Relatório de Desenvolvimento do Primeiro Trabalho de Programação Concorrente

Igor Santana(83154), Leonardo Catharin(83196)

Departamento de Informática
Universidade Estadual de Maringá(UEM) – Maringá, PR – Brasil

{ra83154,ra83196}@uem.br

Resumo. *Este relatório refere-se ao desenvolvimento do primeiro trabalho da disciplina de Programação Concorrente, e mostra os resultados obtidos durante a implementação do Problema do Supermercado e as métricas coletadas.*

1. Introdução

Um programa concorrente pode ser visto como vários fluxos de execução originários de um fluxo principal. Por este motivo, a computação concorrente pode ser utilizada em problemas computacionais que exigem uma computação muito grande, como por exemplo a multiplicação de matrizes, descoberta de números primos, entre outros.

Cada *thread* do programa concorrente tem sua própria pilha de ativação de registros e sua própria cópia dos registradores do CPU, incluindo o ponteiro da pilha e o contador do programa, que juntos descrevem o estado da execução da *thread*. Entretanto, as *threads* em um ambiente *multithread* dividem os dados, código, recursos e o endereço de memória do processo que as originou (CARVER; TAI, 2006).

2. Descrição detalhada do problema

O problema resolvido no trabalho é chamado de **Problema do Supermercado** e é categorizado como um problema produtor-consumidor. O problema do produtor/consumidor consiste de um processo (ou *thread*) que irá produzir os dados da aplicação, e um para consumir os dados que foram produzidos, onde estes dados são armazenados num buffer enquanto esperam para serem utilizados.

O Problema do Supermercado consiste de um processo que irá produzir clientes que serão atendidos por caixas que estarão em processos separados. Cada caixa terá sua própria fila, e os clientes sempre serão adicionados na menor fila e, caso um cliente seja adicionado em uma fila, ele não pode trocar de fila. Caso um caixa tenha terminado de atender os clientes da sua fila, ele pode procurar um cliente em alguma outra fila para atender. Caso não exista nenhum cliente para ser atendido, os caixas ficam bloqueados a espera de outros clientes.

3. Modelagem e Implementação do Problema

Como dito acima, o problema proposto é um problema chamado de produtor-consumidor. Para a otimização do tempo de execução foram usadas técnicas de programação paralela. Entretanto é necessário ter algumas precauções para que os dados acessados afim da melhora de tempo sejam, em todo o decorrer do processamento, íntegros, ou seja, não exista problemas de integridade com os dados e nem de acesso indevido a memória que se torna compartilhada entre as *threads* utilizadas.

Em termos de implementação, foram usadas *threads* que são executadas paralelamente acessando uma memória compartilhada. Para que não ocorram os problemas citados acima, nessa memória foram utilizados semáforos e mutex para o controle de acesso.

As seguintes *threads* são geradas:

- Uma *thread* é utilizada para inserir clientes constantemente até que o número limite (estipulado no código pela constante 'numeroDeClientes') seja atingido. Logo, essa *thread* só é finalizada quando todos os clientes solicitados são inseridos.
- Ao mesmo tempo, são criados N caixas (estipulado no código pela constante 'numeroDeCaixas'). Cada caixa é uma *thread* e só é finalizado quando todos os clientes de todos os caixas são atendidos. Isso se deve, graças ao requisito que solicita que quando um caixa x tem sua fila = 0, porém um caixa y ainda tem clientes na fila para serem atendidos, o caixa x ajuda o caixa y a atendê-los.

Como a *thread* de adição de clientes na fila e as *threads* de atendimento (caixas) utilizam memória compartilhada (filas dos caixas), cada fila possui seu semáforo controlando o acesso a fila. O seguinte caso é possível:

- Se a fila de um caixa é 0, o caixa não deve atender, logo se a *thread* tentar executar a operação de atender o semáforo dessa fila coloca aquela *thread* para "dormir".

Consideramos que não há um limite para o tamanho de cada fila, dessa maneira não se faz necessário tratar um estouro na memória compartilhada.

Se não houvesse o semáforo, os clientes que não foram inseridos na fila seriam atendidos, ocasionando inconsistência de dados e consequentemente erro no resultado final.

Já a relação de atendimento entre caixas depende do escalonador do Sistema Operacional e é necessário uma variável mutex para controlar a permissão de acesso às filas dos caixas. Isso se faz necessário para que, por exemplo, um caixa x não acesse a fila de um caixa y ao mesmo tempo que o próprio caixa y, gerando assim dados inconsistentes.

4. Análise dos Resultados

O problema foi paralelizado em 2, 4, 8 e 16 *threads*. Juntamente com os resultados dos algoritmos paralelos, abaixo são apresentados os resultados do algoritmo sequencial. Dessa maneira, é possível fazer uma análise detalhada de desempenho em relação ao tempo de execução em cada um dos casos citados acima.

As análises se baseiam nas métricas vistas em sala. São elas: *SpeedUp*, Eficiência, Redundância, Utilização, Qualidade, Lei de Gustafson-Barsis, Lei de Amdahl e Karp-Flatt.

Os resultados são apresentados abaixo.

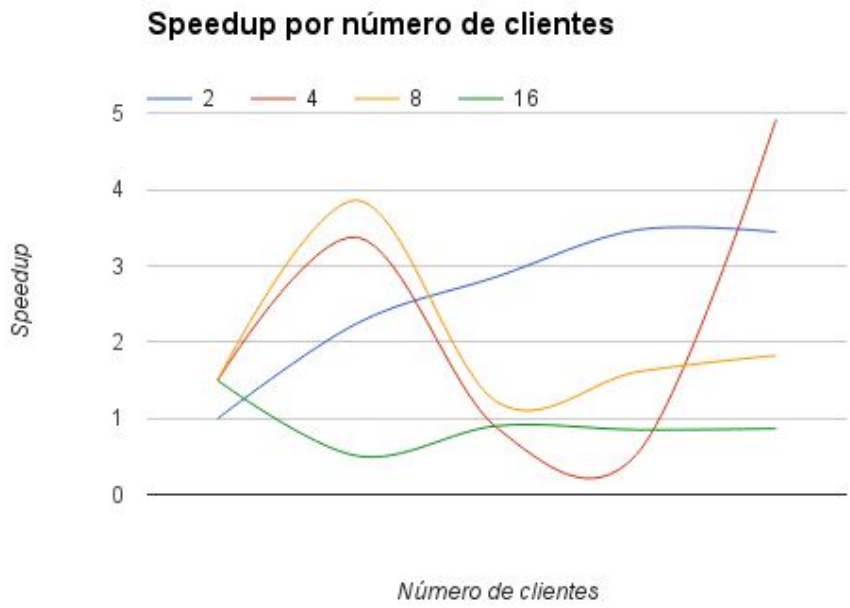
A tabela e o gráfico abaixo apresentam os tempos de execução obtidos em cada uma das execuções citadas:

Tempo de execução					
Clientes / Caixa	1	**2**	**4**	**8**	**16**
1.000	0,003	0,003	0,002	0,002	0,002
10.000	0,027	0,012	0,008	0,007	0,053
100.000	0,263	0,092	0,300	0,216	0,291
1.000.000	2,613	0,753	4,934	1,624	3,063
10.000.000	26,091	7,577	5,302	14,285	30,075

*A coluna preenchida se refere à execução sequencial.

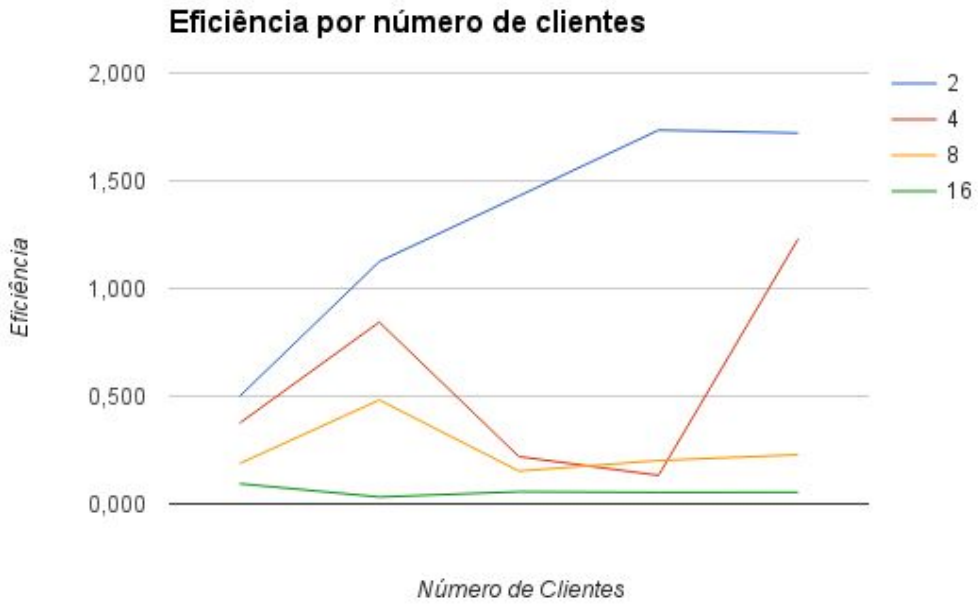
A tabela e gráfico à seguir apresentam os *SpeedUps* obtidos:

SpeedUp				
Clientes / Caixa	**2**	**4**	**8**	**16**
1.000	1,500	1,500	1,000	1,500
10.000	3,857	0,509	2,250	3,375
100.000	1,218	0,904	2,859	0,877
1.000.000	1,609	0,853	3,470	0,530
10.000.000	1,826	0,868	3,443	4,921



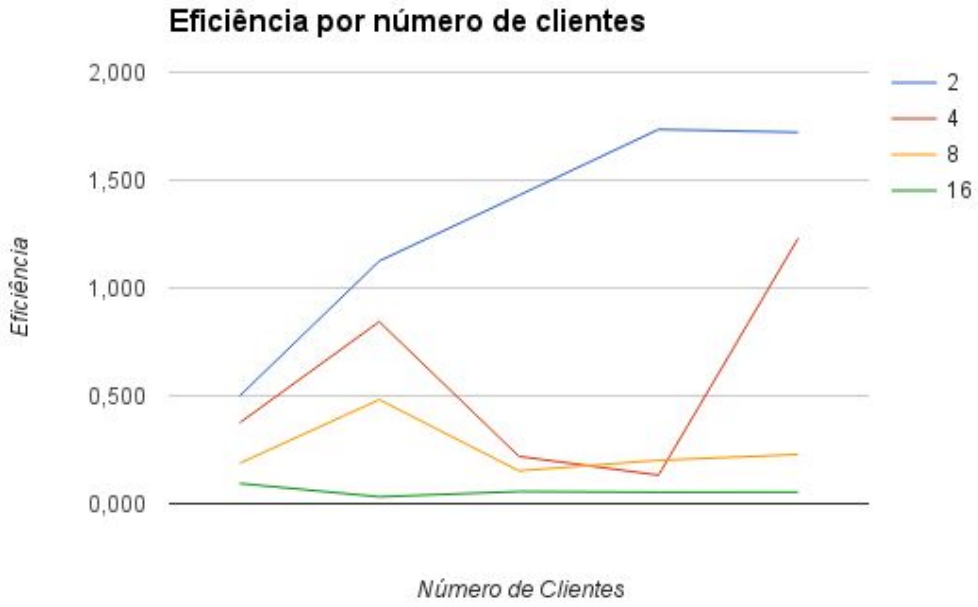
A tabela e gráfico à seguir apresentam os resultados de eficiência obtidos:

Eficiência				
Clientes / Caixa	**2**	**4**	**8**	**16**
1.000	0,500	0,375	0,188	0,094
10.000	1,125	0,844	0,482	0,032
100.000	1,429	0,219	0,152	0,056
1.000.000	1,735	0,132	0,201	0,053
10.000.000	1,722	1,230	0,228	0,054



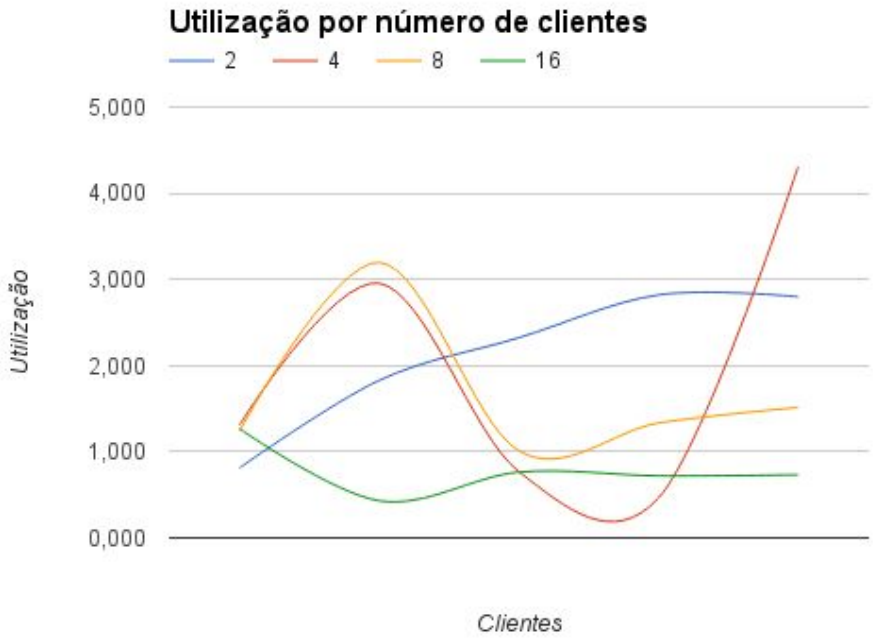
A tabela e gráfico à seguir apresentam os resultados de redundância obtidos:

Redundância			
2	**4**	**8**	**16**
1,625	3,502	6,629	13,510



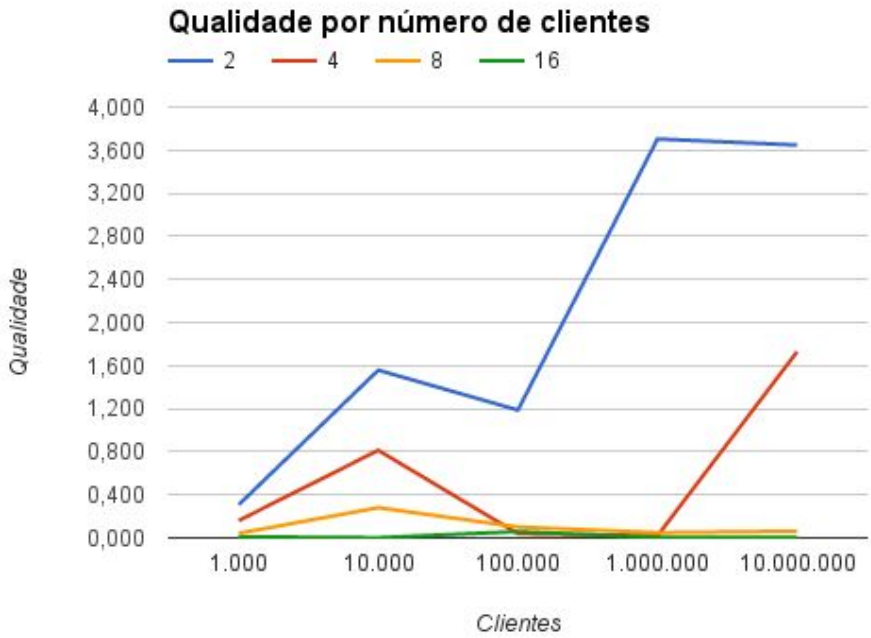
A tabela e gráfico à seguir apresentam os resultados de utilização obtidos:

Utilização				
Clientes / Caixa	**2**	**4**	**8**	**16**
1.000	0,813	1,313	1,243	1,267
10.000	1,829	2,955	3,196	0,430
100.000	2,323	0,768	1,009	0,763
1.000.000	2,820	0,464	1,333	0,720
10.000.000	2,799	4,308	1,514	0,733



A tabela e gráfico à seguir apresentam os resultados de qualidade obtidos:

Utilização				
Clientes / Caixa	**2**	**4**	**8**	**16**
1.000	0,308	0,161	0,042	0,010
10.000	1,557	0,813	0,281	0,001
100.000	1,187	0,039	0,101	0,059
1.000.000	3,704	0,020	0,049	0,003
10.000.000	3,647	1,729	0,063	0,003



Abaixo são apresentados os resultados obtidos a partir da lei de Amdahl. Baseando-se nela é possível afirmar que os resultados apresentados na tabela a seguir são os respectivos *SpeedUps* máximos da quantidade de cada processor testado.

Lei de Amdahl			
2	**4**	**8**	**16**
1,444	2,604	4,818	9,442

Assim como a lei de Amdahl, é possível encontrar o *SpeedUp* máximo a partir da lei de Gustafson-Barsis. Essa lei leva em consideração que a fração sequencial do código sequencial aumenta conforme aumentam os processadores. Os resultados obtidos através dessa lei são exibidos na tabela abaixo:

Lei de Gustafson-Barsis			
2	**4**	**8**	**16**
1,615	3,464	7,340	15,306

Na tabela abaixo, por fim são apresentados os resultados obtidos pela métrica de Karp Flatt:

Métrica de Karp Flatt			
2	**4**	**8**	**16**
1,000	0,556	0,619	0,644

Como os resultados obtidos aumentaram com o números de processadores, podemos concluir que existem custos altos relacionados a comunicação, sincronização e/ou inicialização das *threads* (notou-se esse comportamento entre 4 à 16 *threads*). Entre 2 e 4 *threads*, podemos concluir que não existem esse custos altos citados, porém sofre com pouco paralelismo.

5. Conclusões

Com o desenvolvimento do trabalho, pode-se concluir que a utilização do paralelismo através das *threads* faz com que o problema tenha um tempo de execução menor, desde que seja avaliado corretamente qual a melhor combinação de *threads* para a quantidade de dados que serão processados.

Caso o volume de dados processados seja pequeno, o custo com as operações de comunicação e sincronização fazem com que não se torne viável fazer a paralelização, tornando assim a solução sequencial mais apropriada.

Deve-se ter cuidado também com a região de memória que será compartilhada entre as *threads*, que podem causar problemas de sincronização das *threads*, tornando mais díficil para o programador achar os erros presentes no código. Uma simples inversão de instruções pode ocasionar com que o tempo de execução aumente considerávelmente, por exemplo.

As métricas utilizadas durante este relatório nos ajudaram a entender melhor o funcionamento do algoritmo paralelo durante o desenvolvimento, pois seguiram como um guia para aprimorar a aplicação.

Por fim, recomenda-se o uso do paradigma de programação concorrente quando não existe um acesso massivo das *threads* na região de memória compartilhada, pois isto irá acarretar em grandes esforços para escalonar as threads. Problemas como multiplicação de matrizes e busca em árvores.

6. Referências

1. CARVER, Richard H.; TAI, Kuo-chung. **MODERN MULTITHREADING:** Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs. Hoboken: John Wiley & Sons, Inc, 2006.
2. ROCHA, Ricardo. **Programação Paralela e Distribuída:** Métricas de Desempenho. Porto: Faculdade de Ciências da Universidade do Porto, 2009. Color.
3. FOSTER, Ian. Parallel Algorithm Examples. In: FOSTER, Ian. **Designing and Building Parallel Programs.** Argonne, Il: Pearson, 1995. p. 19-26.