Distributed Linear Algebra

Peter L. Montgomery
Microsoft Research, Redmond, USA

RSA 2000

January 17, 2000

Role of matrices in factoring n

- Sieving finds many $x_j^2 \equiv \prod p_i^{e_{ij}} \pmod{n}$.
- Raise jth to power $s_j = 0$ or 1, multiply.
- Left side always a perfect square. Right is a square if exponents $\sum_j e_{ij} s_j$ are even for all i.
- Matrix equation $\mathbf{Es} \equiv \mathbf{0} \pmod 2$, $\mathbf{E}$ known.
- Knowing $x^2 \equiv y^2 \pmod{n}$, test GCD($x \pm y$, n).
- Matrix rows represent primes p_i. Entries are exponents e_{ij}. Arithmetic is over GF(2).

Matrix growth on RSA Challenge

- RSA–140
- Jan-Feb, 1999
- 4 671 181 × 4 704 451
- Weight 151 141 999
- Omit primes < 40
- 99 Cray-C90 hours
- 75% of 800 Mb for matrix storage

- RSA–155
- August, 1999
- 6 699 191 × 6 711 336
- Weight 417 132 631
- Omit primes < 40
- 224 Cray-C90 hours
- 85% of 1960 Mb for matrix storage

Regular Lanczos

- **A** positive definite (real, symmetric) $n \times n$ matrix.
- Given **b**, want to solve $\mathbf{Ax} = \mathbf{b}$ for **x**.
- Set $\mathbf{w}_0 = \mathbf{b}$.
- $\mathbf{w}_{i+1} = \mathbf{A}\mathbf{w}_i - \Sigma_{0 \le j \le i} c_{ij}\mathbf{w}_j \quad \text{if} \quad i \ge 0$
- $c_{ij} = \mathbf{w}_j^{\mathrm{T}}\mathbf{A}^2\mathbf{w}_i \,/\, \mathbf{w}_j^{\mathrm{T}}\mathbf{A}\mathbf{w}_j$
- Stop when $\mathbf{w}_{i+1} = \mathbf{0}$.

Claims

- $\mathbf{w}_j^T \mathbf{A} \mathbf{w}_j \neq 0$ if $\mathbf{w}_j \neq \mathbf{0}$ ($\mathbf{A}$ is positive definite).
- $\mathbf{w}_j^T \mathbf{A} \mathbf{w}_i = 0$ whenever $i \neq j$
  (by choice of c_{ij} and symmetry of $\mathbf{A}$).
- Eventually some $\mathbf{w}_{i+1} = \mathbf{0}$, say for $i = m$
  (otherwise too many $\mathbf{A}$-orthogonal vectors).
- $\mathbf{x} = \Sigma_{0 \leq j \leq m} (\mathbf{w}_j^T \mathbf{b} / \mathbf{w}_j^T \mathbf{A} \mathbf{w}_j) \mathbf{w}_j$ satisfies $\mathbf{Ax=b}$
  (error $\mathbf{u=Ax-b}$ is in space spanned by $\mathbf{w}_j$'s
  but orthogonal to all $\mathbf{w}_j$, so $\mathbf{u}^T\mathbf{u}=0$ and $\mathbf{u=0}$).

Simplifying c_{ij} when $i > j+1$

- $\mathbf{w}_j^{\mathrm{T}}\mathbf{A}\mathbf{w}_j c_{ij} = \mathbf{w}_j^{\mathrm{T}}\mathbf{A}^2\mathbf{w}_i = (\mathbf{A}\mathbf{w}_j)^{\mathrm{T}}(\mathbf{A}\mathbf{w}_i)$
  $= (\mathbf{w}_{j+1} + \text{linear comb. of } \mathbf{w}_0 \text{ to } \mathbf{w}_j)^{\mathrm{T}}(\mathbf{A}\mathbf{w}_i)$
  $= 0$ (**A**-orthogonality).

- Recurrence simplifies to

  $\mathbf{w}_{i+1} = \mathbf{A}\mathbf{w}_i - c_{ii}\mathbf{w}_i - c_{i,i-1}\mathbf{w}_{i-1}$ when $i \geq 1$.

- Little history to save as i advances.

Major operations needed

- Pre-multiply $\mathbf{w}_i$ by $\mathbf{A}$.
- Inner products such as $\mathbf{w}_j^{\mathrm{T}}\mathbf{A}\mathbf{w}_j$ and $\mathbf{w}_j^{\mathrm{T}}\mathbf{A}^2\mathbf{w}_i = (\mathbf{A}\mathbf{w}_j)^{\mathrm{T}}(\mathbf{A}\mathbf{w}_i)$.
- Add scalar multiple of one vector to another.

Adapting to **Bx=0** over GF(2)

- **B** is $n_1 \times n_2$ with $n_1 < n_2$, not symmetric.
  Solve $\mathbf{Ax} = \mathbf{0}$ where $\mathbf{A} = \mathbf{B}^T\mathbf{B}$. **A** is $n_2 \times n_2$.
  $\mathbf{B}^T$ has small nullspace in practice.
- Right side zero, so Lanczos gives $\mathbf{x} = \mathbf{0}$.
  Solve $\mathbf{Ax} = \mathbf{Ay}$ where **y** is random.
- $\mathbf{u}^T\mathbf{u}$ and $\mathbf{u}^T\mathbf{Au}$ can vanish when $\mathbf{u} \neq \mathbf{0}$.
  Solved by Block Lanczos (Eurocrypt 1995).

Block Lanczos summary

- Let N be the machine word length (typically 32 or 64) or a small multiple thereof.
- Vectors are $n_1 \times N$ or $n_2 \times N$ over GF(2).
- Exclusive OR and other hardware bitwise instructions operate on N-bit data.
- Recurrences similar to regular Lanczos.
- Approximately $n_1/(N–0.76)$ iterations.
- Up to N independent solutions of **Bx=0**.

Block Lanczos major operations

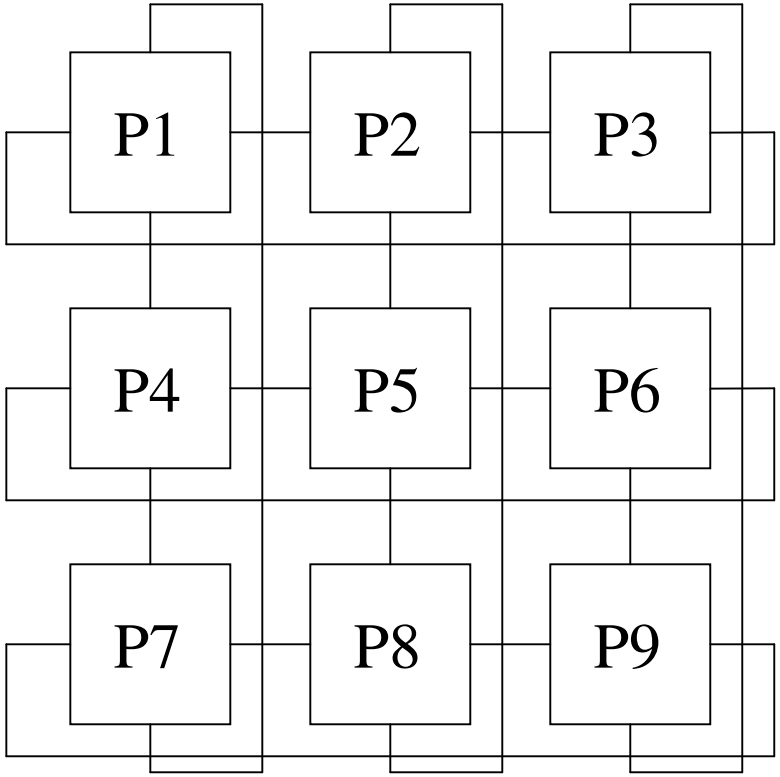
- Pre-multiply $n_2 \times N$ vector by $\mathbf{B}$.
- Pre-multiply $n_1 \times N$ vector by $\mathbf{B}^T$.
- $N \times N$ inner product of two $n_2 \times N$ vectors.
- Post-multiply $n_2 \times N$ vector by $N \times N$ matrix.
- Add two $n_2 \times N$ vectors.

How do we parallelize these?

Assumed processor topology

- Assume a $g_1 \times g_2$ toroidal grid of processors.
- A ***torus*** is a rectangle with its top connected to its bottom, and left to right (doughnut).
- Need fast communication to/from immediate neighbors north, south, east, and west.
- Processor names are p_{rc} where r is modulo g_1 and c is modulo g_2.
- Set **gridrow**(p_{rc}) = r and **gridcol**(p_{rc}) = c.

A torus of processors



Example: 3x3 torus system

Matrix row and column guardians

- For $0 \le i < n_1$, a processor **rowguard**(i) is responsible for entry i, in all $n_1 \times N$ vectors.
- For $0 \le j < n_2$, a processor **colguard**(j) is responsible for entry j, in all $n_2 \times N$ vectors.
- Processor-assignment algorithms aim for load balancing.

Three major operations

- Vector addition is *pointwise*. When adding two $n_2 \times N$ vectors, processor **colguard**(j) does the j-th entries. Data is local.
- Likewise for $n_2 \times N$ vector by $N \times N$ matrix.
- Processors form partial $N \times N$ inner products. Central processor sums them.
- These operations need little communication.
- Workloads are O(#columns assigned).

Allocating **B** among processors

- Let **B** = (b_{ij}) for $0 \le i < n_1$ and $0 \le j < n_2$.
- Processor p_{rc} is responsible for all b_{ij} where **gridrow**(**rowguard**(i)) = r and **gridcol**(**colguard**(j)) = c.
- When pre-multiplying by **B**, the input data from **colguard**(j) will arrive along grid column c, and the output data for **rowguard**(i) will depart along grid row r.

Multiplying **u** = **B**$\mathbf{v}$ where **u** is $n_1 \times N$ and **v** is $n_2 \times N$

- Distribute each **v**[j] to all p_{rc} with **gridcol**(**colguard**(j)) = c. That is, broadcast each **v**[j] along one column of the grid.
- Each p_{rc} processes all of its b_{ij}, building partial **u**[i] outputs.
- Partial **u**[i] values are summed as they advance along a grid row to **rowguard**(i).
- Individual workloads depend upon **B**.

Actions by p_{rc} during multiply

- Send/receive all **v**[j] with **gridcol**(**colguard**(j)) = c.
- Zero all **u**[i] with **rowguard**(i) = $p_{r,c+1}$.
- At time t where $1 \le t \le g_2$, adjust all **u**[i] with **rowguard**(i) = $p_{r,c+t}$ (t nodes east).
- If $t < g_2$, ship these **u**[i] west to $p_{r,c-1}$ and receive other **u**[i] from $p_{r,c+1}$ on the east.
- Want balanced workloads at each t.

Multiplication by $\mathbf{B}^T$

- Reverse roles of matrix rows and columns.
- Reverse roles of grid rows and columns.
- $\mathbf{B}^T$ and $\mathbf{B}$ can share storage since same processor handles $(\mathbf{B})_{ij}$ during multiply by $\mathbf{B}$ as handles $(\mathbf{B}^T)_{ji}$ during multiply by $\mathbf{B}^T$.

Major memory requirements

- Matrix data is split amongst processors.
- With 65536×65536 cache-friendly blocks, an entry needs only two 16-bit offsets.
- Each processor needs one vector of length $\max(n_1/g_1, n_2/g_2)$ and a few of length n_2/g_1g_2, with N bits per entry.
- Central processor needs one vector of length n_2 plus **rowguard** and **colguard**.

Major communications during multiply by **B**

- Broadcast each $\mathbf{v}[j]$ along entire grid column. Ship n_2N bits to each of g_1-1 destinations.
- Forward partial $\mathbf{u}[i]$ along grid row, one node at a time. Total $(g_2-1)n_1N$ bits.
- When $n_2 \approx n_1$, communication for **B** and $\mathbf{B}^T$ is $2(g_1+g_2-2)n_1N$ bits per iteration.
- $2(g_1+g_2-2){n_1}^2$ bits after n_1/N iterations.

Choosing grid size

- Large enough that matrix fits in memory.
- Matrix storage is about $4w/g_1g_2$ bytes per processor, where w is total matrix weight.
- Try to balance I/O and computation times.
- Multiply cost is $O(n_1w/g_1g_2)$ per processor.
- Communications cost $O((g_1+g_2-2)n_1^2)$.
- Prefer a square grid, to reduce g_1+g_2.

Choice of N and matrix

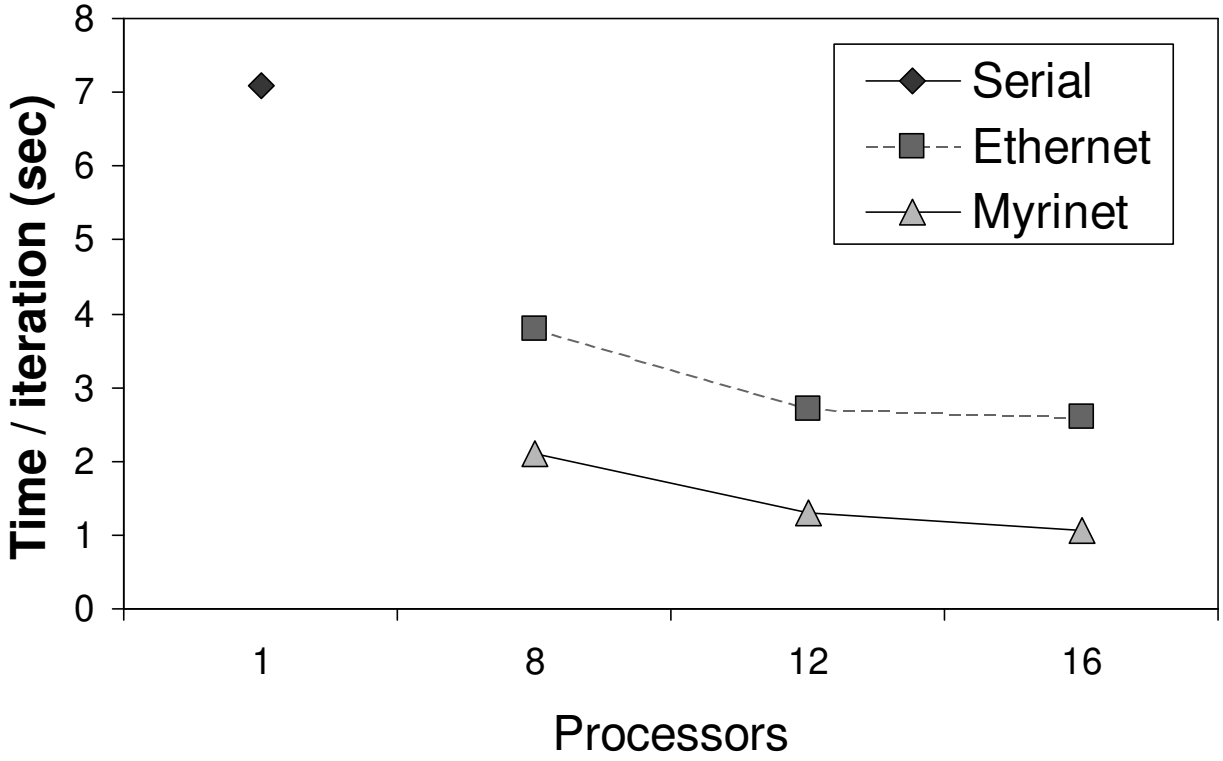
- Prefer smaller but heavier matrix if it fits, to lessen communications.
- Higher N yield more dependencies, letting you omit the heaviest rows from the matrix.
- Larger N means fewer but longer messages.
- Size of vector elements affects cache.
- When N is large, inner products and post-multiplies by $N \times N$ matrices are slower.

Cambridge cluster configuration

- Microsoft Research, Cambridge, UK.
- 16 dual-CPU 300 MHz Pentium II's.
- Each node
  - 384 MB RAM
  - 4 GB local disk
- Networks
  - Dedicated fast ethernet (100 Mb/sec)
  - Myrinet, M2M-OCT-SW8 (1.28 Gb/sec)

Message Passing Interface (MPI)

- Industry Standard
- MPI implementations:
  - exist for the majority of parallel systems & interconnects
  - public domain (e.g. mpich) or commercial (e.g. MPI PRO)
- Supports many communications primitives including virtual topologies (e.g. torus).

Performance data

from MSR Cambridge cluster