An efficient Parallel Implementation of Self Initialization Quadratic Sieve for Integer Factorizations Using Message Passing Interface (MPI)

Kazem Jahanbakhsh
Sharif University
jahanbakhsh@mehr.sharif.edu

Jason Papadopoulos
University of Maryland College Park
jasonp@boo.net

Abstract: *Integer factorization is one of the most important topics in the complex computation domain. With the advent of public key cryptosystems it is also of practical importance. One method used to factor large composite numbers is the Multiple Polynomial Quadratic Sieve. This paper proposes an optimized serial implementation of one of the best variants of the Multiple Polynomial Quadratic Sieve and then presents an efficient parallel version of this serial implementation using the Message Passing Interface (MPI). We also discuss the performance of this implementation.*

Keywords: Integer factorization, RSA Cryptosystem, Self Initializating Quadratic Sieve algorithm, Cluster, Message Passing Interface.

1 Introduction

The security of modern cryptosystems such as Rivest-Shamir-Adelman (RSA) depends on the difficulty of factoring the public keys. The Multiple Polynomial Quadratic Sieve (MPQS) algorithm is currently the fastest algorithm capable of factoring integers around 100 digits in size, and is the second fastest for integers (and RSA public keys) larger than this.
The complexity of MPQS depends mainly on the size of N, the number to be factored. Under plausible assumptions the MPQS algorithm has an expected run time of $O\left(\exp\left(\sqrt{c \ln N \ln\ln N}\right)\right)$ [1], where c is a constant (depending on the details of the algorithm. For MPQS, $c \approx 1$. For RSA moduli the public key N is of the form $N = p \times q$ where p and q equivalent size prime numbers. For this reason the MPQS factorization algorithm is very well suited for testing the security of RSA cryptosystem. In this paper we are interested, particularly, in the parallelization of some variants of MPQS using MPI on a cluster of workstations. Therefore in the first phase we implement an optimized serial version and then in the second phase we focus on the parallel implementation of the serial algorithm and try to improve the parallelized solution as much as possible. Benchmarks of the implementation are presented in the conclusion.

2 The Self Initializing Quadratic Sieve algorithm

2.1 Overview

This section briefly describes the Self Initializing Quadratic Sieve (SIQS) algorithm. Details regarding this algorithm can be found in [2]. The basic idea is to find a random relation in the form of equation (1).

$$X^2 \equiv Y^2 \bmod N \tag{1}$$

If $X \not\equiv \pm Y \bmod N$, then the greatest common divisor (gcd) of $X - Y$ and N will be a proper factor of N with high probability. To find X, Y that satisfy equation (1), we first find several relations in the form

$$u_i^2 \equiv v_i \bmod N \tag{2}$$

In equation (2), v_i factors into small primes (called the factor base) and $u_i^2 \neq v_i$. Such relations will be called smooth.
When enough relations were generated, then one can use some collection of the smooth relations to construct a relation of the form (1). Finding which collection of relations to use is a linear algebra problem. In QS algorithm the numbers v_i are the values of one polynomial with integer coefficient as below

$$g(x) = (x+b)^2 - N \tag{3}$$

In above polynomial, b is chosen to be $\lceil N \rceil$. This makes it easy to factorize the v_i by sieving. For

details of the process, we refer to the some references [2, 3].
The problem with using only one polynomial is that the values of $g(x)$ increase in size as x gets bigger. Thus as the algorithm progresses, smooth relations becomes less frequent. To escape this problem, we want to be able to change to a new polynomial when the residues of the current polynomial become too big.
We can attain an improvement in speed by replacing the single polynomial $g(x)$ in basic QS with a succession of different polynomials. This variation is known as the MPQS algorithm. MPQS sieves polynomials of the form

$$g_{a,b}(x) = (ax+b)^2 - N = a^2x^2 + 2abx + b^2 - N \qquad (4)$$

where a,b are integers and the sieve values *x* range over a fixed interval. However the polynomial switching cost in MPQS is expensive. To overcome this unwanted cost, Pomerance introduced the SIQS method, which involves a fast way to change polynomials. Making the polynomial switching cheaper allows smaller sieve intervals, which increases the probability that sieve values will be smooth. The following sections paraphrase Contini's description [2].

2.2 Initialization stage for all polynomials

SIQS algorithm uses an intelligent mechanism for polynomials initialization at once. For this we find primes $q_1,...,q_s$ in the factor base whose product is $\approx \frac{\sqrt{2N}}{M}$. This value allows the minimum and maximum values of $g_{a,b}(x)$ to have equal magnitude. Let $a = \prod_{l=1}^{s} q_l$. For this 'a' value, there are 2^{s-1} different 'b' values and hence 2^{s-1} independent polynomial with equation (4).
When the sieving with polynomial i finishes, we can initialize polynomial $i+1$ where $1 \le i \le 2^{s-1}-1$. In fact the 'b' coefficient of polynomial i + 1 can be calculated incrementally from the 'b' coefficient of polynomial *i*. Also the roots of every prime p in the factor base (that does not divide a) for polynomial i + 1 ($root_p^i$) can be computed from the previous roots of polynomial *i*.
The idea of generation different 'a' coefficients can be used for parallelizing the factorization process.

2.3 Sieve stage

The most time consuming part of SIQS algorithm is sieving stage where we update a very large sieve array of length $2M+1$ in unpredictable manner. For each odd prime p in the factor base, we update the locations with index of $\mathrm{root}_p + ip$ for all integers i that falls into sieve interval. Notice that the sieving is a very cheap operation in comparison with trial division [3].

2.4 Serial optimization approach

The critical part of SIQS that is sieving is represented schematically in Figure 1.

```
/* Routine for conventional sieving. This is a sophisticated way
of computing the following: */
for (each prime in factor base) {
        for (each of the 2 roots for that prime) {
                start_offset = root;
                while (start_offset < sieve_size) {
                        sieve_array[start_offset] -=
                                log2(prime);
                        start_offset += prime;
                }
        }
}
```

Figure 1: Conventional sieving pseudo code

As you can see from above pseudo code the sieving step requires an enormous number of memory updates; however, the updates usually cause cache misses. A data layout rearrangement technique is the most practical way to improve cache locality. To utilize this technique we use cache blocking mechanism to fit the working set in cache for the duration of the sieving [4]. We also use a hashtable-based method for processing most of the factor base primes [5].
We added large prime variation support to SIQS algorithm. With this variation we could achieve a speed-up of approximately a factor 2.5 or more [8].
The linear algebra stage uses the "Block Lanczos" method to iteratively solve the resulting linear system [6].

3 Parallel Algorithm

From profiling data we conclude that the sieving phase is the major portion of the SIQS algorithm. From Amdahl's law any parallelization efforts must be focused on this phase to increase efficiency.
The implementation was based on a SPMD (Simple Program Multiple Data) approach and on the following procedures:

1- In the partitioning phase of parallel algorithm design, our focus is on defining a large number of small tasks in order to yield what is termed a “fine-grained” decomposition of a problem. We use the domain decomposition approach to problem partitioning. When factoring a large input, the number of polynomials $g_{a,b}(x)$ to be sieved is very large. Therefore we can assume every polynomial is an independent task.
2- The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so as to allow computation to proceed. In the partitioning phase different tasks require data transferred to each other to achieve concurrency. These parameters are the number N to be factored, the sieve length and other startup data in addition to some other parameters for determination of polynomial coefficients. In the final phase after every node performed its assigned sieving job with its polynomials the saved relations must be collected for the Master node to complete the factorization process.
3- The agglomeration and mapping phase tend to become a single phase in SPMD implementation (as the mapping becomes implicit to agglomeration). In this phase the goal is to agglomerate small tasks to reduce the communication costs by increasing locality. Also we specify where each task is to execute. For these reasons it is better to start every node’s task with a separate $'a'$ coefficient, then generate the related polynomials for this $'a'$ and do the sieving completely independent of other nodes. So by using this new proposed technique we can reduce the communication costs and increase the efficiency of the presented parallel algorithm significantly.

3.1 Domain Decomposition technique with $'a'$ coefficients

At the beginning, all we know from the SIQS algorithm is the target value for $'a'$ coefficient ($\text{target\_a} = \frac{\sqrt{2N}}{M}$) to which the product of all factors must multiply. All of the factors chosen should be about the same size, but we don't know how many should be chosen and no duplication with future polynomials is allowed. Obviously, if we choose an identical set of factors later we'll get duplicate relations. Less obviously, if we choose a set of factors later that is identical except for one or two primes, we will also probably get duplicate relations. Finally, the closer the product of the factors is to ’target_a’, the higher the yield of the resulting polynomials. Finally, $'a'$ values should have at least 3 factors; this will allow the $'a'$ value to generate at least $2^{(3-1)}$ $'b'$ values, and is more efficient. These turn out to be stringent constraints. In addition, in our cluster that we have several machines sieving, each machine has to be able to build its own $'a'$ values by itself, without having to communicate its choices to the other machines. The polynomial initialization routine begins the process by figuring out how many factors should go into an $'a'$ value, along with their size. It works by comparing the log of ‘target_a’ to the sum of logs of the factor base primes. One loop in this routine basically goes through the available sizes of factor base primes to make the “(bits in ‘target_a’) - (bits in sum of the chosen factors)” value zero.

We want to favor the larger primes as the factors of the $'a'$ value, because there are more of them, and also because they affect sieving less (the factors of $'a'$ do not contribute to the sieving phase, so if the factors are too small they could reduce the yield of relations because small primes matter more for sieving).

When we determined the number of factors and the size of each factor of future $'a'$ values, the next base polynomial is constructed. In this stage we find the k factors of each $'a'$ value. The first k-1 of them are determined randomly from the collection of factor base primes. The last factor is chosen to get the product of all the factors as close as possible to the exact value of ‘target_a’, and so is not selected randomly. No attempt is made to verify that the collection of factors chosen is unique; that would require saving all of the previous 'a' values. The code does verify that a given factor only occurs once in a particular $'a'$ value; if that was not the case then 'b' values would be generated incorrectly. For factorizations above a certain small size (~25 digits), there are so many possible factor base primes to choose from that it's very unlikely that a duplicate $'a'$ value will be chosen randomly. For factorizations below 25 digits, we only need one or two $'a'$ values to get many more relations than we need.

3.2 Parallel algorithm implementation

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. For load balancing the Master/Slave mechanism was used. In this configuration the master node specifies the jobs of slave nodes dynamically and each slave node communicates with the master node only (see Figure 2).

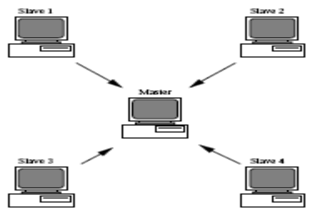

Figure 2: The slave nodes all communicate with the the master node. The master node collects the results and calculates an answer.

First the master node broadcasts 'N' the number to be factored to every slave node. All communication is performed in the default communicator called MPI_COMM_WORLD, which contains the set of all processes. The slave nodes calculate initialization data by themselves because before doing this part there is not any work to do. Then slaves do all the sieving process for generation of specified number of full relations. This value is a factor of $\frac{\max\_relations}{p}$ where $\max\_relations$ is the total number of needed smooth relations and p is the number of nodes (running processes) in the Cluster. Notify that slaves ignore the single/double large prime variation effect. In fact the cycle tracking routine not execute in slave nodes to find smooth relations from the partials [8]. After this, the slaves send the gathered relations with the corresponding $'a'$ factors to the master node block by block. The master node collects the received relation together and run the cycle tracking routine on them. When enough relations were collected by master, it sends termination signal to slaves to stop the sieving process totally.

This dynamic load distribution strategy by MPI concluded to better performance results and saves our cluster resources. Finally the Master does the linear algebra stage of the factorization and finishes the factorization process.

4. Performance Evaluation

4.1 Parallel Cluster Environment

The adoption of clusters, collections of PCs connected by a local network, has virtually exploded since the introduction of the first Beowulf cluster in 1994. The attraction lies in the (potentially) low cost of both hardware and software and the control that builders/users have over their system.

The configuration considered in the article is represented by seventeen loosely-coupled personal computers connected by a 10/100 Mbps fast ethernet switch. The switch has an equivalent function of the interconnection network on a parallel machine. Table 1 shows some characteristics of cluster environment.

Table 1: Cluster environment characteristics

Processor Architecture	Intel Pentium 4
Processor clock rate	2.4 GHz
Layer 2 cache size	512 KB
Memory size	256 Mbytes
Switch throughput	10-100 Mbps
Operating System	Linux kernel 2.4.20-8

Parallel software environments are designed to enhance the execution of concurrent tasks, achieving reasonable parallel speedup. The parallel programming environment used in this work is based on MPI standard [7]. The particular implementation used is mpich 1.2.5.

4.2 Test input

For performance evaluation several numbers of different sizes are factored with different numbers of slave nodes. All numbers in the test runs are composed of two primes of similar size (see Table 2).

4.3 Performance analysis

Execution time

The problems were factored using 2, 4, 8 or 16 slave nodes or using the serial algorithm. Using more

slave nodes resulted in smaller execution time due to parallel algorithm scalability.
Figure 3 shows the total execution time in minutes of the program factoring numbers up to 85 digits.

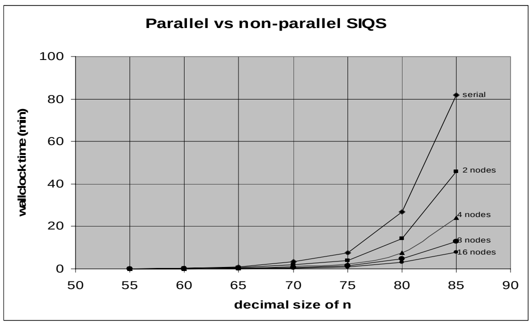

Figure 3: Total execution time

Figure 4 shows the execution times of the sieving part only. The minor difference in execution time between Figure 3 and Figure 4 shows that the sieving phase is the major part of the program.

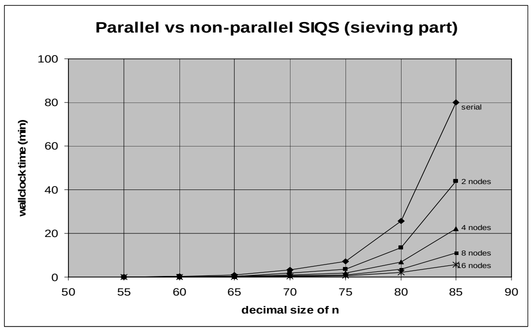

Figure 4: Sieving execution time

Speedup
We let f denotes the fraction of the program that cannot be computed in parallel. Amdahl's law gives the ideal speedup, S_p :

$$S_p = \frac{T_s}{T_p} = \frac{T_s}{fT_s + \frac{(1-f)T_s}{p}} \qquad (5)$$

In equation (5), 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 speedup for 85 digits number with different numbers of processors.

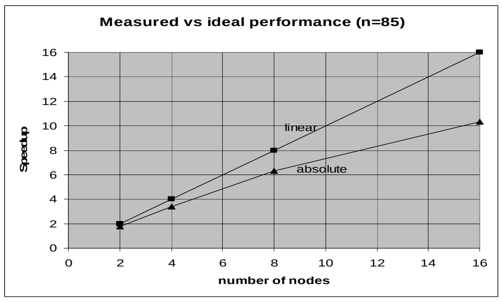

Figure 5: Total speedup

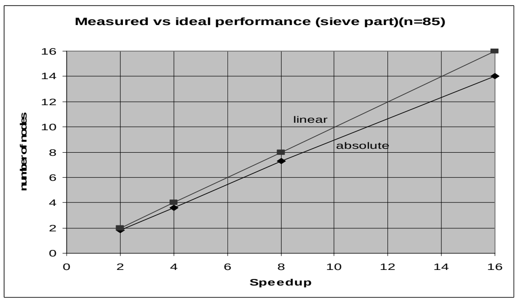

Figure 6: Sieve speedup

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

$$\eta_p = \frac{S_p}{p} \qquad (6)$$

Figure 7 and figure 8 shows the efficiency when factoring the 70 digit number with different numbers of processors.
We factored a 90 digits number (T_{90}) by Linux Cluster (1 master + 16 slaves) in about 19 minutes, the serial factorization of this number took 230 minutes. This factorization gives us a speedup factor about $S_p = \frac{230}{19} \approx 12$. We also factorized one 100-digts number (T_{100}) on Linux Cluster in about 139 minutes. The serial factorization of this 100-digits number takes about 1598 minutes therefore the speedup will be about $S_p = \frac{1598}{139} \approx 11.5$.

Table 2: Numbers to factorize in test runs

name	number
T_{55}	5258175809655438813858639893821229488068733010929393769 = 14238840625543078280423032 39. 36928398511763584700093442 71
T_{60}	284866257934932743912606492770903351821538977623644750168793 = 343512685094385609675326922719. 829274347923021180413710079047
T_{65}	24679783651778727241261683683145624763688041326971533109886273487 = 99438150911435425623745393072337. 99438150911435425623745393072337
T_{70}	6879948044669660936327555317971617639562508342877001422926336453690481 = 80314515302210968680376457284607477. 85662573182213597570465880916304653
T_{75}	773676088993114287725345906011845284482909990901212064345044960155918458201= 1914541104844979346206046100991942504 3.4041052380830230051052038027999779030 7
T_{80}	35257389446386676168774150173023775979486918446710181319009713526629561703858021=3459350325330995034448433770811114643743.1019191065681189844507214995289279607 4747
T_{85}	1014931723333867621143688721862898466487468145769181664597347779330681847768210317 0327=20012777061980517004007682646978341212406 47.50714187251003500208313750857292195722354 41
T_{90}	564281216732686093546301974147866661160062992245411388899347649969328986152613029921955417=563821056840957267298408080358035243161515823. 1000816145275430229670864901596906920133189879
T_{100}	5249037045637627042519927600536275057851062526634801207570717879489115751685860075675089828882745901=4709258869046480010021743503403085799235822198924 9. 11146206211213103445300889716795291039238207000234 9

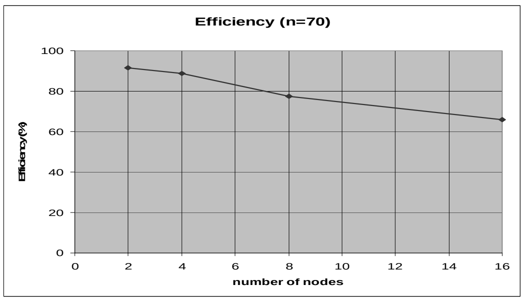

Figure 7: Total efficiency

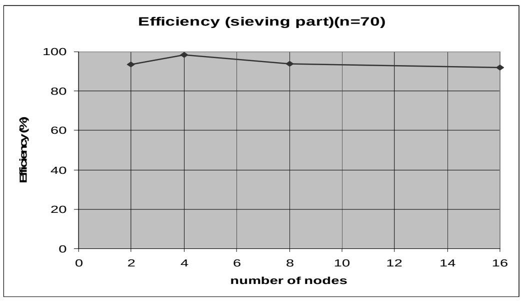

Figure 8: Sieving efficiency

4.4 Sources of inefficiency

One source of inefficiency is MPI overhead for communication; this overhead becomes more important when the length of the factorized number increases. Since the generated relations go to disk files, this matter leads to some I/O bottleneck.

There exist some non parallelized parts in our factoring program, these portions cause to decrease the efficiency. Non efficient software and hardware are other important sources of inefficiency also.

5 Conclusions

In this article we implemented an optimized serial version of SIQS and then in the second step we parallelized it for running on a Linux cluster. The presented benchmarks show that we have a speedup factor about 14 for 16 slave nodes (in 85-digits factorization). Also we could obtain an efficiency more than 90% for 70-digits factorization. These results together indicate that our serial optimization and proposed parallel algorithm strategies behave well.

References

[1] R. Brent, "Recent Progress and Prospects for Integer Factorization Algorithms", Lecture Notes in Comput. Sci. 1858 , pp. 3-22, 2000.

[2] S. Contini, *Factoring integers with the self-initializing quadratic sieve.* Master thesis, U. Georgia, 1997.

[3] R. Crandall and C. Pomerance, *PRIME NUMBERS A Computational Perspective.*, Springer, 2001.

[4] G. Wambach and H. Wettig, "Block Sieving Algorithms", 1995.

[5] K. Aoki and H. Ueda, "Sieving Using Bucket Sort", Asiacrypt, 2004.

[6] P.L. Mongomery, "A Block Lanczos Algorithm for Finding Dependencies over GF(2)", Advances in cryptography, Eurocrypt '95,

Lecture Notes in Comput. Sci. 921 (1995), pp. 106-120.

[7] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *PMPI: The Complete Reference*., The MIT Press, 1996.

[8] A. Lenstra and M. Manasse, “Factoring with two large primes”. Math. Comp., 63:785-798, 1994.