Comparison of Block-Lanczos and Block-Wiedemann for Solving Linear Systems in Large Factorizations

A. Kruppa

Centrum Wiskunde & Informatica
Amsterdam

Workshop on Computational Number Theory 2011

Outline

Outline

Factoring with Congruent Squares

- Sieving-based factoring algorithms (QS, NFS) construct congruent squares: $X^2 \equiv Y^2 \pmod{N}$
- If $X \not\equiv \pm Y \pmod{N}$, then $\gcd(X - Y, N)$ is a proper factor
- So how do we find congruent squares?
  1. Sieving step: Find a lot of relations, i.e., pairs of congruent values that both factor over a small set of primes
  2. Linear Algebra step: Find a subset of them such that in the product both sides are squares

Constructing Congruent Squares: Example

Example: Factor 77

$$\begin{aligned}
80 &= 2^4 \times 5^1 \equiv 3^1 = 3 \\
125 &= 5^3 \equiv 2^4 \times 3^1 = 48 \\
160 &= 2^5 \times 5^1 \equiv 2^1 \times 3^1 = 6 \\
162 &= 2^1 \times 3^4 \equiv 2^3 = 8
\end{aligned}$$

- Want square product: all primes in even exponent. Look at exponent vectors

Constructing Congruent Squares: Example

Example: Factor 77

$80 =$	4		1	$\equiv$		1	$= 3$
$125 =$			3	$\equiv$	4	1	$= 48$
$160 =$	5		1	$\equiv$	1	1	$= 6$
$162 =$	1	4		$\equiv$	3		$= 8$

- Interested only in even or odd: look at exponent vectors over $\mathbb{F}_2$

Constructing Congruent Squares: Example

Example: Factor 77

$80 =$		^{1}	$\equiv$		^{1}	$= 3$
$125 =$		^{1}	$\equiv$		^{1}	$= 48$
$160 =$	^{1}	^{1}	$\equiv$	^{1}	^{1}	$= 6$
$162 =$	^{1}		$\equiv$	^{1}		$= 8$

- Find linear combination of exponent vectors over $\mathbb{F}_2$ that adds to zero vector: write exponent vectors as columns of a matrix, find a kernel vector

Constructing Congruent Squares: Example

Example: Factor 77

$80 =$		1	$\equiv$		1	$= 3$
$125 =$		1	$\equiv$		1	$= 48$
$160 =$	1	1	$\equiv$	1	1	$= 6$
$162 =$	1		$\equiv$	1		$= 8$

- One solution: use relations $80 \equiv 3$, $160 \equiv 6$, and $162 \equiv 8 \pmod{77}$

Constructing Congruent Squares: Example

Example: Factor 77

$80 =$		1	$\equiv$		1	$= 3$
$125 =$		1	$\equiv$		1	$= 48$
$160 =$	1	1	$\equiv$	1	1	$= 6$
$162 =$	1		$\equiv$	1		$= 8$

- One solution: use relations $80 \equiv 3$, $160 \equiv 6$, and $162 \equiv 8 \pmod{77}$
- Product: $1440^2 \equiv 12^2 \pmod{77}$. $\gcd(1440 - 12, 77) = 7$

Constructing Congruent Squares: Example

Example: Factor 77

$$\begin{aligned}
80 &= \quad {}^{1} \equiv \quad {}^{1} = 3 \\
125 &= \quad {}^{1} \equiv \quad {}^{1} = 48 \\
160 &= \quad {}^{1} \quad {}^{1} \equiv \quad {}^{1} \quad {}^{1} = 6 \\
162 &= \quad {}^{1} \equiv \quad {}^{1} = 8
\end{aligned}$$

- One solution: use relations $80 \equiv 3$, $160 \equiv 6$, and $162 \equiv 8 \pmod{77}$
- Product: $1440^2 \equiv 12^2 \pmod{77}$. $\gcd(1440 - 12, 77) = 7$
- **Construct congruent squares from relations by finding kernel vectors of a binary matrix**

Shape of the Matrices

- Sparse overall (few prime factors in each relation=column), rows corresponding to small primes are heavy

RSA768

Input number of 232 digits
Matrix size 192 795 550 $\times$ 192 796 550, weight 27 797 115 920, average column weight 144.2.

RSA190

Input number of 190 digits
Matrix size 33 218 122 $\times$ 33 643 088, total weight 2 115 794 780, average column weight 62.9.

Outline

Algorithms for Finding Kernel Vectors

- Gaussian Elimination, bad: $O(n^3)$, matrix fill in
- Iterative methods instead: Lanczos, Wiedemann: all $O(wn^2)$ (w average column weight)
- Both Block-Lanczos (BL) and Block-Wiedemann (BW) used in practice for factoring

The RSA768 Matrix

- Was solved by BW
- Total CPU time: about 160 core years, 119 days elapsed
- Intended race BW vs. BL
- BW finished too fast, BL code was not ready
- Current project: get BL ready for RSA768 matrix, compare speed

Outline

The Lanczos Algorithm

- Solve $Ax = y$, symmetric A in $K^{n,n}$, $x \in K^n$, $y \neq 0 \in K^n$
- Our matrix B is not symmetric, set $A = B^T B$, compute $Av = B^T(Bv)$
- Create orthogonal base for RHS with known preimage $\{Av_1, \ldots, Av_m\}$, $m = \dim \mathcal{K}(A, v_1)$
- Express y in that base: $y = \sum \frac{\langle y, Av_i \rangle}{|Av_i|^2} Av_i$
- Then $x = \sum \frac{\langle b, Av_i \rangle}{|Av_i|^2} v_i$ is a solution
- Homogeneous system: find distinct x_1, x_2 for random y, $x_1 - x_2$ is kernel vector

The Lanczos Algorithm

- The Lanczos iteration:

$$v_{i+1} = Av_i - \frac{\langle Av_i, Av_i \rangle}{\langle v_i, Av_i \rangle} v_i - \frac{\langle Av_i, Av_{i-1} \rangle}{\langle v_{i-1}, Av_{i-1} \rangle} v_{i-1}$$

- $A(Av_i)$ automatically orthogonal to $Av_1, \ldots, Av_{i-2}$
- Lanczos iteration orthogonalizes Av_{i+1} w.r.t. Av_i, Av_{i-1}
- Needs $m \approx n$ iterations, 2 matrix mul $(B^T(Bv_i))$, fixed number of scalar ops in each
- Problem in $\mathbb{F}_2$: self-orthogonal vectors $\langle v_i, Av_i \rangle = 0$ $\rightarrow$ zero denominator

The Block Lanczos Algorithm

- Block Algorithm: each column vector element is itself a length-b row vector (b blocking factor, e.g, $b = 128$)
- Block vector V_i is basis for vector space of dim $= 128$
- Orthogonalize these subspaces instead of individual vectors
- Cover (almost) 128 dimensions of RHS in each iteration, need only (about) $n/128$ iterations
- Word-wide bit operations ($+$:XOR, $*$: AND) treat whole block element in a single instruction

The Block Lanczos Algorithm

- Block-Lanczos uses modified iteration:

$$V_{i+1} = AV_i + V_i D_{i+1} + V_{i-1} E_{i+1} + V_{i-2} F_{i+1}$$

  where D_i, E_i, F_i are 128×128 matrices
- Scalar products are now $F_2^{n \times b}$ by $F_2^{b \times b}$ matrix products: complexity $O(nb^2)$, limits blocking factor
- Six such operations per iteration: 3 above, $\langle AV_i, V_i \rangle$, $\langle AV_i, AV_i \rangle$, update solution vector X
- Cost of AV_i is in $O(nwb)$
- $O(n/b)$ iterations, total cost $O(n^2 w + n^2 b)$

Outline

The Wiedemann Algorithm

1. Generate Krylov sequence $u^T v, u^T Av, u^T A^2 v \ldots, u^T A^{2n} v$
2. Compute minimal polynomial $f(x)$ s.t. $f(A) = 0$ (Berlekamp-Massey)
3. Evaluate $x = (f(A)/A)v = \sum f_i A^{i-1} v$. (Can patch if $f_0 \neq 0$)

- In principle, no auxiliary operation during (1), (3)
- Can compute several independent Krylov sequences, makes BM harder but still acceptable
- Evaluation can be split into independent pieces by remembering some $A^i v$ from Krylov sequence

Comparison: BL and BW in Theory

Block-Lanczos

1. About $2n/128$ matrix-vector multiplies (half by transpose)
2. Total of 6 auxiliary operations of $O(b^2)$: $\langle AV_i, V_i \rangle$, $\langle AV_i, AV_i \rangle$, $V_i D$, $V_{i-1}E$, $V_{i-2}F$, update solution vector
3. Iterations strictly sequential

Block-Wiedemann

1. $3n/128$ matrix-vector products (Krylov: $2n/128$, evaluation: $n/128$). No transposes
2. *No* auxiliary operations (in theory)
3. Inherent parallelism: split Krylov sequence, evaluation

Outline

Previous Work

- Starting point: complete implementation of Block-Lanczos by P. L. Montgomery
- Support for distributed computing with MPI
- No support for multi-threading
- Support for SSE instructions, but not AltiVec (128-bit SIMD instructions)

MPI/Multi-Threading

- Originally parallelization only via MPI
- Not efficient for shared-memory multi-core machines, overhead
- Added Multi-threading for Av, inner products, scalar products
- On NUMA systems, worthwhile to run separate MPI tasks on each NUMA domain, ensure local accesses
- Tried lots of variants of assigning tasks to threads (e.g., splitting vectors into pieces of half width for Coppersmith multiplication to make tables fit cache) – largely unsuccessful

Cache files

- Problem: matrix start-up very slow (reading, parsing, distributing matrix data)
- For RSA768: more than 10 hours
- Makes test/timing runs cumbersome
- Solution: dump processed matrix data to “cache files”, read back on program start
- Can create cache files single-threaded, in little memory ($\approx$ 5h)
- Cache files depend on topology
- Starting from cache files: 5 minutes

Homogeneous Systems

- Lanczos constructs orthogonal base $\{Av_1, \ldots, Av_m\}$ for RHS ($m = \dim \mathcal{K}(A, v_1)$)
- It orthogonalizes each new vector w.r.t. all previous ones
- If we already have complete base for subspace, new vector Av_{m+1} becomes zero
- But not necessarily $v_{m+1} = 0$, this is a useful kernel vector
- Idea works for Block-Lanczos, produces block of kernel vectors
- Eliminates storage for solution vector, 1 scalar multiply per iteration

Small rank F

- Block-Lanczos iteration:
$$V_{i+1} = AV_i + V_i D_{i+1} + V_{i-1} E_{i+1} + V_{i-2} F_{i+1}$$
- Matrix F chooses columns that were not used for computing V_i
- Number of omitted column is small, avg 0.76
- Thus $\mathrm{rank}\, F$ is small, usually < 3
- No need for $O(b^2)$ block-vector/block-matrix mult
- Find base for F, mul by base vectors, $O(b)$
- Eliminates another $O(b^2)$ operation, now only 4 left

Outline

History, Architecture

- IBM pSeries 575, total of 108 nodes, 16 dual-core IBM Power6 each (3456 cores total)
- Most nodes have 128GB memory, some have 256GB. Total 15.75 TB.
- Nodes are organized as 4 MCM with 4 CPUs each. Shared memory, faster within MCM
- Each node connected with 4 Infiniband links, 160 Gbit/s
- Each Power6 core has 64KB + 64KB L1, 4MB L2, shared 32MB L3 cache. 4.7 GHz clock.
- TOP500: ranked as 28th in November 2008, 303rd currently

RSA768 on Huygens

- Block-Wiedemann on Intel: CPU time: about 160 core years, 119 days elapsed
- Block-Lanczos ($b = 512$, homogeneous, 1 MPI job/MCM, 16 threads/MCM)

Nr. nodes	CPU	Elapsed	Elap.×cores
1	94.3y	612d	53.7y
4	98.1y	210d	73.5y
9	99.4y	123d	97.2y
16	105y	86.8d	122y

RSA768 on BBQ

- Compute workstation "barbecue" at CARAMEL lab, LORIA
- Quad-Hexcore (Xeon E7540), 2GHz, 512GB memory
- Hyper-Threading, 2 threads per core
- Running 4 MPI jobs (bound to node), 12 threads

b	CPU	Elapsed	Elap.×cores	
256	110y	916d	60.2y	
512	98.0y	807d	53.1y	
512	118y	965d	63.5y	(non-homogeneous)

RSA190

- Size 33.2M × 33.6M, weight 2.1G

On Huygens

Nr. nodes	CPU	Elapsed	Elap.×cores
1	1.33y	9.39d	300d

On BBQ

b	CPU	Elapsed	Elap.×cores	
256	344d	9.0d	216d	
512	403d	10.1d	242d	
512	423d	10.5d	252d	(non-homogeneous)

Conclusion

- Block-Lanczos is competitive with Block-Wiedemann if computation happens on one high-end system
- Large factorizations in a research context often use whatever resources are available - often scattered
- Example: RSA768 matrix jobs ran in Lausanne, several GRID5000 sites in France, and in Tokyo
- Block-Wiedemann can make use of such scattered resources, Block-Lanczos can not