Integração Numérica com Quadratura Adaptativa

Aline Freire de Rezende, Gilberto Lopes Inácio Filho

29 de outubro de 2019

Sumário

1 Introdução

O trabalho consiste na implementação de um algoritmo concorrente que calcule a integral numérica de uma função usando a Regra do Ponto Médio, com a estratégia de quadratura adpatativa.

A regra se baseia no cálculo da área do retângulo delimitado pelo intervalo $[a, b]$, com altura $f(x)$ calculada no ponto médio do intervalo. Ao se dividir o problema em um numero suficientemente grande de subproblemas, obtém-se uma boa aproximação da integral.

A estratégia de quadratura adaptativa é usada quando, ao invés de se fixar um número de divisões, os seguintes passos são efetuados:

1. Calcula-se a área original, com base no intervalo $[a, b]$ e altura dada por $f(m)$, sendo m o ponto médio.
2. Divide-se o intervalo original em dois, e calcula-se a área nestes dois subintervalos $[a, m]$ e $[m, b]$, com altura em seus respectivos pontos médios.
3. Compara-se a área original com a soma das áreas dos subproblemas. Se a diferença entre elas for suficientemente pequena, considera-se a área maior uma boa aproximação. Caso contrário, o processo é repetido recursivamente nos subintervalos.

2 Estrutura do código

O código consiste nas aplicações de teste:

- Uma para cada função, executando os algoritmos concorrentes e o sequencial, e imprimindo os resultados calculados e os tempos obtidos.
- Uma aplicação geral, que executa os algoritmos em todas as funções, imprime os resultados, os tempos gastos e o ganho dos algoritmos concorrentes.

Além disso o projeto possui arquivos cabeçalho referentes a:

- Cada um dos algoritmos de solução
- A estrutura de pilha.
- Um utilitário para o cálculo de áreas.
- Um utilitário para a medição do tempo.
- Um mapeamento das funções matemáticas de entrada em código C.

3 Algoritmo Sequencial

O algoritmo implementado na versão sequencial consiste num método recursivo simples de quadratura. A função recebe como entrada os limites $[l, r]$ do intervalo visado, o erro máximo de referência, e a função $f(x)$ que se deseja integrar. Na chamada inicial, o intervalo passado pra função é o intervalo inicial $[a, b]$.

- A área do **retângulo maior** q é calculada tendo como base a distância do intervalo $[l, r]$, e a altura sendo definida como $f(m)$ onde m é o ponto médio do intervalo.
- A área do **retângulo esquerdo** lq tem como referência a base definida pelo subintervalo $[l, m]$, e a altura $f(m_e)$ onde m_e é o ponto médio do subintervalo.
- De modo análogo, a área do **retângulo direito** rq é obtida considerando-se o subintervalo $[m, r]$ e a altura $f(m_d)$, com m_d sendo o ponto médio deste subintervalo.
- Calculadas as áreas, encontra-se a diferença e entre a área do retângulo maior e a soma das áreas dos retângulos menores, $q - (lq + rq)$.

Se a diferença e for menor do que o erro máximo definido, considera-se a área do retângulo maior uma aproximação boa o suficiente para este intervalo, e a função retorna o valor q. Caso contrário, a função retorna a soma das suas chamadas recursivas, uma no subintervalo $[l, m]$ e a outra no subintervalo $[m, r]$.

Ao final de todas as chamadas recursivas, a chamada original retornará a soma de todas as subaproximações consideradas satisfatórias, que representa por sua vez a aproximação da integral definida da função $f(x)$ no intervalo $[a, b]$.

4 Algoritmo Concorrente

Na implementação da solução concorrente, várias estratégias foram consideradas com o objetivo de garantir o balanceamento de carga e o máximo desempenho possível do código. Apesar disso, não houve êxito em implementar de maneira satisfatória e correta uma solução que contemplasse os dois objetivos.

Por isso, foi decidido ilustrar as duas possibilidades que obtiveram o maior grau de sucesso, com outras considerações sobre alternativas sendo feitas ao final do relatório.

Para os algoritmos concorrentes num geral, foi definido o conceito de uma *tarefa*, que representa um intervalo $[l, r]$ cuja área com altura $f(m)$ no ponto médio será avaliada.

4.1 Pilha

Para a execução desta solução, foi necessário o uso de uma pilha de tarefas. Além disso, para melhorar o desempenho o conceito de tarefa foi estendido, de modo que a área maior do intervalo, previamente calculada, é armazenada na tarefa para ser usada na próxima execução.

O código foi implementado usando uma função envelope, que inicializa as variáveis globais de uso comum pelas threads com os valores iniciais apropriados, empilha a tarefa inicial representada pelo intervalo $[a, b]$ e sua **área maior**, dispara as threads, e após os seus términos soma seus resultados parciais ao resultado total.

A lógica de execução das threads se baseia num laço de repetição indefinido. Sua parada é determinada quando não houverem mais tarefas na pilha e todas as threads estiverem inativas. O fluxo pode ser entendido da seguinte maneira:

- A thread avalia se é necessário retirar uma tarefa da pilha. Caso ela tenha divido uma tarefa anteriormente, uma subtarefa terá sido inserida na pilha e outra terá sido retida pela thread, de modo que uma retirada não é necessária.
  - Caso seja necessário, a thread verifica se a pilha está vazia. Se estiver e todas as threads estiverem inativas, a thread sai do laço principal de repetição e armazena seu resultado parcial num vetor global. Se ainda houver threads ativas, a thread se bloqueia até que haja uma tarefa na pilha. Quando ela voltar à execução, retirará uma tarefa.
- A thread calcula as áreas dos subintervalos referentes às metades do intervalo original e compara a soma destas com a área maior anteriormente calculada.
  - Se a diferença for maior do que o erro máximo definido, a thread divide o intervalo da tarefa em duas metades e cria duas novas tarefas com estes intervalos, cada um tendo como área maior a área já calculada para estes pela thread. A thread empilha uma das tarefas e retém a outra como tarefa vigente, para minimizar os acessos à memória. Após isso ela retorna ao começo do laço de repetição.
  - Se a diferença for pequena o suficiente, a thread soma a área maior ao seu acumulador local e retorna ao início do laço de repetição.

Ao término de todas as threads a função envelope soma os resultados parcias ao resultado total, e retorna este valor.

4.1.1 A estrutura

```
// Estrutura de tarefa representando os intervalos
typedef struct tarefa {
  double l;
  double r;
  double area_maior
} tarefa_t;

// Estrutura de pilha com coleção implementada através de vetor dinâmico
typedef struct pilha {
  tarefa_t *colecao;
  int indice;
  int tam;
  int cap;
} pilha_t;

// Função de inicialização da estrutra de pilha e de sua coleção
void p_init(pilha_t **);
// Função chamada ao se tentar inserir numa pilha cheia
// realoca a coleção com o dobro de capacidade
void p_cheia(pilha_t *);
// Retorna 1 se a pilha estiver vazia, 0 caso contrário
int p_vazia(pilha_t *);
// Insere uma tarefa no topo da pilha
p_insere(pilha_t *, tarefa_t);
// Retira uma tarefa do topo pilha
tarefa_t p_retira(pilha_t *);
// Libera a memória da coleção e da estrutura da pilha
void p_destroi(pilha_t *);
```

Figura 1: Cabeçalho da estrutura auxiliar de pilha

4.2 Divisão inicial de tarefas

Este algoritmo é significativamente mais simples e, embora não haja como garantir o balanceamento de carga, se mostrou a solução com melhor desem-

penho na maioria dos casos.

A função envelope divide o intervalo inicial $[a, b]$ em n subintervalos, onde n é o número de threads. Cada thread receberá uma dessas divisões, e executará o algoritmo recursivo sequencial com este intervalo como ponto de partida. O resultado final é guardado no vetor global, e a função envelope irá somar os resultados parciais ao resultado total ao término de cada thread. Este valor é finalmente retornado pela função.

5 Conclusão

Ao longo do projeto, percebeu-se uma grande dificuldade em implementar um algoritmo concorrente que demonstrasse um desempenho superior ao sequencial. As soluções encontradas eram frequentemente muito complexas ou relativamente ineficientes se comparadas com o algoritmo sequencial.

Apesar disso, o algoritmo utilizando a divisão inicial de tarefas se revelou capaz de apresentar um grande ganho de desempenho, que se mostrou ainda mais consistente ao se fixar erros máximos cada vez menores, ou intervalos de cálculo particularmente complexos.

5.1 Ganho

Efetuando-se os testes com intervalos $[-0.9, 0.9]$, $[-10, 10]$ e $[-15, 15]$, com erro máximo fixado em 10^{-10}, os seguintes ganhos foram obtidos em relação ao algoritmo sequencial:

5.1.1 Intervalo $[-0.9, 0.9]$

	f_a	f_b	f_c	f_d	f_e	f_f	f_g
Pilhas	0.0057	0.9935	1.0147	1.2571	1.0859	1.0996	1.7114
Divisão de Tarefas	0.0015	2.1805	2.4577	0.7960	1.9024	0.4080	2.8796

5.1.2 Intervalo $[-10, 10]$

	f_a	f_b	f_c	f_d	f_e	f_f	f_g
Pilhas	0.0066	NaN	0.8304	0.7522	0.6165	0.6215	0.9053
Divisão de Tarefas	0.0032	NaN	4.5189	2.0445	0.9615	0.9807	1.0312

5.1.3 Intervalo $[-15, 15]$

	f_a	f_b	f_c	f_d	f_e	f_f	f_g
Pilhas	0.0075	NaN	0.7992	0.7174	0.5672	0.4924	0.6787
Divisão de Tarefas	0.0037	NaN	2.4278	1.7205	1.0164	1.0134	1.0043

5.2 Estratégias não exploradas

Dentre as outras estratégias que não nos levaram a conclusões diferentes das obtidas até aqui, podemos citar o uso de estruturas diferentes da pilha de vetor dinâmico, como pilhas encadeadas (que possuem um desempenho muito inferior) e pilhas de vetor estático (menos flexíveis e com quase o mesmo desempenho), e filas.

Além disso também houve uma tentativa de usar uma estratégia de quadratura adaptativa "global", com a diferença entre as áreas sendo calculada em função da aproximação da área no intervalo original inteiro, a cada nova divisão dos subintervalos, que também não apresentou um grande ganho no desempenho.

5.3 Considerações finais

Considerando os métodos encontrados para a solução do problema, a técnica mais eficiente se mostrou o uso da divisão inicial de tarefas entre as threads, com o processamento recursivo ocorrendo em seus fluxos de execução.

Embora não tenha sido possível garantir o balanceamento de carga, deve-se levar em consideração que a principal motivação para se procurar o balanceamento é maximizar o desempenho, efeito que foi obtido com a divisão inicial de tarefas.

Além disso existem outras técnicas não exploradas neste trabalho por dificuldades ao se implementá-las, que podem ou não ter um desempenho ainda superior ao encontrado, fazendo um uso mais eficiente da memória e de técnicas de concorrência fora da alçada de conhecimento atual dos participantes.

Apesar disso, a solução obtida já é evidente para indicar que o uso de técnicas de programação concorrente é vantajoso ao ser combinado com métodos numéricos de cálculo, uma vez que estes métodos com frequência se baseiam em numerosas iterações e outras técnicas que apresentam potencial de ganho de eficiência com o uso de múltiplos núcleos de processamento.