Problema do Supermercado: Implementação sequencial e paralela, utilizando de memória compartilhada

Universidade Estadual de Maringá(UEM) – Maringá – PR – Brazil
Departamento de Informática (DIN)
ra62030, ra63284@uem.br

Carlos Henrique Paisca, Juliano Donini

Dezembro de 2016

1 Introdução

A programação concorrente foi usada inicialmente na construção de sistemas operacionais, atualmente ela é utilizda no desenvolvimento de aplicações em diversas áreas da computação. Mesmo com avanços relacionados a programação concorrente, ainda assim a grande maioria dos programas são escritos de forma sequencial.

Os programas são ditos sequênciais, pois possuem apenas um único fluxo de execução. Assim o programa é executado sequencialmente da primeira à última linha de código, seguindo-se rigorosamente a ordem em que o programa foi escrito.

Já programação concorrente é onde o programa é executado sequencialmente concorrendo pela disponibilidade do(s) processador(es) com os demais programas. O programa é dividido em vários "sub-processos"conhecido como Threads que serão executados paralelamente com o processo "Pai". Essa forma de desenvolvimento é melhor aproveitada quando dispõe de dois ou mais núcleos de processamento.

Para o presente trabalho foi proposto a implementação de dois algoritmos um seguindo a abordagem sequêcial e outro paralela, e em seguida análisar as técnicas de medidas de desempenho para verificarmos a eficiencia da paralelização para o problema proposto.

2 Descrição do Problema

O presente trabalho consiste em implementar dois algoritmos para resolver o problema do proposto (Supermercado). O primeiro algoritmo consiste em implementar o código sequencial e o segundo algoritmo consiste em implementar o código paralelo.

O problema citado consiste em "simular" a fila de atendimento de um supermercado com N caixas, onde cada empregado é responsável por atender as pessoas que chegam em seu caixa. Outra situação que pode ocorrer e o empregado precisa lidar, é quando chega um cliente novo, este escolhe em qual caixa deseja ser atendido (menor número de clientes na fila), feito essa escolha o cliente não pode trocar de fila, e o caixa faz o atendimento. Este processo termina quando todas as filas não tiveram mais clientes para serem atendidos e o supermercado estiver fechado.

Após realizar as implementações, tem-se por objetivo comparar os resultados obtidos sequencialmente e confrontá-los com os resultados obtidos paralelamente. Segundo a literatura, se os dois algoritmos foram codificadas corretamente, espera-se que o resultado do algoritmo paralelo apresente uma eficiência considerável em relação ao algoritmo sequencial.

Para saber se a proposta implementada atende o que a literatura sugere, iremos utilizar 5 métricas de avaliação de resultados (SpeedUp, Eficiência, Redundância, utilização e qualidade) que serão discutidas mais a frente.

3 Modelagem do Problema

Buscando satisfazer as condições explicadas nos tópicos anteriores (utilização de um algoritmo sequencial e um algoritmo paralelo), modelamos o algoritmo da forma descrita na seção 3.1 e 3.2.

3.1 Modelagem Sequencial

Visando obter os paramentros necessarios para realizar uma analise de desempenho coerente, foi desenvolvida a versão sequencial, esta segue uma abordagem semelhante a implemetação paralela, porém sem a criação de Threads, que representam operações concorrente.

3.2 Modelagem Paralela

Para modelarmos o problema proposto utilizamos como base o conceito de produtor consumidor, criamos em um primeiro momento uma thread para coletar informações sobre os clientes que estão na fila do supermercado, uma segunda thread para coletar informações dos caixas, como caixa livre, caixa com N pessoas na fila e caixa aberto, além dessas duas threads, criamos uma estrutura de thread capaz de paralelizar o atendimento dos caixas, ou seja, define-se via código a quantidade de Caixas que serão criados para atender os clientes.

O atendimento se baseia nos N caixas criados, onde cada empregado é responsável por atender as pessoas que cheguem na fila do seu caixa, desta forma, toda vez que um novo cliente deseja entrar na fila de atendimento é feito um bloqueio utilizando a função (ptheread_mutex_lock), este lock tem a finalidade de evitar conflito em uma região crítica, pois a fila é lida toda vez que precisa atender um cliente, dando um lock é possível garantir que o valor lido da fila será correto, depois do lock adiciona o cliente na fila e libera a fila para ser lida (ptheread_mutex_unlock). Com essas duas funções evitamos que uma instrução acesse uma região crítica, fazendo assim o bloqueio exclusivo dos dados.

A fila de clientes é um vetor de tamanho definido inicialmente (#define CLIENTES), quando não existem mais clientes na fila e o supermercado está fechado, nenhum cliente pode entrar na fila ou ser atendido. Já para verificar para qual fila é mandado um cliente que está neste vetor, faz-se o uso da função escolherMenorFila, está verifica qual fila tem a menor quantidade de clientes, caso na primeira verificação (caixa 1) esteja com 0 clientes na fila, a verificação é interrompida e o cliente da fila é enviado para este caixa livre, caso contrário faz-se uma verificação em todos os caixas e envia o cliente para a fila com menor número de clientes na fila.

Dentre as possíveis situações que podem ocorrer dentro de um supermercado, temos a situação em que não existe clientes na fila, quando isto ocorre,

o caixa “dorme”, porém se chegar um novo cliente, o caixa “acorda”, realiza o atendimento, verifica se tem mais alguém para atender, caso não tenha ele dorme novamente.

Por fim criamos uma função (sincronizarCaixa) que tem por responsabilidade verificar as filas e as funções que estão ligadas a fila, está função permite ao algoritmo saber se todos os clientes foram atendidos, se existe caixa vazio, ou se o mercado está aberto, com estas condições o algoritmo decide se o mercado pode ser encerrado.

4 Implementação do Problema

Visando demostrar o que foi desenvolvido ao longo do trabalho, temos abaixo algumas funções utilizadas e a explicação de seu funcionamento.

4.1 Implementação Sequencial

Visando demonstrar o que foi desenvolvido na versão sequencial, temos algumas versões abaixo e sua implementação.

A função escolherMenorFila, verifica dentre as filas criadas, qual fila possui a menor quantidade de cliente, se a fila estiver com 0 clientes, a verificação termina e o cliente é enviado para está fila, se as filas pussuirem clientes, então envia-se o cliente para a menor fila.

```
int escolheMenorFila(){
        int a;
        int menor = 0;
        for(a=0; a<CAIXAS; a++){
                if(filas[a] == 0){
                        return a;
                }
                else if(filas[a] < filas[menor]){
                        menor = a;
                }
        }
        return menor;
}
```

Em um primeirmo momento fizemos um cast, pois o gcc estava dando um warning. Após isto inicializamos a fila, esta recebe o id do caixa criado, no if, verifica se a fila possui clientes e se o mercado ainda está aberto, se as duas condições forem satisfeitas, o caixa termina seu atendimento. Se a fila possuir cliente, a função funcionarioAtendendoCliente() é chamada para o atendente realizar o atendimento aos clientes.

```
int threadFuncionarioCaixa(void *thread_id){
                long idCaixa = (long) thread_id;
                int fila = filas[idCaixa];

                if((fila == 0) && (mercadoAindaAberto==0)){
                        return 1;
                }
                if(fila > 0){
                        if(funcionarioAtendendoCliente(idCaixa)){
                                return 1;
                        }
                }
                return 0;
}
```

4.2 Implementação Paralela

A Função escolherMenorFila, faz a verificação em todos os caixas, caso encontre algum caixa com 0 clientes na fila, o novo cliente é enviado para esta fila, caso contrário é realizada uma comparação entre todos os caixas e o novo cliente é enviado para a fila que tiver a menor quantidade de clientes.

No início da verificação utiliza um lock pois estamos fazendo acesso há uma região crítica, depois do lock é feita uma verificação da quantidade de clientes do caixa, em sequida liberamos o caixa, para que o cliente possa ser inserido na menor fila.

```
int escolheMenorFila(){
        int a;
        int menor = 0;
        for(a=0; a<CAIXAS; a++){
                pthread_mutex_lock(&sincronizarFilas[a]);
                if(filas[a] == 0){
                        pthread_mutex_unlock(&sincronizarFilas[a]);
```

```
                return a;
            }else if(filas[a] < filas[menor]){
                menor = a;
            }
            pthread_mutex_unlock(&sincronizarFilas[a]);
        } return menor;
}
```

Função responsável por controlar o tempo de atendimento de cada caixa para simular uma situação real. Para isso criamos um tempo de atendimento na qual foi implementada uma função rand que pode varia de 4 a 8 segundos, atribuindo esse valor a uma função sleep que faz com que a thread durma. Isso corresponde ao tempo de atendimento de uma caixa a um cliente da fila.

```
int funcionarioAtendendoCliente(long idCaixa){
        long descanso = (rand() %5) + 4; // tempo de atendimento
        sleep(descanso);
        return(sincronizarCaixas(idCaixa, 0, 1));
}
```

No primeiro momento se inicia os caixas. Depois disso é feita a utilização do lock para verificar a quantidade de clientes que estão na fila do caixa, passado pelo id, depois de verificar a quantidade se libera o caixa (unlock). Seguindo o código temos mais um lock, este com a função de verificar todos os caixas, se estão atendendo, se existe cliente na fila, caso a fila esteja vazia e o mercado não esteja mais aberto, o caixa pode encerrar seu atendimento e libera a sincronização. Caso a fila ainda tenha clientes, o caixa faz o atendimento e em seguida encerara o atendimento.

```
void* threadFuncionarioCaixa(void *thread_id){
        printf("Caixa %ld esta atendendo\n", (long) thread_id);
        while(1){
                long idCaixa = (long) thread_id;
                pthread_mutex_lock(&sincronizarFilas[idCaixa]);
                int fila = filas[idCaixa];
                pthread_mutex_unlock(&sincronizarFilas[idCaixa]);
                pthread_mutex_lock(&sincronismoMercado);
                if((fila == 0) && (mercadoAindaAberto == 0)){
                printf("Caixa %ld esta encerrando seu atendimento!! \n
                        pthread_mutex_unlock(&sincronismoMercado);
```

```
                        break;
                }
                else{
                        pthread_mutex_unlock(&sincronismoMercado);
                }
                if(fila > 0){
                        if(funcionarioAtendendoCliente(idCaixa)){
                printf("Caixa %ld esta encerrando seu atendimento!! \n
                                break;
                        }
                }
        }
}
```

5 Medidas de desempenho

5.1 Speedup

Speedup é visto como um aumento de velocidade, ou diminuição, de execução de um determinado processo em n processadores, em relação a execução deste mesmo processo em apenas 1 processador, que podemos oberservar na fórmula abaixo, onde T(1) é o tempo de execução com 1 processador e T(n) é o tempo de execução com p processadores.

$$SpeedUp = \frac{T(1)}{T(p)} \tag{1}$$

5.2 Eficiência

A medida de eficiência trata da relação entre o número de processadores utilizados e o speedup obtido, como podemos observar na fórmula abaixo, onde n é o números de processos concorrentes, onde T(1) é o tempo de execução com 1 processador, p é a quantidade de processadores e T(n) é o tempo de execução com p processadores.

$$e = \frac{SpeedUp}{n} = \frac{T(1)}{p * T(p)} \tag{2}$$

5.3 Redundância

A medida de redundância trata da relação entre o número de processadores operações realizadas pela execução sequencial e paralela, onde O(p) é o

número de operações realizadas com p processadores e O(1) é o número de operações realizadas com 1 processador.

$$R(p) = \frac{O(p)}{O(1)} \tag{3}$$

5.4 Utilização

A medida de utilização trata da relação entre a capacidade computacional utilizada durante a computação e a capacidade disponível, onde R(p) é redundância com p processadores e E(p) é eficiência com p processadores.

$$U(p) = R(p) * E(p) \tag{4}$$

5.5 Qualidade

A medida de qualidade trata da importância de utilizar programação paralela, onde S(p) é o SpeedUp com p processadores, E(p) é eficiência com p processadores e R(p) é redundância com p processadores.

$$Q(p) = \frac{S(p) * E(p)}{R(p)} \tag{5}$$

6 Análise dos Resultados

6.1 Ambiente experimental

Ambos os algoritmos foram implementados na linguagem de programação C, no ambiente de desenvolvimento sublime Text 3. As características do computador são as seguintes: 4GB memória RAM, processador i5 com 4 núcleos físicos e o sistema operacional utilizado foi o Ubuntu 14.04.5 LTS (Trusty Tahr).

6.2 Resultados Experimentais

Após a criação da versão do código sequencial e paralela, utilizou-se como parâmetro de comparação a quantidade de clientes atendidos pelo Supermercado e o tempo para atender todos os clientes e fechar o mercado, os seguintes resultados foram obtidos:

Métricas	Thread = 1				
	Entrada 1	Entrada 2	Entrada 3	Entrada 4	Entrada 5
Nº Clientes	100	200	300	400	500
Tempo(s)	775	1.623	2.444	3.214	3.932
Inst. Assembly	438	438	438	438	438

Tabela 1: Algoritmo Sequencial.

Métricas	Thread = 2				
	Entrada 1	Entrada 2	Entrada 3	Entrada 4	Entrada 5
Nº Clientes	100	200	300	400	500
Tempo(s)	301	605	900	1.204	1.504
Inst. Assembly	880	880	880	880	880
SpeedUp	2,57	2,68	2,71	2,67	2,61
Eficiência	1,29	1,34	1,36	1,34	1,31
Redundância	2,01	2,01	2,01	2,01	2,01
Utilização	2,59	2,69	2,73	2,69	2,63
Qualidade	1,65	1,79	1,83	1,78	1,70

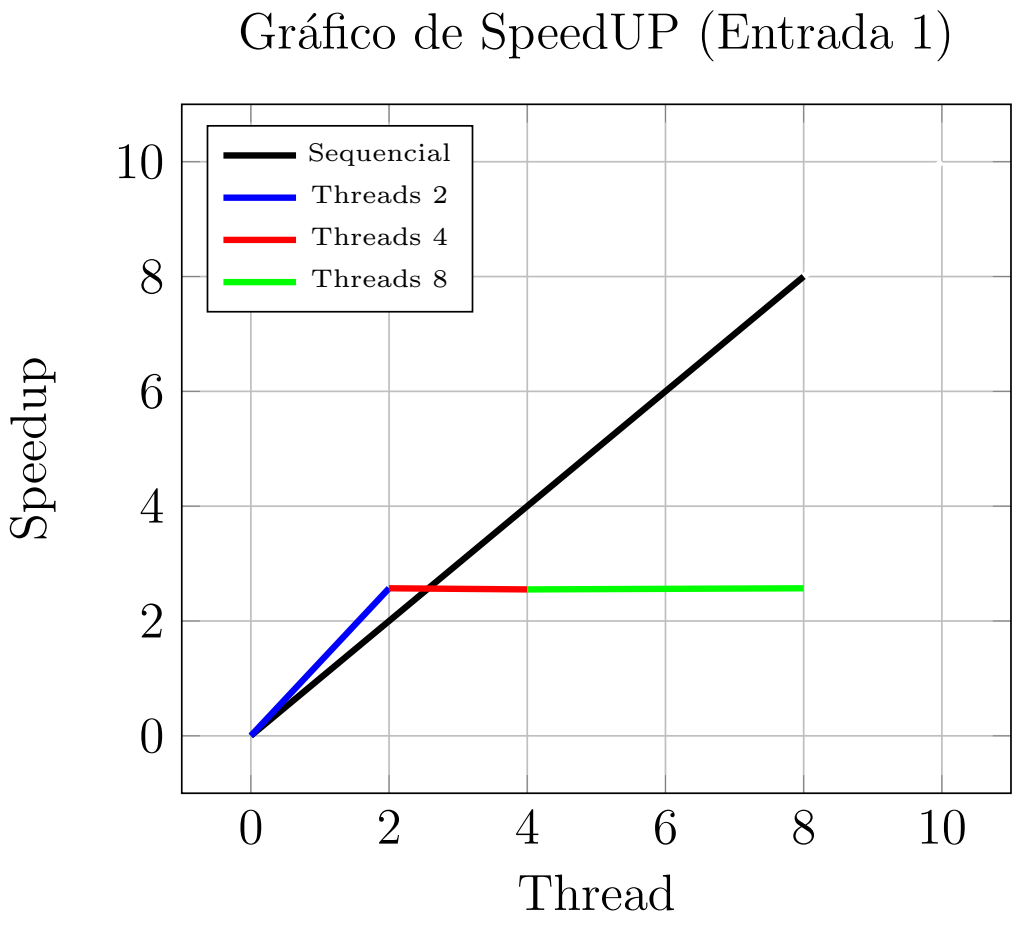
Tabela 2: Algoritmo Paralelo.

Métricas	Thread = 4				
	Entrada 1	Entrada 2	Entrada 3	Entrada 4	Entrada 5
Nº Clientes	100	200	300	400	500
Tempo(s)	304	602	903	1.201	1.502
Inst. Assembly	880	880	880	880	880
SpeedUp	2,55	2,70	2,70	2,68	2,62
Eficiência	0,64	0,68	0,68	0,67	0,66
Redundância	2,01	2,01	2,01	2,01	2,01
Utilização	1,29	1,37	1,37	1,35	1,33
Qualidade	0,81	0,91	0,91	0,89	0,86

Tabela 3: Algoritmo Paralelo.

Métricas	Thread = 8				
	Entrada 1	Entrada 2	Entrada 3	Entrada 4	Entrada 5
Nº Clientes	100	200	300	400	500
Tempo(s)	301	610	909	1.215	1.504
Inst. Assembly	880	880	880	880	880
SpeedUp	2,57	2,67	2,69	2,65	2.61
Eficiência	0,32	0,33	0,34	0,33	0,33
Redundância	2,01	2,01	2,01	2,01	2,01
Utilização	0,64	0,66	0,68	0,66	0,66
Qualidade	0,41	0,44	0,45	0,43	0,43

Tabela 4: Algoritmo Paralelo.



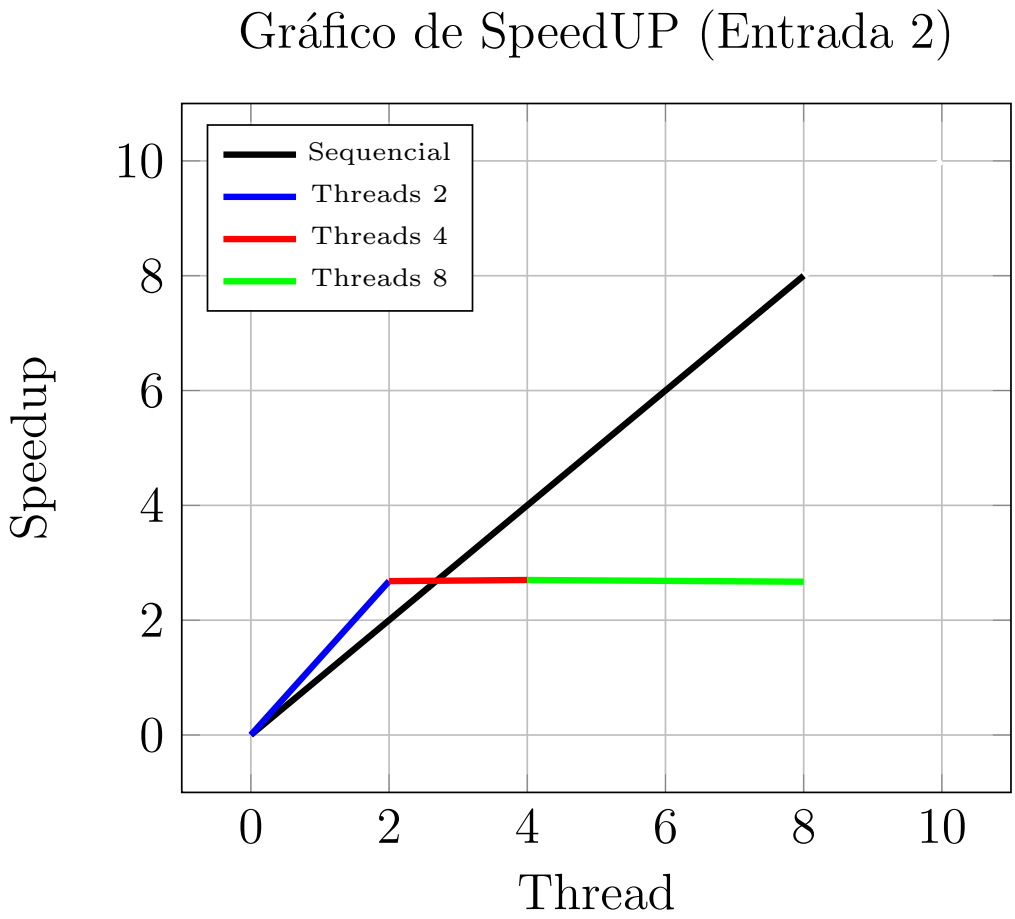


Gráfico de SpeedUP (Entrada 2)
Sequencial
Threads 2
Threads 4
Threads 8
Speedup
Thread
0
2
4
6
8
10

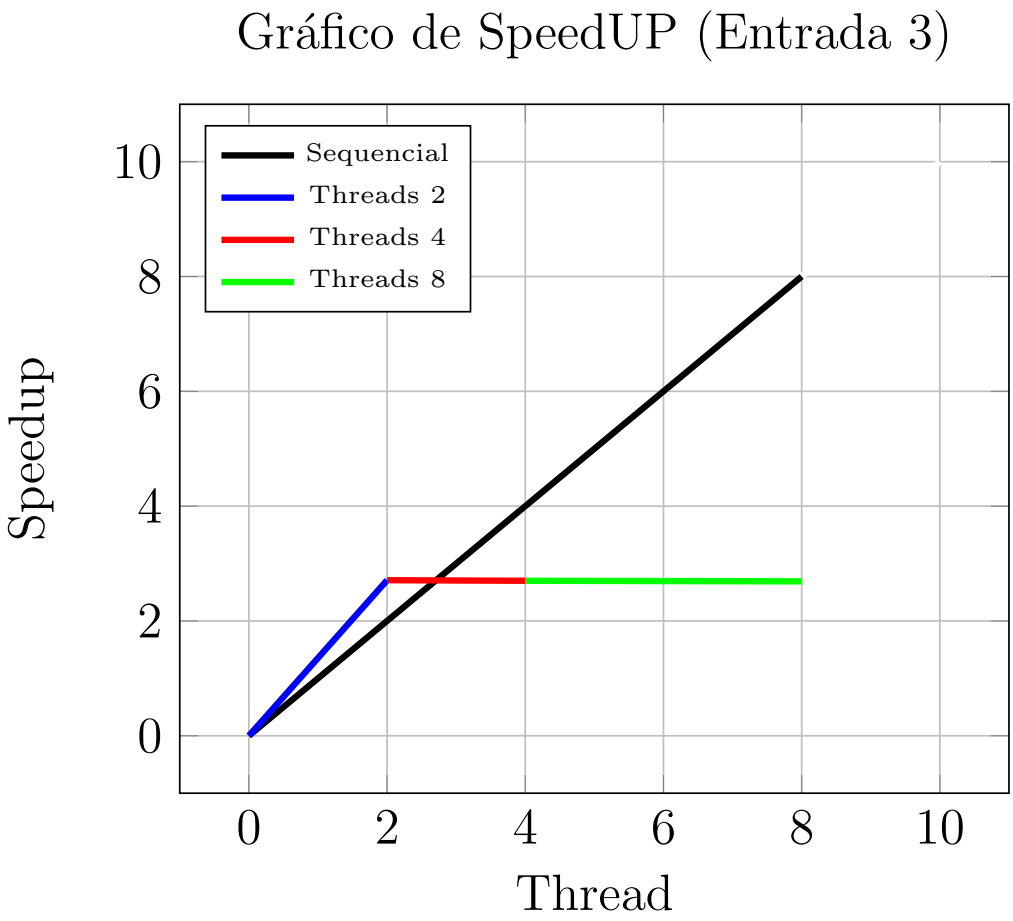


Gráfico de SpeedUP (Entrada 3)
Sequencial
Threads 2
Threads 4
Threads 8
Speedup
Thread
0
2
4
6
8
10

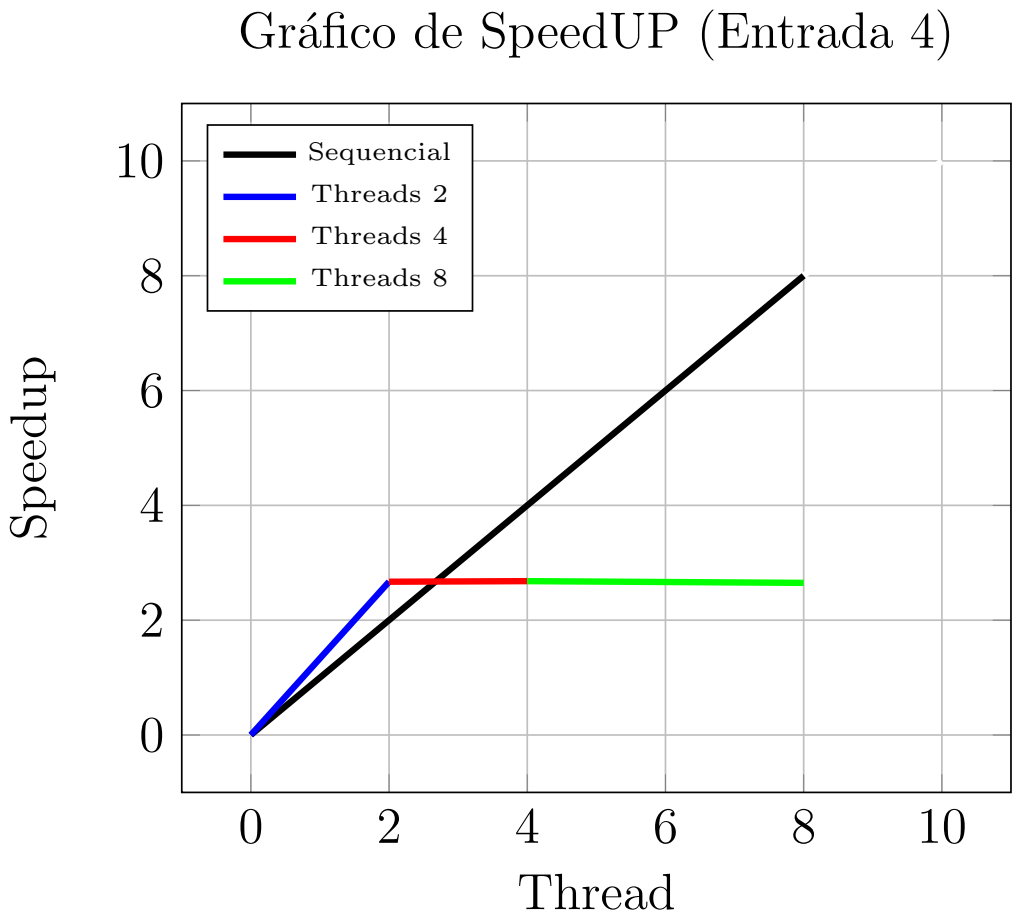


Gráfico de SpeedUP (Entrada 4)
Sequencial
Threads 2
Threads 4
Threads 8
Speedup
Thread
0
2
4
6
8
10

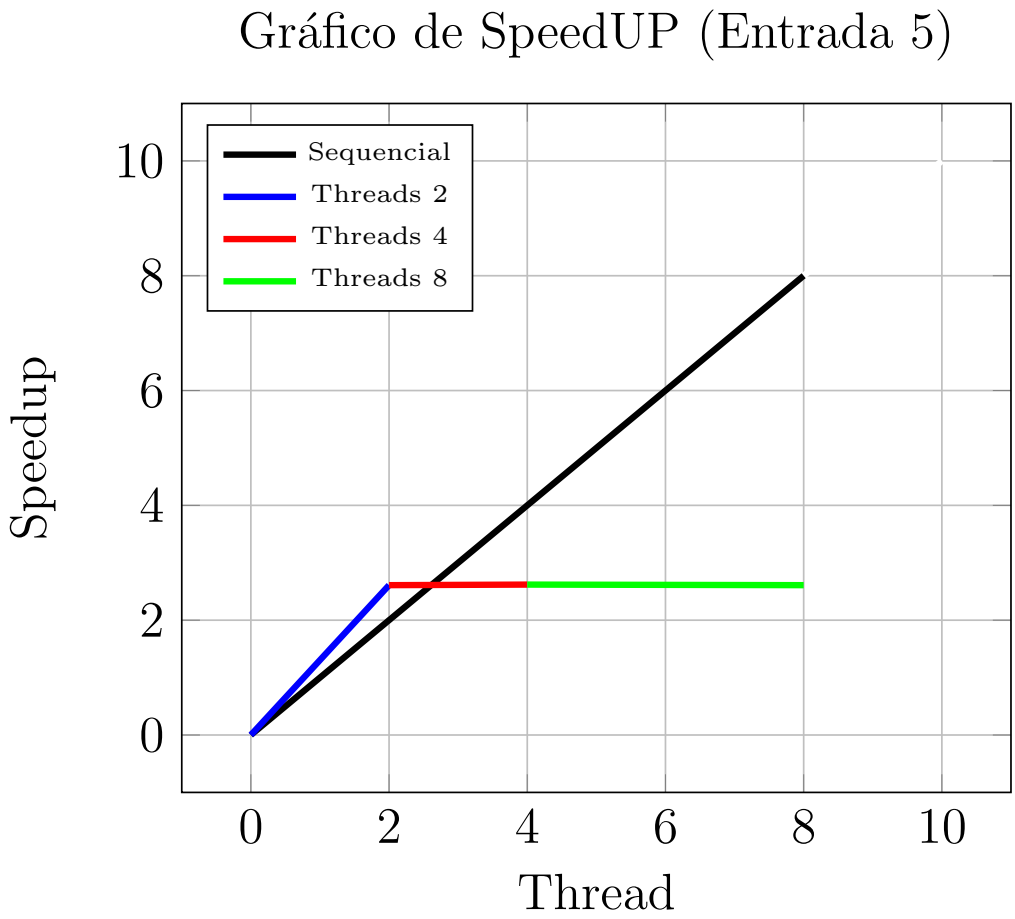


Gráfico de SpeedUP (Entrada 5)
Sequencial
Threads 2
Threads 4
Threads 8
Speedup
Thread
0
2
4
6
8
10

7 Conclusões

Mediante os resultados obtidos experimentalmente, compreendemos que a implementação do problema do supermercado atendeu a ideia principal do trabalho, que foi implementar um algoritmo sequencial e um algoritmo paralelo, e que o algoritmo paralelo apresentasse um menor tempo para solucionar o problema proposto. Isto ocorreu em virtude da natureza do problema e suas características, foi possível modelar de forma eficiente uma solução que resolvesse o problema corretamente.

No decorrer da execução dos experimentos, percebemos que a proposta de paralelização do problema se mostrou satisfatória, visto que a curva de speedup para 2 threads, representa o máximo do processador utilizado com 2 núcleos reais e se comportou com uma curva acima do linear, segundo a literatura quando o speedup é superior a quantidade de processadores utilizados, tem-se um speedup Superlinear, que representa um custo de comunicação praticamente inexistente, subdivisão correta do problema e aumento da capacidade de memória.

Quando aumentamos a quantidade de threads, não tivemos uma diminuição do tempo de execução, este fato pode ser explicado pelo jeito que nossa abordagem foi desenvolvida. Em um primeiro momento são criadas todas as Threads (caixas) que o usuário deseja, são inicializadas, recebem clientes que estão na fila esperando para serem atendidos, porém conforme a fila de clientes a serem atendidos aumenta, o tempo para enviar um cliente para o caixa diminui, isto permite que os primeiros caixas sempre estejam vazios, o que torna as demais Threads inutilizáveis durante o processo. Isto explica porque o aumento de Threads (caixas) não altera o tempo.

Além de se analisar a métrica Speedup para comprovar o porquê 2 Threads tiveram desempenho superior, uma análise pode ser feita sobre a métrica qualidade, esta envolve fatores como Speedup, Eficiência e Redundância, fatores que foram calculados para analisar os resultados detalhadamente. Ao observarmos a tabela 2, percebesse que a Qualidade se aproxima muito da quantidade de processadores físicos utilizados (2), a métrica garante que quando mais próximo o resultado do valor do número de processadores, mais recursos do computador foram utilizados para resolver o problema.

Assim sendo, o problema do supermercado nos possibilitou modelar o problema onde pudéssemos modelar um problema real, envolver os conceitos

aprendidos na disciplina, confrontar os resultados obtidos com o que a literatura sugere e fazer o uso de threads, para paralelizar problemas reais.

8 Referências Bibliográficas

• S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 2003.

• TOSCANI, S., OLIVEIRA, R., CARISSIMI, A. Sistemas Operacionais e Programação Concorrente. Série didática do II-UFRGS, 2003.

• Pereira, Rodolfo Miranda. Programação Concorrente - Conceitos de Processos e Threads. Disponível em: `http://moodlep.uem.br/pluginfile.php/78732/mod_resource/content/0/Aula\%204\%20-%20Conceitos%20de%20Processos%20e%20Threads.pdf`. Acesso em: outubro de 2016.

• Pereira, Rodolfo Miranda. Programação Concorrente - Implementação de Threads. Disponível em: `http://moodlep.uem.br/pluginfile.php/79952/mod_resource/content/0/Aula%205%20-%20Implementa%C3%A7%C3%A3o%20de%20Threads%20-%20Parte%201.pdf`. Acesso em: outubro de 2016.

• Pereira, Rodolfo Miranda. Programação Concorrente - Computadores Paralelos. Disponível em: `http://moodlep.uem.br/pluginfile.php/77341/mod_resource/content/0/Aula%202%20-%20Computadores%20Paralelos.pdf`. Acesso em: Novembro de 2016.

• Pereira, Rodolfo Miranda. Programação Concorrente - Desempenho. Disponível em: `http://moodlep.uem.br/pluginfile.php/78064/mod_resource/content/0/Aula%203%20-%20Desempenho.pdf`. Acesso em: outubro de 2016.

• Vieira, Rafael Siqueira. Uso de Threads no C. Disponível em: `https://www.vivaolinux.com.br/script/Uso-de-threads-no-C`. Acesso em: Outubro de 2016

• Andrade, Kleber. Programando com Threads em C. Disponível em: `https://kleberandrade.files.wordpress.com/2015/02/aula_pthreads.pdf`. Acesso em: Novembro de 2016.