5205 - Programação Concorrente: Primeiro Trabalho

Lucas Marçal Surmani[1], **Rafael Cortez Sanchez**[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
Maringá – PR – Brasil

ra84447@uem.br, ra82357@uem.br

Abstract. *This paper introduces parallel programming concepts, which are further used in the abstraction of a supermarket cashier problem, based on the producer-consumer problem. From experiments envolving both parallel and serial designs for the solution, several efficiency measures were taken and analysed for determining the impact of code parallelization on the current problem. Four processors were used in this experiment. Furthermore, a superlinear speedup was reached in the parallel solution using two processors.*

Resumo. *Esse trabalho introduz conceitos da programação paralela, para então aplicá-los a um problema abstrato de atendimento em supermercado, baseado no problema do produtor/consumidor. A partir de experimentos feitos com um algoritmo sequencial e com outro paralelo, diversas medidas foram tomadas e analisadas, avaliando o impacto da paralelização da solução para esse problema do supermercado, especificamente. Foram utilizados 4 processadores nos experimentos, sendo que para a solução com dois processadores obteve-se um speedup superlinear.*

Introdução

A computação paralela tornou-se mais expressiva nos últimos anos pelo fato de não ser possível se extrair maior capacidade de processamento dos chips pelo aumento da frequência de clock, uma vez que essa abordagem aumenta muito o consumo energético e a dissipação de calor. Para resolver esse problema, computadores atuais usam múltiplos núcleos menos potentes ao invés de um só núcleo com maior desempenho.

Apesar de aparentemente aumentar a eficiência para qualquer problema grande que possa ser dividido, a computação paralela introduz alguns problemas desafiadores relacionados ao compartilhamento de memória. No caso de sistemas multicore, por exemplo, tem-se vários processadores compartilhando uma mesma unidade de memória, a qual pode ter regiões acessadas simultaneamente pelas diferentes threads, executadas em concomitância. Essas regiões de memória compartilhada são denominadas **regiões críticas**.

Para fazer a gestão do acesso às regiões críticas, algumas linguagens de programação oferecem a seus usuários recursos como **semáforos** e **mutexes**. Semáforo é uma variável especial protegida, o que significa que não é permitido o acesso simultâneo dela por mais de uma thread concorrente. Uma variável de semáforo tem 2 operações principais:

- **Wait:** Decrementa o valor da variável semáforo, caso seja maior que zero. Caso contrário, a thread que invocou essa operação é colocada para dormir.

- **Post:** Incrementa o valor da variável semáforo. Caso alguma outra thread esteja dormindo, essa é acordada para realizar sua operação **Wait** pendente.

Mutex é um tipo de variável derivado do semáforo. Nesse caso particular, seu valor só pode assumir 0 ou 1. Em resumo, um mutex é um semáforo binário.

Um dos problemas mais clássicos resolvíveis pela computação paralela é o problema do **produtor/consumidor**. Dado um buffer de um determinado tipo de objeto, existem threads que consomem itens desse buffer e outras que os produzem. Os acessos ao buffer são controlados por dois semáforos: um para registrar as posições livres do buffer e outro para as posições ocupadas. As operações de consumo precisam decrementar o semáforo de posições ocupadas e depois incrementar o semáforo de posições livres, enquanto as operações de produção fazem o oposto.

Para medir a eficiência das soluções paralelas, diversas métricas foram estabelecidas ao longo do tempo. A mais utilizada delas é o valor de **speedup**, que mede o quão rápida a solução paralela fica com a adição de processadores para a resolução do problema. Derivada a partir do speedup, a métrica de **eficiência** mede o aproveitamento do speedup por número de processadores utilizados. Outras métricas são realizadas a partir do número de instruções de máquina executadas, tanto na aplicação sequencial quanto na aplicação paralela, ambas para um mesmo problema.

Descrição do Problema

O objetivo desse trabalho é medir o ganho de eficiência na solução computacional de um problema através da programação paralela. O problema abordado nesse trabalho envolve uma abstração de um supermercado, o qual possui clientes sendo atendidos paralelamente em diversos caixas.

Seja um supermercado com N caixas, com uma fila de clientes para cada um dos caixas. Os clientes procuram sempre entrar na menor fila para serem atendidos mais rapidamente. Cada caixa atende clientes de sua própria fila, mas caso essa esteja vazia, ele pode chamar um cliente de outra fila para ser atendido pelo caixa. Se não houver cliente algum para atender, o caixa se bloqueia e espera por novos clientes.

Uma vez que um cliente escolhe uma fila, ele não pode mudar para uma outra fila, exceto quando for chamado para ser atendido em outro caixa. O tamanho das filas dos caixas é infinito.

Modelagem do Problema

A quantidade de clientes em cada fila é representada na solução como um vetor de semáforos, com índices 0 a $N\_FILAS - 1$, em que N_FILAS é o número de caixas e filas do supermercado. O semáforo é incrementado quando um cliente entra na fila, e decrementado quando ele é atendido pelo caixa correspondente à fila em que está.

Para resolução paralela do problema, são criadas N_FILAS threads chamadas **cliente_thread**, responsáveis por inserir novos clientes nas filas, observando a prioridade pelas filas com o menor número de clientes. Cada thread insere um cliente por vez. Também há N_FILAS threads chamadas de **caixa_thread**, as quais representam os caixas que atendem os clientes das filas. Cada caixa atende um cliente por vez.

As threads **cliente_thread** inserem clientes enquanto a variável $CLIENTES\_A\_CHEGAR$ é positiva. Essa variável é decrementada toda vez que um cliente for inserido, até que não existam mais clientes para ser atendidos nos caixas. Como essa variável é compartilhada pelas várias **cliente_thread**, ela é uma região crítica, e também tem uma variável mutex associada a ela.

Os tipos de threads **cliente_thread** e **caixa_thread** operam como no problema do produtor / consumidor: um deles insere (produz) clientes no problema enquanto o outro os atende (consome), até que não existam mais clientes no supermercado. A diferença do problema base é que, ao invés de se ter apenas um buffer de objetos consumidos, tem-se diversos desses buffers, representados pelas filas dos caixas.

Análise dos Resultados

Os testes realizados foram feitos em um sistema Linux, distribuição Ubuntu 16.04, processador Intel(R) Core(TM) i5 CPU M 430 @ 2.27GHz. A execução da aplicação teste foi feita sob os comandos *taskset* e *time*, respectivamente usados para alocar um número específico de CPUs para a tarefa e para medir variáveis de execução durante a solução, como tempo de processamento e uso de memória.

Para a solução paralela do problema, adotou-se o número de 16 threads: 8 **cliente_thread** (produtora de clientes) e 8 **caixa_thread** (consumidora de clientes). O programa se encerra quando 6400 clientes forem atendidos. Para melhor comparação, a solução sequencial do problema utiliza esses mesmos parâmetros, mas com funções sequenciais ao invés de threads.

Foram feitos os seguintes testes: sequencial (com o algoritmo sequencial), paralelo com 2 núcleos, com 3 núcleos e com 4 núcleos. Para cada um desses casos, 20 testes foram executados, tomando-se nota do tempo de execução de cada um deles.

Tabela 1. Tempos de execução (em segundos)

	Sequencial	2 CPU	3 CPU	4 CPU
Teste 01	297,91	129,10	091,46	72,77
Teste 02	297,88	129,10	096,78	72,85
Teste 03	298,29	129,11	097,76	73,76
Teste 04	298,05	129,13	099,49	74,52
Teste 05	297,80	129,10	100,04	76,19
Teste 06	297,85	129,10	100,12	77,75
Teste 07	299,78	129,11	100,27	76,79
Teste 08	297,98	129,12	101,18	84,23
Teste 09	297,79	129,16	100,37	77,58
Teste 10	297,80	129,13	102,35	80,49
Teste 11	298,25	129,12	102,85	81,65
Teste 12	297,92	129,12	105,84	82,47
Teste 13	298,06	130,20	107,85	82,89
Teste 14	299,08	129,33	108,46	84,29
Teste 15	299,02	130,43	109,85	84,34
Teste 16	298,55	132,26	110,05	84,65
Teste 17	299,18	132,32	110,35	84,73
Teste 18	298,44	133,60	110,22	84,70
Teste 19	298,48	133,13	110,53	84,74
Teste 20	298,38	137,30	111,48	84,79
Média	298,32	130,40	103,86	80,31
Desvio	± 0.54	± 2.16	± 5.59	± 4.42

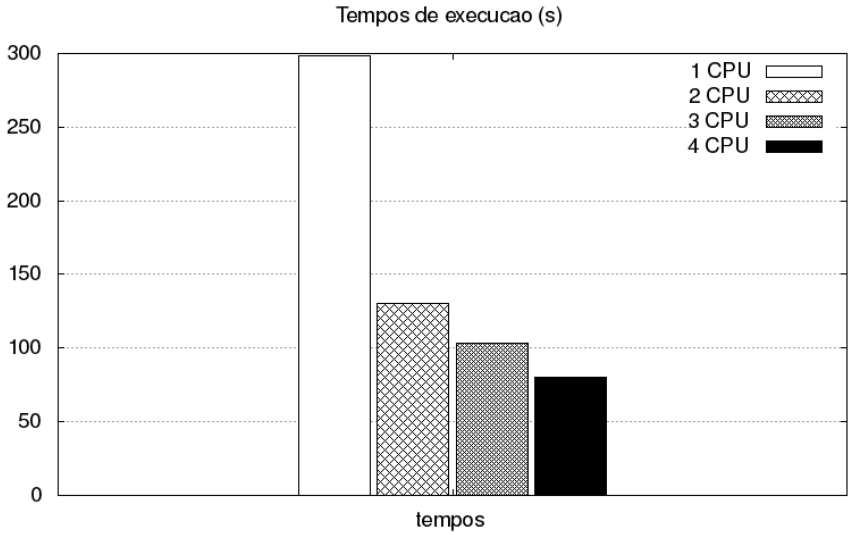

Figura 1. Tempos de Execução

Para realizar todas as métricas de desempenho, também é necessário saber qual o número de instruções de máquina executadas pelo código sequencial, pela parte sequencial do código paralelo, e pela parte paralela do código paralelo. Para isso, utilizou-se a

ferramenta *perf*, disponível no Linux, para contar o número de instruções executadas em cada caso.

Tabela 2. Número de Instruções Executadas

	Número de Instruções
Código Sequencial	322.123.695.502
Total do Código Paralelo	322.125.063.378
Parte Sequencial do Código Paralelo	418.937
Fração Sequencial do Código Paralelo	0,000001301

O *Speedup* mede o quão mais rápido é uma solução paralela em relação à sua correspondente solução sequencial. Pode ser medido por:

$$S(p) = \frac{T(1)}{T(p)}$$

O gráfico da figura 2 mostra, além dos valores de speedup obtidos experimentalmente, quais são os valores teóricos máximos de speedup para cada quantidade de processadores. Para dois processadores, o valor de speedup obtido experimentalmente foi maior que o esperado pelo máximo teórico, caracterizando um **speedup superlinear**.

Tabela 3. Valores de Speedup

	Tempo Médio (s)	Speedup
Sequencial	298,32	1,00
2 CPU	130,40	2,29
3 CPU	103,86	2,87
4 CPU	80,31	3,71

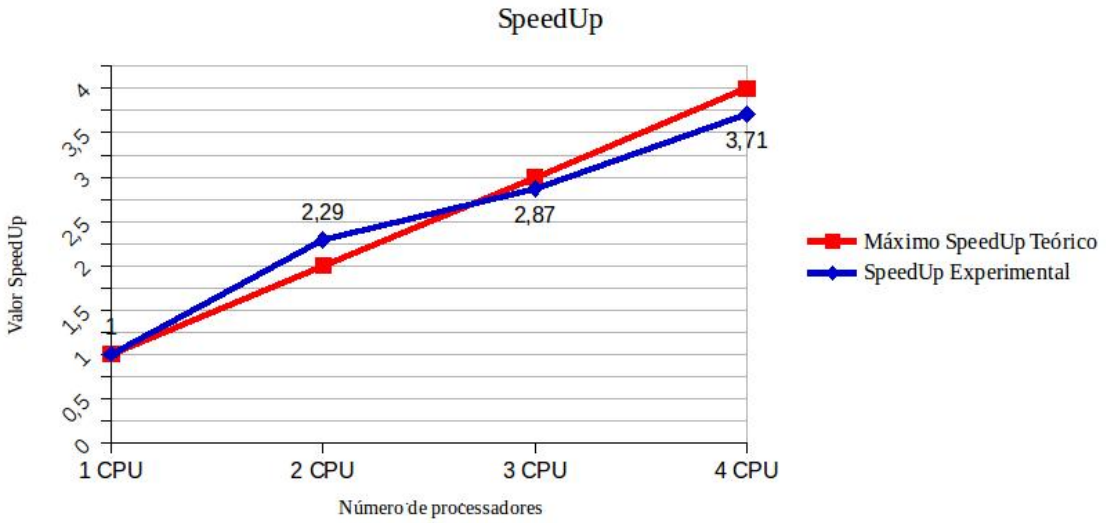

Figura 2. Speedup

A eficiência da paralelização é uma medida calculada por:

$$E(p) = \frac{S(p)}{p}$$

Tabela 4. Eficiência

	Speedup	Eficiência
Sequencial	1,00	1,00
2 CPU	2,29	1,14
3 CPU	2,87	0,96
4 CPU	3,71	0,93

Figura 3. Eficiência

O grau de aumento de computação R(p) mede a proporção entre o número de instruções executadas pela solução paralela, representado por O(p) e o número de instruções executadas pela solução sequencial, representado por O(1). Temos que:

$$R(p) = \frac{O(p)}{O(1)}$$

Tabela 5. Grau de Aumento de Computação

	Valor
Instruções do Sequencial	322.123.695.502
Instruções do Paralelo	322.125.063.378
R(p)	1,000004246

A medida do grau de aproveitamento da capacidade computacional relaciona ambas as medidas $R(p)$ e $E(p)$, com o objetivo de analisar o ganho em capacidade computacional pela paralelização da solução. Ela é dada por: $U(p) = R(p) \times E(p)$.

Tabela 6. Grau de Aproveitamento da Capacidade Computacional

Núcleos	E(p)	R(p)	U(p)
2 CPU	1,14	1,000004246	1,14000484
3 CPU	0,96	1,000004246	1,094404646
4 CPU	0,93	1,000004246	1,017796321

O grau de importância que a solução paralela tem para o problema é definida por:

$Q(p) = \frac{S(p) \times E(p)}{R(p)}$

Tabela 7. Grau de Importância da Solução Paralela

Núcleos	E(p)	R(p)	S(p)	Q(p)
2 CPU	1,14	1,000004246	2,29	2,610588915
3 CPU	0,96	1,000004246	2,87	2,755188301
4 CPU	0,93	1,000004246	3,71	3,450285350

A lei de Amdahl é usada para calcular o máximo speedup teórico de uma solução paralela para dado problema. Considerando f a fração sequencial do código paralelo e p o número de processadores, tem-se que:

$S(p) \leq \frac{1}{f + \frac{1-f}{p}}$

Tabela 8. Limite de Speedup pela Lei de Amdahl

Núcleos	$S(p) \leq$
2 CPU	1,999997398
3 CPU	2,999992194
4 CPU	3,999984388

Com o mesmo propósito da lei de Amdahl, a lei de Gustafson-Barsis considera que o trabalho sequencial do código aumenta com o número de processadores. Considerando f a porção sequencial do código com p processadores, a lei pode ser expressa por:

$S(p) \leq p + f \times (1 - p)$

Tabela 9. Limite de Speedup pela Lei de Gustafson-Barsis

Núcleos	$S(p) \leq$
2 CPU	1,999998699
3 CPU	2,999997398
4 CPU	3,999996097

A métrica de Karp-Flatt permite a obtenção da quantidade de trabalho sequencial (e) da aplicação de forma empírica, a partir do speedup $S(p)$. Essa métrica permite certas inferências sobre o custo de comunicações para a aplicação paralela. Se o valor de e não

variar com o número de processadores, significa que o custo das comunicações não estão prejudicando a eficiência da solução.

$$e = \frac{\frac{1}{S(p)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Tabela 10. Trabalho sequencial por Karp-Flatt

Núcleos	e
2 CPU	$-0,126637555$
3 CPU	$0,022648084$
4 CPU	$0,026055705$

Conclusão

Os experimentos mostraram um speedup visível com o acréscimo de processadores na execução paralela da solução. Os níveis de eficiência, assim como as demais métricas calculadas, sugerem que a paralelização contribui muito para a solução do problema enunciado.

O valor de speedup obtido para a solução paralela com dois processadores foi maior que o previsto, tanto pela lei de Amdahl quanto pela lei de Gustafson-Barsis. O valor se $S(2) = 2,29$ indica um **speedup superlinear** para esse caso específico. Essa ocorrência fica clara no gráfico da figura 2, na qual o limite teórico de speedup é marcado pela linha vermelha. Esse fenômeno pode ter múltiplas razões, entre as quais está uma subdivisão do problema, o que minimiza o número de *cache miss* na execução.

Para a métrica de Karp-Flatt, a quantidade de trabalho sequencial para a solução com dois processadores foi negativa, uma vez que a ocorrência de speedup superlinear foi registrada para esse caso. Para as soluções com 3 e 4 processadores, notou-se uma pequena diferença entre seus respectivos valores da métrica e. Isso sugere que o custo das comunicações não têm tanto impacto na solução paralela quanto o grau de paralelização do problema em si.

Referências

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2012). *Algoritmos: Teoria e Prática*. Elsevier, 3ª Edição.

Hennessy, J. L.; Patterson, D. A.; (2012). *Computer Architecture: A Quantitative Approach*. Elsevier, 5th Edition.