UNIVERSIDADE FEDERAL DE SANTA MARIA
SISTEMAS OPERACIONAIS
SISTEMAS DE INFORMAÇÃO

RHAUANI WEBER AITA FAZUL

ANÁLISE DE DESEMPENHO NO CÁLCULO DE NÚMEROS PRIMOS UTILIZANDO IMPLEMENTAÇÃO SEQUENCIAL, COM PROCESSOS E COM THREADS

Santa Maria, RS, Brasil
Maio de 2017

SUMÁRIO

_

1. INTRODUÇÃO

O presente relatório foi escrito no intuito de comparar a eficiência da procura de números primos em um vetor de diferentes tamanhos com três formas distintas de implementação: Cáculo sequencial, fazendo uso de threads (via biblioteca *pthreads*) e utilizando processos (via chamada de sistema *fork* e utilizando *pipes*).

Os testes foram realizados com diferentes parâmetros de entrada, os quais representam o tamanho do vetor a ser preenchido de forma randômica e a quantidade de threads e processos a serem criados em cada busca. Para a análise de tempo foi feito uso da função *clock_getTime()* pertencente à biblioteca *time.h*. Cada comparação levou em conta trinta execuções do programa para a realização da média aritmética da quantidade de tempo necessária para a busca dos números primos.

Além da comparação entre as formas de busca, também foi feita uma análise envolvendo a melhor quantidade de threads ou processos a serem criados para uma busca mais efetiva, tendo em vista que, ao atingir um determinado ponto, o aumento do número de threads/processos criados acaba prejudicando o desempenho. Tal resultado pode variar de acordo com o processador utilizado para os testes.

2. RESULTADOS

A seguir serão apresentados os resultados encontrados fazendo uso das diferentes formas de implementação.

2.1 SEQUENCIAL

Ao fazer a busca por números primos de forma sequencial em um vetor de diferentes tamanhos foram encontrados os seguintes resultados:

SEQUENCIAL							
tam. vetor	10	100	1000	10000	100000	1000000	10000000
tempo(s)	0,0000233	0,0000493	0,000149	0,0010042	0,00971147	0,10535983	0,9513406

Figure 1 – Resultados encontrados na busca por números primos de forma sequencial

Como esperado, o tempo médio de execução é crescente em função do aumento do tamanho do vetor. Também é visto que o crescimento pode ser considerado exponencial a partir do momento em que o vetor se torna suficientemente grande.

2.2 THREADS

Fazendo uso de threads pode ser observado que o tempo de execução cresce de forma mais lenta conforme o aumento do tamanho do vetor.

THREADS							
tam. vetor	10	100	1000	10000	100000	1000000	10000000
tempo(s)	0,00010717	0,00016383	0,0001843	0,00080443	0,0059235	0,05205167	0,48782723

Figure 2 – Resultados encontrados fazendo uso de threads

Para os testes acima observados foram criadas quatro threads para cada busca, logo cada thread procura números primos em ¼ do vetor.

2.3 PROCESSOS

Utilizando a chamada de sistema *fork* foram criados quatro processos para percorrer o vetor e encontrar números primos.

PROCESSOS							
tam. vetor	10	100	1000	10000	100000	1000000	10000000
tempo(s)	0,00041857	0,0006391	0,00066013	0,0015277	0,0104659	0,10880903	0,98739763

Figure 3 – Resultados encontrados utilizando processos para o cálculo de números primos

Cada processo, assim como as threads, percorre ¼ do vetor. A estrutura entre processos criada foi a de *master-slave* onde o processo pai, rhauani-t2SO, apenas delega as funções para seus filhos.

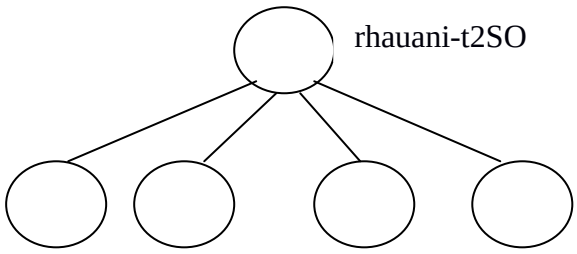

Figura 4 – Estrutura de árvore utilizada

Para evitar ocorrências de processos “zumbis”, o processo pai espera seus filhos acabarem.

3. ANÁLISE E COMPARAÇÃO

O gráfico a seguir, utilizando as médias de tempo de execução vistas na sessão anterior, compara o desempenho das três implementações.

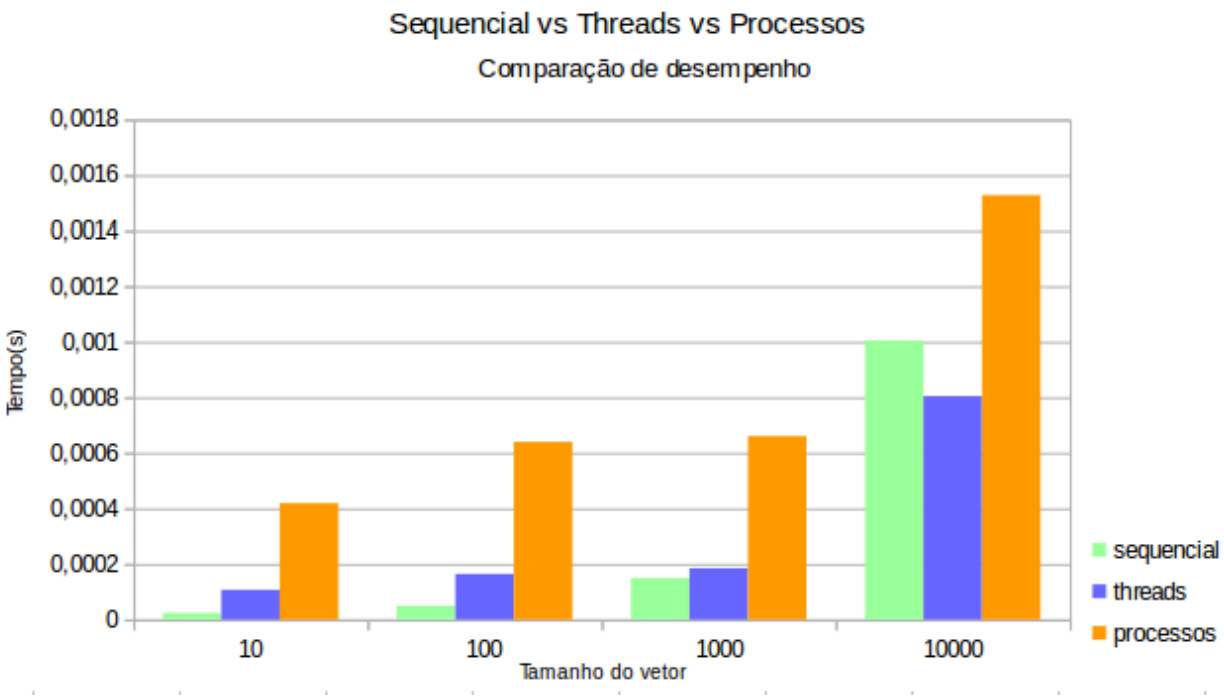

Figure 5 – Gráfico comparando os diferentes modos de busca de números primos em um vetor

Conforme ilustrado acima, percebemos que inicialmente a busca sequencial possui um desempenho superior tanto as threads, quanto aos processos. Porém, ao aumentar o tamanho do vetor, a utilização de threads acaba trazendo mais benefícios, superando o método puramente sequencial.

Isso se da pelo fato de que com poucos números para analisar, o tempo gasto para criar novos processos e threads e, no caso destes últimos, o grande *overhead* da troca de contexto, acaba por não compensar o uso dessas estruturas. Em contrapartida, conforme o tamanho do vetor aumenta, ou seja, muito processamento envolvido, as vantagens trazidas pela distribuição da carga de trabalho entre as threads acabam por se tornar muito mais aparentes.

Quanto aos processos, ao utilizar uma implementação similar a Figura 4, a grande demanda de controle e gerenciamento entre os mesmos não consegue ser diluída a ponto de fazer seu desempenho ser superior ao da busca sequencial, que faz uso de um único processo ao invés de cinco (processo pai mais quatro filhos).

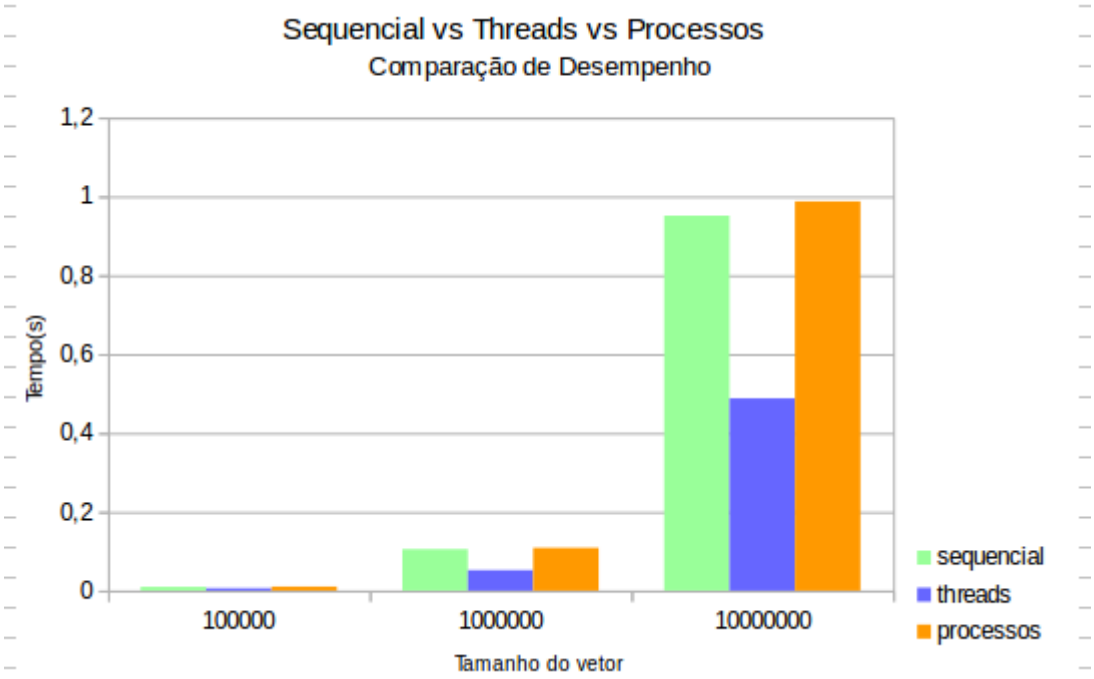

Figure 6 – Desempenho utilizando vetores maiores

Todavia, é possível perceber, conforme o gráfico acima, que se o tamanho do vetor aumentar a valores muito altos, os tempos de execução utilizando processos tendem a se igualar com os da implementação convencional de busca de números primos, embora o uso de threads ainda possua um desempenho muito maior.

Por fim, foi realizada uma análise da melhor quantidade de threads e processos a serem utilizados para a resolução do problema. Foram encontrados os seguintes resultados:

THREADS							
qtd thread	2	4	8	10	50	100	1000
tempo(s)	0,00083137	0,0006985	0,00083273	0,00091773	0,00252433	0,0035016	0,02525217

PROCESSOS							
qtd proc.	2	4	8	10	50	100	1000
tempo(s)	0,00181823	0,00213017	0,003088	0,00342047	0,01165127	0,02263863	0,2914642

Figure 7 – Número de threads e processos Vs. desempenho

Embora varie de acordo com o computador utilizado para os testes, podemos perceber que ao utilizar um grande número de threads ou processos o desempenho é significativamente prejudicado. Isso se deve, além do aumento da complexidade envolvida no gerenciamento dessas

estruturas, ao limite físico do computador (quantidade de núcleos, etc), ou seja, embora existam muitas threads/processos disponíveis para serem escalonados, não existe recurso (processador) disponível até que o processo anterior acabe.