CSE291-14: The Number Field Sieve

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

Emmanuel Thomé

February 22, 2022

Part 6d

The block Wiedemann algorithm

Plan

Block Wiedemann

BW is a direct translation of Wiedemann to using vector blocks.

Things to do:

- properly define the notion of linear generator.
- show that using vector blocks reduces the number of needed iterations.

The expected benefits versus Wiedemann are clear:

- Better use of arithmetic power of CPUs (block operations).
- Hopefully better success probability.

We may state it and use it either over $K = \mathbb{F}_2$, or $K = \mathbb{F}_\ell$.

This presentation: try to solve $M \times v = 0$.

BW: the blocking parameters

The Wiedemann algorithm had vectors x and y in $\mathbb{F}_\ell^N$.

Blocking parameters

Block Wiedemann chooses two parameters m and n.

- x becomes a block of m vectors: $x \in \mathbb{F}_\ell^{N\times m}$.
- y becomes a block of n vectors: $y \in \mathbb{F}_\ell^{N\times n}$.

BW workplan

Let m and n be the blocking parameters.

- Initial setup. Choose starting blocks of vectors x and y.
- Sequence computation. Want L first terms of the sequence:

$$a_i = x^T M^k y \text{ (a_i are $m \times n$ matrices !).}$$

  The length L will be given by the analysis.
- Compute some sort of linear generator.
- Build solution as:

$$v = \sum_{k=0}^{\deg f} M^k y f_k.$$

  coefficients f_k here are $n \times r$ matrices, so that can combine things together.

Plan

Coppersmith's block Wiedemann algorithm

- The sequence step: `krylov`
- The linear generator step: `lingen`
- The solution step in Block Wiedemann: `mksol`

BW: the sequence step

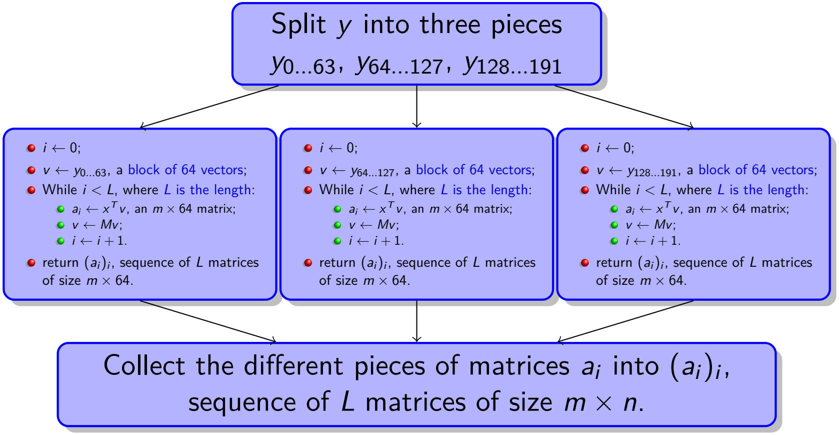
- $i \leftarrow 0$;
- $v \leftarrow y$, a block of n vectors;
- While $i < L$, where L is the length:
  - $a_i \leftarrow x^T v$, an $m \times n$ matrix;
  - $v \leftarrow Mv$;
  - $i \leftarrow i + 1$.
- return $(a_i)_i$, sequence of L matrices of size $m \times n$.

For example, a straightforward case with bits: our black box deals with (say) 64-bit machine words. BW with $n = 64$ is as here.

In Cado-NFS, this sequence step is done by the `krylov` program.

BW: sub-sequences

What if we have $n = 192$, and our black box still does only 64-bit?



Split y into three pieces

$y_{0\ldots63}$, $y_{64\ldots127}$, $y_{128\ldots191}$

```
• $i \leftarrow 0$;
• $v \leftarrow y_{0\ldots63}$, a block of 64 vectors;
• While $i < L$, where $L$ is the length:
    • $a_i \leftarrow x^T v$, an $m \times 64$ matrix;
    • $v \leftarrow Mv$;
    • $i \leftarrow i + 1$.
• return $(a_i)_i$, sequence of $L$ matrices
  of size $m \times 64$.
```

```
• $i \leftarrow 0$;
• $v \leftarrow y_{64\ldots127}$, a block of 64 vectors;
• While $i < L$, where $L$ is the length:
    • $a_i \leftarrow x^T v$, an $m \times 64$ matrix;
    • $v \leftarrow Mv$;
    • $i \leftarrow i + 1$.
• return $(a_i)_i$, sequence of $L$ matrices
  of size $m \times 64$.
```

```
• $i \leftarrow 0$;
• $v \leftarrow y_{128\ldots191}$, a block of 64 vectors;
• While $i < L$, where $L$ is the length:
    • $a_i \leftarrow x^T v$, an $m \times 64$ matrix;
    • $v \leftarrow Mv$;
    • $i \leftarrow i + 1$.
• return $(a_i)_i$, sequence of $L$ matrices
  of size $m \times 64$.
```

Collect the different pieces of matrices a_i into $(a_i)_i$,
sequence of L matrices of size $m \times n$.

Sub-sequences

Sub-sequences

In BW, the processing with the various sub-sequences is completely independent.
We define sub-sequences that match the optimal block width.

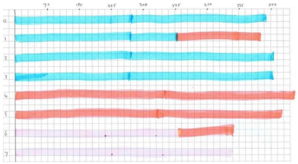
- over $\mathbb{F}_2$, we may for example define sub-sequences of width 64.
- over $\mathbb{F}_\ell$, we will probably define sub-sequences of width 1. This will STILL be block Wiedemann because we have matrices $(a_i)_i$ to handle, yet our black box will not really do blocks by itself.

A sub-sequence is identified by which range of columns of y it processes. Its output is the same range of columns for all the final matrices $(a_i)_i$.

Sub-sequences

Example: RSA-768 — we had $n = 512 = 8 \times 64$

- A contributor in Japan had a slow network and preferred to use two sub-sequences at the same time in interleaving fashion.
- Other 6 sub-sequences were processed independently in France and Switzerland.
- By periodically saving iterates $M^i y$ (say when $1000 \mid i$), we have a trivial chekpoint/restart feature.
- We actually exchanged sub-sequences to adapt to the various processing speeds.



More recent examples

In the 2016 kilobit hidden-SNFS-DLP computation, we had $n = 12$.

- Each black box deals with one product at a time.
- 12 independent sequences, 6 on each side of the ocean.
- Progress leveling every now and then (by hand).

More of the same with the records in 2019-2020, which all used multiple sequences.

BW: the length

The length of the sequence step is:

$$L = N/m + N/n + O(n/m + m/n).$$

The $O()$ term is less than 1000 for all practical ranges.
In practice we always have $\max(n/m, m/n) \ll N$.

Number of matrix-times-vector products

Whether or not we split into sub-sequences, the L steps of the sequence computation are performing:

$$n \times L = N \cdot (1 + n/m + o(1))$$

matrix-times vector products.
This is better than the $2N$ we had with the Wiedemann algorithm.

Plan

Coppersmith's block Wiedemann algorithm

- The sequence step: `krylov`
- The linear generator step: `lingen`
- The solution step in Block Wiedemann: `mksol`

The name of the game

Out of the sequence computation step (`krylov`), we have:

$$A(X) \in \mathbb{F}_\ell[X]^{m \times n},\ \deg A = \frac{N}{m} + \frac{N}{n}.$$

Wanted: matrix linear generator

We search for $F(X) \in \mathbb{F}_\ell[X]^{n \times n}$ and $G(X) \in \mathbb{F}_\ell[X]^{m \times n}$ such that:

$$A(X)F(X) = G(X) + O(X^{N/m+N/n}),$$
$$\deg F, G \leq \frac{N}{n}.$$

This involves arithmetic with matrices of polynomials.

Lingen algorithms

Several algorithms, rediscovered multiple times.
Costs (with $m = n$, to make things simpler):

- Coppersmith, 1994. $O(nN^2)$.
- Beckermann-Labahn, 1994. $O(nN^2)$, but also fast version $O(n^2(n + \log N)N \log N)$. This is the most general setting.
- T. 2001. $O(n(n + \log N)N \log N)$.
- Giorgi, Lebreton, 2014. Current state of the art, + online behaviour.

[T. 2001] is used for large NFS computations.

The `lingen` computation has significant memory requirements.
(Proportional to the input size when m and n are constant.)

Linear generator: basic idea

Fairly similar to Berlekamp-Massey.

- Analyze what can be done in quadratic complexity.
- Then build a recursive version.

Linear generator: quadratic base case

Recall $A(X) \in \mathbb{F}_\ell[X]^{m\times n}$. Ultimate goal:

$$A(X)F(X) = G(X) + O(X^L).$$

- Work with $m+n$ candidates (in columns) at a time.
- Extend F and G to
  - $F(X) \in \mathbb{F}_\ell[X]^{n\times(m+n)}$, and
  - $G(X) \in \mathbb{F}_\ell[X]^{m\times(m+n)}$.
- Initial error matrix $E(X) = (A(X)F(X) - G(X)) \operatorname{div} X^{\text{something}}$. We have $E(X) \in \mathbb{F}_\ell[X]^{m\times(m+n)}$.

We want a transformation matrix $\pi(X) \in \mathbb{F}_\ell[X]^{(m+n)\times(m+n)}$ such that

$$E(X)\pi(X) \equiv 0 \mod X^t$$

Linear generator: quadratic base case

What does it take to get $E(X)\pi(X) \equiv 0 \mod X^t$?

- we need to find solutions to $m \times (m+n) \times t$ linear constraints.
- with $\deg \pi < d$, we have $(m+n) \times (m+n) \times d$ degrees of freedom.
- Therefore, we should be able to do it with $d \approx \frac{m}{m+n} t$.

Advancing by t steps (in time $O(t^2)$)

We find $\pi(X)$ such that $E(X)\pi(X) \equiv 0 \mod X^t$ by setting approximately $\frac{m}{m+n} t$ coefficients in each matrix entry in $\pi(X)$. This is completely doable with a sort of Gaussian elimination.

More precise complexity: dependence on m and n is subtle.

Linear generator: recursive

- Compute an initial “error matrix” $E(X)$.
- Truncate to degree $\lceil L/2 \rceil$.
- Recurse, find $\pi(X)$ such that $E(X) \times \pi(X) \equiv 0 \mod X^{\lceil L/2 \rceil}$.
- Middle product (full) $E(X) \times \pi(X) \div X^{\lceil L/2 \rceil}$.
- Recurse a second time.
- Multiply $\pi(X)\pi'(X)$.

Again, quasi-linear algorithms and so on. More on this later.

One or many solutions ?

The linear generator step works with $m+n$ candidates internally, but eventually finds n solutions.

This is exactly similar to Berlekamp-Massey working with 2 candidates internally, but finding one generator.

How do we tell generators from non-generators eventually? By observing the fact that all matching columns in the error vector are canceled all of a sudden.

Interesting part of the linear generator

The linear generator step really outputs $F(X) \in \mathbb{F}_\ell[X]^{n\times n}$.
Input length: $L \approx \frac{N}{m} + \frac{N}{n}$. Output length: $\frac{m}{m+n}L \approx \frac{N}{n}$.

More on the linear generator matrix

The linear generator matrix, from a computer algebra perspective, has many interesting properties.

- Its determinant is close to (the reciprocal of) χ_M.
- Its Smith normal form is very close to the Smith normal form of $M - XI_N$ (invariant factors).
- $F(X)$ is very much a useful computer algebra thing!

However, we will not need these fancy properties.

Plan

Equation for one solution vector

The generator is a matrix in $\mathbb{F}_\ell[X]^{n\times n}$.

- Each column of this matrix will yield a solution of the linear generator problem.

  $$(\sum_i a_i X^i)F(X) = G(X). \quad \text{(infinite precision!)}$$

  One column is made of n polynomials.
- There is mathematical ground to say that the set of columns of the generator matrix form a basis of the set of solutions (according to a $\mathbb{Z}[X]$-module structure that is not hard to introduce).

Next step: move from a solution of the linear generator problem to a solution of the homogenous linear problem that we try to solve.

Many coefficients

Assume $\deg F \leq N/n$, $\deg G < N/n$. Write matrix $F(X)$ as

$$F_{i,j}(X) = \sum_{k=0}^{N/n} f_{i,j,k} X^k.$$

Coefficients of degree $N/n + d$ in $A(X)F(X)$ are zero, $\forall d \geq 0$.

$$\forall d \geq 0 \quad [X^{N/n+d}](A(X)F(X)) = 0.$$

More precisely, if columns j of F and G have degrees $\delta_{j,F}$ and $\delta_{j,G}$, then coefficients of degree $\delta_j = \max(\delta_{j,F}, 1 + \delta_{j,G})$ and above in the j-th column of $A(X)F(X)$ are zero.

Fact: we have columns j for which $\delta_j > \delta_{j,F}$.

Many coefficients

Since $a_i = x^T M^i y$, we have (still for any j, and δ_j as above):

$$\forall d \geq 0 \quad [X^{\delta_j+d},\ \text{column } j](A(X)F(X)) = 0,$$

$$\sum_{k=0}^{\delta_{j,F}} a_{d+\delta_j-\delta_{j,F}+k}[X^{\delta_{j,F}-k},\ \text{column } j]F(X) = 0,$$

$$x^T M^d \cdot M^{\delta_j-\delta_{j,F}} \underbrace{\sum_{k=0}^{\delta_{j,F}} M^k \sum_{i=0}^{n} \overset{\text{column } i \text{ of } y}{y_i} f_{i,j,\delta_{j,F}-k}}_{v_j} = 0.$$

- $M^{\delta_j-\delta_{j,F}} v_j$ is orthogonal to many vectors in $\mathbb{F}_\ell^N$.
  We can quantify the probability that it be zero: it is high.
- we may rearrange the expression so that v_j really looks like a combination of evaluations of polynomials at M (and y).

Equation for one solution vector

A combination of polynomial evaluations

$$v_j = \sum_{k=0}^{\delta_{j,F}} M^k \sum_{i=0}^{n} y_i f_{i,j,\delta_{j,F}-k}.$$

The equation of the solution is a bit like this:

$$v_j = \widehat{F_{0,j}}(M) y_0 + \widehat{F_{1,j}}(M) y_1 + \cdots + \widehat{F_{n-1,j}}(M) y_{n-1}.$$

where $(\widehat{F_{0,j}} \cdots \widehat{F_{n-1,j}}) = (X^{\delta_{j,F}} F_{i,j}(1/X))_i$.

This is:

- a bit like what we had with the Wiedemann algorithm
- except that we blend the different columns of the vector block y together.

mksol: procedures

$$v_j = \widehat{F_{0,j}}(M)y_0 + \widehat{F_{1,j}}(M)y_1 + \cdots + \widehat{F_{n-1,j}}(M)y_{n-1}.$$

This evaluation, called mksol, can be arranged in multiple ways.

- Compute all n solutions (w_0 to w_{n-1}) that are given by the n columns of F (not all will be linearly independent).
  Output would be a block of n vectors.
- Restrict to only r among n solutions. E.g. $r = 64$ or $r = 1$.
  Output would be a block of r vectors.
- Evaluate with a Horner scheme or not.

1st approach: `mksol`, n solutions, no Horner

```
$k \leftarrow 0$;
$v \leftarrow y$, a block of $n$ vectors;
$w \leftarrow 0$, a block of $n$ solutions;
While $k \leq \deg F$;
    $\forall i,\ w \leftarrow w + v_i \times$ (coefficients of degree $k$ in $\widehat{F_{i,0\cdots n-1}}(X)$);
    $v \leftarrow Mv$; (block width is $n$)
    $k \leftarrow k + 1$.
return $w$.
```

1st approach: `mksol`, n solutions, no Horner

```
$k \leftarrow 0$;
$v \leftarrow y$, a block of $n$ vectors;
$w \leftarrow 0$, a block of $n$ solutions;
While $k \leq \deg F$;
    $\forall i,\ w \leftarrow w + v_i \times$ (coefficients of degree $k$ in $\widehat{F_{i,0\cdots n-1}}(X)$);
    $v \leftarrow Mv$; (block width is $n$)
    $k \leftarrow k + 1$.
return $w$.
```

- Our black box deals with n vectors at a time (or, equivalently, we may split into sub-sequences).
- Note: we're reusing exactly the same vector iterates $M^i y$ as in the `krylov` step.
- This used to be the way I had always used BW until 2016.
- For $K = \mathbb{F}_2$ and $n = 64$, this is an entirely valid way to proceed. Not much else to do.

Benefits of using the same iterates

Since we use the same iterates $M^i y$ as in the `krylov` step, we can trade storage for more parallelism.

If we saved a few iterates $M^i y$ in the `krylov` (e.g. for $1000 \mid i$):

- As we already said, this provides checkpoint/restart for `krylov`.
- But this also allows us to compute the result of the `mksol` as the sum of many independent calculations.

k intermediary vectors saved $\leftrightarrow$ k-fold distribution for `mksol`.

2nd approach: fewer solutions, no Horner

```
$k \leftarrow 0$;
$v \leftarrow y$, a block of $n$ vectors;
$w \leftarrow 0$, a block of $r$ vectors. Goal: solutions $s$ to $s + r - 1$;
While $k \leq \deg F$;
    $\forall i,\ w \leftarrow w + v_i \times$ (coefficient of degree $k$ in $F_{i,\widehat{s\cdots s+r-1}}(X)$);
    $v \leftarrow Mv$; (block width is $n$)
    $k \leftarrow k + 1$.
return $w$.
```

- This saves a little bit on the vector multiplication part.
- We are still going through the same vector iterates.

`mksol` cost, no Horner

The degree of F is $\approx N/n$. Therefore the previous process does N/n application of the black box, of width n.

- In this setting, `mksol` costs N matrix-times-vector products.
- The total cost of `krylov`+`mksol` is now

$$(2 + n/m)N$$

  matrix-times-vector products.
- Better than non-block (if $m > n$), but still more expensive than (block) Lanczos.
- Increasing m and n only works to a certain extent, since the linear generator step becomes more expensive as $m + n$ grows.

3rd approach: `mksol`, n solutions, Horner

```
• $k \leftarrow \deg F$;
• $v \leftarrow y$, a block of $n$ vectors;
• $w \leftarrow 0$, a block of $n$ solutions;
• While $k \geq 0$, where $L$ is the length:
    • $w \leftarrow Mw$ (block width is $n$);
    • $\forall i,\ w \leftarrow w + y_i \times$ (coefficients of degree $k$ in $\widehat{F_{i,0\cdots n-1}}(X)$);
    • $k \leftarrow k - 1$.
• return $w$.
```

- We are no longer using the same iterates.
- However, we can still reuse $M^{1000}y$ in order to compute the contribution of the terms of degree 1000 to 1999 in the sum!

A piece of the Horner computation

$$\text{fragment of } v_j = \sum_{k=1000}^{1999} M^k \sum_{i=0}^{n} y_i f_{i,j,\delta_{j,F}-k},$$

This is exactly the same as a degree-999 evaluation of the same kind, with M^{1000} as a starting vector.

- This means that with Horner evaluation, we can still benefit from the checkpoints that we have saved in the Krylov space.
- However, our computation $w \leftarrow Mw$ is still operating on a block of n vectors.

4th approach: `mksol`, r solutions, Horner

- $k \leftarrow \deg F$;
- $v \leftarrow y$, a block of n vectors;
- $w \leftarrow 0$, a block of r vectors. Goal: solutions s to $s+r-1$;
- While $k > 0$, where L is the length:
  - $w \leftarrow Mw$ (block width is now r here);
  - $\forall i,\ w \leftarrow w + y_i \times$ (coefficient of degree k in $F_{i,\widehat{s\cdots s+r-1}}(X)$);
  - $k \leftarrow k-1$.
- return w.

We can do new things!

- $r = 1$ solution with only N/n matrix times vector products, with a block width of 1 (typical with large ℓ).
- or $r = 64$ solutions with
  - rN/n matrix times vector products,
  - or equivalently, N/n matrix times vector (block) products, with a block width of 64.

Improved cost

New cost: rN/n for `mksol` (for r solutions).

The total cost of `krylov`+`mksol` is now

$$(1 + n/m + r/n)N$$

matrix-times-vector products.

References: Kaltofen95, FGHT17.

New

In this setting, for N large enough and fixed r, we can choose parameters so that the cost of BW is

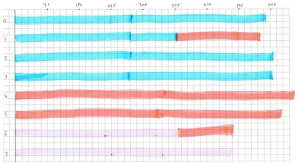
$$(1 + o(1))N$$

matrix-times-vector products.

Splitting the computation in pieces

In `krylov` we may periodically save the vectors $M^{k\times 1000}y$.

- This makes it possible to checkpoint and restart.
- Of course we cannot compute from iteration $k \times 1000$ until we have at least reached this iteration.



`mksol` checkpoints

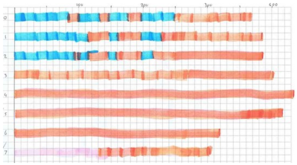
These same checkpoints can also be used:

- by `mksol`/no-horner, trivially;
- by `mksol`/horner also: we let $M^{k\times 1000}y$ play the role of y, and we compute a part of the final sum.

Of course the value $\boxed{\texttt{interval}=1000}$ can be adjusted:

- Smaller = more checkpoints, more disk, many independent tasks;
- Larger = fewer checkpoints, fewer (longer) tasks.

Note: all necessary checkpoints are already there when `mksol` starts! We can do everything in parallel if we want.



Example

In FGHT17, we had $N = 28.3 \times 10^6$ and $m = 24$, $n = 12$.

Total number of products: 44×10^6.

We could have made this lower but:

- we were not absolutely confident about whether the `lingen` step would go smoothly;
- this was our very first experiment with this strategy.

	sieving	linear algebra		
		sequence	generator	solution
cores	≈3000	2056	576	2056
CPU time (core)	240 years	123 years	13 years	9 years
calendar time	1 month	1 month		

Example

In BGGHTZ20, for DLP240, we had

$$N = 36 \times 10^6, \quad m = 48, \; n = 16.$$

Total number of products: 50×10^6.

	sieving	linear algebra		
		sequence	generator	solution
cores	$\geq$10000	3072	576	26880
CPU time (core)	2400 years	700 years	12 years	70 years
calendar time	6 months	3 months	62h	1 day

Example

In BGGHTZ20, for RSA240, we had

$$N = 282 \times 10^6, \quad m = 512 = 8 \times 64,\ n = 256 = 4 \times 64,\ r = 64.$$

Total number of products (block width 64): 7.7×10^6.

	sieving	linear algebra		
		sequence	generator	solution
cores	$\geq$10000	2048	512	2048
CPU time (core)	800 years	70 years	10 months	13 years
calendar time	2 months	37 days	13h	7 days

Note: linear algebra computation done in best-effort mode, calendar time is not really meaningful.

Guarding against errors

We can check the data on disk. It is useful because data on disk could be corrupted (disk errors, disk full, ...).

Simple idea:

- let C_0 be a random vector (or vector block);
- compute $C_{1000} = (M^T)^{1000} C_0$ (pre-compute);
- check that $C_{1000}^T (M^{k \times 1000} y) = C_0^T (M^{(k+1) \times 1000} y)$.
- we detect errors with good probability.

Caveat: C_0 must not have zero coefficients: it would limit our ability to detect errors.

Different steps

There are two ways to run the block Wiedemann algorithm.

BW has several steps, and Cado-NFS has several binaries.
Some steps are computational, some are mere bookkeeping.

Steps in BW

- Let m, n be… ....
- Let x, y be… ....
- Compute C_{1000}. ....
- Compute $A(X) = \sum_i {}^t x M^i y X^i$. .....
- Compute $F(X)$. ....
- Compute $\sum_i M^i y f_i$. piecewise,… then the sum

Steps in BWC

- command line
- `prep`
- `secure`
- **`krylov`**
- **`lingen`**
- **`mksol`**
- `gather`

Plan

Block algorithms

Both block algorithms we know of use a block black box.

That black box is able to deal with blocks of (say) n_1 vectors at the same time.

- When the base field is $\mathbb{F}_2$, we probably want to choose $n_1 = 64$, while for larger fields it is likely that $n_1 = 1$ is best.
- Per se, the black box rather offers a SIMD mode of operation (a.k.a. table soccer) rather than parallelism.
- Whenever we can do some n_1, it is trivial to emulate n_1 twice or three times larger (with a loop!)

The 1st level of “parallelism” is SIMD

SIMD level: optimal value for n_1

When matrices are sparse, most of the time in the matrix-times-vector operation comes from memory throughput rather than from CPU computation.

- Using SSE-2 (128-bit) types instead of 64-bit types might take a bit less than twice the time per iteration.
- But it is not even clear.
- Furthermore, doing too much SIMD can hamper parallelism at higher levels.

Thread-level / SMP

$$\begin{tikzcd}
\boxed{\text{One core has the matrix data and multiplies it by a block of } n_1 \text{ vectors}} \arrow[d] \\
\boxed{n_2 \text{ cores each have } 1/n_2\text{-th of the matrix data and collectively work to multiply it by a block of } n_1 \text{ vectors}}
\end{tikzcd}$$

The 2nd level of parallelism is threads (intra-node, SMP)

Implicitly, the thread level can make nice use of shared memory.

- NUMA is something we have to pay attention to,
- our communication pattern must be well thought.

MPI-level / cluster

One node (n_2 cores) has the matrix data and multiplies it by a block of n_1 vectors

$\downarrow$

n_3 nodes each have $1/n_3$-th of the matrix data and collectively work to multiply it by a block of n_1 vectors

The 3rd level of parallelism is MPI (inter-node)

The interconnect topology is important. Again, we must pay attention to our communication pattern.

Distribution

One cluster (n_3 nodes) has the matrix data and some init data, and is busy for time T

$$\downarrow$$

n_4 clusters each have the matrix data and some init data, and are busy for time T/n_4

The 4th level of parallelism is the distribution level
Only the block Wiedemann algorithm can do this

Practically no communication between clusters, at this level (only dispatch & reconcile).

Three caveats

We must pay attention to three important things:

- Scaling, esp. at the MPI- (n_3) and thread- (n_2) levels, because communication costs are pure overhead.
- Global block size ($n_1 n_4$), and how it should not go out of control.
- Choice of n_1.

Scaling

For fixed n_1:

- we expect levels $2, 3, 4$ to bring time T to $T/(n_2 n_3 n_4)$;
- in practice it might not be so, esp. if n_2 and n_3 are large.

Answers:

- careful implementation and thread placement. CPU binding is particularly important.
- well-organized communication patterns.

Block size

Given our presentation with multiple levels, the block size that we see from the global algorithm point of view is $n = n_1 n_4$.

- Block Wiedemann `lingen` has some cost related to the block size, of the order of $\widetilde{O}(nN)$.
  We must really pay attention to it.
- Block Lanczos, too, has some additional costs that are proportional to n ($n = n_1$ for BL, since $n_4 = 1$).

Choosing n_1 properly

When $K = \mathbb{F}_2$, a black box iteration with $n_1 = 1$ or $n_1 = 8$ take the same time. The time is well sub-linear until some block size, and then super-linear.

Two examples on my laptop:

matrix	`rsa100`, 135krows 100 iterations	`c163`, 10Mrows 4 iterations
$n_1 = 8$	2.25	16.80
$n_1 = 16$	2.75	19.79
$n_1 = 32$	3.93	23.65
$n_1 = 64$	5.00	27.44
$n_1 = 128$	5.85	35.25
$n_1 = 256$	17.86	68.40

This is no definite truth, but it indicates that 128-bit looks like a sweet spot.

Choosing n_1 properly

Whatever the sweet spot, a large n_1 certainly forces us to reduce n_4 if we would like their product to remain bounded.

$\Rightarrow$ too much SIMD may actually be a nuisance.

Plan

Coppersmith's block Wiedemann algorithm

Parallelization levels

Parallelization of the linear generator step

FFT in the linear generator step

The main operations of the linear generator step in BW are

> Multiplications of matrices of polynomials over finite fields.

We want to use asymptotically fast algorithms. here!

- First approach: $c_{i,j} = \sum_k a_{i,k} \times b_{k,j}$
- Better complexity: use the fact that we are using FFT-based algorithms.
  - Compute all forward transforms $\widehat{a_{i,k}}$.
  - Compute all forward transforms $\widehat{b_{k,j}}$. linear
  - Compute all convolutions $\widehat{c_{i,j}} = \sum_k \widehat{a_{i,k}} * \widehat{b_{k,j}}$
  - Compute all inverse transforms $\widehat{\widehat{c_{i,j}}} = c_{i,j}$.
- Caveat: memory goes totally out of control.

Memory cost of fast multiplication

How much memory do we need to multiply two integers of the same size?

Input size		Peak memory
bits	MB	MB
2^{23}	1	18
2^{24}	2	28
2^{25}	4	49
2^{26}	8	90
2^{27}	16	172
2^{28}	32	336
2^{29}	64	664
2^{30}	128	1320
	...	
2^{40}	128GB	1.3TB

One Fourier transform = about 5 times the input size!

Parallelization of the linear generator step

Two reasons to parallelize:

- Use more CPU power and get the result faster.
- Have more memory available.

This requires appropriate scheduling of the computation of the transforms.

Guiding principles:

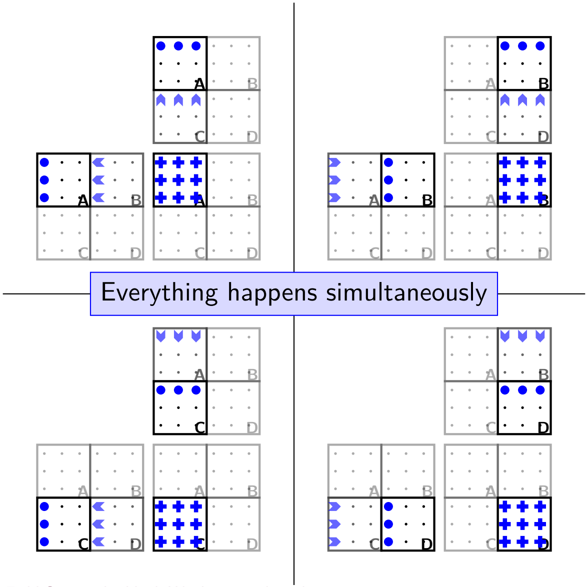
- limit the lifetime of transforms as much as we can.
- adapt the control flow when relevant.

Parallelization of the linear generator step

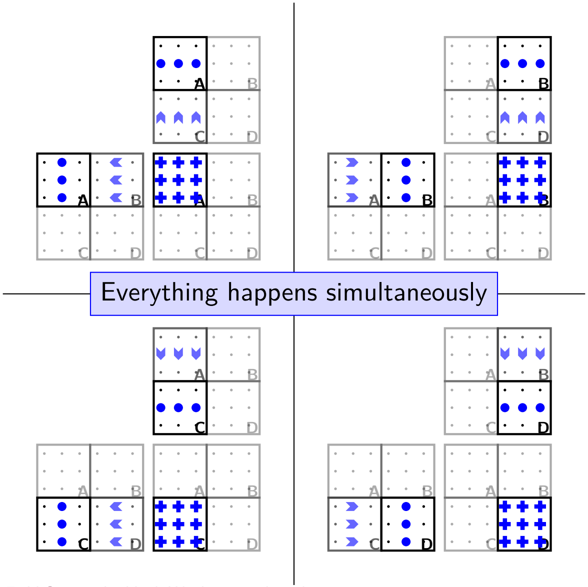
Typical context:

- r^2 nodes participate in a big matrix product of two $n \times n$ matrices. (n: dozens)
- Each "owns" a submatrix $\frac{n}{r} \times \frac{n}{r}$ of both inputs and the output.
- Simple case: each node is ok with allocating space for $\frac{n^2}{r^2}$ transforms, but not much more.

Parallelizing `lingen` carefully

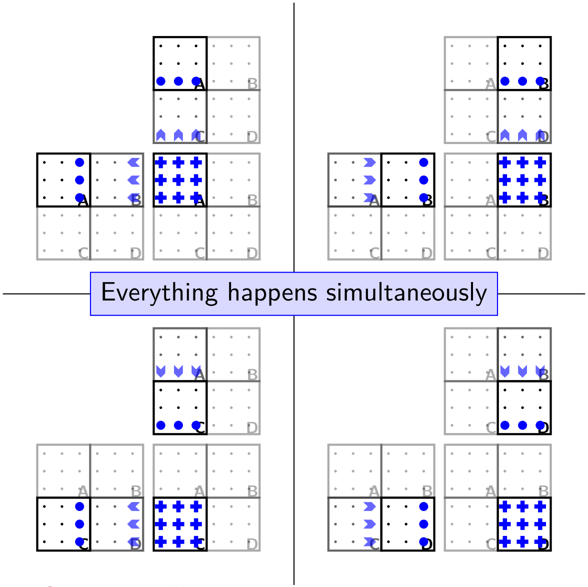


Parallelizing `lingen` carefully



Everything happens simultaneously

Parallelizing `lingen` carefully



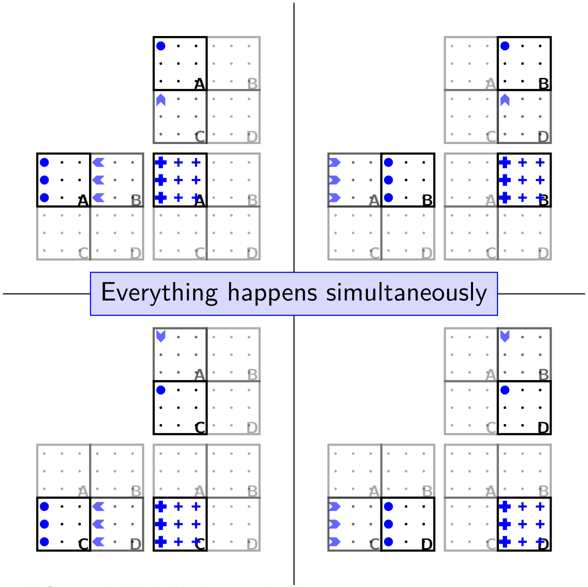
Everything happens simultaneously

Memory cost

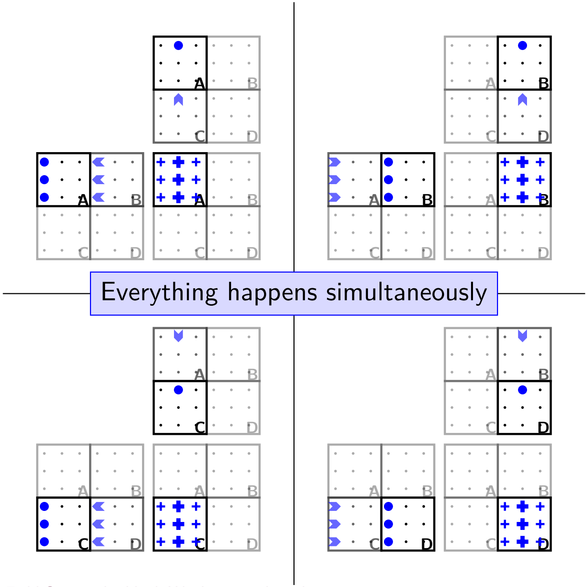
Each node here needs space for $\frac{n^2}{r^2}$ AND for $2(r-1)\frac{n}{r}$ transforms from other nodes.

This may be too much in certain cases.

Parallelizing `lingen`: less memory

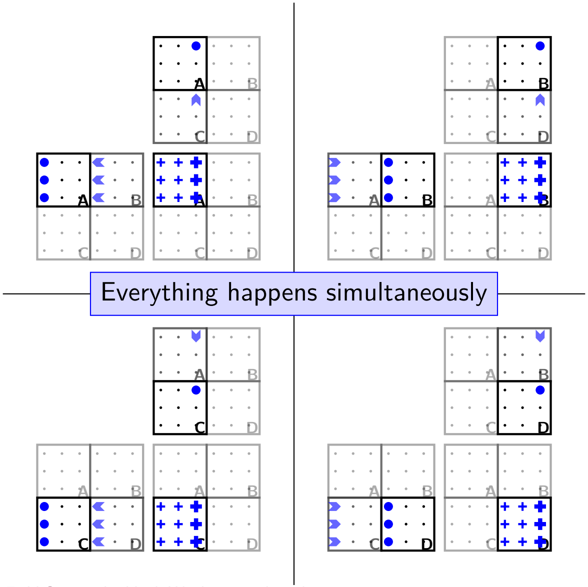


Parallelizing `lingen`: less memory

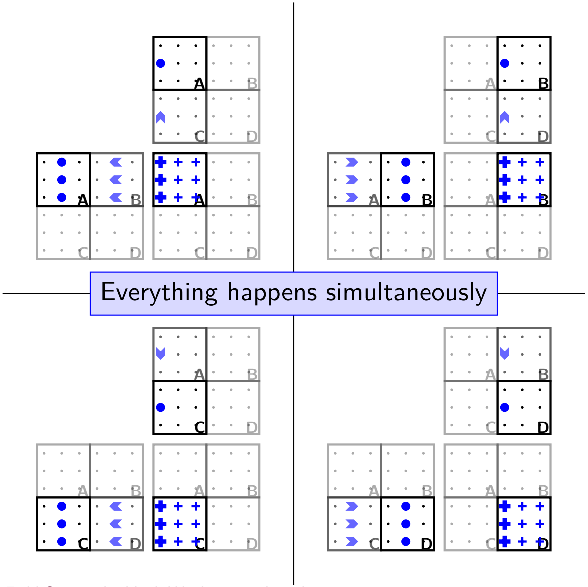


Everything happens simultaneously

Parallelizing `lingen`: less memory

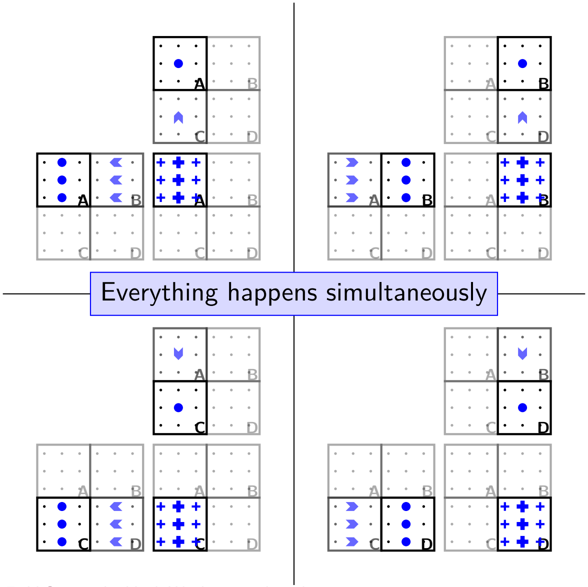


Parallelizing `lingen`: less memory

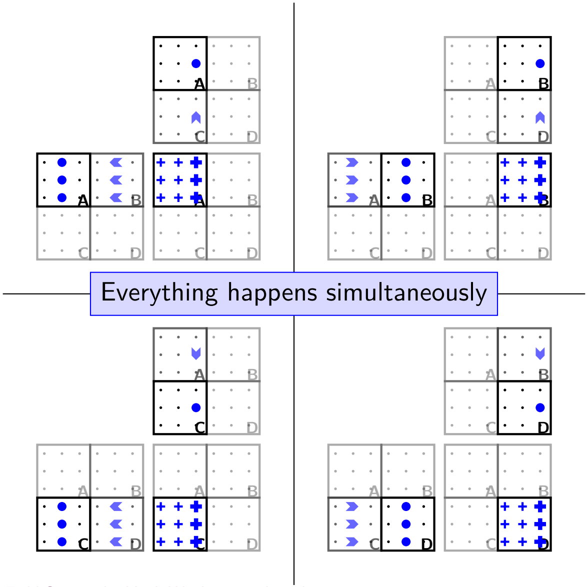


Everything happens simultaneously

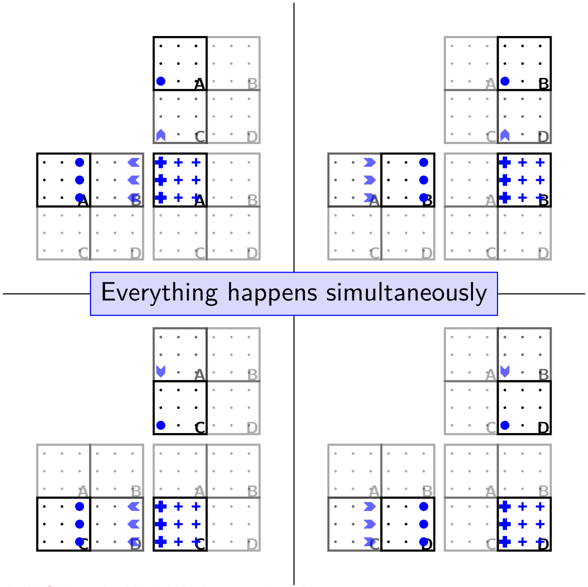
Parallelizing `lingen`: less memory



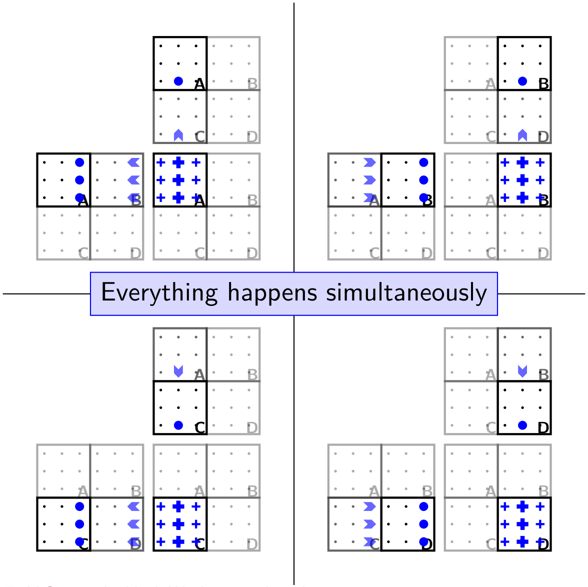
Parallelizing `lingen`: less memory



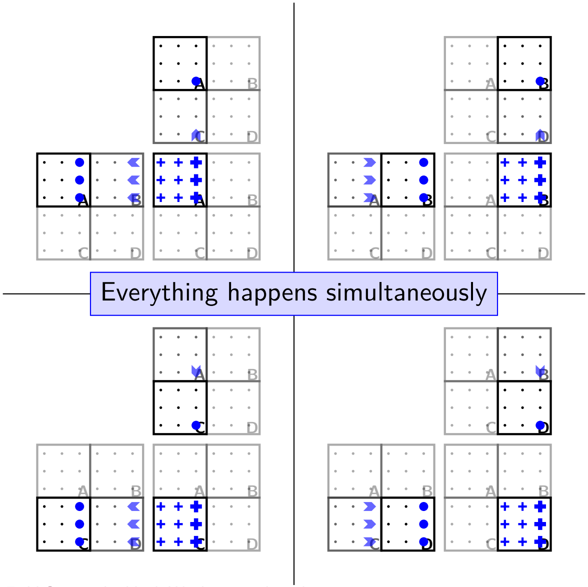
Parallelizing `lingen`: less memory



Parallelizing `lingen`: less memory



Parallelizing `lingen`: less memory

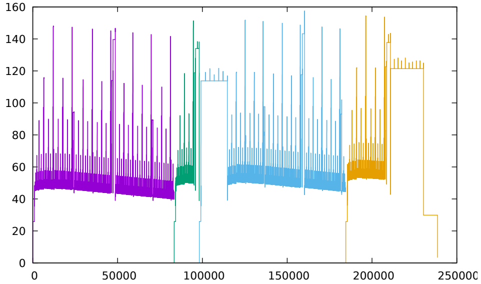


Memory cost

Each node here needs space for $\frac{n^2}{r^2} + (r-1)\frac{n}{r} + (r-1)$ transforms.

- This is achieved only by reorganizing the scheduling of computations and communications.
- Now this may still be too much in certain cases. Then we may want to split the computation even more, at the expense of recomputing several transforms.

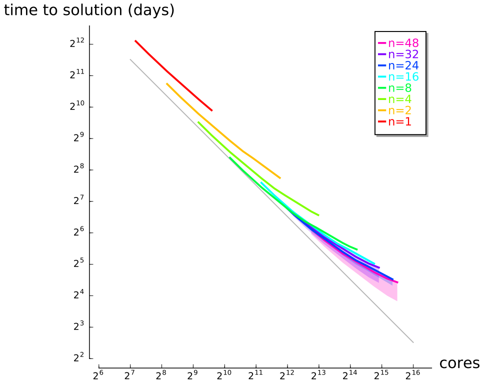
Keeping track of memory is important!



We can adjust the scheduling at each recursion depth.

Better memory usage $\rightarrow$ better scaling

We can predict the total runtime of BW quite well.



BW scales! (more than people tend to think).