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

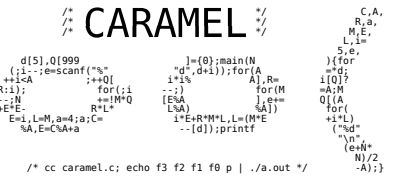
T. Kleinjung[1], L. Nussbaum[2], E. Thomé[3]

[1] EPFL, LACAL, Lausanne

[2] Univ. Nancy 2/INRIA, Algorille, Nancy

[3] INRIA, Caramel, Nancy

INRIA



1. **Context**
2. **Linear system and algorithms**
3. **Workplan**
4. **Job placement and I/O**
5. **Protecting against errors**

Context

Fairly commonplace task: solve a linear system. Here on $\mathrm{GF}(2)$.

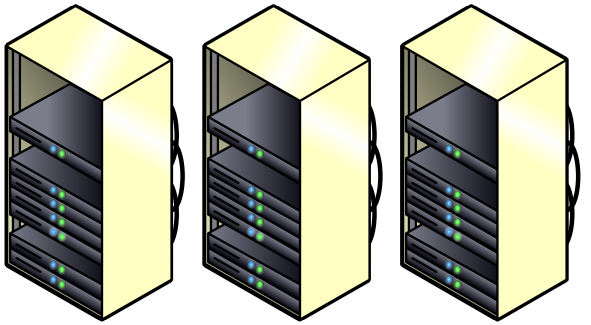
- Where does the problem come from ?
- How do we want to solve it ? Why is it interesting ?
- What are the difficulties ?

Where does the problem come from ?

Our starting point is the Integer factorization problem ($N = pq \rightarrow$ find p, q).

Factoring integers is important in cryptography: security of the RSA cryptosystem.

vs.



We care about assessing the feasibility limit.

Feasibility limit is important

Important distinction:

- For showing off in the media, we are “breaking cryptosystems”.
- Truth: we don’t care about people’s private data.
- We rather want to know what it takes to break a cryptosystem.

In particular here: given access to a shared resource like a grid platform, is the task made any easier ?

Also, the amount of required resources for factoring matters.

Big picture: factoring algorithm used

Ultimate goal: succeed in the factorization of a 768-bit RSA modulus.

Previous record: 663 bits, 2005.

The algorithm used is the Number Field Sieve.

Two computationally intensive steps:

- Sieving: collect many, many relations;
  - $\sim$ 20 years ago, this was already done in a massively parallel way.
- Linear algebra: solve an homogeneous linear system over $\mathrm{GF}(2)$.
  - $\sim$ 10 years ago: a supercomputer was required.
  - $\leq$ 2007: an “in-house” HPC cluster was required.
  - 2009: do it on Grid’5000. Prove that we can.

What is this all about

This work is not about whether solving the large linear system can be done on an HPC cluster. Of course it is.

We care about:

- whether a grid platform can be useful in this context ;
- what are the stumbling blocks for fitting this computation on a grid.

The computation will be split in small parts for which some HPC concerns are valid, but it's not the $\#1$ issue here.

1. Context
2. **Linear system and algorithms**
3. Workplan
4. Job placement and I/O
5. Protecting against errors

Linear system

M is an $N \times N$ matrix over $\mathrm{GF}(2)$. Search for a vector w s.t. $Mw = 0$.

The matrix M is large, (very) sparse, and defined over $\mathrm{GF}(2)$.

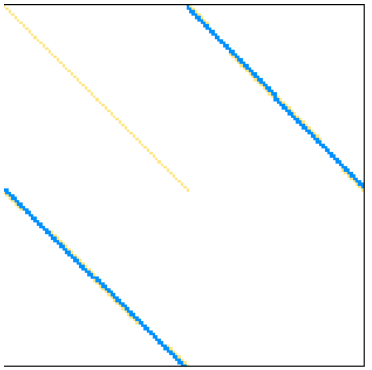
- $N = 192,796,550,$
- $\approx 27 \times 10^9$ non-zero coefficients: ≈ 140 per row.
- (if the matrix were dense, $4200+$ TB would be needed).

Because of sparsity, we want a black box algorithm.

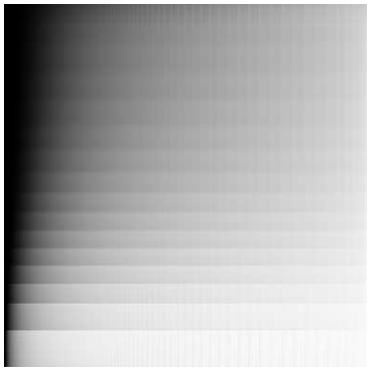
Specifics

We are doing exact linear algebra. Finite fields are not $\mathbb{R}$ or $\mathbb{C}$.

- Forget about convergence, fixed points;
- Forget about dominant eigenvalues;
- One wrong bit anywhere thoroughly voids the whole computation.
- Shape of matrix $\Rightarrow$ hard to get fast matrix times vector mult.



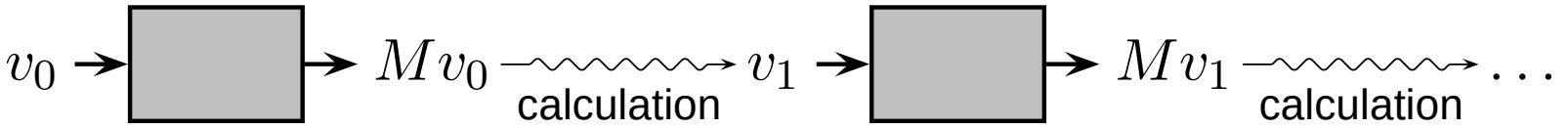
(PDE example)



(`rsa768` matrix)

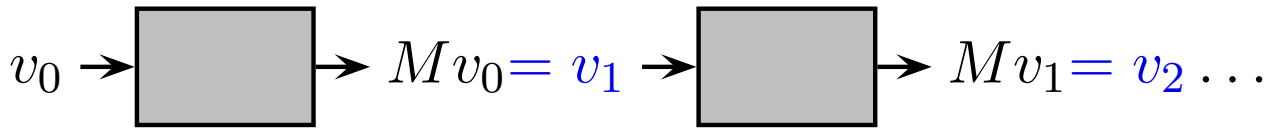
Iterative black-box algorithm: Lanczos

(block) Lanczos = Gram-Schmidt, adapted to finite fields.

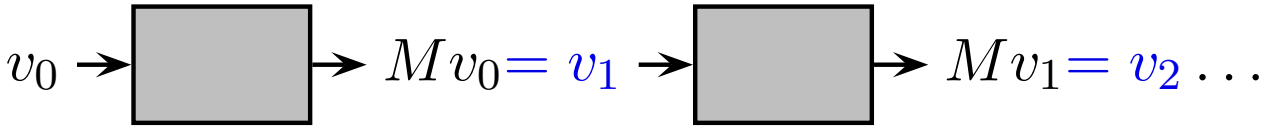


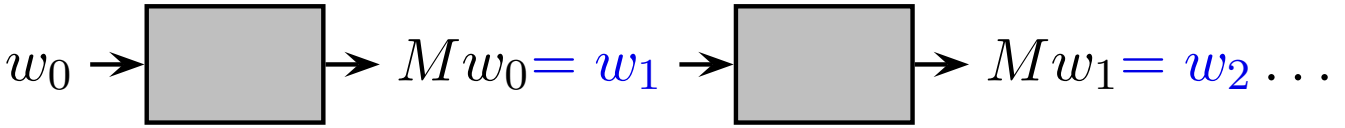
- Because M is over $\mathrm{GF}(2)$, vectors v are blocks of vectors: 64 (or more) bit vectors glued together.
- Intermediate calculations mix data, and use previous state.
- This requires the computation be synchronous.
- There's no way part of the computation can go faster.
- This mandates that we use one (large) cluster.
- Rather ok from an HPC perspective, but not tempting for grids.

Another pattern: block Wiedemann



- Output = concatenation of some bits extracted from each v_i.
- v_i is a vector block. Column j depends only on column j of previous vector blocks.
- Easy to consider blocks of many vectors (not infinitely many though). Use several clusters, and blocks of 64 vectors on each.





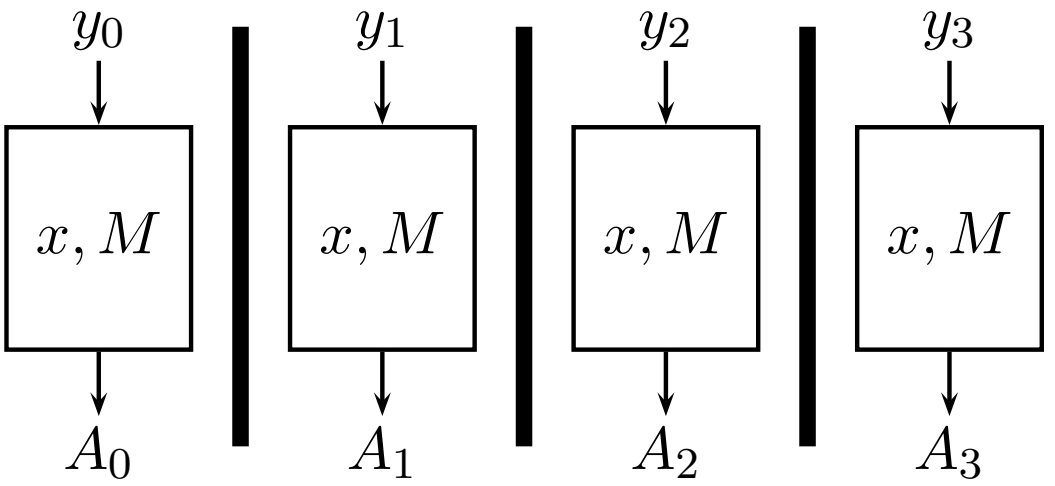
The number of iteration required drops proportionally.

$\Rightarrow$ Use this !

Block Wiedemann

Intermediate data $A_i = \sum_{k=0}^{L} \left(x^T M^k y_i \right) X^k$ for $i = 0, \ldots, n' - 1$.

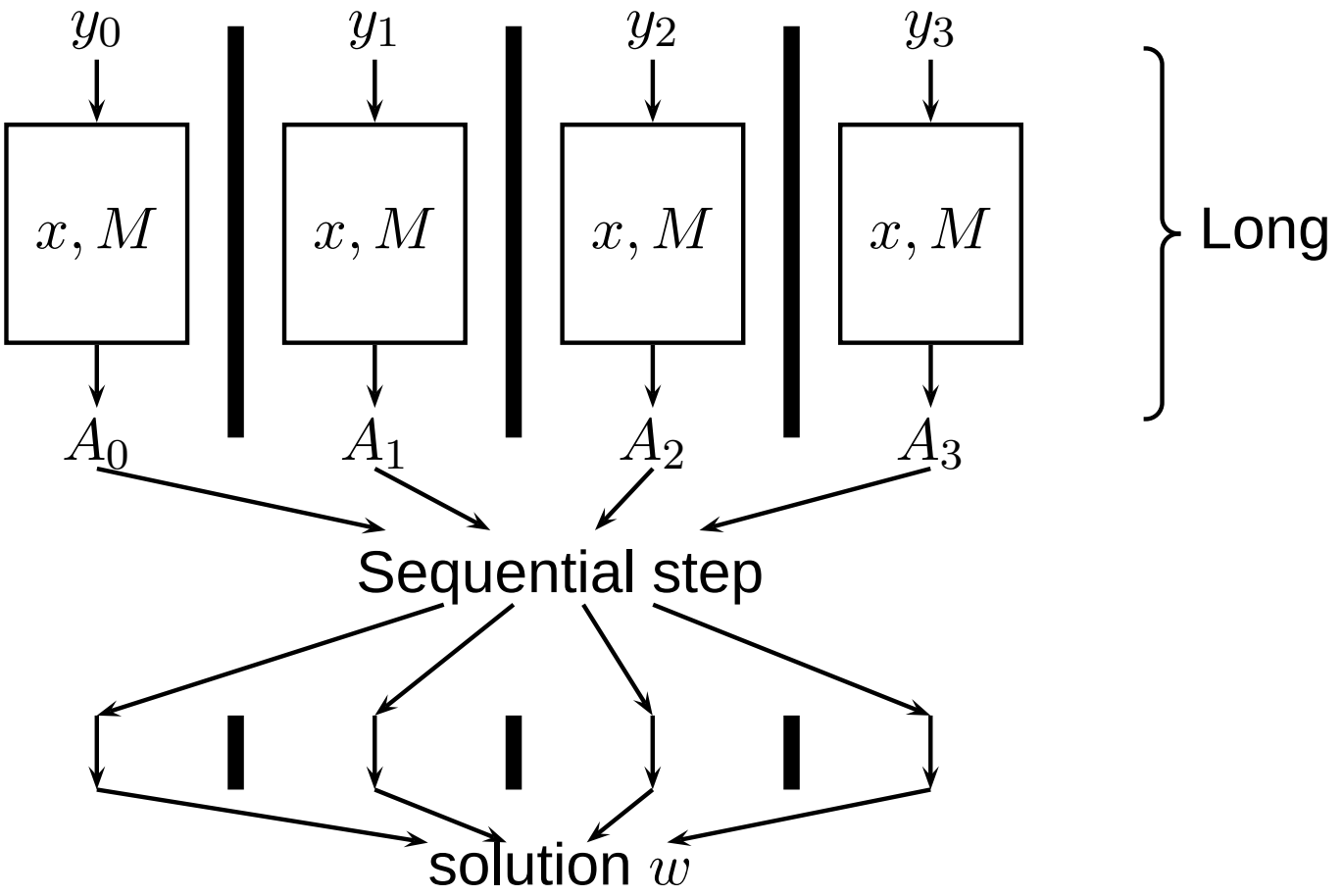
Do n' independent computations. Cluster i knows M, x, y_I.



Block Wiedemann

Intermediate data $A_i = \sum_{k=0}^{L} \left(x^T M^k y_i \right) X^k$ for $i = 0, \ldots, n' - 1$.

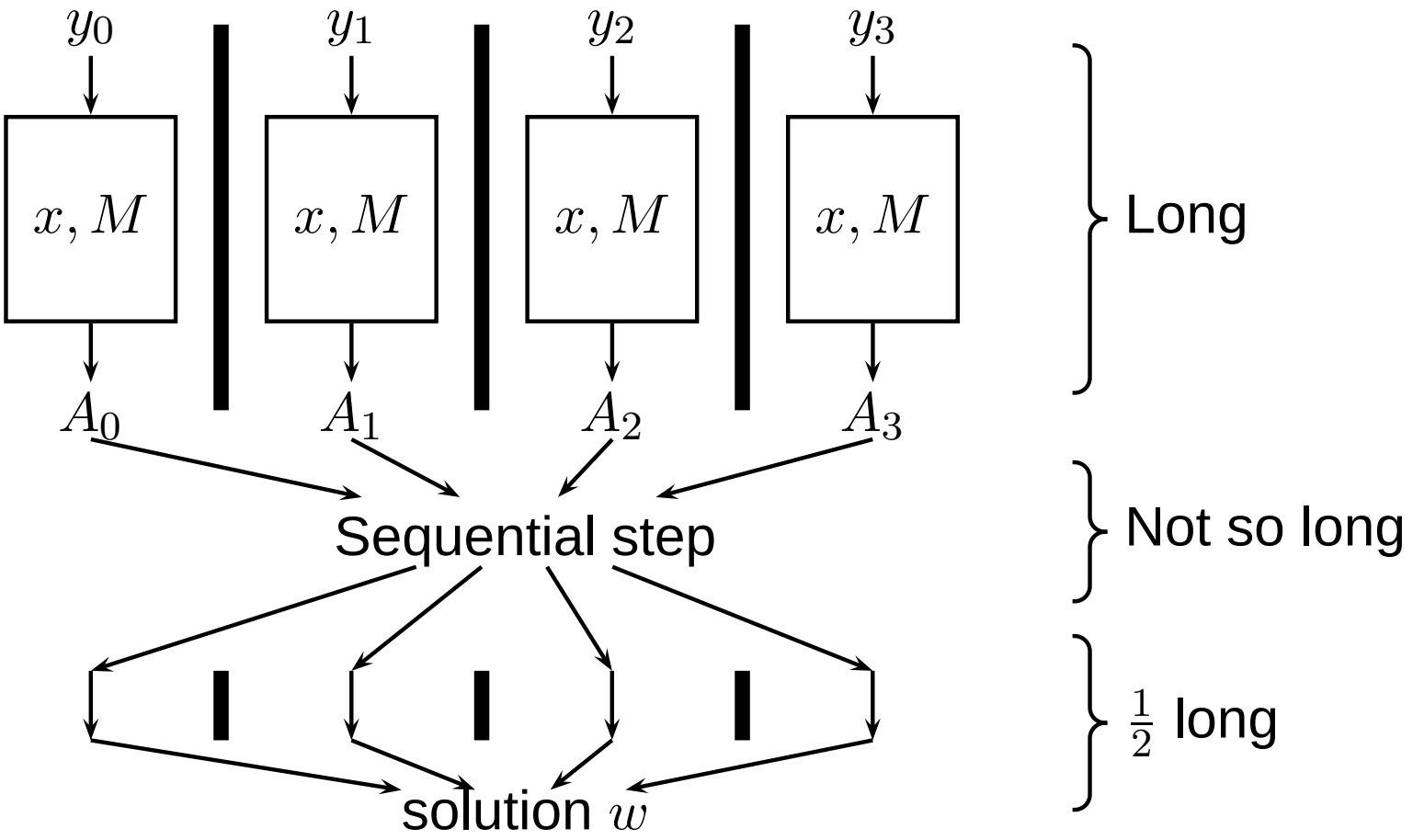
Do n' independent computations. Cluster i knows M, x, y_I.



Block Wiedemann

Intermediate data $A_i = \sum_{k=0}^{L} \left(x^T M^k y_i\right) X^k$ for $i = 0, \ldots, n' - 1$.

Do n' independent computations. Cluster i knows M, x, y_I.



1. Context
2. Linear system and algorithms
3. **Workplan**
4. Job placement and I/O
5. Protecting against errors

Setting up the workplan

The computation has been split into several groups.

- EPFL (Lausanne, Switzerland); uses lab cluster; $\sim 50\%$
- NTT (Tokyo, Japan); uses lab cluster; $\sim 10\%$
- INRIA/CARAMEL (Nancy, France) uses grid platform; $\sim 40\%$

Each group has a couple of sequences to work on (total 8).

For reference: one sequence = 4 weeks on a fast cluster.

Using the Grid'5000 platform

Grid'5000 (g5k) is an experimental platform.

- 9 different sites.
- $\sim$ 25 different homogeneous clusters, some with fast interconnect.
- g5k's aim is not to provide long time slots. Not a production platform.
- No persistent data possible on nodes.

Our jobs are meant not to disturb other users.

- Jobs during off-peak use times. Typical max is 10 hours at night, 60 hours during week-ends.
- Best-effort jobs, which are lowest priority and may be killed at any given time.

Splitting into small jobs

One job is described as e.g:

- Pick sequence 2 from where we left it off,
- and proceed for as many iterations as possible.

Since we have 4 sequences, we can have up to 4 simultaneous jobs.
$\Rightarrow$ need to choose among available fast clusters if >4 are available.

Two main aspects of jobs:

- Job size and placement;
- How we do computations;
  Typically 2 to 10 seconds for a matrix times vector product.
- I/O.

Splitting into small jobs

One job is described as e.g:

- Pick sequence 2 from where we left it off,
- and proceed for as many iterations as possible.

Since we have 4 sequences, we can have up to 4 simultaneous jobs.
$\Rightarrow$ need to choose among available fast clusters if >4 are available.

Two main aspects of jobs:

- Job size and placement;
- How we do computations; essentially an HPC concern.
  Typically 2 to 10 seconds for a matrix times vector product.
- I/O.

1. Context
2. Linear system and algorithms
3. Workplan
4. **Job placement and I/O**
5. Protecting against errors

Job placement

The available resources at a given time varies.

Placement done daily when off-peak starts.

For placing one job, we have to:

- choose a homogeneous cluster on which to run;
- choose a job size in number of nodes.
  sometimes a large cluster may fit two jobs.

Constraints:
- The total RAM for one job must be $\sim$ 200 GB.
  (75 GB matrix, + buffers, + vectors).
- #nodes must match one of the pre-configured sizes.
  (the cooked matrix data takes several hours to build).
- high-speed networks only (IB, MX).

Optimization criterion: cumulated number of iterations per second.
Gather performance data to drive automatic choice.

I/O

Now for the harder part... We are working with real data.

I/O tasklist for a job:

- Import the data set:
  total 75GB of cooked matrix data, one fraction (one file) per node.
- Import the starting vector: 1.5 GB.
- Export results (or, periodically, checkpoints) to central storage:
  - step 1 of BW: tiny (few MB), proportional to # iterations computed.
  - step 3 of BW: 12GB, no matter for how long the job has run.
- Before terminating, export last v_i to central storage: 1.5GB

A storage point is needed, with a few hundreds of GB.

Trivial when jobs are scheduled to run for long. Here they are not.

I/O transfer options

It is important to lower the job start-up time.

How do we import the matrix data ?

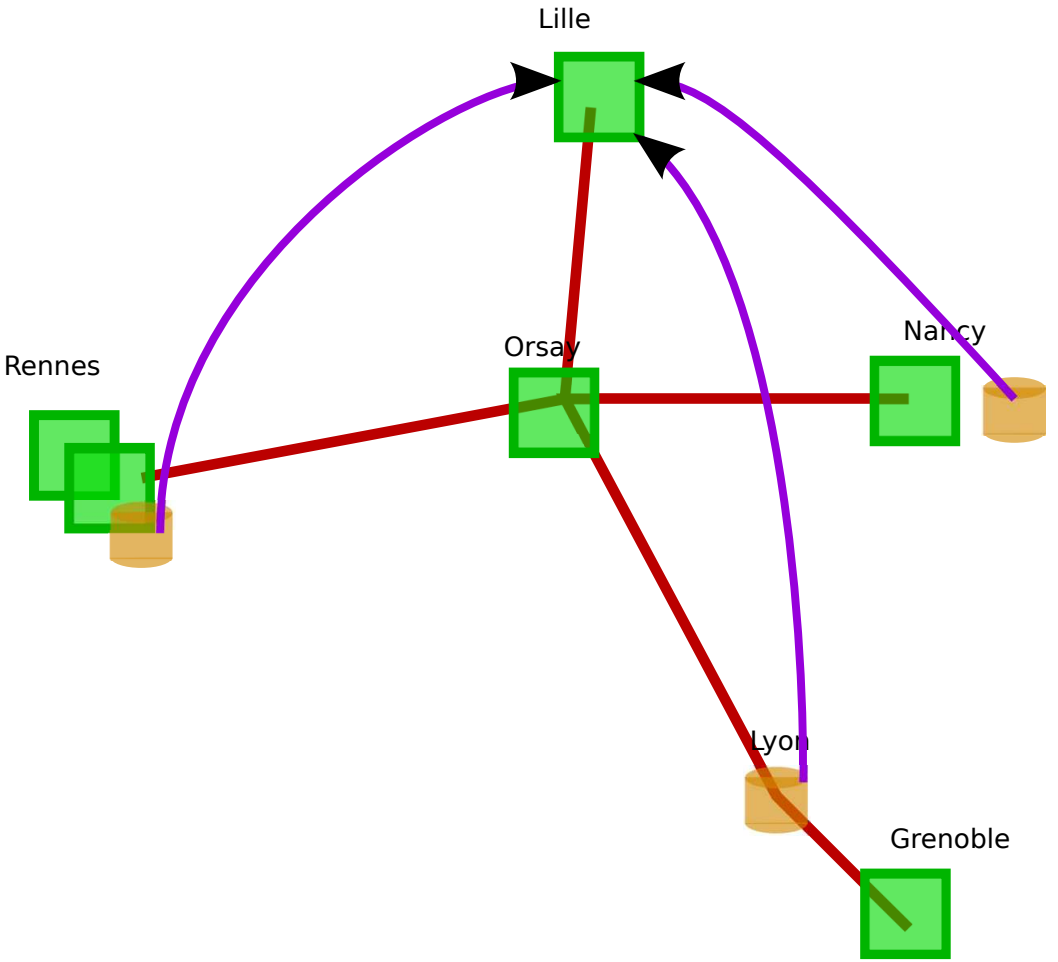
- Each site has NFS, but storing hundreds of GB on each is a lot. (at least 75 GB $\times$ # of pre-configured job sizes)
- Anyway, throughput is catastrophic. At least 30mn to import, sometimes more.

Better option:

Run persistent VMs on a few sites, and `rsync` daemons.

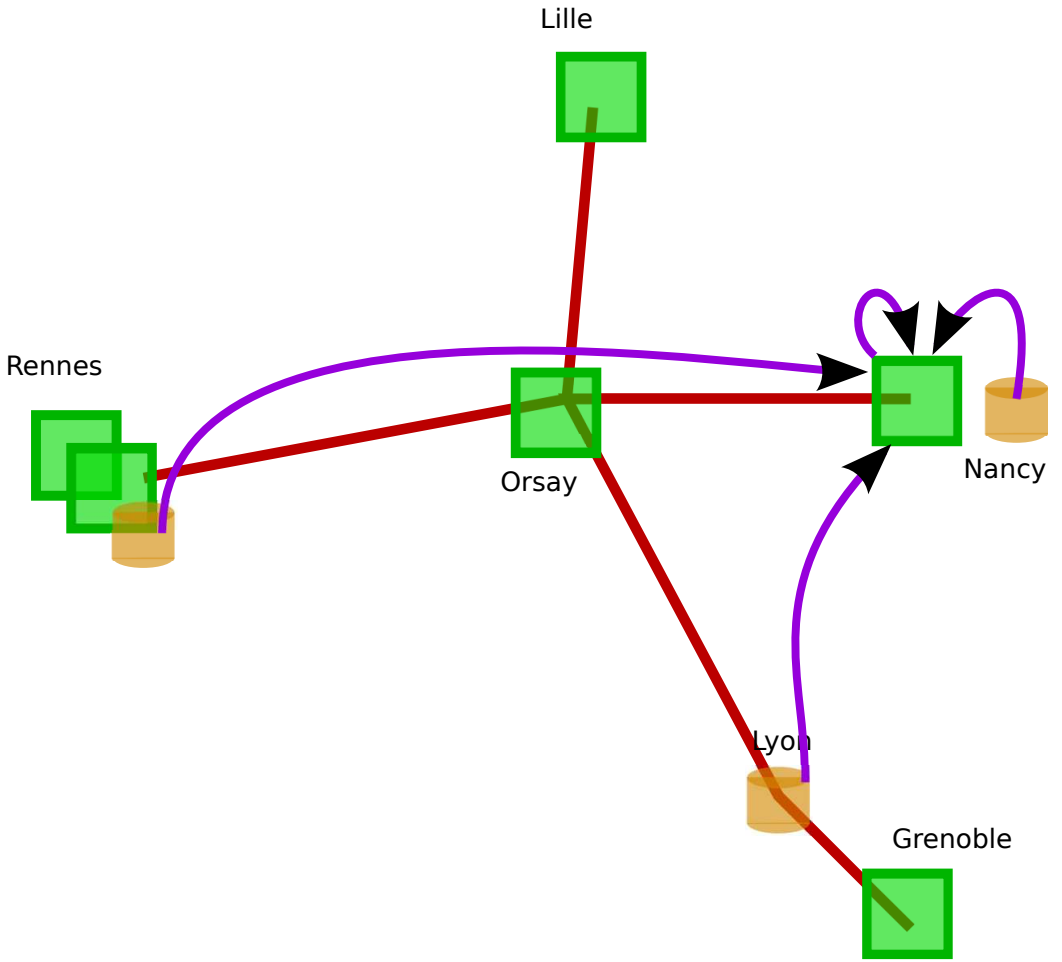
- Replicates data. Import done in parallel ;
- Provides fault tolerance. Some sites go down unexpectedly.

Importing (and exporting) data



- This redundancy provides a much lower start-up time: $\sim$ 5-10mn
- Makes it possible to use best-effort jobs (far from easy !).

Importing (and exporting) data



- This redundancy provides a much lower start-up time: $\sim$ 5-10mn
- Makes it possible to use best-effort jobs (far from easy !).

1. Context
2. Linear system and algorithms
3. Workplan
4. Job placement and I/O
5. **Protecting against errors**

Typical night or week-end job: checkpoints

Night jobs are not best-effort. Yet we save periodic checkpoints so as to:

- Guard against the possibility that part of the cluster crashes;
- Provide a way to check intermediate results using checkpoints;
- Avoid putting too much pressure on the final save.

Period chosen: one checkpoint every 4096 iterations: several hours.

When the end of the job is near, stop computing. Save current checkpoint to central storage.

- we do not know in advance exactly how long the save takes;
- risk: possibility that last iterations are lost;
- problem: all jobs tend to do the final save at the same time !

Best-effort jobs

A low start-up time makes it is realistic to run best-effort jobs.

- A best-effort job is not notified that it's going to be killed.
- Thus we need more frequent checkpoints. Chose one every 80 minutes, plus one every 4096 iterations.

 A lot of data is created when checkpoints are frequent.
Max aggregated throughput of $>$ 30MB/s.

- Offload data from storage points to avoid exceeding capacity ;
- Grabbing 30MB/s is barely acceptable..
- Doing some post-treatment on the VMs which are used for storage would have been better than offloading.

Linalg stats

(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 1: 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 ; (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.

Conclusion

Overall, for the `rsa768` matrix:

- Grid'5000 did 40% of the work.
- Calendar time about 3 months, only about 2 months fully working.
- First time a computation of this kind is done (partly) on a grid (for the linear system).
- Proved the viability of a grid setup for this purpose.

Moderately larger matrices using a similar approach ?

- w.r.t. computational resources, easy.
- w.r.t. I/O, some adjustments can be needed.

Thanks ! Questions ?