CSE291-14: The Number Field Sieve

https://cseweb.ucsd.edu/classes/wi22/cse291-14

Emmanuel Thomé

February 15, 2022

Part 6b

Linear algebra: introduction

Context

Algorithms

How to multiply a sparse matrix by a vector?

Thread and MPI-level parallelism

Plan

Context

Algorithms

How to multiply a sparse matrix by a vector?

Thread and MPI-level parallelism

The linear algebra step

The linear algebra step per se comes right after the filtering step and before the characters step. However from an abstract point of view, both of the latter are also linear algebra.

Key facts for a bird's eye view:

- The linear algebra step in the Number Field Sieve is the second hardest problem after the sieving step, in terms of computation time.
- In contrast to sieving, which is embarrassingly parallel, linear algebra is more difficult to parallelize. We are led to use hardware with fast interconnect capabilities, and that tends to be more expensive.
- There are some subtle differences between linear algebra for factoring and linear algebra for the discrete logarithm.

Multiple questions

We commonly refer to the linear algebra step. Yet one may ask several questions:

- is the system homogenous / inhomogenous?
- what are the unknowns? Rows or ideals?
- is the system singular?
- is there such a thing as a partial solution?
- does the world collapse if there is an error in my matrix?

Context recap: factoring

We must combine relations so that they consist of only squares.

This rewrites as a linear system.

- Sparse matrix M: relations $=$ rows, columns $=$ prime ideals.
- We seek several (say 64) solutions v to the system

$$vM = 0 \mod 2.$$

Context for DLP

We will learn about the NFS-DL variant for discrete logarithms, which brings in the following differences:

- Linear algebra over $\mathbb{Z}/\ell\mathbb{Z}$.
- Coefficients are most often ± 1.
- We look for a right kernel vector.

Things in common

The matrix in both cases is (almost) the same. It is large and very sparse.

- Theory: in the matrix before the filtering, we can show that asymptotically the number of non-zero coefficients per row is $O(\log\log(\text{number of rows}))$.
- Practice: filtering comes into play and asymptotics are only asymptotics.
  - 2019: 795-bit factoring: 282M rows/cols, density ≈ 200.
  - 2019: 795-bit DLP: 36M rows/cols, density ≈ 250.

Square matrices?

Both examples favored rectangular matrices.

In both cases the excess will be mostly consumed by filtering. However some mild excess is ok, we can in both cases pad with zero columns, so that we get a square matrix.

Exact versus numeric

In both cases, we are dealing with exact linear algebra.

Exact linear algebra

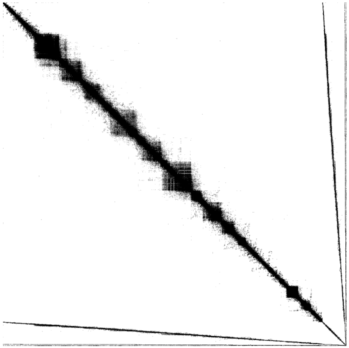
This implies two things.

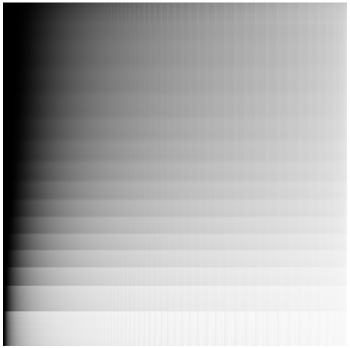
- We really want the solution, not an approximate one.
- There is no notion of convergence anyway.

90% to 100% of the nice “linear algebra for PDE solving” book on your shelf is useless.

$\mathbb{F}_2$ is exact, and positive characteristic

The matrices are not the same, either.


(some PDE example)


(a factoring matrix)

Nothing like “lower-order” either

It is not uncommon that the far-from-diagonal blocks in a PDE matrix are handled in a delayed way.

- Sometimes, we structurally have zero blocks there.
- Often, coefficients in this blocks have lower-order significance can be ignored, or approximated.

None of these shortcuts is valid in our case.

Plan

Context

Algorithms

How to multiply a sparse matrix by a vector?

Thread and MPI-level parallelism

Which direction

We have seen that both $vM = 0$ and $Mv = 0$ existed as problems to be solved.

For most of the internal dealings of the linear algebra solving, this does not matter much. We can transpose everything if we so desire.

Choice for exposition

In these slides, we present the case where we want to solve

$$Mv = 0$$

over some field $K = \mathbb{F}_\ell$, and we definitely have $\ell = 2$ on the radar.

Black box

Caveat

We like to use N for the number of rows and columns, but we must pay attention not to confuse that with the integer to be factored.

From now on, given that we have no use of the integer to be factored, N denotes the number of rows.

We have an $N \times N$ matrix M. We want to solve $Mw = 0$.

The matrix M is large, (very) sparse, and defined over $K = \mathbb{F}_\ell$.

Because of sparsity, we want a black box algorithm.

$$v \longrightarrow \boxed{M} \longrightarrow M \times v$$

Gauss?

Why can't we use Gaussian elimination?

Any algorithm that modifies the matrix inevitably causes fill-in.

- Smart recipes can be used to minimize fill-in somewhat, but it is still there.
- Example for the RSA768 matrix: a dense bit matrix of this size costs 4200TB to store.

This is the justification for black-box algorithms.

Are black boxes useful at all?

An example in numerical analysis.

- Take a random vector v.
- Iterate $v \leftarrow Mv/\|Mv\|$.
- If M has a dominant eigenvalue λ, $\frac{\|Mv\|}{\|v\|} \to |\lambda|$.

If we can do such things, no doubt we can do more.

How much of this black-box technology applies to finite fields?

Some terminology

In the numerical linear algebra world, the following distinction exists:

- direct methods may modify the matrix: Gauss, LU, . . .
- indirect (or iterative) methods: same as black-box methods.

In the numerical context, an indirect method that involves an $N \times N$ matrix can very well obtain a satisfactory result with only $o(N)$ applications of the black box.

In contrast, in the exact setting, we typically need $\Theta(N)$ applications of the black box!

Existing black box algorithms

The numerical context knows several indirect methods, often ranked according to the trade-off between iteration complexity, numerical stability, and convergence speed.

In practice, the simplest one is the Lanczos method.

- Very simple iteration ($\approx$ Gram-Schmidt orthogonalization).
- We work with the symmetric matrix $M^T M$.
- Application to finite fields is a bit of a gamble, since we have isotropic vectors: $x^T M^T M x = 0 \mod \ell$.
  - Not that much of a problem in large characteristic. Failure rate about $1/\ell$, no big deal.
  - Much more annoying if $\ell = 2$!!

Adaptation to finite fields

A very inefficient method to adapt to any finite field:

- Embed into $\mathbb{F}_{2^n}$, consider $M^T DM$ with D diagonal.
- This entails a more than n-fold overhead.

Much better alternatives:

- Finite field-native algorithms.
- Block methods.

Block algorithms

Extend the black box notion, pretty much in a SIMD way.

$$\mathbf{v} \longrightarrow \boxed{M} \longrightarrow M \times \mathbf{v}$$

$\mathbf{v}$ and $M\mathbf{v}$: blocks of vectors.

Example for binary matrices

A “block black box” makes perfect sense over $\mathbb{F}_2$.

- Let $\mathbf{v}$ be a sequence of N 64-bit integers.
  This can be viewed as 64 vectors of bits.
- Compute $M\mathbf{v}$ with bitwise XORs on 64-bit types.

$$\mathbf{v}'_i = \sum_{j, M_{i,j}\neq 0} \mathbf{v}_j.$$

Main observations and questions

- We do more work in almost the same time.
- Can this be put to some use? E.g., use the block black box fewer times that we would have used the (non-SIMD) black box?

Algorithms for finite fields

- 1986: Wiedemann's algorithm.
- 1991-1995: Block Lanczos algorithm (Montgomery), Block Wiedemann algorithm (Coppersmith).

The probability of success of these algorithms has been studied a lot by the computer algebra community (mid 1990s to early 2010s, mostly).

Plan

Context

Algorithms

How to multiply a sparse matrix by a vector?

Thread and MPI-level parallelism

First things first: transposition

All matrix-times-vector implementations can be transposed easily to give an implementation of $v \leftarrow Mv$ based on an example code that does $v \leftarrow vM$, and vice-versa.

- This is true at least in theory, as there's really a generic code transform which does that.
- In practice, it depends on how the code is written.
- Performance may not be 1:1 (read and write are not exactly the same!)

Simplistic approach

Assume that we have two large memory areas for the input and output vectors. (64-bit integers, or integers mod ℓ).

```
for($i$ = 0 ; $i$ < $N$ ; ++$i$) {
   $w_i$ = 0;
   for($j$ = 0 ; $j$ < $N$ ; ++$j$) {
      if ($M_{i,j}$ != 0)
         $w_i$ = $w_i + M_{i,j} v_j$;                  // XOR or addmul_si
   }
}
```

Multiple problems.

Some obvious improvements

Only store the locations (and values) of non-zero coefficients in M.

```
for($i$ = 0 ; $i$ < $N$ ; ++$i$) {
  $w_i$ = 0;
  for($j \in$ indices of non-zero coefficients in row $i$) {
    $w_i$ = $w_i + M_{i,j} v_j$;                    // XOR or addmul_si
  }
}
```

- Possible discussions about the data format (size, random access, etc). Note that in our context, adapting to any data format is certainly affordable!
  (basically, data is code. Decide on the code flow, then adjust the data format accordingly)
- Still very poor performance.

Accesses are very scattered

The j indices of non-zero coefficients in a row are very far apart.

- Most reads of v_j will be cache misses.
- This yields (inverse) throughputs of hundreds of CPU cycle per coefficient.

Improvement strategies

Straightfoward approach: split the matrix in blocks.

- Either fixed-size blocks (but what about the processing order?)
- Or: NW, NE, SE, SW and recurse, until some cut-off.

Problem: density is not uniform at all!
The vast majority of coefficients will still incur cache misses.

Nonuniform density

Because of the nonuniform density:

- Coefficients in the heaviest columns are processed very fast, even with the naive methods.
- As we reach more sparse areas, performance drops sharply.

This calls for some kind of adaptive mechanism.

- Process the “dense” vertical band with the heaviest columns with a fast simple-minded approach.
- Find a way to deal with the more sparse parts.

Two-pass

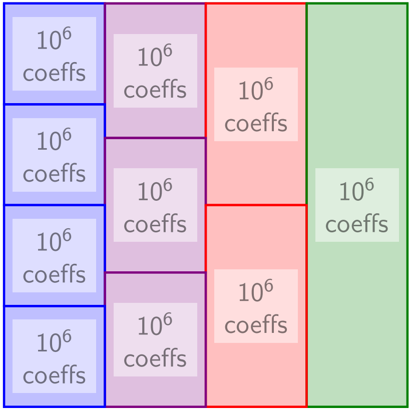
Preferred approach (totally reminiscent of bucket sieving!)

Assume that column density of M is $\searrow$. Let the control flow evolve as column density decreases.

- Have several temporary lists, on per column density cutoff.
- Load coefficients, arrange them in temporary memory in a way that is:
  - quickly accessible at the moment we do the memory store.
  - convenient to read back when we eventually store to the output vector.
- Access the temporary lists, store to output vector.

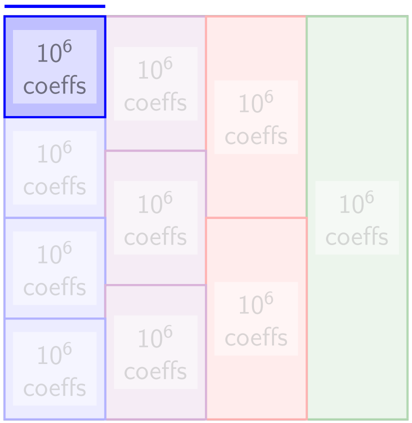
Example for $M \times v$

Note: The dimensions of the vertical bands may be uneven.
We want to limit to the reach of efficient random access.



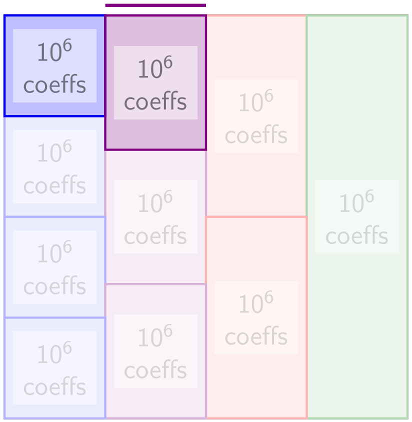
Example for $M \times v$

Load a batch of coefficients from the source vector.
Copy to list L_1, by increasing row index.



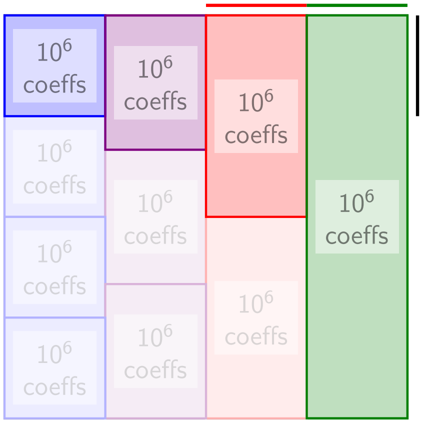
Example for $M \times v$

Load some new coefficients. Do the same, fill list L_2.



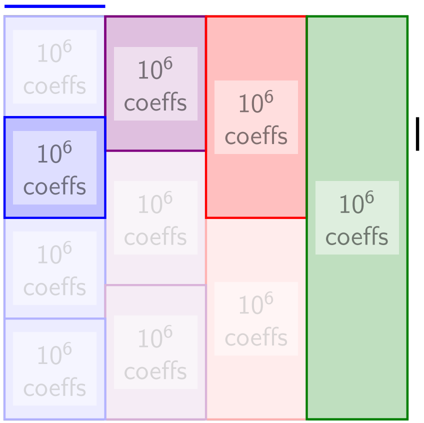
Example for $M \times v$

We have four lists which can be read in order, allowing us to compute a sequence of coefficients from the destination vector.



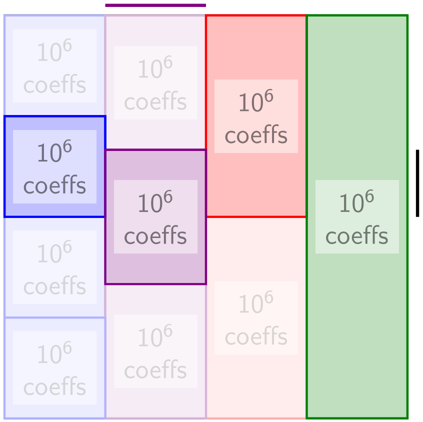
Example for $M \times v$

Periodically, we need to read again a batch of coefficients from the source vector, in order to refill the list L_1.



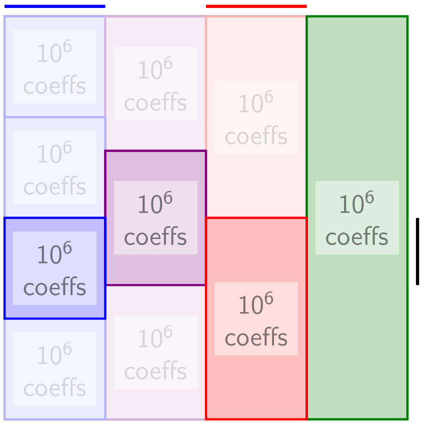
Example for $M \times v$

Periodically, we need to read again a batch of coefficients from the source vector, in order to refill the list L_1.



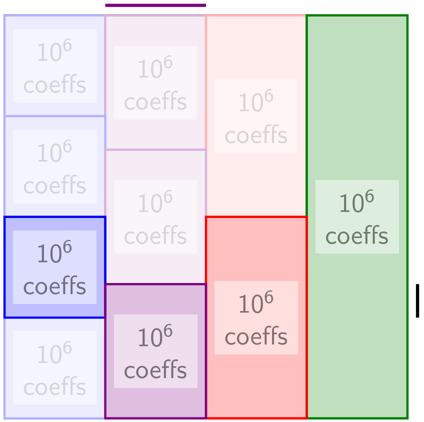
Example for $M \times v$

Periodically, we need to read again a batch of coefficients from the source vector, in order to refill the list L_1.



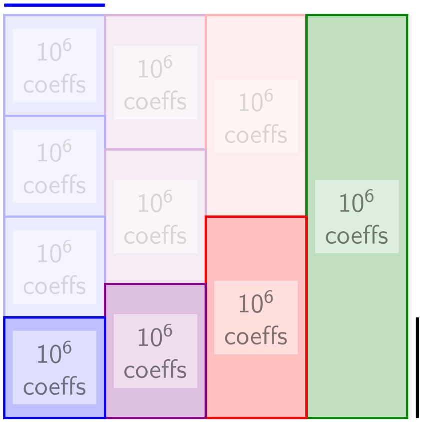
Example for $M \times v$

Periodically, we need to read again a batch of coefficients from the source vector, in order to refill the list L_1.



Example for $M \times v$

Periodically, we need to read again a batch of coefficients from the source vector, in order to refill the list L_1.



Analysis

- Each time the source vector is read, its data is used as much as we can.
  This amortizes the cost of reading data outside cache!
- We fill the list L_i with source vector coefficients exactly in the order they will be read later on.
- The destination vector is written to progressively.
  At the same time, data is read from many lists L_i in parallel.

Note: this applies also to $v \times M$:

- progressive writes $\rightarrow$ progressive reads.
- random-access reads $\rightarrow$ random-access $+=$.

Bucketized matrix-times-vector

Implementation details are somewhat hairy.

- Ideally, dynamic tuning would be welcome: we need to appreciate at runtime how many random-access writes / parallel reads can be sustained by multiple cores simultaneously. Hard-coded thresholds are unsatisfactory.
- Cutoffs don't necessarily match our picture that well.
- Maybe a two-layer approach could make sense.

Plan

Context

Algorithms

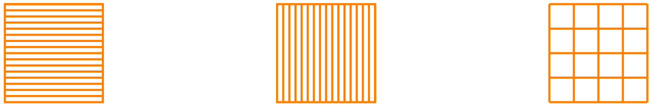
How to multiply a sparse matrix by a vector?

Thread and MPI-level parallelism

Thread- and MPI- level

To a certain extent, the concerns are similar.

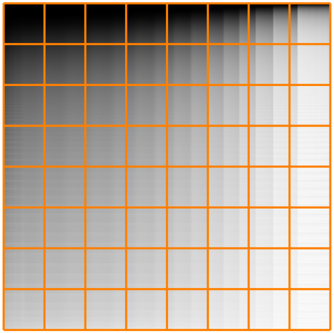
How could we split our matrix ?



The latter is preferred because it goes in the direction of favoring data locality.

Data exchange happens on each row or each column of the mesh.

Plans for splitting the matrix



Quite clearly a bad idea.

- Some nodes or threads would have a much harder time than others doing the product.
- All other nodes will have to wait for them.

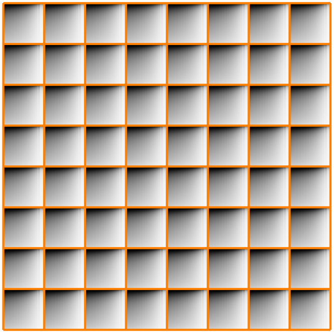
We may reorganize data

We are not bound to the internal representation of the matrix. We may rearrange coefficients (in a compatible way) so the splitting becomes better.

Balancing per-node weights

We may:
- sort columns, and distribute evenly $\Rightarrow T \in \mathfrak{S}_N$.
- sort rows, and distribute evenly $\Rightarrow S \in \mathfrak{S}_N$.

…because $Mv = 0$ or $SMTv = 0$ are reducible to one another !

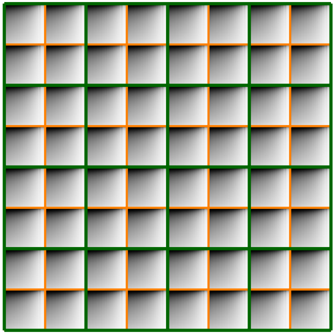


But the balancing story goes much beyond this.

Balancing per-node weights

We may:
- sort columns, and distribute evenly $\Rightarrow T \in \mathfrak{S}_N$.
- sort rows, and distribute evenly $\Rightarrow S \in \mathfrak{S}_N$.

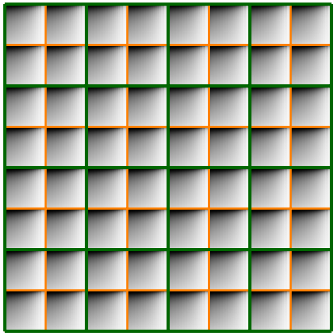
... because $Mv = 0$ or $SMTv = 0$ are reducible to one another !



- Core-level splits
- Node-level splits

But the balancing story goes much beyond this.

Balancing per-node weights



- Core-level splits
- Node-level splits

In Cado-NFS, the core-level and thread-level splits are specified with `thr=` and `mpi=` (here, 2x2 and 4x4).

A simple communication algorithm

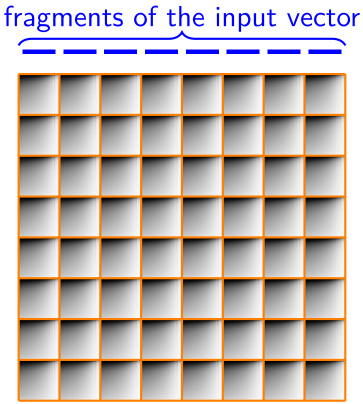
How do several nodes work together for computing $M \times v$?

Simplifying assumptions:
- square mesh of size $t \times t$.
- ignore core split vs node split.

Task list for each node (mesh row i, mesh column j):

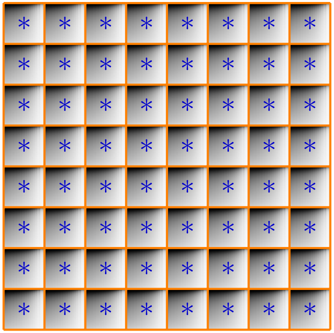
- Have a fraction $(1/t)$ of the input vector (j-th part).
- Compute the local product.
- Sum all the contributions across one row, for the fraction of the destination vector.
  Accumulate the sum on the i-th node of the row.
- On each column, the j-th node broadcasts the part of the computed vector for the next product.

Simple MPI product, graphically



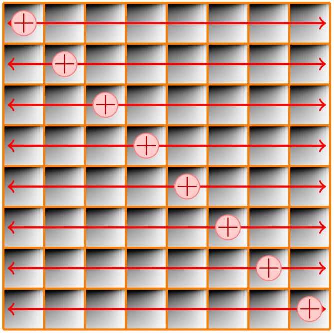
- Initial situation:
all fragments of the input vector are present in columns.

Simple MPI product, graphically



- Step 1: local multiplications.
No communication here.

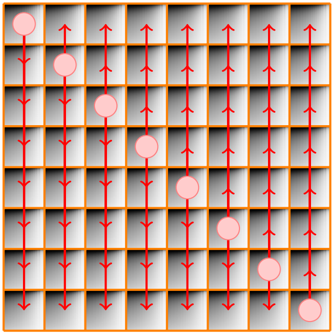
Simple MPI product, graphically



- Step 2: reduction across rows.

  `MPI_Reduce`

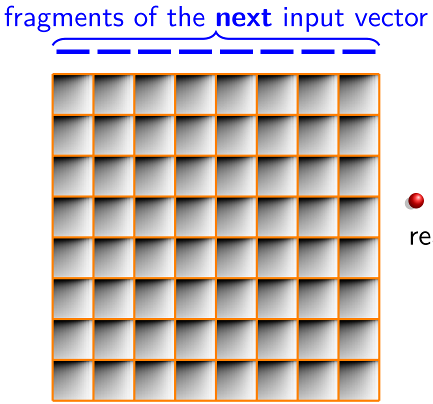
Simple MPI product, graphically



- Step 3: broadcast across columns.

  `MPI_Bcast`

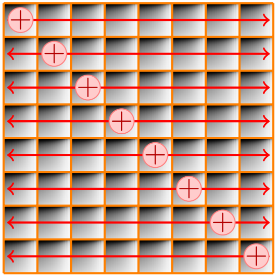
Simple MPI product, graphically



- Final situation:
ready for next iteration !

There are some downsides with this approach.

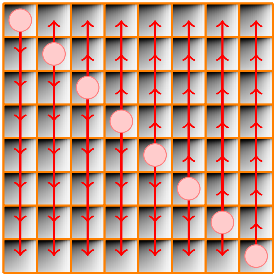
Limitations of the simple scheme



During the reduction step.

- I/O not balanced across rows.
- If linear complexity algorithm:
  - work ratio $1 : n$.
- If log-complexity algorithm:
  - work ratio $1 : \log_2 n$.
  - Logarithmic delay incurred.

Limitations of the simple scheme

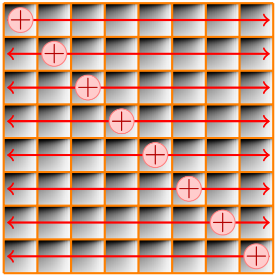


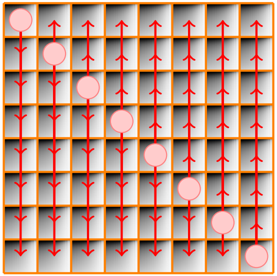
During the broadcast step.

- Exactly the same issue.
- Hypothetical hardware-level multicast? The answer is mostly no.

(AFAIK, no MPI impl. uses IB mcast !)

Limitations of the simple scheme



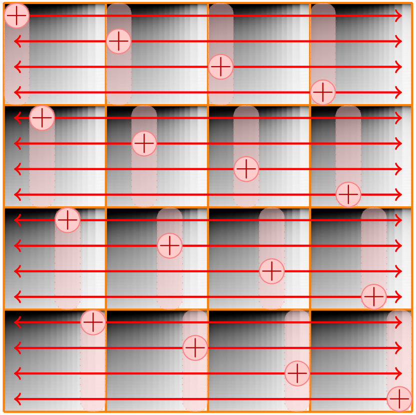


(Obvious) answer

It is better to parallelize work.

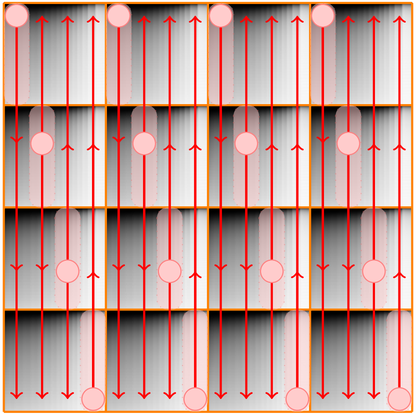
- Reduction step to collect parts of size $\frac{1}{t^2}$ on each t nodes.
- Bcast step to run t broadcasts of size $\frac{1}{t^2}$ in each col.

Parallelized MPI collectives



This even bears a name in MPI dialect: ● `MPI_Reduce_scatter`.

Parallelized MPI collectives



This even bears a name in MPI dialect:

- `MPI_Reduce_scatter`.
- `MPI_AllGather`.

Important for performance: `cpubinding`

Cado-NFS uses a feature called CPU binding.

Threads are “pinned” to specific cores, or groups of cores.

We prevent them from wandering too far from the memory they use the most.

A configuration file has to be passed to `bwc.pl`. It should be adjusted specifically for each machine.

- Cado-NFS looks for a section that matches the current machine, and then looks for an entry that matches the `thr` parameter.
- An example file (good starting point) with documentation is in `parameters/misc/cpubinding.conf`
- Trial and error is the only good way.

Building for MPI

Caveat: by default, Cado-NFS builds in non-MPI mode.

In order to build for MPI, one must pass an environment variable to `make`, e.g.:

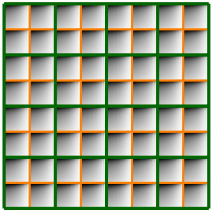
```
MPI=/opt/openmpi-x.y.z/ make -j4
```

The MPI-enabled binaries should preferrably not be mixed with the non-MPI ones. Wipe your build directory first.

Caveat: jitter

We have:

- sorted columns, and distributed them evenly.
- sorted rows, and distributed them evenly.



Things don't always go well

Use case: we split a matrix with average density ≈ 200 across (say) 64 nodes with 64 threads on each (4096 cores total)

- Even though rows and columns are evenly balanced globally, the weights of the 4096 sub-matrices can vary a lot!
- The CPU-bound workload per thread will likely differ, and this will cause wait times. Averting this is really hard.