Universidade Federal de Minas Gerais

Departamento de Ciência da Computação

Projeto e Análise de Algoritmos

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

Autor: Fernando Antonio Fernandes Júnior

Monitor: Fabiano Cupertino Botelho

Professor: Nivio Ziviani

Belo Horizonte
28 de junho de 2006

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

1 Introdução

Nesse trabalho será realizado um estudo do Crivo de Eratóstenes, um algoritmo clássico para encontrar todos os números primos menores ou iguais a um número inteiro positivo n. O algoritmo é estudado sob o ponto de vista da programação paralela em uma máquina MIMD do tipo NOW – *Network of Worksations* [1].

A seção 2 apresenta uma descrição da abordagem utilizada pelo Crivo de Eratóstenes. A subseção 2.1 apresenta motivações para o estudo do problema, além de possibilidades de aplicação do algoritmo.

Na seção 3 são listadas soluções para o problema. Inicialmente o algoritmo serial é apresentado na subseção 3.1. Em seguida são apresentadas soluções onde algoritmos paralelos são propostos, explorando as diversas possibilidades de paralelização do crivo. A subseção 3.2 introduz um algoritmo para resolver o problema utilizando paralelismo de dados. Na subseção 3.3 um algoritmo baseado em paralelismo de controle é apresentado. Uma terceira solução, que utiliza princípios do paralelismo de controle e de dados, é apresentada na subseção 3.4. Para cada proposta de solução, o algoritmo é descrito, sua complexidade é analisada e a maneira como foi implementada é explicada brevemente.

A partir das implementações das soluções, foram realizados diversos experimentos. As metodologias utilizadas e os resultados desses experimentos são apresentados na seção 4. A subseção 4.1 mostra o comportamento do algoritmo serial. Na subseção 4.2 é feita uma análise do *speed-up* obtido a partir de cada uma das soluções paralelas implementadas.

2 Crivo de Eratóstenes

O Crivo de Eratóstenes (276 A.C. - 194 A.C.) [9] é um método simples e prático de se encontrar números primos até um certo valor limite n. O algoritmo começa com a lista de números naturais de 2 até n. A cada passo, o menor fator primo (ainda não utilizado) p é escolhido e todos os múltiplos de p são removidos da lista. O algoritmo termina quando todos os fatores primos menores ou iguais a $\sqrt{n}$ tenham sido utilizados

As Figuras 1 e 2 exemplificam o funcionamento do crivo para $n = 10$. A primeira mostra a lista de números naturais de 2 até 10, antes que qualquer número seja marcado.

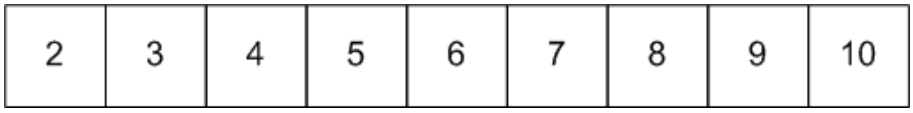

Figura 1: Lista de números antes do início da execução do Crivo de Eratóstenes

A Figura 2 mostra a mesma lista após a realização do crivo. Como

$$\sqrt{n} = \sqrt{10} \approx 3,16$$

não é um número inteiro, o valor considerado é

$$\lfloor \sqrt{n} \rfloor = \lfloor \sqrt{10} \rfloor = 3$$

Assim, os fatores primos utilizados para cortar os números compostos são aqueles menores ou iguais a 3, ou seja, 2 e 3. Na Figura 2, os valores cortados por serem múltiplos de 2 são superpostos por traços como /, enquanto aqueles cortados por serem múltiplos de 3 estão superpostos por traços com a direção perpendicular a /. Como 6 é múltiplo de ambos os fatores, ele se apresenta superposto pelos dois padrões.

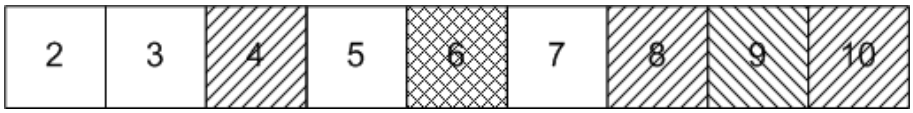

Figura 2: Lista de números após o fim da execução do Crivo de Eratóstenes

Conforme pode-se notar na Figura 2, os valores não marcados correspondem aos números primos menores ou iguais a 10: 2, 3, 5 e 7.

2.1 Motivação e Aplicações Práticas

O Crivo de Eratóstenes é um método para descobrir todos os números primos menores ou igual a um inteiro positivo n. O estudo de números primos foi considerado, por muito tempo, um exemplo de matemática pura, sem aplicações fora do próprio tema. Porém, em meados da década de 1970, essa visão foi abandonada

quando foi anunciado que os números primos podem ser usados como a base para a criação de chaves públicas em algoritmos de criptografia. Além disso, números primos são muito importantes na criação de tabelas *hash*.

Um dos algoritmos de criptografia por chave pública mais utilizados atualmente é o RSA [8]. Ele foi o primeiro algoritmo criado para esse fim que suportava tanto assinatura digital quanto criptografia. Seu funcionamento envolve duas chaves, uma pública e outra privada. As mensagens só podem ser decriptadas utilizando a chave privada. Em outras palavras, qualquer pessoa é capaz de encriptar uma mensagem, mas apenas o detentor da chave privada é capaz de decriptá-la e interpretá-la.

A segurança do RSA está baseada, entre outros fatores, na dificuldade de se encontrar os fatores primos de um número muito grande (1024-2048 bits). Isso impede que a chave privada seja descoberta a partir da pública.

Outra aplicação importante para os números primos é a construção de tabelas *hash* [6]. É muito comum escolher um tamanho da tabela que seja primo. Isso é feito para evitar que valores altos utilizados para endereçar a tabela tenham divisores em comum com o tamanho da tabela, o que poderia induzir um alto número de colisões após a operação de módulo.

3 Soluções Apresentadas

Nessa seção será realizado um estudo do algoritmo conhecido como Crivo de Eratóstenes. Inicialmente será estudada sua versão serial (ou seqüencial). Em seguida serão exploradas as possibilidades de paralelização do algoritmo através da proposta de três algoritmos paralelos: um utilizando paralelismo de dados, outro utilizando paralelismo de controle e um terceiro que procura mesclar as duas abordagens.

3.1 Algoritmo Serial

Conforme foi explicado na seção 2, o Crivo de Eratóstenes encontra todos os números primos menores ou iguais a um inteiro positivo n. Para tal, ele parte de uma lista de números naturais de 2 até n e, para cada primo p tal que

$$2 \leq p \leq \lfloor \sqrt{n} \rfloor$$

exclui os múltiplos de p.
O método é descrito em pseudo-código no Algoritmo 1.

Algoritmo 1 Crivo de Eratóstenes Serial

```
{Inicializa a lista}
for i ← 2 até n do
  Lista_i ← Desmarcado
end for
p ← 2
while p^2 ≤ n do
  {Marca os múltiplos de p}
  k ← p^2
  while k ≤ n do
    Lista_k ← Marcado
    k ← k + p
  end while
  {Escolhe o próximo fator primo p}
  while Lista_p ≠ Desmarcado do
    p ← p + 1
  end while
end while
{Imprime a lista}
for i ← 2 até n do
  if Lista_i = Desmarcado then
    print i
  end if
end for
```

3.1.1 Análise de Complexidade

A análise de complexidade do algoritmo levará em conta o limite superior n da lista de primos.

- **Tempo**

O Algoritmo 1 mostra que o Crivo de Eratóstenes é composto por três laços. O primeiro percorre toda a lista desmarcando todos os seus $(n-2)+1 = n-1$ elementos. Logo a complexidade de tempo desse laço é $n-1$. Utilizando a notação O:

$$n - 1 = O(n)$$

O segundo laço requer uma análise mais cuidadosa. Para cada fator primo p, a lista deve ser percorrida de p^2 até n, incrementando o iterador de p. Assim o tempo de execução desse laço **para cada fator primo** p **é**:

$$(n - p^2 + 1)/p$$

utilizando a notação O é simplesmente

$$(n - p^2 + 1)/p = O(n)$$

Para finalizar a análise do segundo laço, resta saber quantos fatores primos são utilizados para marcar elementos na lista. Como foi mostrado na descrição do algoritmo, são utilizados todos os números primos menores ou igual a $\sqrt{n}$. A quantidade de tais fatores primos é dada pela função de contagem de primos $\pi(x)$ [7], conjecturada por Gauss e Legendre. A função de contagem de primos π pode ser aproximada por [7]

$$\pi(x) = \Theta\left(\frac{x}{\log x}\right)$$

Entretanto, em [7] é mostrado que a razão $x/\pi(x)$ é aproximadamente constante, mesmo para valores muito altos de x, o que sugere que uma boa aproximação para $\pi(x)$ é

$$\pi(x) = O(\log x)$$

Assim tem-se que o segundo laço é executado aproximadamente

$$\pi(\sqrt{n}) = O(\log \sqrt{n})$$

vezes. Utilizando a aproximação $\sqrt{n} = O(\log n)$, tem-se que o segundo laço é executado

$$\pi(\log n) = O(\log \log n)$$

vezes. Como a complexidade **para cada fator primo** p **é** $O(n)$, a complexidade final do segundo laço é

$$n \cdot O(\log \log n) = O(n \cdot \log \log n)$$

O terceiro laço percorre novamente toda a lista investigando quais elementos não foram marcados, e, portanto, são números primos. Assim como o primeiro laço, a complexidade de tempo é, utilizando a notação O:

$$n - 1 = O(n)$$

Como agora já é conhecida a complexidade de cada laço, a complexidade final do algoritmo é dada por

$$\begin{aligned} T(n) &= O(n) + O(n \cdot \log\log n) + O(n) \\ &= \max(O(n), O(n \cdot \log\log n), O(n)) \\ &= O(n \cdot \log\log n) \end{aligned}$$

Portanto a complexidade de tempo $T(n)$ do Crivo de Eratóstenes no **pior caso**, **melhor caso** e **caso médio** é

$$\boxed{T(n) = O(n \cdot \log\log n)}$$

- **Espaço**

O algoritmo requer que, para cada um dos $n - 2 + 1 = n - 1$ membros da lista, seja armazenado seu estado (marcado ou desmarcado). Assim a complexidade de espaço $E(n)$ do algoritmo no **pior caso**, **melhor caso** e **caso médio** é

$$E(n) = n - 1$$

utilizando a notação O:

$$\boxed{E(n) = O(n)}$$

3.1.2 Implementação

O Algoritmo 1 foi implementado em linguagem C através da função

```
unsigned long crivo(unsigned long n)
```

Essa função faz parte do arquivo `serial.c`.

O único parâmetro dessa função é o valor limite n conforme estabelecido na descrição do algoritmo. O valor de retorno é a quantidade de números primos menores ou iguais a n.

Para armazenar o estado de cada elemento da lista foi alocado um arranjo contendo $n - 1$ elementos do tipo `char`. A alocação é feita dinamicamente para evitar a alocação desnecessária de recursos.

Foi criado um pequeno programa chamado `serial.e` que realiza a chamada dessa função. O valor de n é passado como argumento na linha de comando. Abaixo é mostrado um exemplo de utilização:

```
> ./serial.e 500000000
```

O programa imprime na saída padrão a quantidade de números primos menores ou iguais a n, como mostrado abaixo:

```
26355867
```

A listagem completa do código-fonte pode ser encontrada no Apêndice A.

3.2 Paralelismo de Dados

O paralelismo de dados proposto para o Crivo de Eratóstenes consiste em atribuir, para cada processo, uma parcela da lista de números. Assim, cada processo seria responsável por marcar (riscar da lista) os múltiplos, de todos os fatores primos, em sua parcela da lista.

A primeira decisão a ser tomada é como distribuir a lista de n números entre os p processos. É conveniente que cada processo seja responsável por um bloco contíguo da lista, ao invés de posições intercaladas, uma vez que isso favorece o balanceamento de carga [4].

Além disso, a quantidade de elementos que cada processo receberá também requer alguns cuidados. Caso n seja múltiplo de p, cada processo fica responsável por exatamente n/p posições da lista, caso contrário é impossível distribuir a lista de maneira completamente balanceada. A distribuição adotada nesse trabalho é baseada em dois princípios:

- Nenhum processo deve ser responsável por mais que $\lceil n/p \rceil$ ou menos que $\lfloor n/p \rfloor$ elementos. Isso garante que a diferença entre as listas de cada processo seja de, no máximo, 1 elemento;

- Os processos responsáveis por um número maior de elementos não devem estar concentrados em algum conjunto específico, como, por exemplo, os $n \mod p$ primeiros processos. Essa distribuição deve ser uniforme entre o conjunto de processos.

Assim, a divisão dos elementos entre os processos pode ser dada pelas seguintes relações;

O elemento inicial da sub-lista alocada para o processo i é dado por:

$$I(i) = \lfloor in/p \rfloor$$

O elemento final da sub-lista designada para o processo i é o elemento imediatamente anterior ao início da sub-lista do processo $i + 1$:

$$F(i) = \lfloor (i + 1)n/p \rfloor - 1$$

O processo responsável pelo elemento na posição j é:

$$P(j) = \lfloor (p(j + 1) - 1)/n \rfloor$$

Essa forma de divisão de elementos entre os processos traz ainda outra vantagem. Garantindo que o primeiro processo (processo raiz) seja responsável pelos $\lfloor \sqrt{n} \rfloor$ primeiros números, ou seja $\lfloor n/p \rfloor \geq \lfloor \sqrt{n} \rfloor$, apenas ele fica responsável pela determinação de todos os fatores primos utilizados para cortar elementos da lista. Isso faz com que a determinação do próximo fator primo não necessite de nenhuma comunicação entre os processos já que o processador raiz controla toda essa parcela da lista.

Portanto, o algoritmo paralelo pode ser descrito nos passos seguintes. O processo 0 descobre o próximo fator primo e o envia através de um *broadcast* para todos os processos. Conhecido o fator, todos os processos (inclusive o 0) marcam os múltiplos

desse primo na parte da lista que lhe foi atribuída. Quando o fator primo for maior que $\sqrt{n}$, todos os processos contam a quantidade de primos encontrados na sua parcela da lista e os valores parciais de cada processo são totalizados no processo 0, por meio de uma operação de redução. Todas as operações de entrada e saída ficam a cargo do processo 0.

A Figura 3 apresenta um grafo que mostra a relação entre os processos durante a execução do algoritmo.

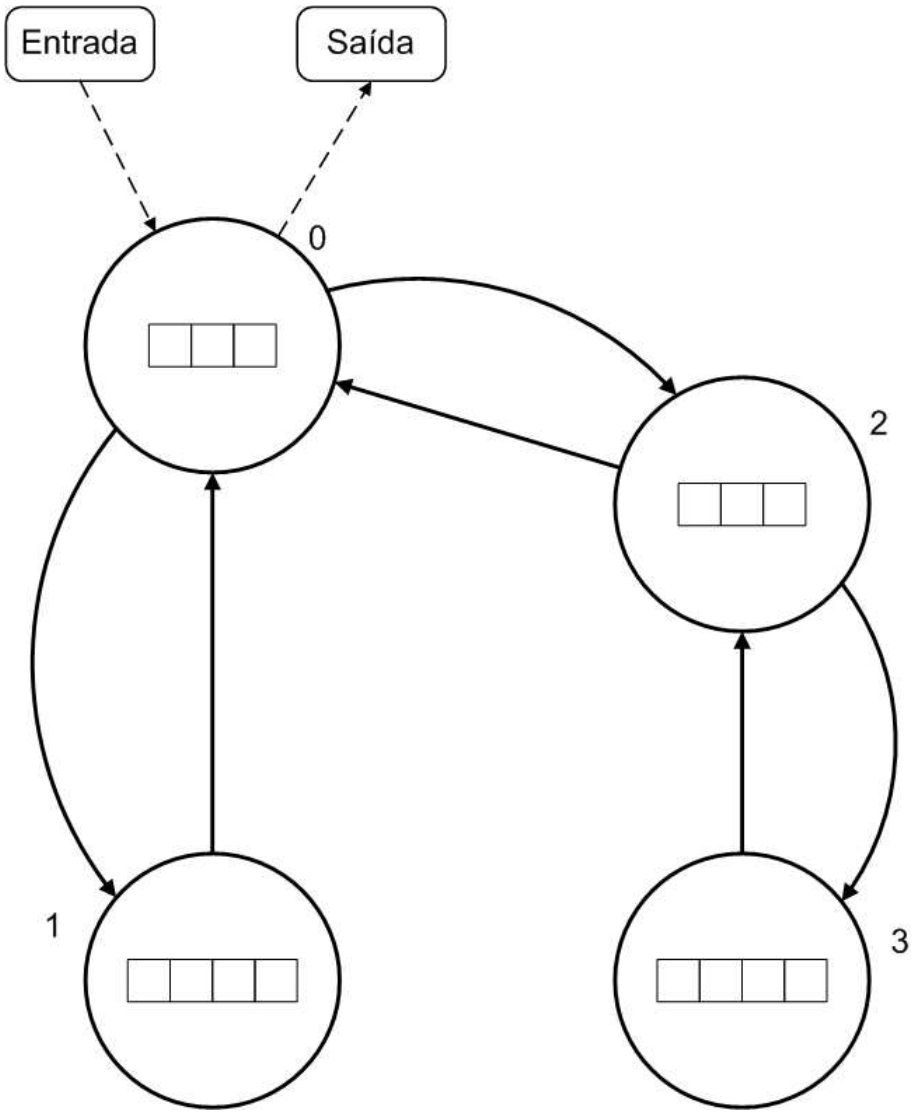

Figura 3: Grafo de processos para o paralelismo de dados. Linhas curvas representam a comunicação relativa aos *broadcasts* enquanto linhas retas representam a comunicação relativa à redução. Linhas pontilhadas representam entrada e saída.

O Algoritmo 2 descreve em pseudo-código a solução proposta. A sub-rotina Broadcast(Fator, 0) envia o valor armazenado na variável Fator do processo 0 para a variável fator dos demais processos. Na chamada da sub-rotina Redução(TotalPrimos, 0, SOMAR), os valores da variável TotalPrimos em todos os processos são somados e armazenados nessa variável do processo 0. Essa etapa produz o total geral de

primos, somando os totais parciais de cada processo.

3.2.1 Análise de Complexidade

A análise de complexidade do algoritmo paralelo levará em conta o limite superior n da lista de primos e número de processos p.

- **Tempo**

Conforme foi mostrado na seção 3.1.1, a complexidade para marcar todos os fatores primos na lista inteira contendo n elementos é

$$O(n \log \log n)$$

Porém, no algoritmo paralelo, esse trabalho é dividido de maneira aproximadamente igual entre os p processos. Logo a complexidade dessa etapa passa a ser

$$O((n \log \log n)/p)$$

Além disso, no laço principal (onde o fator primo é atualizado a cada iteração) há uma operação de *broadcast*. Essa operação tem tempo de execução [4]

$$O(\log p)$$

Essa operação é executada sempre que um novo fator primo selecionado para cortar elementos da lista. A quantidade de tais fatores é igual à quantidade de números primos menores que $\sqrt{n}$, que de acordo com a função de contagem de primos [7] é

$$\pi(\sqrt{n}) = \Theta(\sqrt{n}/\log \sqrt{n})$$

Portanto o custo de todos os *broadcasts* é

$$O((\sqrt{n}/\log \sqrt{n}) \cdot \log p)$$

Finalmente, o custo total do algoritmo para o **pior caso**, **melhor caso** e **caso médio** é

$$\boxed{T(n,p) = O(((n \log \log n)/p) + (\sqrt{n}/\log \sqrt{n}) \cdot \log p)}$$

Aplicando ainda propriedades da notação O, vemos que a etapa de marcação dos múltiplos domina assintoticamente o tempo de execução:

$$\begin{aligned} T(n,p) &= O(((n \log \log n)/p) + (\sqrt{n}/\log \sqrt{n}) \cdot \log p) \\ &= O(\max((n \log \log n)/p, (\sqrt{n}/\log \sqrt{n}) \cdot \log p) \\ &= O((n \log \log n)/p) \end{aligned}$$

Assim a complexidade final é:

Algoritmo 2 Crivo de Eratóstenes – Paralelismo de Dados

```
{Inicializa a lista}
InícioLista ← I(id)
FimLista ← F(id)
for i ← InícioLista até FimLista do
  Lista_i ← Desmarcado
end for
Fator← 2
while Fator^2 ≤ n do
  {Descobre qual o primeiro da sub-lista a ser cortado}
  if Fator^2 > InícioLista then
    k ← Fator^2− InícioLista
  else
    if InícioLista  mod  Fator = 0 then
      k ← 0
    else
      k ← Fator − (InícioLista  mod Fator)
    end if
  end if
  {Marca os múltiplos do Fator}
  k ← Fator^2
  while k ≤ FimLista do
    Lista_k ← Marcado
    k ← k + Fator
  end while
  {Processo 0 escolhe o próximo fator primo}
  if id = 0 then
    while Lista_Fator ≠ Desmarcado do
      Fator ← Fator + 1
    end while
  end if
  Broadcast(Fator, 0)
end while
{Totalização parcial dos primos}
TotalPrimos ← 0
for i ← InícioLista até FimLista do
  if Lista_i = Desmarcado then
    TotalPrimos ← TotalPrimos + 1
  end if
end for
{Totalização geral dos primos no processo 0}
Redução(TotalPrimos, 0, SOMAR)
{Processo 0 imprime o total de primos}
if id = 0 then
  print  TotalPrimos
end if
```

$$\boxed{T(n,p) = O((n \log\log n)/p)}$$

- **Espaço**

Mais uma vez, a quantidade de memória auxiliar corresponde ao arranjo alocado para armazenar o estado de cada elemento da lista de 2 até n. Entretanto, cada processo aloca apenas a parcela da lista sobre a qual ele irá operar. Logo cada processo precisa alocar, **pior caso**, **melhor caso** e **caso médio**, uma quantidade de memória

$$\boxed{E(n,p) = O(n/p)}$$

3.2.2 Implementação

A implementação do Crivo de Eratóstenes utilizando paralelismo de dados foi baseada no código-fonte listado em [4]. Foram realizadas adaptações para que o programa atendesse aos requisitos desse trabalho.

O Algoritmo 2 foi implementado em linguagem C no arquivo `dados.c` através da função

```
unsigned int crivo_paralelo_dados(unsigned int n, int id, int p)
```

O parâmetro `n` dessa função é o valor limite conforme estabelecido na descrição do algoritmo. O parâmetro `id` é o número do processo em execução, enquanto `p` é o número total de processos. O valor de retorno é a quantidade de números primos menores ou iguais a n.

A troca de mensagens entre os processos foi realizada utilizando a interface MPI (*Message Passing Interface*) [3].

Para armazenar o estado de cada elemento da lista, cada processo aloca um arranjo contendo aproximadamente $(n-1)/p$ elementos do tipo `char`. Esse arranjo é alocado dinamicamente para evitar desperdício de recursos.

A partir da implementação foi criado um programa chamado `dados.e` que realiza a chamada da função que implementa o algoritmo. O valor de n é passado como argumento na linha de comando. Além disso, para a utilização da interface MPI, o programa deve ser executado indiretamente, através do utilitário `mpirun`. Abaixo é mostrado um exemplo de utilização:

```
> mpirun -np 4 -machinefile lista_maquinas.txt ./dados.e 500000000
```

Conforme ilustra o exemplo, o utilitário pede que sejam passados o número de processos a serem disparados (precedido de `-np`) e a um arquivo texto contendo a lista de máquinas onde os processos serão executados (precedido de `-machinefile`).

O programa imprime a quantidade de números primos menores ou iguais a n na saída padrão da máquina onde é executado o processo 0, como mostrado abaixo:

```
26355867
```

A listagem completa do código-fonte pode ser consultada no Apêndice A.

3.3 Paralelismo de Controle

A segunda possibilidade de paralelismo do crivo explorada nesse trabalho é o paralelismo de controle. A idéia principal do algoritmo é que todos os processos trabalhem sobre toda a extensão da lista de números, mas cada um exclui os apenas múltiplos de alguns fatores primos.

Nessa abordagem, o algoritmo é visto como uma série de tarefas da seguinte forma: excluir da lista todos os múltiplos de 2; excluir da lista todos os múltiplos de 3; e assim por diante. Cada uma dessas tarefas será atribuída a um dos processos, que a executa sobre toda a lista de números.

Definidos esses padrões, o funcionamento do algoritmo será descrito. No primeiro passo cada processo identifica independentemente todos os fatores primos menores ou iguais a $\sqrt{n}$. Em seguida, cada processo utiliza $1/p$ fatores primos para excluir números da lista. Esse fatores primos serão distribuídos alternadamente, ou seja, processo 0 utiliza fator primo 2, processo 1 utiliza 3, e assim sucessivamente, voltando ao processo 0 após o último processo. Por fim, todas as listas são reunidas no processo 0 e aqueles números que não tenham sido excluídos por nenhum processo são os números primos.

A Figura 4 mostra um grafo que explica esquematicamente as interações entre os processos. É importante observar que, como cada algoritmo constrói independentemente sua lista de primos até $\sqrt{n}$, não há comunicação durante a exclusão de números da lista. Apenas o passo de redução, que acontece para reunir as listas de todos os processos, requer comunicação entre eles. A Figura mostra também que, mais uma vez, o processo 0 é responsável por todas as operações de entrada e saída do algoritmo.

O Algoritmo 3 descreve solução de paralelismo de controle proposta para o Crivo de Eratóstenes. A chamada da sub-rotina CrivoEratóstenes(Lista, $\lfloor\sqrt{n}\rfloor$) executa o crivo de de maneira seqüencial sobre a lista e até o valor máximo $\lfloor\sqrt{n}\rfloor$. Essa chamada representa o passo em que cada processo identifica independentemente a lista de primos até $\lfloor\sqrt{n}\rfloor$. Na chamada da sub-rotina Redução(Lista, 0, OU), as lista de todos os processos são reunidas na lista do processo 0, onde permanecem apenas os números que não foram marcados (excluídos) por nenhum dos processos.

3.3.1 Análise de Complexidade

Nessa seção será analisada a complexidade do Crivo de Eratóstenes explorando paralelismo de controle. Essa análise levará em conta duas variáveis: o valor limite n para busca de primos e o número de processadores p.

- **Tempo**

O algoritmo começa com cada processo inicializando sua lista de números composta por $n - 1$ elementos. A complexidade desse passo, utilizando a notação O é

$$n - 1 = O(n)$$

Em seguida, cada um dos processos constrói de maneira independente a lista de primos menores que $\sqrt{n}$. Essa lista é gerada executando o crivo de modo seqüencial até $\sqrt{n}$. De acordo com a seção 3.1.1, o tempo necessário para esse passo é

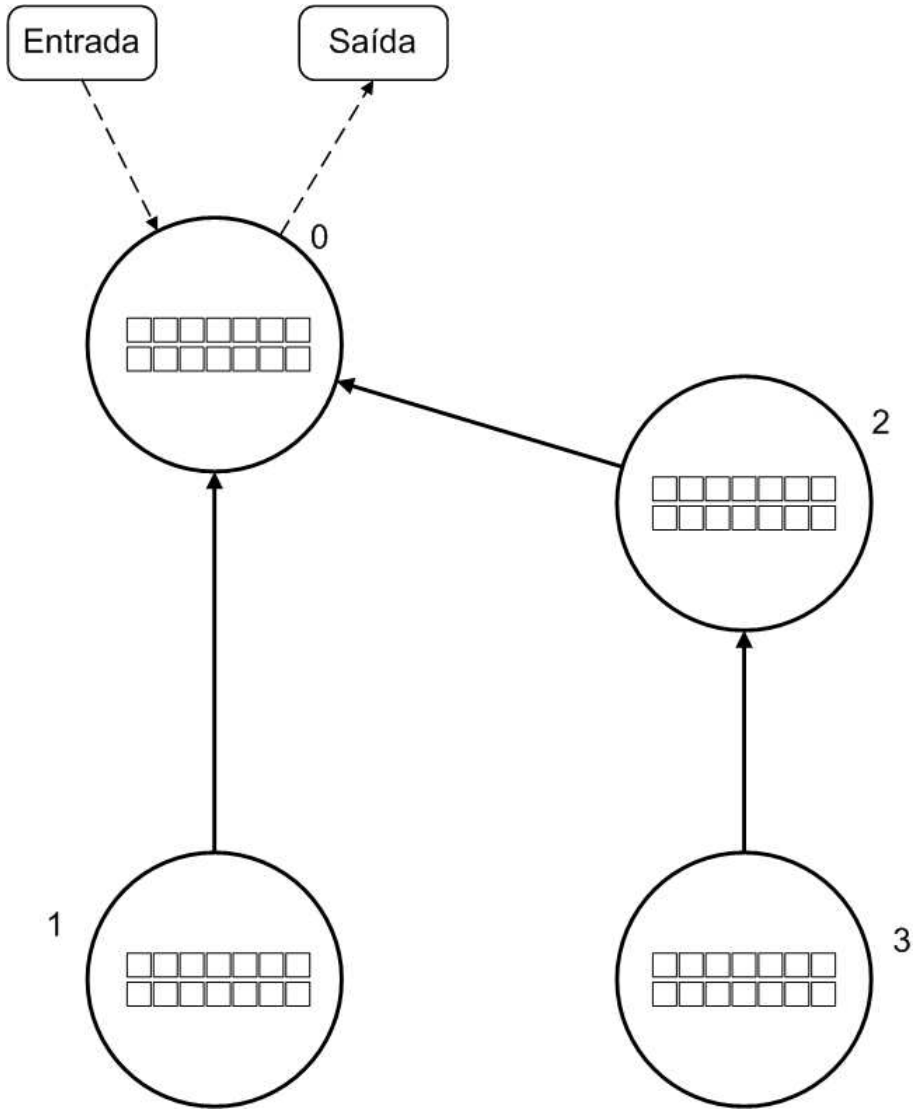

Figura 4: Grafo de processos para o paralelismo de controle. Linhas retas representam a comunicação relativa à redução. Linhas pontilhadas representam entrada e saída.

$$O(\sqrt{n} \cdot \log\log\sqrt{n})$$

Com a lista de fatores primos construída, cada processo exclui da lista os múltiplos dos fatores primos. Porém, cada processo utiliza apenas $1/p$ desses fatores para realizar a exclusão. Assim, utilizando o resultado apresentado na seção 3.1.1 a complexidade desse passo em cada processo é

$$O((n \cdot \log\log n)/p)$$

Portanto o custo total de execução do algoritmo é dado por

$$\boxed{T(n,p) = O(n) + O(\sqrt{n} \cdot \log\log\sqrt{n}) + O((n \cdot \log\log n)/p)}$$

Algoritmo 3 Crivo de Eratóstenes – Paralelismo de Controle

```
{Inicializa a lista}
for i ← 2 até n do
  Lista_i ← Desmarcado
end for
Lista ← CrivoEratóstenes(Lista, ⌊√n⌋)
p ← 2
OrdemPrimo ← 0
while p² ≤ n do
  while OrdemPrimo mod p ≠ id do
    {Rejeita primos que não estão atribuídos a ele}
    {Escolhe o próximo fator primo p}
    while Lista_p ≠ Desmarcado do
      p ← p + 1
    end while
    OrdemPrimo ← OrdemPrimo + 1
  end while
  {Marca os múltiplos de p}
  k ← p²
  while k ≤ n do
    Lista_k ← Marcado
    k ← k + p
  end while
end while
{Redução das listas no processo 0}
Redução(Lista, 0, OU)
{Processo 0 imprime o total de primos}
if id = 0 then
  TotalPrimos ← 0
  for i ← 2 até n do
    if Lista_i = Desmarcado then
      TotalPrimos ← TotalPrimos + 1
    end if
  end for
  print TotalPrimos
end if
```

Utilizando propriedades da notação O, tem-se que a tempo de execução **para cada processo** é dado por

$$\begin{aligned} T(n,p) &= O(n) + O(\sqrt{n} \cdot \log\log\sqrt{n}) + O((n \cdot \log\log n)/p) \\ &= O(\max(n, \sqrt{n} \cdot \log\log\sqrt{n}, (n \cdot \log\log n)/p) \\ &= O((n \cdot \log\log n)/p) \end{aligned}$$

Portanto a complexidade final é

$$\boxed{T(n,p) = O((n \cdot \log\log n)/p)}$$

- **Espaço**

A memória auxiliar utilizada pelo algoritmo corresponde ao arranjo alocado para armazenar o estado de cada elemento da lista de 2 até n. Cada processo aloca independentemente toda a lista. Logo cada processo precisa alocar, no **pior caso**, **melhor caso** e **caso médio**, uma quantidade de memória

$$\boxed{E(n,p) = O(n)}$$

3.3.2 Implementação

A implementação do Algoritmo 3 foi realizada em linguagem C através da função

```
unsigned int crivo_paralelo_controle(unsigned int n, int id, int p)
```

Essa função faz parte do arquivo `controle.c`.

O parâmetro `n` dessa função é o valor limite para a busca de primos, conforme estabelecido na descrição do algoritmo. O parâmetro `id` é o identificador do processo em execução, enquanto `p` é a quantidade total de processos. A função retorna a quantidade de números primos menores ou iguais a n.

Para a troca de mensagens entre os processos foi utilizada a interface MPI (*Message Passing Interface*) [3].

Para armazenar o estado de cada elemento da lista, cada processo aloca um arranjo contendo $n-1$ elementos do tipo `char`. A alocação desse arranjo é realizada dinamicamente para evitar desperdício de recursos.

Embora a utilização de um arranjo de elementos do tipo `char` seja bastante prática, apenas 2 valores são possíveis em cada elemento (marcado e desmarcado). Assim, apenas 1 bit seria necessário para representar o estado de cada número da lista, apesar do tipo `char` conter, em geral, 8 bits. Isso faz com que o arranjo tenha tamanhos consideráveis para valores de n muito grande. Para $n = 500.000.000$, por exemplo, o arranjo ocupa aproximadamente 477 MB.

Nesse algoritmo paralelo, o tamanho do arranjo é crítico para o desempenho do programa por dois motivos:

- Cada processo envia para o processo 0 todo o seu arranjo,

- O tempo de redução é proporcional ao tamanho do arranjo.

Dessa forma, foi adotada a seguinte solução para reduzir o tamanho do arranjo: após as exclusão dos múltiplos de todos os fatores primos, o arranjo é condensado de forma que passe a utilizar um elemento para representar o estado de 8 números da lista. Isso faz com que o arranjo condensado tenha $\lceil n/8 \rceil$ elementos. Para esse fim, foi implementada a função

```
void condensar_array_bits(unsigned char *array, unsigned int tam_original, unsigned int *
    elementos_completos)
```

O parâmetro `array` é um ponteiro para o primeiro elemento do arranjo; `tam_original` é o número de elementos no arranjo antes da compressão e `elementos_completos` é um ponteiro para a variável onde será armazenado o número de elementos após a execução da função.

Assim, a redução é feita sobre o arranjo condensado e utilizando a operação ou-bit-a-bit (*bitwise-or*), diminuindo a quantidade de dados envolvidos na comunicação e encurtando o tamanho do arranjo sobre o qual a operação de redução trabalha.

Ao fim da operação de redução, o processo 0 realiza a contagem dos números primos diretamente sobre o arranjo condensado, chamando a função

```
unsigned int contar_nao_marcados_array_condensado(unsigned char *array, unsigned int tam_original
    )
```

Mais uma vez o parâmetro `array` é um ponteiro para o primeiro elemento do arranjo e o parâmetro `tam_original` é o número de elementos do arranjo antes da compressão.

Foi criado um programa chamado `controle.e` a partir da implementação do algoritmo. Ele realiza a chamada da função `crivo_paralelo_controle`. O valor de n é passado como argumento na linha de comando. Para a utilização da interface MPI, o programa deve ser executado indiretamente, através do utilitário `mpirun`. Abaixo é mostrado um exemplo de utilização:

```
> mpirun -np 4 -machinefile lista_maquinas.txt ./controle.e 500000000
```

Conforme ilustra o exemplo, o utilitário pede que sejam passados o número de processos a serem disparados (precedido de `-np`) e um arquivo texto contendo a lista de máquinas onde os processos serão executados (precedido de `-machinefile`).

O programa imprime a quantidade de números primos menores ou iguais a n na saída padrão da máquina onde é executado o processo 0, como mostrado abaixo:

```
26355867
```

No Apêndice A pode ser encontrada a listagem completa do código-fonte.

3.4 Solução Híbrida – Paralelismo de Dados e Controle

Nesse trabalho, uma terceira possibilidade de paralelismo do Crivo de Eratóstenes é explorada. A solução proposta é chamada de Solução Híbrida, pois combina características do paralelismo de dados e do paralelismo de controle.

A idéia consiste em melhorar características das duas propostas anteriores, mas é baseada mais fortemente no paralelismo de controle. Naquela proposta, todos os processos fazem, de forma independente, a construção da lista de primos até $\sqrt{n}$. Nessa nova proposta, apenas o processo 0, chamado nesse novo algoritmo de processo **mestre**, constrói a lista de primos até $\sqrt{n}$ e distribui esses fatores entre os demais processos, chamados de processos **escravos**. Essa distribuição de tarefas caracteriza o paralelismo de controle.

A construção da lista de primos até $\sqrt{n}$ é feita executando o Crivo de Eratóstenes no processo mestre de maneira seqüencial, o que poderia gerar um longo tempo de espera para os processos escravos. Para reduzir esse tempo, a distribuição é feita baseada no seguinte princípio:

> Sempre que o processo mestre identifica um número primo k para ser cortado no seu crivo (que vai de 2 até $\sqrt{n}$), todos os números não marcados menores que k^2 podem ser enviados aos processos escravos pois são garantidamente primos.

Essa é uma conseqüência imediata da definição do algoritmo. Quando o crivo vai cortar os múltiplos de 2, ele inicia os cortes de $2^2 = 4$. Logo, todos os números não cortados menores que 4 (no caso apenas o número 3) podem ser considerados primos. Após cortar os múltiplos de 2, o próximo fator primo utilizado para excluir elementos é o 3. Nesse instante, todos os valores não marcados menores que $3^2 = 9$ (números 5 e 7) podem ser declarados primos. Assim, tão logo o processo mestre identifique que um número é primo, ele o envia para que um dos processos escravos corte os múltiplos desse número no crivo principal (até n). Esse procedimento agiliza a distribuição de fatores primos para os processos escravos.

A distribuição é realizada utilizando uma política *round-robin*, ou seja, os processos são enfileirados, o primeiro da fila recebe um fator primo e é deslocado para o fim da fila. A distribuição se repete até que não haja mais fatores primos.

Outra otimização importante incorporada ao algoritmo é que os processos escravos só cortam os valores maiores que $\sqrt{n}$. Isso é possível porque o processo mestre, durante o construção da lista dos primos até $\sqrt{n}$, já executa o crivo de 2 até $\sqrt{n}$. Dessa maneira, os processos escravos precisam de trabalhar apenas na parcela da lista que vai de $\sqrt{n}$ até n. Essa divisão da lista entre os processadores caracteriza o paralelismo de dados.

Após a distribuição de todos os fatores primos e a exclusão de todos os seus múltiplos, as listas são reduzidas no processo 0. Essa redução é feita de maneira que apenas os elementos que não foram marcados em nenhum dos processos residam na lista do processo 0.

A Figura 5 mostra um grafo com a relação entre os processos durante a execução do algoritmo. Nesse grafo, o processo 0 é considerado o processo mestre e os demais são os processos escravos.

O Algoritmo 4 mostra em pseudo-código as tarefas executadas pelo processo mestre. A sub-rotina Enviar(PróximoPrimo, PróximoProcAtendido) envia o número primo PróximoPrimo para o processo cujo identificador é PróximoProcAtendido. Já a chamada Enviar$(-1, i)$, dessa mesma sub-rotina, sinaliza para o processo escravo i que não há mais fatores primos a serem processados. A execução dessa sub-rotina garante que as tarefas chegam ao seu destino na mesma ordem em que são enviadas. Além das tarefas já descritas, o processo mestre também é responsável por todas as operações de entrada e saída.

As instruções executadas pelos processos escravos são descritas em pseudo-código no Algoritmo 5. Nota-se que as tarefas dos processos escravos são muito parecidas com o Crivo de Eratóstenes seqüencial, exceto pelo fato de que os fatores primos utilizados para cortar elementos são recebidos do processo mestre. Por fim, os escravos enviam suas listas parciais para o processo mestre através da instrução de redução. A chamada da sub-rotina Receber(Fator, 0) recebe o valor (que pode ser um número primo ou um *flag* indicando que não há mais primos) enviado pelo processo mestre e o armazena em Fator.

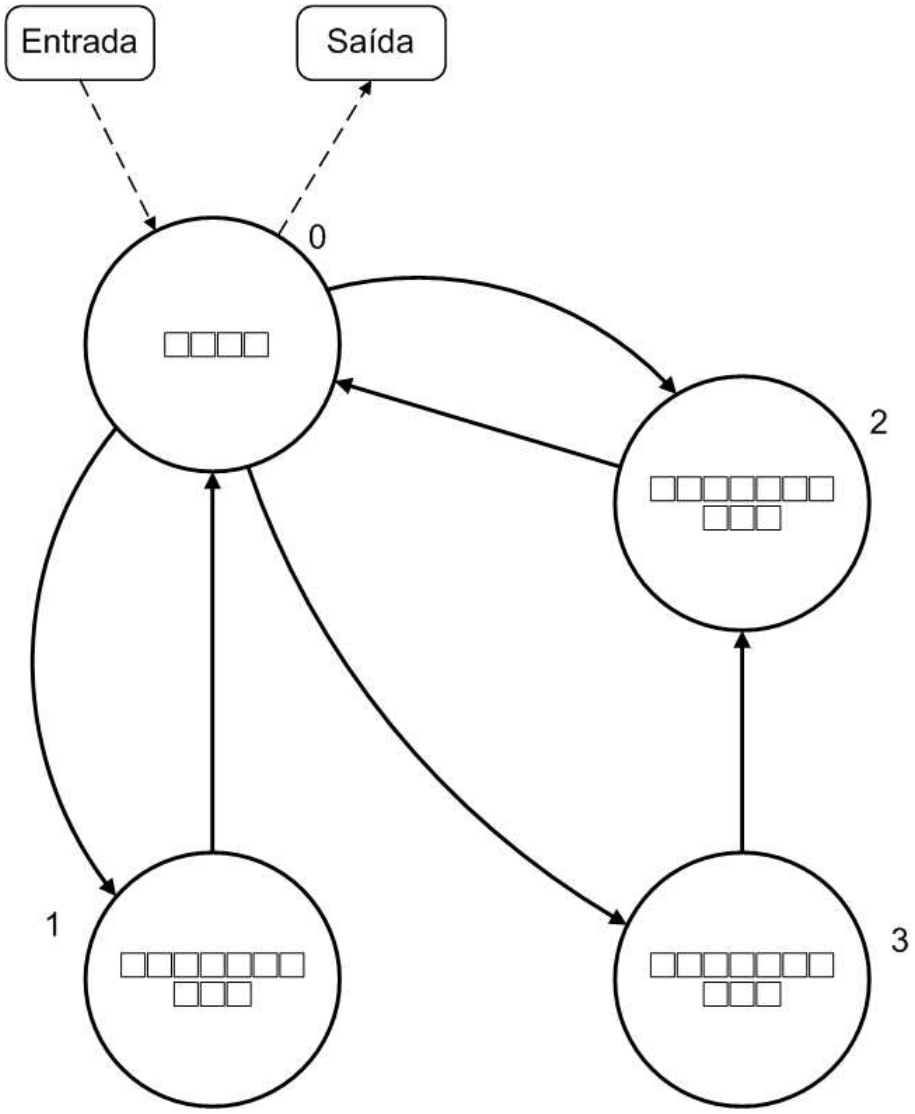

Figura 5: Grafo de processos para o paralelismo híbrido. Linhas curvas representam a comunicação relativa ao envio de primos para os processos escravos enquanto linhas retas representam a comunicação relativa à redução. Linhas pontilhadas representam entrada e saída.

3.4.1 Análise de Complexidade

A análise de complexidade desse algoritmo paralelo será realizada considerando como variáveis o valor limite n para a busca de primos e o número de processos p. A complexidade será analisada para o tempo de execução e para a utilização de espaço em memória.

- **Tempo**

A complexidade de tempo será realizada primeiramente para o processo mestre. O trabalho desse processo consiste basicamente em executar o Crivo de Eratóstenes para identificar os números primos até $\sqrt{n}$. Conforme é mostrado na seção 3.1.1, a complexidade desse passo é

Algoritmo 4 Crivo de Eratóstenes – Paralelismo Híbrido – Mestre

```
{Inicializa a lista}
for i ← 2 até n do
  Lista_i ← Desmarcado
end for
Fator ← 2
PróximoPrimo ← 2
PróximoProcAtendido ← 1
while Fator^2 ≤ √n do
  {Distribui todos os primos conhecidos até o momento}
  while PróximoPrimo < Fator^2 e PróximoPrimo ≤ √n do
    if Lista_PróximoPrimo = Desmarcado then
      Enviar(PróximoPrimo, PróximoProcAtendido)
      PróximoProcAtendido ← (PróximoProcAtendido mod (p − 1)) + 1
    end if
    PróximoPrimo ← PróximoPrimo + 1
  end while
  {Verifica se todos os fatores já foram distribuídos}
  if PróximoPrimo > √n then
    for i ← 1 até p do
      Enviar(−1, i)
    end for
  end if
  {Marca os múltiplos de Fator}
  k ← Fator^2
  while k ≤ √n do
    Lista_k ← Marcado
    k ← k + Fator
  end while
  {Escolhe o próximo fator primo p}
  while Lista_Fator ≠ Desmarcado do
    Fator ← Fator + 1
  end while
end while
{Redução das listas no processo 0}
Redução(Lista, 0, OU)
{Processo 0 imprime o total de primos}
TotalPrimos ← 0
for i ← 2 até n do
  if Lista_i = Desmarcado then
    TotalPrimos ← TotalPrimos + 1
  end if
end for
print TotalPrimos
```

$$\boxed{T(n,p)_{\text{Mestre}} = O(\sqrt{n}\log\log\sqrt{n})}$$

Algoritmo 5 Crivo de Eratóstenes – Paralelismo Híbrido – Escravo

```
{Inicializa a lista}
for i ← 2 até n do
  Lista_i ← Desmarcado
end for
Receber(Fator, 0)
while Fator ≠ −1 do
  {Marca os múltiplos de Fator maiores que √n}
  k ← (⌊√n⌋ − (⌊√n⌋ mod Fator)) + Fator
  while k ≤ n do
    Lista_k ← Marcado
    k ← k + Fator
  end while
  {Recebe o próximo fator primo}
  Receber(Fator, 0)
end while
{Redução das listas no processo 0}
Redução(Lista, 0, OU)
```

Os $p-1$ processos escravos realizam o Crivo de Eratóstenes com paralelismo de controle. Entretanto, eles operam somente no intervalo da lista que vai de $\sqrt{n}$ até n. Portanto, utilizando o resultado da seção 3.3.1 tem-se

$$\boxed{T(n,p)_{\text{Escravo}} = O(((n-\sqrt{n}) \cdot \log\log(n-\sqrt{n}))/(p-1))}$$

- **Espaço**

A memória auxiliar utilizada pelo algoritmo corresponde ao arranjo alocado para armazenar o estado de cada elemento da lista. Embora o processo mestre manipule apenas os elementos de 2 até $\sqrt{n}$ e os escravos de $\sqrt{n}$ até n, todos têm que alocar a lista completa para viabilizar a operação de redução. Logo cada processo precisa alocar, no **pior caso**, **melhor caso** e **caso médio**, uma quantidade de memória

$$\boxed{E(n,p) = O(n)}$$

3.4.2 Implementação

A implementação dos Algoritmos 4 e 5 foi realizada em linguagem C através da função

```
unsigned int crivo_paralelo_hibrido(unsigned int n, int id, int p)
```

Essa função faz parte do arquivo `hibrido.c`.

O parâmetro `n` dessa função é o valor limite para a busca de primos. O parâmetro `id` é o identificador do processo e `p` é a quantidade total de processos. O valor de retorno da função é a quantidade de números primos menores ou iguais a n.

Para a troca de mensagens entre os processos foi utilizada a interface MPI (*Message Passing Interface*) [3].

Cada processo aloca um arranjo contendo $n - 1$ elementos do tipo `char` para armazenar o estado de cada elemento da lista (marcado e desmarcado). O arranjo é alocado dinamicamente para evitar desperdício de recursos.

Assim como foi descrito na seção 3.3.2, a utilização de um arranjo de caracteres pode ser muito dispendioso em termos da quantidade de comunicação e processamento para realizar a redução. Portanto, assim como na implementação do paralelismo de controle, a implementação do paralelismo híbrido realiza a condensação do arranjo através da função

```
void condensar_array_bits(unsigned char *array, unsigned int tam_original, unsigned int *
    elementos_completos)
```

O parâmetro `array` é um ponteiro para o primeiro elemento do arranjo; `tam_original` é o número de elementos no arranjo antes da compressão e `elementos_completos` é um ponteiro para a variável onde será armazenado o número de elementos após a execução da função.

A redução é feita sobre o arranjo condensado e utilizando a operação ou-bit-a-bit (*bitwise-or*), reduzindo a quantidade de dados envolvidos na comunicação e encurtando o tamanho do arranjo sobre o qual a operação de redução trabalha.

Após a operação de redução, o processo 0 realiza a contagem dos números primos diretamente sobre o arranjo condensado, chamando a função

```
unsigned int contar_nao_marcados_array_condensado(unsigned char *array, unsigned int tam_original
    )
```

Mais uma vez o parâmetro `array` é um ponteiro para o primeiro elemento do arranjo e o parâmetro `tam_original` é o número de elementos do arranjo antes da compressão.

Foi criado um programa chamado `hibrido.e` a partir da implementação do algoritmo onde a função `crivo_paralelo_hibrido` é chamada. O valor de n é passado como argumento na linha de comando. Para a utilização da interface MPI, o programa deve ser executado indiretamente, através do utilitário `mpirun`. Abaixo é mostrado um exemplo de utilização:

```
> mpirun -np 4 -machinefile lista_maquinas.txt ./hibrido.e 500000000
```

O utilitário pede que sejam passados o número de processos a serem disparados (precedido de `-np`) e um arquivo texto contendo a lista de máquinas onde os processos serão executados (precedido de `-machinefile`).

Ao fim da execução o programa imprime a quantidade de números primos menores ou igual a n na saída padrão da máquina onde é executado o processo 0, como mostrado abaixo:

```
26355867
```

A listagem completa do código-fonte pode ser encontrada no Apêndice A.

4 Resultados Experimentais

A partir da implementação das soluções propostas, foram realizados experimentos para analisar o comportamento dos diversos algoritmos.

Os experimentos realizados se dividiram em dois grupos principais. O primeiro visa analisar o comportamento do tempo de execução do Crivo de Eratóstenes seqüencial. Os resultados desses experimentos são interpretados na subseção 4.1. O segundo grupo de experimentos avalia o *speed-up* obtido a partir das diversas estratégias de paralelização propostas nas subseções 3.2, 3.3 e 3.4. Os resultados desses experimentos são analisados na subseção 4.2.

Para a realização dos experimentos foram utilizadas 8 computadores pessoais do Departamento de Ciência da Computação da UFMG. Os nomes e IPs das máquinas estão listados na Tabela 1.

Tabela 1: Lista de máquinas utilizadas nos experimentos

Nome	IP
acrux.grad	150.164.6.35
aldebaran.grad	150.164.6.36
antares.grad	150.164.6.38
deneb.grad	150.164.6.42
mimosa.grad	150.164.6.41
pollux.grad	150.164.6.39
regulus.grad	150.164.6.43
spica.grad	150.164.6.37

Todas as máquinas utilizadas possuem o configurações de *hardware* muito semelhantes:

Processador Dois processadores Intel Pentium 4 3.00 GHz

Memória Cache 2048 kB de cache L2 em cada processador

Memória Principal 1019384 kB

As máquinas também possuem configuração de *software* muito parecidas:

Distribuição Linux SUSE Linux 10

Kernel Versão 2.6.13-15.10

MPI MPICH 1.1.2

GCC Versão 4.0.2 20050901

Todas as máquinas estão ligadas por uma rede local ethernet com largura de banda nominal de 100 Mbits.

4.1 Comportamento do Algoritmo Serial

O experimentos com o algoritmo serial consistiram em executar programa que implementa o algoritmo variando o valor de n de 10.000.000 até 500.000.000, incrementando esse parâmetro de 10.000.000 em cada execução. O tempo de execução de cada experimento foi medido.

Esse processo foi realizado em todas as máquinas listadas na Tabela 1, sendo obtidos a média e desvio padrão das execuções. Esse cuidado foi tomado para que os resultados do algoritmo serial pudessem ser comparados com os dos algoritmos paralelos, sem que variações no desempenho dos equipamentos contaminassem os resultados.

A Figura 6 mostra o tempo de execução do algoritmo serial em função do tamanho do valor limite n. O gráfico mostra um crescimento do tempo de execução ligeiramente superior ao linear, especialmente para os valores menores de n. Esse comportamento está inteiramente de acordo com a análise de complexidade do algoritmo, mostrada em 3.1.1, que é de $O(n \cdot \log \log n)$.

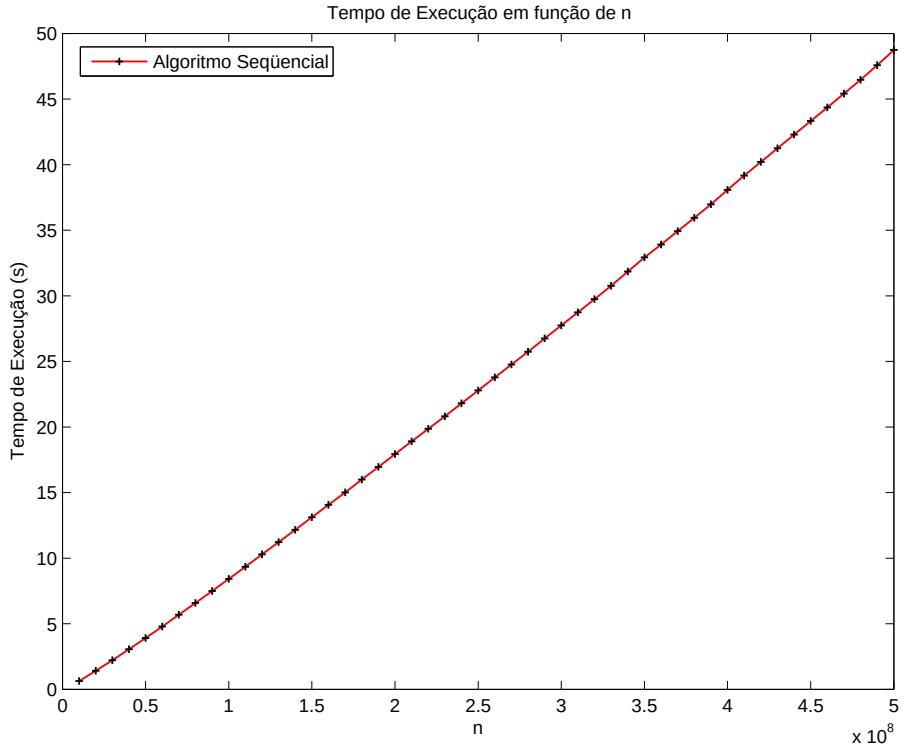

Figura 6: Tempo de execução do algoritmo serial em função do valor limite n.

Embora as máquinas utilizadas nos experimentos tenham configurações de *hardware* e *software* muito parecidas, foi constatada uma variação relativamente grande entre os tempos de execução em cada equipamento. A Figura 7 mostra o tempo de execução em função de n, juntamente com as barras de desvio padrão para cada valor médio.

O gráfico mostra que o desvio padrão das médias do tempo de execução ficam em torno de 20% do valor médio, um valor bastante alto levando-se em conta que os equipamentos são semelhantes e as condições de cada máquina no momento da

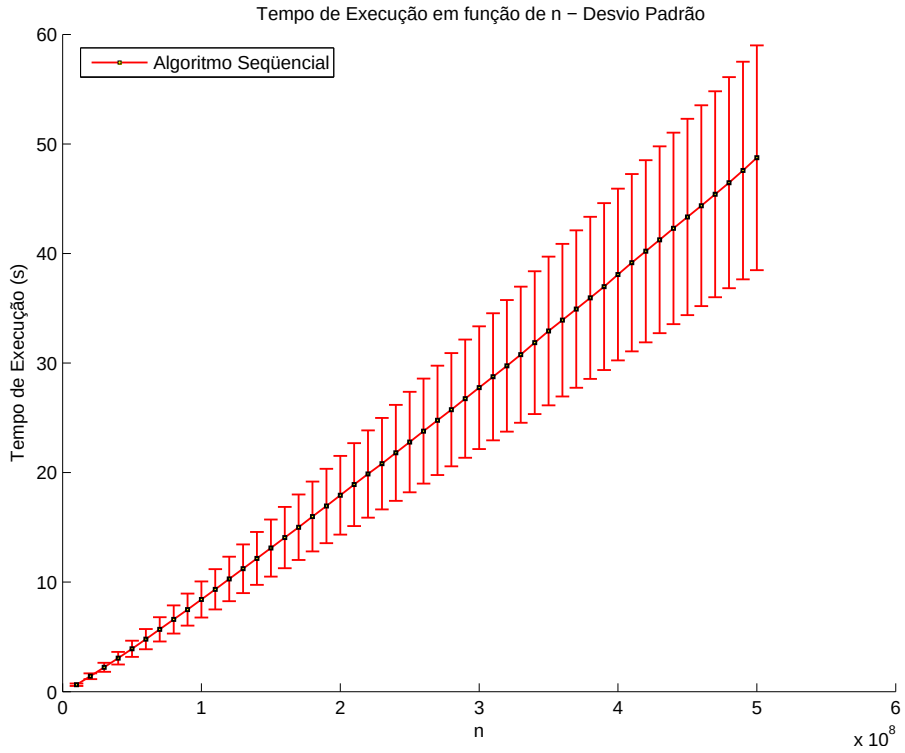

Figura 7: Tempo de execução do algoritmo serial em função do valor limite n. Desvio padrão

execução (processos concorrentes, memória disponível, entre outros) era similar. Esses altos desvios podem ser atribuídos a diferenças inerentes a cada equipamento, que não podem ser determinados pelos atributos listados nesse trabalho.

Felizmente esses desvios não afetam os resultados, uma vez que o algoritmo foi executado em todas as máquinas e o valor médio foi considerado.

A listagem completa dos resultados dos experimentos pode ser encontrada no Apêndice B.

4.2 Speed-up

Os experimentos para análise de *speed-up* dos algoritmos paralelos foram realizados variando-se o número de processos utilizados. Para esses testes, foi adotado o valor de n de 500.000.000, que se mostrou grande o bastante para uma análise do comportamento do algoritmo e não comprometeu as medições com a influência de fatores como paginação do sistema de memória virtual.

Cada experimento foi repetido 10 vezes para evitar que desvios aleatórios contaminassem os resultados. A partir dessas medições, o valor médio e o desvio padrão foram calculados.

A Figura 8 mostra o resultado do tempo de execução de cada algoritmo paralelo em função do número de processadores. No mesmo gráfico foram plotados os tempos de execução do paralelismo de dados, de controle e da solução híbrida. Não há valor de tempo na solução híbrida para $p = 1$ pois esse algoritmo requer a utilização de, no mínimo, 2 processos.

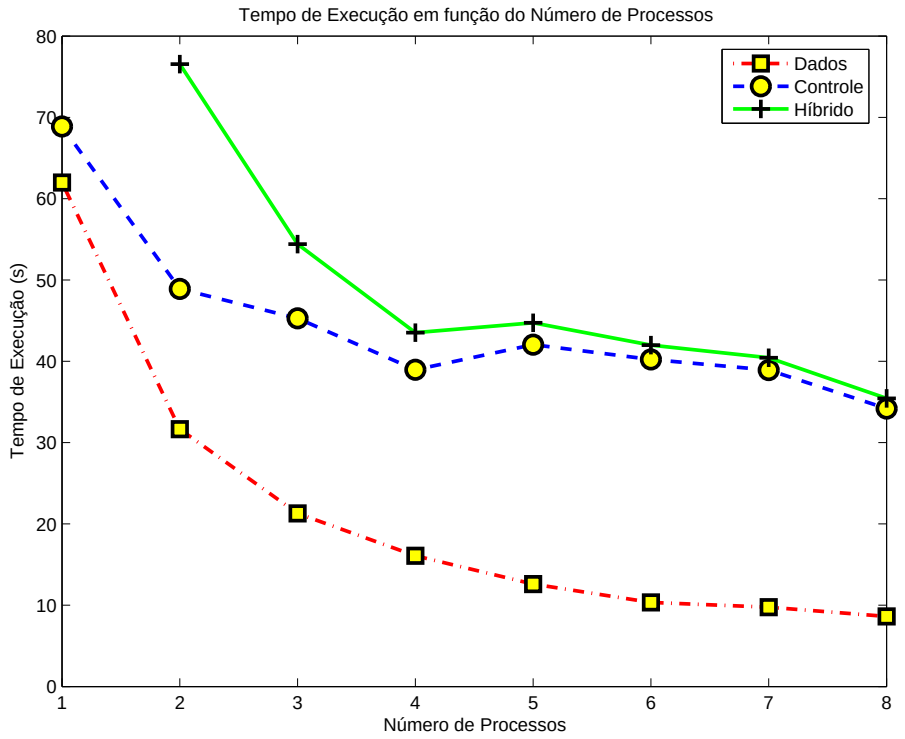

Figura 8: Tempo de execução dos algoritmos paralelos em função do número de processos p.

O gráfico mostra que o paralelismo de dados apresenta um tempo de execução menor do que os outros dois algoritmos para qualquer quantidade de processadores. Além disso, sua escalabilidade é bem maior que a dos demais, pois a adição de um novo processador permite uma divisão melhor da carga de trabalho sem acarretar um grande custo adicional.

Essas vantangens do paralelismo de dados sobre as duas outras alternativas decorrem principalmente do fato de esse algoritmo exigir um quantidade de comunicação relativamente baixa. O paralelismo de controle e a solução híbrida precisam que, após as listas de números terem sido processadas paralelamente, elas sejam sumarizadas em uma lista apenas. Isso implica uma quantidade de comunicação enorme, já que as listas ocupam uma espaço de memória proporcional ao valor de n, além de incorrem em um maior trabalho serial para que as listas sejam reduzidas. Já o paralelismo de dados permite que as listas parciais sejam totalizadas em cada processo e apenas os totais parciais têm de ser somados e reunidos no processo raiz.

Outra vantagem do paralelismo de dados sobre os outros dois algoritmos é que, à medida em que número de processos aumenta, cada processo precisa manipular uma quantidade menor de memória. Isso ajuda a combater um dos maiores problemas do Crivo de Eratóstenes, sua baixa localidade de referência, já que o algoritmo manipula de maneira dispersa um arranjo que tende a ser muito grande (proporcional ao valor de n), provocando um índice de acerto de cache muito baixo. Portanto, reduzir o tamanho do arranjo manipulado por cada processo ajuda a combater esse problema, melhorando o desempenho do algoritmo.

Comparando o paralelismo de controle com a solução híbrida vemos que o primeiro

tem melhor desempenho que o segundo, especialmente para valores baixos de p. Isso é explicado pelo fato de que a solução híbrida dedica um processador para realizar a coordenação do processo, executando o processo mestre, o que, para um número baixo de processadores, leva a um balanceamento de carga muito ruim. À medida em que o número de processos aumenta, essa diferença tende a diminuir.

Um fato notável no gráfico da Figura 8 é que os tempos do paralelismo de controle e da solução híbrida apresentam um mínimo local em $p = 4$. Isso sugere que para esse número de processos ocorra um *trade-off* favorável entre divisão da carga e penalização imposta pela comunicação.

O gráfico da Figura 9 mostra o *speed-up* obtido por cada algoritmo paralelo em função do número de processos p. Os valores plotados foram obtidos por meio da seguinte fórmula:

$$\text{Speed-Up}(p) = \frac{\text{TempoSerial}}{\text{TempoParalelo}(p)}$$

Onde TempoSerial é o tempo de execução do algoritmo serial para $n = 500.000.000$ (veja subseção 4.1) e TempoParalelo(p) é o tempo de execução do algoritmo paralelo utilizando p processos (Figura 8). Não há valor de *speed-up* na solução híbrida para $p = 1$ pois esse algoritmo requer a utilização de, no mínimo, 2 processos.

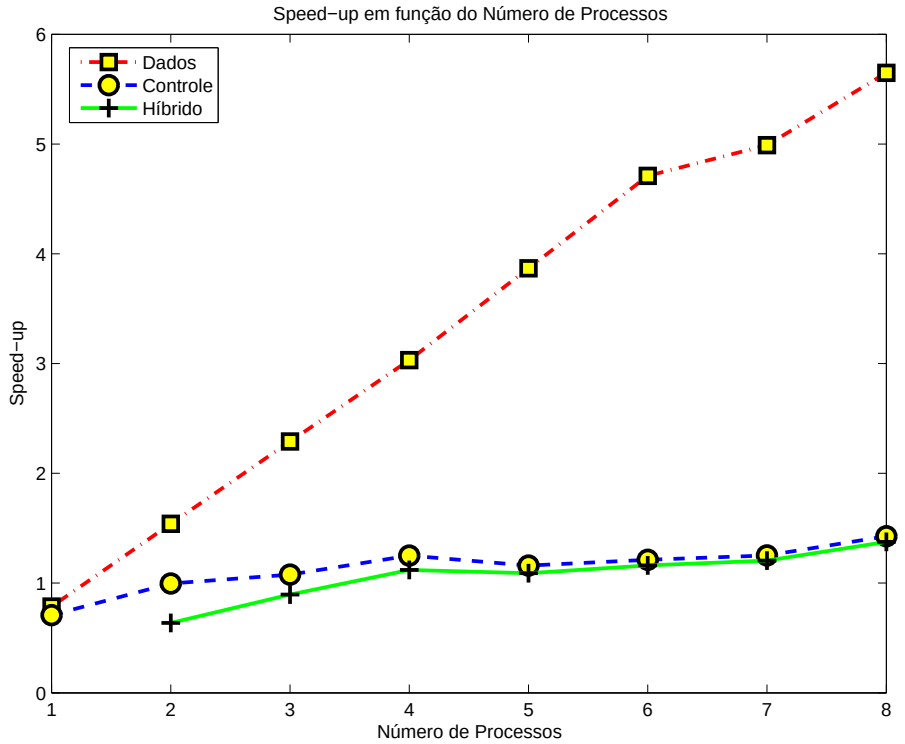

Figura 9: *Speed-up* dos algoritmos paralelos em função do número de processos p.

O gráfico da Figura 9 reafirma a superioridade do paralelismo de dados sobre os outros dois algoritmos. A relação entre paralelismo de controle e a solução híbrida também é percebida conforme a análise anterior.

Nota-se que o paralelismo de dados apresenta uma escalabilidade bastante satisfatória, com um *speed-up* linear e de coeficiente angular menor, mas relativamente

próximo de 1. Com 8 processadores, o algoritmo atingiu um valor de *speed-up* maior que $5,5$, o que representa um resultado muito satisfatório. Já o paralelismo de controle e a solução híbrida não chegaram a atingir um *speed-up* de $1,5$, mesmo utilizando 8 processadores.

Com $p = 1$, tanto o paralelismo de dados quanto de controle têm *speed-up* inferior a 1. Isso era esperado pois, utilizando o mesmo número de processadores, o algoritmo serial tende a ser mais rápido pois não inclui as operações necessárias para o controle da execução paralela.

O Apêndice B contém a listagem completa dos resultados.

5 Conclusões

Nesse trabalho foi realizado um estudo do Crivo de Eratóstenes, um algoritmo clássico para encontrar todos os números primos menores ou iguais a um número inteiro positivo n. O algoritmo foi estudado sob o ponto de vista da programação paralela em uma máquina MIMD do tipo NOW.

Foram exploradas três possibilidades de paralelismo: paralelismo de controle, de dados, e uma solução híbrida, que combina características das duas outras abordagens. A análise da complexidade do algoritmo serial foi realizada, bem como dos três algoritmos paralelos propostos.

Foram realizados experimentos para avaliação do tempo de execução dos três algoritmos, além de experimentos com o algoritmo seqüencial para análise do *speed-up*.

Os resultados experimentais mostraram o tempo de execução resultante da paralelização do Crivo de Eratóstenes pode variar muito de acordo com a abordagem escolhida. O paralelismo de dados se mostrou a alternativa mais escalável, apresentando um *speed-up* superior aos outros dois algoritmos para todas as quantidades de processos experimentadas.

O paralelismo de controle e a solução híbrida, além de serem algoritmos de implementação mais complexa, mostraram um *speed-up* muito reduzido, principalmente em função da grande quantidade de comunicação envolvida na execução desses algoritmos.

Por fim, o trabalho mostrou que a utilização de algoritmos paralelos pode apresentar uma alternativa interessante para a resolução de problemas de alto custo computacional. Caso o algoritmo paralelo seja bem projetado, ele pode viabilizar soluções que não seriam possíveís utilizando apenas algoritmos seqüenciais.

A Listagem dos Códigos-Fonte

Algoritmo Serial

Listagem 1: serial.c

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

#include <sys/time.h>
#include <sys/resource.h>

#define TRUE 1
#define FALSE 0

unsigned long crivo(unsigned long n)
{
  unsigned long max_fator = (unsigned long)sqrt((double)n);
  unsigned char* lista = (unsigned char*)malloc((n+1)*sizeof(unsigned char));

  unsigned long i = 0;
  for(i = 0; i <= n; i++)
    lista[i] = FALSE;

  unsigned long fator = 2;
  while(fator <= max_fator)
  {
    // Escolhe proximo primo
    while(lista[fator] == TRUE)
    {
      fator++;
    }

    unsigned long indice = fator*fator;
    while(indice <= n)
    {
      lista[indice] = TRUE;
      indice += fator;
    }

    fator++;
  }

  unsigned long contador = 0;
  for(i = 2; i <= n; i++)
    if(lista[i] == FALSE)
      contador++;

  free(lista);

  return contador;
}

int main(int argc, char *argv[])
{
  struct timeval TempoInicio;
  gettimeofday(&TempoInicio, NULL);

  unsigned long max = (unsigned long)atol(argv[1]);
  crivo(max);

  struct timeval TempoFim;
  double TempoEmSegundos;
  gettimeofday(&TempoFim, NULL);
  TempoEmSegundos = (TempoFim.tv_sec - TempoInicio.tv_sec) * 1000000;
  TempoEmSegundos += TempoFim.tv_usec - TempoInicio.tv_usec;
  TempoEmSegundos /= 1000000;

  printf("%lu\t%lf\n", max, TempoEmSegundos);

  return 0;
}
```

Paralelismo de Dados

Listagem 2: dados.c

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define MIN(a,b)  ((a)<(b)?(a):(b))

#define ID_ROOT 0

#define FALSE 0
```

```
#define TRUE 1

unsigned int crivo_paralelo_dados(unsigned int n, int id, int p)
{
  unsigned int     count;        /* Local prime count */
  unsigned int     first;        /* Index of first multiple */
  unsigned int     high_value;   /* Highest value on this proc */
  unsigned int     i;
  unsigned int     index;        /* Index of current prime */
  unsigned int     low_value;    /* Lowest value on this proc */
  char            *marked;       /* Portion of 2,...,'n' */
  unsigned int     proc0_size;   /* Size of proc 0's subarray */
  unsigned int     prime;        /* Current prime */
  unsigned int     size;         /* Elements in 'marked' */

  /* Figure out this process's share of the array, as
  well as the integers represented by the first and
  last array elements */
  low_value = 2 + id*(n-1)/p;
  high_value = 1 + (id+1)*(n-1)/p;
  size = high_value - low_value + 1;

  /* Bail out if all the primes used for sieving are
  not all held by process 0 */
  proc0_size = (n-1)/p;
  if((2 + proc0_size) < (int) sqrt((double) n))
  {
    if (id == ID_ROOT)
      printf("Too many processes\n");
    MPI_Finalize();
    exit (1);
  }

  /* Allocate this process's share of the array. */
  marked = (char *) malloc (size * sizeof(char));
  if(marked == NULL)
  {
    printf("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit (1);
  }

  for (i = 0; i < size; i++)
    marked[i] = FALSE;

  if (id == ID_ROOT)
    index = 0;
  prime = 2;
  do
  {
    if (prime * prime > low_value)
    {
      first = prime * prime - low_value;
    }
    else
    {
      if((low_value % prime) == 0)
        first = 0;
      else
        first = prime - (low_value % prime);
    }

    for (i = first; i < size; i += prime)
      marked[i] = TRUE;

    if (id == ID_ROOT)
    {
      while(marked[++index] == TRUE);
      prime = index + 2;
    }

    if(p > 1)
      MPI_Bcast (&prime,  1, MPI_INT, ID_ROOT, MPI_COMM_WORLD);

  } while (prime * prime <= n);

  count = 0;
  for (i = 0; i < size; i++)
    if (marked[i] == FALSE)
      count++;

  free(marked);

  return count;
}

int main (int argc, char *argv[])
{
  unsigned int count;            /* Local prime count */
  double elapsed_time;           /* Parallel execution time */
  unsigned int global_count;     /* Global prime count */
  int id;                        /* Process ID number */
  unsigned int n;                /* Sieving from 2, ..., 'n' */
  int p;                         /* Number of processes */

  MPI_Init (&argc, &argv);
```

```
  /* Start the timer */
  MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Barrier(MPI_COMM_WORLD);
  elapsed_time = -MPI_Wtime();

  if (argc != 2)
  {
    if(id == 0)
      printf ("Command line: %s <n>\n", argv[0]);
    MPI_Finalize();
    exit (1);
  }

  n = (unsigned int)atoi(argv[1]);

  count = crivo_paralelo_dados(n, id, p);

  if (p > 1)
    MPI_Reduce (&count, &global_count, 1, MPI_UNSIGNED, MPI_SUM, 0, MPI_COMM_WORLD);
  else
    global_count = count;

  /* Stop the timer */
  elapsed_time += MPI_Wtime();

  /* Print the results */
  if (id == 0)
  {
/*    printf ("There are %u primes less than or equal to %u\n", global_count, n);
    printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);*/
    printf("%d\t%1f\n", p, elapsed_time);
  }

  MPI_Finalize();
  return 0;
}
```

Paralelismo de Controle

Listagem 3: controle.c

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define MIN(a,b)  ((a)<(b)?(a):(b))

#define ID_ROOT 0

#define FALSE 0
#define TRUE 1

inline unsigned char descondensar_elemento_array_bits(unsigned char *array,
  unsigned int elemento, unsigned int posicao_no_elemento)
{
  return (array[elemento] >> posicao_no_elemento) & 1;
}

unsigned int contar_nao_marcados_array_condensado(unsigned char *array, unsigned int tam_original
    )
{
  unsigned int bits_por_elemento = sizeof(unsigned char)*8;
  unsigned int elementos_completos = tam_original / bits_por_elemento;
  unsigned int tam_ultimo_elemento = tam_original % bits_por_elemento;

  unsigned int total = 0;

  unsigned int i = 0, j = 0;
  for(i = 0; i < elementos_completos; i++)
  {
    for(j = 0; j < bits_por_elemento; j++)
    {
      unsigned char descondensado =
        descondensar_elemento_array_bits(array, i, j);
      if(descondensado == FALSE)
        total++;
    }
  }

  for(j = 0; j < tam_ultimo_elemento; j++)
  {
    unsigned char descondensado =
      descondensar_elemento_array_bits(array, elementos_completos, j);
    if(descondensado == FALSE)
      total++;
  }

  return total - 2;
}
```

```
void condensar_array_bits(unsigned char *array, unsigned int tam_original,
  unsigned int *elementos_completos)
{
  unsigned int bits_por_elemento = sizeof(unsigned char)*8;
  *elementos_completos = tam_original / bits_por_elemento;
  unsigned int tam_ultimo_elemento = tam_original % bits_por_elemento;

  unsigned int i = 0, j = 0;
  for(i = 0; i < *elementos_completos; i++)
  {
    array[i] = 0;
    for(j = 0; j < bits_por_elemento; j++)
    {
      array[i] |= array[i*bits_por_elemento + j] << j;
    }
  }

  array[*elementos_completos] = 0;
  for(j = 0; j < tam_ultimo_elemento; j++)
  {
    array[*elementos_completos] |= array[(*elementos_completos)*bits_por_elemento + j] << j;
  }
}

unsigned int crivo(unsigned int n, unsigned char* lista)
{
  unsigned int max_fator = (unsigned int)sqrt((double)n);

  unsigned int i = 0;
  for(i = 0; i <= n; i++)
    lista[i] = FALSE;

  unsigned int fator = 2;
  while(fator <= max_fator)
  {
    // Escolhe proximo primo
    while(lista[fator] == TRUE)
    {
      fator++;
    }

    unsigned int indice = fator*fator;
    while(indice <= n)
    {
      lista[indice] = TRUE;
      indice += fator;
    }

    fator++;
  }

  unsigned int contador = 0;
  for(i = 2; i <= n; i++)
    if(lista[i] == FALSE)
      contador++;

  return contador;
}

void imprimir_primos(char *lista, int n)
{
  int i = 0;
  for(i = 2; i <= n; i++)
    if(lista[i] == FALSE)
      printf("%d ", i);

  printf("\n");
}

void imprimir_array_char(unsigned char *lista, int n)
{
  int i = 0;
  for(i = 0; i < n; i++)
    printf("%x ", 0x000000ff & lista[i]);

  printf("\n");
}

unsigned int crivo_paralelo_controle(unsigned int n, int id, int p)
{
  unsigned char* lista = (unsigned char*)malloc((n+1)*sizeof(unsigned char));
  unsigned int max_fator = (unsigned int)sqrt((double)n);
  crivo(max_fator, lista);

  unsigned int i = 0;
  for(i = max_fator+1; i <= n; i++)
    lista[i] = FALSE;

  unsigned int fator = 1;
  int ordem_primo = -1;
  while(fator <= max_fator)
  {
    // Seleciona apenas os primos designados para esse processo
    do
    {
      // Escolhe proximo primo
      do
```

```
      {
        fator++;
      }while(lista[fator] == TRUE);

      ordem_primo++;

    }while((ordem_primo % p) != id);

    unsigned int indice = fator*fator;
    while(indice <= n)
    {
      lista[indice] = TRUE;
      indice += fator;
    }
  }

  //Condensar array
  unsigned int elementos_completos;
  condensar_array_bits(lista, n+1, &elementos_completos);

  unsigned char *receive = NULL;
  if(id == ID_ROOT)
    receive = (unsigned char*)malloc((elementos_completos+1) * sizeof(unsigned char));

  //Reduzir
   MPI_Reduce(lista, receive, elementos_completos+1, MPI_UNSIGNED_CHAR, MPI_BOR, ID_ROOT,
       MPI_COMM_WORLD);

  free(lista);

  //Totalizar primos
  unsigned int total_primos = 0;
  if(id == ID_ROOT)
  {
    total_primos = contar_nao_marcados_array_condensado(receive, n+1);
    free(receive);
  }

  return total_primos;
}

int main (int argc, char *argv[])
{
  double elapsed_time;          /* Parallel execution time */
  unsigned int global_count;    /* Global prime count */
  int id;                       /* Process ID number */
  unsigned int n;               /* Sieving from 2, ..., 'n' */
  int p;                        /* Number of processes */

  MPI_Init (&argc, &argv);

  /* Start the timer */
  MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Barrier(MPI_COMM_WORLD);
  elapsed_time = -MPI_Wtime();

  if (argc != 2)
  {
    if(id == 0)
      printf ("Command line: %s <n>\n", argv[0]);
    MPI_Finalize();
    exit (1);
  }

  n = (unsigned int)atoi(argv[1]);

  global_count = crivo_paralelo_controle(n, id, p);

  /* Stop the timer */
  elapsed_time += MPI_Wtime();

  /* Print the results */
  if (id == ID_ROOT)
  {
/*    printf ("There are %u primes less than or equal to %u\n", global_count, n);
    printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);*/
    printf("%d\t%lf\n", p, elapsed_time);
  }

  MPI_Finalize();
  return 0;
}
```

Solução Híbrida

Listagem 4: hibrido.c

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

```
#include <stdlib.h>
#define MIN(a,b)  ((a)<(b)?(a):(b))

#define ID_ROOT 0

#define FALSE 0
#define TRUE 1

inline unsigned char descondensar_elemento_array_bits(unsigned char *array,
  unsigned int elemento, unsigned int posicao_no_elemento)
{
  return (array[elemento] >> posicao_no_elemento) & 1;
}

unsigned int contar_nao_marcados_array_condensado(unsigned char *array, unsigned int tam_original
    )
{
  unsigned int bits_por_elemento = sizeof(unsigned char)*8;
  unsigned int elementos_completos = tam_original / bits_por_elemento;
  unsigned int tam_ultimo_elemento = tam_original % bits_por_elemento;

  unsigned int total = 0;

  unsigned int i = 0, j = 0;
  for(i = 0; i < elementos_completos; i++)
  {
    for(j = 0; j < bits_por_elemento; j++)
    {
      unsigned char descondensado =
        descondensar_elemento_array_bits(array, i, j);
      if(descondensado == FALSE)
        total++;
    }
  }

  for(j = 0; j < tam_ultimo_elemento; j++)
  {
    unsigned char descondensado =
      descondensar_elemento_array_bits(array, elementos_completos, j);
    if(descondensado == FALSE)
      total++;
  }

  return total - 2;
}

void condensar_array_bits(unsigned char *array, unsigned int tam_original,
  unsigned int *elementos_completos)
{
  unsigned int bits_por_elemento = sizeof(unsigned char)*8;
  *elementos_completos = tam_original / bits_por_elemento;
  unsigned int tam_ultimo_elemento = tam_original % bits_por_elemento;

  unsigned int i = 0, j = 0;
  for(i = 0; i < *elementos_completos; i++)
  {
    array[i] = 0;
    for(j = 0; j < bits_por_elemento; j++)
    {
      array[i] |= array[i*bits_por_elemento + j] << j;
    }
  }

  array[*elementos_completos] = 0;
  for(j = 0; j < tam_ultimo_elemento; j++)
  {
    array[*elementos_completos] |= array[(*elementos_completos)*bits_por_elemento + j] << j;
  }
}

void imprimir_primos(char *lista, int n)
{
  int i = 0;
  for(i = 2; i <= n; i++)
    if(lista[i] == FALSE)
      printf("%d ", i);

  printf("\n");
}

void imprimir_array_char(unsigned char *lista, int n)
{
  int i = 0;
  for(i = 0; i < n; i++)
    printf("%x ", 0x000000ff & lista[i]);

  printf("\n");
}

unsigned int crivo_paralelo_hibrido(unsigned int n, int id, int p)
{
  MPI_Status stat;

  unsigned char* lista = (unsigned char*)malloc((n+1)*sizeof(unsigned char));
  unsigned int max_fator = (unsigned int)sqrt((double)n);
  unsigned int fim_vetor_marcacao = n;
```

```
  if(id == ID_ROOT)
  {
    fim_vetor_marcacao = max_fator;
    max_fator = (unsigned int)sqrt((double)max_fator);
  }

  unsigned int i = 0;
  for(i = 0; i <= n; i++)
    lista[i] = FALSE;

  unsigned int fator = 1;
  unsigned int proximo_primo = 2;
  int proximo_proc_atendido = 1;
  while(fator <= max_fator)
  {
    unsigned int fator_quadrado;
    unsigned int indice = 0;
    if(id == ID_ROOT)
    {
      // Escolhe proximo primo
      do
      {
        fator++;
      }while(lista[fator] == TRUE);

      //printf("Proc Root vai cortar %d\n", fator);

      fator_quadrado = fator*fator;
      while((proximo_primo < fator_quadrado) && (proximo_primo <= fim_vetor_marcacao))
      {
        //Enviar todos os primos menores que fator*fator
        if(lista[proximo_primo] == FALSE)
        {
          MPI_Send(&proximo_primo, 1, MPI_UNSIGNED, proximo_proc_atendido, 1, MPI_COMM_WORLD);
          proximo_proc_atendido = (proximo_proc_atendido % (p-1)) + 1;
        }

        proximo_primo++;
      }

      if(proximo_primo > fim_vetor_marcacao)
      {
        for(i = 0; i < p-1; i++)
        {
          unsigned int fim_primos = 0;
          MPI_Send(&fim_primos, 1, MPI_UNSIGNED, proximo_proc_atendido, 1, MPI_COMM_WORLD);
          proximo_proc_atendido = (proximo_proc_atendido % (p-1)) + 1;
        }
      }

      indice = fator_quadrado;
    }
    else
    {
      //Receive o primo designado
      MPI_Recv(&fator, 1, MPI_UNSIGNED, ID_ROOT, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);

      //Verifica se acabaram-se os primos
      if(fator == 0)
        break;

      indice = (max_fator - (max_fator % fator)) + fator;
      //printf("%d\tFator: %d\tIndice: %d\n", id, fator, indice);
    }

    while(indice <= fim_vetor_marcacao)
    {
      lista[indice] = TRUE;
      indice += fator;
    }
  }

//  printf("%d\t", id);
//  imprimir_array_char(lista, n);

  //Condensar array
  unsigned int elementos_completos;
  condensar_array_bits(lista, n+1, &elementos_completos);

  unsigned char *receive = NULL;
  if(id == ID_ROOT)
    receive = (unsigned char*)malloc((elementos_completos+1) * sizeof(unsigned char));

  //Reduzir
   MPI_Reduce(lista, receive, elementos_completos+1, MPI_UNSIGNED_CHAR, MPI_BOR, ID_ROOT,
       MPI_COMM_WORLD);

  free(lista);

  //Totalizar primos
  unsigned int total_primos = 0;
  if(id == ID_ROOT)
  {
    total_primos = contar_nao_marcados_array_condensado(receive, n+1);
    free(receive);
  }
```

```
  return total_primos;
}

int main (int argc, char *argv[])
{
  double elapsed_time;          /* Parallel execution time */
  unsigned int global_count;    /* Global prime count */
  int id;                       /* Process ID number */
  unsigned int n;               /* Sieving from 2, ..., 'n' */
  int p;                        /* Number of processes */

  MPI_Init (&argc, &argv);

  /* Start the timer */
  MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Barrier(MPI_COMM_WORLD);
  elapsed_time = -MPI_Wtime();

  if (p < 2)
  {
    if(id == 0)
      printf("Erro. Sao necessarios no minimo 2 processos. %d encontrados. Abortando.\n", p);
    MPI_Finalize();
    exit (1);
  }

  if (argc != 2)
  {
    if(id == 0)
      printf ("Command line: %s <n>\n", argv[0]);
    MPI_Finalize();
    exit (1);
  }

  n = (unsigned int)atoi(argv[1]);

  global_count = crivo_paralelo_hibrido(n, id, p);

  /* Stop the timer */
  elapsed_time += MPI_Wtime();

  /* Print the results */
  if (id == ID_ROOT)
  {
/*    printf ("There are %u primes less than or equal to %u\n", global_count, n);
    printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);*/
    printf("%d\t%lf\n", p, elapsed_time);
  }

  MPI_Finalize();
  return 0;
}
```

B Listagem Completa dos Resultados Experimentais

Algoritmo Serial

n	Tempo (s)	Desv. Pad.	n	Tempo (s)	Desv. Pad.
10000000	0.634596	0.113139	260000000	23.785732	4.795384
20000000	1.408606	0.259457	270000000	24.768746	4.988454
30000000	2.219366	0.415025	280000000	25.736725	5.170845
40000000	3.057232	0.576345	290000000	26.746751	5.393101
50000000	3.911844	0.743749	300000000	27.749821	5.599202
60000000	4.788419	0.920813	310000000	28.743274	5.801773
70000000	5.685587	1.105236	320000000	29.742482	6.004041
80000000	6.589690	1.286276	330000000	30.760400	6.218154
90000000	7.491798	1.466003	340000000	31.856989	6.517052
100000000	8.418306	1.647727	350000000	32.928497	6.786062
110000000	9.342369	1.836209	360000000	33.913207	6.966144
120000000	10.284431	2.029592	370000000	34.930567	7.184491
130000000	11.220344	2.221740	380000000	35.947616	7.397196
140000000	12.160793	2.412720	390000000	36.977286	7.619802
150000000	13.113513	2.605586	400000000	38.078233	7.843584
160000000	14.066478	2.802002	410000000	39.161139	8.094428
170000000	15.006344	2.990522	420000000	40.205501	8.311025
180000000	15.988933	3.184298	430000000	41.253897	8.523183
190000000	16.949454	3.390194	440000000	42.291313	8.741578
200000000	17.930203	3.588106	450000000	43.336711	8.959442
210000000	18.902619	3.788200	460000000	44.362341	9.168172
220000000	19.867600	3.984978	470000000	45.409594	9.400590
230000000	20.815348	4.178639	480000000	46.464634	9.635311
240000000	21.802778	4.382146	490000000	47.574885	9.928154
250000000	22.788998	4.584443	500000000	48.739465	10.256409

Paralelismo de Dados

p	Tempo (s)	Desv. Pad.
1	61.987324	0.022618
2	31.646074	0.017986
3	21.287851	0.010578
4	16.080351	0.008872
5	12.602935	0.016693
6	10.347113	0.010864
7	9.768111	0.012755
8	8.628697	0.007817

Paralelismo de Controle

p	Tempo (s)	Desv. Pad.
1	68.868025	0.073262
2	48.894132	0.072700
3	45.276104	0.041349
4	38.963062	0.006684
5	42.050781	0.057713
6	40.229580	0.006720
7	38.922296	0.007131
8	34.180629	0.010747

Solução Híbrida

p	Tempo (s)	Desv. Pad.
2	76.540804	0.020091
3	54.411614	0.012013
4	43.528246	0.013059
5	44.742977	0.012013
6	41.983470	0.009083
7	40.444249	0.005437
8	35.440083	0.043668

Referências

[1] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] Donald Knuth. *The Art of Computer Programming - Vol 3: Searching and Sorting*. Addison-Wesley, 1973.

[3] MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883, Portland, OR, November 1993. IEEE CS Press.

[4] Michael J. (Michael Jay) Quinn. *Parallel programming in C with MPI and OpenMP*. McGraw-Hill, pub-MCGRAW-HILL:adr, 2003.

[5] R. L. Rivest e C. Stein T. H. Cormen, C. E. Leiserson. *Introduction to Algorithms, 2ed.* McGraw-Hill, 2001.

[6] Wikipedia. Hash table — Wikipedia, the free encyclopedia, 2006. [Online; accessed 27-Jun-2006].

[7] Wikipedia. Prime counting function — Wikipedia, the free encyclopedia, 2006. [Online; accessed 26-Jun-2006].

[8] Wikipedia. RSA — Wikipedia, the free encyclopedia, 2006. [Online; accessed 27-Jun-2006].

[9] Wikipedia. Sieve of Eratosthenes — Wikipedia, the free encyclopedia, 2006. [Online; accessed 26-Jun-2006].

[10] Nívio Ziviani. *Projeto de Algoritmos com Implementações em Pascal e C*. Pioneira Thomson, 2004.