Factoring large integers using parallel Quadratic Sieve

Olof Åsbrink
d95-oas@nada.kth.se

Joel Brynielsson
joel@nada.kth.se

14th April 2000

Abstract

Integer factorization is a well studied topic. Parts of the cryptography we use each day rely on the fact that this problem is difficult. One method one can use for factorizing a large composite number is the Quadratic Sieve algorithm. This method is among the best known today. We present a parallel implementation of the Quadratic Sieve using the Message Passing Interface (MPI). We also discuss the performance of this implementation which shows that this approach is a good one.

1 Introduction

Modern cryptography relies on the fact that integer factorization is a hard problem. It is easy to create large composite numbers by multiplying large primes together but it is very hard to reconstruct the prime factors from a large number.

An example of a system that uses this fact is the famous RSA-system that is widely used today. Development of factorization algorithms is therefore of great interest and research on this is being made all over the world. The algorithms are getting better, but no one has found a very good algorithm. On the other hand, no one has succeeded in proving that a good algorithm does not exist.

2 The Quadratic Sieve algorithm

2.1 Overview

The Quadratic Sieve algorithm for factoring large numbers has several variations. The main idea is to come up with two distinct integers, x and y, such that $x^2 \equiv y^2 \bmod n$ and $x \not\equiv \pm y \bmod n$. If we can do this we know that

$$n|(x^2 - y^2) \iff n|(x+y)(x-y)$$

We also know that n does not divide $x+y$ or $x-y$ so $\gcd(n, x+y)$ and $\gcd(n, x-y)$ must be proper factors of n.

Carl Pomerance came up with a suggestion of how to find the two distinct integers[8]. Take the polynomial

$$f(x) = (x + \lfloor\sqrt{n}\rfloor)^2 - n$$

After studying this polynomial we note that $(x + \lfloor\sqrt{n}\rfloor)^2 \equiv f(x) \bmod n$. We also note that equality in this congruence will never occur. Now suppose we find some integers $x_1, x_2, \ldots, x_k$ such that the product $f(x_1)f(x_2)\ldots f(x_k)$ is a square, say y^2. If we let $x = (x_1 + \lfloor\sqrt{n}\rfloor)(x_2 + \lfloor\sqrt{n}\rfloor)\ldots(x_k + \lfloor\sqrt{n}\rfloor)$ we have a solution to our main equation, $x^2 \equiv y^2 \bmod n$.

Note that we have a 50% chance of receiving trivial solutions (less if n is the product of more than two primes). Therefore we might have to run our algorithm several times.

Now we have to find $x_1, x_2, \ldots, x_k$ such that the discussion above works. We realize that we should only consider x_is such that $f(x_i)$ is not divisible by a large prime. If $f(x_i)$ would be divisible by a large prime, we would need to find some other x_j such that $f(x_j)$ is also divisible by that same large prime. We do not want that and therefore we should work with x_is such that $f(x_i)$ can be completely factored into small primes. The sieving part of the Quadratic Sieve algorithm gives us a technique for finding all the x_is such that the $f(x_i)$s are all products of some of the (small) primes in a special vector. This vector is referred to as the *factor base*.

Now let us assume that the factor base consists of the primes $p_1, p_2, \ldots, p_k$. We then calculate values of $f(x_i)$ such that they completely factor over the factor base (the sieving step). Then we build a matrix where each row of the matrix consists of the exponents of the factorizations of $f(x)$, i.e., if $f(x) = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$ then the corresponding row of the matrix becomes $(a_1, a_2, \ldots, a_k)$. After this we use Gaussian elimination to find the linear combination of the $f(x_i)$s that becomes a perfect square. That means that the column sums for the chosen $f(x_i)$-rows are even.

2.2 Finding the factor base

To be able to use our factorizations in the sieving step we want our $f(x)$s to factor completely over the factor base so that $f(x) = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$. Therefore a prime, p_i, in our factor base divides $f(x)$. We also know that p_i does not divide n (our algorithm requires trial division before start). After looking at the polynomial we are working with we see that this implies the following result:

$$n \equiv (x + \lfloor \sqrt{n} \rfloor)^2 \bmod p_i$$

This means that n must be a quadratic residue modulo p_i and thus our factor base should only consist of primes, p_i, such that n is a quadratic residue modulo p_i.

2.3 Sieving

Values of $f(x_i)$ that can be completely factored by the primes in the factor base will be very rare[9]. Therefore we need to be able to quickly determine whether or not a given value will factor completely. Let p be a prime in the factor base. By our restriction on the primes in the factor base above, n must be a quadratic residue modulo p. This implies that n is the square of one of two quadratic residues modulo p, i.e., $n \equiv t^2 \bmod p$ or $n \equiv (-t)^2 \bmod p$. This also means that $x + \lfloor \sqrt{n} \rfloor$ must be congruent to either t or $-t$ modulo p. We also know that p divides $f(x)$ ($f(x)$ factors over the factor base where p is). Furthermore, if we look at the polynomial $f(x)$, we see that p will also divide $f(x + p), f(x + 2p)$ etc.

This gives us the basis for the sieving operation. Since $f(x)$ is the product of some primes from the factor base, the logarithm of $f(x)$ is the *sum* of the logarithms of these primes.

We consider a bunch of xs in a large block. For each x_i we then compute $z_i = \log f(x_i)$. Then for each prime, p, in the factor base we check if $x_i \equiv t \bmod p$ or $x_i \equiv -t \bmod p$. If that is the case, we subtract $\log p$ from z_i. If $f(x_i)$ factors completely over the factor base, the value of z_i will be theoretically reduced to zero.

However, since we are using computers, we have to take round-off into account. Also, we don't check if a prime could be a factor with multiple power. With this in mind, we understand that the value of z_i will never become zero, so we accept all values that are "close enough" and specify such an interval.

2.4 Gaussian elimination

To guarantee solutions we should make sure our matrix consists of more rows than columns. Each extra row gives us another chance of finding a factor. The number of columns are equal to the number of primes in the factor base so the number of factored $f(x)$s should be the number of primes in the factor base increased with some constant (usually 10).

We want our elimination to produce a linear combination so that each prime power in the combination is even (then the product will become a perfect square). Thus, the Gaussian elimination takes place in the field $GF[2]$ so all calculations can be made modulo 2. Therefore, all operations needed are exclusive or's and row swaps.

3 Parallel algorithm

When considering a parallel implementation of an algorithm one has to consider the time complexity for the different parts in the algorithm. Very often an algorithm has a "heavy" part and a lot of other parts where the time complexity is neglectable compared to this heavy part.

In the Quadratic Sieve algorithm the sieving is the heavy part. This part is ideally suited for parallel implementation. The sieving is performed over blocks with different intervals. These blocks are easily distributed to the different processors. With this kind of implementation the communication between the different processors is kept to a minimum compared to the job that is done by each processor. A master

process collects the results from all the processors and builds the matrix.

Another time consuming part worth mentioning is the Gaussian elimination. However, the time complexity for this is minor compared to the time complexity for the sieving part. Good polynomial time algorithms exist[2]. Also, Gaussian elimination is not very well suited for parallel implementation. Therefore the master node performs the Gaussian elimination.

To summarize, an effective Quadratic Sieve algorithm has a master node that shares the sieving job to the slave processors. When the matrix is full, the master node performs the Gaussian elimination and calculates the result.

An interesting "parallel" version of this idea has been constructed by Lenstra and Manasse[7] who distribute their program and collect the results via electronic mail. They used a slightly different version of the Quadratic Sieve which uses different polynomials. Their idea could have been equally well used for the ordinary algorithm.

4 Implementation

The algorithm was implemented in ANSI C which gives good performance. The message passing was implemented with the Message Passing Interface (MPI).

To represent arbitrary large numbers that are used in the algorithm the GNU Multiple Precision Arithmetic Library (GMP)[4] has been used. GMP contains almost every basic function on large numbers needed in the Quadratic Sieve algorithm. GMP is not available on all architectures, though. This gave us some problems in our analysis.

4.1 Serial

The program has three major phases:

- calculating the factor base,
- sieving,
- Gaussian elimination.

4.1.1 Factor base

The factor base is implemented as an array of factor base items. Each factor base item represents a prime in the factor base and contains the prime itself, the logarithm of the prime and the numbers t and $-t$ (discussed in section 2.3).

The primes are generated using the GMP-function `mpz_probab_prime_p` used for primality testing.

4.1.2 Sieving

The sieving process wants to find large numbers $x_i = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_n}$ that factorizes completely into primes from the factor base. The numbers, x_i, are stored in an array of type `mpz_t` (this is the datatype for large integers in GMP) and the factor exponents of the primes are stored in an exponent matrix.

The object of the sieving process is to find numbers that are likely to factor completely over the factor base and factor those numbers using a naive trial division algorithm.

Numbers are sieved in blocks, B. First an array, *sumlog*, of the same size as the sieved block, B, is initialized with $log(x_i)$ ($\forall x_i \in B$). Then the program loop over the primes, p_j, in the factor base and for each number x_i that is congruent to t or $-t$ we subtract $log(p_j)$ from $sumlog[x_i]$. Then trial division is performed on every x_i where $|sumlog[x_i]|$ is less than a threshold value. This threshold value is often chosen to be the logarithm of the largest prime in the factor base, and we have applied this view in our implementation. All logarithm operations are performed with base 2 and only with integer precision. GMP does not contain a log_2 operation so $log_2(x_i)$ is approximated with the size of x_i in bits, i.e., the length of the number in binary representation, divided by 2.

4.1.3 Gaussian elimination

Gaussian elimination is applied once on a large matrix in $GF[2]$, i.e., the matrix consists only of ones and zeros. The amount of memory needed to factor a number using a factor base of size 4096 is approximately 200 Megabyte. One matrix is $4096 \times 4096 \times 4$ byte (integer size) ≈ 64 Megabyte and there are three matrices:

- the exponent matrix,
- the bit matrix,
- an identity matrix used during the elimination.

Both the bit matrix and the identity matrix contains only ones and zeros so representing a bit

with an integer would increase the memory usage by a factor of 32 (bit size of an integer). Therefore integers are used to represent a group of 32 bits in a row in the matrices. Memory access of a single bit requires more operations than accessing a single integer, but exclusive or on 32 bits can be performed in one operation. The size of the matrices with this representation is approximately 4 Megabyte each and the total memory needed is therefore approximately 72 Megabyte.

4.2 Parallel

The parallelism is achieved using the MPI-package for message passing. The program starts with splitting up into multiple processes where each node gets one process (`MPI_Init` is called to set up the MPI environment). All communication is performed in the default communicator called `MPI_COMM_WORLD`, which contains the set of all processes. Each process is assigned a rank and the process with rank equal to zero is assigned the role of the master node. The other processes become slave nodes. Each slave node communicates with the master node only (see figure 1).

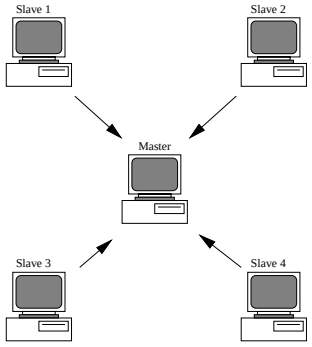

Figure 1: The slave nodes all communicate with the master node. The master node collects the results and calculates an answer.

The factor base is needed by every node and the time to compute it is minor compared to the total execution time. Since no node can do any work before it receives the factor base, there is no difference in performance to let every node compute the factor base by itself. The factor base computation is implemented in the same way as in the serial implementation.

The Gaussian elimination is performed by the master node only. Therefore the Gaussian elimination is implemented in the same way as in the serial version.

4.2.1 Parallel sieving

When the program reaches the sieving phase the master node enters a receive loop and the slave nodes enter a compute-send loop.

Each slave node sieves over blocks, B, as described above. Which block to sieve is determined by the rank of the node, i.e., if there are 16 slave nodes, each node sieves over every 16th block of x_is. When a preset number of x_is – such that $f(x_i)$ factors completely over the factor base – is found then a block of $f(x_i)$s and their factorizations, called a `sendblock`, is sent to the master node. The send is a synchronized blocking send using `MPI_Send`.

The `sendblock` consists of $f(x_i)$s as `mpz_t`s, the exponent matrix and the corresponding bit matrix. The `sendblock` is sent to the master node using three consecutive `MPI_Send`s. A `MPI_Datatype` called `mpi_mpz_t` is defined for the `mpz_t` datatype as a string of characters. The $f(x_i)$s are sent as a contiguous array of `mpi_mpz_t` and the matrices are sent as contiguous arrays of `MPI_INT`.

In the receive loop the master node is calling `MPI_Recv` until it receives more $f(x_i)$s than the size of the factor base. The master node receives a complete `sendblock` from a slave node before receiving another sendblock from another slave node. To receive a complete block three `MPI_Recv`s are called. The first message is received (which contains the `mpi_mpz_t`s) from any slave node. The second and third receive is exclusively received from the slave node that sent the first message. The first receive can come from any slave node since the order of the $f(x_i)$s that are received by the master node is of no importance.

After a complete block is received, the master node acknowledge this by sending a confirmation to the slave node. However, if the master node has received enough $f(x_i)$s to continue with Gaussian elimination, it tells the slave node to terminate. The other slave nodes that are in the middle of sieving when the master node starts with Gaussian elimination will finish sieving their current `sendblock`. Then they will have to wait until the master node is finished with the Gaussian elimination to get an acknowledgment to terminate.

4.3 Reservations

In the current state of the implementation the program does not factor a number n completely. It finds any two proper factors f_1 and f_2, not necessarily primes, for which $f_1 \cdot f_2 = n$ holds. We are generally only interested in numbers that are composite of two primes more or less of equal size. For numbers that consist of many smaller primes, other algorithms like Pollard-ρ or trial division are more suitable.

Not much time has been spent on optimizing and no claims are made that this implementation is optimal.

4.4 Improvements

The sieving step in the algorithm is the major part of the running time. Therefore the improvements should be made in this area. Since the master process is idle most of the time one easy way to get more performance would be to have a slave process running on the same processor as the master process resides on.

Currently communication between a slave process and the master process uses Synchronized blocking send and receive (`MPI_Send` and `MPI_Recv`) so if the master process is occupied with another slave process a slave node could be blocked and precious CPU time wasted.

Maybe the master and slave approach is not the best. MPI offers functions for virtual topologies and group communication like `MPI_Gather` and `MPI_Scather`. A solution using these functions could perhaps lead to a more efficient program. An argument against that is that Gaussian elimination is hard to parallelize so in the end one processor must gather all data and perform the Gaussian elimination.

Other parts of the program could also be improved. The method to generate primes in the factor base is maybe not the best one. Using some other method like the Sieve of Eratosthenes may turn out to be better.

5 Performance evaluation

5.1 Test environment

The program is compiled using the mpich compiler. All test runs are made on Sun™ Ultra5, 270 MHz model with 128Mb RAM, workstations if not stated otherwise.

5.2 Test input

For performance evaluation several numbers of different size are factored with different numbers of slave nodes. All numbers in the test runs are composed of two primes of similar size (see Table 1).

5.2.1 Size of the factor base

The size, m, of the factor base is crucial to the execution time. A small size implies that only a few $f(x_i)$s need to be found, but these will be very rare and time consuming to find. A larger factor base will make it easier to find $f(x_i)$s but more of them are needed. Heuristics suggest[9] that $m = L(n)^{1/2}$, where $L(n)$ is the running time. According to [9] the running time is given by the formula $L(n) = e^{(1+o(1))\sqrt{(\log n \log\log n)}}$.

Figure 2 illustrates the execution time of T_{40} factored with different factor base sizes, m.

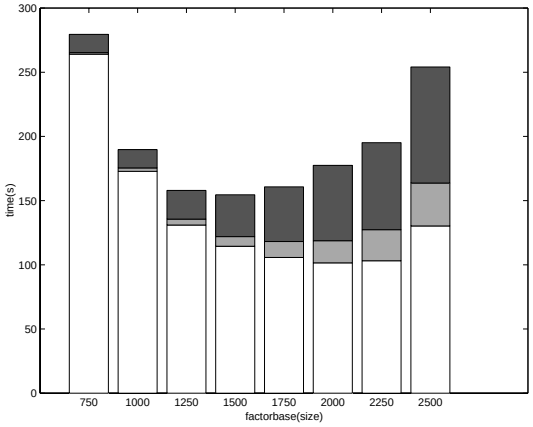

Figure 2: Execution times with different factor base sizes. White is sieve time, dark grey is Gaussian elimination time and light grey is other computation time.

Table 2 shows the sizes of the factor base used in the test runs.

5.3 Program profile

To evaluate how much time is spent in the sieve part and in the Gaussian elimination part, several of the numbers were factored using a profiled version of the serial program. Table 3 shows that, for the number considered, the complexity of the sieve part grows faster than the

name	number
T_{20}	$18567078082619935259 =$ $28540307599 \cdot 650556341$
T_{30}	$350243405507562291174415825999 =$ $4634293795844903 \cdot 75576435361433$
T_{40}	$57059795506186704463085788585542675373983 =$ $78492223909528900351 \cdot 72694838627523822433$
T_{45}	$732197471686198597184965476425281169401188191 =$ $46116492876183969306047 \cdot 1587712824676502602915 3$
T_{50}	$53468946676763197941455249471721044636943883361749 =$ $89496215866082509910478 37 \cdot 59744365903438144501259 77$
T_{55}	$59453265815375131570386363169672578543223939 3895035230547 =$ $78948694531305070581222996 79 \cdot 75306205084620456791220769 3$
T_{60}	$67629227571655824650260523089719136646955176409218136277975 9 =$ $21882269935787119823829190355856 11 \cdot 30905947038452506088894666 9$

Table 1: Numbers to factorize in test runs.

number	factor base size
T_{20}	100
T_{30}	200
T_{40}	1000
T_{45}	3000
T_{50}	3800
T_{55}	5000
T_{60}	6000

Table 2: Numbers to factorize in test runs.

complexity of the Gaussian elimination[1].

number	sieve	gauss. elim.
T_{40}	93.8	1.5
T_{45}	81.8	11.8
T_{50}	93.2	5.1
T_{55}	95.5	4.2

Table 3: Time spent [%]

For larger numbers the Gaussian elimination will grow faster, but it will be infeasible to factor numbers so large using the Quadratic sieve method anyway.

5.4 Performance analysis

The problems were factored using 2, 4, 8 or 16 slave nodes or using the serial algorithm. T_{60} was only factored using 16 slave nodes. T_{20}, T_{30} and T_{40} were factored with a maximum of 2, 4 and 8 slave nodes respectively. Using more slave nodes resulted in longer execution time due to longer initialization time of the MPI-environment.

5.4.1 Execution time

Figure 3 shows the total execution time in minutes of the program factoring numbers up to T_{55}. The execution time for T_{60} using 16 slave nodes was nearly 400 minutes.

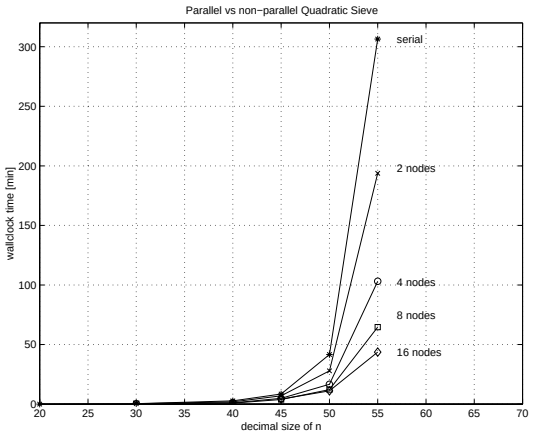

Figure 3: Total execution time.

Figure 4 shows the execution times of the sieve part only. The minor differences in execution time between Figure 3 and Figure 4 shows

[1]The values in Table 3 are looking a bit strange due to non optimal factor base sizes in Table 2.

that the sieving phase is a major part of the program.

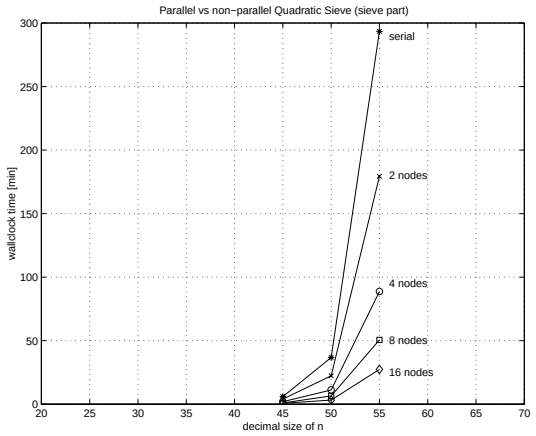

Figure 4: Sieve execution time.

5.4.2 Speed-up

We let f denote the fraction of the program that cannot be computed in parallel. *Amdahl's law* gives the ideal speed-up, S_p:

$$S_p = \frac{T_s}{T_p} = \frac{T_s}{fT_s + \frac{(1-f)T_s}{p}}$$

where T_s denotes the best sequential time for the best sequential program, T_p is the parallel running time and p is the number of processors. Figure 5 and Figure 6 shows the speed-up for T_{55} with different numbers of processors.

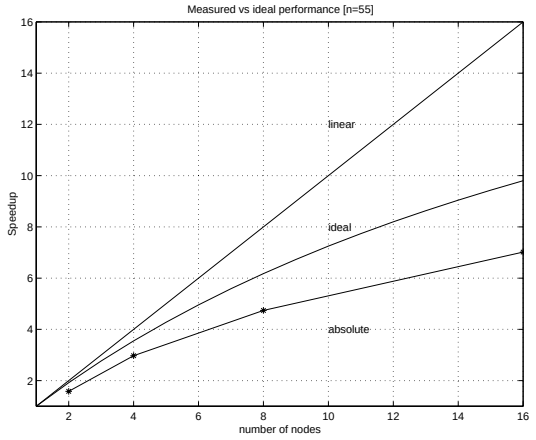

Figure 5: Total speedup.

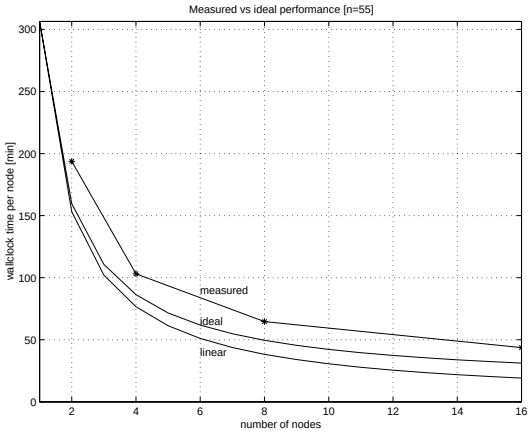

Figure 6: Measured time vs ideal time.

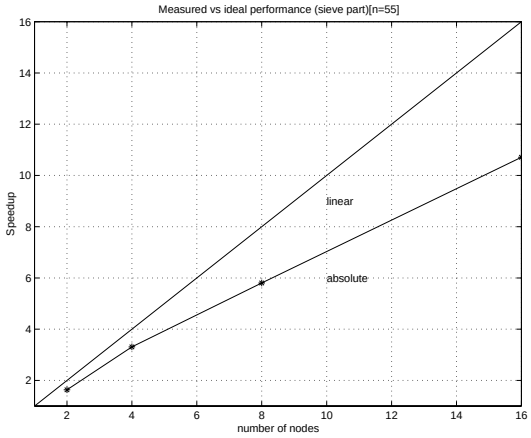

Figure 7: Sieve speedup.

5.4.3 Efficiency

The efficiency, η_p, of a p-node computation with speed-up S_p is given by:

$$\eta_p = \frac{S_p}{p}$$

Figure 9 shows the efficiency when factoring T_{55} with different numbers of processors.

5.4.4 Sources of inefficiency

One source of inefficiency is communication overhead. The number of messages sent is $O\left(\frac{|factorbase|}{|sendblock|}\right)$. The average rate of messages is so low that the master node is idle most of the time. Therefore, the communication overhead is neglectable.

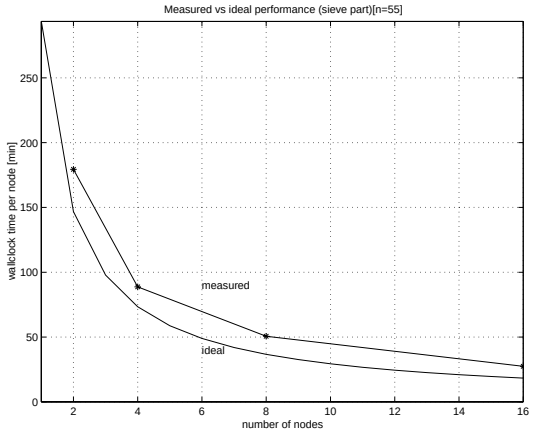

Figure 8: Measured sieve time vs ideal sieve time.

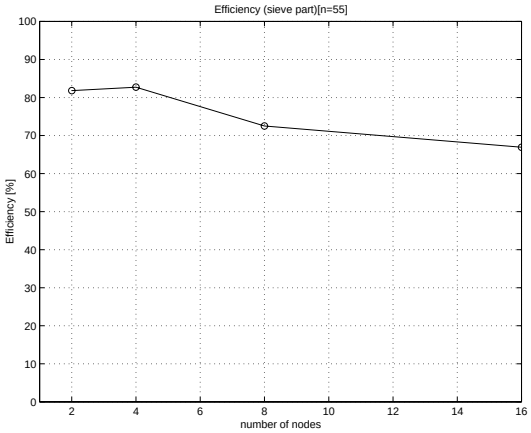

Figure 10: Sieve efficiency

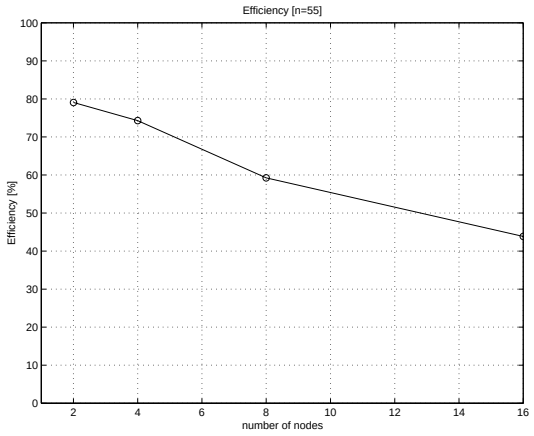

Figure 9: Total efficiency

Another source of inefficiency is load imbalance. All the slave nodes work independently so the nodes have equal work load. If the master node is included in the efficiency computation it lowers the efficiency notably since the master node is idle most of the time. Since the master node is idle most of the time it is possible to run it on the same processor as one of the slave nodes without major performance loss. As an example, the sieve phase of factoring T_{40} with a factor base size of 1019 and a sendblock size of 8 was completed in 66.8 seconds with the master node sharing a processor with a slave node, compared to 64.3 seconds when the master node used a processor for itself.

6 Conclusions

We have verified that the Quadratic Sieve algorithm is a very powerful algorithm for factoring large composite numbers. Together with other methods, like trial division and Pollard-ρ, it can be used for factoring any type of numbers with a very good result.

The tests that we have made on the parallel implementation shows that the algorithm is very well suited for parallel use. Theoretically the Gaussian elimination will be a problem when you want to factor really large numbers. With the resources available today this is not a problem, though.

Some parts of modern cryptography works with really big numbers that one wants to factor. This implies the use of libraries that can handle large numbers, like GMP[4]. When developing our algorithm here at the Royal Institute of Technology we have been able to use a lot of different computer resources. We soon discovered that it is not very common that a library that can handle big numbers is installed. If you want to use for example GMP you have to install it yourself. We think that this problem will be solved and more standardized in the future.

7 Acknowledgments

We got all the help we needed from our teachers here at the Royal Institute of Technology, Sweden. We wish to thank Johan Håstad at the theoretical computer science group at the De-

partment of Numerical analysis and computer science (Nada) for helping us to come up with an interesting problem and for the joy of letting us join his courses once again to beat all other participants with our implementation. We also wish to thank Mike Hammill at the Center for Parallel Computers (PDC) for answering questions on the Message Passing Interface (MPI).

References

[1] Steve Beattie. A Java Implementation of the Quadratic Sieve. Technical report, Oregon Graduate Institute, 1997.

[2] Gerd Eriksson. Numeriska algoritmer med *MATLAB*. Technical report, Royal Institute of Technology, Stockholm, June 1998.

[3] Joseph L. Gerver. Factoring large numbers with a quadratic sieve. *Mathematics of Computation*, 41(163):287–294, July 1983.

[4] Torbjörn Granlund. *The GNU Multiple Precision Arithmetic Library*. TMG Datakonsult, Boston, MA, USA, 2.0.2 edition, June 1996.

[5] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.

[6] Johan Håstad. Notes for the course advanced algorithms. Technical report, Royal Institute of Technology, Stockholm, 1998.

[7] Arjen K. Lenstra and Mark S. Manasse. Factoring by electronic mail. In J. J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology—EUROCRYPT 89*, volume 434 of *Lecture Notes in Computer Science*, pages 355–371. Springer-Verlag, 1990, 10–13 April 1989.

[8] Carl Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and I. Ingemarsson, editors, *Advances in Cryptology: Proceedings of EUROCRYPT 84*, volume 209 of *Lecture Notes in Computer Science*, pages 169–182. Springer-Verlag, 1985, 9–11 April 1984.

[9] Carl Pomerance. Factoring. In Carl Pomerance, editor, *Cryptology and Computational Number Theory*, volume 42 of *Proceedings of Symposia in applied Mathematics*, pages 27–47. American Mathematical Society, 1990.

[10] Douglas R. Stinson. *Cryptography Theory and Practice*. CRC Press, Boca Raton, 1995.