Circuits for integer factorization

D. J. Bernstein
University of Illinois at Chicago

Exercise for the reader:

Find a nontrivial factor of

6366223796340423057152171586.

Exercise for the reader:
Find a nontrivial factor of
6366223796340423057152171586.

Small prime factors
are easy to find.

Larger primes are harder.

"Elliptic-curve method" (ECM)
scales surprisingly well.
(1987 Lenstra)

ECM has found a prime $\approx 2^{219}$.
(2005 Dodson; rather lucky;
$\approx 3 \cdot 10^{12}$ Opteron cycles)

`www.loria.fr/~zimmerma/records/p66`

For worst-case integers with two very large prime factors, ECM does not scale as well as "number-field sieve" (NFS). (1988 Pollard, et al.)

Latest record: NFS has found two prime factors $\approx 2^{332}$ of "RSA-200" challenge. (2005 Bahr/Boehm/Franke/Kleinjung; $\approx 5 \cdot 10^{18}$ Opteron cycles)

How much more difficult is it to find prime factors $\approx 2^{512}$ of an integer $n \approx 2^{1024}$?

`www.loria.fr/~zimmerma/records/rsa200`

This talk focuses on scalability.

Example: Trial division finds primes $\leq y$ dividing n using $y^{1+o(1)}$ easy operations.

(Here $o(1)$ means a function of y that converges to 0 as $y \to \infty$; could be $1/y$ or $-1/\log y$ or $10^6(\log\log\log y)^5/\log\log y$.)

ρ method (1975 Pollard), assuming standard conjectures: $y^{0.5+o(1)}$; therefore much faster than trial division once y is sufficiently large.

ECM finds primes $\leq y$ in n using
$\exp \sqrt{(2 + o(1))\log y \log\log y}$
easy operations. (1987 Lenstra)

Compare to trial division and ρ:
$y^{1+o(1)} = \exp((1 + o(1)) \log y)$;
$y^{0.5+o(1)} = \exp((0.5 + o(1)) \log y)$.

Easily see from these formulas
that ECM is much faster
than trial division and ρ
once y is sufficiently large.

(What is "sufficiently large"?
Many papers analyzing details.)

Extreme case, $y = \lfloor \sqrt{n} \rfloor$:
ECM finds all primes in n using
$\exp \sqrt{(1 + o(1)) \log n \log \log n}$
easy operations as $n \to \infty$.

NFS has better scalability:
NFS finds all primes in n using
$L^{1.901\ldots + o(1)}$ easy operations
as $n \to \infty$, where $L =$
$\exp((\log n)^{1/3} (\log \log n)^{2/3})$.

(1/3, exponent 1.922 . . .:
1993 Buhler/Lenstra/Pomerance;
1.901 . . .: 1993 Coppersmith)

These NFS operations take
$L^{1.901...+o(1)}$ seconds

on a standard serial computer
costing $L^{0.950...+o(1)}$ €.

"TWINKLE": another circuit
costing $L^{0.950...+o(1)}$ €

that performs same operations in
$L^{1.901...+o(1)}$ seconds.

(2000 Lenstra/Shamir)

A better-designed circuit costing
$L^{0.950...+o(1)}$ €

can perform same operations in
$L^{1.426...+o(1)}$ seconds.

(2001 Bernstein)

Better parameter choices:

Can find all primes in n using $L^{1.185...+o(1)}$ seconds

with an NFS circuit costing $L^{0.790...+o(1)}$ €.

(2001 Bernstein)

Can vary circuit size, but $L^{1.976...+o(1)}$ € · seconds is

best price-performance ratio

in this class of algorithms.

Also vary serial-computer size.

Best price-performance ratio: $L^{2.760...+o(1)}$ € · seconds.

(2002 Pomerance)

Conclusion: Circuit factors n
much more quickly than
standard serial computer
of the same size,
once n is large enough.

(What about $n \approx 2^{1024}$?
Much more difficult analysis.
Many estimates in new papers,
usually < 1 year for $< 10^9$ €.)

How is this possible?
How can a circuit be
so much faster than
a standard serial computer?

Computational complexity

Start with simpler problem.
How fast is sorting?

Input: array of n numbers.
Each number in $\{1, 2, \ldots, n^2\}$,
represented in binary.

Output: array of n numbers,
in increasing order,
represented in binary;
same multiset as input.

A machine is given the input
and computes the output.
How much time does it use?

The answer depends on
how the machine works.

Possibility 1: The machine is a
"1-tape Turing machine
using selection sort."

Specifically: The machine has
a 1-dimensional array
containing $n^{1+o(1)}$ "cells."
Each cell stores $n^{o(1)}$ bits.

Input and output are
stored in these cells.

The machine also has a
"head" moving through array.
Head contains $n^{o(1)}$ cells.

Head can see the cell at
its current array position;
perform arithmetic etc.;
move to adjacent array position.

Selection sort: Head
looks at each array position,
picks up the largest number,
moves it to the end of the array,
picks up the second largest,
etc.

Moving to adjacent array position takes $n^{o(1)}$ seconds.

Moving a number to end of array takes $n^{1+o(1)}$ seconds.

Same for comparisons etc.

Total sorting time:
$n^{2+o(1)}$ seconds.

Cost of machine:
$n^{1+o(1)}$ €
for $n^{1+o(1)}$ cells.

Negligible extra cost for head.

Possibility 2: The machine is a
"2-dimensional RAM
using merge sort."

Machine has $n^{1+o(1)}$ cells
in a 2-dimensional array:
$n^{0.5+o(1)}$ rows, $n^{0.5+o(1)}$ columns.
Machine also has a head.

Merge sort: Head recursively
sorts first $\lfloor n/2 \rfloor$ numbers;
sorts last $\lceil n/2 \rceil$ numbers;
merges the sorted lists.

Merging requires $n^{1+o(1)}$ jumps to "random" array positions.

Average jump: $n^{0.5+o(1)}$ moves to adjacent array positions.

Each move takes $n^{o(1)}$ seconds.

Total sorting time:
$n^{1.5+o(1)}$ seconds.

Cost of machine: once again $n^{1+o(1)}$ €.

Possibility 3: The machine is a "pipelined 2-dimensional RAM using radix-2 sort."

Machine has $n^{1+o(1)}$ cells in a 2-dimensional array.

Each cell in the array has network links to the 2 adjacent cells in the same column.

Each cell in the bottom row has network links to the 2 adjacent cells in the bottom row.

Machine also has a CPU
attached to bottom-left cell.

CPU can read/write any cell by
sending request through network.
While waiting for response,
can send subsequent requests.

CPU can read an entire row
of $n^{0.5+o(1)}$ cells
in $n^{0.5+o(1)}$ seconds.
Sends all requests,
then receives responses.

Radix-2 sort: CPU
shuffles array using bit 0,
even numbers before odd.
3 1 4 1 5 9 2 6 $\mapsto$
4 2 6 3 1 1 5 9.

Then using bit 1:
4 1 1 5 9 2 6 3.

Then using bit 2:
1 1 9 2 3 4 5 6.

Then using bit 3:
1 1 2 3 4 5 6 9.

etc.

Each shuffle takes
$n^{1+o(1)}$ seconds.

$n^{o(1)}$ shuffles.

Total sorting time:
$n^{1+o(1)}$ seconds.

Cost of machine: once again
$n^{1+o(1)}$ €.

Possibility 4: The machine is a
"2-dimensional mesh
using Schimmler sort."

Machine has $n^{1+o(1)}$ cells
in a 2-dimensional array.
Each cell has network links
to the 4 adjacent cells.

Machine also has a CPU
attached to bottom-left cell.
CPU broadcasts instructions
to all of the cells, but
cells do most of the processing.

Sort row of $n^{0.5+o(1)}$ cells
in $n^{0.5+o(1)}$ seconds:

Sort each pair in parallel.

$\underline{3\ 1}\ \underline{4\ 1}\ \underline{5\ 9}\ \underline{2\ 6} \mapsto$

1 3 1 4 5 9 2 6

Sort alternate pairs in parallel.

$1\ \underline{3\ 1}\ \underline{4\ 5}\ \underline{9\ 2}\ 6 \mapsto$

1 1 3 4 5 2 9 6

Repeat until number of steps
equals row length.

Sort *each* row, in parallel,
in $n^{0.5+o(1)}$ seconds.

Schimmler sort:

Recursively sort quadrants
in parallel. Then four steps:

- Sort each column in parallel.
- Sort each row in parallel.
- Sort each column in parallel.
- Sort each row in parallel.

With proper choice of
left-to-right/right-to-left
for each row, can prove
that this sorts whole array.

For example, assume that this 8×8 array is in cells:

3	1	4	1	5	9	2	6
5	3	5	8	9	7	9	3
2	3	8	4	6	2	6	4
3	3	8	3	2	7	9	5
0	2	8	8	4	1	9	7
1	6	9	3	9	9	3	7
5	1	0	5	8	2	0	9
7	4	9	4	4	5	9	2

Recursively sort quadrants,
top $\rightarrow$, bottom $\leftarrow$:

1	1	2	3	2	2	2	3
3	3	3	3	4	5	5	6
3	4	4	5	6	6	7	7
5	8	8	8	9	9	9	9
1	1	0	0	2	2	1	0
4	4	3	2	5	4	4	3
7	6	5	5	9	8	7	7
9	9	8	8	9	9	9	9

Sort each column
in parallel:

1	1	0	0	2	2	1	0
1	1	2	2	2	2	2	3
3	3	3	3	4	4	4	3
3	4	3	3	5	5	5	6
4	4	4	5	6	6	7	7
5	6	5	5	9	8	7	7
7	8	8	8	9	9	9	9
9	9	8	8	9	9	9	9

Sort each row in parallel, alternately $\leftarrow$, $\rightarrow$:

0	0	0	1	1	1	2	2
3	2	2	2	2	2	1	1
3	3	3	3	3	4	4	4
6	5	5	5	4	3	3	3
4	4	4	5	6	6	7	7
9	8	7	7	6	5	5	5
7	8	8	8	9	9	9	9
9	9	9	9	9	9	8	8

Sort each column

in parallel:

0	0	0	1	1	1	1	1
3	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	5	4	4	4	4
6	5	5	5	6	5	5	5
7	8	7	7	6	6	7	7
9	8	8	8	9	9	8	8
9	9	9	9	9	9	9	9

Sort each row in parallel,

$\leftarrow$ or $\rightarrow$ as desired:

0	0	0	1	1	1	1	1
2	2	2	2	2	2	2	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	5
5	5	5	5	5	5	6	6
6	6	7	7	7	7	7	8
8	8	8	8	8	9	9	9
9	9	9	9	9	9	9	9

Sort one row
in $n^{0.5+o(1)}$ seconds.

All rows in parallel:
$n^{0.5+o(1)}$ seconds.

Total sorting time:
$n^{0.5+o(1)}$ seconds.

Cost of machine: once again
$n^{1+o(1)}$ €.

($n^{0.5+o(1)}$ on mesh:
1977 Thompson/Kung;
this very simple algorithm:
1987 Schimmler)

"VLSI algorithms" literature
contains similar improvements
in price-performance ratio
("AT") for many computations.

Consider, e.g.,
multiplying two n-bit integers.

Time $n^{1+o(1)}$
on standard serial computer
with $n^{1+o(1)}$ bits of memory.
(1971 Schönhage/Strassen,
using FFT; see also 2007 Fürer)

Knuth: "we leave the domain of conventional computer programming..."

Time $n^{1+o(1)}$
on a 1-dimensional mesh
of size $n^{1+o(1)}$.
(1965 Atrubin, elementary)

Time $n^{0.5+o(1)}$
on a 2-dimensional mesh
of size $n^{1+o(1)}$.
(1981 Brent/Kung, using FFT)

Some philosophical notes

1-tape Turing machines,
RAMs, 2-dimensional meshes
compute the same functions.

Prove this by proving that
each machine can simulate
computations on the others.

(We believe that *every*
reasonable model of computation
can be simulated by a
1-tape Turing machine.
"Church-Turing thesis.")

1-tape Turing machines,
RAMs, 2-dimensional meshes
compute the same functions
in polynomial time
at polynomial cost.

Prove this by proving that
simulations are polynomial.

(Is this true for every
reasonable model of computation?
Is it possible to build
a large quantum computer?
Poly-size quantum computer
can factor in polynomial time.
Can Turing machine do that?)

1-tape Turing machines,
RAMs, 2-dimensional meshes
do not compute
the same functions
within, e.g., time $n^{1+o(1)}$
and cost $n^{1+o(1)}$.

Example: 1-tape Turing machine
cannot sort in $n^{1+o(1)}$ seconds.
Too local.

Example: 2-dimensional RAM
cannot sort in $n^{0.5+o(1)}$ seconds.
Too sequential.

Review of sorting times,
measured in seconds, for
machine costing $n^{1+o(1)}$ €:

$n^{2.0+o(1)}$: 1-tape Turing machine.
$n^{1.5+o(1)}$: 2-dimensional RAM.
$n^{1.0+o(1)}$: pipelined RAM.
$n^{0.5+o(1)}$: 2-dimensional mesh.

Why does anyone say that
sorting time is $n^{1+o(1)}$?
Why choose third machine?
Silly! Once n is large enough,
fourth machine is better.

Myth:
Parallel computation cannot
improve price-performance ratio;
p parallel computers
may reduce time by factor p
but increase cost by factor p.

Reality: Can often convert
a *large* serial computer
into p *small* parallel cells
with only mild slowdown.
Cost does *not*
increase by factor p.

Myth:
Designing a new machine
cannot produce more than a
small constant-factor speedup
compared to, e.g., a Pentium.
What matters is special-purpose
streamlining, such as reducing
instruction-decoding costs.

Reality: In 1997, DES Cracker
was 1000 times faster than a
set of Pentiums at the same price.
What matters is parallelism.

Future computers will be
massively parallel meshes.
Look at $o(1)$ details to see that
we've reached large enough n.

Computer designers will laugh at
today's RAM-style machines,
just as we laugh at
a 1-tape Turing machine.

Older meshes such as MasPar
had a limited market,
but n is much larger now.
See new wave of FPGA-based
supercomputers from SRC etc.

New myth:
We can continue
designing algorithms
and writing programs
for conventional computers,
and then put them on
mesh computers to reduce cost.

Reality: Optimizing AT
on a 2-dimensional mesh
has huge differences from
optimizing "operations"
on a conventional computer.
Example: NFS circuits use
completely different subroutines.

Current algorithm-analysis culture
—focus on operation counts;
maybe mention machine size
and communication complexity,
but only as a secondary issue—
will eventually be considered
shortsighted, archaic, obsolete.

Yes, it's fun, but it's doomed!

Have to redesign algorithms
and rewrite programs
from the ground up,
analyzing communication cost
and price-performance ratio.

NFS circuits in a nutshell

Most important NFS step:
find all factors $\leq y$ of
auxiliary numbers related to n.

Traditional method, "sieving":
$AT \in L^{2.85...+o(1)}$.

Parallel: $AT \in L^{2.37...+o(1)}$.

Better scalability from
many parallel ECM circuits:
$AT \in L^{2.08...+o(1)}$.

Also parallel linear algebra:
$AT \in L^{1.976...+o(1)}$.