TOP RED COMP/SIST DISTRIB II
Quadratura Adaptativa em MPI
Marcelo Paulon, Rayanne Souza
22 de Outubro de 2018

1 Introdução

Este documento apresenta a implementação da técnica quadratura adaptativa paralelizada com MPI. Três versões foram implementadas. A primeira consistiu em paralelizar o cálculo com um número de intervalos igual ao número de processadores usados. Na segunda e na terceira o programa usa um bolsa de tarefas para armazenar os subintervalos a serem calculados. A segunda versão corresponde à primeira variante do método com o uso de uma bolsa. Nessa versão os subintervalos são gerados no início do programa. A terceira versão corresponde à segunda variante do método da bolsa de tarefas. Nesta versão, os trabalhadores colocam tarefas na bolsa quando um determinado subintervalo não atende a tolerância usada no cálculo da área, enviando um dos novos intervalos para a bolsa e trabalhando no outro.

Nas três versões calculamos a área representada pela integral abaixo.

$$\int_0^1 \frac{10\sinh(12)\log(10+x^2)\cos(\sqrt{(1+x^3)^{3/2}})\arctan(\sqrt{2+x^2})}{(1+x^2)^2(2+x^2)\sqrt{\log(2+x)}}dx = 272471 \qquad (1)$$

2 Solução da quadratura adaptativa com o número de intervalos igual ao número de processadores

No código 1, os intervalos de integração são passados como parâmetro. Na linha 25, w define a largura entre os n_cores intervalos. Para cada um dos n_cores processos, a área do trapézio formada pelo seu subintervalo é calculada e passada como parâmetro para a função responsável pelo cálculo da área total de um intervalo (linhas 31-33). Através da funçao MPI_Send (linha 37), os (n_cores - 1) processos criados na paralelização enviam a subárea calculada ao processo mestre. Por sua vez, o processo mestre (p_id = 0), aguarda a recepção de cada uma das áreas calculadas pelos (n_cores -1) processos e soma essas subáreas a fim de obter a área total da curva. Note que o processo mestre cálcula a área do primeiro subintervalo. Assim, na linha 34 a área total é inicializada com a área do primeiro subintervalo.

```
1 #include <stdio.h>
2 #include <math.h>
3 #include "mpi.h"
4 #include <stdlib.h>
5
6 #define  TOL 1e-16
7
8
9 int n_cores;
10 double function(double x);
11 double compute_trap_area(double l, double r);
```

```c
double curve_subarea(double a, double b, double area);

int main(int argc, char *argv[]){
    int p_id;
    double l, r, w;
    double start_t, end_t, total_t;

    l =  atoi(argv[1]);
    r =  atoi(argv[2]);

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &p_id);
    MPI_Comm_size(MPI_COMM_WORLD, &n_cores);

    start_t = MPI_Wtime();
    w = (r - l)/n_cores;

    double a, b, trap_area, local_area;

    a = l + p_id*w;
    b = l + (p_id + 1)*w;
    trap_area = compute_trap_area(a, b);

    local_area = curve_subarea(a, b, trap_area);
    double total_area = local_area;

    if(p_id != 0){
      MPI_Send(&local_area, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  }
    else {
        for (int i = 1; i < n_cores; i++) {
          MPI_Recv(&local_area, 1, MPI_DOUBLE, i, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_area += local_area;
        }
    }
    if(p_id == 0){
        end_t = MPI_Wtime();
        printf("The area under the curve is \%lf \n", total_area);
        total_t = (double)(end_t - start_t);
        printf("Total time taken by CPU: \%.16f\n", total_t);
    }
    MPI_Finalize();

    return 0;
}

double function(double x){
    double num = 10*sinh(2+10)*log(10+x*x)*cos(sqrt(pow(sqrt(1+x*x*x),3)))*
    atan(sqrt(2 + x*x));
    double den = pow((1 + x*x)*sqrt(2+x*x), 2)*sin(sqrt(2))*sqrt(log(2+x));

    return num/den;
}

double compute_trap_area(double l, double r){
    return (function(l) + function(r))*(r - l)*0.5;
}
```

```
69
70
71 double curve_subarea(double a, double b, double area){
72     double m, l_area, r_area, error;
73
74     m = (a + b)*0.5;
75     l_area = compute_trap_area(a, m);
76     r_area = compute_trap_area(m, b);
77     error = area - (l_area + r_area);
78
79     if(fabs(error) <= TOL){
80         return l_area + r_area;
81     }
82     else {
83         return curve_subarea(a, m, l_area) + curve_subarea(m, b, r_area);
84     }
85 }
```

Listing 1: Quadratura adaptativa com o número de intervalos igual ao número de processadores

A Tabela 1 mostra que a paralelização com 8 processadores reduziu em aproximadamente 83.76% o tempo de execução do cálculo da integral. Além disso, observa-se que o tempo de execução reduz com o aumento do número de processadores.

N processadores	Execução 1	Execução 2	Execução 3	Valor Médio
1	35.00179 s	33.96819 s	35.80122 s	34.92373 s
2	18.48059 s	20.86468 s	20.53822 s	19.96116 s
4	11.27724 s	10.89652 s	10.86314 s	11.01230 s
8	5.38719 s	5.35194 s	6.27248 s	5.67053 s

Tabela 1: Tempo de execução do programa com 1, 2, 4 e 8 processadores

3 Solução da quadratura adaptativa com bolsa de tarefas

Para as duas versões apresentadas neste item, a bolsa de tarefas foi implementada usando uma pilha. É necessário o uso de no mínimo 2 processadores pois 1 processador é reservado para operar como mestre e é preciso ao menos 1 trabalhador. No protocolo de comunicação entre mestre-trabalhadores definimos 3 tags:

EXECUTE_TASK Tag enviada pelo mestre para comunicar uma tarefa a um trabalhador.

NO_MORE_TASKS Tag enviada pelo mestre para notificar os trabalhadores que não há mais tarefas a serem feitas.

WORKER_AVAILABLE Tag enviada pelos trabalhadores para indicar o término de uma tarefa e consequentemente sua disponibilidade para novas tarefas.

3.1 Variante 1: subintervalos gerados no início do programa

Além dos limitedo cálculo da integral, o número de tarefas a serem colocadas na bolsa também é passado como parâmetro. Entre as linhas 29-33 insere-se os subintervalos na pilha. No início da execução do mestre, ele aguarda que algum trabalhador envie uma mensagem indicando sua disponibilidade. Enquanto que no início da execução dos trabalhadores, cada um envia uma mensagem ao mestre indicando sua disponibilidade e 0 como valor de área calculada (linha 84). Essa primeira mensagem serve de inicialização visto que nenhuma tarefa foi enviada aos trabalhadores anteriormente. Posto que a comunicação entre mestre e trabalhador foi iniciada, o mestre fica em loop aguardando a recepção de uma área calculada e, por consequência, de um trabalhador disponível, soma essa porção de área à área total, retira um elemento da bolsa de tarefas e o envia para esse trabalhador disponível (EXECUTE_TASK). O mestre sai do loop quando a bolsa de tarefas não possuir mais elementos. Ao término de sua execução o mestre envia a tag NO_MORE_TASKS a todos os trabalhadores para indicar o término das tarefas (linha 71-73). Note que enquanto a tag NO_MORE_TASKS não é recebida pelos trabalhadores, sua execução se mantém computando a área, enviando-a ao mestre e aguardando a recepção de uma nova tarefa (linha 88-99).

```
1  int main(int argc, char *argv[]){
2      int p_id;
3      double l, r, w, trap_area;
4      double start_t, end_t, total_t;
5
6      double total_area = 0;
7
8      MPI_Init(&argc, &argv);
9      MPI_Comm_rank(MPI_COMM_WORLD, &p_id);
10     MPI_Comm_size(MPI_COMM_WORLD, &n_cores);
11
12     start_t = MPI_Wtime();
13
14     if(n_cores < 2) {
15         printf("A minimum of 2 cores is required for this task to work (1
       master and at least one worker)");
16         exit(-1);
17     }
18
19     l = atoi(argv[1]);
20     r = atoi(argv[2]);
21     n_task = atoi(argv[3]);
22     w = (r - l)/(n_task);
23
24     start_t = MPI_Wtime();
25
26     stack_data *stack = stack_create();
27
28     for(int i = 0; i < n_task; i++) {
29         double a = l + i * w;
30         double b = l + (i + 1) * w;
31         stack_push(stack, a, b);
32     }
33
34     double *local_area = (double *) malloc(sizeof(double));
35     if(local_area == NULL) {
36         printf("Unable to create local_area");
```

```
37          exit(-1);
38      }
39
40      if(p_id == 0) {
41          MPI_Status mstatus;
42
43          double k = 0;
44          while(!stack_is_empty(stack)) {
45              MPI_Recv(local_area, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
    WORKER_AVAILABLE,
46                       MPI_COMM_WORLD, &mstatus);
47
48              total_area += *local_area;
49
50              stack_node *node = stack_pop(stack);
51
52              if(node == NULL) {
53                  printf("Error - node is null");
54                  exit(-1);
55              }
56
57              MPI_Send(node, 2, MPI_DOUBLE, mstatus.MPI_SOURCE, EXECUTE_TASK,
     MPI_COMM_WORLD);
58
59              k++;
60          }
61
62          // Wait for remaining cores to finish
63          for(int i = 1; i < n_cores; i++) {
64              MPI_Recv(local_area, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
    WORKER_AVAILABLE,
65                       MPI_COMM_WORLD, &mstatus);
66
67              total_area += *local_area;
68          }
69
70          // Notify all of the cores that there are no tasks left.
71          for(int i = 1; i < n_cores; i++) {
72              MPI_Send(&k, 1, MPI_DOUBLE, i, NO_MORE_TASKS, MPI_COMM_WORLD);
73          }
74
75          end_t = MPI_Wtime();
76          printf("The area under the curve is \%.16f \n", total_area);
77          total_t = (double)(end_t - start_t);
78          printf("Total time taken by CPU: \%.16f\n", total_t);
79      }
80      else {
81          MPI_Status status;
82          double temp[2] = {0, 0};
83
84          MPI_Send(temp, 1, MPI_DOUBLE, 0, WORKER_AVAILABLE, MPI_COMM_WORLD);
     // Worker initialized. Send message to master saying it's available (
    send 0 as previous computation)
85
86          MPI_Recv(temp, 2, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
    status); // Wait for the task values
87
88          while(status.MPI_TAG != NO_MORE_TASKS){
```

```
89              double a = temp[0];
90              double b = temp[1];
91
92              trap_area = compute_trap_area(a, b);
93
94              *local_area = curve_subarea(a, b, trap_area);
95
96
97              MPI_Send(local_area, 1, MPI_DOUBLE, 0, WORKER_AVAILABLE,
    MPI_COMM_WORLD);
98              MPI_Recv(temp, 2, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
    status);
99          }
100
101     }
102
103     MPI_Finalize();
104
105     free(local_area);
106     stack_destroy(stack);
107
108     return 0;
109 }
```

Listing 2: Quadratura adaptativa com subintervalos gerados e inseridos na bolsa no início do programa

Os dados da Tabela 2 mostram que o tempo de execução reduziu a medida que aumentamos o número de processadores na paralelização. Para um número maior de processadores (8), uma bolsa com 1000 tarefas proporcionou melhor desempenho. Um número alto de subintervalos implica na redução do tamanho da tarefa a ser executada pelos processadores. Isto combinado com um número de processadores relativamente alto pode tornar mais dinâmica a execução em cada processador. Para um número baixo de processadores (2 e 4), a redução do tamanho da tarefa (1000 subintervalos) revelou um resultado pior se comparado com os resultados para tarefas de tamanho maior (10 subintervalos).

N processadores	Subintervalos	Execução 1	Execução 2	Execução 3	Valor Médio
2	2	37.93103 s	38.52947 s	38.06137 s	38.17395 s
	10	31.75973 s	30.61999 s	31.70852 s	31.36274 s
	1000	35.43417 s	40.06360 s	37.16391 s	37.55389 s
4	4	18.41651 s	17.46228 s	17.57564 s	17.81814
	10	12.20700 s	11.83831 s	12.06968 s	12.03833 s
	1000	12.39824 s	12.40765 s	12.50504 s	12.43697 s
8	8	7.12248 s	7.58404 s	7.51093 s	7.40581 s
	10	7.49442 s	7.46962 s	7.41072 s	7.45825 s
	1000	5.00108 s	5.20746 s	5.23567 s	5.14807 s

Tabela 2: Tempo de execução para 2, 4 e 8 processadores e 3 subintervalos distintos

3.2 Variante 2: geração dinâmica de subintervalos

Neste item, além das tags já existentes, criamos a ADD_TASK. O uso desta tag permite que um trabalhor informe ao mestre que um novo item deve ser adicionado na bolsa de tarefas. Para permitir essa adição dinâmica, modificamos a função *curve_subarea* conforme explícito no trecho de código 3. Quando um determinado subintervalo não

atende à tolerância, o trabalhador envia o novo subintervalo superior para a bolsa (linha 21) e continua trabalhando com o cálculo para o subintervalo inferior ao chamar de forma recursiva a função *curve_subarea* (linha 22). Caso o cálculo para o subintervalo inferior também não atenda a tolerância, novos subintervalos serão gerados. O uso do parâmetro maySendBackToMaster (linha 20) garante que o mesmo trabalhador continue trabalhando nesses novos subintervalos.

```
1  void sendBackToMaster(double a, double b) {
2      double temp[2] = {a, b};
3      MPI_Isend(temp, 2, MPI_DOUBLE, 0, ADD_TASK, MPI_COMM_WORLD, &rq);
4  }
5
6
7  void curve_subarea(double a, double b, double area, double *result, int
       maySendBackToMaster){
8      double m, l_area, r_area, error;
9
10     m = (a + b)*0.5;
11     l_area = compute_trap_area(a, m);
12     r_area = compute_trap_area(m, b);
13     error = area - (l_area + r_area);
14
15     if(fabs(error) <= TOL){
16         result[0] = l_area + r_area;
17         result[1] = b - a;
18     }
19     else {
20         if(maySendBackToMaster) {
21             sendBackToMaster(m, b);
22             curve_subarea(a, m, l_area, result, 0);
23         }
24         else {
25             curve_subarea(a, m, l_area, result, 0);
26             double temp[2] = {0, 0};
27             curve_subarea(m, b, r_area, temp, 0);
28             result[0] += temp[0];
29             result[1] += temp[1];
30         }
31     }
32 }
```

Listing 3: Quadratura adaptativa com inserção dinâmica de subintervalos na bolsa

Em comparação com a variante 1, o código do mestre foi alterado conforme explícito no trecho de código 4. Diferentemente da variante 1, onde o mestre ficava em loop até a bolsa se tornar vazia, o mestre na variante 2 se mantém em loop infinto até que a pilha esteja vazia, todos os trabalhadores estejam esperando por tarefas e a soma dos intervalos corresponde ao intervalo inicial (linhas 79-80). Note que nas linhas 7 e 8 definimos as variáveis *progress* e *expected*, onde a primeira armazena a soma total dos intervalos já computados e a segunda corresponde ao valor que *progress* deve alcançar. Quando o mestre recebe a tag WORKER_AVAILABLE de um trabalhador, ele adiciona à area total a area recebida e incrementa a variável *idle*, que indica quantos trabalhadores estão parados. Se a bolsa estiver vazia, o mestre não envia nenhuma tarefa a esse trabalhador e verifica se o número de trabalhadores em idle atingiu ao número total de trabalhadores disponíveis (linha 79). Caso a tenha algum elemento na bolsa, o mestre decrementa a variável *idle* (linha 44) e envia uma nova tarefa a esse trabalhador (linha 46). Note

que além das operações de incremento e decremento da variável *idle*, usamos o vetor *availableWorkers* para registrar quais trabalhadores estão em idle (linha 33 e 45). O mestre usa a informação contida nesse vetor no processo de inserção de um novo elemento na bolsa (linhas 51-61). Quando o mestre recebe a tag ADD_TASK, primeiro ele consuta a variável *idle* para saber se existe algum trabalhador disponível (linha 52). Caso a resposta seja positiva, o mestre decrementa o número de trabalhadores em idle, percorre esse vetor até encontrar o primeiro trabalhador disponível e envia que seria colocada na bolsa para esse trabalhador. Caso a resposta seja negativa, o mestre insere a tarefa na bolsa.

```
1  void master(stack_data *stack, double *params, double total_area, double
       initial_l, double initial_r) {
2      MPI_Status mstatus;
3      int idle = 0;
4
5      int k = 0;
6
7      double progress = 0;
8      double expected = initial_r - initial_l; // delta interval
9
10     int *availableWorkers = (int *) malloc(sizeof(int) * (n_cores - 1));
11
12     if(availableWorkers == NULL) {
13         printf("Unable to create available workers list. Exiting.\n");
14         exit(-1);
15     }
16
17     for(int i = 0; i < n_cores - 1; i++) {
18         availableWorkers[i] = 1;
19     }
20
21     int itCount = 0;
22
23     while(1) {
24         MPI_Recv(params, 2, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
25                  MPI_COMM_WORLD, &mstatus);
26
27         if(mstatus.MPI_TAG == WORKER_AVAILABLE) {
28             total_area += params[0];
29             progress += params[1];
30
31             idle++;
32
33             availableWorkers[mstatus.MPI_SOURCE - 1] = 1;
34
35             if(!stack_is_empty(stack)){
36
37                 stack_node *node = stack_pop(stack);
38
39                 if (node == NULL) {
40                     printf("Error - node is null");
41                     exit(-1);
42                 }
43
44                 idle--;
45                 availableWorkers[mstatus.MPI_SOURCE - 1] = 0;
46                 MPI_Send(node, 2, MPI_DOUBLE, mstatus.MPI_SOURCE,
```

```
        EXECUTE_TASK, MPI_COMM_WORLD);
47
48              }
49
50          }
51          else if(mstatus.MPI_TAG == ADD_TASK) {
52              if(idle){
53                  idle--;
54
55                  int nextAvailableWorker = -1;
56                  for(int i = 0; i < n_cores - 1; i++) {
57                      if(availableWorkers[i] == 1) {
58                          nextAvailableWorker = i + 1;
59                          break;
60                      }
61                  }
62
63                  if(nextAvailableWorker == -1) {
64                      printf("Error - idle indicated there was an available
        worker, but it was not found on available workers list. Exiting.\n");
65                      exit(-1);
66                  }
67
68                  MPI_Send(params, 2, MPI_DOUBLE, nextAvailableWorker,
        EXECUTE_TASK, MPI_COMM_WORLD);
69              }
70              else {
71                  stack_push(stack, params[0], params[1]);
72              }
73          }
74          else {
75              printf("Unknown tag. Exiting.\n");
76              exit(-1);
77          }
78
79          if(idle == (n_cores - 1) && stack_is_empty(stack) && progress/
        expected >= 1.0)
80              break;
81      }
82      // Notify all of the cores that there are no tasks left.
83      for(int i = 1; i < n_cores; i++) {
84          MPI_Send(&k, 2, MPI_DOUBLE, i, NO_MORE_TASKS, MPI_COMM_WORLD);
85      }
86
87      end_t = MPI_Wtime();
88      printf("The area under the curve is %.16f \n", total_area);
89      total_t = (double)(end_t - start_t);
90      printf("Total time taken by CPU: %.16f\n", total_t);
91
92      free(availableWorkers);
93  }
```

Listing 4: Mestre da variante 2

Os dados da Tabela 3 mostram que o tempo de execução reduz com o aumento do número de processadores. Observa-se que houve uma redução de 83.04% do tempo de execução com 8 processadores, se comparado ao resultado para 2 processadores.

N processadores	Execução 1	Execução 2	Execução 3	Valor Médio
2	36.930991 s	38.26990 s	39.18371 s	38.12820 s
4	12.58967 s	12.17632 s	13.33621 s	12.70073 s
8	6.32556 s	6.89510 s	6.17767 s	6.46611 s

Tabela 3: Tempo de execução do programa com geração dinâmica de tarefas para 2, 4 e 8 processadores

Comparando as Tabelas 1, 2 e 3, a variante 1 da quadratura adaptativa com bolsa de tarefas obteve o melhor desempenho na paralelização com 8 processadores e 1000 subintervalos.