UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UFRJ

Daniel Lopes de Sousa
Lucas Clemente Miranda Braga

Problema de Quadratura para Integração Numérica:
Primeiro Trabalho
Disciplina de Computação Concorrente

Professora: Silvana Rossetto

Sumário

1. Introdução

Este relatório tem como objetivo documentar o desenvolvimento do primeiro trabalho feito para disciplina de Computação Concorrente no período de 2019-2 do curso de Ciência da Computação da Universidade Federal do Rio de Janeiro.

1.1 Objetivos

O principal objetivo do presente trabalho é o de aprendizado na prática de conceitos teóricos aprendido em sala. Aprendizado este que proporciona a oportunidade de crescimento como alunos e desenvolvedores.

Além disso, o projeto tem como objetivo implementar uma solução de aproximação de integrais de funções definidas em determinado intervalo através do método de integração numérica retangular. Este método consiste em aproximar a área sob a curva de uma dada função *f(x)* dividindo-a em retângulos cada vez menores, até que este valor aproximado seja satisfatório de acordo com um fator de erro máximo pré-definido.

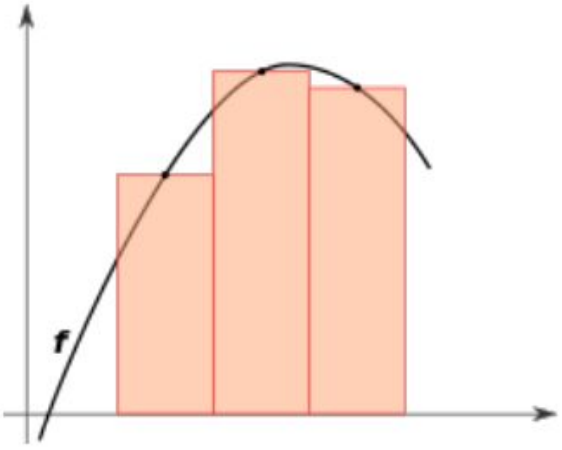

Figura 1. Aproximação por retângulos utilizando ponto médio.

Por fim, neste trabalho foram implementadas duas soluções distintas para o problema apresentado, uma sequencial e uma concorrente. E como objetivo final, temos a comparação de ganho de desempenho entre as duas soluções. Qual das duas soluções é mais eficiente? Qual é o

ganho em termos de eficiência em relação a solução contrária? Essas perguntas deverão ser respondidas através de testes de tempo de execução das soluções implementadas.

1.2 Tecnologias utilizadas

Para a implementação do projeto foi utilizada apenas a linguagem de programação C, juntamente com as bibliotecas relacionadas ao uso de funções matemáticas ("*math.h*") e para uso e controle de threads ("*pthread*"), ambas bibliotecas padrão da linguagem.

Além disso, foi utilizada uma função passada pela professora em sala de aula, definida no arquivo *timer.h*, com a única finalidade de calcular os tempos gastos na execução das soluções sequencial e concorrente para comparação.

2. Organização do projeto

Nesta seção estão especificadas as soluções para o problema (estruturas e funções utilizadas e a lógica do programa), primeiro a sequencial seguida pela solução concorrente.

Porém, antes de adentrar nas soluções, vale ressaltar alguns elementos e funções comuns a ambas as soluções. O primeiro deles são as funções que foram utilizadas como teste para o método de aproximação do valor da integral num intervalo. São elas:

- $f(x) = 1 + x$
- $f(x) = \sqrt{1 - x^2}, \ -1 < x < 1$
- $f(x) = \sqrt{1 + x^4}$
- $f(x) = sen(x^2)$
- $f(x) = cos(e^{-x})$
- $f(x) = cos(e^{-x}) * x$
- $f(x) = cos(e^{-x}) * (0.005 * x^3 + 1)$

Assim, nas soluções temos a definição destas funções como funções globais a serem chamadas nas funções de cálculo da aproximação da integral.

```
double func1(double x){ return 1 + x; }
double func2(double x){ return sqrt(1 - pow(x, 2)); }
double func3(double x){ return sqrt(1 + pow(x, 4)); }
double func4(double x){ return sin(pow(x,2)); }
double func5(double x){ return cos(exp(-x)); }
double func6(double x){ return cos(exp(-x)) * x;}
double func7(double x){ return cos(exp(-x)) * ((0.005 * pow(x, 3)) + 1); }
```

Figura 2. Definição das funções de teste para a integração.

Além disso, como especificado no trabalho, devíamos criar uma solução adaptativa para o problema da quadratura para integração numérica. Isto é, não podendo ser definido um número N inicial de repetições para a divisão da área sob a curva da função em retângulos cada vez menores. Assim, foi decidido que a melhor maneira de resolver o problema é com uma função recursiva, que calcula o ponto médio "*m*" do intervalo *[a,b]* passado e divide a área em três retângulos, utilizando o valor da função *f(x)* definida para a execução. Numa segunda repetição a função é chamada duas outras vezes com os intervalos *[a,m]* e *[m,b]* e assim por diante.

Esta função recursiva será a principal função de cálculo da aproximação da integral tanto na solução sequencial quanto na concorrente. Ambas diferem apenas nas estruturas utilizadas para armazenar os intervalos a serem calculados e na forma como essas funções recursivas são chamadas.

2.1 Solução Sequencial

Para a solução sequencial, a função principal recebe como parâmetros da linha de comando o intervalo de integração e o valor de erro máximo. A leitura é feita como *<nome do programa> <primeiro valor do intervalo> <segundo valor do intervalo> <erro máximo>*, onde o primeiro valor do intervalo é armazenado em uma variável "*a*", o segundo valor em uma variável "*b*" e o valor de erro máximo é armazenado em uma variável *"maxError"*.

Abaixo segue um exemplo de entrada do usuário para o programa e o trecho de código em que é feita a leitura dessa entrada e armazenados os valores nas variáveis correspondentes.

```
./main-seq -10 30 0.00005
```

Figura 3. Exemplo de entrada do programa sequencial.

```
//le e valida os parametros de entrada
if(argc < 4) {
  fprintf(stderr, "Digite: %s <intervalo de integracao> <erro maximo>.\n", argv[0]);
  exit(EXIT_FAILURE);
}

a = atoi(argv[1]);
b = atoi(argv[2]);
maxError = atof(argv[3]);
```

Figura 4. Definição e preenchimento das variáveis de intervalo e erro.

Com isso, temos então a chamada para a função que irá calcular a integral das funções já citadas acima. Foi escolhido calcular a aproximação de todas as funções num só programa, com o objetivo de facilitar os testes e a implementação.

Assim, temos sete chamadas para a função recursiva, uma para cada função de teste. O tempo de execução de cada uma é calculado e impresso juntamente com o resultado da integração. Temos abaixo um trecho de código da chamada da função recursiva para uma das funções juntamente com o cálculo de tempo.

```
//executa todas as funcoes calculando o tempo de execucao de cada uma
//------------------------------------------------------------------------------
GET_TIME(inicio);

printf("Resultado para a função 1 = %f\n", retangulosInicial(func1, a, b, maxError));

GET_TIME(fim);

//calcula e exibe o tempo gasto com a multiplicacao
delta = fim - inicio;
printf("Tempo de calculo da integral: %.8lf\n\n", delta);
//------------------------------------------------------------------------------
```

Figura 5. Chamada à função principal de cálculo e exibição do tempo de execução.

2.1.1 Estruturas e funções geradas

Na formulação do programa sequencial, definimos apenas duas funções para a integração. A primeira apenas para inicializar o problema e a segunda para continuar o processo recursivo de solução.

A função inicial *"retangulosInicial"* executa a primeira divisão da área sob o gráfico da

função em retângulos na tentativa de aproximação do valor da integral. Ela é também a responsável por chamar a função recursiva que continua a subdivisão em retângulos nos intervalos cada vez menores da direita e da esquerda do ponto médio inicial.

Para isso, recebe como entrada a função cuja integral será calculada, os limites *"a"* e *"b"* do intervalo e o valor do erro.

```
double retangulosInicial(double (*func) (double), double limiteA, double limiteB, double erro) {
```

Figura 6. Parâmetros da função que inicia o cálculo da integral.

No escopo da função são definidos e calculados os valores do ponto médio *"m"*, a distância entre os pontos *"a"* e *"b"* (*b* - *a*), assim como os valores da função passada em *"a", "b"* e *"m"* e por fim a primeira aproximação da integral da curva, definida por

$$\int_a^b f(x)\,dx \simeq (b-a)f(\tfrac{a+b}{2})$$

```
double valorMedio;
double dist;
double funcEmA, funcEmB, funcEmMedio;
double integral;

valorMedio = (limiteA + limiteB)/2;
dist = limiteB - limiteA;

funcEmA = func(limiteA);
funcEmB = func(limiteB);
funcEmMedio = func(valorMedio);

integral = dist * funcEmMedio;
```

Figura 7. Escopo da função *retangulosInicial()*.

No retorno desta função temos uma chamada à função *"retangulosRecursiva"*, que recebe como parâmetros de entrada a função cuja integral será calculada, os limites *"a"* e *"b"* do intervalo, o valor do erro, a primeira aproximação da integral calculada na função inicial, além dos valores da função nos pontos *"a", "b"* e *"m"*.

```
return retangulosRecursiva(func, limiteA, limiteB, erro, integral, funcEmA, funcEmB, funcEmMedio);
```

Figura 8. Retorno da função *retangulosInicial()*.

Já a função *"retangulosRecursiva"* realiza a divisão da área sob o gráfico em retângulos cada vez menores até que a aproximação do cálculo da área debaixo da função (o valor da integral) seja satisfatório de acordo com o erro definido na entrada.

Os parâmetros de entrada desta função são similares aos da função "*retangulosInicial()"*, contendo a função cuja integral será calculada, os limites *"a"* e *"b"* do intervalo e o valor do erro. E, como parâmetros adicionais temos os valores da função em *"a", "b"* e no ponto médio, além do valor da integral calculada na primeira divisão em retângulos. Valor esse que será aproximado cada vez mais do valor real.

```
double retangulosRecursiva(double (*func)( double), double limiteA, double limiteB,
                           double erro, double integral, double funcEmA, double funcEmB, double funcEmMedio) {
```

Figura 9. Parâmetros da função *retangulosRecursiva()*.

O escopo da função, da mesma forma, é similar ao escopo da função *"retangulosInicial"*, onde são definidos e calculados o valor médio e a distância entre os limites *"a"* e *"b"*. Porém, nessa função calculamos outros dois valores médios, da direita e da esquerda do valor médio inicial. É calculado então o valor da função nesses dois novos pontos médios e a área sobre cada metade da função. A integral é então definida como a soma das integrais das duas metades da área sob a função.

```
double valorMedio;
double dist;
double valorMedioEsquerda, valorMedioDireita;
double funcaoEmEsquerda, funcaoEmDireita;
double integralEsquerda, integralDireita;
double novaIntegral;

valorMedio = (limiteA + limiteB) / 2;
dist = limiteB - limiteA;

valorMedioEsquerda = (limiteA + valorMedio) / 2;
valorMedioDireita = (valorMedio + limiteB) / 2;

funcaoEmEsquerda = func(valorMedioEsquerda);
funcaoEmDireita = func(valorMedioDireita);

integralEsquerda = ( dist / 2 ) * funcaoEmEsquerda;
integralDireita = ( dist / 2 ) * funcaoEmDireita;

novaIntegral = integralEsquerda + integralDireita;
```

Figura 10. Parte do escopo da função *retangulosRecursiva()*.

Isso significa que a área entre os limites *"a"* e *"b"* recebidos pela função é dividida em dois e o valor da integral passa a ser a soma dessas duas áreas. O que a função inicial *"retangulosInicial"* faz é definir um retângulo apenas, calcular a área deste retângulo (um valor aproximado da integral da função), então dividir a função em dois pedaços que serão passados para duas instâncias de *"retangulosRecursiva"*. A função recursiva, por sua vez divide a área sob a função em dois retângulos e a integral é definida como a soma da área destes dois retângulos, criando uma aproximação um pouco mais exata do que a calculada inicialmente.

Assim, no primeiro passo temos um retângulo. Já no segundo temos 4 retângulos, e assim por diante.

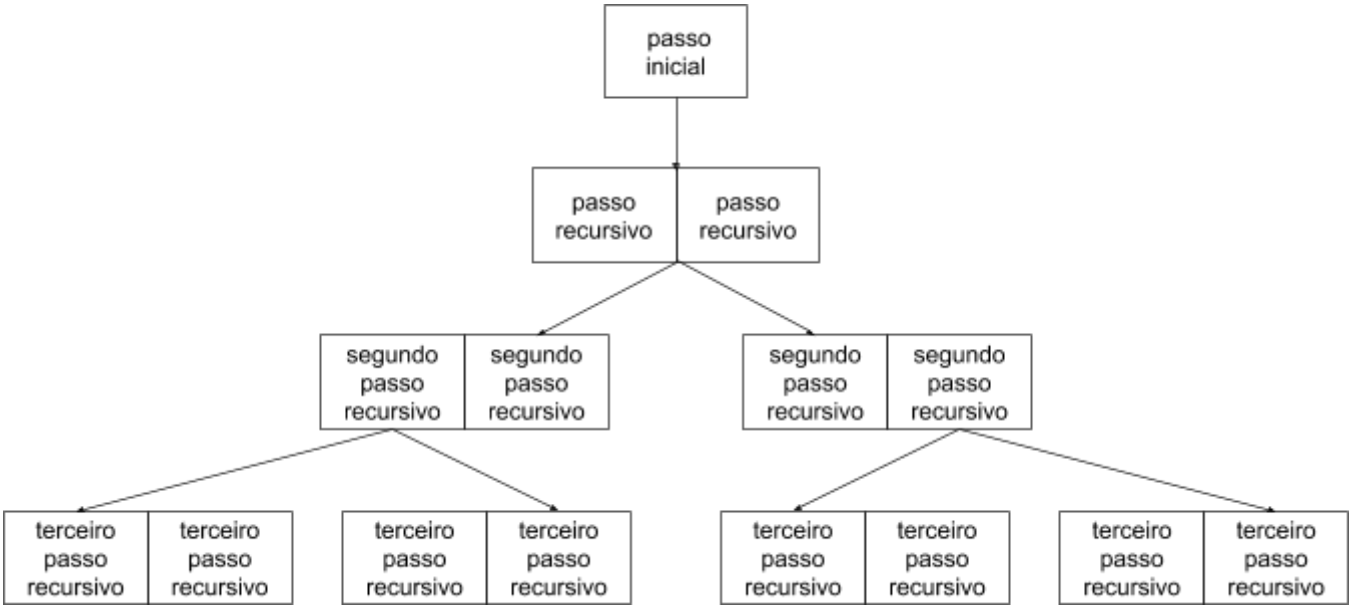

Figura 11. Número de retângulos em cada passo de execução.

Continuando no escopo da função *"retangulosRecursiva"*, temos por final a checagem da condição de parada e o retorno da função.

A condição de parada da divisão em retângulos é simples e diz respeito ao erro que pretendemos alcançar. Isso quer dizer que quando a diferença entre a soma da nova integral dos dois retângulos que acabaram de ser divididos e a integral do retângulo único antes de ser dividido for menor que nosso erro, a função retorna o valor da nova integral. Ou seja, considerando um retângulo A, cuja integral é a aproximação da área sob o gráfico definida como a área de A. Esse retângulo é dividido em dois outros retângulos iguais B e C, cuja integral é a aproximação da área sob o gráfico da função definida como a soma X das áreas de B e C. Se o valor de X subtraído da área de A for menor do que o erro definido, significa que X é uma aproximação satisfatória da área sob a função no intervalo definido pelos limites dos retângulos e pode ser retornada.

2.2 Solução Concorrente

Para a solução concorrente, a função principal recebe como parâmetros da linha de comando o intervalo de integração, o valor de erro máximo e o número de threads do programa. A leitura é feita como *<nome do programa> <primeiro valor do intervalo> <segundo valor do intervalo> <erro máximo> <número de threads>*. Os valores são armazenados nas mesmas variáveis

utilizadas na versão sequencial da solução, diferenciando-se apenas a variável de threads que será armazenada em *"nthreads"*.

2.2.1 Estruturas e funções geradas

Sendo a estrutura geral do programa concorrente basicamente a mesma do programa sequencial, iremos focar nas mudanças feitas nas funções que calculam a aproximação da integral.

Mas antes, é importante definir a estratégia utilizada para construir a concorrência para o problema. Foi utilizada uma estrutura de fila, armazenada num arquivo *"fila.h"*, que implementa em cada nós os valores de um limite superior e um inferior, o valor do erro, a aproximação atual da integral, além do valor da função nos pontos *"a"*, *"b"* e no ponto médio.

```
struct Values{
  double (*func) (double);
  double limiteA;
  double limiteB;
  double erro;
  double integral;
  double funcEmA;
  double funcEmB;
  double funcEmMedio;
  struct Values * next;
};
```

Figura 12. Estrutura da fila implementada.

Assim, temos uma estrutura que determina os valores atuais para cada um desses itens, independente do intervalo em questão. E todas as threads podem acessar essa informação, a fim de executar a função de aproximação. Com isso, o problema se transforma numa situação de produtor/consumidor, em que o buffer é preenchido com intervalos e seus valores (ou retângulos, como os chamamos) e as threads buscam esses valores na fila para processá-los e gerar novos valores a serem colocados na fila.

Dito isso, vejamos as variáveis globais definidas:

```
int threadsExecutando = 0;
int threadsProntas = 0;
int nthreads;
int * iteracoes;

pthread_mutex_t mutex;
pthread_cond_t cond;

fila * nbuffer;

double * resultado;
```

Figura 13. Variáveis globais.

A função que inicia a integração permanece idêntica à sua versão sequencial, com exceção do retorno. Seu retorno é substituído pela criação e primeiras inserções na fila, iniciando todo o processo. Essa função é chamada para a função cuja integral será calculada antes mesmo de serem criadas as threads.

Já quando as threads são criadas, a função de cada thread passa a ser a função recursiva *"retangulosRecursiva"*. Nesta função o método de cálculo da integral também permanece o mesmo já descrito nas seções anteriores, portanto serão ressaltadas apenas as porções referentes à concorrência da função.

Ao iniciarmos a função, enquanto as threads estiverem rodando e o buffer estiver preenchido com pelo menos um elemento, podemos entrar na porção inicial da função. Essa porção verifica mais uma vez se o buffer não tem nenhum elemento e, se sim, a thread para de executar e espera o buffer ser preenchido. Quando isso acontece, a variável que guarda o número de threads em execução é incrementada e a função pega os valores do buffer para processá-los.

```
pthread_mutex_lock(&mutex);

//printf("threads executando = %d e tamanho = %lld\n", threadsExecutando, nbuffer->size);
if(nbuffer->size == 0){ threadsExecutando--; pthread_mutex_unlock(&mutex); break;}

while(nbuffer->size == 0) {   }
threadsExecutando++;
valores = defila(nbuffer);

pthread_mutex_unlock(&mutex);
```

Figura 14. Parte do escopo da função **retangulosRecursiva()*.

Após o processamento, se a condição de parada ainda não foi atingida, a thread enfila os valores obtidos e executa um *broadcast* para outras threads que estejam aguardando sinal para execução. Se a condição de parada foi atingida, ela armazena o valor obtido num array com os resultados de cada thread. Este array será varrido no final da execução de todas as threads e seus valores somados.

Em seguida, a thread entra em modo de espera caso existam outras threads ativas e o buffer esteja vazio de elementos.

```
pthread_mutex_lock(&mutex);
threadsExecutando--;
if (threadsExecutando != 0){
  while (nbuffer->size == 0 && threadsExecutando > 0) { pthread_cond_wait(&cond, &mutex); }
}
pthread_mutex_unlock(&mutex);
```

Figura 15. Parte do escopo da função **retangulosRecursiva()*.

3. Testes realizados

Como testes para os programas sequencial e concorrente, foram aplicados diversos intervalos com diversos valores de erro, a fim de verificar a veracidade dos resultados.

3.1 Testes da solução sequencial

Temos nesta seção alguns exemplos de testes aplicados na solução sequencial do programa, incluindo as informações de entrada e a saída do algoritmo.

```
./main-seq -1 1 0.005
```

Figura 16. Exemplo de entrada do programa sequencial.

```
Resultado para a função 1 = 2.000000
Tempo de calculo da integral: 0.00018978

Resultado para a função 2 = 1.589450
Tempo de calculo da integral: 0.00005317

Resultado para a função 3 = 2.065656
Tempo de calculo da integral: 0.00004911

Resultado para a função 4 = 0.616037
Tempo de calculo da integral: 0.00003290

Resultado para a função 5 = 0.676882
Tempo de calculo da integral: 0.00001907

Resultado para a função 6 = 0.608128
Tempo de calculo da integral: 0.00001502

Resultado para a função 7 = 0.678588
Tempo de calculo da integral: 0.00002003
```

Figura 17. *Exemplo de saída do programa sequencial.*

Segue outro exemplo:

```
./main-seq 2 10 0.001
```

Figura 18. Exemplo de entrada do programa sequencial.

```
Resultado para a função 1 = 56.000000
Tempo de calculo da integral: 0.00015402

Funcao 2 fora do intervalo!

Resultado para a função 3 = 330.863800
Tempo de calculo da integral: 0.00013399

Resultado para a função 4 = -0.221049
Tempo de calculo da integral: 0.00028014

Resultado para a função 5 = 7.999286
Tempo de calculo da integral: 0.00001001

Resultado para a função 6 = 47.997145
Tempo de calculo da integral: 0.00001407

Resultado para a função 7 = 20.472901
Tempo de calculo da integral: 0.00004292
```

Figura 19. Exemplo de saída do programa sequencial.

Com um intervalo maior, temos o exemplo a seguir:

```
./main-seq 0 100 0.001
```

Figura 20. Exemplo de entrada do programa sequencial.

```
Resultado para a função 1 = 5100.000000
Tempo de calculo da integral: 0.00028300

Funcao 2 fora do intervalo!

Resultado para a função 3 = 333334.560511
Tempo de calculo da integral: 0.00377202

Resultado para a função 4 = 0.302760
Tempo de calculo da integral: 0.04443908

Resultado para a função 5 = 100.000000
Tempo de calculo da integral: 0.00002789

Resultado para a função 6 = 5000.000000
Tempo de calculo da integral: 0.00001001

Resultado para a função 7 = 125099.757251
Tempo de calculo da integral: 0.00172782
```

Figura 21. Exemplo de saída do programa sequencial.

3.2 Testes da solução concorrente

Temos nesta seção alguns exemplos de testes aplicados na solução concorrente do programa, incluindo as informações de entrada e a saída do algoritmo.

```
./main-conc -1 1 0.005 2
```

Figura 22. Exemplo de entrada do programa concorrente.

```
função 1 + x :

numero de iterações de cada thread no cálculo desta integral :
thread 0 : 1
thread 1 : 1
Valor da integral: 2.00000000
Tempo = 0.000639

função sqrt(1 - pow(x, 2)) :

numero de iterações de cada thread no cálculo desta integral :
thread 0 : 7
thread 1 : 1
Valor da integral: 1.58945001
Tempo = 0.000218

função sqrt(1 + pow(x, 4)) :

numero de iterações de cada thread no cálculo desta integral :
thread 0 : 1
thread 1 : 1
Funcao fora do intervalo -1 < x < 1função sin(pow(x,2)) :

numero de iterações de cada thread no cálculo desta integral :
thread 0 : 7
thread 1 : 1
Valor da integral: 0.61603745
Tempo = 0.000147

função cos(exp(-x)) :

numero de iterações de cada thread no cálculo desta integral :
thread 0 : 5
thread 1 : 1
Valor da integral: 0.67688190
Tempo = 0.000095

função cos(exp(-x)) * x :

numero de iterações de cada thread no cálculo desta integral :
thread 0 : 5
thread 1 : 1
Valor da integral: 0.60812840
Tempo = 0.000085

função cos(exp(-x)) * ((0.005 * pow(x, 3)) + 1) :

numero de iterações de cada thread no cálculo desta integral :
thread 0 : 5
thread 1 : 1
Valor da integral: 0.67858750
Tempo = 0.000099
```

Figura 23. Exemplo de saída do programa concorrente.

4. Avaliação de desempenho

Observando os testes da seção acima, podemos perceber que a solução concorrente não foi capaz de ultrapassar o tempo da solução sequencial, atingindo um tempo maior em quase todos os casos. Isso aconteceu porque o grupo encontrou problemas durante a concepção da solução concorrente, não conseguindo obter um balanceamento de carga entre as threads.

Apesar disso, podemos perceber que mesmo sem balanceamento, a solução concorrente permaneceu próxima do tempo da sequencial.

5. Conclusão

Como projeto, o trabalho proposto mostrou-se desafiador, o que salienta duas questões importantes com respeito à disciplina em questão e ao desenvolvimento de aplicações utilizando métodos concorrentes. A primeira delas é que as ferramentas e métodos necessários à construção de um programa concorrente podem, às vezes, parecer simples, mas na verdade são fruto de muito conhecimento agregado e estudo realizado na área. Muitas soluções foram implementadas, uma a partir da outra, para que pudéssemos usufruir dos métodos como o fazemos hoje. Em segundo lugar, um software concorrente demanda muita atenção do desenvolvedor. Um programa compilado pode, muitas das vezes, ser um programa errado. E para corrigir os erros que inevitavelmente irão ocorrer, se faz necessário conhecimento sobre o assunto, com o presente trabalho mostrando-se como uma ferramenta poderosa no aprofundamento deste conhecimento.

5.1 Problemas e dificuldades encontrados

Durante o desenvolvimento do presente projeto, foram encontradas diversas dificuldades e problemas que resultaram em saídas inesperadas no programa final. Seguem as principais delas.

Não conseguimos implementar uma solução concorrente com balanceamento de carga. O exercício proposto neste trabalho se mostrou difícil de ser traduzido em uma solução sequencial para os membros do grupo, o que ocasionou o problema. Até a data de entrega não foi possível implementar uma solução balanceada. Este desbalanceamento de carga entre as threads também foi

a causa de outro problema: o tempo de execução da solução concorrente foi maior do que o da solução sequencial.

Isso também gera, em alguns casos de execução um problema de starvation, em que uma thread sempre acessa os recursos e a outra acaba não os acessando durante todo o tempo de execução.

E, além disso, quando executada com quatro threads, a solução pode, em alguns casos não rodar completamente, parando em algum ponto por tempo indeterminado. Até o momento da entrega deste trabalho, o grupo não sabe o que pode ter ocasionado este problema, que não ocorre quando o programa é executado com uma ou duas threads.

5.2 Alcance dos objetivos

Apesar de todos os problemas já citados, o projeto e desenvolvimento deste trabalho foi de grande valia para o grupo, que irá melhor desenvolver um algoritmo concorrente depois dos erros encontrados. Portanto, podemos dizer que o objetivo de aprendizado foi conquistado.

Quanto à construção de uma solução para o problema, o algoritmo desenvolvido resolve o problema e mostra os resultados adequadamente, o que pode ser considerado como alcançar o objetivo de produzir uma saída correta. Mas quanto à resolver o problema de forma concorrente, podemos dizer que, infelizmente, não foi alcançado o objetivo, visto que a solução concorrente não reflete um modo ideal de solucionar o problema concorrentemente.