Bound Optimization for Parallel Quadratic Sieving Using Large Prime Variations

T_{80}	$50529886889309962095129829271690197814291652588549587494032309395334675070111733 =$ $77846537490955596932233475185335826831 * 649096138607072269623077829406411690414 3$

Andrew G. West

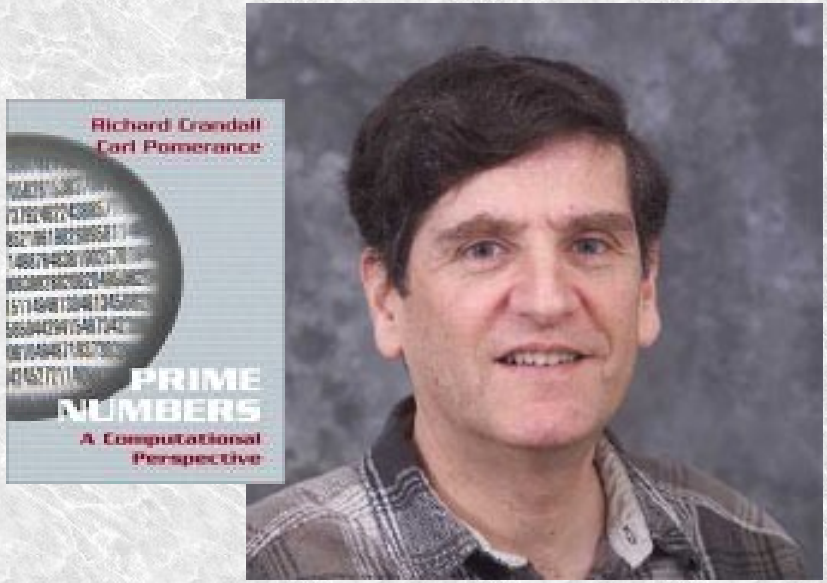
Advisor: Rance Necaise

Integer Factorization

- Fundamental Theorem of Arithmetic
  - Euclid: 300 B.C.
  - Gauss formalized in 1801
  - “Every prime greater than 1 has a unique representation as a product of primes”
- Significance of Factorization
  - Presumed difficulty forms basis of several crypto systems
  - Notably the RSA, a system securing large portion of e-commerce

Modern Factorization Methods

- Three Sub-Exponential Algorithms Exist
- Elliptic Curve Method (ECM)
  - Good for moderately sized factors (<25 digits), not huge semiprimes
  - Floating point arithmetic
- Quadratic Sieve (QS)
  - Pomerance, 1985
  - Has had many improvements
    - PPSIMPQS!
    - We will talk about PQS
  - RSA-129, 8 months, 600 CPUs
    - 40 quadrillion year estimate



- Number Field Sieve (NFS)
  - Builds on QS with algebraic rings and higher order polynomials
  - RSA-200, 18 months, dedicated 80 node cluster
  - Fastest known GP algorithm

Mathematical Background

- Trial Division = BAD!
  - Division costly at machine level
  - High overhead of finding primes
- Fermat's Difference of Squares:

$$N = a^2 - b^2 = (a + b)(a - b).$$

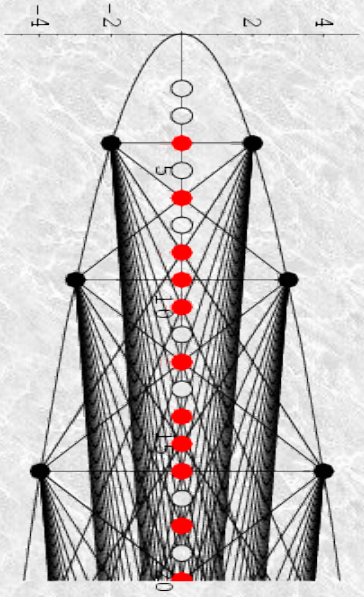
- Equality too constricting, instead...
- Congruence of Squares:

$$a^2 \equiv b^2 \mod N \text{ , where } a \not\equiv \pm b$$

$$a^2 - b^2 \equiv (a + b)(a - b) \equiv 0 \mod N$$

$$\gcd(a \pm b, N)$$

- Because congruence is weaker than equality, (a,b) pairs may produce trivial results in GCD (e.g. 1 or N)
- Thus methods like Dixon's, QS, and NFS find multiple (a,b) pairs to probabilistically ensure success.



Dixon's Factorization Method

- Predecessor and simplified version of QS:
- Select smoothness bound B, and add all primes <= B to set {FB}, the factor base
- Then use function:

$$f(x) = x^2 \mod N \text{ , where } x > \sqrt{N} \text{ and } x \in \mathbb{N}$$

- Test many x_i for the condition that $f(x_i)$ factors completely over the factor base, or mathematically, are {FB}-smooth
- Pairs (x_i, $f(x_i)$) that meet this criteria are called relations, and are stored

- We want to find a subset of relation $f(x_i)$ whose product is a perfect square, P_1. The corresponding subset of x_i can be multiplied to form a second product, P_2. Because of the nature of the generating function at left, it can then be shown that:

$$P_1 \equiv P_2^2 \mod N$$

- Problem is, how do we find a subset that is a perfect square?

Modulo 2 Exponent Vectors

- To find a perfect square subset, we use "modulo 2 exponent vectors":
- For each relation, construct a vector of length |{FB}| containing the exponent of each {FB} element in $f(x_i)$, then mod 2.
- For example:

$$\text{if } \{FB\} = \{p_1, p_2, p_3 \dots p_n\}$$
$$\text{and } f\ (x_i) = p_1^0 + p_2^1 + p_3^3 + \dots p_n^2$$
$$\text{then vector } V = [\ 0\ 1\ 3\ \dots\ 2\ ]$$
$$\text{which mod } 2 = [\,0\,1\,1\,\dots\,0\,].$$

- Insert vectors as rows into a matrix and augment the identity
- Then perform Gaussian Elimination (or other matrix solve method) in F_2
- We want to find a combination of exponent vectors that produce a zero vector. Zero vectors correspond to perfect squares because even powers (0 mod 2) of any integer (or product thereof) are trivially just squares of themselves
- RREF of Gaussian Elimination ensures zero rows reside at bottom of matrix
- Linear Algebra also tells us |{FB}|+1 relations are required to guarantee a zero row, we find |{FB}|+10 to guarantee sol.

Quadratic Residues & Legendre

- Critical construct in moving from Dixon's method to the Quadratic Sieve
- By definition, a number N is called a quadratic residue modulo p if...

$$\exists r \text{ such that } N \equiv r^2 \mod p$$

- And if a solution r does exist, it will occur on the interval...

$$0 < r < \frac{p}{2} \text{ for r } \in \mathbb{N}$$

- And a second solution r' = p – r, will also exist. Values r and r' are important for QS

- A construct called the Legendre (shown below) is often used in descriptions of quad. residues

$$\left(\frac{N}{p}\right) = \begin{cases} 0, & \text{if } p \text{ divides } N \text{ , otherwise:} \\ 1, & \text{if } N \text{ is a quadratic residue modulo } p \\ -1, & \text{if } N \text{ is a quadratic non-residue modulo } p \end{cases}$$

- Like the relationship between primality testing and factorization, determining the Legendre symbol is much quicker than actually finding solve values

The Quadratic Sieve Algorithm

- By applying residues to Dixon's method, we have a basic Quadratic Sieve.
- One still selects smoothness bound B, but the {FB} definition is now stricter...

$$\{FB\} = \{p \mid p \text{ is prime} \leq B \text{ and } \left(\frac{N}{p}\right) = 1\}$$

- The function also changes and operates on a defined interval...

$$f(x) = x^2 - N \text{ where } \sqrt{N} < x < \sqrt{2N} \text{ and } x \in \mathbb{N}$$

- The new f(x) does away with inherently costly modulus operator, but the critical congruences of Dixon's method hold since both old and new f(x) are cong.

- But most importantly, the residue solve values for each prime tell for which $f(x_i)$ that prime is a divisor:

If $p_n \in \{FB\}$ and r_n, r'_n are residue sols. then p_n divides $f(x)$ when $x = w * p_n + r_n$ or $x = w * p_n + r'_n$ for $w \in \mathbb{N}$.

- Benefits:
  - FAR less costly divisions
  - Shorter exponent vectors
  - Faster matrix elimination

Large Prime Variations

- Consider an $f(x_i)$ is near {FB}-smooth.
  - Factors except for the cofactor, C≠1.
  - Because C divides a $f(x_i)$, Legendre == 1.
  - Known C > B, else it would be in {FB}
  - If $C < B^2$ then the SoE tells us that C is prime.
  - Any $B < C < B^2$ is a large prime variant (LPV) and the $(x_i, f(x_i))$ is called a partial relation.
  - By combining partial relations that have the same LPV, it is possible to create a relation that is {FB}-smooth (See Below).
  - (One often chooses a LPV bound $< B^2$ and term it B_2, and (old) B becomes B_1)

$N = 2701$ using smoothness bound $B = 13$ produces $\{FB\} = \{2, 3, 5, 13\}$.
This setup will produce the two $f(x_i)$:

$$\begin{aligned} f(62) \Rightarrow 62^2 &\equiv 3^2 * 127 && \mod 2701 \\ f(65) \Rightarrow 65^2 &\equiv 2^2 * 3 * 127 && \mod 2701 \end{aligned}$$

Combining these together, congruence holds across multiplication:

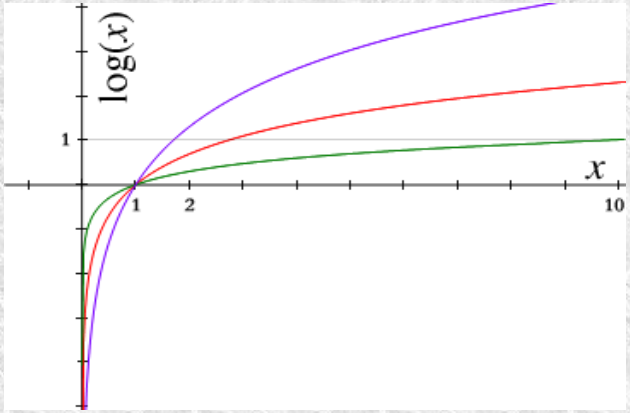
$$(62 * 65)^2 \equiv 2^2 * 3^3 * 127^2 \mod 2701$$

Finally, by multiply both sides of the equation by $(127^{-1})^2 \mod 2701$:

$$\begin{aligned} (1191 * 62 * 65)^2 &\equiv 2^2 * 3^3 * 127^2 * 127^{-2} && \mod 2701 \\ 53^2 &\equiv 2^2 * 3^3 && \mod 2701 \\ 53^2 - 2701 &\equiv 2^2 * 3^3 && \mod 2701 \\ f(53) &\equiv 2^2 * 3^3 && \mod 2701 \end{aligned}$$

Implementation Optimizations

- Block Strategy
  - With Dixon's Method it was possible to consider individual $f(x_i)$, attempt to factor over {FB}, and handle accordingly.
  - Individual consideration is bad for QS. We populate a block with thousands of sequential $f(x_i)$ then "sieve" through it.
  - Each new block comes w/setup overhead



- Logarithmic Estimation of Factorization
  - If Z = z*z' then log(Z)-log(z)-log(z') = 0
  - Log subtraction mimics division
  - Notice that 2 completely factors 32, but log(32) – log (2) ≠ 0.
  - Unable to compensate for multiple occurrences of a factor
  - Therefore "probable" relations must not reduce to 0, but < errorBound
  - Probable relations then undergo brute-force trial division over {FB} to determine the cofactor
  - Comparatively tiny $\log(f(x_i))$ now populates blocks

Optimizations Continued...

- GMP Multiple Precision Integer Library
  - Used to represent integers beyond the scope of hardware primitives.
  - Provides optimized manipulation (math) functions, some even in assembly code
  - Very particular from a memory management perspective

- Constant time block population
  - Log(f(x_i)) populate blocks, but no GMP function exists to calculate it
  - Instead we count the number of bits in f(x_i), a crude integer-accuracy log2 estimation
  - As f(x_i) grow large, there could be millions or more consecutive block elements with the same integer-accuracy logarithm, so why waste time with precise calculation?
  - Just take median block element, calculate its logarithm, and apply to entire block
  - Disaster for initial blocks, but beneficial for later ones; BLOCKSIZE considerations

Sequential Implementation Notes

- Huge amount of work spent on more subtle optimizations: Time/space decisions, type choices, memory management, and even down to operand order in the critical sieving loops.
- Few authors write about the algorithm at an implementation level, so it is hard to evaluate how effective a solution ours is. The many modifications to QS invalidate comparisons.
- At the very least: The implementation uses well-established optimization strategies, and code has undergone many refinements. The well documented source would serve as an excellent foundation for anyone looking to experiment with the Quadratic Sieve

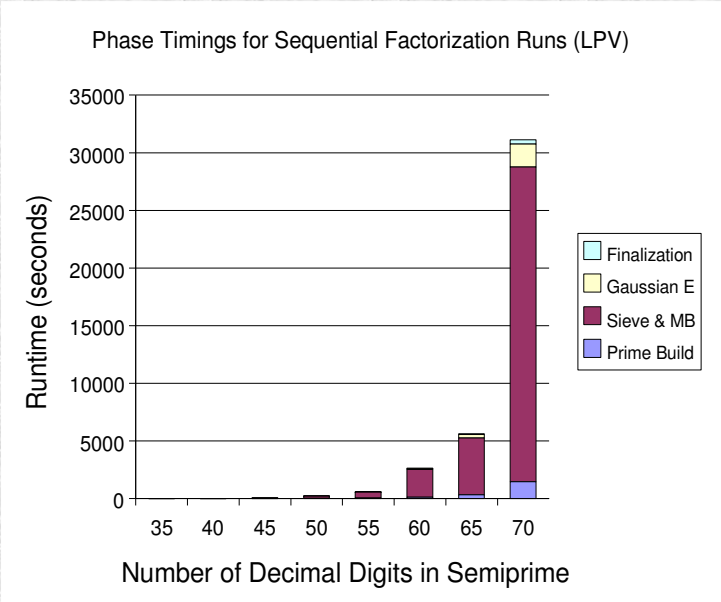
```
476 ulong makePartPB(ulong partPB[], ulong smoothBound, ulon
477   ulong numPartP = 0;
478   ulong curPrime = smoothBound;
479   while(true){
480     curPrime = nextprime(curPrime);
481     while(legendre(n, curPrime) != 1)
482       curPrime = nextprime(curPrime);
483     if(curPrime > partBound)
484       break;
485     else{
486       partPB[numPartP] = curPrime;
487       numPartP++;
488     } // Store only large primes that pass Legendre test
489   } // While curPrime <= partBound
490   return numPartP;
491 }
492
493   // Compute for which array entry in a block a particul
494   // will first appear for a given residue.
495 ulong blkEntry(mpz_t blockStart, ulong prime, ulong resi
496   ulong startPos = -1 *(mpz_fdiv_ui(blockStart, prime)) -
497   if(startPos > prime)
498     startPos -= prime;
499   return startPos;
500 }
501
502   // Verify probable-relations. Return cofactor of trial
503   // indicates num is smooth over smoothPB[], and expRow
504   // Returns ULONG_MAX if cofactor causes ULONG overflow
505 ulong isRel(mpz_t expRow, mpz_t probRel, ulong smoothPB[
506   mpz_t tempNum;
507   mpz_init_set(tempNum, probRel);
508   for(ulong i=0; i < numSmoothP; i++){
509     if((mpz_remove_ui(tempNum, tempNum, smoothPB[i]) % 2
510       mpz_setbit(expRow, i);
511   } // For all primes, attempt removal from num; set vec
512   if(mpz_fits_ulong_p(tempNum) == 0)
513     return ULONG_MAX;
514   ulong returnVal = mpz_get_ui(tempNum);
515   mpz_clear(tempNum);
516   return returnVal;
517 }
518
519   // Return index of a prime in partial (large) prime ba
520   // Needed because partial arrays all have indice corre
521 ulong lookupLP(ulong cofact, ulong partPB[], ulong numPa
```

Matrix Elimination for QS

- Factoring large numbers using QS will create huge matrices. A matrix w/10^{12} or more elements is certainly in the realm of possibility
- Despite being a "basic" elimination method, Gaussian Elim. is surprisingly efficient in F_2
- First, consider that a bit matrix "naturally lends itself to computer implementation."
- Gaussian Elimination uses only elementary row operations, all of which are simplified in F_2 compared to their decimal forms:
  - Swapping: Just swap row pointers
  - Scaling: Not required since the only non-zero array element is 1.
  - Subtraction: Reduces to bitwise XOR between rows
- GE still inappropriate for huge matrices with an $O(n^3)$ complexity
- Wiedmann, Block Lanczos, and Conjugate Gradient algorithms have all been brought to bear on problem.
- Some methods have added benefit of using list representations (beneficial since the matrix is very sparse), with huge memory savings
- Block Lanczos currently most popular, since entire matrix need not reside in physical memory to eliminate.
- Our implementation uses a sequential version of GE, but elimination is not relevant to our research topic.

Sequential Timings and Data

- Now presented are some sample timings and data of interest from sequential runs of the algorithm



# of Digits	Runtime (secs)
35	3.13
40	7.77
45	29.69
50	203.48
55	568.3
60	2627.51
65	5629.19
70	31126.53

```
General Params
Factored Int    : 502245573234263810325/
                  982557836527322147450/
                  377681972219491560182/
                  1957969
Block Size      : 100,000
Bound Ratio     : 40.000000

Base Data
Smooth P-Bound  : 1,107,477
Smooth Primes   : 43,207
Partial P-Bound : 44,299,080
Partial Primes  : 1,295,827

Relation Data
Partials Stored : 87,448
Partial Rels    : 14,234
Smooth Rels     : 28,983
Total Relations : 43,217

Efficiency
Num Blocks      : 3,262,339
Rels per Block  : 0.013247
F(X) per Rel    : 7,548,740
Time per Block  : 0.008365

Timings
PrimeB Build TM : 1,464.0656
Sieve & MB Time : 27,289.628
GaussianE Time  : 1,992.9402
Finalization TM : 379.893164
Runtime (secs)  : 31,126.527

Factors         : 844665220678363226444519532068481735946090367387246105514247184493765 3
```

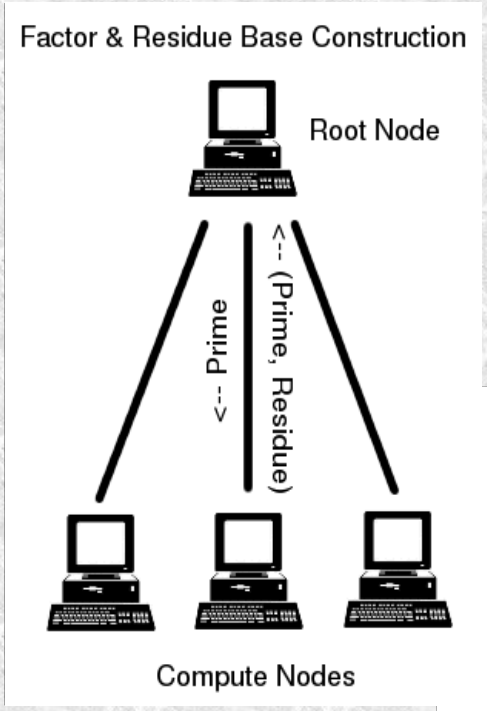
- Exponential graphs are boring to look at, and charts do data more justice, as the above graph shows.
- The runtime output at right demonstrates just how rare relations are among $f(x_i)$.

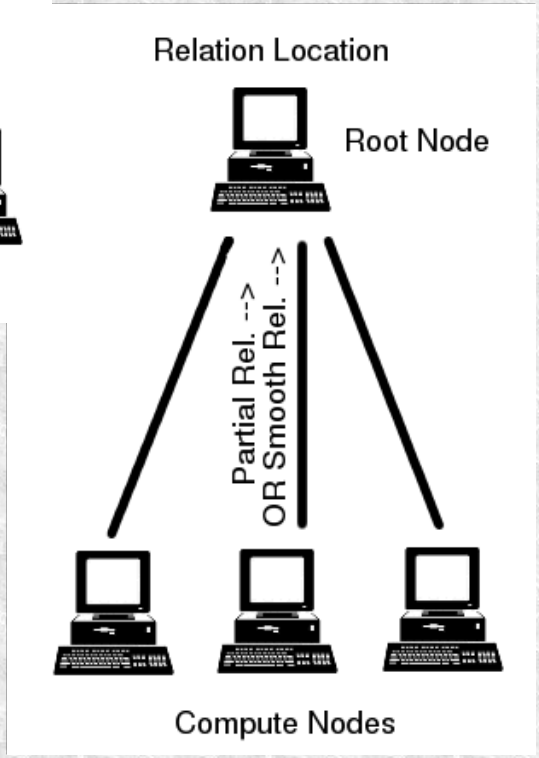
Parallel Algorithm Implementation

- A parallel implementation is necessary to factor numbers of a significant magnitude
- With our research topic in mind, we build a parallel implementation for relation acquisition portion of the algorithm (ignoring elimination).
- A master-slave relationship defines the parallel arrangement, with a root node handling workload distribution and data storage, while compute nodes (as their name suggests) complete the majority of the computational workload.
- The parallel portion of the algorithm can be broke into two main sections:
  - Factor and residue base construction
  - Relation location
- When QS was a hot topic of research in the mid-1980's, shared-memory Cray vector machines were almost always used. Our implementation on a Beowulf styled commodity cluster must take into account the increased cost of network communications.

Parallel Algorithm Notes

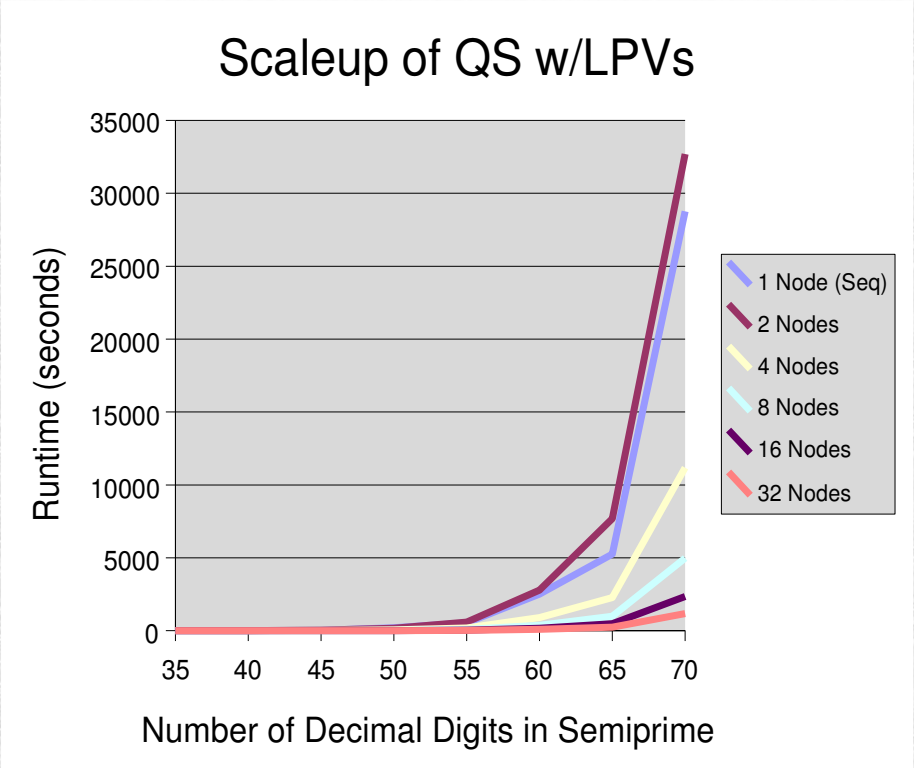
- Factor and Residue Base Construction
  - Advantageous to solve quadratic residues in parallel because of difficulty
  - Root node distributes Legendre-passed primes to compute nodes on as needed basis until B_1 is reached.
- Relation Location
  - Root collects relations until sufficient number are found or all compute nodes reach upper limit on sieving.
  - Compute nodes determine which blocks they are responsible for locally based on # of nodes in VM (quicker than workpool)
  - Root handles storage/combination of partial relations

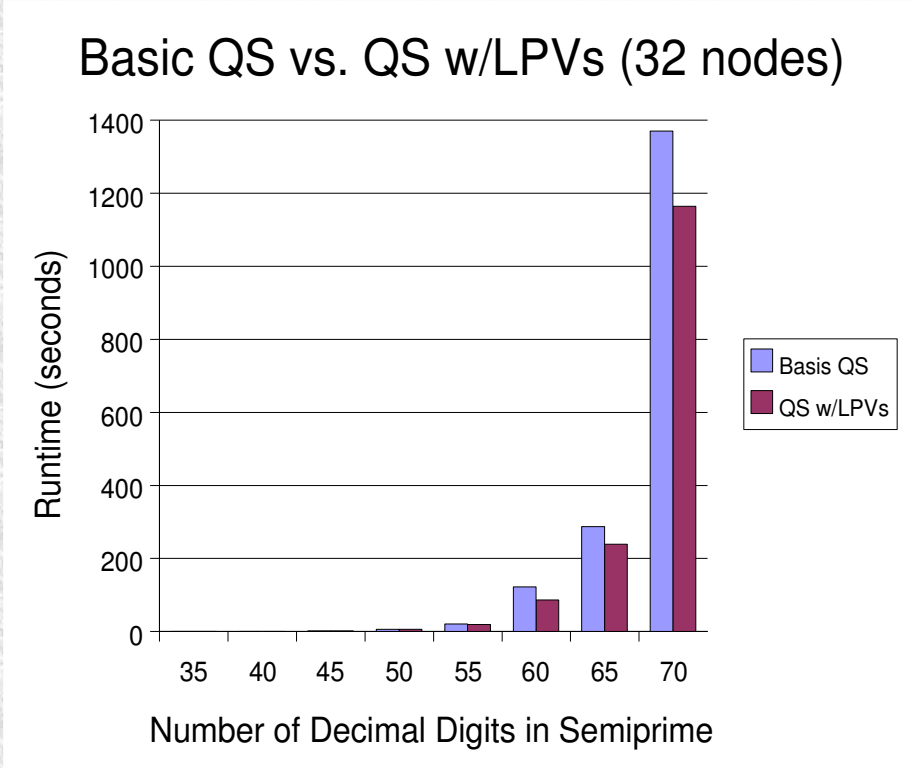




Parallel Speedup and Timings

- The parallel implementation is similar to the sequential one, but incurs the added time of communication costs. Speedup values are very good, and large virtual machines make possible the factorization of large numbers in reasonable time frames.





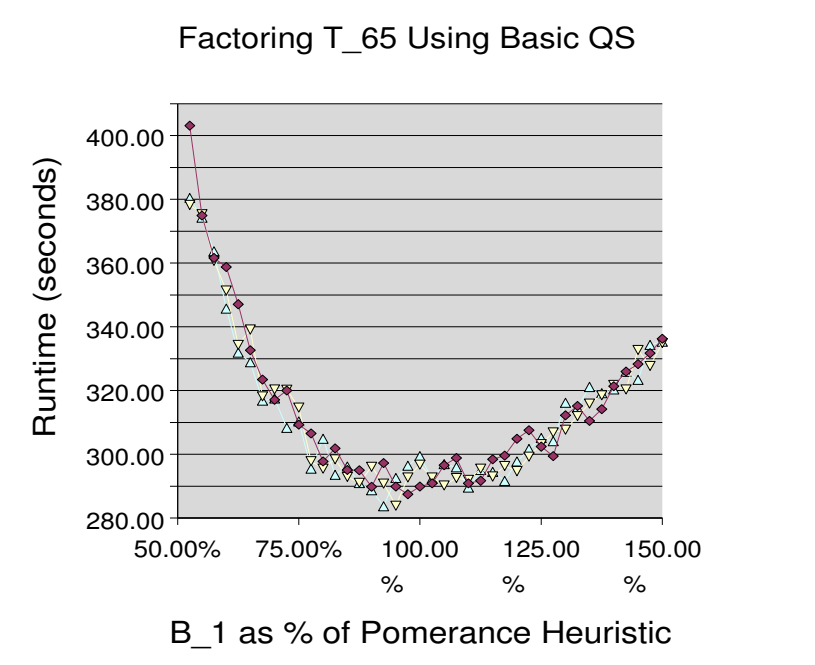
Research Question: Bound Optimization

- **Recall: B_1= The "smoothness bound" serving as upper limit on {FB}**
  B_2= Upper bound on cofactors considered Large Prime Variations (LPVs)

- We want to choose bounds such that relation acquisition runtime is minimized (ignoring the effects they have on matrix elimination).
- Bound selection should be determined primarily by the size of N to be factored.
- Understanding how B_1 and B_2 selection affect algorithm internals is exhausting, so let's just touch on the main points:
  - B_1: Controls exponent vector length and the # of relations to be found. Large B_1 dictates many easy-to-find relations are required, a smaller selection requires a smaller number of hard-to-find relations.
  - B_2: Less dependencies than B_1, but too large a selection clogs comm. w/useless LPVs that will never find a match. Also a huge consumer of memory.

Bound Optimization for Basic QS

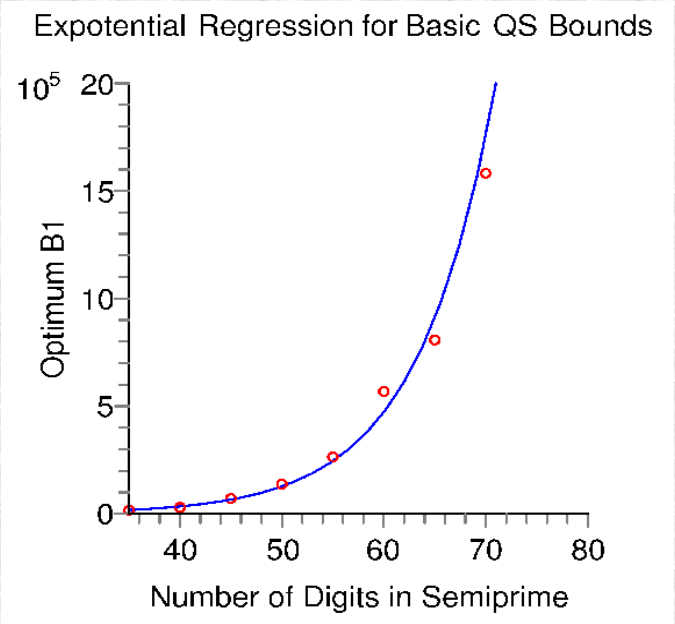
- We first tried to optimize B_1 for basic QS, a version without LPVs (and thus no B_2), in hopes it could guide later analysis.
- Fortunately, the runtimes result in a well-behaved polynomial across varying bounds:



- First guided by theory and then iterative approaches to runtime reduction, we arrive at the following optimum B_1 for our test nums:

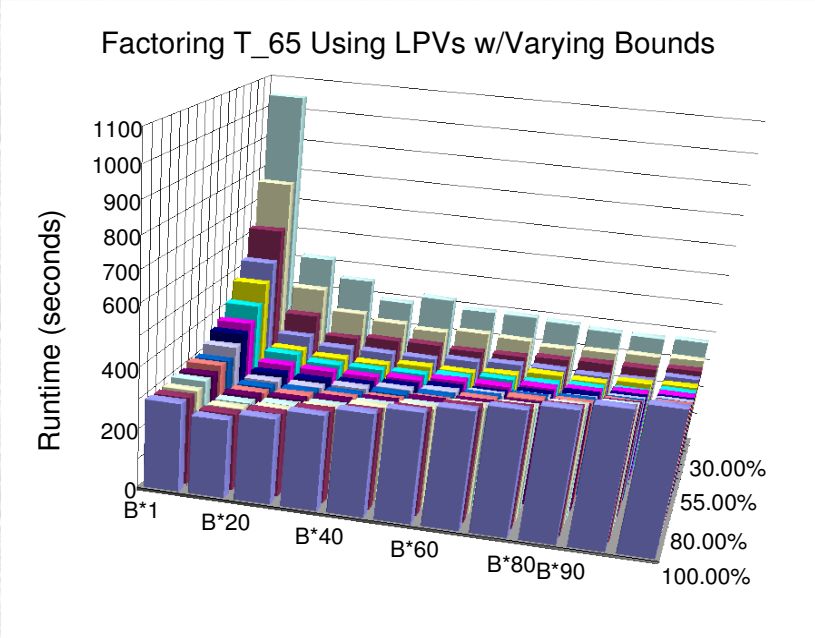
# of Digits	Optimum B_1
35	17,023
40	29,723
45	70,575
50	137,134
55	263,382
60	568,835
65	807,450
70	1,582,110

- These optimum bounds had consistent enough growth to apply an exponential least squares regression:
- Positive extrapolations



Bound Optimization for QS w/LPVs

- Trying to optimize B_1 and B_2 with runtimes results in a problem in 3-Dimensions, which is ugly from an optimization standpoint.



- With a huge number of test runs, we are still able to optimize each number, resulting in the following table:

# Digits	B1	% Basic B1	B2	B2/B1
35	17,023	100	17,023	1
40	29,723	100	29,723	1
45	49,403	70	247,015	5
50	82,280	60	1,645,600	20
55	184,367	70	1,843,670	10
60	312,859	55	3,128,590	10
65	524,843	65	5,248,430	10
70	1,107,477	70	44,299,080	40

- Conclusions on LPV bound optimization:
  - Optimum B_1 seems to consistently be 60%-70% of what basic QS required (good)
  - B_2 selection, however, is all over the place
    - Las Vegas algorithm nature?
    - Inherent in the algorithm?
    - Makes regression finding impossible
  - Attempts at larger factorizations

Questions?