Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

PROJETO E ANÁLISE DE ALGORITMOS

Quarto Trabalho: disponível em:
http://www.dcc.ufmg.br/~lcrocha/paa/tp4

Código Fonte: disponível em:
http://www.dcc.ufmg.br/~lcrocha/paa/tp4/src

Leonardo Chaves Dutra da Rocha

Professor - Nivio Ziviani

Belo Horizonte
19 de julho de 2006

Resumo

Neste trabalho apresentamos um a série de implementações paralelas para o Crivo de Eratóstenes, além da avaliação das mesmas. Nossas implementações foram divididas em dois grupos principais: as implementações de paralelismo de dados e as de paralelismo de controle. Para cada um dois grupos, avaliamos uma implementação que consideravam os números pares e uma que não considerava. Para a implementação desses algoritmos utilizamos uma biblioteca C de *Message-Passing*, desenvolvida para ser padrão em ambientes de memória distribuída, em "Message-Passing"e em computação paralela, a MPI(*Message Passing Interface*). Em nosso trabalho apresentamos também uma breve análise de implementações utilizando estruturas diferentes disponibilizadas pela biblioteca. Como resultado de nossas análises tivemos que as implementações que utilizaram paralelismo de dados conseguiram um bom Speedup, tanto a que utilizava todos os valores quanto a que utilizava apenas valores ímpares. Já com as implementações que utilizaram o paralelismo de controle não obtivemos o mesmo sucesso, todas elas não conseguiram um Speedup, com exceção de uma delas na qual utilizamos um pipeline onde as marcações eram feitas a medida ao mesmo tempo que os resultados parciais eram computados. Nessa última conseguimos um bom Speedup, mas não tão bom quanto o que conseguimos com o paralelismo de dados.

Sumário

1 Introdução

O objetivo deste trabalho é projetar algoritmos paralelos para serem executados em máquinas paralelas MIMD do tipo NOW-Network of Workstations [2, 3] para o problema de descobrir os primos existentes em um intervalo de 2 a N.

Como solução desse problema foi utilizado o Crivo de Eratóstenes, descrito na seção 1.1. Os algoritmos foram construídos utilizando a linguagem C juntamente com uma biblioteca de *Message-Passing* conhecida como MPI(*Message Passing Interface*). Uma breve descrição dessa biblioteca e de suas principais funções é apresentado na seção 1.2.

1.1 Crivo de Eratóstenes

Para encontrar todos os números primos numa lista de números inteiros pequenos, o modo mais rápido e fácil é o de Erastótenes. Tendo em conta que a multiplicação é uma operação mais fácil de realizar do que a divisão, Erastótenes de Cirene (no século III a.C.) teve a brilhante idéia de organizar estas computações em forma de crivo ou peneira, que ficou conhecido como Crivo de Erastótenes [8].

O funcionamento desse algoritmo é bastante simples, basta fazer uma lista com todos os inteiros maiores que um e menores ou igual a N e riscar os múltiplos de todos os primos menores ou igual à raiz quadrada de n $\sqrt{n}$. Os números que não estiverem riscados são os números primos.

Foram feitas várias implementações desse algoritmo, sendo divididas em três grupos principais: uma implementação seqüencial, apresentada na seção 2; duas implementações utilizando paralelismo de dados, apresentadas na seção 3 e; quatro implementações utilizando paralelismo de controle, apresentadas na seção 4. Antes de descrever os algoritmos implementados, apresentamos na seção seguinte uma breve descrição de paralelismo e da biblioteca MPI utilizada na implementações.

1.2 Paralelismo e MPI

Tradicionalmente, os programas de um computador são elaborados para serem executados em máquinas seriais, ou seja, máquinas que possuem apenas um processador e somente uma instrução é executada por vez. Dessa forma o tempo de execução irá depender de quão rápido a informação se "movimenta"pelo hardware.

No entanto existem diversas aplicações cujo o custo computacional é extremamente alto e implementações serias das mesmas é, na prática, inviável. Além dessas aplicações existem outras que necessitam de um tempo de resposta alto, e que muitas vezes não pode ser obtido com programas seqüenciais. Segue abaixo alguns exemplos dessas aplicações:

- Cosmologia e Astrofísica;
- Projeto de Materiais e Supercondutividade;
- Modelagem de Meio Ambiente;
- Engenharia Genética;

- Dinâmica dos Fluídos e Turbulência;
- Química Quântica, Estatísticas da Mecânica, e Física Relativista;
- Aplicações de Manipulação de imensas bases de dados como Mineração de Dados.

Uma estratégia que vem sendo bastante utilizada para obter resultados mais rápidos, de grandes e complexas tarefas é o paralelismo. O paralelismo consiste em dividir uma tarefa em várias pequenas tarefas e distribuir cada uma dessas pequenas tarefas entre várias unidades de processamento que irão executar as mesmas simultaneamente [26]. Conseqüentemente deve existir também uma coordenação dessas diversas pequenas tarefas para que nada de anormal ocorra. Dessa forma, a computação paralela consiste na resolução de um problema usando computadores paralelos. Ela é caracterizada pelo uso de várias unidades de processamento ou processadores para executar uma computação. O processamento paralelo é uma forma pela qual a demanda computacional é suprida através do uso simultâneo de recursos computacionais como processadores para solução de um problema.

Diversos fatores tem impulsionado o crescimento do uso de Algoritmos e Computação Paralela, dentre eles podemos destacar a criação de computadores de computadores de alto desempenho com paralelismo massivo, dotados de uma grande quantidade de processadores e a redução dos custos de processadores pessoais de alto desempenho o que possibilita a criação de redes de computadores com o mesmo poder computacional de supercomputadores.

A arquiteturas de computadores seriais e paralelas são classificadas utilizando a taxonomia de Flynn [10], embora a mesma não seja tão abrangente. Assim podemos distinguir 4 classes:

- **SISD (Single Instruction Single Data):** Arquitetura de von Neumann. A maioria dos computadores seriais estão nessa categoria. Apesar de poder executar as instruções utilizando um *pipeline*, computadores dessa categoria decodificam apenas uma instrução por tempo. Neste tipo de arquitetura é comum haver várias unidades funcionais trabalhando juntas, mas todas sobre a direção de uma única unidade de controle.
- **SIMD (Single Instruction Multiple Data):** Algoritmos assumem memória global compartilhada, com acesso a qualquer processador a custo constante. Um vetor de processadores sincronizados executa um único bloco de instrução, mas contém um número de unidades aritméticas, cada uma capaz de manipular seus próprios dados.
- **MISD (Multiple Instruction Single Data):** Nesta categoria, teríamos múltiplas instruções em um único dado, no entanto não existem computadores que se encaixam neste tipo de arquitetura.
- **MIMD (Multiple Instruction Multiple Data):** Arquiteturas formada por processadores que executam instruções independentemente. Contém a maioria dos sistemas multiprocessados que podem ser divididos em duas categorias:

- *multiprocessadores*: a comunicação é feita através de uma memória compartilhada entre os processadores;
- *multicomputadores*: a comunicação ocorre via mensagem, onde cada processo tem sua própria mensagem.

A maneira como os processadores se comunicam é dependente da arquitetura de memória utilizada no ambiente, que por sua vez, irá influenciar na maneira de se escrever um programa paralelo. Dessa forma existem dois tipos de arquitetura de memória: Memória Compartilhada e Memória Distribuída.

A memória compartilhada consiste em múltiplos processadores operando de maneira independente, mas compartilham os recursos de uma única memória, onde somente um processador opera por vez na memória. Dessa forma, é necessário uma sincronização no acesso, para leitura e gravação da memória, pelas tarefas paralelas. A "coerência"do ambiente deve ser mantida pelo Sistema Operacional. Essa arquitetura tem a vantagem de ser de fácil utilização e a transferência de dados é bastante rápida, no entanto pode limitar o número de processadores.

A memória distribuída consiste em múltiplos processadores operando independentemente, sendo que, cada um possui sua própria memória. Os dados entre os processos são compartilhados através de uma interface de comunicação (rede ou switch), utilizando-se "Message-Passing". A sincronização dos processos é de responsabilidade do programador. A vantagem dessa arquitetura é que o número de processadores é ilimitado e cada processador acessa, sem interferência e rapidamente, sua própria memória. A desvantagem dessa abordagem é o elevado "overhead"devido a comunicação e além disso o programador é responsável pelo envio e recebimento dos dados.

Na criação de programas paralelos devemos decompor o programa em sub-tarefas que serão executadas em paralelo. Existem basicamente dois tipos de decomposição: Decomposição Funcional cujo o problema é decomposto em diferentes tarefas, gerando diversos programas, que serão distribuídos por entre múltiplos processadores, para execução simultânea e; Decomposição de Domínio cujos os dados são decompostos em grupos, que serão distribuídos por entre múltiplos processadores que executarão, simultaneamente, um mesmo programa. A abordagem de paralelismo que utiliza a decomposição funcional é chamada de paralelismo de controle e a abordagem que utiliza decomposição de domínio é chamada de paralelismo de controle.

Em programação paralela, é essencial que se compreenda a comunicação que ocorre entre os processadores. Para isso existem duas maneiras distintas:

- *Message-Passing Library:* A programação da comunicação é explicita, ou seja, o programador é responsável pela comunicação entre os processos;
- *Compiladores Paralelos:* O programador não necessita entender dos métodos de comunicação, que é feita pelo compilador

Os algoritmos desenvolvidos nesse trabalho consideram a arquitetura MIMD(*Multiple Instruction Multiple Data)* com multiprocessadores, utilizando memória distribuída. Apresentamos soluções utilizando paralelismo de dados e soluções utilizando paralelismo de controle, utilizando uma biblioteca de *Message-Passing* denominada MPI(*Message Passing Interface*).

O modelo de "Message-Passing"é um dos vários modelos computacionais para conceituação de operações de programa. O modelo "Message-Passing"é definido

como o conjunto de processos que possuem acesso à memória local e a comunicação dos processos baseados no envio e recebimento de mensagens. A transferência de dados entre processos requer operações de cooperação entre cada processo (uma operação de envio deve "casar"com uma operação de recebimento). O conjunto operações de comunicação, formam a base que permite a implementação de uma biblioteca de "Message-Passing", como é o caso da biblioteca utilizada, MPI.

Assim, MPI (Message Passing Interface) é uma biblioteca de "Message-Passing", desenvolvida para ser padrão em ambientes de memória distribuída em "Message-Passing"e em computação paralela. É uma "Message-Passing"portável para qualquer arquitetura e tem aproximadamente 125 funções para programação e ferramentas para se analisar o desempenho. Utilizada por programas em C e FORTRAN, a plataforma alvo para o MPI, são os ambientes de memória distribuída, máquinas paralelas massivas, "clusters"de estações de trabalho. Nessa biblioteca, todo paralelismo é explícito, ou seja, o programador é responsável em identificar o paralelismo e implementar um algoritmo utilizando construções com o MPI.

Existem diversas implementações de MPI, em nosso trabalho utilizamos a implementação MPICH (ANL/MSU) de domínio público [18, 14, 16, 15, 17]. Após a instalação do pacote da biblioteca, para construir um programa que utilize as implementações disponíveis pela biblioteca é necessário primeiramente incluir o arquivo de cabeçalho da mesma em seu programa. como no exemplo abaixo:

```
# include <math.h>
# include <stdio.h>
      .
      .
      .
# include <mpi.h>
```

Feito isso, após a inicialização da função *main*, e após a declaração das variáveis globais, devemos inicializar o ambiente MPI, como mostrado abaixo:

```
int id,p;
int ...
.
.
.
MPI_Init (& argc, & argv);
.
.
.
```

É essa a função que faz a leitura dos parâmetros de configuração como número de processos, quais os computadores serão utilizados etc e faz a "cópia"do programa em questão para as computadores selecionados.

Após a inicialização dos processos, devemos realizar a identificação dos mesmos, além de buscar o número total de processos que estão sendo executados. Para isso temos duas funções, utilizadas como mostrado a seguir:

```
MPI_Comm_rank(MPI_COMM_WORLD, & id);
MPI_Comm_size(MPI_COMM_WORLD, & p);
```

Na variável *id* teremos o identificador do processo e na variável *p* teremos o número total de processos. O *MPI_ COMM_ WORLD* é o grupo de comunicação padrão do MPI que contém todos os processos. Com essa informações e com a inicialização do MPI, seu programa está pronto para executar em paralelo. Através do identificador é possível fazer com que cada processo realize tarefas distintas.

Para a construção do programa, existem diversas funções implementadas pelo MPI que podem auxiliar na construção de seu programa. Segue abaixo a relação de algumas das principais funções MPI:

- *MPI_ Barrier* usada para sincronizar a execução de processos de grupo. Ao ser chamada essa função, todo o processamento fica parado até que todos os processos cheguem no mesmo local;
- *MPI_ Send* função responsável pelo envio de mensagem entre processos. Essa é uma função blocante;
- *MPI_ Recv* função responsável pelo recebimento de mensagens de um outro processador. Essa também é uma função blocante;
- *MPI_ ISend* função responsável pelo envio de mensagem entre processos. Essa é uma função não-blocante;
- *MPI_ IRecv* função responsável pelo recebimento de mensagens de um outro processador. Essa também é uma função não-blocante;
- *MPI_ Reduce* operação que combina diversos valores, produzindo um único resultado. Para utilizar essa função, todos os processos devem chamar essa função, e o processo responsável pela produção do resultado único deve ser feita em consenso com todos os processos;
- *MPI_ Bcast* utilizada para enviar mensagens com o mesmo conteúdo de um processador para os demais. Nesse caso, a própria função implementa o *send* no caso do processo que chama esse função e ela também implementada o receive nos processos receptores.

E por fim, para finalizar o ambiente, todos os processos deve chamar a função de finalização, como mostrado abaixo:

```
MPI_Finalize
```

1.3 Ambiente de Execução

A execução de todos os experimentos realizados nessa trabalho foram realizados em um cluster de 20 máquinas. Cada uma das máquinas do cluster é um AMD Athlon

3.2Mhtz, 64 bits, com 2Gb de memória e 512Mb de cache. Essas máquinas estão todas interconectadas através de de uma rede Gigabit Ethernet.

Como métrica de avaliação da qualidade das implementações paralelas realizas utilizamos a métrica *Speedup*:

- **Speedup:** Relação entre o tempo de execução de um processo em um único processador e a execução em p processadores.

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

Feita a apresentação da arquitetura, do ambiente de programação paralela utilizado e do ambiente de execução dos experimentos, apresentamos na seção a seguir o algoritmo seqüencial implementado para posteriormente apresentarmos as implementações paralelas.

2 Implementação Seqüencial

O algoritmo seqüencial foi implementado de forma bastante simples. Para o programa é passado o valor de N que corresponde ao limite superior do intervalo onde a pesquisa será feita. A partir desse valor de N alocamos um vetor na memória do tipo *CHAR*. O tipo CHAR foi escolhido uma vez que cada posição do vetor ocupa apenas 1 byte e a esse vetor só serão atribuídos os valores "0"que indica que é primo, ou seja, posição do vetor não marcada e "1"que indica que **não** é primo, ou seja, posição do vetor marcada. O vetor já é alocado em memória e inicializado com "0", ou seja, inicialmente consideramos todos os valores como primos (não marcados). A partir daí o algoritmo começa a sua execução.

A partir da segunda posição do vetor verifica-se se o mesmo se encontra marcado ou não. Caso essa posição no vetor seja 0, ele é tido como primo e a partir do quadrado dessa posição, todos as posições múltiplas da mesma são marcadas como não primos, ou seja, são marcadas com 1. Feita essa marcação, a posição é incrementada até se encontrar um nova que se encontra com valor igual a 0, ou seja, desmarcada. Essa posição é então tomada como primo e a marcação a partir do quadrado dessa posição e feita até a posição N do vetor. No entanto, seguindo a idéia de Eratóstenes, essa procura por primos é feita somente até $\sqrt{n}$, isso porque depois desse valor todos os valores que não foram marcados como múltiplos dos primos encontrados até $\sqrt{n}$ são primos também.

Finalizada essa marcação, basta varrer o vetor procurando por posições iguais a 0 pois são esses os primos encontrados. Para exemplificar o funcionamento do algoritmo acima descrito, segue abaixo algumas ilustrações onde são determinados os primos menores ou igual a 20:

1. Inicialmente faz-se a lista dos inteiros de 1 a 20, no entanto a posição 1 é ignorada.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20

Figura 1: Passo 1

2. O primeiro número (2) está desmarcado e portanto é primo. Vamos mantê-lo e riscar todos os seus múltiplos. Desta forma, obtemos:

	2	3	4	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~

Figura 2: Passo 2

3. O próximo número não marcado é o 3, outro primo. Vamos mantê-lo e riscar seus múltiplos a partir do seu quadrado, ou seja, 9. Isso porque os demais menores que 9 já foram marcados pelo primo 2:

Figura 3: Passo 3

4. O próximo número primo é 5, porém não é necessário repetir o procedimento porque 5 é maior que a raiz quadrada de 20 ($\sqrt{20} = 4{,}4721$). Os números restantes são primos, destacados abaixo:

Figura 4: Passo 4

O código dessa implementação está em anexo com esse documento no Apêndice A.

2.1 Análise de complexidade

Para a análise de complexidade do algoritmo apresentamos primeiramente o algoritmo implementado:

```
void calculaPrimos(long long numero, long long *contador){
        char *Vetor;
        long long raizQuadrada = 0;
        long long i;
        long long j=0;
        (*contador) = 0;
        Vetor =  calloc (numero+1, sizeof(char));
        raizQuadrada = sqrt(numero);
        for(j=2;j<=raizQuadrada;j++){
                if(!(Vetor[j])){
                        for(i=j*j;i<=numero;i=i+j){
                                Vetor[i] = 1;
                        }
                }
        }
}
```

Analisando o algoritmo temos que o primeiro loop é executado até no máximo $\sqrt{N}$. Já o segundo loop são feitas as marcações dos múltiplos dos primos encontrados e o salto do loop e dado pelo incremento do primo, ou seja, a cada primo encontrado apenas algumas posições do vetor são visitadas, mais especificamente, as posições múltiplas do primo encontrado. Dessa forma, temos que para cada primo encontrados apenas $\frac{n}{primo}$ posições são visitadas. Assim, a análise de complexidade do algoritmo pode ser dada pelo seguinte somatório:

$$\sum_{i=2}^{\sqrt{n}} \frac{n}{i}$$

Que para uma primeira simplificação temos:

$$n \sum_{i=2}^{\sqrt{n}} \frac{1}{i}$$

Temos dessa maneira uma série harmônica monotônicamente decrescente que deve ser resolvida através da técnicas de aproximação por integrais. No entanto, analisando com mais cuidados nosso algoritmo e comparando com o somatório temos que nem todos os valores entre 2 e $\sqrt{N}$ são primos, apenas alguns e portanto esse somatório deve ser executado apenas para os primos encontrados. Dessa forma, para a resolução desse somatório devemos utilizar a Série Harmônica de Primos [34, 19, 21]. Essa série harmônica nos diz que:

$$\sum_{primo=2}^{x} \frac{1}{primo} = \ln\ln x + B_1 + o(1)$$

Onde $B_1 = 0.2614972128....$ Dessa forma, ignorando essa constante e $o(1)$ temos que a solução do somatório que representa a complexidade do algoritmo é dado por

$$n \sum_{i=2}^{\sqrt{n}} \frac{1}{i} = n(\ln\ln\sqrt{n})$$

Assim, temos que a complexidade de nosso algoritmo é $\theta(n \ln\ln n)$, o que é perfeitamente coerente com o que encontrado na literatura [25, 27]. Ainda sim, no intuito de verificar essa análise, realizamos alguns experimentos utilizando essa algoritmo variando o valor de N.

Variamos o valor de N entre 1 milhão e 400 milhões e contabilizamos o tempo gasto para cada uma das execuções. Foram feitas 5 execuções diferentes para cada valor de N e o tempo médio de execução dessas entradas é apresentado na Tabela 1. Observando a tabela temos que, assim como o esperado, temos um comportamento bem **próximo** do linear um vez que nossa complexidade de tempo foi $\theta(n \ln\ln n)$, mas não linear realmente.

Foi feita uma aproximação desses dados com a curva $f(x) = a(x(log(log(x))))$ através de uma regressão utilizando a ferramenta Gnuplot [9]. Dessa forma encontramos a função $f(x) = 2.3594(e-08)(x(log(log(x))))$, ou seja, comprovamos mais uma vez a complexidade de tempo $\theta(n \ln\ln n)$. O gráfico dos dados medidos e da curva encontrada pode ser observado na Figura 5, mostrando a convergência das mesmas.

Tabela 1: Relação Tempo de Execução pelo Valor de N

Valor de N	Tempo Execução(s)
1000000	0.039993
2000000	0.091986
3000000	0.144977
4000000	0.201969
5000000	0.255961
6000000	0.315951
7000000	0.372943
8000000	0.432934
9000000	0.491925
10000000	0.549916
20000000	1.163823
30000000	1.805725
40000000	2.464625
50000000	3.122525
60000000	3.808421
70000000	4.498316
80000000	5.188211
90000000	5.885105
100000000	6.578999
200000000	13.791903
300000000	21.183779
400000000	28.499667

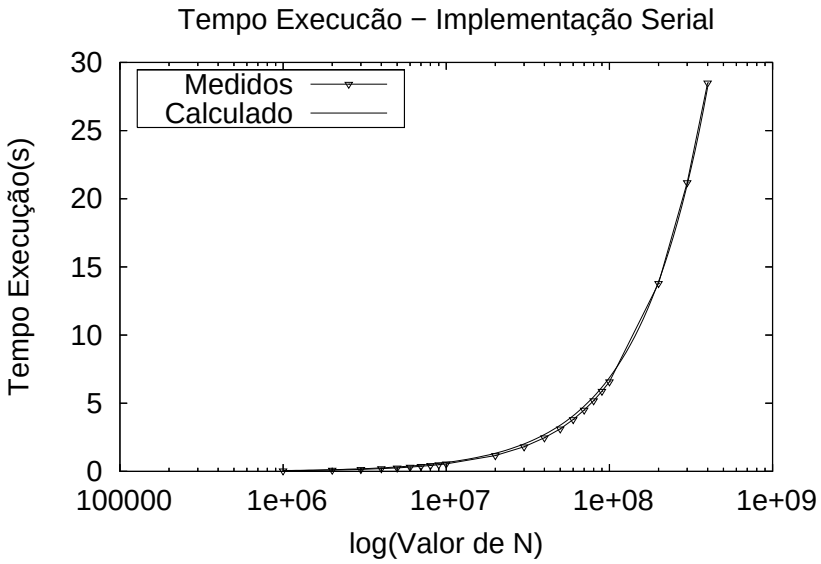


Figura 5: Análise de Complexidade

No algoritmo implementado, os dados que são armazenados é um vetor de tamanho N a qual armazena se o inteiro está ou não marcado. Dessa forma temos que a complexidade de espaço para nosso algoritmo é $O(n)$.

2.2 Exemplo de Funcionamento

Para mostrar o funcionamento do programa implementado, apresentamos a seguir alguns dos testes realizados, com diferentes valores de N.

- Primos encontrados para $N = 1000000$:

```
Execução:
./crivo.e -n 1000000 -o saida
Saída:
Quantidade de primos em 1000000 inteiros é 78498
```

- Primos encontrados para $N = 10000000$:

```
Execução:
./crivo.e -n 10000000 -o saida
Saída:
Quantidade de primos em 10000000 inteiros é 664579
```

- Primos encontrados para $N = 30000000$:

```
Execução:
./crivo.e -n 30000000 -o saida
Saída:
Quantidade de primos em 30000000 inteiros é 1857859
```

- Primos encontrados para $N = 200000000$:

```
Execução:
./crivo.e -n 200000000 -o saida
Saída:
Quantidade de primos em 200000000 inteiros é 11078937
```

- Primos encontrados para $N = 400000000$:

```
Execução:
./crivo.e -n 400000000 -o saida
Saída:
Quantidade de primos em 400000000 inteiros é 21336326
```

3 Paralelismo de dados

Como visto anteriormente, temos que o paralelismo de dados é a utilização de múltiplas unidades funcionais para aplicar a mesma operação simultaneamente aos elementos de um conjunto de dados. Um aumento de k no número de unidades funcionais leva a um aumento de k no *throughput*[1] do sistema, caso não haja nenhum *overhead* associado ao aumento de paralelismo.

Observando o algoritmos serial apresentado na seção 2, a chave do paralelismo se encontra no passo de marcação dos números não primos, ou seja, os números compostos. Para paralelizarmos esse algoritmo utilizando a abordagem de paralelismo de dados, devemos decompor os dados em fatias menores e distribuir essa fatias entre os processos. No entanto essa decomposição deve ser muito bem feita para que nenhum processo fique sobrecarregado em termos de processamento e também para que haja o mínimo de comunicação entre os processos.

A sobrecarga sobre um determinado processo pode fazer com que todo o processamento das demais tarefas fiquem paradas a espera da finalização desse processo sobrecarregado sub-utilizando o sistema como um todo. Já a comunicação entre os processos deve ser a menor possível pois o *overhead* gerado pela transmissão de dados, dada a latência da rede, pode levar também a uma sub-utilização da sistema. Assim, o ponto crucial de uma implementação paralela que utilizamos paralelismo de dados é a forma com que o dado total é dividido entre os processos. Na próxima seção discutimos três opções de decomposição apresentada por [27], sendo duas delas a utilizada em nossas implementações.

3.1 Estratégias de Decomposição de Dados

Uma primeira estratégia de decomposição dos dados e que é utilizada em outros contextos é particionar os dados utilizando o resto da divisão do índice do vetor pelo número de processadores para determinar quem é o processo responsável por aquele dado. Assumindo que p contém o número de processadores, esse esquema é como mostrado a seguir:

- **processo 0:** responsável pelo números 2, $2+p$, $2+2p$, ...
- **processo 1:** responsável pelo números 3, $3+p$, $3+2p$, ...
- **processo 2:** responsável pelo números 4, $4+p$, $4+2p$, ...
- **processo 3:** responsável pelo números 4, $5+p$, $4+2p$, ...

A grande vantagem dessa estratégia de decomposição é a facilidade com que é possível determinar quem é o processo responsável pelo dado de índice i, bastando aplicar a função *mod* sobre p ($i \mod p$). No entanto, a desvantagem do uso dessa estratégia em nosso contexto é que com essa decomposição podemos gerar um desbalanceamento considerável entre os processos. Por exemplo, se dois processos estão marcando os múltiplos de 2, o processo 0 irá marcar aproximadamente $\lceil (n-1)/2 \rceil$ elementos, enquanto que o processo 1 não irá marcar nenhum.

[1] Número de resultados gerados por unidade de tempo

Outra estratégia é dividir os dados em blocos contínuos de dados igualmente distribuídos, ou seja, blocos contínuos de dados de mesmo tamanho para cada processador. Caso o valor de N seja múltiplo do número de processos essa divisão é direta. No entanto, caso essa divisão não seja exata, o que é mais esperado, algumas considerações são necessárias.

Vamos supor $N = 1024$ e que $p = 10$. Nesse caso, $1024/10 = 102,4$, assim se 102 elementos forem atribuídos a cada processo, 4 elementos não seriam atribuídos a nenhum processo. Por outro lado, não temos 103 elementos para todos os processos. Uma decisão simples seria atribuir N/p aos $p-1$ primeiros processos e o último processo ficaria com os restantes elementos. Em nosso caso, o último processo ficaria com apenas 4 elementos a mais que o restante, mas se pensarmos em um ambiente de produção com 65.000 processadores e para valores de N grandes, temos que o último processo poderia ficar com até 65.000 elementos a mais que os demais processos, o que seria então um desbalanceamento considerável. Assim o que precisamos é quebrar os dados de tal forma que alguns processos fiquem com $\lceil (N/p) \rceil$ e outros com $\lfloor (N/p) \rfloor$. Nesse caso apresentamos dois métodos a seguir.

No primeiro método, primeiramente computa-se o valor de r tal que $r = N \mod p$. Se o valor de r é 0, significa que nada precisa ser feito e todos os processos recebem blocos de tamanho N/p. Caso o valor de r seja maior que 0, temos que os r primeiros processos recebem $\lceil (N/p) \rceil$ e os demais recebem $\lfloor (N/p) \rfloor$. Já no segundo método, os blocos de tamanho $\lceil (N/p) \rceil$ são distribuídos alternadamente entre o processos. Veja a Figura 6 que ilustra esses dois métodos.

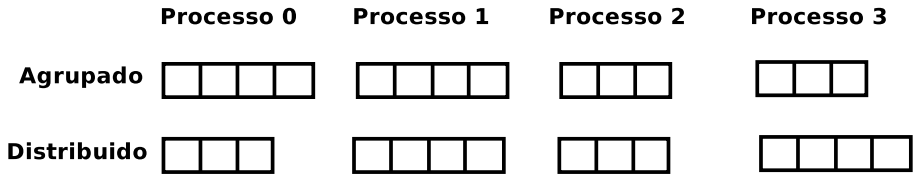


Figura 6: Métodos de Decomposição e Distribuição de Blocos

Dadas ambas as estratégias de divisão, existem duas questões que precisam ser resolvidas. A primeira é determinar qual a faixa de valores que é controlada por um determinado processador. A segunda é determinar, dado um valor, qual é o processo responsável pelo mesmo.

Para o primeiro método, temos que o valor do primeiro elemento controlado pelo processador i e dado por:

$$primeiro = i\lfloor N/p \rfloor + min(i, r)$$

Para esse mesmo método, temos que o valor do último elemento controlado pelo processo i é dado por:

$$ultimo = (i+1)\lfloor N/p \rfloor + min(i+1, r) - 1$$

E por fim, dado um elemento j, o processo que controla o mesmo é dado pela fórmula:

$$min(\lfloor j/(\lfloor N/p \rfloor + 1) \rfloor, \lfloor (j-r)/\lfloor N/p \rfloor \rfloor)$$

Já para o segundo método, temos que o valor do primeiro elemento controlado pelo processador i e dado por:

$$primeiro = \lfloor iN/p \rfloor$$

Para esse mesmo método, temos que o valor do último elemento controlado pelo processo i é dado por:

$$ultimo = \lfloor (i+1)N/p \rfloor - 1$$

E por fim, dado um elemento j, o processo que controla o mesmo é dado pela fórmula:

$$\lfloor (p(j+1) - 1)N/ \rfloor$$

Observando os cálculos que precisam ser feitos para cada uma das estratégias, utilizamos ambos os métodos, um em cada implementação realizada. Em termos de desempenho, ambas estratégias tem o mesmo valor, uma vez que o custo computacional é mesmo.

3.2 Implementação dos Algoritmos

Feita a escolha de como serão decompostos os blocos, devemos determinar qual ou quais processos serão responsáveis pela geração dos primos a serem marcados em todos os blocos de dados. Utilizando a idéia de Eratóstenes, temos que a procura por primos deve ser feita somente até $\sqrt{n}$, isso porque depois desse valor todos os valores que não foram marcados como múltiplos dos primos encontrados até $\sqrt{n}$ são primos também. Dessa forma, assim como a proposta apresentado por [27], optamos que os primos fossem calculados apenas pelo processo 0 e esse processo se torna, dessa forma, o responsável por repassar os primos encontrados para os demais processos.

Para que o processo 0 seja capaz de determinar quais são os primos cujos os múltiplos devem ser marcados pelos demais processos, ele deve ficar responsável por pelo menos os $\sqrt{n}$ primeiros elementos. Isso gera uma certa limitação em nossa implementação em relação ao número de processos a serem utilizados, limitando o mesmo a $\sqrt{n}$ processos. No entanto, essa é uma limitação razoável, uma vez que para uma valor de N como 1 bilhão, limitamos ao uso de 31622 processos, o que é bastante razoável.

Assim o algoritmo funciona da seguinte maneira, o processo 0, a medida que descobre os primos e marca seus múltiplos em seu bloco de dados, ele envia os mesmos para que os demais processos também marquem seus múltiplos em seus respectivos blocos. Essa parte do algoritmo é ilustrada a na Figura 7

Encontrados todos os primos e feitas todas as marcações em cada um dos blocos, cada processo é então responsável por enviar o total de primos encontrados em seu bloco para o processo 0. O processo 0 então contabiliza todos os primos de todos os processos e imprime a resposta em um arquivo de saída. Um preocupação que se teve durante toda a implementação é com relação a distinção entre o índice global do vetor de dados e seu índice local em cada um dos processos. Esse segunda parte do algoritmo é ilustrada na Figura 8

Implementamos duas versões utilizando esse funcionamento descrito acima: uma considerando todos os números até N e outra considerando os valores ímpares no

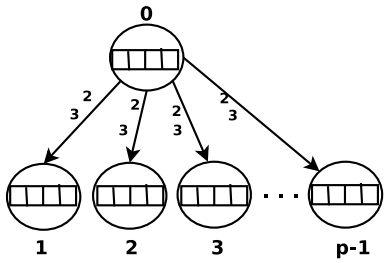


Figura 7: Calculando os Primos e Distribuindo entre Processos

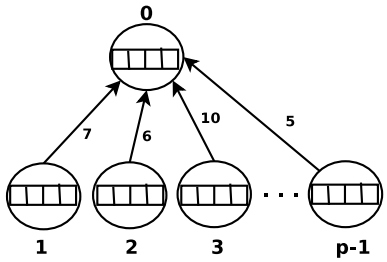


Figura 8: Envio dos Primos Encontrados e Contabilização Final

intervalo de 3 à N. Basicamente o que muda em cada uma dessas implementações são os cálculos das faixas que cada processo é responsável e os cálculos de transformação de índice global para local. Uma melhor discussão sobre cada uma dessas versões e dada nas seções seguintes.

3.2.1 Considerando Todos os Valores

Primeiramente realizamos as inicializações necessárias para a execução do algoritmo utilizando MPI, como inicialização do ambiente MPI, identificação do identificador do processo e do número de processos sendo executados. Nessa etapa colocamos também uma barreira esperando até que todos os processos cheguem a um certo ponto para que a contagem de tempo seja iniciada.

```
//---------------------------------------
// Inicializa o ambiente de execução MPI
//---------------------------------------
MPI_Init (&argc, &argv);
//---------------------------------
// Busca o identificado do processo
//---------------------------------
MPI_Comm_rank (MPI_COMM_WORLD, &id);
//-----------------------------------------
// Busca o numero de processadores utilizados
//-----------------------------------------
MPI_Comm_size (MPI_COMM_WORLD, &p);
//---------------------------------------
```

```
// Bloqueia todos os processos para iniciar
// a contagem de tempo
//-----------------------------------------
MPI_Barrier(MPI_COMM_WORLD);
//----------------------------
// Inicia a contagem de tempo
//----------------------------
tempo_execucao = -MPI_Wtime();
```

Feito isso, devemos calcular os limites dos processo, dado seu identificador, além do número total de elementos que o mesmo deve processar. Como nessa implementação consideramos todos os valores até N, esses cálculos são feitos conforme já mencionados na seção 3.1 com a diferença que consideramos apenas os valores maiores ou iguais a 2. Veja abaixo como esse cálculo é feito:

```
limite_inferior = 2 + id*(params.n-1)/p;
limite_superior = 1 + (id+1)*(params.n-1)/p;
elementos       = limite_superior - limite_inferior + 1;
```

O próximo passo é verificar se o número de processadores não ultrapassa o valor de $\sqrt{n}$. Para isso é feita a verificação se o número de elementos do processo 0 é no mínimo igual a $\sqrt{n}$. No entanto, novamente como avaliamos apenas os valores maiores que 2 em nosso algoritmo, isso deve ser considerado nesse cálculo, como mostrado a seguir:

```
tam_processo_0 = (params.n-1)/p;
if ((2 + tam_processo_0) < ((int) sqrt((double) params.n))) {
    if (!id) printf ("O número de processadores é maior que
    o permitido\n");
    MPI_Finalize();
    exit (1);
}
```

Para finalizar essa parte inicial do programa, devemos criar o vetor no qual as marcações deverão ser efetuadas. Além disso, inicialmente todos os números são considerados como primos, portanto todas as posições do vetor devem ser inicializadas com 0. Assim, utilizamos a função *calloc* de C que já faz essa inicialização.

```
marcados = (char *) calloc (elementos,sizeof(char));
```

Feitos todos os cálculos e todas a inicializações a parte principal do algoritmo é finalmente iniciada. Primeiramente o índice global do vetor é inicializado pelo processo 0 que é o responsável pelo mesmo. Esse índice é utilizado para o calculo do próximo primo. Além disso, todos os processo inicializam a marcação do múltiplos do primo 2.

```
if (!id) indice = 0;
primo = 2;
```

Cada processo deve determinar qual é o ponto de partida para que as marcações possam ser realizadas, ou seja, cada processo deve realizar seu cálculo da transformação do índice global para o local. Para cada primo recebido pelo processo, primeiramente o processo verifica se o quadrado do primo é menor que o limite inferior. Caso seja, o processo então verifica se os mesmos são múltiplos, nesse caso o ponto de partida é o próprio índice 0, caso contrário o índice é o primo subtraído do resto da divisão do seu limite inferior de seu bloco pelo primo. Caso a raiz quadrada do primo seja maior que o limite inferior, basta subtrair do quadrado dos primos o limite inferior. Como abaixo:

```
if (primo * primo <= limite_inferior)
   if (!(limite_inferior % primo)) primeiro = 0;
   else primeiro = primo - (limite_inferior % primo);
else {
   primeiro = primo * primo - limite_inferior;
}
```

Descoberto o ponto de partida da marcação, basta realizar a marcação dos múltiplos do primo em questão.

```
for (i = primeiro; i < elementos; i += primo) marcados[i] = 1;
```

O processo 0 é quem fornece os primos para os demais processos assim é ele que procura o próximo primo a ser utilizado. Ao final da procura, sempre desloca de 2 pois o primeiro termo do vetor do processo 0 é 2.

```
if (!id) {
   do{
      indice++;
   }while ((indice < elementos) && (marcados[indice]));
   primo = indice + 2;
}
```

Encontrado o próximo primo, o processo 0 deve enviar o mesmo para os demais processos. Em nosso caso implementamos duas maneiras diferentes de realizar esse envio. A primeira é através de uma simples chamada a função *MPI_Bcast* que faz o broadcast automáticos para os demais processos. Na outra implementação, o processo 0 informa apenas ao processo 1 o próximo primo. Esse por sua vez informa ao processo 2 e assim sucessivamente até que todos os processo saibam o primo corrente. Abaixo mostramos essas duas implementações, mas no entanto em nossas futuras análises consideramos apenas a implementação que realiza broadcast, uma vez que resultados preliminares mostraram que o uso da outra implementação não é recomendável.

```
if (params.broadcast){
  MPI_Bcast(&primo, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
}else
if (p > 1) {
```

```
        if (id > 0) {
           MPI_Recv (&primo, 1, MPI_LONG_LONG, id-1, 0,
           MPI_COMM_WORLD, &status);
        }
        if (id < (p-1)) {
           MPI_Send (&primo, 1, MPI_LONG_LONG, id+1, 0,
           MPI_COMM_WORLD);
        }
     }
```

Todo esse processo acima é executado enquanto o primo corrente seja menor que a raiz quadrada de N, seguindo a idéia do Crivo de Eratóstenes.

```
   } while (primo * primo <= params.n);
```

Ao final, marcados todos os múltiplos dos primos encontrados, cada processo sumariza seus dados contabilizando o total de elementos não marcados em seu bloco.

```
   count = 0;
   for (i = 0; i < elementos; i++)
       if (!marcados[i]) count++;
```

E finalmente agrupa-se todos os resultados em um resultado total e finalizamos também a contagem de tempo de execução. O agrupamento é feito utilizando a função *MPI_Reduce*.

```
   MPI_Reduce (&count, &total_count, 1, MPI_LONG_LONG, MPI_SUM, 0,
   MPI_COMM_WORLD);
   tempo_execucao += MPI_Wtime();
```

O código dessa implementação está em anexo com esse documento no Apêndice B.

3.2.2 Considerando Apenas os Ímpares

Primeiramente realizamos as inicializações necessárias para a execução do algoritmo utilizando MPI, como inicialização do ambiente MPI, identificação do identificador do processo e do número de processos sendo executados. Nessa etapa colocamos também uma barreira esperando até que todos os processos cheguem a um certo ponto para que a contagem de tempo seja iniciada.

```
   //--------------------------------------
   // Inicializa o ambiente de execução MPI
   //--------------------------------------
   MPI_Init (&argc, &argv);
   //--------------------------------
   // Busca o identificado do processo
```

```
//---------------------------------
MPI_Comm_rank (MPI_COMM_WORLD, &id);
//-----------------------------------------
// Busca o numero de processadores utilizados
//-----------------------------------------
MPI_Comm_size (MPI_COMM_WORLD, &p);
//-----------------------------------------
// Bloqueia todos os processos para iniciar
// a contagem de tempo
//-----------------------------------------
MPI_Barrier(MPI_COMM_WORLD);
//----------------------------
// Inicia a contagem de tempo
//----------------------------
tempo_execucao = -MPI_Wtime();
```

Nessa implementação consideramos os valores ímpares entre 3 e N, assim, primeiramente devemos calcular o número total de valores ímpares nesse intervalo.

```
impares = (params.n-1) / 2;
```

Como já mencionado, nessa implementação consideramos o método de divisão de blocos que agrupa os de tamanho maior no início. Assim devemos calcular o tamanho dos blocos maiores e o tamanho dos blocos menores, além do número processos que irão receber os blocos maiores e finalmente distribuir esses blocos entre os processos

```
menor_tamanho = impares/p;
maior_tamanho = menor_tamanho + 1;
num_dados_maiores = impares % p;
// Os primeiros num_dados_maiores processadores
// recebem os blocos maiores.
if (id < num_dados_maiores){
    elementos = maior_tamanho;
// Os demais processadores receberam os menores
}else{
    elementos = menor_tamanho;
}
```

Feito isso, devemos calcular os limites dos processo, dado seu identificador. Esses cálculos são feitos conforme já mencionados na seção 3.1 onde detalhamos a decomposição de blocos agrupada, no entanto, para esse cálculo consideramos os número total de ímpares nesse intervalo.

```
limite_inferior = 2*(id*menor_tamanho +
MENOR(id, num_dados_maiores)) + 3;
limite_superior = limite_inferior + 2*(elementos - 1);
```

O próximo passo é verificar se o número de processadores não ultrapassa o valor de $\sqrt{impares}$. Para isso é feita a verificação se o número de elementos do processo 0 é no mínimo igual a $\sqrt{impares}$. Essa verificação é feita levando-se em consideração que apenas os inteiros ímpares estão sendo avaliados.

```
// Verifica o tamanho do processo 0
if (num_dados_maiores > 0){
     tam_processo_0 = maior_tamanho;
}else{
     tam_processo_0 = menor_tamanho;
}
if ((1 + 2*tam_processo_0) < (int) sqrt((double)params.n)) {
   if (!id) printf ("O número de processadores é maior
   que o permitido\n");
   MPI_Finalize();
   exit (1);
}
```

Para finalizar essa parte inicial do programa, devemos criar o vetor no qual as marcações deverão ser efetuadas. Além disso, inicialmente todos os números são considerados como primos, portanto todas as posições do vetor deve ser inicializadas com 0. Assim, utilizamos a função *calloc* de C que já faz essa inicialização.

```
marcados = (char *) calloc (elementos,sizeof(char));
```

Feitos todos os cálculos e todas a inicializações a parte principal do algoritmo é finalmente iniciada. Primeiramente o indice global do vetor é inicializado pelo processo 0 que é o responsável pelo mesmo. Esse índice é utilizado para o calculo do próximo primo. Além disso, todos os processo inicializam a marcação do múltiplos do primo 3, uma vez que estamos avaliando apenas os números ímpares.

```
if (!id) indice = 0;
primo = 3;
```

Cada processo deve determinar qual é o ponto de partida para que as marcações devam ser realizadas, ou seja, cada processo deve realizar seu cálculo da transformação do índice global para o local. Para cada primo recebido pelo processo, primeiramente o processo verifica o ponto de partida.

```
   // Para cada novo primo, primeiro deve-se procurar no vetor
   // onde a  marcação deve começar!!
   if (primo * primo <= limite_inferior){
      // Se o quadrado do primo eh menor que limite_inferior,
      // entao se o resto da dividao entre o limite_inferior
      // e o quadrado for 0, a primeira ocorrência eh o
      // primeiro
      resultado = limite_inferior % primo;
```

```
        if (!resultado){
           primeiro = 0;
        }else{
           // Se o resto da divisão não for 0, então
           // é necessário ainda avaliar se o primo menos o
           // resto da divisão eh zero ou naun
           if ((primo - resultado)&1){
               primeiro = (2*primo - resultado)/2;
           }else{
               primeiro = (primo - resultado)/2;
           }
        }
     }else {
        // caso a raiz quadrada do primo seja maior que
        // o limite inferior, basta subtrair do quadrado
        // dos primos o limite_inferior e dividir por dois
        primeiro = (primo * primo - limite_inferior)/2;
     }
```

Descoberto o ponto de partida da marcação, basta realizar a marcação dos múltiplos do primo em questão.

```
     for (i = primeiro; i < elementos; i += primo) marcados[i] = 1;
```

O processo 0 é quem fornece os primos para os demais processos assim é ele que procura o próximo primo a ser utilizado. Ao final da procura, sempre desloca de 3 pois o primeiro termo do vetor do processo 0 é 3 e além disso multiplica-se por dois uma vez que estamos considerando apenas os ímpares.

```
     if (!id) {
        do{
           indice++;
        }while ((indice < elementos) && (marcados[indice]));
        primo = indice + 2;
     }
```

Encontrado o próximo primo, o processo 0 deve enviar o mesmo para os demais processos. Em nosso caso implementamos duas maneiras diferentes de realizar esse envio. A primeira é através de uma simples chamada a função *MPI_Bcast* que faz o broadcast automáticos para os demais processos. Na outra implementação, o processo 0 informa apenas ao processo 1 o próximo primo. Esse por sua vez informa ao processo 2 e assim sucessivamente até que todos os processo saibam o primo corrente. Abaixo mostramos essas duas implementações, mas no entanto em nossas futuras análises consideramos apenas a implementação que realiza broadcast, uma vez que resultados preliminares mostraram que o uso da outra implementação não é recomendável.

```
    if (params.broadcast){
      MPI_Bcast(&primo, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
    }else
    if (p > 1) {
       if (id > 0) {
          MPI_Recv (&primo, 1, MPI_LONG_LONG, id-1, 0,
          MPI_COMM_WORLD, &status);
       }
       if (id < (p-1)) {
          MPI_Send (&primo, 1, MPI_LONG_LONG, id+1, 0,
          MPI_COMM_WORLD);
       }
    }
```

Todo esse processo acima executa enquanto o primo corrente seja menor que a raiz quadrada de N, seguindo a idéia do Crivo de Eratóstenes.

```
   } while (primo * primo <= params.n);
```

Ao final, marcados todos os múltiplos dos primos encontrados, cada processo sumariza seus dados contabilizando o total de elementos não marcados em seu bloco.

```
   count = 0;
   for (i = 0; i < elementos; i++)
       if (!marcados[i]) count++;
```

E finalmente agrupa-se todos os resultados em um resultado total e finalizamos também a contagem de tempo de execução. O agrupamento é feito utilizando a função *MPI_Reduce*.

```
   MPI_Reduce (&count, &total_count, 1, MPI_LONG_LONG, MPI_SUM, 0,
   MPI_COMM_WORLD);
   tempo_execucao += MPI_Wtime();
```

O código dessa implementação está em anexo com esse documento no Apêndice C.

3.3 Análise de Complexidade

Nessa seção apresentamos a análise de complexidade dos algoritmos paralelos que utilizam paralelismo de dados. Como já mencionamos, basicamente ambos os algoritmos se comportam da mesma maneira, o que muda em cada uma dessas implementações são os cálculos das faixas que cada processo é responsável e os cálculos de transformação de indice global para local, que para o caso da análise de complexidade, não influencia. Assim a diferença na complexidade de cada um desses algoritmos será que, enquanto um deles consideramos N, no outro consideramos $N/2$.

Cada um dos processos desses algoritmos trabalha da mesma forma, ou seja, marca todos os múltiplos de um determinado primo, com exceção do processo 0 que além de marcar é responsável pelo envio dos primos para os demais processos. A marcação dos números compostos é feita da mesmas maneira que foi feita em nossa implementação serial, ou seja, assim temos que a complexidade é dado pelo seguinte somatório:

$$\frac{1}{p}\sum_{i=2}^{\sqrt{n}}\frac{n}{i}$$

Que para uma primeira simplificação temos:

$$\frac{n}{p}\sum_{i=2}^{\sqrt{n}}\frac{1}{i}$$

Dessa forma, para a resolução desse somatório, mais uma vez devemos utilizar a Série Harmônica de Primos [34, 19, 21]. Assim temos que a solução do somatório que representa a complexidade do algoritmo é dado por:

$$\frac{n}{p}\sum_{i=2}^{\sqrt{n}}\frac{1}{i} = \frac{n}{p}(\ln\ln\sqrt{n})$$

Assim, temos que a complexidade de nosso algoritmo é $\theta(\frac{n\ln\ln n}{p})$, o que é perfeitamente coerente com o que encontrado na literatura [27]. No caso do algoritmo que considera apenas os números ímpares o n é sempre dividido por dois, no entanto a complexidade também é $\theta(\frac{n\ln\ln n}{p})$.

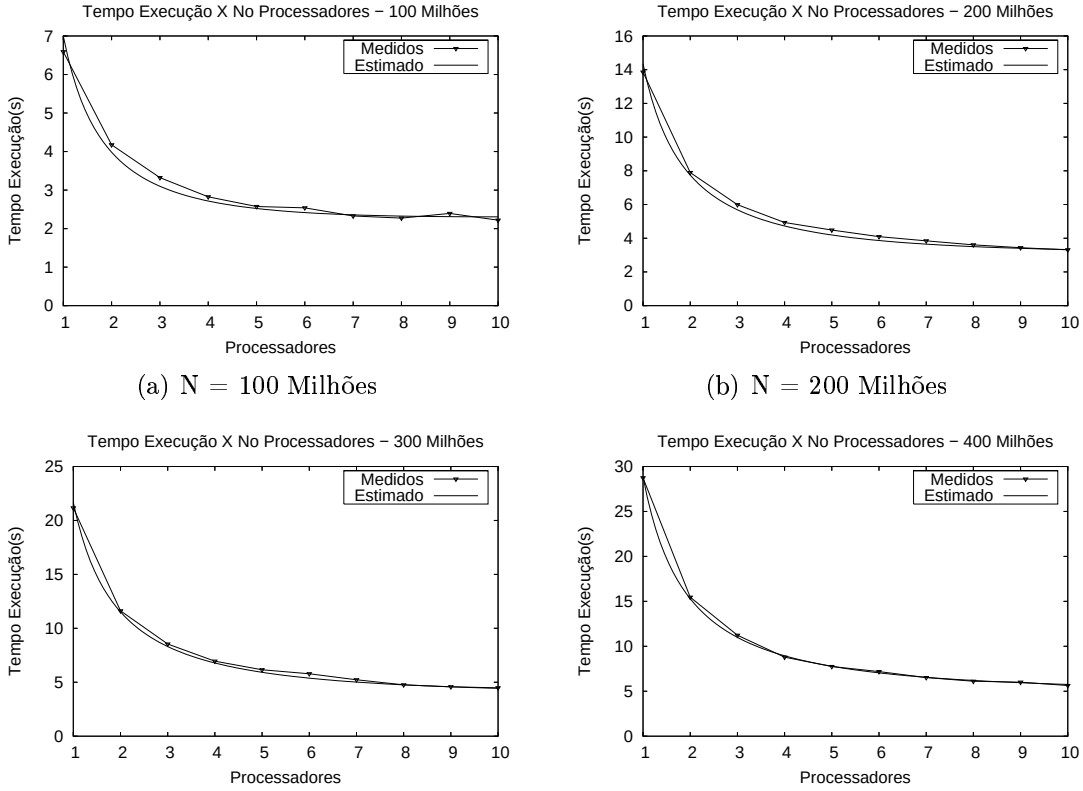
A partir da análise de complexidade dos algoritmos, é possível criar uma função que estima o tempo de execução do algoritmo, conforme mostrado em [27]. Vamos chamar essa função de $Tempo(N,p)$. Vamos tomar χ como sendo o tempo necessário para marcar uma célula particular como sendo múltipla de um primo. Esse tempo inclui não somente o tempo necessário para assinalar 1 a posição do vetor, mas também o tempo necessário para incrementar o loop e o tempo de avaliação da condição de terminação do loop. Como já vimos, a ordem de complexidade de tempo do algoritmo seqüencial é $\theta(n\ln\ln n)$. O valor de χ pode ser determinado experimentalmente executando o algoritmo seqüencial, assim o tempo de execução esperado para o algoritmo seqüencial é aproximadamente $\chi(n\ln\ln n)$. Em nosso caso, esse cálculo foi feito através da regressão dos dados utilizando a ferramenta Gnuplot [9], encontramos o valor $\chi = 2.3594(e-08)$, como mostrado na seção 2.1. No caso do algoritmo paralelo, temos que essa complexidade é dividida pelo número de processadores.

Além disso, temos que a cada iteração é feito um broadcast, e o custo de cada broadcast é aproximadamente $\lambda\lceil log(p)\rceil$, onde λ é a latência da mensagem [27]. O número de primos entre 2 e N é por volta de $n/\ln n$ [34, 19, 21], assim uma boa aproximação para o número de iterações do loop é $\sqrt(n)/\ln(\sqrt(n))$. Assim, temos que a função de tempos de execução estimado pode ser dado da seguinte forma:

$$Tempo(N,p) = \chi(\frac{n\ln\ln n}{p}) + \sqrt(n)/\ln(\sqrt(n))\lambda\lceil log(p)\rceil$$

Avaliamos a qualidade dessa função através da execução de experimentos, tanto para o algoritmo que considera todos os valores quanto para o que considera apenas os números ímpares.

Primeiramente apresentamos os resultados para o algoritmo que considera todos os valores até N. Para isso, executamos nosso algoritmo para os valores de N 100Milhões, 200Milhões, 300Milhões e 400Milhões, incrementando o número de processadores utilizados. Para cada combinação (N, p) foram feitas 5 execuções e ao final tomamos como resultado a média dessas execuções e geramos arquivos para cada valor de N relacionando o tempo de execução com o número de processadores utilizados. Para calcular o valor de χ, foi feita uma aproximação desses dados com a curva $Tempo(N,p) = \chi(\frac{n \ln \ln n}{p}) + \sqrt(n)/\ln(\sqrt(n))\lambda\lceil log(p)\rceil$ através de uma regressão utilizando a ferramenta Gnuplot [9]. Para as regressões realizadas sobre cada um dos arquivos, os valores de χ e λ foram bem próximos, o que mostra coerência uma vez que χ representa o tempo de marcação e λ a latência de uma mensagem. Assim, temos que os valores encontrados foram $\chi = 2.4164e-08s$ e $\lambda = 0.001421s$.



(a) N = 100 Milhões

(b) N = 200 Milhões

(c) N = 300 Milhões

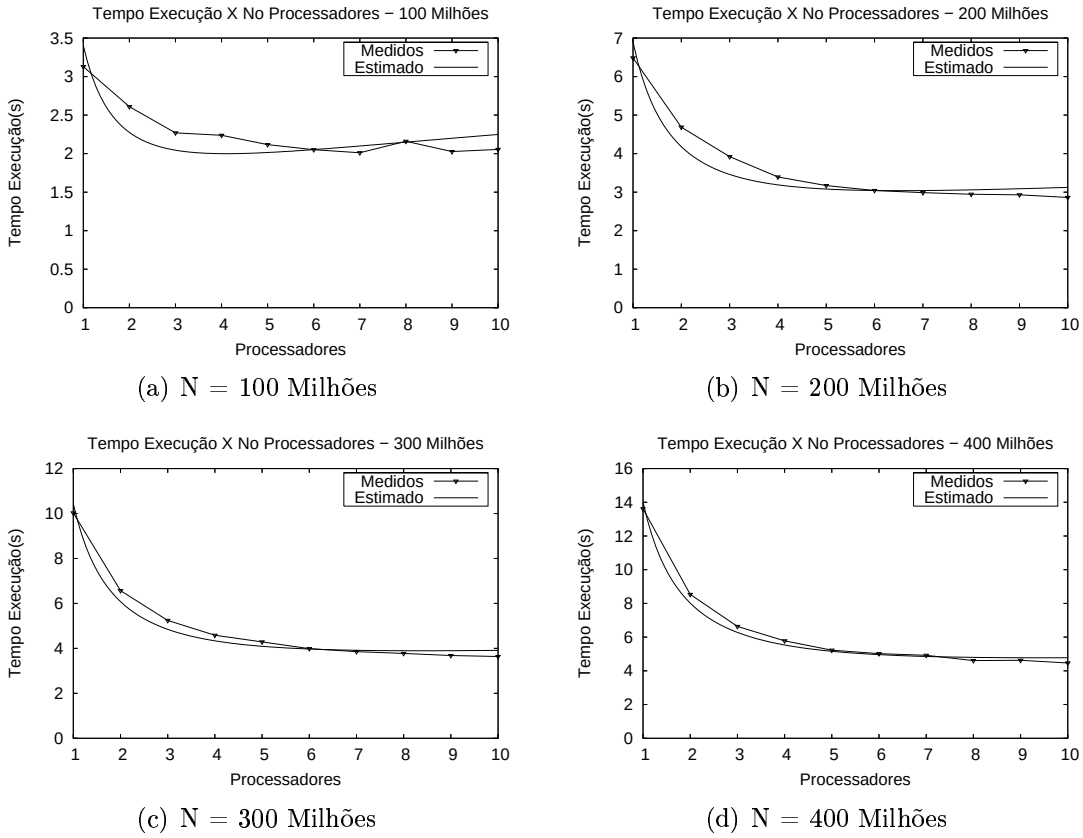
(d) N = 400 Milhões

Figura 9: Tempo Estimado vs Tempo Medido - Todos os Valores

Os gráficos 9(a) 9(b) 9(c) 9(d) apresentam como as curvas de tempo medido e tempo estimado se convergem para 100 Milhões, 200 Milhões, 300 Milhões e 400 Milhões, respectivamente. Observando as curvas temos que o tempo estimado sempre foi bem próximo do tempo medido, o que comprova a efetividade da função de estimativa de tempo de execução.

Agora apresentamos os resultados para o algoritmo que considera apenas os

números ímpares até N. mais uma vez, executamos nosso algoritmo para os valores de N 100Milhões, 200Milhões, 300Milhões e 400Milhões, incrementando o número de processadores utilizados. Para cada combinação (N, p) foram feitas 5 execuções e ao final tomamos como resultado a média dessas execuções e geramos arquivos para cada valor de N relacionando o tempo de execução com o número de processadores utilizados. Para calcular o valor de χ, foi feita uma aproximação desses dados com a curva $Tempo(N/2, p) = \chi(\frac{n/2 \ln \ln n/2}{p}) + \sqrt{(}n/2)/\ln(\sqrt{(}n/2))\lambda\lceil log(p)\rceil$ através de uma regressão utilizando a ferramenta Gnuplot [9]. Para as regressões realizadas sobre cada um dos arquivos, os valores de χ e λ foram mais uma vez bem próximos, o que mostra coerência uma vez que χ representa o tempo de marcação e λ a latência de uma mensagem. Assim, temos que os valores encontrados foram $\chi = 2.39857e{-}08$ e $\lambda = 0.001921s$.



(a) N = 100 Milhões

(b) N = 200 Milhões

(c) N = 300 Milhões

(d) N = 400 Milhões

Figura 10: Tempo Estimado vs Tempo Medido - Valores Ímpares

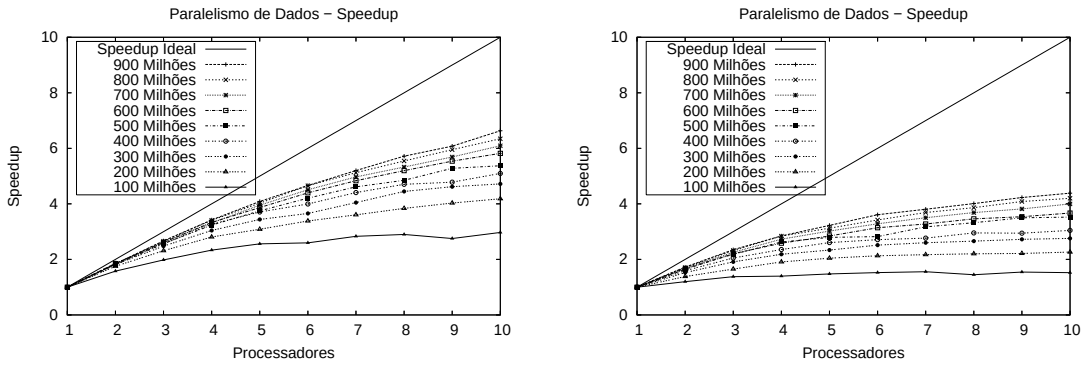
Os gráficos 10(a) 10(b) 10(c) 10(d) apresentam como as curvas de tempo medido e tempo estimado se convergem para 100 Milhões, 200 Milhões, 300 Milhões e 400 Milhões, respectivamente. Observando as curvas temos que o tempo estimado sempre foi bem próximo do tempo medido, o que comprova a efetividade da função de estimativa de tempo de execução.

3.4 Speedup e Análise do Comportamento dos Algoritmos

Para avaliarmos a qualidade dos algoritmos com paralelismo de dados implementados e analisar o comportamento dos mesmos sobre diversos valores de N e p, apresentamos

nessa seção o Speedup dos mesmo. Para isso, realizamos experimentos variando o valor de N entre 100 Milhões e 900 Milhões e incrementamos o número de processadores utilizados. Para cada combinação (N, p) foram feitas 5 execuções e ao final tomamos como resultado a média dessas execuções e geramos arquivos para cada valor de N relacionando o tempo de execução com o número de processadores utilizados. Isso foi feito para os dois algoritmos com paralelismo de dados implementados.

Realizados os experimentos, geramos as curvas de Speedup para cada um dos arquivos criados e plotamos todas elas nos gráficos 11(a)(que considera todos os valores de N) e 11(b) (que considera apenas os ímpares).



(a) Considerando Todos os Valores

(b) Considerando Apenas os Ímpares

Figura 11: Speedup do Paralelismo de Dados

Observando as curvas de Speedup, temos que para os menores valores de N o Speedup é bem menor. Isso acontece uma vez que, nesse caso, como o valor de N é pequeno, o tamanho dos blocos de cada processo também é pequeno. Conforme apresentamos na análise de complexidade de tempo do algoritmo serial, o tempo de processamento desses algoritmos é muito pequeno, assim, com blocos menores, a marcação é muito rápida e com isso os processos ficam mais sensíveis ao *overhead* gerado pela troca de mensagens (geração dos primos). Além disso, observando a Figura 12, temos que proporcionalmente ao valor de N, o número de primos é bem maior para valores menores de N do que valores maiores de N, ou seja, inicialmente a curva cresce muito rápido e posteriormente esse crescimento é amenizado. Isso torna o *overhead* ainda maior, pois a proporção de primos por valor de N é maior para valores menores de N fazendo assim com que o número de mensagens enviadas sejam também proporcionalmente maiores.

Ainda observando as curvas de Speedup, temos que a medida que aumentamos o valor de N temos que que o Speedup também aumenta, uma vez que o sistema como um todo fica menos sensível ao *overhead* de transmissão e geração dos primos, pois cada bloco é maior e com isso o processamento é mais intenso em cada processador e os mesmos ficam menos ociosos a espera dos próximos primos, ou seja, o sistema como um todo é bem melhor utilizado.

Entretanto, a medida em que aumentamos o número de processadores o Speedup tende a ser menor, diminui. Isso acontece porque a medida em que aumentamos o número de processadores, o tamanho dos blocos por processador também diminui proporcionalmente. Quanto menor o bloco, como já foi dito, maior é a sensibilidade do processo em relação ao *overhead* de transmissão e geração dos primos.

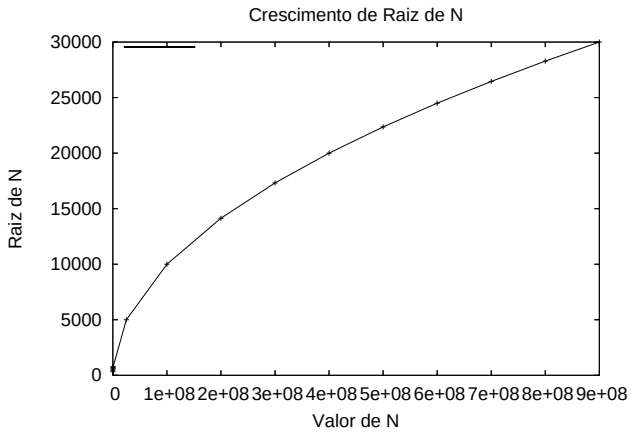


Figura 12: Comportamento de Raíz de N

Assim, nas seções 3.3 e 3.4 apresentamos não só uma análise de complexidade e avaliação de Speedup, mas também uma análise rigorosa do comportamento dos algoritmos utilizando paralelismo de dados comparando os resultados analíticos com os resultados experimentais.

3.5 Exemplo de Funcionamento

Para mostrar o funcionamento dos programa implementados, apresentamos a seguir alguns dos testes realizados, com diferentes valores de N.

- Primos encontrados para $N = 1000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file ./crivoCP.e -n 1000000
-o saida
Saída:
Quantidade de primos em 1000000 inteiros é 78498
```

- Primos encontrados para $N = 10000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file ./crivoSP.e -n 10000000
-o saida
Saída:
Quantidade de primos em 10000000 inteiros é 664579
```

- Primos encontrados para $N = 30000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file ./crivoCP.e -n 30000000
-o saida
Saída:
Quantidade de primos em 30000000 inteiros é 1857859
```

- Primos encontrados para $N = 200000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file ./crivoSP.e -n 200000000
-o saida
Saída:
Quantidade de primos em 200000000 inteiros é 11078937
```

- Primos encontrados para $N = 300000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file ./crivoCP.e -n 300000000
-o saida
Saída:
Quantidade de primos em 300000000 inteiros é 16252325
```

- Primos encontrados para $N = 400000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file ./crivoSP.e -n 400000000
-o saida
Saída:
Quantidade de primos em 400000000 inteiros é 21336326
```

4 Paralelismo de Controle

Como visto anteriormente, temos que o paralelismo de controle, em contraste com o paralelismo de dados onde o paralelismo é conseguido aplicando a mesma operação sob conjunto de dados diferentes, é obtido aplicando-se operações diferentes sob diferentes elementos da coleção simultaneamente. Observando o algoritmos serial apresentado na seção 2, a chave do paralelismo se encontra no passo de marcação dos números não primos, ou seja, os números compostos. Como nosso objetivo é o paralelismo de controle, a chave portanto está em fazer com que os processos marquem primos diferentes entre si, ou seja, cada processo fica responsável pela marcação de um conjunto de primos diferentes.

No entanto, como estamos tratando de memória distribuída, o problema crucial da implementação com paralelismo de controle é manter a consistência entre as memórias de tal forma que nenhum processo marque múltiplos de um inteiro que não é primo e não permitir que dois processos diferentes marquem os múltiplos do mesmo primo gerando trabalho desnecessário. Assim apresentamos nesse trabalho duas estratégias diferentes de paralelismo de controle para o problema do crivo. Essa implementações são apresentadas nas seções seguintes.

4.1 Estratégia de Replicação do Vetor

Na primeira delas existe uma cópia do vetor em cada um dos processos e cada um dos processos descobre todos os primos existentes até $\sqrt{n}$. A medida que o processo vai descobrindo cada primo ele determina então se aquele primo é ou não de sua responsabilidade, se o primo é de sua responsabilidade ele então marca todos os seus múltiplos até N e caso contrário marca todos os seus múltiplos apenas até $\sqrt{n}$. Veja o esquema de funcionamento na Figura 13

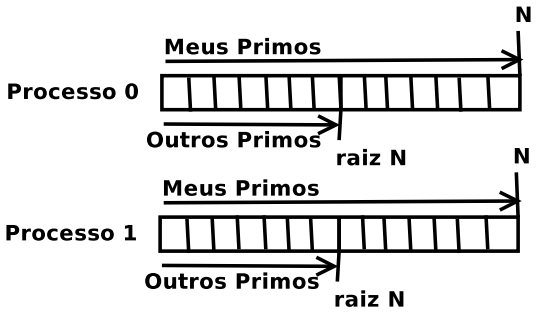


Figura 13: Replicação do Vetor

Para descobrir se o primo que ele encontrou é de sua responsabilidade ou não, o processo realiza uma divisão entre a ordem do primo encontrado e o número de processadores e o primo é de sua responsabilidade se o resto dessa divisão é igual ao seu identificador. Por exemplo, assumindo que o contador de primos encontrados seja 0 e encontramos o primo 2, dividimos 0 por p e como o resto dessa divisão é 0, esse primo passar ser de responsabilidade apenas do processo 0 e os demais processos realizam as marcações apenas até a $\sqrt{n}$. Ao final da execução de todos os processos,

cada um deles envia o resultado de suas marcações para o processo 0 e esse processo contabiliza o total de primos encontrados. Veja Figura 14

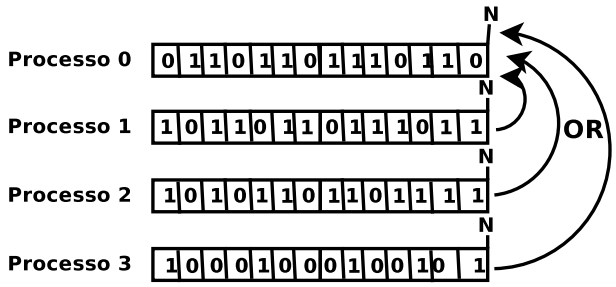


Figura 14: Redução em Um Processo

Utilizando essa estratégia, realizamos três implementações diferentes descritas a seguir.

4.1.1 Considerando Todos os Valores e Redução Única

Nessa implementação, consideramos todos os valores até N e todos os processos enviam seus resultados para um único processador. Assim, primeiramente realizamos as inicializações necessárias para a execução do algoritmo utilizando MPI, como inicialização do ambiente MPI, identificação do identificador do processo e do número de processos sendo executados. Nessa etapa colocamos também uma barreira esperando até que todos os processos cheguem a um certo ponto para que a contagem de tempo seja iniciada.

```
//--------------------------------------
// Inicializa o ambiente de execução MPI
//--------------------------------------
MPI_Init (&argc, &argv);
//---------------------------------
// Busca o identificado do processo
//---------------------------------
MPI_Comm_rank (MPI_COMM_WORLD, &id);
//-----------------------------------------
// Busca o numero de processadores utilizados
//-----------------------------------------
MPI_Comm_size (MPI_COMM_WORLD, &p);
//-----------------------------------------
// Bloqueia todos os processos para iniciar
// a contagem de tempo
//-----------------------------------------
MPI_Barrier(MPI_COMM_WORLD);
//----------------------------
// Inicia a contagem de tempo
//----------------------------
tempo_execucao = -MPI_Wtime();
```

Para finalizarmos a parte de inicialização devemos declarar o vetor onde as marcações serão feitas. Como todos os processos enviam o resultado de suas marcações, o processo 0 deve ter um vetor extra onde serão armazenadas a computação de todas os processos.

```
if (!id){
     marcados_total = (char *) calloc (params.n+1,sizeof(char));
}
marcados = (char *) calloc (params.n+1,sizeof(char));
```

Feitas as inicializações, basta realizar as devidas marcações. Como explicado anteriormente, cada processo verifica se o primo que ele encontrou é de sua responsabilidade. Caso seja, realiza marcações de seus múltiplos até N, caso contrário realiza as marcações somente até $\sqrt{n}$. As marcações são feitas alterando o valor da posição do vetor para 1.

```
indice = 2;
do{
     if (!marcados[indice]){
        if ((conta_primo%p) == id){
            varre_ate = params.n;
        }else{
            varre_ate = raizQuadrada;
        }
        for (i = indice*indice; i <= varre_ate; i += indice)
            marcados[i] = 1;
        conta_primo++;
     }
     ultimo_primo = indice;
     indice++;
}while(ultimo_primo * ultimo_primo <= params.n);
```

Finalizada as marcações o passo final é contabilização final onde devemos avaliar todos os resultados de todos os processos. Como marcamos com 1 as posições dos números compostos, nesse passo basta realizar uma operação OR entre todos os vetores gerados por cada processo. Em MPI, existe uma função que realiza essa redução e a operação , no caso OR, pode ser escolhida, a MPI_Reduce com a operação LOR. No entanto, como os vetores avaliados eram muito grandes, a chamada dessa função deve ser chamada por blocos, pedaços de dados de cada processo.

```
tam_bloco  = TAMANHO_MAXIMO/(2*p);
num_blocos = ceil((double) params.n/tam_bloco);
for(i=0;i<num_blocos;i++){
    if((params.n - ((i+1)*tam_bloco))<0){
        tam_msg = params.n - (i*tam_bloco) + 1;
    }else{
        tam_msg = tam_bloco;
```

```
        }
        MPI_Reduce(&(marcados[i*tam_bloco]),
        &(marcados_total[i*tam_bloco]), tam_msg,
        MPI_CHAR, MPI_LOR, 0,MPI_COMM_WORLD);
    }
```

Ao final, basta varrer o vetor de resultados a procura de posições com 0, ou seja, primos.

```
    if(!id){
        total_count = 0;
        for(i=2;i<=params.n;i++){
            if(!(marcados_total[i])){
                    total_count++;
            }
        }
   }
```

O código dessa implementação está em anexo com esse documento no Apêndice D.

4.1.2 Considerando Valores Ímpares e Redução Única

Nessa implementação, consideramos apenas os valores ímpares até N e todos os processos enviam seus resultados para um único processador. O processo de inicialização é mesmo processo já mostrado na seção anterior, com a diferença que os vetores são alocados apenas para os valores ímpares.

```
    impares = (params.n-1) / 2;
    if (!id){
        marcados_total = (char *) calloc (impares,sizeof(char));
    }
    marcados = (char *) calloc (impares,sizeof(char));
```

Feito a inicialização resta realizar a marcação dos números compostos. No entanto, como apenas os números ímpares são considerados, devemos realizar um cálculo de quanto será o salta para a marcação.

```
    conta_primo = 0;
    indice = 0;
    do{
       if (!marcados[indice]){
         if ((conta_primo%p) == id){
             varre_ate = impares;
         }else{
             varre_ate = raizQuadrada;
         }
         salto = 2*indice + 3;
```

```
        for (i = (((indice+1)*salto)+indice); i <= varre_ate;
             i += salto) marcados[i] = 1;
        conta_primo++;
      }
      ultimo_primo = salto;
      indice++;
    }while(ultimo_primo * ultimo_primo <= params.n);
```

Finalizada as marcações o passo final é contabilização final onde devemos avaliar todos os resultados de todos os processos. Como marcamos com 1 os números compostos, nesse passo basta realizar uma operação OR entre todos os vetores gerados por cada processo. Em MPI, existe uma função que realiza essa redução e a operação, no caso OR, pode ser escolhida, a MPI_Reduce com a operação LOR. No entanto, como os vetores avaliados eram muito grandes, a chamada dessa função deve ser chamada por blocos, pedaços de dados de cada processo. Isso foi feito da mesma da maneira apresentada na seção anterior. Ao final, basta varrer o vetor de resultados a procura de posições com 0, ou seja, primos.

O código dessa implementação está em anexo com esse documento no Apêndice E.

4.1.3 Considerando Valores Ímpares e Redução em Árvore

Nessa implementação, consideramos apenas os valores ímpares até N e o processo de redução, ou seja, de contabilização final é feita em árvore como mostrado na Figura 15

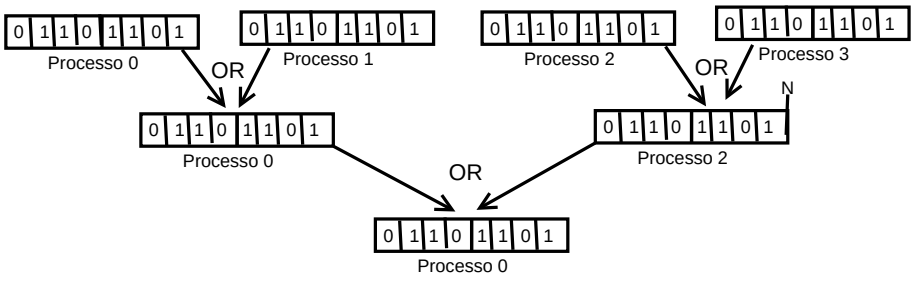


Figura 15: Redução em Árvore

Toda essa implementação segue a implementação já mencionada na seção anterior, com exceção da parte de redução, ou seja, da parte de contabilização do resultado final onde realizamos a redução em árvore. Nesse caso, ao invés de utilizar a função MPI_Reduce nós utilizamos as funções MPI_Send e MPI_Recv. Nesse caso, de dois em dois processos a contabilização dos primos encontrados é feita até que tenhamos apenas um processo com o resultado, o 0.

Primeiramente devemos calcular o número de iterações que serão necessárias para que, de dois em dois a redução aconteça. Esse número de iterações é dada pela altura da árvore que combina dois a dois os processos até chegar ao processo raiz, o processo 0.

```
iteracoes = 0;
folhas = 1;
while(folhas<p){
     iteracoes++;
     folhas = (int)pow((double)2,(double)iteracoes);
}
```

Para cada iteração, as reduções são feitas sempre entre dois processos consecutivos e o resultado parcial é sempre armazenado no processo de menor identificador. A cada nova iteração, os processos que armazenaram os resultados da iteração anterior são combinados e assim até que o resultado final esteja no processo 0.

```
for(i=0;i<iteracoes;i++){
     offset = (int)pow((double)2,(double)i);
     int aux = (int)pow((double)2,(double)(i+1));
     char *marcados_total=NULL;
     if((id%aux)==0){
      // Sou Raiz e naun preciso reduzir!!
         if((id + offset) < p){
           marcados_total = (char *)calloc(impares,sizeof(char));
           grupo[0] = id;
           grupo[1] = id + offset;
           MPI_Recv(marcados_total,impares,MPI_CHAR, grupo[1],
           0,MPI_COMM_WORLD,&status);
           int j;
           for(j=0;j<impares;j++){
             marcados_total[j] = marcados[j] || marcados_total[j];
           }
           free(marcados);
           marcados = marcados_total;
         }
     }else{
     // Nao sou raiz e preciso reduzir!
         if ((id%offset)==0){
             grupo[0] = id - offset;
             grupo[1] = id;
             MPI_Send(marcados, impares, MPI_CHAR, grupo[0],0,
             MPI_COMM_WORLD);
         }
     }
}
```

Ao final, basta varrer o vetor de resultados a procura de posições com 0, ou seja, primos. O código dessa implementação está em anexo com esse documento no Apêndice F.

4.2 Estratégia de Pipeline

Como veremos nos resultados que serão apresentados nas próximas seções, observamos que o grande ponto de conteção para as três implementações apresentadas nas seções anteriores estava na redução dos valores que cada um dos processos contabilizou. Dessa forma propomos e implementamos uma nova versão que utiliza pipeline entre as marcações dos números compostos e a contabilização dos mesmos. Essa estratégia nós dividimos em dois passos principais.

O primeiro passo consiste em determinar quais são os primos que cada um dos processos é responsável pela marcação de seus múltiplos. Nessa fase utilizamos a mesmas idéia já explicada anteriormente. Todos os processos possuem uma cópia do vetor, no entanto com apenas $\sqrt{n}$ elementos. Todos os processos, com exceção do processo 0, descobrem todos os primos até $\sqrt{n}$ e seleciona aqueles que são de sua responsabilidade. Isso é feito em paralelo pelos processos e o processo cujo identificador é $p-1$ é responsável em comunicar ao processo 0 a quantidade de primos encontrados. A Figura 16 ilustra esse passo.

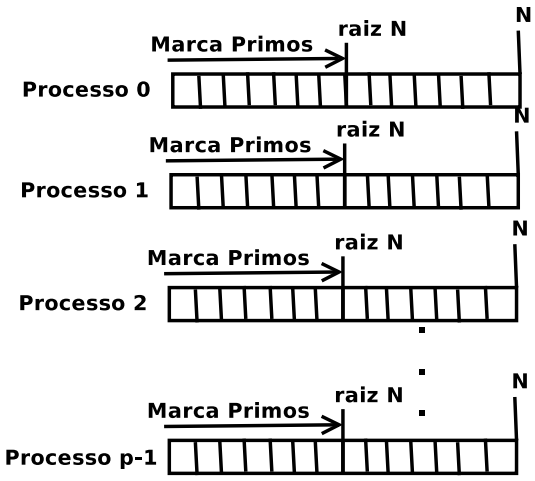


Figura 16: Passo 1 do Pipeline

Como podemos ver, esse passo gera um trabalho desnecessário no processos pois todos os processos realizam a mesmas tarefa sobre os mesmos dados, ou seja, podemos dizer que estamos sub-utilizando o sistema. No entanto, apesar desse re-trabalho em todos os processos, acreditamos que o mesmo ainda é melhor do que a realização de diversas trocas de mensagens entre os processos para que os mesmos encontrem os primos de sua responsabilidade, uma vez que o *overhead* de troca de mensagens pode ser muito alto.

A segunda fase é o pipeline propriamente dito. A partir do momento em que o processo 0 recebe o total de primos encontrados até $\sqrt{n}$, significa que os processos estão prontos para realizar suas marcações. Dessa forma o processo 0 quebra o restante dos dados em blocos menores e dispara os mesmos para o processo 1. Esse processo realiza suas marcações e repassa o bloco para o processo consecutivo ao mesmo tempo que recebe um novo pacote do processo 0. Isso acontece sucessivamente até que todos os processos estejam realizando marcações em blocos diferentes. A Figura 17 ilustra esse passo.

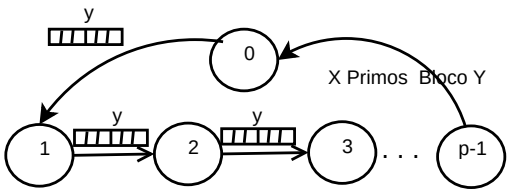


Figura 17: Passo 2 do Pipeline

Como todo pipeline, nossa estratégia tem um período de *warm up*, ou seja, aquecimento, até que todos os processos estejam realizando alguma marcação em algum bloco. Além disso, temos também um período de *cold down*, que é o momento que os primeiros processos já realizaram todas as marcações em todos os blocos e apenas os últimos processos realizam marcações. Temos que para melhorar o emprego dessa estratégia devemos sempre nos preocupar em reduzir tanto o tempo de *warm up* quanto o de *cold down*, principalmente não permitindo que os mesmos se intercedam. Para isso devemos avaliar sempre o melhor tamanho de bloco para cada caso, assim como o número de blocos que circulam simultaneamente pelo pipeline.

Em nossa implementação dessa estratégia utilizamos apenas os números ímpares até N. Assim como na implementação dos algoritmos de paralelismo de dados, como quebramos os dados em blocos, para todo bloco recebido existe a necessidade de transformação do índice local do vetor para o índice global para identificarmos os limites daquele bloco para que as marcações possam ser efetuadas. A forma com que implementamos foi bastante semelhante a implementada no paralelismo de dados que considerava apenas ímpares. Assim, a seguir mostramos algumas partes importantes da implementação do algoritmo. Segue agora como cada parte do programa foi implementado.

Chamamos cada processo com identificador diferente de 0 como processos escravos pois são eles os responsáveis pela marcação e o processo 0 nós chamamos de processo mestre uma vez que ele é quem delega os blocos a serem marcados para cada processo. A primeira etapa do algoritmo é feita apenas pelos processos escravos que é a seleção de todos os primos até $\sqrt{n}$. A medida que as marcações são feitas, os processos armazenem em um vetor os primos que cada um deles é o responsável.

```
if(id){
    // Processo escravo!!
    // Primeiramente monta a lista dos seus primos
    int *lista_primos = (int *) calloc(TAMANHO_MAXIMO,sizeof(int));
    indice = 0;
    conta_primo = 0;
    ultimo_primo = 0;
    do{
         if (!marcados[indice]){
           if ((conta_primo%(p-1)) == (id-1)){
               lista_primos[ultimo_primo] = 2*indice + 3;
               ultimo_primo++;
           }
           conta_primo++;
```

```
          salto = 2*indice + 3;
          for (i = (((indice+1)*salto)+indice); i <= raizQuadrada;
          i += salto)
             marcados[i] = 1;
        }
        indice++;
   }while(indice<=raizQuadrada);
```

Feita a marcação, apenas o último processo escravo envia a quantidade de primos encontrados para o processo 0.

```
if (id == p-1){
  MPI_Send(&conta_primo, 1, MPI_LONG_LONG, 0,0, MPI_COMM_WORLD);
}
```

O processo 0, por sua vez, ao receber essa mensagem, dispara os blocos as serem marcados para o processo 1. Uma vez que o tamanho dos blocos já são pré-definidos, a única verificação que deve ser feita é que o último bloco pode ter um tamanho menor que o tamanho do bloco e esse cálculo deve ser feito para repassar o tamanho do bloco para o processo 1.

```
   MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
   MPI_Recv(&primos,1, MPI_LONG_LONG, status.MPI_SOURCE, 0,
   MPI_COMM_WORLD, &status);
   total_count+=primos;
   i=0;
   j=0;
   do{
      if ((janela) && (i<num_blocos)){
         if(((impares-(raizQuadrada+1)) - ((i+1)*tam_bloco))<0){
             tam_msg = (impares-(raizQuadrada+1)) - (i*tam_bloco);
         }else{
             tam_msg = tam_bloco;
         }
         MPI_Send(&(marcados[0]), tam_msg, MPI_CHAR, 1,i,
         MPI_COMM_WORLD);
         janela--;
         i++;
       }
```

Enquanto os blocos são enviados pelo processo mestre, os processos escravos vão recebendo os blocos, realizam suas marcações e repassam os mesmos para o processo da frente no pipeline. A marcação é feita conforme foi feita no algoritmo de paralelismo de dados que marca somente os ímpares. Primeiro devemos calcular o índice do primeiro número que deve ser marcado, ou seja, o cálculo do índice local em relação ao global. Posteriormente basta marcar os múltiplos realizando saltos considerando também que só tratamos de ímpares. Isso deve ser feito para cada um

dos primos armazenados como sendo de responsabilidade do processo. Essa foi a parte mais problemática da implementação pois chegar a uma fórmula que realiza esses cálculos não é trivial e além disso é a principal parte do programa.

```
limite_inferior = (2*(tam_bloco*indice))
                  + 2*(raizQuadrada +1) +3;
for(i=0;i<conta_primo;i++){
    ultimo_primo = lista_primos[i];
    // Para cada novo primo, primeiro deve-se procurar no vetor
    // onde a marcação deve começar!!
    if (ultimo_primo * ultimo_primo <= limite_inferior){
        resultado = limite_inferior%ultimo_primo;
        if (!resultado){
            primeiro = 0;
        }else{
            if ((ultimo_primo - resultado)&1){
                primeiro = (2*ultimo_primo - resultado)/2;
            }else{
                primeiro = (ultimo_primo - resultado)/2;
            }
        }
    }else {
      primeiro = (ultimo_primo * ultimo_primo
                  - limite_inferior)/2;
    }
    for (j = primeiro; j < elementos_rec; j += ultimo_primo) {
        marcados[j] = 1;
    }
}
```

Todos os processo escravos, após realizarem suas marcações no bloco repassa o bloco para o processo consecutivo no pipeline, com exceção do último processo (identificado igual a $p-1$) que deve contabilizar o números de posições não marcadas, ou seja, o número de primos encontrados naquele bloco e repassar esse valor ao processo mestre para que o processo mestre realize a agregação.

```
if (id != p-1){
  MPI_Send(marcados, elementos_rec, MPI_CHAR, id+1,status.MPI_TAG,
  MPI_COMM_WORLD);
}else{
 for (i = 0; i < elementos_rec; i++) if (!marcados[i])
    total_primos++;
 MPI_Send(&total_primos, 1, MPI_LONG_LONG, 0,0, MPI_COMM_WORLD);
}
```

O processo mestre por fim, a medida que vai recebendo as mensagens do processo escravo $p-1$ vai contabilizando o total de primos encontrados.

```
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
MPI_Recv(&primos,1, MPI_LONG_LONG, status.MPI_SOURCE,
status.MPI_TAG, MPI_COMM_WORLD, &status);
total_count+=primos;
```

Apresentado cada um dos algoritmos implementados utilizando a estratégia de paralelismo de controle, apresentamos a seguir uma análise detalhada do comportamento de cada um desses algoritmo em termos de tempo de execução e Speedup.

O código dessa implementação está em anexo com esse documento no Apêndice G.

4.3 Análise do Comportamento dos Algoritmos

Nessa seção apresentamos a análise do comportamento dos algoritmos paralelos que utilizam paralelismo de controle. Como já mencionamos, temos que foram implementados algoritmos que utilizaram duas abordagens: uma que replica os vetores em todos os processos e; uma que utiliza um pipeline. Nas seções seguintes segue uma análise para cada uma dessas abordagens.

4.3.1 Replicação do Vetor

Nos algoritmos implementados utilizando a essa abordagem, a essência do programa é a mesma para todas as implementações realizadas. O que muda entre esses algoritmos é a forma com que a redução dos resultados de cada processo é realizado. Assim, primeiramente, vamos desconsiderar esse tempo de redução e analisar o restante do algoritmo.

Temos que todos os processos realizam a marcação de todos os números compostos até $\sqrt{n}$. Como todos realizam a mesma coisa ao mesmo tempo, não temos paralelismo, assim essa primeira parte é serial e pode ser representada pelo seguinte somatório:

$$\sum_{i=2}^{\sqrt{n}} \frac{\sqrt{n}}{i}$$

Dessa forma, para a resolução desse somatório, mais uma vez devemos utilizar a Série Harmônica de Primos [34, 19, 21]. Assim temos que a solução do somatório que representa a complexidade de tempo da parte serial do algoritmo algoritmo é dado por:

$$\sum_{i=2}^{\sqrt{n}} \frac{\sqrt{n}}{i} = \sqrt(n)(\ln\ln\sqrt{n})$$

Além disso existe o a parte paralela em que todos os algoritmos realizam suas marcações simultaneamente. Essa parte é representada pelo seguinte somatório:

$$\frac{1}{p}\sum_{i=2}^{\sqrt{n}} \frac{n-\sqrt{n}}{i}$$

Mais uma vez, utilizando a Série Harmônica de Primos [34, 19, 21] temos como solução desse somatório:

$$\frac{1}{p}\sum_{i=2}^{\sqrt{n}}\frac{n-\sqrt{n}}{i} = \frac{(n-\sqrt{n})\ln\ln(\sqrt{n})}{p}$$

Assim de forma simplificada temos que a complexidade desses algoritmos é dada por:

$$\theta(\sqrt{n}(\ln\ln\sqrt{n})) + \theta(\frac{(n-\sqrt{n})\ln\ln(\sqrt{n})}{p})$$

Assim temos que a ordem de complexidade pode ser simplificada por $\theta(\frac{(n)\ln\ln(n)}{p})$. No entanto, devemos lembrar que deixamos de considerar o tempo de redução dos resultados a um resultado único e final. Assim como na análise apresentada em 3.3 vamos tomar χ como o tempo de marcação dos elementos e considerando red o tempo de redução como sendo linearmente proporcional ao número de processos envolvidos, podemos apresentar uma análise mais detalhada desses algoritmos tomando o tempo estimado de execução como sendo:

$$Tempo(N,p) = \chi\sqrt{n}(\ln\ln\sqrt{n}) + \chi\frac{(n-\sqrt{n})\ln\ln(\sqrt{n})}{p} + red * p$$

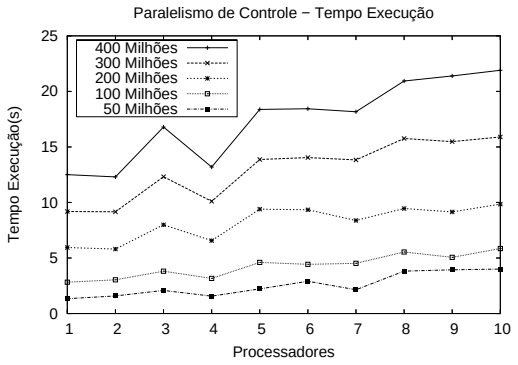
Realizamos alguns experimentos com esses algoritmos para valores de N iguais a 50 Milhões, 100 Milhões, 200 Milhões, 300 Milhões e 400 Milhões incrementando o número de processadores até 10. Para cada combinação $Algoritmo, N, p$ realizamos 5 execuções e os resultados apresentados nos gráficos da Figura 18 se referem a média dessas execuções.

O que podemos observar nos gráficos é que a medida que aumentamos o número de processadores o tempo gasto para executar as tarefas aumentam. Podemos observar também que quando consideramos apenas os números ímpares os tempos de execução são melhores, apesar de também aumentarem a medida que incrementamos o número de processadores. Além disso, podemos ver pelos gráficos que o tempo de execução para os métodos de redução único e distribuído em árvore foram praticamente iguais, com uma vantagem bem pequena para o método de redução distribuído em árvore.

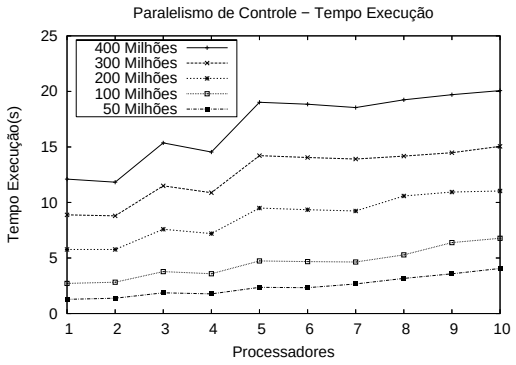
Com o objetivo de avaliar qual era o ponto de conteção das implementações feitas, realizamos os mesmos experimentos já descritos, no entanto desconsideramos o tempo de redução, ou seja, contabilizamos o tempo de execução somente até antes da redução dos valores a um valor único e final. Nossa suposição é que esse ponto de conteção se refere ao processo de redução, seja ele qual for. Assim, apresentamos nos gráficos da Figura 19 o resultados desses experimentos.

Observando os gráficos da Figura 19 temos o tempo de execução dos algoritmos desconsiderando o tempo de redução reduz-se significativamente a medida em que aumentamos o número de processadores, ou seja, nossa suposição a respeito do ponto de conteção está correta.

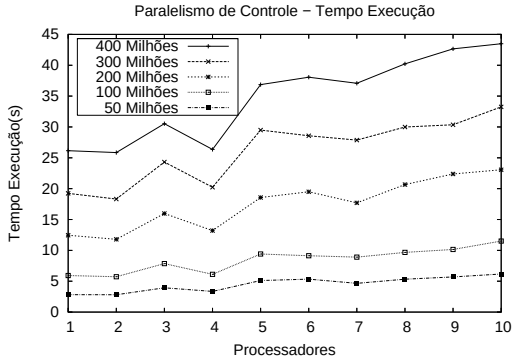
Essa contenção acontece porque o tempo de processamento, de marcação, é extremamente rápido e todos os processos realizam essa parte com extrema rapidez e ficam todos ociosos, parados a espera de uma redução. A redução, por sua vez,



(a) Valores Ímpares - Redução Única

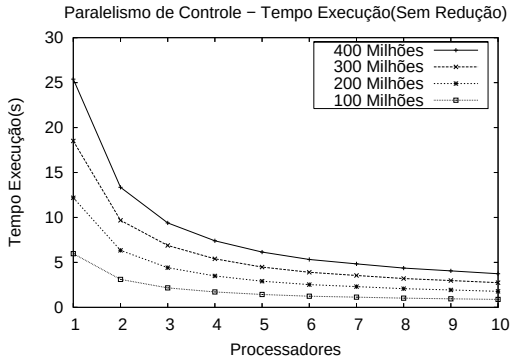


(b) Valores Ímpares - Redução Árvore

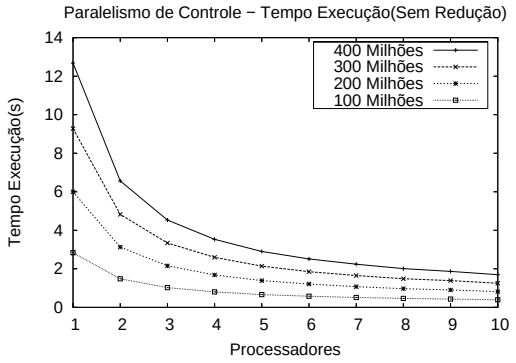


(c) Todos Valores - Redução Única

Figura 18: Tempos de Execução - Paralelismo de Controle - Replica Vetor



(a) Todos Valores

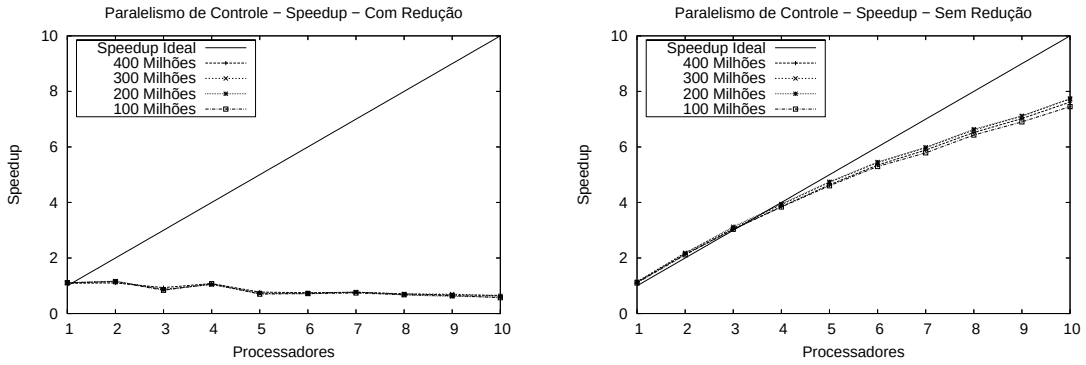


(b) Valores Ímpares

Figura 19: Tempos de Execução - Paralelismo de Controle - Sem Tempo de Redução

acontece através de transmissão de mensagens(a função MPI_Reduce é implementada internamente através de envios e recebimentos de mensagens) pela rede. Essa transmissão gera um *overhead* muito alto uma vez que o *delay* é alto. Com isso tempos que, quanto mais processadores envolvidos no processamento, mais e mais mensagens serão necessárias para a realização da redução e conseqüentemente o tempo total de execução também aumentará.

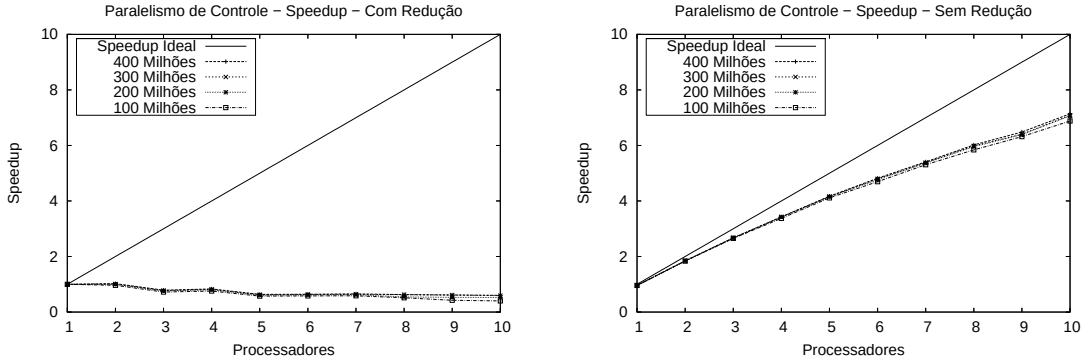
Para completar nossas análises sobre os algoritmos que utilizam uma redução no final, ou seja, que utilizam a replicação do vetor, apresentamos as curvas de Speedup para o mesmo. Para isso utilizamos os mesmos resultados obtidos nos



(a) Com Redução (b) Sem Redução

Figura 20: Speedup Controle - Replica Vetor - Todos Valores

experimentos anteriores. Para reforçar o quanto a redução dos valores ao final da execução prejudica o Speedup das implementações, apresentamos as curvas de Speedup considerando e não considerando o tempo de redução. Essa curvas podem ser observadas pelos gráficos das Figuras 20, 21. Os gráfico 20(a) e 21(a) são referentes ao Speedup considerando o tempo e os gráficos 20(b) e 21(b) não consideram.



(a) Com Redução (b) Sem Redução

Figura 21: Speedup Controle - Replica Vetor - Valores Ímpares

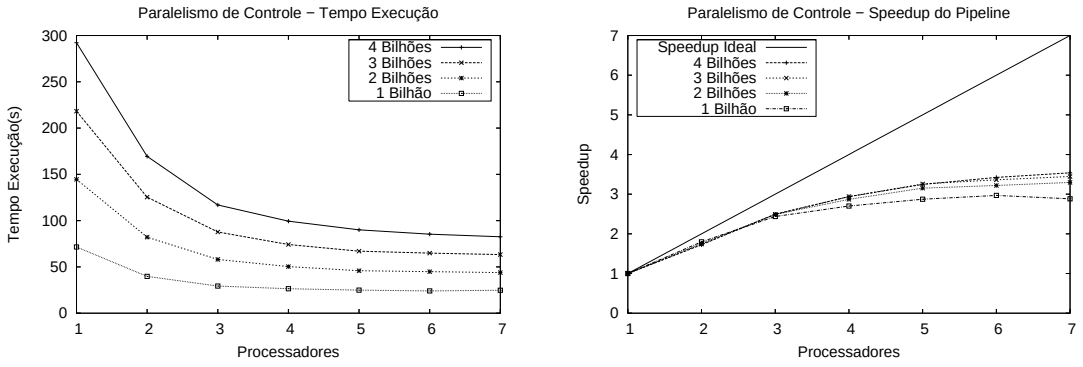
O Speedup baixo dessas implementações nos motivou a realizar a implementação descrita na seção 4.2. Sabendo que o ponto de contenção era a redução do total de primos encontrados ao final da execução, o que propomos foi a redução do resultados a medida em que outras marcações são realizadas.

Como já mencionamos, depois que todos os processos varrem o vetor até $\sqrt{n}$, o restante dos dados é quebrado em blocos pelo processo 0 e os mesmos são enviados ao processo 1 que marca e repassa para o processo consecutivo e assim por diante. Assim, dois pontos importantíssimos que são chave de nosso algoritmo são o tamanho dos blocos e o número de blocos consecutivos no pipeline. O tamanho do bloco é importante uma vez que, com blocos pequenos, assim como acontecia nos algoritmos com paralelismo de dados, o tempo de marcação de cada bloco é feita de modo muito rápida pelos processos. Assim, com blocos pequenos, todos os processos realizam suas marcações rapidamente e ficam ao final aguardando a redução dos resultados em um único resultado, e como já sabemos, isso gera um ponto de contenção.

Além disso, o número de pacotes que circulam ao mesmo tempo devem ser tais que maximizem o uso do sistema, ou seja, o número de pacotes deve ser grande o suficiente para que todos os processos possam se manter ocupados o máximo de tempos de forma paralela, não permitindo que ocorra a sobreposição entre o tempo de *warm up* e *cold down*, reduzindo assim a ociosidade dos mesmos a mesmo tempo que os resultados parciais são computados. Assim temos que ter a preocupação de manter uma boa relação entre tamanho de bloco e número de pacotes.

Para avaliar essa nossa outra estratégia, realizamos alguns experimentos com esse algoritmo para valores de N iguais a 1 Bilhão, 2 Bilhões, 3 Bilhões e 4 Bilhões incrementando o número de processadores até 7. Para cada combinação (N, p) realizamos 5 execuções e os resultados considerados se referem a média dessas execuções. O tamanho dos blocos foi de 50 milhões cada, e o número de blocos circulando paralelamente no pipeline era o máximo possível, ou seja, $N/TamanhoBloco$.

A partir dos resultados dos experimentos realizados, plotamos a curva do tempo de execução medido do algoritmo em relação ao número de processadores. Essa curvas podem ser vistas no gráfico 22(a). Com esses mesmos dados, calculamos as curvas de Speedup do algoritmo e as mesmas são apresentadas no gráfico 22(b).



(a) Tempo de Execução (b) Speedup

Figura 22: Estratégia Pipeline

Observando o gráfico 22(a) temos que, para todos os valores de N, a medida que aumentamos o número de processadores aumenta o tempo de execução diminuiu. No entanto, a medida em que o números de processadores aumenta muito essa redução no tempo tende a ser menor. Observando as curvas de Speedup, temos que para os menores valores de N o Speedup é bem menor. Isso acontece uma vez que, nesse caso, como o valor de N é pequeno, o tamanho dos blocos que circulam pelo pipeline também é pequeno. Conforme apresentamos na análise de complexidade de tempo do algoritmo serial, o tempo de processamento desses algoritmos é muito pequeno, assim, com blocos menores, a marcação é muito rápida e com isso os processos ficam mais sensíveis ao *overhead* gerado pela redução dos resultados parciais(processam rápido e ficam parados esperando). Ou seja, não consegue-se explorar ao máximo a potência do pipeline.

Ainda observando as curvas de Speedup, temos que a medida que aumentamos o valor de N temos que o Speedup também aumenta, uma vez que o sistema como um todo fica menos ocioso, pois cada bloco é maior e com isso o processamento é mais intenso em cada processador e os mesmos ficam menos ociosos a mesmo tempo

que os resultados parciais são computados, reduzindo assim o tempo de espera no final.

Entretanto, a medida em que aumentamos o número de processadores o Speedup tende a ser menor, diminuir. Isso acontece porque a medida em que aumentamos o número de processadores, a quantidade de blocos que circula pelo pipeline tende a ser menor proporcionalmente ao número de processos. Com isso, os processos ficam menos tempo trabalhando de forma paralela, aumentando a ociosidade do sistema como um todo. Além disso, aumenta-se a chance de ocorrer a sobreposição entre o tempo de *warm up* e *cold down* e esse tempos se tornam maiores também, o que diminui o uso eficiente do sistema.

Assim, nessa seção apresentamos não só uma análise de complexidade e avaliação de Speedup, mas também uma análise rigorosa do comportamento dos algoritmos utilizando paralelismo de controle comparando os resultados analíticos com os resultados experimentais.

4.4 Exemplo de Funcionamento

Para mostrar o funcionamento dos programa implementados, apresentamos a seguir alguns dos testes realizados, com diferentes valores de N.

- Primos encontrados para $N = 1000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file
./crivo_controle_CP.e -n 1000000 -o saida
Saída:
Quantidade de primos em 1000000 inteiros é 78498
```

- Primos encontrados para $N = 10000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file
./crivo_controle_SP_unica.e -n 10000000 -o saida
Saída:
Quantidade de primos em 10000000 inteiros é 664579
```

- Primos encontrados para $N = 30000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file
./crivo_controle_SP_arvore.e -n 30000000 -o saida
Saída:
Quantidade de primos em 30000000 inteiros é 1857859
```

- Primos encontrados para $N = 200000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file
```

```
./crivo_controle_SP_pipeline.e -n 200000000 -o saida
Saída:
Quantidade de primos em 200000000 inteiros é 11078937
```

- Primos encontrados para $N = 300000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file
./crivo_controle_SP_pipeline.e -n 300000000 -o saida
Saída:
Quantidade de primos em 300000000 inteiros é 16252325
```

- Primos encontrados para $N = 400000000$:

```
Execução:
mpirun -np 2 -machinefile machine_file
./crivo_controle_SP_arvore.e -n 400000000 -o saida
Saída:
Quantidade de primos em 400000000 inteiros é 21336326
```

5 Trabalhos Relacionados

Nessa seção apresentamos alguns trabalhos relacionados a descoberta de primos além de algoritmos de determinação de primalidade de um número inteiro qualquer. A geração de números primos e uma parte essencial para criação de chaves de criptografia. Existem diversas pesquisas na área de criptografia que utilizam crivos para encontrar primos e a partir desses primos determinar chaves para criptografia cujo a decompisição das mesmas levaria anos ou até mesmo séculos para serem quebradas [11].

Existem vários matemáticos que realizam pesquisas em busca do maiores primos existentes, existindo inclusive uma premiação de 1 milhão de dolares toda vez que um primo é descoberto [28]. Várias técnicas de paralelismo são utilizadas em grids, clusters, etc e varias técnicas são aplicadas. Atualmente, o maior primo foi encontrado em dezembro de 2005 por Dr. Curtis Cooper e Dr. Steven Boone, professores da Central Missouri State University, e o valor é $2^{30,402,457} - 1$.

A quebra de chaves de criptografia consistem basicamente em realizar decomposições de números e encontrar quais os primos que compõe as mesmas. Atualmente, existem organizações expecializadas que realizam esse tipo de processamento utilizando um grid formado por computadores pessoais [6]. Em [29] os autores discutem alguns dos mais recentes algoritmos de fatoração de inteiros através de números primos e como rede de computadores como clusters e Grids podem ser usados para executar esses algoritmos rapidamente. Esses trabalhos auxiliam na verificação da qualidade de chaves de criptografia.

Existem diversas propostas de algoritmos que avaliam a primalidade de números inteiros. Esses algoritmos também ajudam a determinar a qualidade de primos que podem ser utilizados para a construção de chaves. O artigo [13] apresenta um algoritmo probabilístico polinomial para testar a primalidade de um inteiro utilizando curvas elípticas. Outras diversas propostas de avaliação de primmalidade também existem como [12, 24] entre várias outras. Em [4] o autor compara os 21 métodos mais populares até 2004 que distinguem números primos de números compostos, e atualmente o algoritmo mais rápido que testa a primalidade de um primo foi apresentado por [1, 33] a 4 anos atrás e ainda não foi superado.

Apresentados os artigos que mostram a aplicabilidade no uso de números primos, apresentamos brevemente a seguir uma coleção considerável de propostas de algoritmos paralelos para a descoberta de primos. Algoritmos paralelos para a descoberta de primos já vêm sendo estudos a algum tempo [7]. Para o próprio Crivo de Eratóstenes, propostas diferentes de implementação foram dadas, como a apresentada em [20]. No trabalho [32] os autores apresentam uma proposta de paralização do Crivo de Eratóstenes utilizando multiprocessadores hypercúbico com um Speedup quase linear. Em [5] apresenta-se um algoritmo paralelo utilizando paralelismo de controle muito rápido para o crivo de Eratóstenes, no entanto utilizando memória compartilhada.

Outras porpostas também vem sendo realizadas, como em [31] onde um algoritmo paralelo para procura de primos que prioriza a economiza de espaço é apresentando. Em [23] encontramos um nova proposta de um algoritmo paralelo para encontrar todos os numeros primos em um intervalo [2..n] baseado em SMER(Scheduling by Multiple Edge Reversal). Um trabalho recente [22], de 2005, apresenta a primeira abordagem de paralelismo para o Crivo da Roda(*Wheel Sieve*). E finalmente em

[30] encontramos dois algoritmos paralelos muito rápidos para procura de primos.

Assim o que podemos concluir é que a procura de primos, apesar de ter sido iniciada a muito tempo atrás, ainda é uma linha de pesquisa muito forte nos dias de hoje. Atualmente não só matemáticos tem a a necessidade de encontrar esses primos, mas também pesquisadores da área de computação, pois são de suma impartância para a criação de chaves de criptografia. Existem várias propostas de algoritmos paralelos para esse problema, dos quais apresentamos alguns nessa seção, e ainda sim novas propostas estão surgindo e continuarão a surgir dado ao crescimento da necessidade desses valores para a criação de chaves seguras.

6 Conclusões

Nesse trabalho realizamos várias implementações para o problema do Crivo de Eratóstenes, ora utilizando paralelismo de dados, ora utilizando paralelismo de controle, todas em memória distribuída. Nessas implementações avaliamos diversas estruturas do MPI. Como resultado de nossas análises tivemos que as implementações que utilizaram paralelismo de dados conseguiram um bom Speedup, tanto a que utilizava todos os valores quanto a que utilizava apenas valores ímpares. Já com as implementações que utilizaram o paralelismo de controle não obtivemos o mesmo sucesso, todas elas não conseguiram um Speedup, com exceção de uma delas na qual utilizamos um pipeline onde as marcações eram feitas a medida ao mesmo tempo que os resultados parciais eram computados. Nessa última conseguimos um bom Speedup, mas não tão bom quanto o que conseguimos com o paralelismo de dados. Nossa análise do comportamento desses algoritmos foi bastante rigorosa na qual os resultados analíticos foram comparados com os os resultados experimentais.

Quando se trata de aplicações com computação intensa e com poucos estados de computação, em geral, temos que uma boa prática de paralelismo é o de controle. Em contrapartida, temos que aplicações cujo a computação não é tão intensa mas o número de estados da mesma é alto, uma boa prática de programação paralela é utilizar o paralelismo de dados. Ou seja, ao paralelizarmos uma aplicação devemos sempre ter em mente a preocupação com a relação processamento e quantidade de estados.

O Crivo de Eratóstenes é uma aplicação cujo a computação exigida é baixa, pois apenas marcações são feitas, no entanto temos que a mesmas possui vários estados que são os estados intermédiarios do vetor. Assim, para essa aplicações, a melhor solução é uma implementação que explore o paralelismo de dados, como podemos confirma pelo Speedup alcançado com nossa implementação. Já alcançar um bom Speedup utilizando paralelismo de controle não é uma tarefa trivial, como observamos com nossos resultados. Mesmo assim ainda foi possível realizar uma implementação com um Speedup razoável.

7 Agradecimentos

Agradeço ao grupo de estudos composto pelos alunos Alex Borges, Elisa Tuler, Guilherme Trielli, Leonardo Chaves, Marcelo Maia, Renata Braga, Bruno Coutinho, entre outros pela ajuda e discussões sobre os resultados e decisões de implementação que contribuíram positivamente para um melhor entendimento de todo trabalho.

Referências

[1] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p, 2002.

[2] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[3] R. H. Arpaci, A. C. Dusseau, A. H. Vahday, L. T. Liu, T. E. Anderson, and D. A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–78, Ottawa, Canada, May 1995.

[4] Daniel J. Bernstein. Distinguishing prime numbers from composite numbers: the state of the art in 2004. URL: `http://cr.yp.to/papers.html#prime2004`. Note: submitted.

[5] W. Robert Collins. Tasking solutions to the sieve of eratosthenes. *Ada Lett.*, XVIII(4):107–110, 1998.

[6] Distributed.net. Distributed computing technologies inc., 2005. http://www.distributed.net/.

[7] K. B. Dunham. Algorithm 372: An algorithm to produce complex primes, csieve [a1]. *Commun. ACM*, 13(1):52–53, 1970.

[8] Eratóstenes. Crivo de eratóstenes, 2006. http://pt.wikipedia.org/wiki/Erat

[9] John Fletcher et.al. Gnuplot, 2005. http://www.gnuplot.info/index.html.

[10] M. Flynn. Some computer organizations and their effectiveness. C-21:948–960, 1972.

[11] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed rsa-key generation. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 663–672, New York, NY, USA, 1998. ACM Press.

[12] S Goldwasser and J Kilian. Almost all primes can be quickly certified. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 316–329, New York, NY, USA, 1986. ACM Press.

[13] Shafi Goldwasser and Joe Kilian. Primality testing using elliptic curves. *J. ACM*, 46(4):450–472, 1999.

[14] W. Gropp and E. Lusk. The MPI communication library: its design and a portable implementation. In *Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi*, pages 160–165, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[15] W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.

[16] W. Gropp and E. Lusk. Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):103–114, Summer 1997.

[17] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[18] William D. Gropp and Ewing Lusk. *Installation Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/5.

[19] G. H. Hardy and E. M Wright. *An Introduction to the Theory of Numbers.* Clarendon Press, 1979.

[20] Xuedong Luo. A practical sieve algorithm finding prime numbers. *Commun. ACM*, 32(3):344–346, 1989.

[21] Trygve Nagell. *Introduction to Number Theory.* Wiley, 1951.

[22] Gabriel Paillard. A fully distributed prime numbers generation using the wheel sieve. In Thomas Fahringer and M. H. Hamza, editors, *Parallel and Distributed Computing and Networks*, pages 651–656. IASTED/ACTA Press, 2005.

[23] Gabriel Paillard, Christian Lavault, and Felipe Franca. A distributed prime sieving algorithm based on scheduling by multiple edge reversal. In *ISPDC '05: Proceedings of the The 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)*, pages 139–146, Washington, DC, USA, 2005. IEEE Computer Society.

[24] János Pintz, William Steiger, and Endre Szemerédi. Two infinite sets of primes with fast primality tests. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 504–509, New York, NY, USA, 1988. ACM Press.

[25] Paul Pritchard. Fast compact prime number sieves (among others). *J. Algorithms*, 4(4):332–344, 1983.

[26] Michael J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, 1994.

[27] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill, 2004.

[28] The Great Internet Mersenne Prime Search. Mersenne prime search, 2005. http://mersenne.org/.

[29] Robert D. Silverman. Massively distributed computing and factoring large integers. *Commun. ACM*, 34(11):95–103, 1991.

[30] Jonathan Sorenson and Ian Parberry. Two fast parallel prime number sieves. *Inf. Comput.*, 114(1):115–130, 1994.

[31] Jonathan P. Sorenson. Trading time for space in prime number sieves. *Lecture Notes in Computer Science*, 1423:179–??, 1998.

[32] R. L. Wainwright. Parallel sieve algorithms on a hypercube multiprocessor. In *CSC '89: Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 232–238, New York, NY, USA, 1989. ACM Press.

[33] Eric W Weisstein. Aks primality test, 2005. http://mathworld.wolfram.com/AKSPrimalityTest.html.

[34] Eric W Weisstein. Harmonic series of primes, 2005. http://mathworld.wolfram.com/HarmonicSeriesofPrimes.html.