

UFRN

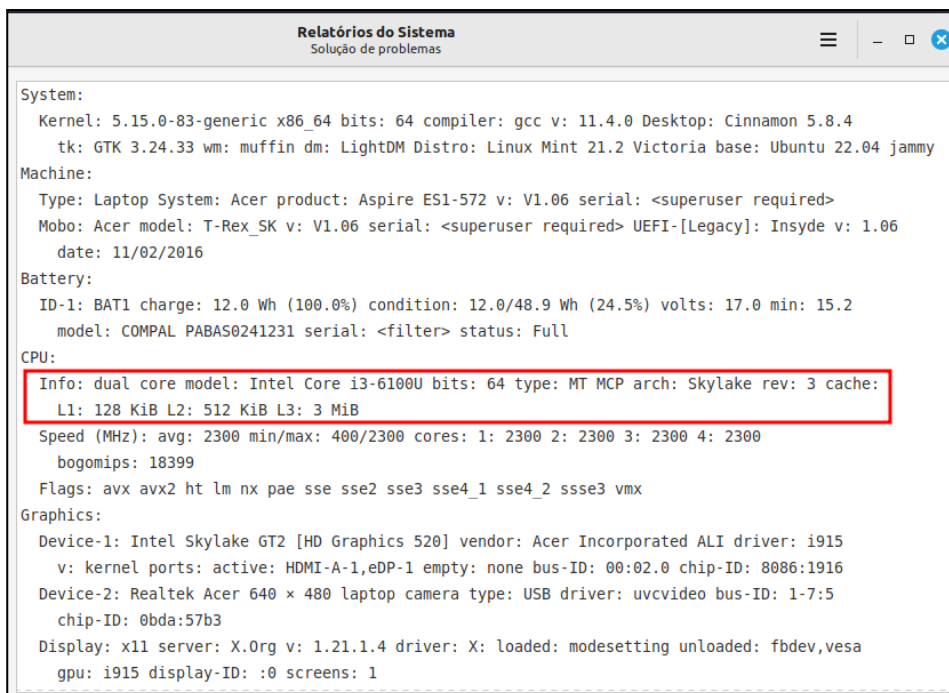
TECNOLOGIA DA INFORMAÇÃO
IMD - NATAL - BACHARELADO - N
Disciplina de Sistemas Operacionais

Trabalho Prático da Unidade 1: Processos e Threads

Flávio Roberto Guerra Seabra - 20200040620 - T03 (2023.2)

Características do Computador

As figuras 1A, 1B e 1C mostra as características do computador no qual os experimentos foram executados, evidenciando tratar-se de um modelo dual core, fator este que será importante na discussão dos resultados obtidos.



```
System:
  Kernel: 5.15.0-83-generic x86_64 bits: 64 compiler: gcc v: 11.4.0 Desktop: Cinnamon 5.8.4
  tk: GTK 3.24.33 wm: muffin dm: LightDM Distro: Linux Mint 21.2 Victoria base: Ubuntu 22.04 jammy
Machine:
  Type: Laptop System: Acer product: Aspire E51-572 v: V1.06 serial: <superuser required>
  Mobo: Acer model: T-Rex_SK v: V1.06 serial: <superuser required> UEFI-[Legacy]: Insyde v: 1.06
  date: 11/02/2016
Battery:
  ID-1: BAT1 charge: 12.0 Wh (100.0%) condition: 12.0/48.9 Wh (24.5%) volts: 17.0 min: 15.2
  model: COMPAL PABAS0241231 serial: <filter> status: Full
CPU:
  Info: dual core model: Intel Core i3-6100U bits: 64 type: MT MCP arch: Skylake rev: 3 cache:
  L1: 128 KiB L2: 512 KiB L3: 3 MiB
  Speed (MHz): avg: 2300 min/max: 400/2300 cores: 1: 2300 2: 2300 3: 2300 4: 2300
  bogomips: 18399
  Flags: avx avx2 ht lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx
Graphics:
  Device-1: Intel Skylake GT2 [HD Graphics 520] vendor: Acer Incorporated ALI driver: i915
  v: kernel ports: active: HDMI-A-1,eDP-1 empty: none bus-ID: 00:02.0 chip-ID: 8086:1916
  Device-2: Realtek Acer 640 x 480 laptop camera type: USB driver: uvcvideo bus-ID: 1-7:5
  chip-ID: 0bda:57b3
  Display: x11 server: X.Org v: 1.21.1.4 driver: X: loaded: modesetting unloaded: fbdev,vesa
  gpu: i915 display-ID: :0 screens: 1
```

Figura 1A - Configurações do computador

```
flaviorgs@flaviorgs-Aspire-ES1-572: ~/Área de Trabalho/SO
Arquivo Editar Ver Pesquisar Terminal Ajuda
(base) flaviorgs@flaviorgs-Aspire-ES1-572:~/Área de Trabalho/SO$ ./ex2_p
thread_affinity 8
Numero de cores: 4
Essa é a execução de thread 0 da CPU: 2
Essa é a execução de thread 1 da CPU: 3
Essa é a execução de thread 2 da CPU: 1
Essa é a execução de thread 3 da CPU: 3
Essa é a execução de thread 5 da CPU: 3
Essa é a execução de thread 4 da CPU: 0
Essa é a execução de thread 7 da CPU: 1
Essa é a execução de thread 6 da CPU: 0
(base) flaviorgs@flaviorgs-Aspire-ES1-572:~/Área de Trabalho/SO$
```

Figura 1B - Impressão no terminal após a execução de código contendo o comando mostrado na figura 1C mostrando que o computador no qual os testes foram feitos dispõe de 4 núcleos.

```
NPROC = sysconf(_SC_NPROCESSORS_ONLN);
printf("Numero de cores: %d\n", NPROC);
```

Figura 1C - Trecho de código utilizado para mostrar a quantidade de núcleos disponíveis na CPU (figura 1B) do computador no qual foi executado o teste

Experimento 1

O gráfico 1, mostra a execução do código Sequencial nos diferentes tamanhos de matrizes.

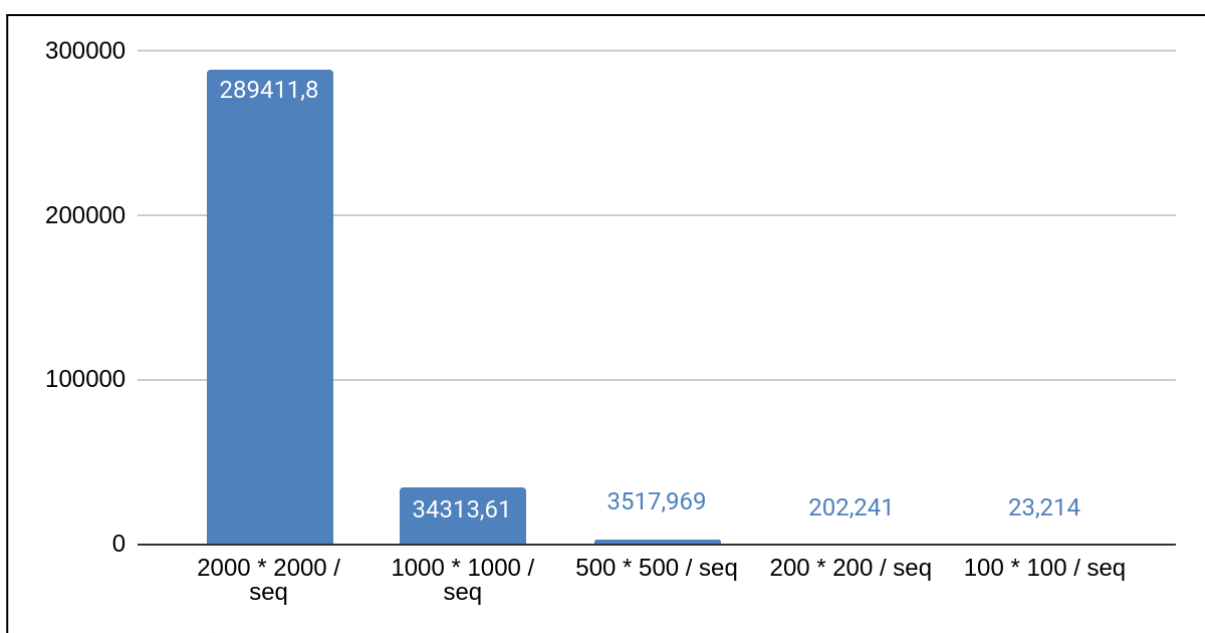


Gráfico 1 - Tempos, em milissegundos das execuções do código sequencial, dos diferentes tamanhos de matrizes

Evidencia-se nesse gráfico a diferença nos tempos de execução em relação aos tamanhos das matrizes. Enquanto o código sequencial consegue executar a multiplicação de matrizes de tamanho de até 200 x 200 em menos de 1 segundo e de matrizes de 500 x 500 e 1000 x 1000 em menos de 1 minuto, para as matrizes de 2000 x 2000 foram necessários aproximadamente 5 minutos em média para realizar a multiplicação.

O gráfico 2 mostra todas as execuções com os diferentes valores de P dos códigos Threads e Processos, bem como, para efeito de comparação, do código Sequencial, o qual não se aplica a variação de valor de P.

Para melhor visualização dos valores a serem comparados, o gráfico 3 foi construído apenas com os valores das multiplicações das matrizes menores que 1000 x 1000 e o gráfico 4 foi construído apenas com os valores das multiplicações das matrizes até o tamanho de 200 x 200. Ressalte-se que os dados de ambos os gráficos 3 e 4 também constam no gráfico 2.

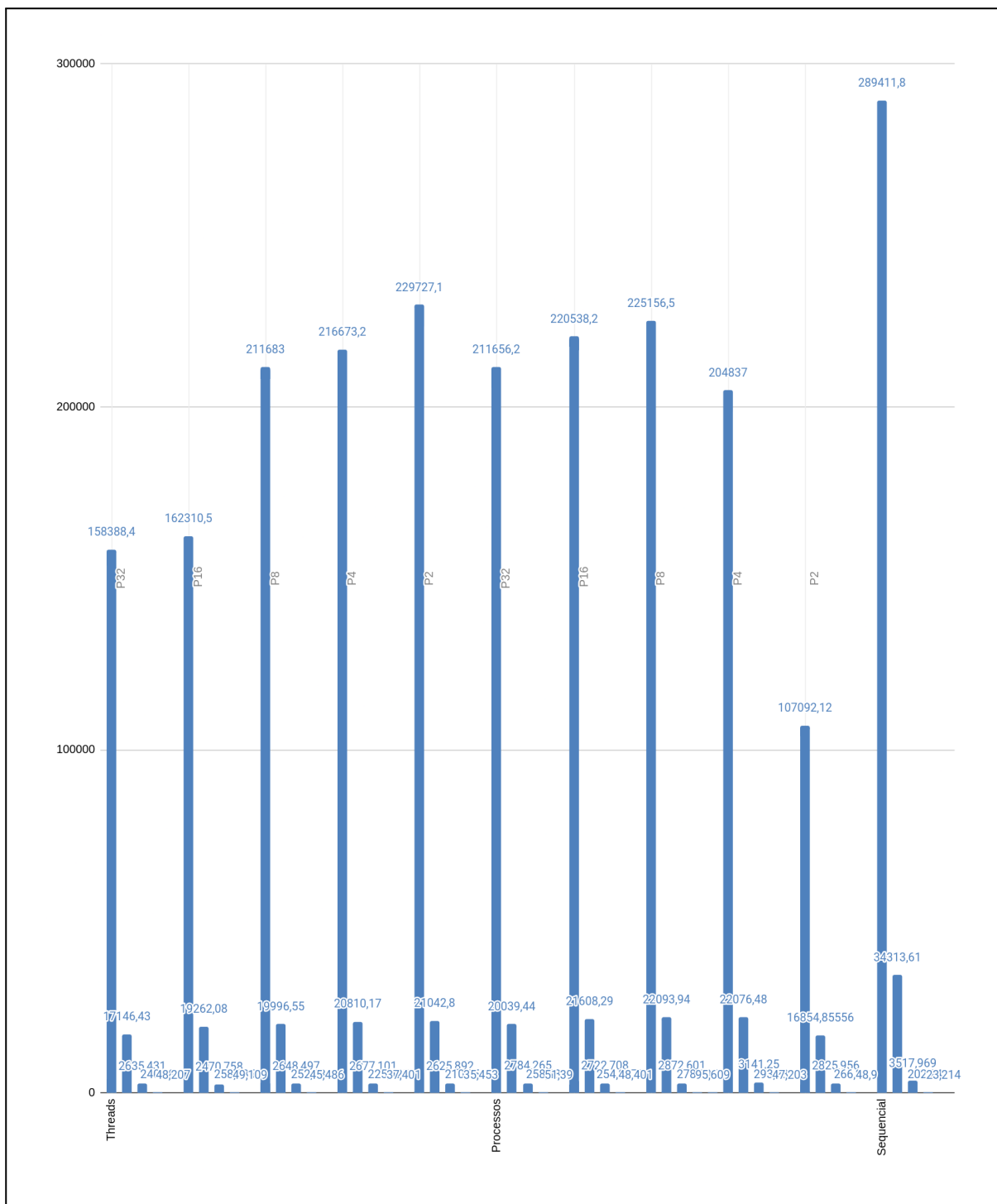


Gráfico 2 - Tempos, em milissegundos das execuções dos códigos Threads e Processos com valores de P=2, P=4, P=8, P=16 e P=32, e do Sequencial, dos diferentes tamanhos de matrizes (2000 x 2000, 1000 x 1000, 500 x 500, 200 x 200 e 100 x 100)

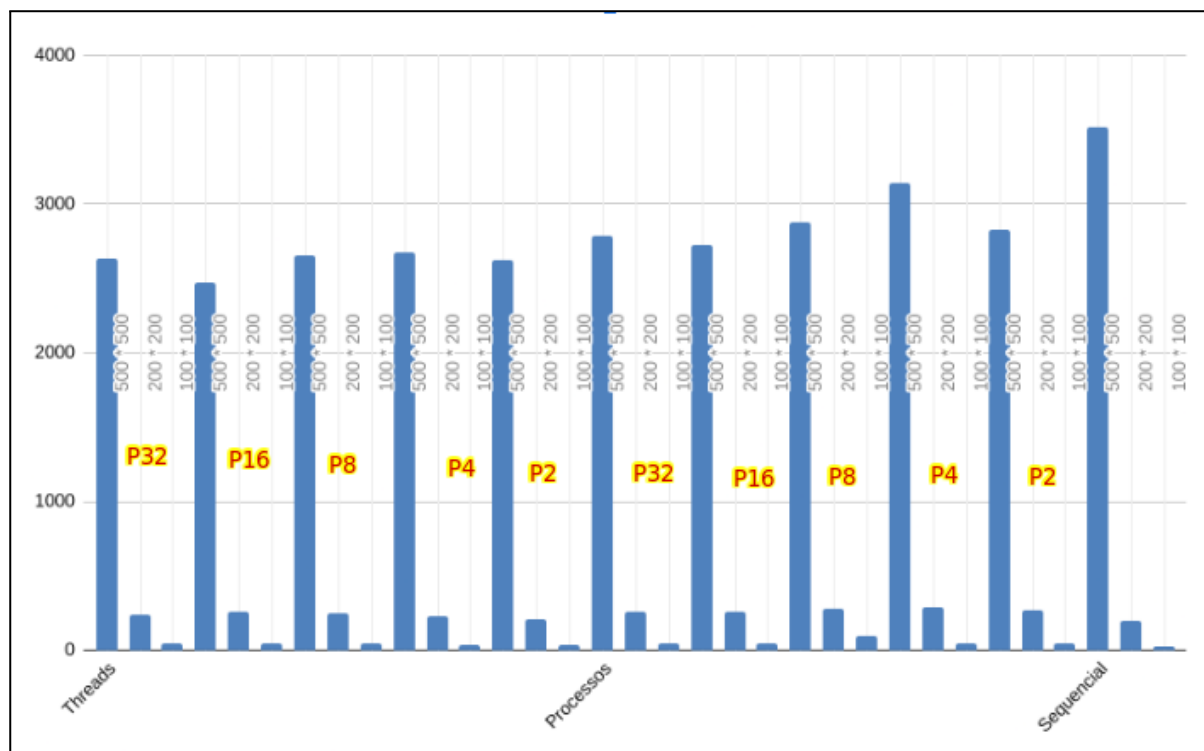


Gráfico 3 - Tempos, em milissegundos das execuções dos códigos Threads e Processos com valores de P=2, P=4, P=8, P=16 e P=32, e do Sequencial, dos tamanhos de matrizes 500 x 500, 200 x 200 e 100 x 100.

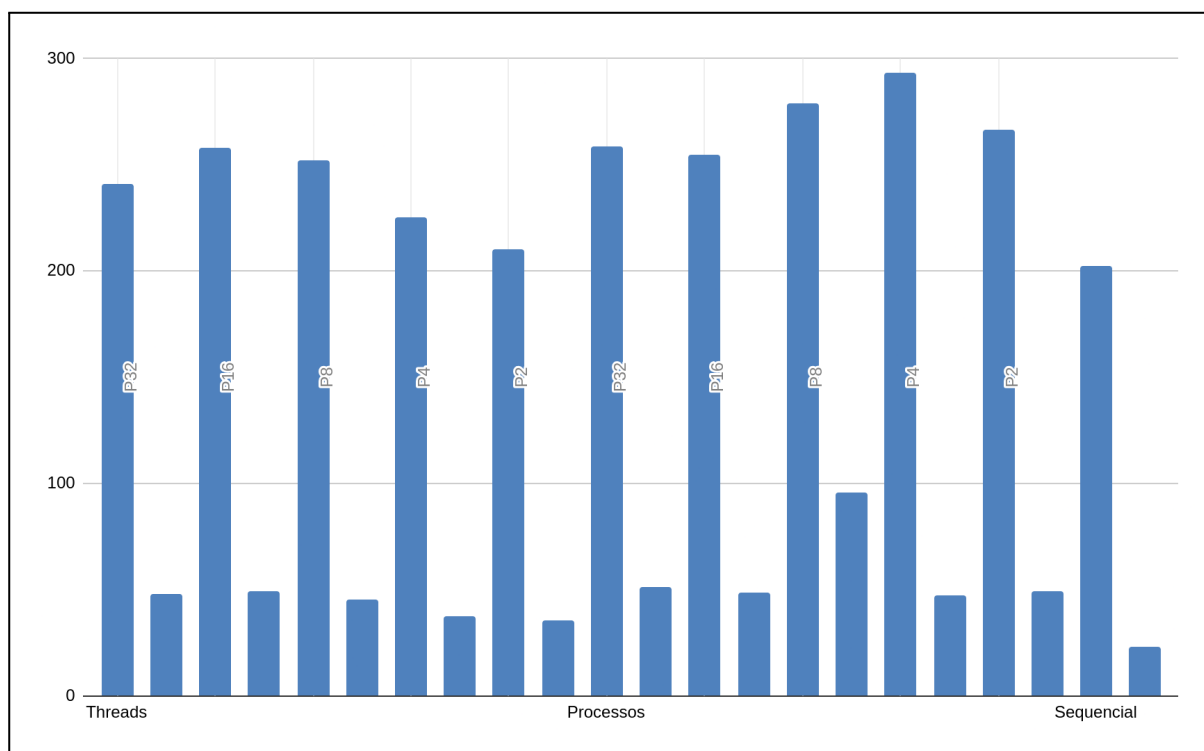


Gráfico 4 - Tempos, em milissegundos das execuções dos códigos Threads e Processos com valores de P=2, P=4, P=8, P=16 e P=32, e do Sequencial, dos tamanhos de matrizes 200 x 200 e 100 x 100.

No gráfico 2, é possível ver que o tempo de execução das multiplicações das matrizes de tamanho acima de 500 x 500 é reduzido nos códigos Threads e Processos em relação ao código Sequencial para todos os valores de P. O mesmo ocorre para as matrizes de tamanho 500 x 500.

No entanto, pela questão das escalas do gráfico 2, é mais fácil de se observar esse dado no gráfico 3.

Para as matrizes de até 200 x 200, esse fenômeno se inverte e o código sequencial apresentou-se mais eficiente que os códigos Threads e Processos para todos os valores de P, como é possível se verificar no gráfico 4.

Experimento 2

O gráfico 5 mostra a execução dos códigos Sequencial, Threads e Processos, com diferentes valores de P apenas nas matrizes de tamanho 2000 x 2000.

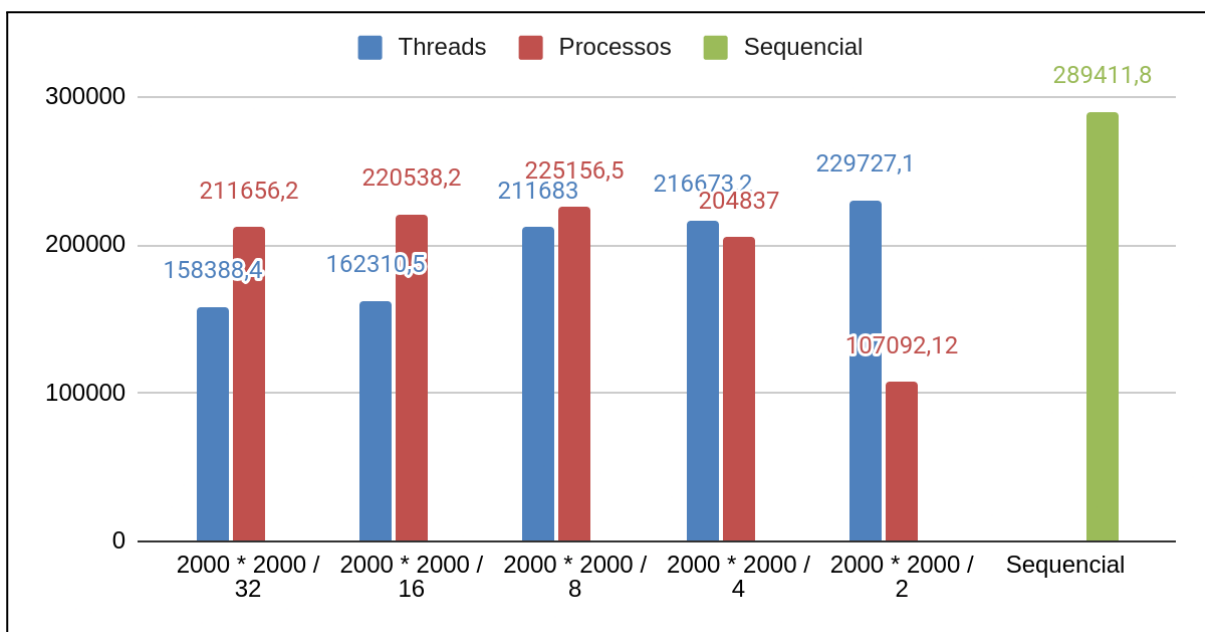


Gráfico 5 - Tempos, em milissegundos das execuções dos códigos Threads e Processos com valores de P=2, P=4, P=8, P=16 e P=32, e do Sequencial, das matrizes de tamanho 2000 x 2000.

Observa-se que a eficiência da multiplicação das matrizes 2000 x 2000 é maior nos códigos que executam a multiplicação utilizando a estratégia de threads e processos que no código que executa esta multiplicação de forma sequencial, como já tinha sido inclusive demonstrado no gráfico 2.

É perceptível também o aumento da eficiência de execução do código Threads à medida que se aumenta o valor de P, consequentemente, reduzindo-se a quantidade de threads criadas.

No entanto, o efeito inverso foi percebido no código Processos, onde à medida que se reduziu o valor de P, consequentemente aumentando-se a quantidade de processos criados, aumentou-se a eficiência da multiplicação.

Discussões

As multiplicações de matrizes menores foram sempre executadas em tempos menores, o que é possível ver em todos os gráficos. Isso, obviamente, é o esperado pois quanto menor a quantidade de dados a serem processados menor o tempo para que esse processamento seja feito.

Para entendermos os pontos seguintes da discussão, temos que entender como foi feita a implementação dos processos e das threads no código.

As figuras 2 e 3 mostram a forma como foram feitas as implementações das threads no código Threads e dos processos no código Processos.

```
// Calcula o número de threads com base em P
int tamA = (int)sizeA;
int numThreads = sizeA / P; ← A
if ((tamA * tamA) % P != 0) {
    numThreads++;
}

////////// threads //////////
vector<thread> threads;

int particao = tamA * tamA / numThreads; ← B

for (int i = 0; i < numThreads; ++i) {
    int start = i * particao;
    int end = (i == numThreads - 1) ? tamA * tamA : (i + 1) * particao;
    threads.emplace_back(multiplyAndWriteResult, i, start, end, tamA);
}

// Aguarda todas as threads terminarem
for (auto& thread : threads) {
    thread.join();
}
```

Figura 2 - Trecho do código Threads que mostra como foi feita a implementação das threads. Em A o trecho que mostra o cálculo de quantas threads seriam implementadas em função da variável P passada pelo argumento argv. Em B a forma usada para particionar os valores das matrizes para que cada thread executasse apenas a parte das multiplicações que lhe cabia.

```
// Calcula o número de processos com base em P
int tamA = (int)sizeA;
int numProcesses = sizeA / P; ← A
if ((tamA * tamA) % P != 0) {
    numProcesses++;
}
```

Figura 3 - Trecho do código Processos (A) que mostra que foi feita uma implementação semelhante à implementação feita no código Threads (Figura 2).

Nas implementações é possível notar que quanto maior o valor da variável P menor o valor variável *numThreads* no código Threads e *numProcesses* no código Processos, o que resulta, por sua vez, em valores maiores dos locais onde as matrizes seriam particionadas. Isto resulta em uma execução com menor quantidade de threads e/ou processos, assim cada thread/processo precisa executar uma quantidade maior de multiplicações (Figura 4).

↑ P	↓ Threads/ Processos	↑ Quantidade de multiplicações por cada thread/ processo
↓ P	↑ Threads/ Processos	↓ Quantidade de multiplicações por cada thread/ processo

Figura 4. Relação entre valor de P, quantidade de threads e a quantidade de multiplicações executadas por cada thread.

Era de se esperar que quando a implementação fosse feita na forma de threads ou de processos houvesse uma redução no tempo de execução das multiplicações das matrizes em relação à implementação sequencial. Esse efeito foi observado para matrizes maiores que 200 x 200 (gráfico 2), mas não para as matrizes menores (gráfico 4).

Esse fato é explicado pela latência do despacho.

Quando há a mudança de execução de um processo para outro ou uma thread para outra, a CPU precisa ser interrompida, parando o que está fazendo naquele momento, e dando início à execução de outro processo ou thread. Essa ação recebe o nome de mudança de contexto. Para que a mudança de contexto ocorra é necessário que o sistema operacional guarde a informação do ponto onde aquele processo ou thread foi interrompido e, em seguida, inicie a execução do outro processo ou thread. Todo esse processo é controlado pelo despachante, ou dispatcher, e leva um certo tempo para acontecer. A esse tempo é dado o nome de Latência do Despacho.

O tempo gasto com a latência do despacho se mostra compensador no caso de matrizes maiores porque o ganho na eficiência das multiplicações é tal que supera o aumento do tempo gasto pela própria latência do despacho. No entanto, no caso da multiplicação de matrizes menores, o tempo adicionado pela latência do despacho acaba superando o tempo que a multiplicação sequencial leva para executar toda a tarefa.

Esses fatores devem ser ainda considerados no contexto de que o experimento foi feito em um computador equipado com processador dual core (Figura 1A, 1B e 1C) com 4 núcleos, o que torna a quantidade de threads e de processos que podem ser executadas simultaneamente bastante limitada devido à competição pelos 4 núcleos do processador, resultando em diminuição da eficiência.

Na análise apenas da multiplicação das matrizes 2000 x 2000, é possível verificar no gráfico 5 que a implementação na forma de threads resulta em aumento do tempo de execução à medida em que se aumenta a quantidade de threads (redução do valor de P) embora os tempos sempre se mantenham menores que o tempo da execução sequencial.

A redução do tempo em relação à implementação sequencial é explicada pelo fato de que cada thread precisa executar uma quantidade menor de multiplicações que a multiplicação sequencial.

Já o aumento do tempo quando se aumenta o número de threads pode ser explicado pelo fato do experimento ter sido executado em um computador dual core e o overhead de criação e gerenciamento de threads se torne mais significativo do que os benefícios da paralelização quando o número de threads excede em muito o número de núcleos disponíveis.

Na implementação por processos, a eficiência se manteve relativamente inalterada entre $P=32$ e $P=4$ e depois aumentou consideravelmente em $P=2$. Como houve a implementação por particionamento, cada processo executa as operações em uma parte diferente da matriz. Além disso, como cada processo tem seu próprio espaço de endereço de memória e não compartilha recursos com outros processos, a criação de múltiplos processos acabou sendo uma abordagem mais eficiente do que a criação de múltiplas threads.

As matrizes de 500 x 500 pareceram, no presente experimento, serem um ponto de corte onde não se observou diferenças significativas entre implementação em threads versus implementação em processos nem entre quantidade de threads e de processos. Para matrizes maiores a implementação em threads e/ou processos torna a multiplicação mais eficiente, sendo a eficiência maior na implementação em processos do que em threads e quando as quantidades de processos filhos criados são maiores.

Conclusão

Considerando o uso de um computador equipado com processador dual core:

- matrizes menores permitem maior eficiência de multiplicação que matrizes maiores, independentemente do tipo de implementação sequencial, threads ou processos;
- há maior eficiência na multiplicação de matrizes pequenas, de até 200 x 200, quando a implementação é feita na forma sequencial que quando é feita na forma de threads e/ ou processos;
- para matrizes a partir de 500 x 500 a implementação em threads e/ou processos possibilita maior eficiência na multiplicação;
- para matrizes de 500 x 500 não foi possível verificar diferença significativa na eficiência entre as implementações por threads e por processos nem entre as quantidades de threads e processos de cada implementação;
- para matrizes a partir de 1000 x 1000, observou-se maior eficiência na implementação com menor número de threads comparado à implementação com maior número de threads, bem como com implementação com maior número de processos comparado à implementação com menor número de processos;
- a implementação considerada mais eficiente para matrizes maiores, de 2000 x 2000, foi a implementação com o maior número possível de processos filhos.

Códigos utilizados:

Os códigos utilizados bem como as medições feitas em cada execução podem ser encontrado no link: https://github.com/fseabra93/trab_SO_1unid.git

Link para o vídeo:

https://youtu.be/sqg0hgho_RI