just to multicores but also to the decreasing amount of RAM per core. This will prove useful in later projects and will greatly influence the feasibility of a 1024-bit factoring attempt.

3 The Computational Challenge

When faced with a factoring problem, one first checks for small factors. For RSA challenges this step can be omitted, since the RSA challenge numbers are known to be the product of two primes of about the same size and therefore they have no small factor. The fastest known algorithm to factor RSA challenges is the *number field sieve* (NFS, [12]), which works by *combining relations*, as illustrated below. The two main steps mentioned in Section 1 are the most CPU intensive steps of NFS: in the first step relations are generated, in the second step they are combined. In this section we present a more complete outline of NFS, concentrating on the computational effort and data sizes for RSA-768, while avoiding all underlying mathematical details. In brief, we describe the five main steps of the overall workflow.

We give a simple example to show how relations are combined to factor an integer while avoiding the intricacies involved in the NFS. For the integer 143 a *relation* would be given by $17^2 \equiv 3 \bmod 143$ because the difference $17^2 - 3$ is an integer multiple of 143. Similarly, $19^2 \equiv 3 \cdot 5^2 \bmod 143$ is a relation. These relations can be *combined* into the relation $17^2 \cdot 19^2 \equiv 3^2 \cdot 5^2 \bmod 143$ with squares on both sides. The square roots $17 \cdot 19$ and $3 \cdot 5$ of both sides follow immediately. The greatest common divisor of 143 and the difference $17 \cdot 19 - 3 \cdot 5$ of the square roots turns out to be 11, a factor of 143. Relations for NFS are more complex and involve *algebraic integers*. Defining those requires proper polynomials, which are selected in the first step of NFS:

1. Preparatory step: Polynomial selection. The runtime of NFS depends strongly on the parameter choice. The most important choice is that of a pair of irreducible polynomials f, g that define two *algebraic number fields*. For RSA-768 we fixed $\text{degree}(f) = 6$ and $\text{degree}(g) = 1$, implying that the number field defined by g is the field $\mathbf{Q}$ of the rational numbers. The best current method to find good f and g is a mathematically rather sophisticated, embarrassingly parallel search. A good pair was found in 2005 already, after three months on 80 AMD Opteron cores in Bonn. A comparable effort at EPFL, in 2007, did not turn up a better pair. Overall, more than $2 \cdot 10^{18}$ pairs were considered, at a rate of 1.6 billion pairs per core per second. Although this is a considerable computation, as it would have required a day of computing on a 15 000 core cluster, it is dwarfed by the other steps.

2. First main step: Sieving. In this step many relations are sought: co-prime pairs of integers a, b such that $f(a,b) \cdot g(a,b)$ has no large prime factors. How many relations are needed depends on the size of those prime

Table 1: For each range of special q values, the contributor that sieved that range is listed, along with the amount of RAM used for the sieving program, the number of relations found, and the approximate number of relations that was found per task. The number of relations per task decreases with increasing special q values. Independently, the number of tasks per fixed length range decreases because the number of primes per range decreases. Together these effects contribute to the overall drop in the number of relations found per fixed length range (as exemplified by the two arrows), though for some machines it turned out to be faster to produce fewer relations per range. Note also that, on average, fewer relations are found per task if less RAM is available: except for the low range of special q values, we therefore preferred to use machines with at least 2GB RAM per core to sieve the more productive ranges.

q-range (millions)	contributor	RAM	number of relations found	percentage of relations	relations per task	percentage of tasks
100- 170:	EPFL Greedy (89% done)	.5GB	530 837 179	0.83%	159	0.72%
170- 400:	not assigned					
400- 444:	CWI	.5GB	493 758 264	0.77%	223	0.48%
444- 450:	not assigned					
450- 1100:	NTT	1GB	6 040 634 011	9.39%	190	6.84%
1100- 1200:	EPFL Lacal304	2GB	1 085 485 063	1.69%	227	1.03%
1200- 1500:	EGEE	1-2GB	2 906 539 451	4.52%	204	3.06%
1500- 2000:	Bonn	2GB	4 953 637 869	7.70%	211	5.05%
2000- 2035:	AC3	1.7GB	278 083 916	0.43%	170	0.35%
2035- 2100:	EPFL { Callisto, Lacal140, Lacal304 }	2GB	583 487 657	0.91%	193	0.65%
2100- 2400:	EPFL Lacal304	2GB	2 644 305 334	4.11%	204	2.79%
2400- 2500:	INRIA	2GB	889 307 119	1.38%	192	1.00%
2500- 2600:	INRIA	1-2GB	729 836 401	1.13%	158	0.99%
2600- 2700:	EPFL Lacal304	2GB	811 399 503	1.26%	176	0.99%
2700- 2800:	CWI	1-2GB	742 575 917	1.15%	161	0.99%
2800- 3000:	INRIA	2GB	1 633 654 656	2.54%	178	1.97%
3000- 3300:	EPFL Callisto	2GB	2 256 163 004	3.51%	164	2.96%
3300- 3600:	EPFL Lacal140	2GB	2 177 658 504	3.38%	159	2.95%
3600- 4000:	INRIA	1-2GB	2 526 184 293	3.93%	139	3.91%
4000- 4200:	INRIA	2GB	1 449 153 442	2.25%	160	1.95%
4200- 4600:	INRIA	1GB	2 320 916 889	3.61%	129	3.87%
4600- 4700:	not assigned					
4700- 4760:	NTT	1GB	273 747 997	0.43%	102	0.58%
4760- 4800:	Bonn	2GB	258 785 877	0.40%	144	0.39%
4800- 5200:	EPFL Lacal304	2GB	2 554 062 089	3.97%	143	3.84%
5200- 5400:	EPFL Lacal140	2GB	1 245 110 392	1.94%	139	1.93%
5400- 5600:	EPFL Callisto	2GB	1 235 783 457	1.92%	139	1.91%
5600- 5800:	EPFL Lacal304	2GB	1 219 439 733	1.90%	137	1.91%
5800- 6000:	EPFL Callisto	2GB	1 202 926 042	1.87%	135	1.92%
6000- 6200:	EPFL Lacal140	2GB	1 182 875 721	1.84%	133	1.91%
6200- 6300:	INRIA } not done					
6300- 6500:	EPFL Lacal304 } not done					
6500- 7000:	INRIA	1-2GB	2 476 812 744	3.85%	112	4.76%
7000- 7900:	NTT	1GB	3 574 335 463	5.56%	90	8.54%
7900- 8900:	INRIA	1GB	4 589 325 052	7.13%	105	9.40%
8900- 9300:	INRIA	1GB	1 776 088 161	2.76%	102	3.75%
9300- 9400:	CWI	1-2GB	495 380 881	0.77%	114	0.93%
9400- 9500:	EPFL Greedy (80% done)	1GB	351 107 747	0.55%	101	0.75%
9500- 9600:	Leyland	1GB	443 023 506	0.69%	102	0.93%
9600-10000:	INRIA	1GB	1 729 354 187	2.69%	99	3.76%
10000-11000:	INRIA	1GB	4 201 641 235	6.53%	97	9.32%
11000-11100:	CWI	1-2GB	471 070 974	0.73%	109	0.93%

factors. For RSA-768 we used the bound 2^{40}. We analyzed that enough relations could be found by searching through $|a| < 6.3 \cdot 10^{11}$ and $0 < b < 1.4 \cdot 10^7$. This implies that for more than 10^{19} co-prime pairs a, b the value $f(a,b) \cdot g(a,b)$ had to be tested for divisibility by the almost 38 billion primes $< 2^{40}$. Per prime p this can be done for many a, b pairs simultaneously using *sieving*: if a polynomial value such as $f(a,b)$ is a multiple of p, then so is $f(a+mp, b+np)$ for integers m and n.

The sieving can be distributed, in an embarrassingly parallel fashion, by assigning disjoint ranges of b-values to different contributors. Given a b-value one just sieves all $|a| < 6.3 \cdot 10^{11}$. This straightforward approach was used in the earliest distributed NFS factoring efforts. A more efficient and still embarrassingly parallel strategy is to assign disjoint ranges of primes q to different contributors, and to limit the search, given such a *special* q, to all relevant a, b pairs for which $f(a,b)$ is divisible by q. Each special q results in a number of different sieving tasks that varies from zero to degree(f) (i.e., six, for RSA-768). This approach, which is a bit harder to program, has gained popularity since the mid 1990s. We used it for RSA-768 and we could fully inspect 15 to 20 million a, b pairs per second on a 2.2GHz core with 2GB RAM. Sieving task were distributed among the contributors depending on their available computer resources.

Overall, about 465 million sieving tasks were processed, for special q values between 10^8 and $1.11 \cdot 10^{10}$. An average sieving task took about 100 seconds on a

core as above, and resulted in about 134 relations at about 150 bytes per relation. Sieving started in the summer of 2007 and lasted for almost two years. With about 1500 core years, we achieved a sustained performance equivalent to more than 700 cores, full time for two years. Table 1 gives a breakdown of the ranges of special q values processed by the different contributors. Section 4 presents more details of the infrastructures used. Including duplicates, in total more than 64 billion relations were generated. They were collected at EPFL, with several backups, also off-campus.

The main input for a processor contributing to the sieving is the range of special q values to be processed. The number of sieving tasks per range behaves roughly as the number of primes in it. Thus, it slowly drops off with increasing q values, and for a range $[L, U]$ can be estimated as $\frac{U}{\ln(U)-1} - \frac{L}{\ln(L)-1}$. For a range of length 1000 with $L \approx 10^9$ this results in about 48 sieving tasks, which is reduced to about 43 for $L \approx 10^{10}$. Such ranges can typically be processed in less than two hours.

Compressed storage of the relations, along with the factorizations of the $f(a, b) \cdot g(a, b)$-values, took 5 TB. This amount of storage is by no means exceptional, and should not be hard to deal with. Nevertheless, storage problems caused most stress while sieving for RSA-768, mostly due to the lack of reliability of the storage devices. Not just disks failed (with RAID servers as a first line of defense), disk casings failed as well, with unforeseeable consequences for the disks and RAID servers. In the course of the sieving we decided to hedge our bets by spreading the risk over a variety of manufacturers and vendors. Additionally, human errors are unavoidable and rigid rules had to be enforced to minimize the consequences.

Otherwise, sieving is the least stressful step of NFS, as it is not just embarrassingly parallel but also tolerant to sloppiness and errors. All that counts is if ultimately enough relations will be found, for the rest one mostly needs patience. It does not matter what special q was used to find a relation, and it does not matter much – except for a minor loss of efficiency, and unless it occurs systematically – if not all special q values in a range are properly processed or if occasionally some of the data generated gets lost or corrupted (as the correctness of a relation can easily be verified at the central repository: anything that is not correct is simply discarded). A crash of one or more processors does not affect the results of any of the other processors, and a task that may be left unfinished due to some mishap can be reassigned to another processor or it can be dropped altogether.

3. Intermediate step: Filtering. After duplicate removal, useless relations are removed. These include, for instance, relations for which $f(a, b)$ contains a prime factor that does not occur in any other relation. This can only conveniently be done if all data reside at a single location. The surviving relations are used to build an over-square bit-matrix with rows determined by the exponent-vectors of the primes in the remaining $f \cdot g$-values, or combinations thereof. While building this bit-matrix, many choices can be made. Because dependencies among the rows will be determined in the next step, it pays off to aim for a low dimension and overall weight (i.e., number of non-zero matrix entries).

For RSA-768, the 64 billion relations resulted in 48 billion non-duplicates. Several matrices were built, the best of which had 193 million rows and 28 billion non-zero entries. It required about 105GB of disk space. The entire process to convert the raw relations into a matrix took about two weeks of computing on a 304-core cluster at EPFL: relatively speaking quite modest but rather cumbersome as large amounts of data had to be moved around.

4. Second main step: Matrix. Although the sieving step requires more CPU time, the matrix step is considered to be the most challenging step of current large scale factoring efforts. Gaussian elimination was used for factoring related matrices until the early 1990s. It was abandoned in favor of the block Lanczos algorithm [6] which requires much less time and memory due to the sparseness of the input matrix. A disadvantage of block Lanczos (which it shares with Gaussian elimination) is that it does not allow independent parallelization: it must be run on a single tightly coupled massively parallel machine. We are still in the process of evaluating the feasibility of doing this step using block Lanczos at a single location.

Because of this disadvantage, we preferred block Wiedemann [7]. Though not embarrassingly parallel, the computation can be split up into a limited number of chunks. Each chunk can be processed on a tightly coupled cluster, independently of the other chunks each of which may simultaneously be processed at some other location.

More precisely, block Wiedemann works in three stages: a first stage that can be split up as above, a brief central stage that needs to be done at one location, and a final stage that is less work than the first stage and that can be split up into any number of chunks if enough checkpoints are kept from the first stage. The first and final stages both consist of iterations of matrix $\times$ vector multiplications, where the matrix is the fixed, sparse bit-matrix resulting from the filtering step, and where the (bit-)vector is constantly updated (as

the result of the previous multiplication). It is possible to use k different initial bit-vectors and to reduce the number of multiplications per bit-vector by a factor of k, as long as the total number of multiplications by bit-vectors remains constant. This comes at various penalties, though, and complicates the brief central stage if k gets large.

For RSA-768 we used $k = 8 \cdot 64 = 512$ and 8 chunks each of which processed 64 bit-vectors at a time. Per chunk, 565 000 matrix × vector multiplications had to be done in the first stage, 380 000 in the third stage, for our matrix of dimension 193 million with 28 billion non-zero entries. Running a first or third stage chunk required 180GB RAM. Table 3 in Appendix A lists the various clusters used for the first and third stage, along with the time required per multiplication per chunk. Obviously, the timings vary considerably depending on the type of processor, number of cores, and type of interconnect. For instance, on 12 dual AMD 2427 nodes (hex-core, thus 144 cores, with 16GB RAM per node) with InfiniBand, a multiplication takes about 4.5 seconds. This implies that on 48 such nodes (576 cores – 56 such nodes with 672 cores were installed at EPFL while the first stage was underway) all eight chunks for stages one and three could have been completed in about 100 days, for about 160 core years of computing. The central stage took a bit more than 17 hours on the 56 freshly installed nodes, using all available 896GB RAM (except for a short period when a terabyte was needed and swapping occurred), but just 224 of the 672 available cores. On the variety of clusters that was actually used the entire block Wiedemann step took 119 days.

Unlike sieving, no errors can be tolerated during the matrix step. The iterations thus included frequent checkpoints to ensure that the computation was still on-track. We experienced no glitches. In the original distributed block Wiedemann all chunks consisted of an equal amount of work, i.e., the same number of multiplications on matrices and vectors of identical sizes. We used a more flexible version of the algorithm, so that faster jobs can do more multiplications and slower ones fewer, as long as the same overall number of multiplications as before is reached.

5. Finishing up: Square root. Finding out if the dependencies as produced by the matrix step are correct is probably the most nerve-racking part of any large scale factoring project. Also from a mathematical point of view, deriving a factorization from a dependency is one of the more exciting steps. Computationally speaking, however, it is usually the least challenging step. For RSA-768 it took about one core day and resulted in the following factorization.

RSA-768 =
12301866845301177551304949583849627207728535695
95334792197322452151726400507263657518745202199
78646938995647494277406384592519255732630345373
15482685079170261221429134616704292143116022212
40479274737794080665351419597459856902143413
=
33478071698956898786044169848212690817704794983
71376856891243138898288379387800228761471165253
174308773814467999489
×
36746043666799590428244633799627952632279158164
34308764267603228381573966651127923337341714339
681027009279873630891 7.

The correctness of the result, once obtained after 1700 core years of computing, can be verified in a fraction of a second.

Table 2 gives the overall workflow, along with the percentages contributed (for the sieving measured in different ways).

4 Heterogeneous Compute Infrastructure

In this section we describe the heterogeneous environment used for the factorization of RSA-768, with a focus on the management of the sieving step.

As set forth in Section 3, sieving consists of processing a range of special q values, where for each special q value at most six sieving tasks have to be performed. Given a range, this is carried out by a `C` program. This program, called `lasieve`, resulted from many years of research, development and refinements at the university of Bonn. All collaborators got statically linked versions of `lasieve`, geared toward their hardware (processor type and cache size) and operating systems. The number of relations found per task drops off with increasing special q values. One therefore tries to completely process all smaller ranges before moving to larger ones, leaving as few unprocessed gaps as possible.

At the highest level, EPFL distributed relatively large, disjoint ranges of special q values among the collaborators, depending on the specifics of the cluster(s) or machines to be used. The way a range is processed depends on how `lasieve` is run, the cluster usage agreements, and the job scheduler. In any case, a large range assigned to a site must be partitioned into smaller subranges, each of which can be processed in a reasonable amount of time by a CPU core running `lasieve`: as mentioned, a range of length 1000 takes about two

Table 2: Workflow of the project. The sieving percentages are only rough indications for the overall sieving contributions because tasks for larger special q values are less productive, and as a consequence also faster to process: the truth is biased toward the relation contribution percentage. The last two rows contain approximations for core years ("cy") spent and dates ($yyyy$:mm:dd) of the calculation.

polynomial selection		sieving		filtering	matrix stage 1	matrix stage 2	matrix stage 3	squareroot
Bonn	Lacal140			Lacal304	Callisto 1.8% Lacal672 32.5% INRIA 46.8% NTT 18.9%	Lacal672	Lacal672 78.2% INRIA 17.3% NTT 4.5%	Lacal672
		relations	*tasks*					
		AC3	0.43% 0.35%					
		Bonn	8.10% 5.44%					
		Callisto	7.60% 7.01%					
		CWI	3.42% 3.33%					
		EGEE	4.52% 3.06%					
		Greedy	1.37% 1.47%					
		INRIA	37.80% 44.68%					
		Lacal140	7.46% 7.01%					
		Lacal304	13.23% 10.79%					
		Leyland	0.69% 0.93%					
		NTT	15.37% 15.96%					
20 cy	20 cy	≈ 1500 cy		< 12 cy	95 cy	< $\frac{1}{2}$ cy	63 cy	< $\frac{1}{10}$ cy
2005	2007:06	2007:08 - 2009:06		2009:08	2009:09-11	2009:11:03	2009:11-12	2009:12:12

hours to process. The naive approach to assign subranges is to do so upfront at the job scheduler's level before any particular CPU core has been allocated to process that subrange using `lasieve`. It allows for *manual* range partitioning and assignment. This works if, barring exceptional irregularities, one may assume that all jobs, once put in the queue to be executed, will eventually be taken into execution and that, when taken into execution, they will finish their allotted range. This situation may apply if one is the sole user or owner of a desktop machine or cluster, or if otherwise favorable access conditions have been granted to the compute resources. It applied to some contributors.

Even so, several set-ups used for the sieving used an *automated* approach where range assignment is postponed to the moment that a CPU core is ready to start sieving. It avoids range fragmentation caused by the apparently unavoidable fact of life that in some environments there are always jobs that disappear from the queue without ever having been taken into execution. Nevertheless, and in either case, it may be desirable to conduct post-mortems of occasional crashes. This would involve cumbersome analysis of partial output files to extract (and reassign) previously assigned but unfinished ranges. Several such systems were used (and are described below) that are semi-automated in the sense that ranges were assigned automatically, but that make the implicit assumption that range fragmentation will be kept to a minimum, i.e., that normally speaking assigned ranges will be fully processed and will not be left unfinished. We stress again that the existing systems that we used are heterogeneous and do not share a common software layer. Additionally, the systems were not always available at or for the same time, and resource allocation and availability was not guaranteed neither at the beginning nor during the computational runs.

This assumption, which is based on a 100% completion model of assigned ranges, can certainly not always be made. Traditionally, sieving jobs are only run on processors that would otherwise be idle. For example, in [13], the more than 20 year old, first collaborative sieving effort that we are aware of, usage is cited of a "machine idle" tool to identify machines that have not recently been used and that thus may be added to the pool of sievers. However, sieving jobs were terminated instantaneously as soon as a machine was reclaimed, for instance by hitting a key. See also [10]. In the cluster job scheduling system OAR [4] the possibility to exploit otherwise idle resources in a similarly volatile way is created by *best-effort* jobs. With such jobs, partial processing of assigned ranges is systematic, making extraction of parts of ranges that are left unfinished mandatory, and resulting in range fragmentation that quickly becomes unmanageable for humans. A convenient way to fully automate range management – including reclaiming ranges from interrupted jobs – was implemented at INRIA and is described below.

Compared to the sieving jobs, the matrix jobs require very close supervision. Gaps cannot be tolerated, and work left unfinished by crashed jobs has to be completed starting from the most recent checkpoint. Thus, preferably the matrix is not done using best-effort types of jobs but using supercomputers or relatively large dedicated (sub)clusters for extended periods of time. This is what we did for the RSA-768 project, simultaneously using various clusters, all with manually managed jobs, only a small percentage of which were best-effort jobs. In particular during stage 3 of the matrix step the three participants contributing to that part of the calculation frequently discussed task assignments, with

some clusters taking over jobs previously assigned to others. The matrix step is not further discussed below. Some details on how and where it was run are listed in tables 2 and 3.

4.1 AC3

The Australian Centre for Advanced Computing and Communications provides high performance computing platforms for academic and research staff at eight Australian universities. A selection of machines is available for staff to apply for system units of computing time. We were allocated computing resources equivalent to full-time use of 16 nodes of the machine Barossa, a Dell Beowulf cluster having 155 3GHz dual processor Pentium 4 nodes with 2GB RAM per core. Of this memory, each node reserves some for the operating system, the batch queue system, and video sharing. As a result, only about 1.7GB are available for submitted jobs. Although less than the ideal 2GB, we submitted jobs restricted to 1.7GB RAM and to 2GB of virtual memory, and these jobs ran nearly as efficiently as machines that had the full 2GB RAM. So, swapping was minimal.

Submission of jobs to Barossa is via the PBS batch queue system. In addition to the memory restrictions, a job is only allowed to run for two days or else an automated program kills the violating submissions. From a few experiments, we determined how many special q values could be handled safely under the two day limit. A simple shell script was used to submit new ranges while keeping track of what ranges had already been done. Running of this shell script was done manually on a daily basis. Moreover, uploading of the data to EPFL and dealing with the rare occurrences of failed jobs was also done manually. Participation was cut short when the Australian participant (Scott Contini) left Macquarie University for a full-time position in industry.

4.2 CWI

At the Centrum Wiskunde & Informatica we utilized workstations outside the usual office hours. All workstations run a recent version of Fedora Linux. The home and project directories are hosted on the NFS[2] file server located at SARA (the Academic Computing Centre Amsterdam), over a network based on UTP switched gigabit Ethernet.

All workstations that participated are x86-64 machines with varying numbers of cores, clock rates, cache sizes and amounts of memory per core. At the outset, most were single-core machines with 1GB RAM, with a small number of Intel dual-core machines. During the sieving, almost all single-core machines were replaced by dual-core ones, whereas some dual-cores were replaced by quad-cores with 2GB RAM per core. We had 120 to 180 cores at our disposal, depending on hardware upgrades and on users willing to share their workstations.

After getting a large special q range from EPFL, we used a script to generate jobs that invoke `lasieve` on subranges that a single core can complete within three to five nights or a weekend. All jobs were placed in the input queue which is located at the NFS file server accessible from every workstation. The jobs were managed and run as follows:

`factord`. On each participating workstation, the shell script `factord` was invoked by crontab after working hours. It manages the supply of jobs in the following manner:

1. checking existence in the machine's working directory of a checkpoint file of a previous job, and if so re-invoking that job (using `sieving task` as described below).
2. fetching a new job from the input queue if a checkpoint file is not present;
3. moving output produced by jobs that completed their range to the output queue;
4. terminating if the input queue is empty;
5. sending a termination signal to `lasieve` early in the morning on working days.

Early on in the project, the simultaneous start of many sieving jobs crashed the `automounter` daemon on some single-core machines, thereby preventing `lasieve` to start. This was solved by randomly spreading the starting times over a period of half an hour. When invoked, `factord` reschedules its next start. The script is used to manage other factorization projects as well.

`sieving task`. This is a shell script that ensures the proper start of `lasieve`. If a checkpoint file exists in the machine's sieving working directory, the script resumes `lasieve` with the old configuration, starting from the last used special q. Otherwise, `sieving task` determines the number of available cores and RAM per core, in order to set proper input parameters for `lasieve`. On many multi-core machines we utilized all but one core, keeping one core available for applications by the workstation's owner.

`monitoring`. A monitoring tool was used to check regular progress of all jobs. If a job is found to be stalled, for instance due to a user program or hard-

[2] In this section NFS stands for Network File System instead of Number Field Sieve.

ware failure, the tool moves all relations from the machine's working directory to the output queue and reassigns the remaining special q values to a new job which is put in the input queue. Once the host is available for sieving again, it fetches a new job from the input queue as described above.

4.3 EGEE

The infrastructure provided by the Enabling Grids for E-SciencE (EGEE, cf. [8]) is the biggest production Grid infrastructure in the world. It is open to various types of scientific domains, applications and users. Typically, scientific applications are organized in Virtual Organizations (VOs) and are shared among several users. Since there was no VO available that would suit our factoring attempt, we first used an existing VO and later created our own crypto VO.

As we demonstrated in [17] the gLite [9] Grid middleware that underlies EGEE's job submission and execution, though suitable for embarrassingly parallel jobs, focuses on optimizing throughput for many users and applications rather than for a single application or user. This is due to a complex interaction of the Grid's meta-scheduler and each site's local resource management system, and is compounded by job queue latencies affecting perceived performance, scheduler failures that cause jobs to vanish (as noted above), and heterogeneous hardware causing diverse runtimes. We therefore adopted the approach proposed in [17] which integrates in the gLite Grid middleware a runtime-sensitive BOINC-like system with a task server, as illustrated in Figure 1. Compared to the traditional way EGEE jobs are handled, the main advantage of our approach is that it adapts automatically to the different runtimes required on the heterogeneous EGEE worker nodes, thereby maximizing throughput. The EGEE infrastructure, thus adapted, was successfully used for several months, processing up to a thousand ranges in parallel at more than 20 sites across Europe.

The overall workflow included the following software components.

`siever-submit.pl`. This perl script uses the gLite job submission command line interface to submit `siever-worker.pl` jobs to the gLite resource broker (Step 1a in Figure 1), which for each job selects a suitable worker node (depending on required RAM) and submits it there for execution (Step 1b in Figure 1). As long as a certain configurable minimum number of running jobs is not reached (we used from 100 to 1000 parallel jobs; once running, jobs may abort due to failure or because they exceed their maximum runtime), the script keeps submitting new jobs. Additionally, the script monitors how many jobs have finished (Step 6 in Figure 1) and displays the status on a webpage (Step 7 in Figure 1).

`siever-worker.pl`. This is an (in principle) everlasting perl script that runs on a worker node and that

1. attempts to obtain a range of special q values by submitting an HTTP request to the task server (steps 2 and 3 in Figure 1);
2. terminates if no range was received;
3. runs `lasieve` on the range obtained (Step 4 in Figure 1);
4. upon completion of the range, notifies the task server and transfers the results of the calculation to a Grid storage element (step 5a and 5b in Figure 1);
5. returns to Step 1.

Because several EGEE job submission systems limit job runtimes to 24 hours, the script (and thus its most recently spawned `lasieve` job) may be terminated, giving rise to assigned but unfinished ranges.

`Taskserver.pl`. The task server is initially loaded (Step 0 in Figure 1) with a special q range, partitioned in subranges of length 1000. If notification of completion of a range is not received within some fixed period of time (say, one day) after it was assigned, the task server returns the entire range to the pool. No attempts were made to avoid recomputation of data by analyzing partial output files to extract unprocessed subranges.

The task server has an HTTP interface for task assignment and management and, since the task server keeps track of completed ranges, to determine which output files have been uploaded to the Grid storage element. Internally, the task server uses a relational database management system back-end to manage tasks (start date, maximum allowed runtime to avoid zombie tasks, end date, etc.).

`gLite StorageElement`: This is a regular Storage Resource Manager (SRM) based Grid storage system that securely and reliably stores result files.

4.4 EPFL

EPFL DIT. Of the various computing resources provided by EPFL's Domaine IT (Central IT services), the Callisto cluster and the campus grid "Greedy" contributed to the sieving. Callisto is a general-purpose cluster, usable by EPFL researchers; Greedy is EPFL's desktop grid (also known as a campus grid), meant to

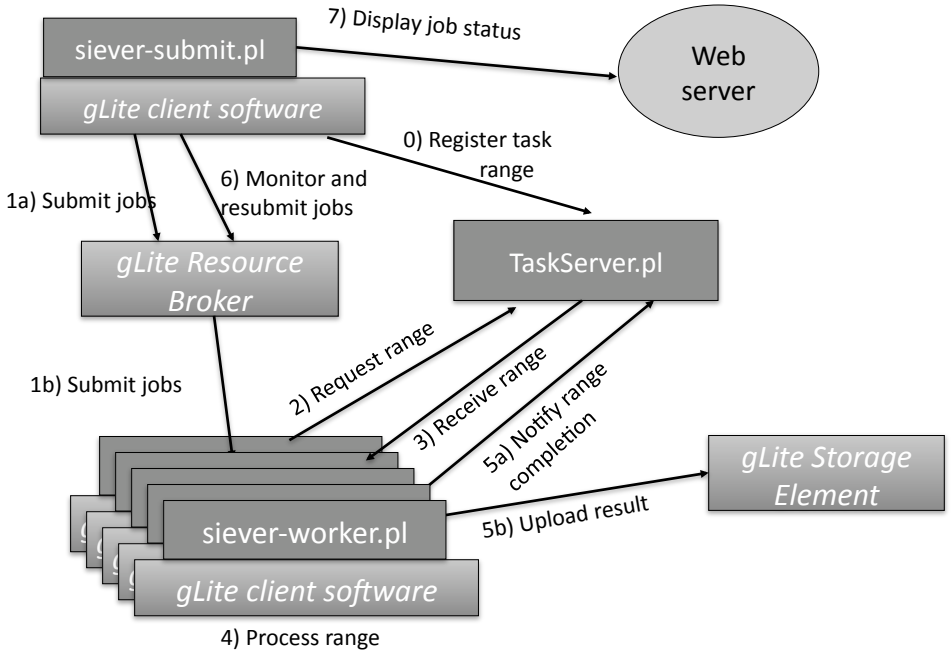

Fig. 1: Job submission and execution infrastructure are based on gLite (components indicated by italic font). The controlling script `siever-submit.pl` generates sieving jobs which are executed on nodes of the various EGEE clusters.

enable recovering unused CPU power. Table 3 lists some of the hardware specifications for both systems.

Callisto is served by two front-end nodes, also acting as file servers for an 18TB General Parallel File System (GPFS). Behind the two front-ends sit 128 dual quad-core compute nodes, interconnected via a fast Dual Data Rate (DDR) InfiniBand fabric. All the compute nodes are running SuSE Linux 10.2, and job scheduling is done with PBS ("Portable Batch System") Pro 10. For the sake of energy consumption and ease of administration, the compute nodes are in "blade" format: a single chassis can host up to 14 blades, leading to higher electrical efficiency and less cable clutter behind the racks. Furthermore, in an effort to relieve the air-cooling infrastructure, the cluster is installed in water-cooled racks, with almost no heat dissipation into the machine room air. At the start of the sieving, access to Callisto was free, though closely regulated with a focus on parallel jobs that can profit from the fast interconnect. In 2009 Callisto's access policies were changed, by requiring laboratories wishing to use the cluster to give a partial financial contribution toward its purchase.

Greedy is the second axis of the computing resources landscape of DIT (the third being a massively parallel super-computer), with an emphasis on grid computing and single-core jobs. Access to Greedy is free to all members of EPFL. The goal of Greedy is to federate unused CPU power across campus, by harvesting otherwise unused cycles of classroom and office PCs at EPFL. In order to not bother the user of the machine with continuous computations (fan noise can be bothersome in an office environment), grid jobs can run only during nights and weekends, when the probability of having someone working on the machine is low. Additionally, if user activity is detected while a grid job is running, the grid job will be suspended.

Greedy uses the software stack Condor [5], a "High Throughput Computing" middleware. All policies governing job startups and suspensions are made with Condor mechanisms, without relying on external tools. Besides two centrally-managed servers controlling the grid infrastructure, more than 1000 compute cores are available on the grid, mainly from classrooms. Greedy is a highly heterogeneous environment, with operating systems ranging from Linux on 32-bit machines to Windows 7 on 64-bit ones, and a variety of combinations of CPU types and memory amounts. Due to this heterogeneity, mechanisms are put in place that select the proper platform for a given set of jobs, so that they run

on the operating system/CPU combination they were compiled for.

EPFL LACAL. EPFL's Laboratory for Cryptologic Algorithms has a variety of clusters at its disposal. Two of these clusters (Lacal140 and Lacal304) were used for the sieving. A third (Lacal672) was used only for the matrix, as it was purchased when the sieving was already complete. Table 3 lists some hardware specifications for these three clusters.

Lacal140 is hooked up to a front-end with a 4.2TB NFS file system and a 6.1TB Parallel Virtual File System (version 2) provided by 8 IO servers. It is integrated in EPFL's Pleiades2 HPC Linux cluster, running SuSE Linux 10.2 and using Torque with Maui for job scheduling. Lacal304 ran Gentoo Linux, and used a single front-end server that acted as a file server for a 1.5TB NFS file system. Formerly located at EPFL's Sciences de Base clusterroom, Lacal304 was partially dismantled with the arrival of the new cluster Lacal672 (cf. below) and 24 of its nodes now serve as the freely accessible general purpose 192-core cluster VEGA at EPFL DIT.

The 56 Lacal672 nodes are connected to a single front-end server that acted before as Lacal304's file server (for the 1.5TB NFS file system that was used by Lacal304). It was installed, at EPFL's Sciences de Base clusterroom, while stage 1 of block Wiedemann was already in progress.

Sieving at EPFL. With the exception of the Greedy pool, Paul Leyland's `cabalc` and `cabald` were used to run sieving jobs on EPFL's DIT and LACAL clusters. This is fully described in Section 4.6.

On the Greedy pool Condor jobs are submitted with parameters for machines with 1GB RAM. Because most of the workstations on the grid run a Windows operating system, special binaries were created that do not use our fast assembly routines. As a consequence, processing a range of length 2000 would take five to six hours, i.e., a bit slower than usual but typically less than a night. Therefore, jobs with ranges of length 2000 were submitted to the grid, in batches of 5000 as that would produce at most 10 gigabytes of data. Transferring the output to the storage facilities at LACAL was done manually.

4.5 INRIA

The Aladdin-Grid'5000 ("`g5k`") is an HPC grid funded by several French research institutions, including INRIA, and intended for experimental research. Started in 2004 it consists of about 5000 CPU cores (taking the latest hardware upgrades into account there are currently more than 6400 cores), spread over nine sites across France. Each site hosts up to five clusters of identical nodes. The `g5k` clusters used for this project are listed in Table 3. Nodes at the same site have access to a shared NFS volume, but no global NFS filesystem is shared across the sites.

Access to `g5k` is not exclusively limited to experimental research projects. Long running, resource-hungry applications such as sieving are allowed as well, as long as they comply with the platform's policies for this type of jobs. Thus we agreed to limit our jobs to at most 25% of a site's resources at any given time, running at the lowest priority. In the job submission system OAR [4] used by `g5k` these are *best-effort* jobs, as mentioned above.

Premature termination of best-effort jobs is a normal event. If it occurs it affects all nodes allocated to the job. Notification of termination may never be received. Furthermore, as alluded to above, due to scheduler errors it cannot be guaranteed that all queued jobs are eventually taken into execution: sometimes jobs vanish. To deal with the range fragmentation that would result from premature termination, and to avoid fragmentation due to jobs disappearing from the queue, a framework consisting of simple shell and perl scripts was designed that resulted in a very effective and user friendly range management system. It was successfully used for about a year, contributing substantially to the sieving effort.

The workflow described below, as it applied to each of the `g5k` clusters that we used, is superficially similar to the EGEE workflow. The details are quite different.

OAR jobs. These are best-effort jobs, allocated to the nodes by the scheduler. The jobs may differ in the number of nodes targeted, but are otherwise identical and carry no information about the calculation to be performed. The number of OAR jobs that can be submitted at the same time is limited because, due to our best-effort constraint, in total never more than 25% of the nodes may be used, and because the job scheduler performs suboptimally if there are many jobs in the queue (recent software upgrades have improved it). More down-to-earth, the web-based grid occupancy visualization tool uses one line per job irrespective of the job's size: if many sieving jobs are displayed negative feedback can be expected. To deal with these issues, and to make sure that we always had jobs small enough to "fit in the holes", we submitted OAR jobs that allocate $n/2$, $n/4$, $n/8$, $n/16$, and $n/16$ nodes, where n is a quarter of the number of nodes at the site, targeting a total of n nodes.

An OAR job starts one core job on each CPU core of the nodes it had been given access to. Upon (expected) interruption of OAR jobs, a new one needs to be submitted to sustain the throughput. The required functionality of resubmitting interrupted jobs is provided by the OAR scheduler: so-called "idempotent" jobs, if they are left unfinished, may be restarted with the same command lines. Given our generic OAR jobs it thus sufficed to set their time limit to infinity (actually, one week), with the result that the scheduler made them persistent. As a result we did not have to rely on scripts that automatically submit jobs and that, in our experience, often lack the robustness they should have (due to communication glitches and time drifts).

Core jobs. All core jobs, over all nodes and all OAR jobs on the same cluster, are identical. A core job is an (in principle) everlasting shell script that sequentially performs the following steps.

1. It attempts to obtain a range of special q values that does not intersect with any other range that has been completed or that is currently under execution.
   As different core jobs may make concurrent requests for ranges of special q values, range allocation must be atomic. Because file renaming (moving) on an NFS partition is atomic, ranges may be claimed by a core job by trying to move a file containing a range from the `queue/` directory to the `inprogress/` directory. Per file the move is guaranteed to succeed for exactly one core job, which gets the claimed range. Core jobs that fail to move a file sleep for a couple of seconds before trying again.
   The `queue/` directory contains ranges to be processed encoded in names of otherwise empty files, allowing for convenient sequential processing of the available ranges assuming files are claimed in lexicographic order. Obviously, different clusters receive non-intersecting ranges.
2. It terminates if no range could be obtained.
3. It runs `lasieve` on the range obtained.
4. Upon completion of the range, it marks the output as clean.
   If a call to `lasieve` terminates because it finished the assigned range, the core job compresses the output created in the directory `working/`, moves it to `results/`, and removes the corresponding file from `inprogress/`, thereby marking that output as clean.
5. It returns to Step 1.

Although a core job is not meant to terminate (except on range starvation), it dies as soon as the scheduler decides to abort the best-effort OAR job that spawned it. Abrupt termination during execution of `lasieve` was handled as described below.

Watchdog job. The `working/` directory will contain partial output files of interrupted `lasieve` jobs, along with still active output files. The watchdog job identifies output files that have not been touched for longer than reasonable if its `lasieve` job were still alive (say, for 15 minutes, which is 5 to 10 times more than the expected delay between subsequent writes). It analyses each of these partial output files, returns the unprocessed part of the range to the `queue/` directory, renames and compresses the output file (possibly after truncation) so its name reflects the processed part of the range, moves it to the `results/` directory, and removes the corresponding file from the `inprogress/` directory. Due to the lightweight approach of encoding ranges in names of empty files the file system could easily cope with the range fragmentation.
This approach makes sure that, eventually, all special q values assigned to the cluster are processed, without human supervision. All that needed to be done was keeping an eye on the `queue/` directory to make sure that there was an adequate supply of ranges. The I/O and CPU footprint of the watchdog job are not significant, so it could be run on the submission front-end.

Data movement. Storage nodes on `g5k` are not meant to host large amounts of data. Results were therefore regularly copied from `g5k` to INRIA Nancy, where several partitions totaling 13TB were used for storage and backups. Relations were assembled to larger files corresponding to ranges of length e.g. 10^6 (about half a GB compressed), checked for correctness, and copied to EPFL.

4.6 Leyland

In comparison with the computation as a whole, Leyland's sieving contribution was relatively minor. At any one time at most 25 machines, most of them dual-core systems, were in use. Accordingly a relatively simple client / server harness was used to allocate special q ranges and a simple script used to automate uploading the results to an sftp site located at EPFL. Monitoring of progress, detection of error conditions and recovery from them was performed manually. The scripts running on the client and server side are named `cabalc` and `cabald`, respectively, because they were developed for the factorization of $2^{773}+1$ in 2000 by a team using the *nom-de-plume* 'The Cabal' [3], some of whom contributed to the factorization of RSA-768. The scripts,

described in more detail below, were also used for the sieving on the clusters at EPFL DIT and LACAL.

cabalc. `cabalc` uses a configuration file to specify the IP address of the machine running its `cabald`; the port on which to communicate; and a prototypical command to be run. It can execute an arbitrary command with parameters derived from a pair of numbers provided by `cabald`, allowing it to correctly run `lasieve` for any range of special q values.

Upon start, `cabalc` clears a Boolean 'work-to-do' variable, reads the server's address and port from the configuration file and enters an endless loop. There, it first attempts to open communications to the server. If nothing is forthcoming, `cabalc` waits for a few seconds and tries again. If 'work-to-do' is clear a request is made of the server for a special q range. The range is stored and 'work-to-do' set. The command given in `cabalc`'s configuration file is then run with proper command line arguments to process the newly received range. When that subprocess completes, `cabalc` returns to the start of its endless loop. This time around, 'work-to-do' is set so the saved initial and final values of the completed range are returned to `cabald` before a new task is requested.

cabald. `cabald` maintains a configuration file which contains a list of special q values which have been allocated to clients; one or more `pool` lines to specify unallocated special q values; a list of zero or more `fragments`; a single value, `blocksize`, which specifies the maximum special q range to be allocated to each client; and the network port on which it communicates to its clients. Initially, there are no `fragments` and a single `pool` containing a large special q range. A final set of lines contains information about which special q ranges have been allocated to clients by earlier invocations of `cabald`.

When `cabald` starts it reads its configuration file and creates a data structure which contains one or more ranges (lower and upper limits) of special q values which have not yet been allocated. Under normal circumstances, this would be a single range given by a `pool` line. Very occasionally, a second such line would be added to the configuration file when the existing pool was close to exhaustion. A more frequent occurrence would be after one or more clients had crashed. In this situation, the `cabald` process would be stopped, the unsieved special q ranges extracted from partial output files and added to the server's configuration file as `fragment` lines. Any corresponding allocation lines in the configuration file would be deleted. Upon restart, `cabald` also places the fragment data into the unallocated tasks data structure. `cabald` then opens a log file for appending status messages and enters an endless loop waiting for `cabalc` client requests.

On receipt of a client communication, the returned special q values are used to update the unallocated tasks structure. A new range of special q values, of size at most `blocksize`, is then sent to the client. Allocation is made from the `pool`(s) only when all the `fragments` have been exhausted. The log file is then updated with an entry which records the IP address of the client, the special q values, if any, returned by the client and the special q range just allocated. Finally, the `cabald` configuration file is re-written so that the current state of the `pool` and/or `fragments` is available for subsequent runs of `cabald`.

The script `cabald` was very stable. It never crashed unexpectedly and was stopped only for scheduled system shutdowns or for maintenance of its configuration file when fragments or a new pool were added.

monitoring. As noted above, `cabald` and `cabalc` provide neither detection of errors nor uploading of output data. The latter was performed by an uploader script which compressed all but the most recently modified `lasieve` output file (on the assumption that the latter was still being written by an active `lasieve`); uploaded the result to a fixed directory of a sftp server at EPFL; and then moved the compressed files to another directory where they could be recovered if necessary and yet not interfere with subsequent activity.

At sporadic intervals, usually once a day or so, the machines supposed to be sieving would be examined to see whether they were in fact doing so. A trivial script was written to contact all machines in the set of clients and to determine whether `lasieve` was running the correct number of times (a multi-cored system usually ran several copies). If a client failed to respond or if they were not sieving the situation would be investigated by hand. First, the uploader script would be run. Any remaining output files were examined to determine the special q at which the siever failed. Finally, `cabalc` was restarted.

Despite not being fully automated, the scripts `cabalc` and `cabald` between them allowed one person (Leyland) to manage several dozen siever instances with little effort. That the same scripts also worked satisfactorily to manage hundreds of sieving jobs at EPFL is probably due to the fact that there we restricted ourselves to stable resources fully dedicated to the sieving.

4.7 NTT

Nippon Telegraph and Telephone Corporation provided the following computational resources that were fully dedicated to the sieving:

- 113 Pentium D 3.0GHz (amd64) + 2GB RAM[3];
- 32 Pentium 4 (Northwood) 3.2GHz (i386) + 2GB RAM[4];
- 2 Pentium 4 (Prescott) 3.6GHz (amd64) + 2GB RAM;
- 1 Pentium 4 (Northwood) 2.8GHz (i386) + 2GB RAM;
- 2 Athlon 64 2.2GHz (amd64) + 3GB RAM;
- 2 Opteron 2.0GHz (amd64) + 4GB RAM;
- 8TB of storage via NFS.

The nodes are connected with gigabit Ethernet and each node is equipped with a local disk. To manage sieving assignments we used two perl scripts that were also used during the sieving for M1039 [1]: `ds2c` on the client side and `ds2` on the server side.

`ds2c`. For each client on which it is running, the script `ds2c` requests a special q range from the server, and runs `lasieve` while recording its standard input, error and return values. After `lasieve` finishes its assigned range, `ds2c` sends all resulting data to the server and requests a new range. When `ds2c` cannot connect to the server, `lasieve` is invoked with a range of special q values that is randomly chosen from a range previously communicated by the server.

`ds2`. The server script `ds2` has an interface that allows a human operator to provide a new range of special q values, typically of length 1 000 000. Upon request from a client, `ds2` assigns to the client a subrange, typically of length 1000 as a range of that size can be processed in a few hours. If a client does not report back within, say, 8 hours, its range is reassigned. At any time the operator may change priority of range assignments. Data corresponding to a range that is reported back are stored. The server may also receive data for ranges other than those it assigned; occasionally, correctness of such spurious data is verified manually. Logging mechanisms are in place to allow recovery from mishaps (or scheduled maintenance).

Although these two scripts can deal with many exceptions, they are unable to detect a full disk. Every working day manually invoked scripts and commands are therefore run to confirm client node status, to merge any duplicate assignments, to roughly confirm the consistency of all data stored by `ds2`, to compress the data, and to send them to EPFL.

[3] These nodes got more RAM for the matrix step (cf. Table 3), resulting in 5GB RAM for most nodes (13 nodes got 8GB RAM). Further details can be found in [2].

[4] One of these nodes broke down during the sieving. It was not repaired.

4.8 University of Bonn

At the University of Bonn sieving took place at only one location, the Himalaya cluster at the Institute for Numerical Simulation. On this cluster jobs have to be submitted via a queueing system. This was done using a simple C-program. It checked periodically how many sieving jobs are in the queue and, if this number is below a certain threshold, it submitted new jobs. All problems and inconsistencies, caused by jobs that were never taken into execution, jobs that crashed, etc., were resolved manually.

5 Conclusion

We described the heterogeneous hardware resources and diverse management tools used during a period of about two years at many different locations to solve a cryptanalytic challenge. The computational effort required, though large given the resources available, was considerable though not exceptionally large: it would require a couple of weeks using the full "Ranger" supercomputer at the University of Texas at Austin.

Our result is a good indication for the size cryptanalytic effort that can successfully be undertaken in a more or less acceptable amount of time by a rather loosely coupled, widely scattered and mostly academic team of volunteers. Pulling off a substantially larger effort in comparable or less time would require tighter management or more funding than customary in academic cryptanalytic circles. A greater appreciation of the HPC community for cryptanalytic activities could change this picture overnight.

Acknowledgements. This work was supported by the Swiss National Science Foundation under grant numbers 200021-119776 and 206021-128727 and by NWO, the Netherlands Organization for Scientific Research, as project 617.023.613. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed 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`). Condor middleware was used on EPFL's Greedy pool. We acknowledge the help of Cyril Bouvier during filtering and merging experiments.

References

1. Aoki, K., Franke, J., Kleinjung, T., Lenstra, A.K., Osvik, D.A.: A kilobit special number field sieve factorization. In: Asiarcypt, *LNCS*, vol. 4833, pp. 1–12. Springer (2007)
2. Aoki, K., Shimoyama, T., Ueda, H.: Experiments on the linear algebra step in the number field sieve. In: IWSEC, *LNCS*, vol. 4752, pp. 58–73. Springer (2007)
3. Cabal factorization of $2^{773} + 1$. http://www.mail-archive.com/mersenne@base.com/msg05260.html
4. Capit, N., Costa, G.D., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., Richard, O.: A batch scheduler with high level components. In: CCGRID, pp. 776–783. IEEE Computer Society (2005)
5. Condor. http://cs.wisc.edu/condor
6. Coppersmith, D.: Solving linear equations over GF(2): block Lanczos algorithm. Linear Algebra and its Applications **192**, 33–60 (1993)
7. Coppersmith, D.: Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. Math. Comput. **62**, 333–350 (1994)
8. Egee. http://www.eu-egee.org
9. glite. http://www.glite.org
10. Gordon, D.M., McCurley, K.S.: Massively parallel computation of discrete logarithms. In: Crypto, *LNCS*, vol. 740, pp. 312–323. Springer (1992)
11. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H.J.J., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit RSA modulus. In: Crypto, *LNCS*, vol. 6223, pp. 333–350. Springer (2010)
12. Lenstra, A.K., Lenstra, Jr., H.W.: The Development of the Number Field Sieve, *LNM*, vol. 1554. Springer-Verslag (1993)
13. Lenstra, A.K., Manasse, M.S.: Factoring by electronic mail. In: Eurocrypt, *LNCS*, vol. 434, pp. 355–371. Springer (1989)
14. Nfs@home. http://escatter11.fullerton.edu/nfs
15. The rsa factoring challenge faq. http://www.rsa.com/rsalabs/node.asp?id=2094
16. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE Computer Society (1994)
17. Stockinger, H., Pagni, M., Cerutti, L., Falquet, L.: Grid approach to embarrassingly parallel CPU-intensive bioinformatics problems. In: E-SCIENCE '06. IEEE Computer Society (2006)

A Clusters used and block Wiedemann timings

Table 3: Overview of the EPFL, Aladdin-Grid'5000, and NTT clusters used, including first and third stage block Wiedemann timings.
∗: using an older, slower binary.
†: figure per multiplication per chunk when two chunks are processed in parallel, in which case a part of the communication time is hidden in the local computation time (the communication number shows the pure communication percentage), for all other figures just a single chunk is processed.

									matrix				
Cluster location	Cluster name	Sieving (S) Matrix (M)	number of nodes	CPU type	clock speed (GHz)	cores per node	GB RAM per node	inter-connect	nodes per job	cores per job	seconds/multiplication stage 1	seconds/multiplication stage 3	communi-cation
Bordeaux	bordemer	S	48	2×Opteron 248	2.2	2	2	mx2g					
	bordereau	S	93	2×Opteron 2218	2.6	4	4	eth1g					
Grenoble	genepi	S&M	34	2×Xeon E5420	2.5	8	8	ib20g	24	144	3.7	not used	30%
Lausanne	Callisto	S&M	128	2×Xeon 5160	3	8	32	ib20g	8	64	13.8∗	not used	29%
	Greedy	S	1034	various									
	Lacal140	S	35	2×Xeon 5150	2.66	4	8	eth1g					
	Lacal304	S	38	2×Xeon E5430	2.66	8	16	eth1g					
	Lacal762	M	56	2×Opteron 2427	2.2	12	16	ib20g	12	144	4.3-4.5	4.8	40%
Lille	chicon	S	26	2×Opteron 285	2.6	4	4	mx10g					
	chinqchint	S&M	46	2×Xeon E5440	2.83	8	8	mx10g	36	144	3.1	3.3	31%
									32	256	3.8	not used	38%
									24	144	4.4	not used	33%
	chiti	S	20	2×Opteron 252	2.6	2	4	mx10g					
	chuque	S	53	2×Opteron 248	2.2	2	4	eth1g					
Lyon	capricorne	S	56	2×Opteron 246	2.0	2	2	mx2g					
	sagittaire	S	79	2×Opteron 250	2.4	2	2	eth1g					
Nancy	griffon	M	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	not used	5.0	31%
									16	64	not used	6.5	19%
	grillon	S	47	2×Opteron 246	2.0	2	2	eth1g					
Nice	azur	S	49	2×Opteron 246	2.0	2	2	mx2g					
	helios	S	56	2×Opteron 275	2.2	4	4	mx2g					
	sol	S	50	2×Opteron 2218	2.6	4	4	eth1g					
Orsay	gdx	S	180	2×Opteron 246	2.0	2	2	mx10g					
	gdx	S&M	132	2×Opteron 250	2.4	2	2	mx10g	98	196	2.8	3.9	32%
	netgdx	S	30	2×Opteron 246	2.0	2	2	eth1g					
Rennes	paradent	M	64	2×Xeon L5420	2.5	8	32	eth1g	49	196	6.2	not used	67%
									24	144	8.4	not used	67%
									18	144	10.0	not used	68%
									8	64	not used	18.0	56%
	paramount	S&M	33	2×Xeon 5148	2.33	4	8	mx10g	64	256	2.5	2.7	37%
	paraquad		64				4		49	196	2.9	3.5	33%
	paravent	S	99	2×Opteron 246	2.0	2	2	ib10g					
Tokyo		S&M	110	2×Pentium 4	3.0	2	5	eth1g	110	220	5.8†, 6.4	7.8	33%†, 44%
Toulouse	pastel	S	80	2×Opteron 2218	2.6	4	8	eth1g					