Avaliação de Desempenho de Abordagens Paralelas do Algoritmo de Fatoração por Tentativa de Divisão

Rodrigo Caetano de Oliveira Rocha[1], **Paulo Vitor Morais Magnani**[1], **Wallace Dias Ferreira**[1], **Henrique Cota de Freitas**[1]

[1] Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais
30.535-901 – Belo Horizonte – MG – Brasil

Abstract. *The difficulty of factoring a number is the safety of some algorithms of public-key cryptography, such as RSA. Integer factorization is a problem whose complexity increases exponentially, as the size, in digits, of the prime factors increases linearly. However, since it is not known whether or not there exists an efficient algorithm for this purpose, there is still room for questions and future works. This paper presents a comparative study of different approaches for decomposing a number into its prime factors, using a factorization algorithm by trial division. The studied approaches were: a sequential program, a parallelized program for multiple cores and a program using a cluster. Based on the evaluation made by this article, the approach using a cluster achieved better speedup values when compared to the other approaches, presenting lower execution times.*

Resumo. *A dificuldade de se fatorar um número é a segurança de alguns algoritmos de criptografia de chave pública, como o RSA. Fatoração de inteiros é um problema cuja complexidade cresce exponencialmente, a medida que o número de dígitos dos fatores primos cresce linearmente. Porém, visto que ainda não se sabe se existe ou não um algoritmo eficiente para este propósito, ainda há espaço para questionamentos e trabalhos futuros. Este trabalho apresenta um estudo comparativo entre diferentes abordagens para se decompor um número em seus fatores primos, usando o algoritmo de fatoração por tentativa de divisão. As abordagens estudadas foram: um programa sequencial, um programa paralelizado para múltiplos núcleos e um programa utilizando um cluster. Com base na avaliação realizada por este artigo, a abordagem usando cluster obteve melhores valores de speedup em relação às outras abordagens, apresentando menores tempos de execução.*

1. Introdução

Sabe-se que decompor um número composto em seus fatores primos é uma tarefa custosa. Encontrar os fatores primos de números grandes é uma tarefa demorada até mesmo para computadores. Porém, a condição de um número ser grande para um dado sistema computacional fatorar é relativo ao poder de processamento do sistema computacional.

A dificuldade de se fatorar um número é a segurança de alguns algoritmos de criptografia de chave pública, como o algoritmo de Rivest-Shamir-Adelman (RSA). Para o RSA, a segurança está na incapacidade de se fatorar, em tempo viável, inteiros muito

grandes [Landquist 2001, Kaliski 2006]. No entanto, fatoração de inteiros é um recurso da Matemática que facilita cálculos algébricos para aplicações nas mais diversas áreas. Sendo assim, algoritmos de fatoração de alto desempenho são essenciais.

Fatoração de inteiros é um problema cuja quantidade de operações necessárias para se fatorar um número cresce exponencialmente, a medida que o número de dígitos dos fatores primos cresce linearmente [Landquist 2001]. Porém, não se sabe se métodos mais eficientes podem ser encontrados, assim como não foi provado que tais métodos não existam. Por este motivo, apesar de ser um problema de muitos anos de estudo, fatoração de inteiros continua sendo uma área importante para se pesquisar [Kaliski 2006, Smiljanic and Ivanis 2011].

Este trabalho apresenta um estudo comparativo entre diferentes abordagens para se decompor um número em seus fatores primos. Foram feitos experimentos com um programa sequencial, um programa paralelizado para múltiplos núcleos e um programa utilizando um *cluster* de computadores. O programa paralelizado em múltiplos núcleos foi desenvolvido com a linguagem de programação C utilizando a API do *Open Multi-Processing* (OpenMP). O programa para execução em *cluster* foi desenvolvido com a linguagem de programação C utilizando a biblioteca de *Message Passing Interface* (MPI).

Como os computadores utilizados possuem uma arquitetura de 32 bits, portanto, suportando um inteiro máximo de 4294967295, houve a necessidade de se utilizar uma biblioteca que ofereça suporte a números grandes. A biblioteca utilizada para esta finalidade foi a biblioteca *GNU Multiple Precision Arithmetic* (GMP) [Granlund 2011].

A dificuldade de fatorar inteiros grandes expressa-se pelo tempo alto de processamento dos algoritmos. O problema abordado neste artigo trata da questão relacionada a este tempo e pode ser descrita através da seguinte pergunta: como aumentar o desempenho de fatoração de números grandes?

A técnica de fatoração utilizada por este trabalho é a fatoração por tentativa de divisão. Esta técnica de fatoração é uma técnica de simples entendimento e possui uma característica de concorrência bem definida. Este trabalho não tem como objetivo avaliar a melhor técnica de fatoração, pois existem diversas maneiras de se fatorar um inteiro, como apresentado pelo trabalho de [Brent 1999]. O objetivo deste trabalho é propor e avaliar abordagens paralelas para alto desempenho do algoritmo de fatoração por tentativa de divisão, analisando qual a melhor abordagem com base em um estudo experimental comparativo entre as abordagens utilizadas. Como contribuição, está a avaliação dos modelos sequencial e paralelos, além de compará-los entre si, com e sem otimização no código compilado.

Com base na avaliação realizada por este artigo, observa-se que apesar dos experimentos usando o *cluster* terem obtido maiores *speedups*, em relação à quantidade de processadores, os experimentos com o algoritmo de OpenMP obtiveram uma maior eficiência. Entretanto, considerando a escalabilidade, a abordagem por passagem de mensagem é uma boa solução para o problema de fatoração quanto a tempo de execução, pois para os casos críticos obteve melhores valores de *speedup*, apresentando um alto desempenho em relação às outras abordagens.

2. Método de Fatoração

Em teoria dos números, pelo teorema fundamental da aritmética temos que todo inteiro positivo n se fatora em um produto de números primos. Além disso, a fatoração de n em primos é única, a menos da permutação dos fatores primos, devido à propriedade comutativa da multiplicação.

O algoritmo de fatoração utilizado por este trabalho é a fatoração por tentativa de divisão. Fatoração por tentativa de divisão é uma técnica direta e de simples entendimento que busca por potenciais divisores, d, de n até que:

- $d > \sqrt{n}$, neste caso n é primo;
- $d < n$ e $d \mid n$, neste caso deve-se analisar se d é um fator primo ou composto.

Existem alguns refinamentos que podem ser feitos. Se n for ímpar, podem ser testados apenas divisores potenciais que sejam ímpares [Brent 1990].

A decomposição de um número composto em seus fatores primos pode ser feita usando um algoritmo recursivo de divisão e conquista com base na fatoração por tentativa de divisão, de maneira que se o divisor encontrado não for primo, este será decomposto em fatores primos recursivamente. Esta redução do problema é possível pelo seguinte lema: tendo $a, b, p \in Z$ e p primo, se $p \mid ab$, então $p \mid a$ ou $p \mid b$.

Sejam $n \in N$ e F a função de fatoração dada por:

$$F(n) = \begin{cases} n & \text{para } n \text{ primo} \\ F(d) \sim F(q) & \text{para } n = d \times q \end{cases}$$

onde $\sim$ é uma operação de concatenação de n-tuplas, e.g. $(a, b) \sim (c, d, e) = (a, b, c, d, e)$.

A função de fatoração F aplica a decomposição por recursão, cujo caso base é n ser um número primo. A recursão é aplicada sobre o divisor, d, e o quociente, q, de n. A Figura 1 apresenta um exemplo da execução do método de fatoração apresentado pela função descrita anteriormente. O exemplo fatora o inteiro 864, resultando nos fatores primos (2, 2, 2, 2, 2, 3, 3, 3), cuja multiplicação $2 \times 2 \times 2 \times 2 \times 2 \times 3 \times 3 \times 3 = 864$ retorna ao inteiro inicial, 864.

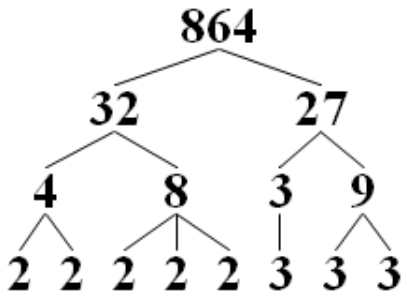

Figura 1. Decomposição em divisores primos do número 864

3. Trabalhos Relacionados

O trabalho de [Brent 1990] apresenta os principais algoritmos de fatoração de inteiros que possuem características de concorrência, sendo eles o algoritmo de fatoração por tentativa de divisão, o algoritmo *rho* de Pollard, o algoritmo (p-1) de Pollard, o algoritmo de curvas elípticas de Lenstra e o algoritmo de *Quadratic sieve*. Segundo [Brent 1990] o algoritmo de fatoração por tentativa de divisão tem ordem de complexidade $O(p \cdot logN)$ para encontrar o menor fator primo p de N. A implementação paralela deste algoritmo é direta e tem uma melhoria linearmente proporcional ao número de processadores sendo utilizados. Em [Brent 1999] é apresentado um maior detalhamento dos algoritmos vistos em [Brent 1990].

No trabalho de [Silva 2010] os autores apresentam uma técnica para fatoração de semiprimos (inteiros compostos por apenas dois primos, não necessariamente distintos). Esta técnica se baseia na reversão dos dígitos decimais do número e que, como os autores mostram, é altamente escalável. Por fim, são propostos os requisitos básicos de um *cluster* para a fatoração de um número de 309 dígitos decimais num intervalo de um ano.

Em uma abordagem diferente, o trabalho [Tordable 2010] obteve um ganho de *performance* na fatoração de inteiros através de uma técnica denominada *MapReduce*, a qual divide a carga de trabalho em blocos no método *Map* e depois converge os resultados através do método *Reduce*. A própria implementação do *MapReduce* fica responsável pela comunicação em rede, tolerância a falhas e balanceamento de cargas, em um sistema distribuído. Após análise, os autores concluem que esta técnica é bem escalável e adequada para problemas de tamanho médio.

Outro trabalho relacionado que visa tornar a fatoração de inteiros mais eficiente, mais especificamente aplicada à criptografia de chave pública do RSA, foi publicado por [Ambedkar et al. 2011]. No trabalho, é proposto e avaliado um algoritmo não probabilístico baseado no método de divisão de Fermat para fatorar um número inteiro positivo, processo o qual constitui a segurança do algoritmo RSA devido à sua complexidade. Com base em resultados experimentais, identificou-se uma melhoria crescente na velocidade de fatoração utilizando o método proposto em relação à aquela realizada pelo método de tentativa por divisão.

Outra abordagem sobre o tema de fatoração, aplicado à criptografia RSA, foi feita no estudo comparativo de [Smiljanic and Ivanis 2011], no qual se avalia a complexidade e eficiência de diversos algoritmos de fatoração, dentre eles, o método de tentativa por divisão. A partir das análises, reforça-se o fato de que a dificuldade na fatoração de números inteiros constitui toda a segurança do algoritmo RSA e, visto que ainda não se sabe se existe ou não um algoritmo eficiente para este propósito, ainda há espaço para questionamentos e trabalhos futuros.

O diferencial deste trabalho em relação aos trabalhos correlatos está na comparação entre o modelo sequencial, o modelo de memória compartilhada e o modelo de passagem de mensagem, além de compará-los entre si, com e sem otimização no código compilado. Este trabalho propõe e avalia abordagens paralelas para alto desempenho do algoritmo de fatoração de inteiros quaisquer, e não apenas para fatoração de semiprimos, abrangendo as mais diversas aplicações possíveis para fatoração de inteiros.

4. Implementação Sequencial

Uma implementação da abordagem sequencial do método de fatoração é apresentada pelo Algoritmo 1.

Algoritmo 1 Pseudo-algoritmo sequencial de fatoração

```
function FACTORING(n)
    T ← {}
    F ← {}
    while n ≠ 0 do
        d ← divisor(n)
        if d = 0 then
            F.push(n)
            if T.empty() ≠ True then
                n ← T.pop()
            else
                n ← 0
            end if
        else
            T.push(d)
            n ← n ÷ d
        end if
    end while
end function
```

Pode-se observar que o algoritmo apresentado não possui recursividade. Para eliminar a necessidade da recursão, o algoritmo faz uso de pilhas explícitas, sendo que F é a pilha que conterá todos os fatores primos ao final da execução do algoritmo.

A função $divisor$ retorna um divisor de n, caso n seja composto, ou retorna zero, caso n seja primo. A função $divisor$ é a parte do algoritmo que foi identificada como sendo concorrente. Esta função é apresentada em maiores detalhes pelo Algoritmo 2.

Algoritmo 2 Pseudo-algoritmo sequencial da busca por um divisor

```
function DIVISOR(n)
    if n > 2 and 2 | n then
        return 2
    else
        d ← 3
        while d ⩽ √n do
            if d | n then
                return d
            end if
            d ← d + 2
        end while
    end if
    return 0
end function
```

Analisando o algoritmo sequencial, é possível identificar trechos concorrentes na busca por um divisor d de n. De maneira mais específica, esta concorrência se encontra na estrutura de repetição que percorre valores de $d \geqslant 3$, enquanto $d \leqslant \sqrt{n}$, testando se d divide n e, caso divida, interrompe a busca e retorna d como sendo um divisor de n. A concorrência se dá ao fato de que testar se d_i divide n independe de já ter testado se d_{i-1} divide n. Portanto, esta busca pode ser realizada em intervalos independentes e desordenados, o que apresenta um potencial paralelismo o qual pode ser efetivamente explorado caso a arquitetura alvo ofereça suporte.

5. Implementação Paralela com OpenMP

Esta seção apresenta uma implementação do algoritmo de fatoração, voltada para uma arquitetura com processadores de múltiplos núcleos e que segue o modelo de memória compartilhada. Para a programação de algoritmos paralelos com hierarquia de memória compartilhada, o OpenMP é uma API (*Application Programming Interface*) que oferece um poderoso método de se alcançar escalabilidade em um software [Dagum and Menon 1998]. Em comparação com outras bibliotecas de mesmo propósito, como a *Pthreads*, o OpenMP apresenta um maior nível de escalabilidade e simplicidade de implementação, permitindo que seja utilizado para a paralelização de algoritmos que de outra forma poderiam acabar exigindo mais esforço de programação que a própria aplicação. Por este motivo, o OpenMP foi o escolhido para esta tarefa. O OpenMP é oferecido juntamente com o *GNU Compiler Collection* (*GCC*).

O *OpenMP* oferece um modelo de programação orientado a diretivas, que permitem ao programador especificar como o código deve ser executado em uma arquitetura de memória compartilhada. Através dessas diretivas é possivel, por exemplo, especificar laços de repetição que serão paralelizados, seções do código executadas simultaneamente por diversas *threads*, regiões sequenciais e críticas do código e barreiras para garantir sincronismo e integridade da aplicação.

Identificado o paralelismo potencial na função que busca por divisores para um determinado valor de n, foi possível instrumentar o algoritmo sequencial, com o uso de diretivas do *OpenMP*, transformando-o em um algoritmo paralelo. A função de busca paralelizada, bem como as diretivas *OpenMP* utilizadas em sua elaboração, são apresentadas pelo Algoritmo 3.

Como pode ser visto, o trecho realmente paralelo no código é situado na busca por algum $d \geqslant 3$ que divida n. Neste caso, por meio da diretiva *#pragma omp parallel*, definiu-se uma seção no código que é executada simultaneamente por todas as *threads* que foram especificadas para o programa. Nesta mesma seção paralela, cada *thread* ficou responsável pela identificação e processamento de um subintervalo de busca exclusivo, inferido com base no identificador único de cada *thread*, o qual é retornado por funções internas do *OpenMP*.

Visto que a arquitetura segue o modelo de memória compartilhada, foi necessário definir corretamente a política de visibilidade de variáveis manipuladas nas *threads* criadas. No algoritmo em questão, ambas as variáveis *interval* -a qual define o tamanho do intervalo que cada *thread* irá processar - e *foundDivisor* - a qual especifica se um divisor foi ou não encontrado -, precisam ser acessadas dentro do escopo das *threads* e, portanto, essas variáveis foram listadas como *shared* na diretiva *#pragma omp parallel*,

assim como a varável div, onde será armazenado o divisor encontrado. Após definidos os intervalos de busca de cada *thread*, cada uma irá processar um subintervalo na tentativa de encontrar um divisor de n. Caso algum divisor seja encontrado, então a variável compartilhada *foundDivisor* é atualizada para informar as demais *threads* - as quais continuam processando seus respectivos intervalos até que um divisor seja encontrado ou outra *thread* tenha o feito - para encerrar a busca por divisores e prosseguir com o processo de fatoração considerando o divisor recém-encontrado. Este ciclo se repete até que não existam mais valores de n para os quais deve-se tentar obter um divisor.

Algoritmo 3 Pseudo-algoritmo paralelo da busca por um divisor

```
function DIVISOR(n)
    foundDivisor ← False
    if n > 2 and 2 | n then
        foundDivisor ← True
        return 2
    else
        numThread ← getNumThreads()
        threadId ← getThreadNum()
        interval ← √n ÷ numThread
        div ← 0
        #pragma omp parallel shared(interval, foundDivisor, div){
        if threadId = 0 then
            start ← 3
            end ← interval
        else
            start ← interval × threadId
            end ← interval × (threadId + 1)
        end if
        d ← start
        while d ⩽ end and foundDivisor = False do
            if d | n then
                #pragma omp critical {
                if foundDivisor = False then
                    foundDivisor ← True
                    div ← d
                end if
                }
            end if
            d ← d + 2
        end while
        }
        return div
    end if
end function
```

6. Implementação Distribuída com MPI

Como um método alternativo para a programação paralela, tem-se o MPI (*Message Passing Interface*) [Snir et al. 1995], que é um padrão para passagem de mensagens. Porém, seu uso é mais complexo se comparado ao OpenMP, devido à necessidade de estar controlando explicitamente, por meio do uso de funções, toda a comunicação entre os nós, o envio e recebimento de mensagens síncronas e assíncronas, além da sincronização entre os nós ao longo da aplicação. Por outro lado, o uso do MPI implica, na maioria dos casos, em menores custos com hardware, além de ser altamente escalável. Para esta abordagem, será utilizada a biblioteca MPICH2.

A abordagem distribuída em *cluster* é bem semelhante à abordagem paralela utilizando múltiplos núcleos. Entretanto, elas se diferenciam pois a abordagem distribuída tem a necessidade de fazer uso de uma rede de computadores para se comunicar com os outros nós de processamento. A abordagem distribuída faz uso de uma arquitetura de mestre e escravos. O nó mestre é responsável por passar para os demais nós qual é o inteiro para o qual se está procurando por um divisor. O nó mestre é responsável também por armazenar as listas dos fatores encontrados e pela sincronização e controle da busca. Os algoritmos são apresentados em maiores detalhes pelos Algoritmo 4 e Algoritmo 5.

Algoritmo 4 Pseudo-algoritmo distribuído de fatoração (nó mestre)

Entrada: n

- Enquanto $2 \mid n$
  - Insere 2 como fator primo
  - Se $n = 2$, finalizar fatoração
    Senão $n \leftarrow n \div 2$
- Repete
  - Envia n para os nós
  - Calcula tamanho intervalo, $intervalo \leftarrow \sqrt{n} \div Num\_Processos$
  - Envia tamanho intervalo para os nós
  - Cada nó calcula seu intervalo de busca (início, fim)
  - Para cada inteiro, d, no intervalo de busca
    - Se receber algum divisor, div, de algum nó
      - Requisitar termino de todos os nós
    - Se $d \mid n$
      - $div \leftarrow d$
      - Requisitar termino de todos os nós
- Enquanto existir nó buscando por divisor
  - Se receber algum divisor, div, de algum nó
    - Requisitar termino de todos os nós
- Se encontrou algum divisor, div
  - Inserir div como fator temporário
  - $n \leftarrow n \div div$

  Senão
  - Inserir n como fator primo
  - Se existir fator temporário
    - Remover o próximo n dos fatores temporários

  Senão, finalizar fatoração

Algoritmo 5 Pseudo-algoritmo distribuído de fatoração (nós escravos)

- Repete
  - Receber n e receber tamanho do intervalo de busca
  - Cada nó calcula seu intervalo de busca (início, fim)
  - Para cada inteiro, d, no intervalo de busca
    - Se receber requisição de termino, finalizar busca
    - Se $d \mid n$
      - Enviar divisor, d, ao nó mestre
      - Finalizar busca
  - Se não encontrou divisor
    - Avisar seu termino ao master

O algoritmo distribuído acontece em três fases principais: fase de configuração, fase de busca pelo divisor e a fase de sincronização. A fase de configuração ocorre quando o nó mestre informa para os outros nós qual é o inteiro para o qual será feito a busca por um divisor. Esta comunicação é realizada usando um método de comunicação bloqueante síncrona. É nesta primeira fase também que cada nó calcula o seu intervalo de busca, semelhante ao que é feito no algoritmo paralelo com OpenMP. A fase de busca é o momento em que cada nó percorre o seu intervalo de busca, procurando por algum d que divida n. A terceira fase, a fase de sincronização, pode ocorrer de dois modos separados. O primeiro modo é quando um dos nós encontra um divisor de n. Ao encontrar um divisor, caso o nó não seja o nó mestre, este envia uma mensagem assíncrona ao nó mestre, informando-o que um divisor foi encontrado. Ao encontrar um divisor ou receber a mensagem de que algum divisor foi encontrado, o nó mestre informa aos demais nós, também de forma assíncrona, que um divisor foi encontrado e que a busca pode ser interrompida. Em seguida, será iniciada uma nova iteração, voltando para a primeira fase. O segundo modo ocorre quando nenhum nó encontra um divisor. Neste momento, todos os nós avisam para o nó mestre que nenhum divisor foi encontrado e, então, uma nova iteração é executada, voltando para a primeira fase.

7. Resultados

Esta seção apresenta a metodologia de avaliação, bem como uma descrição do ambiente para experimentos, e alguns resultados obtidos por experimentação com as diferentes abordagens apresentadas por este trabalho.

7.1. Metodologia de Avaliação

Os experimentos foram realizados nos laboratórios do ICEI da Pontifícia Universidade Católica de Minas Gerais. As máquinas disponíveis para experimentos possuem como configuração um processador Intel Core 2 Duo e 3 GB de memória RAM.

Foi gerado aleatoriamente um conjunto de inteiros positivos com tamanhos variados entre 1 e 35 dígitos. Este conjunto de inteiros foi utilizado para realizar os experimentos dos algoritmos apresentados por este artigo, de maneira que todos os modelos apresentados executaram a fatoração para todos os inteiros do conjunto. As comparações dos modelos foram feitas entre os tempos de fatoração para cada inteiro do conjunto e o tamanho dos inteiros fatorados. É importante garantir que a comparação seja feita usando os mesmo inteiros e não apenas o tamanho do inteiro pois a complexidade da fatoração

está no tamanho dos fatores primos encontrados, ou seja, inteiros cujos fatores primos sejam grandes terão um tempo de fatoração elevado.

O algoritmo sequencial foi compilado pelo *GNU Compiler Collection* (GCC). Foram executados dois grupos de experimentos para o algoritmo sequencial. O primeiro grupo de experimentos foi realizado com o código executável gerado sem nenhum parâmetro de otimização. O segundo grupo de experimentos foi realizado com o código executável gerado com o parâmetro de otimização nível 3, o parâmetro -O3. A Figura 2 apresenta o gráfico dos dois grupos de experimentos com o algoritmo sequencial. Com base nos experimentos, observa-se que não houveram melhorias pelo parâmetro de otimização para o algoritmo sequencial implementado.

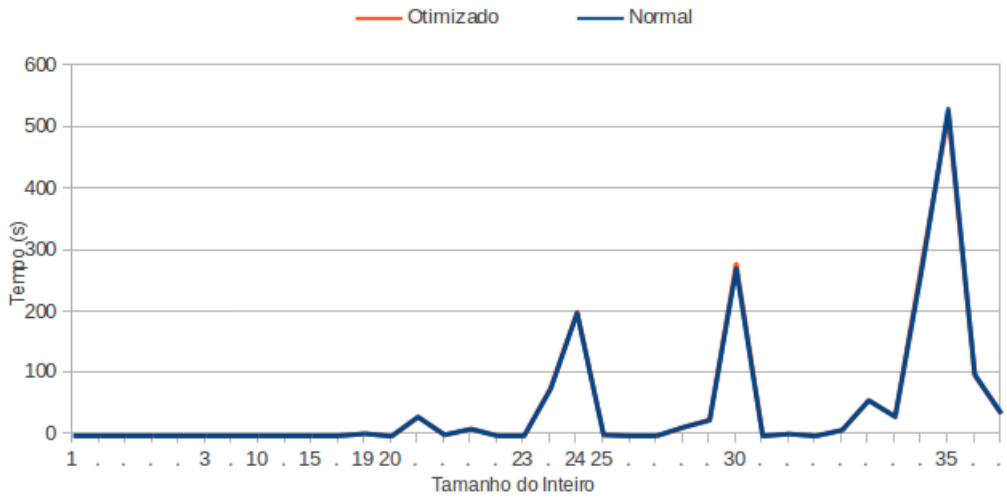

Figura 2. Experimentos com o algoritmo sequencial. Como otimização, está sendo considerando o parâmetro de otimização do compilador GCC, -O3

O algoritmo paralelizado em múltiplos núcleos foi compilado pelo GCC com a biblioteca OpenMP, que vem juntamente com a distribuição do pacote do *GNU Compiler Collection* (GCC). Foram executados dois grupos de experimentos para o algoritmo paralelo. O primeiro grupo de experimentos foi realizado com o código executável gerado sem nenhum parâmetro de otimização. O segundo grupo de experimentos foi realizado com o código executável gerado com o parâmetro de otimização nível 3, o parâmetro -O3. A Figura 3 apresenta o gráfico dos dois grupos de experimentos com o algoritmo paralelizado em múltiplos núcleos utilizando o OpenMP. Com base nos experimentos, observar-se que o programa executável gerado com o parâmetro de otimização apresenta um desempenho superior se comparado ao mesmo programa gerado sem o uso de otimizações de compilação.

O algoritmo distribuído foi compilado pelo GCC juntamente com a biblioteca MPICH2. Foram executados dois grupos de experimentos para o algoritmo distribuído. O primeiro grupo de experimentos foi realizado em um cluster com 10 máquinas. O segundo grupo de experimentos foi realizado em um cluster com 24 máquinas. Em ambos os grupos de experimentos, todas as máquinas usadas possuíam as mesmas configurações, sendo todas elas dos laboratórios do ICEI da PUC-MG. A Figura 4 apresenta o gráfico dos dois grupos de experimentos com o algoritmo distribuído utilizando MPI. Segundo

os experimentos, a implementação executada em um cluster com 24 nós foi a que obteve maior desempenho.

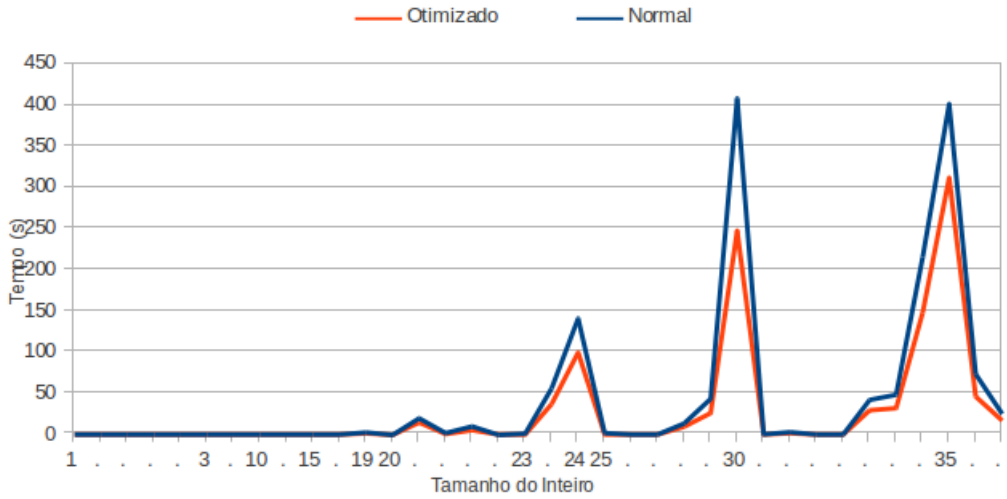

Figura 3. Experimentos com o algoritmo paralelo, utilizando OpenMP. Como otimização, está sendo considerando o parâmetro de otimização do compilador GCC, -O3

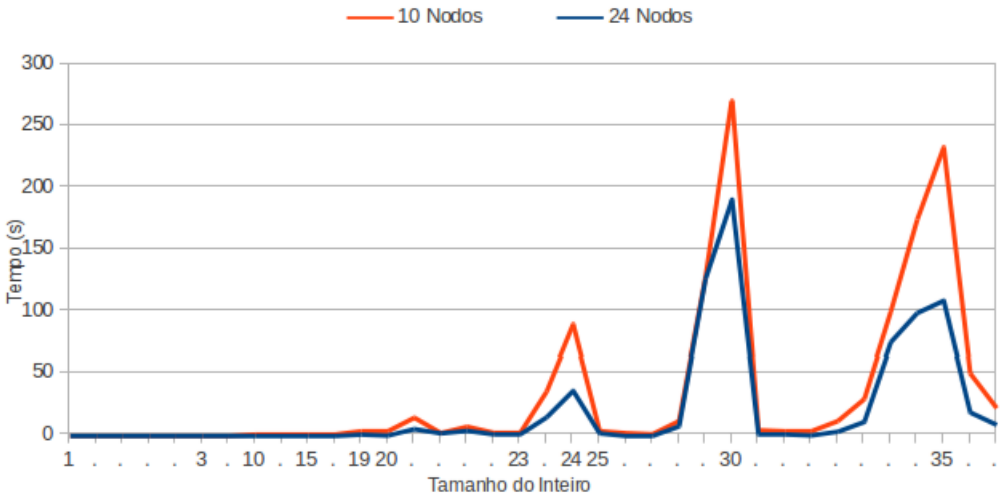

Figura 4. Gráfico comparativo entre o desempenho do cluster com 10 nós e do cluster com 24 nós.

7.2. Avaliação dos Resultados

A Figura 5 apresenta um gráfico contendo a comparação entre as diversas abordagens apresentadas por este artigo. Com base nos experimentos realizados, apesar de simples, o algoritmo sequencial apresenta o pior desempenho dentre todas as abordagens estudadas. Com isto, observa-se a necessidade de se ter uma abordagem mais eficiente caso seja necessário fatorar inteiros grandes. A abordagem distribuída foi a que apresentou melhor desempenho e se mostrou como sendo bem escalável. O algoritmo paralelo usando o OpenMP obteve grandes melhorias de desempenho quando comparado apenas ao sequencial.

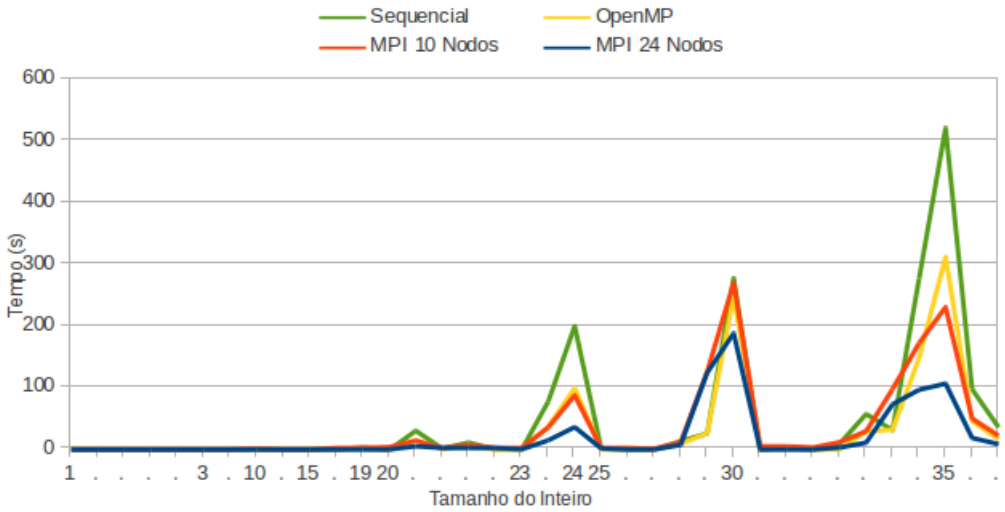

Figura 5. Comparação do desempenho entre as abordagens estudadas por este artigo.

Pela lei de Amdahl é possível realizar uma análise cuidadosa dos pontos mais relevantes apresentados no gráfico da Figura 5, que são os inteiros para os quais pode-se observar de forma clara uma diferença entre os algoritmos. A Tabela 1 apresenta uma comparação entre os *speedups* obtidos comparando o algoritmo sequencial com os demais algoritmos, o algoritmo usando OpenMP com otimização de compilação e o algoritmo usando MPI em um *cluster* com 24 nós. O *speedup* é obtido com base nos tempos gastos para se fatorar um determinado inteiro, isto é:

$$S_p = \frac{T_1}{T_p}$$

Onde p é o número de processadores, T_1 é o tempo de execução do algoritmo sequencial e T_p é o tempo de execução do algoritmo paralelizado com p processadores.

A Tabela 2 apresenta uma comparação entre os mesmos experimentos analisados na Tabela 1, porém agora fazendo uma análise da eficiência dos algoritmos. Eficiência, como visto pela lei de Amdahl, é uma relação entre o *speedup* e a quantidade de processadores usados, isto é:

$$E_p = \frac{S_p}{p} \quad \text{onde } p \text{ é o número de processadores.}$$

Pelos valores de *speedup* e de eficiência, pode-se observar que apesar dos experimentos usando o *cluster* terem obtido maiores *speedups*, em relação à quantidade de processadores, os experimentos com o algoritmo de OpenMP obtiveram uma maior eficiência. Entretanto, considerando a escalabilidade, a abordagem por passagem de mensagem é uma boa solução para o problema de fatoração quanto a tempo de execução, pois para os casos críticos obteve melhores valores de *speedup*, apresentando um alto desempenho em relação às outras abordagens.

Tabela 1. Speedup em relação ao algoritmo sequencial

Tamanho do Inteiro	OpenMP	MPI
20	2,0582	5,2083
24	2,0003	5,4311
30	1.1199	1,4709
30	1,9330	4,9339
35	1,6922	4,9264
35	2,1439	5,1478
35	2,1037	3,9103

Tabela 2. Eficiência em relação ao algoritmo sequencial

Tamanho do Inteiro	OpenMP	MPI
20	1,0291	0.2170
24	1,0002	0.2263
30	0.5599	0.0613
30	0,9665	0.2056
35	0,8461	0.2053
35	1,0720	0.2145
35	1,0519	0.1629

8. Conclusões

A segurança de alguns algoritmos de criptografia de chave pública, como o RSA, está na incapacidade de se fatorar, em tempo viável, inteiros muito grandes. Esta dificuldade de fatorar inteiros grandes expressa-se pelo tempo alto de processamento dos algoritmos, pois fatoração de inteiros é um problema cuja complexidade cresce exponencialmente, a medida que o número de dígitos dos fatores primos cresce linearmente. Este artigo apresentou um estudo comparativo, avaliando as abordagens paralelas para alto desempenho do algoritmo de fatoração por tentativa de divisão. A paralelização do algoritmo de fatoração foi feita pelo uso do modelo de memória compartilhada e do modelo de passagem de mensagem, separadamente.

Foram realizados experimentos comparativos entre o modelo sequencial, o modelo de memória compartilhada e o modelo de passagem de mensagem. Segundo os experimentos, a implementação em cluster com 24 nós foi a que obteve maior desempenho. Para um determinado inteiro de 35 dígitos, a implementação em cluster com 24 nós obteve um *speedup* de 5,1478 em relação à implementação sequencial do algoritmo.

Com base nos experimentos realizados, apesar de simples, o algoritmo sequencial apresenta o pior desempenho dentre todas as abordagens estudadas. Com isto, observa-se a necessidade de se ter uma abordagem mais eficiente caso seja necessário fatorar inteiros grandes. A abordagem por passagem de mensagem (MPI) foi a que apresentou melhor desempenho e se mostrou como sendo bem escalável, ou seja, considerando a escalabilidade, a implementação usando MPI é uma boa solução para o problema de fatoração quanto a tempo de execução, pois para os casos críticos obteve melhores valores de *speedup*, apresentando um alto desempenho em relação às outras abordagens.

Referências

[Ambedkar et al. 2011] Ambedkar, B., Gupta, A., Gautam, P., and Bedi, S. (2011). An efficient method to factorize the rsa public key encryption. *Communication Systems and Network Technologies, International Conference on*, 0:108–111.

[Brent 1990] Brent, R. P. (1990). Number theory and cryptography. chapter Parallel algorithms for integer factorisation, pages 26–37. Cambridge University Press, New York, NY, USA.

[Brent 1999] Brent, R. P. (1999). Some parallel algorithms for integer factorisation. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par '99, pages 1–22, London, UK, UK. Springer-Verlag.

[Dagum and Menon 1998] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55.

[Granlund 2011] Granlund, T. (2011). GNU multiple precision arithmetic library. http://gmplib.org/.

[Kaliski 2006] Kaliski, B. (2006). The mathematics of the rsa public-key cryptosystem. Technical report, RSA Laboratories.

[Landquist 2001] Landquist, E. (2001). The quadratic sieve factoring algorithm.

[Silva 2010] Silva, J. C. L. (2010). Factoring semiprimes and possible implications for rsa. In *Electrical and Electronics Engineers in Israel (IEEEI), 2010 IEEE 26th Convention of*, pages 000182 –000183.

[Smiljanic and Ivanis 2011] Smiljanic, J. and Ivanis, P. (2011). Attacks on the rsa cryptosystem using integer factorization. In *Telecommunications Forum (TELFOR), 2011 19th*, pages 550 –553.

[Snir et al. 1995] Snir, M., Otto, S. W., Walker, D. W., Dongarra, J., and Huss-Lederman, S. (1995). *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA.

[Tordable 2010] Tordable, J. (2010). Mapreduce for integer factorization. *CoRR*, abs/1001.0421.