Universidade de Brasília - UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programação Paralela – 2/2016
Marcelo de Araujo Lopes Júnior - 150016794
André Fernandes Freire - 150005539

Exercício de Programação - 1

1. Instruções de Compilação

Para a compilação foi utilizado um utilitário make, assim, para compilar o exercício é necessário apenas ir a pasta ./src e utilizar o comando make.

```
$ cd src
$ make
```

2. Instruções de Execução

Para executar o programa basta, no diretório onde está localizado o Makefile (./src) e digitar ./testEP1 <limite> <opção> <numero de threads>

EX:

```
./testEP1 1000 time 4
```

Isto executará o programa para calcular os números primos de 0 a 1000 com 4 threads e mostrará ao final da execução o tempo que levou para realizar o cálculo.

3. Descrição do Algoritmo

O algoritmo calcula todos os números primos menores que um determinado limite, retornando ao usuário o tempo de execução do cálculo, os números calculados ou a soma de todos os primos calculados, além disso há uma opção que imprime todas as saídas anteriores.

Para se determinar os números primos todos os números menores que um limite foram verificados. É testado se o número é divisível por outro além dele mesmo e o número 1, para isso não é necessário testar todos os números menores que ele, mas apenas até um limite que corresponde à raiz quadrada dele.

Os testes também não precisam verificar a divisibilidade do número por todos os outros menores que sua raiz, mas apenas os números primos menores desse intervalo, diminuindo o número de testes.

O algoritmo desenvolvido buscas seguir essas regras para diminuir os testes, nele temos as seguintes funções:

- getPrimos - Função que testa todos os números de um intervalo e retorna uma lista com os números primos dele;
- printList - Função que imprime a lista de primos na tela;
- printTime - Função que imprime o tempo de execução na tela;
- printSum - Função que imprime a soma de todos os números primos encontrados.

4. Descrição da implementação paralela

O algoritmo partiu de uma versão sequencial que consistia em um loop que testava se o número era divisível por algum número presente em uma lista de primos, caso fosse primo era adicionado a essa lista. O algoritmo sequencial seguia a regra de testar apenas até um limite que corresponde à raiz do próprio número.

Não é possível paralelizar o algoritmo sequencial mantendo sua estrutura, então foi necessário dividí-lo em duas seções paralelas. A primeira seção calcula todos os números primos menores que a raiz do maior número do intervalo desejado. Nessa seção os números são comparados com todos os números menores que sua raiz e as comparações são realizadas paralelamente.

Cada thread adiciona os números primos encontrados em uma lista, para isso foi determinada uma seção critica que previne os possíveis casos de corrida. As threads também somam os números primos obtidos em uma variável global que vai conter o somatório de todos os números primos do intervalo.

A lista obtida será ordenada assim que estiver completa e pode-se então calcular os demais números usando apenas a lista de primos para a comparação, descartando assim todos os números múltiplos. Essas comparações são executadas na segunda seção paralela, mas dessa vez cada thread gera uma lista particular contendo os primos encontrados por ela, somando-os também ao somatório dos números.

Assim que a seção é finalizada, as listas particulares são concatenadas com a lista encontrada na primeira seção, formando assim uma lista contendo todos os números primos do intervalo de interesse.

5. Ambiente de Testes

Para a realização dos testes foi utilizado um computador com as seguintes especificações:

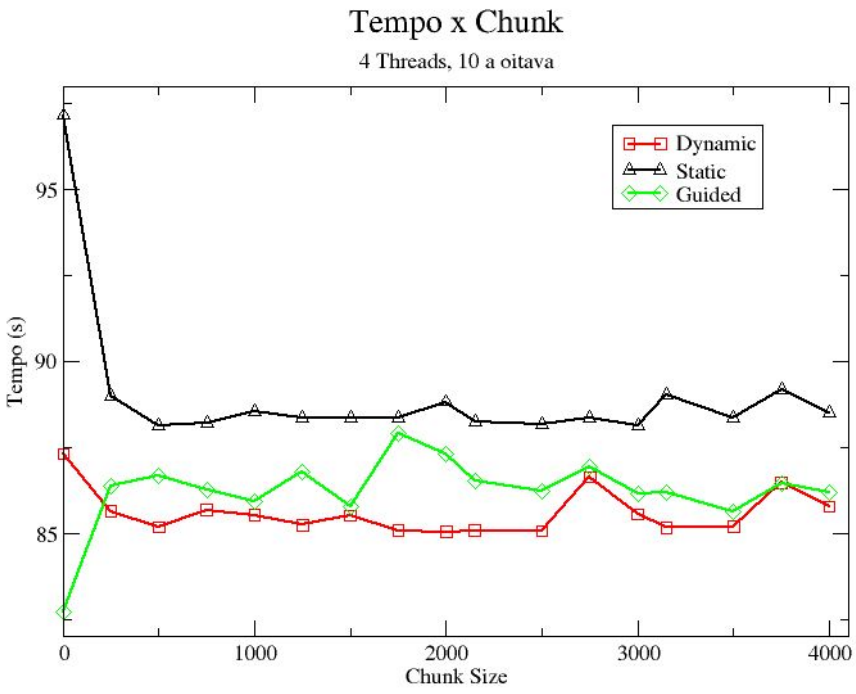
1. RAM: 8GB
2. HD: 1TB
3. Processador: Intel Core i7 4500u - 4MB Cache, 1.8GHz até 3.0 GHz []
4. HyperThread - sim

6. Testes

Durante a realização dos testes, obtivemos diversos resultados. Variando entre diversas configurações de escalonamento, tamanho de entrada, número de threads e tamanho do *"chunk".*

Todos os testes foram realizados com um limite de 10^8, o suficiente para termos diferenças mais significantes entre os tempos.

6.1 - Variações do chunk

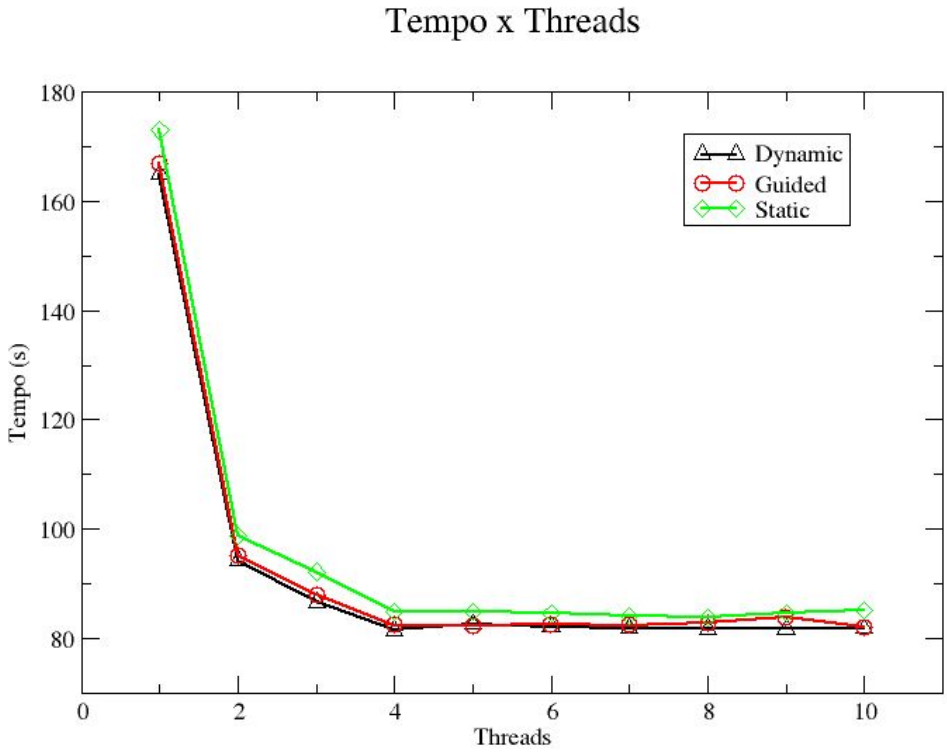


Podemos ver que há muitas diferenças entre os diversos escalonamentos, o pior foi o static devido não conseguir balancear muito bem a carga de trabalho nas diversas threads, já entre os melhores, dynamic e guided, o guided apresentou todos os tempos melhores que o escalonamento estático, porém uma média maior que o dinâmico. Já o dinâmico para um chunk de tamanho 1, obteve um resultado pior se comparado ao guided, porém com um chunk de tamanho 500, os resultados foram melhores e tiveram uma média menor que o escalonamento guiado, com um baixo desvio padrão.

	Média	**Desvio Padrão**
Estático	89,0001	2,0624
Dinâmico	85,6016	0,6236
Guiado	86,2970	1,0346

6.2 - Tempos

Tempos foram obtidos executando o programa 5 vezes e obtendo a média dos tempos obtidos, para vários escalonamentos e números de threads diferentes executando o programa.



Podemos ver do gráfico que como o gráfico anterior o escalonamento dinâmico resultou em tempos menores, mas não tão menores quanto o escalonamento em vermelho, porém foi alguns segundos melhor que o escalonamento estático, tornando-o o pior escalonamento novamente.

Podemos ver as menores diferenças entre os tempos. Pegamos o menor tempo obtido em cada um e comparamos.

	Diferenças
Estático - Guiado	1,9138
Estático - Dinâmico	2,2986
Dinâmico - Guiado	0,4848

6.3 - Melhores configurações

A partir dos gráfico feitos a partir das amostras de testes, podemos ver qual configuração nos obteve um melhor resultado, sendo ela o escalonamento dinâmico com um *chunk size* de tamanho 250, ou 500, e a partir de 4 threads. Assim a partir de agora para realizar a medição de algumas métricas importantes nessa aplicação como o speedup e a eficiência, usaremos estas configurações, com 4 threads.

6.4 - *Métricas*

Linear *Speedup:*

$$T_{parallel} = T_{serial} / cores$$

$$T_{parallel} = \frac{166.734}{4} = 41.683$$

Speedup:

$s = \frac{T_{serial}}{T_{parallel}}$, assim, podemos calcular o speedup real de nossa aplicação.

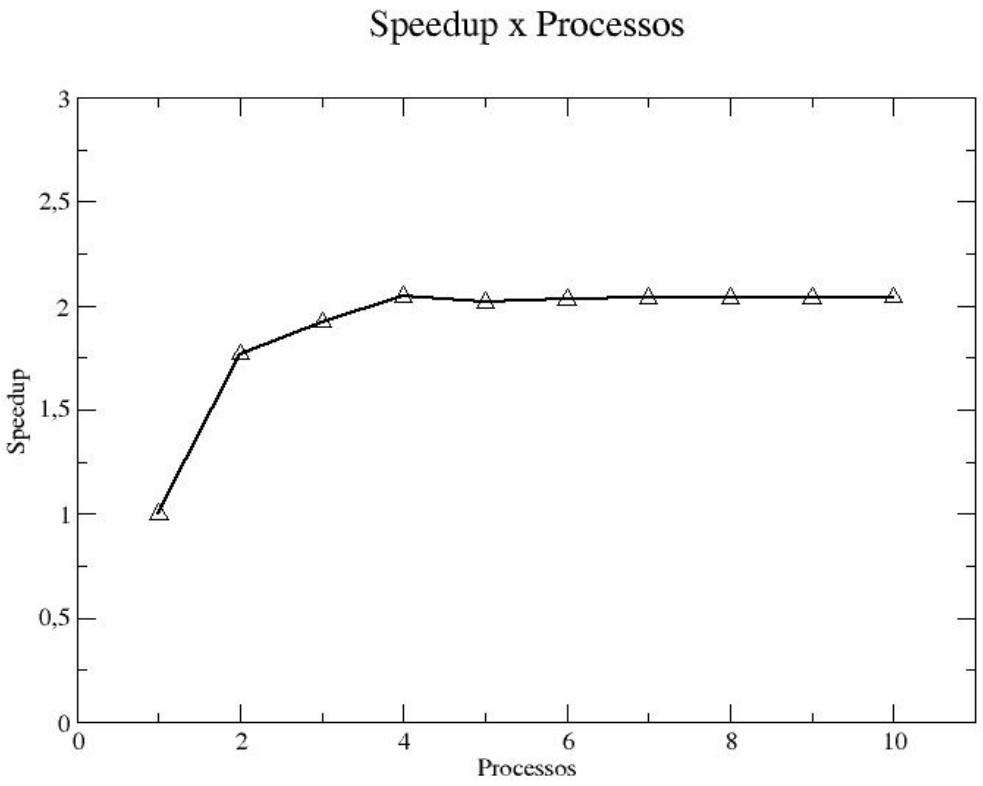
$s = \frac{166,734}{81,6099} = 2,0431$, assim temos o speed up.

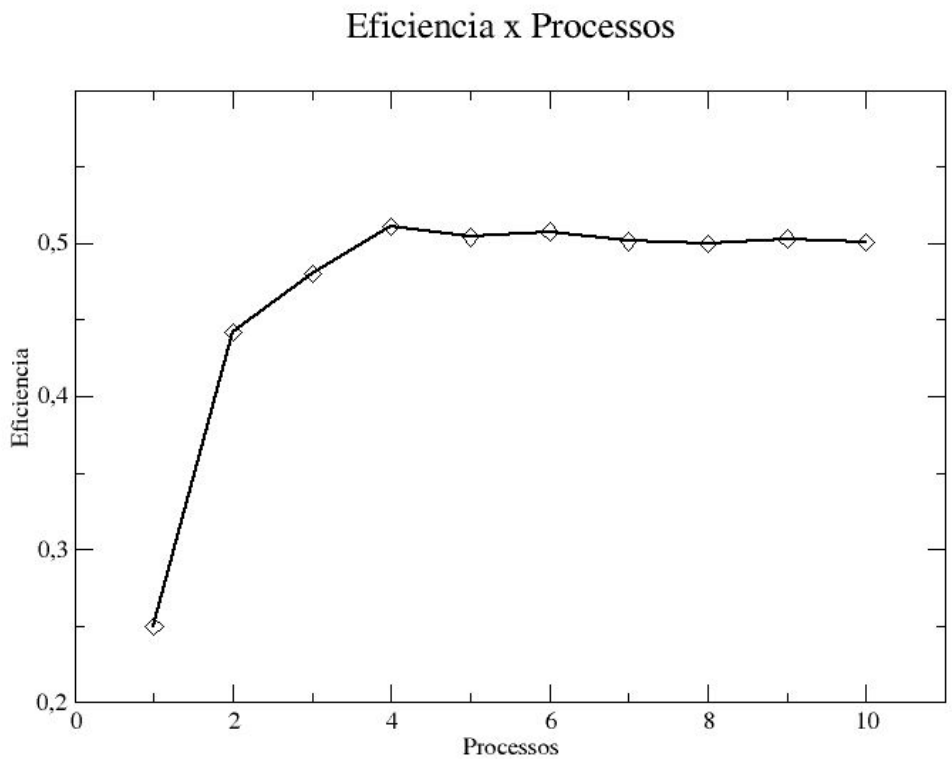
Eficiência:

$E = \frac{s}{cores}$,assim podemos ver a eficiência para o número de cores usados (4)

$$E = \frac{2,0431}{4} = 0.510775$$

6.4.1 - Aplicando essas métricas ao tempos obtidos em relação a outras threads temos os seguintes gráficos, representando a eficiência e o *speedup* para o número de processos.





7. Análise de resultados

7.1 - O que valeu a pena:

A maior parte do gasto de tempo de nosso algoritmo é na procura pela existência de um divisor de um determinado número, utilizar uma lista de números primos que foram achados até a raíz do limite nos garante que os próximos números a serem verificados após a raíz do número limite deverão ter um divisor dentro desta lista, graças a uma propriedade dos números, onde todo número pode ser escrito como um produto de primos, e que se o número é primo, o seu divisor está até a raiz de tal número.

Assim essa garantia ajudou a reduzir bastante o tempo de execução do algoritmo serial e paralelo, limitando o limite de verificação.

7.2 - O que não valeu a pena:

Uma parte do nosso algoritmo realiza a concatenação das sublistas usadas por cada thread concatenando-as de forma ordenada, tentar paralelizar essa parte é extremamente complicado, e já que o tempo gasto por essa parte para um limite de dez a nona não superou mais de 2 segundos, não sendo um grande problema no final.

7.3 - Estrategias a bordadas anteriormente:

Anteriormente foi feito um algoritmo extremamente simples para a solução, ele consistia em buscar todos os divisores de 2 até o número desejado para descobrir se era primo ou não, porém realizava muitas comparações desnecessárias. Em seguida fizemos um que a comparação era feita de 2 até a raíz do número, nesse, o tempo melhorou bastante, porém a estratégia adotada foi bastante melhor.