Experimento de Performance com e sem Threads

AT2/N1 - Atividade prática coletiva

Lucas Emanuel Rego Von Lohrmann – UC22103108

Guilherme Yago Valente – UC22200104

Lucas Carvalho da Silveira – UC22200038

Eduardo Coelho Dos Santos e Silva – UC22200329

O que são Threads

Threads são pequenas unidades de execução dentro de um processo que permitem a execução de múltiplas tarefas simultaneamente. De acordo com Castro e Chamon (1998), uma thread pode ser vista como uma “linha de execução” que compartilha os mesmos recursos de um processo principal, mas que pode operar de forma independente, tornando possível o paralelismo na execução de tarefas computacionais. Diferente de um processo completo, que possui seu próprio espaço de memória, uma thread compartilha o espaço de memória do processo em que está inserida, possibilitando a troca rápida de informações entre elas sem a necessidade de mecanismos de comunicação mais complexos.

Como Threads Funcionam Computacionalmente

O funcionamento computacional das threads está intimamente ligado à forma como os sistemas operacionais e os processadores gerenciam a execução de tarefas. Um processo pode ser considerado uma entidade de controle de execução, e dentro desse processo, uma ou mais threads podem ser geradas. Segundo Maia e Machado (1997), as threads operam de forma independente, mas compartilham os mesmos recursos do processo, como memória e arquivos abertos, facilitando a troca de dados entre elas e reduzindo o overhead que seria necessário na comunicação entre processos distintos.

Cada thread tem seu próprio contador de programa, registradores e pilha, mas compartilha o restante dos recursos do processo. Isso permite que um sistema operacional execute várias threads simultaneamente, usando a técnica de **time-slicing** (fatia de tempo) ou **multithreading preemptivo**, onde o tempo de CPU é dividido entre as threads em execução. Em sistemas multicore, threads podem ser executadas em paralelo em núcleos diferentes, melhorando ainda mais a eficiência da execução.

Como o Uso de Threads Pode Afetar o Tempo de Execução de um Algoritmo

O uso de threads tem um impacto direto no tempo de execução de um algoritmo, especialmente quando se trata de tarefas que podem ser divididas em subtarefas independentes. Ao introduzir paralelismo no código, threads podem reduzir o tempo total de execução ao permitir que múltiplas operações sejam realizadas simultaneamente. Isso é particularmente benéfico em sistemas multicore, onde diferentes threads podem ser alocadas para diferentes núcleos do processador.

No entanto, nem sempre o uso de threads leva a uma melhoria de desempenho. Em situações onde as threads precisam compartilhar intensamente os mesmos dados ou acessar recursos críticos do sistema, pode ocorrer **contention** (disputa) por esses recursos, o que pode gerar o que é conhecido como **overhead** de sincronização. Em casos extremos, isso pode até aumentar o tempo de execução do algoritmo, conforme mencionado no artigo da IBM (2023). Quando múltiplas threads precisam sincronizar o acesso a variáveis compartilhadas, por exemplo, o tempo gasto esperando pela liberação desses recursos pode anular os ganhos de performance obtidos pelo paralelismo.

Relação Entre Computação Concorrente e Paralela e a Performance dos Algoritmos

A computação concorrente e a paralela estão diretamente ligadas ao uso de threads e à performance de algoritmos. No modelo de computação concorrente, várias tarefas são executadas de forma que parecem estar sendo realizadas simultaneamente, mas na verdade estão sendo alternadas pelo processador em intervalos muito curtos. A concorrência utiliza threads para realizar essa alternância, permitindo que tarefas como leitura de arquivos ou comunicação em rede não bloqueiem o fluxo principal de um programa.

Já o modelo de computação paralela vai além: ele envolve a execução simultânea de múltiplas tarefas em diferentes processadores ou núcleos de um processador. De acordo com o glossário da Escola LBK (2023), na computação paralela, as threads podem ser alocadas a diferentes núcleos, permitindo a verdadeira execução simultânea. Isso geralmente resulta em um aumento significativo de performance, especialmente em algoritmos que podem ser divididos em tarefas menores e independentes.

A relação entre esses dois modelos e a performance de algoritmos está no tipo de tarefa que se está realizando. Algoritmos com forte dependência de dados compartilhados ou que precisam ser executados de forma sequencial tendem a se beneficiar mais da computação concorrente, enquanto algoritmos que podem ser divididos em partes independentes se beneficiam enormemente da computação paralela.

No entanto, como destacado por Maia e Machado (1997), o aumento do número de threads ou processos não garante, por si só, uma melhora linear no desempenho, especialmente quando o overhead de comunicação e sincronização entre threads é elevado. A escolha correta entre computação concorrente e paralela deve ser feita com base no tipo de tarefa, na arquitetura de hardware e nos objetivos de desempenho do algoritmo.

Exibição e Explicação dos Resultados

O experimento comparou o tempo de execução de um programa de processamento de 320 arquivos CSV contendo dados de temperaturas de várias cidades, utilizando diferentes números de threads para otimizar a performance. Foram realizadas 20 versões do experimento, onde o número de threads variou conforme duas abordagens: uma em que as threads foram atribuídas para processar grupos de cidades, e outra onde foram criadas threads tanto para as cidades quanto para os anos processados.

Análise dos Resultados:

1. Threads por Cidades

Nos primeiros 10 experimentos, as threads foram atribuídas proporcionalmente à quantidade de cidades: uma thread para 320 cidades, duas para 160 cidades, quatro para 80 cidades, e assim por diante. Isso significa que conforme o número de threads aumentava, cada thread processava um número menor de cidades, permitindo a distribuição da carga de trabalho.

Os resultados mostram que o uso de threads impacta diretamente no tempo de execução do programa. No **Experimento 1** (1 thread), o tempo médio de execução ficou entre **4748 ms e 5437 ms**, com uma leve variabilidade entre as rodadas. Já no **Experimento 2** (2 threads), o tempo médio caiu para a faixa de **3528 ms a 4228 ms**, representando uma melhoria significativa de performance.

Conforme o número de threads aumentava, houve uma diminuição contínua no tempo de execução até o **Experimento 5** (16 threads), onde o tempo médio variou entre **3200 ms e 3566 ms**. No entanto, após esse ponto, o ganho de performance com o aumento do número de threads foi menos acentuado, com os resultados começando a estabilizar ou até mesmo aumentar levemente no **Experimento 10** (320 threads), onde o tempo médio ficou entre **3406 ms e 4076 ms**. Esse fenômeno pode ser atribuído ao overhead gerado pela criação e gerenciamento de muitas threads, que acaba por anular os ganhos de performance.

Experimento	N Threads	Rodada									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1	5437	5117	5166	5000	4748	4888	4961	4975	5169	4798
2	2	3942	3989	4228	3630	3700	3616	3819	4076	3752	3528
3	4	3530	3381	3677	3701	3419	3390	3480	3426	3423	3538
4	8	3606	3427	3728	3391	3282	3448	3355	3442	3557	3275
5	16	3329	3566	3348	3369	3236	3200	3227	3503	3277	3312
6	32	3367	3135	3196	3296	3221	3311	3265	3242	3470	3280
7	64	3496	3250	3290	3460	3472	3238	3274	3302	3249	3134
8	80	3670	3240	3363	3369	3348	3409	3421	3066	3455	3315
9	160	3149	3502	3309	3521	3793	3413	3298	3612	3373	3329
10	320	4076	3470	3611	3614	3857	3472	3428	3819	3406	3463

2. Threads por Cidades e Anos

Nos experimentos 11 a 20, foi adotada uma abordagem mais granular, onde cada thread foi responsável pela leitura e processamento de 160 cidades, e, para cada cidade, criou-se uma thread adicional para cada ano existente nos dados. Esse aumento no número de threads deveria teoricamente reduzir ainda mais o tempo de execução, porém, o overhead associado ao gerenciamento de um grande número de threads começou a penalizar o desempenho.

No **Experimento 11** (1 thread), o tempo médio de execução variou entre **3709 ms e 6152 ms**, semelhante à primeira abordagem. No entanto, à medida que o número de threads aumentou, o tempo de execução diminuiu até o **Experimento 14** (8 threads), onde o tempo médio variou entre **3083 ms e 6194 ms**. Esse experimento apresentou resultados mistos, onde algumas

rodadas mostraram um tempo de execução relativamente baixo, mas outras indicaram uma sobrecarga significativa, sugerindo um possível ponto de ineficiência no uso de muitas threads para gerenciar as cidades e anos.

Nos experimentos subsequentes, especialmente no **Experimento 17** (64 threads) e nos experimentos seguintes, o tempo de execução começou a aumentar drasticamente, variando entre **6239 ms e 7875 ms**, no caso do experimento 17. Finalmente, no **Experimento 20** (320 threads), o tempo de execução médio foi extremamente alto, alcançando até **23603 ms**, evidenciando que a criação de um número muito elevado de threads gera mais overhead do que ganhos de performance.

Experimento	N Threads	Rodada									
		1	2	3	4	5	6	7	8	9	10
11	1	6152	4929	3821	4266	4093	4613	3851	3901	3709	4058
12	2	3759	3068	3141	3366	3154	3315	2834	2926	3081	3035
13	4	3938	3573	3072	3229	3142	3364	3172	3048	3105	3230
14	8	6194	3716	3359	3572	3514	3411	3083	3154	3542	3287
15	16	4342	4806	5897	3989	4688	5239	4977	3714	4029	3523
16	32	5355	4444	4243	5884	4279	4323	4249	5184	6246	5146
17	64	7875	6813	6501	6820	6386	6810	6601	6239	6371	6326
18	80	8499	8113	7997	7575	7155	7197	7245	6894	7095	6803
19	160	12496	11930	11689	11791	12230	11655	10756	10875	10724	10667
20	320	23603	21675	18470	26508	24275	23167	22032	22651	22866	20524

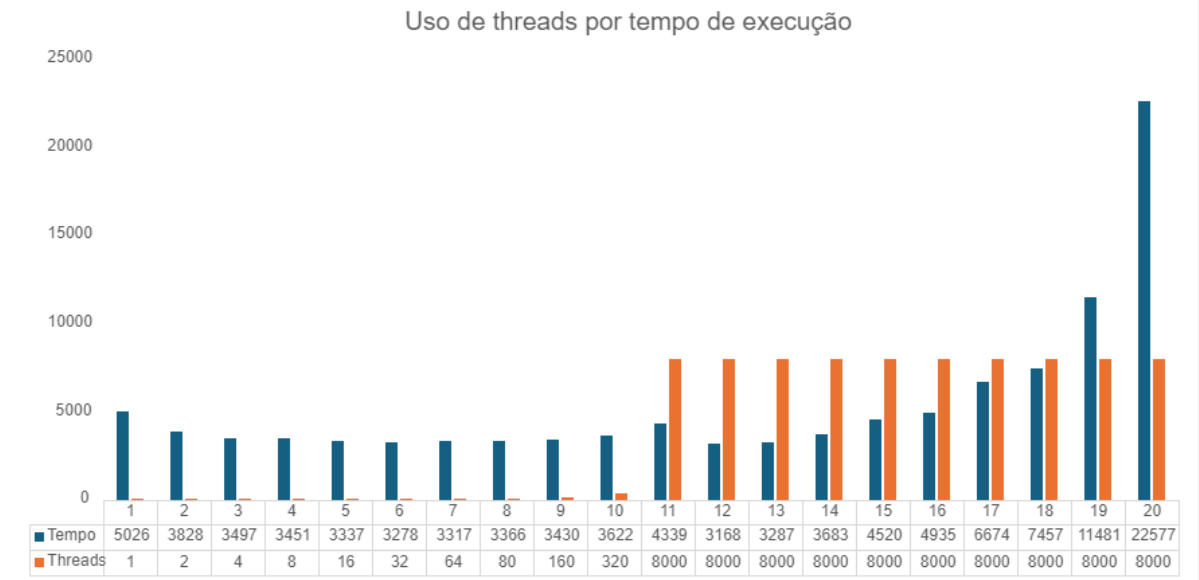
Comparação Geral

A comparação entre os experimentos revela que o uso moderado de threads pode proporcionar ganhos significativos em performance, enquanto o uso excessivo pode ser contraproducente. A abordagem mais eficiente ocorreu quando o número de threads foi bem balanceado em relação à carga de trabalho, como visto nos experimentos com 8 e 16 threads na primeira abordagem e 4 threads na segunda.

Gráfico de Comparação

O gráfico a seguir ilustra a relação entre o número de threads e o tempo médio de execução para as duas abordagens do experimento.

Experimento	N Threads	Rodada										
		1	2	3	4	5	6	7	8	9	10	MÉDIA
1	1	5437	5117	5166	5000	4748	4888	4961	4975	5169	4798	5026
2	2	3942	3989	4228	3630	3700	3616	3819	4076	3752	3528	3828
3	4	3530	3381	3677	3701	3419	3390	3480	3426	3423	3538	3497
4	8	3606	3427	3728	3391	3282	3448	3355	3442	3557	3275	3451
5	16	3329	3566	3348	3369	3236	3200	3227	3503	3277	3312	3337
6	32	3367	3135	3196	3296	3221	3311	3265	3242	3470	3280	3278
7	64	3496	3250	3290	3460	3472	3238	3274	3302	3249	3134	3317
8	80	3670	3240	3363	3369	3348	3409	3421	3066	3455	3315	3366
9	160	3149	3502	3309	3521	3793	3413	3298	3612	3373	3329	3430
10	320	4076	3470	3611	3614	3857	3472	3428	3819	3406	3463	3622
11	8000	6152	4929	3821	4266	4093	4613	3851	3901	3709	4058	4339
12	8000	3759	3068	3141	3366	3154	3315	2834	2926	3081	3035	3168
13	8000	3938	3573	3072	3229	3142	3364	3172	3048	3105	3230	3287
14	8000	6194	3716	3359	3572	3514	3411	3083	3154	3542	3287	3683
15	8000	4342	4806	5897	3989	4688	5239	4977	3714	4029	3523	4520
16	8000	5355	4444	4243	5884	4279	4323	4249	5184	6246	5146	4935
17	8000	7875	6813	6501	6820	6386	6810	6601	6239	6371	6326	6674
18	8000	8499	8113	7997	7575	7155	7197	7245	6894	7095	6803	7457
19	8000	12496	11930	11689	11791	12230	11655	10756	10875	10724	10667	11481
20	8000	23603	21675	18470	26508	24275	23167	22032	22651	22866	20524	22577



Referências Bibliográficas

- CASTRO, G.; CHAMON, V. *Dicionário de Informática*. 3. ed. Rio de Janeiro: Campus, 1998.
- MAIA, Luiz Paulo; MACHADO, Francis B. *Arquitetura de computadores*. 2.ed. Rio de Janeiro: LTC, 1997.
- IBM. *Understanding Threads and Processes*. Disponível em: https://www.ibm.com/docs/pt-br/aix/7.3?topic=programming-understanding-threads-processes. Acesso em: 12 set. 2024.
- Escola LBK. *O que é thread*. Disponível em: https://escolalbk.com.br/glossario/o-que-e-thread/. Acesso em: 12 set. 2024.