UNIVERSIDADE FEDERAL DO RURAL DO SEMI-ÁRIDO
CENTRO MULTIDISCIPLINAR DE PAU DOS FERROS
DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA
PET1681 - TÓPICOS ESPECIAIS EM ENGENHARIA DE SOFTWARE III / PEX0271 - TÓPICOS ESPECIAIS - ENGENHARIA DE SOFTWARE
PROFESSOR: ÍTALO ASSIS

Exercícios Preparatórios para Maratonas de Programação Paralela

Sumário

1,2,3,4,5 = FEITAS

1 Livro [Pacheco and Malensek, 2022] - Capítulo 5: Shared memory programming with OpenMP

1. Suponha que lançamos dardos aleatoriamente em um alvo quadrado. Vamos considerar o centro desse alvo como sendo a origem de um plano cartesiano e os lados do alvo medem 2 pés de comprimento. Suponha também que haja um círculo inscrito no alvo. O raio do círculo é 1 pé e sua área é π pés quadrados. Se os pontos atingidos pelos dardos estiverem distribuídos uniformemente (e sempre acertamos o alvo), então o número de dardos atingidos dentro do círculo deve satisfazer aproximadamente a equação

$$qtd\_no\_circulo$$

$$num\_lancamentos = \frac{\pi}{4} \quad (1)$$

já que a razão entre a área do círculo e a área do quadrado é $\frac{\pi}{4}$.
Podemos usar esta fórmula para estimar o valor de π com um gerador de números alea tórios:

```
qtd_no_circulo = 0;
for (lancamento = 0; lancamento < num_lancamentos; lancamento++) { x =
   double aleatório entre -1 e 1;
   y = double aleatório entre -1 e 1;
   distancia_quadrada = x * x + y * y;
   if (distancia_quadrada <= 1) qtd_no_circulo++;
}
estimativa_de_pi = 4 * qtd_no_circulo/((double) num_lancamentos);
```

Isso é chamado de método "Monte Carlo", pois utiliza aleatoriedade (o lançamento do dardo).

Escreva um programa OpenMP que use um método de Monte Carlo para estimar π. Leia o número total de lançamentos antes de criar as *threads*. Use uma cláusula de reduction para encontrar o número total de dardos que atingem o círculo. Imprima o resultado após encerrar a região paralela. Você deve usar long long ints para o número de acertos no círculo e o número de lançamentos, já que ambos podem ter que ser muito grandes para obter uma estimativa razoável de π.

```c
typedef long long int ll;

double monte_carlo(ll num_lancamentos) {
 ll qtd_no_circulo = 0;
 double distancia_quadrada;
 double x, y;

#pragma omp parallel private(x, y) shared(num_lancamentos)
#pragma omp for reduction(+ : qtd_no_circulo)
 for (ll lancamento = 0; lancamento < num_lancamentos; lancamento++) {

   unsigned rank = omp_get_thread_num();

   x = rand_r(&rank) / (double)RAND_MAX * 2 - 1;
   y = rand_r(&rank) / (double)RAND_MAX * 2 - 1;

   distancia_quadrada = x * x + y * y;

   if (distancia_quadrada <= 1)
     qtd_no_circulo++;
 }

 return 4 * qtd_no_circulo / ((double)num_lancamentos);
```

```
}
```

```
int main(int argc, char *argv[]) {
 ll num_lancamentos = std::atoi(argv[1]);
 double s, e;

 s = omp_get_wtime();
 std::cout << "" << monte_carlo(num_lancamentos) << std::endl;
 e = omp_get_wtime();

 std::cout << "Elapsed time: " << e - s << std::endl;
 return 0;
}
```

2. *Count sort* é um algoritmo de ordenação serial simples que pode ser implementado da seguinte forma:

```
void Count_sort(int a[], int n) {
   int i, j, count;
   int* temp = malloc(n*sizeof(int));
   for (i = 0; i < n; i++) {
      count = 0;
      for (j = 0; j < n; j++)
         if (a[j] < a [i])
            count++;
         else if (a[j] == a[i] && j < i)
```



```
            count++;
      temp[count] = a[i];
}
memcpy(a, temp, n*sizeof(int));
free(temp);
}
```

A ideia básica é que para cada elemento a[i] na lista a, contemos o número de ele mentos da lista que são menores que a[i]. Em seguida, inserimos a[i] em uma lista temporária usando o índice determinado pela contagem. Há um pequeno problema com esta abordagem quando a lista contém elementos iguais, uma vez que eles podem ser atribuídos ao mesmo slot na lista temporária. O código lida com isso incrementando a contagem de elementos iguais com base nos índices. Se a[i] == a[j] e j <i, então contamos a[j] como sendo "menor que"a[i].

Após a conclusão do algoritmo, sobrescrevemos o *array* original pelo *array* temporário usando a função da biblioteca de *strings* memcpy.

(a) Se tentarmos paralelizar o laço for i (o laço externo), quais variáveis devem ser

privadas e quais devem ser compartilhadas?
Privadas = i,j, count;
Compartilhadas = a, temp, n;

(b) Se paralelizarmos o laço for i usando o escopo especificado na parte anterior, haverá alguma dependência de dados no laço? Explique sua resposta.

Não haverá dependência, pois cada thread irá trabalhar sob um pedacinho de a. Fazendo suas contagens individualmente, cada thread acessa um índice diferente do vetor *temp.*

(c) Podemos paralelizar a chamada para memcpy? Podemos modificar o código para que esta parte da função seja paralelizável?

Podemos aplicar uma distribuição em blocos para a chamada do memcpy dentro da região paralela, bastava sincronizar as threads antes da implementação.

(d) Escreva um programa em C que inclua uma implementação paralela do *Count sort*.

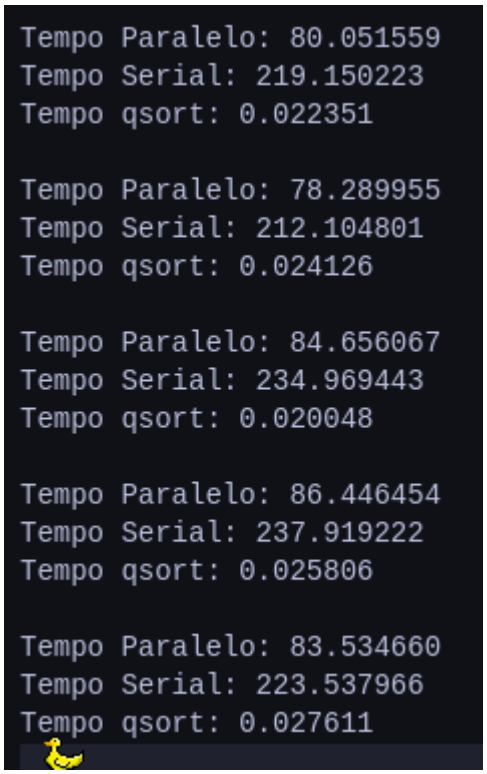
```c
/*Author: Felipe Hidequel
 Date: 2025-05-14
 Description: Parallel implementation of counting sort using OpenMP.
 Execution: ./run.sh <size>
 Note: The array is filled with random numbers between 0 and 9.
 The array is sorted in ascending order.
*/

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/*  count_sort_parallel - Parallel implementation of counting sort
 @a: array to be sorted
 @n: size of the array
*/
void count_sort_parallel(int a[], int n) {
 int i, j, count;
 int *temp = malloc(n * sizeof(int));
 int rest, id, chunksize;
 int start;

#pragma omp parallel default(none) private(i, j, count, start, id, chunksize)  \
   shared(a, temp, n, rest)
 {
#pragma omp for
   for (i = 0; i < n; i++) {
     count = 0;
     for (j = 0; j < n; j++)
       if (a[j] < a[i])
         count++;
```

```
      else if (a[j] == a[i] && j < i)
        count++;
    temp[count] = a[i];
  }

#pragma omp barrier
  rest = n % omp_get_num_threads();
  id = omp_get_thread_num();
  chunksize = n / omp_get_num_threads() + (id < rest);

  start = id * chunksize + (id < rest ? id : rest);
  memcpy(&a[start], &temp[start], chunksize * sizeof(int));
 }

 free(temp);
}

int compara(const void *a, const void *b) {
 int fa = *(const int *)a;
 int fb = *(const int *)b;

 if (fa < fb)
   return -1;
 else if (fa > fb)
   return 1;
 else
   return 0;
}

int main(int argc, char *argv[]) {
 if (argc < 2) {
   printf("Usage: <executavel> <tamanho do vetor>\n");
 }

 double start, end;
 int n = atoi(argv[1]);
 int a[n], b[n], c[n];

 for (int i = 0; i < n; i++) {
   a[i] = rand() % 10;
   b[i] = a[i];
   c[i] = a[i];
 }

 printf("\n");

 start = omp_get_wtime();
 count_sort_parallel(a, n);
 end = omp_get_wtime();
```

```c
 printf("Tempo Paralelo: %f\n", end - start);

 start = omp_get_wtime();
 count_sort(b, n);
 end = omp_get_wtime();
 printf("Tempo Serial: %f\n", end - start);

 start = omp_get_wtime();
 qsort(c, n, sizeof(int), compara);
 end = omp_get_wtime();
 printf("Tempo qsort: %f\n", end - start);


 return 0;
}
```

(e) Como o desempenho da sua paralelização do *Count sort* se compara à classificação serial? Como ela se compara à função serial qsort?

Para N = 300000 e 4 threads



A implementação paralela demonstrou até **63.97% de ganho** em relação à versão serial. Já o `qsort` mostrou até **99.98% de ganho** em relação à minha proposta. Isso

pode ser explicado pela complexidade do `qsort` ser O(n log n), muito melhor quando comparada à O(n²) da versão com `counting`.

3. Lembre-se de que quando resolvemos um grande sistema linear, frequentemente usa mos a eliminação gaussiana seguida de substituição regressiva. A eliminação gaussiana converte um sistema linear $n \times n$ em um sistema linear triangular superior usando "ope rações de linha".

   • Adicione um múltiplo de uma linha a outra linha

   • Troque duas linhas

   • Multiplique uma linha por uma constante diferente de zero

   Um sistema triangular superior tem zeros abaixo da "diagonal"que se estende do canto superior esquerdo ao canto inferior direito. Por exemplo, o sistema linear

$$\begin{aligned} 2x_0 - 3x_1 &= 3 \\ 4x_0 - 5x_1 + x_2 &= 7 \\ 2x_0 - x_1 - 3x_2 &= 5 \end{aligned}$$

   pode ser reduzido à forma triangular superior

$$\begin{aligned} 2x_0 - 3x_1 &= 3 \\ x_1 + x_2 &= 1 \\ -5x_2 &= 0 \end{aligned}$$

   3

   e este sistema pode ser facilmente resolvido encontrando primeiro x_2 usando a última equação, depois encontrando x_1 usando a segunda equação e finalmente encontrando x_0 usando a primeira equação.

   Podemos desenvolver alguns algoritmos seriais para substituição reversa. A versão "ori entada a linhas"é

```
for (lin = n-1; lin >= 0; lin--) {
   x[lin] = b[lin];
   for (col = lin+1; col < n; col++)
      x[lin] -= A[lin][col]*x[ col];
   x[lin] /= A[lin][lin];
}
```

   Aqui, o "lado direito"do sistema é armazenado na matriz b, a matriz bidimensional de coeficientes é armazenada na matriz A e as soluções são armazenadas na matriz x. Uma alternativa é o seguinte algoritmo "orientado a colunas":

```
for (lin = 0; lin < n; lin++)
   x[lin] = b[lin];
for (col = n-1; col >= 0; col--) {
   x[col] /= A[col][col];
   for (lin = 0; lin < col; lin++)
   x[lin] -= A[lin][col]*x[ col];
```

}

(a) Determine se o laço externo do algoritmo orientado a linhas pode ser paralelizado.

O valor de x[lin] depende de x[col], o que impossibilita eu paralelizar o laço externo. Seria como resolver a linha de cima antes de resolver a linha de baixo.

(b) Determine se o laço interno do algoritmo orientado a linhas pode ser paralelizado.

Pode, sim, daria para paralelizar ele com uma redução

(c) Determine se o (segundo) laço externo do algoritmo orientado a colunas pode ser paralelizado.
Não. Existe uma dependência de leitura e escrita, pois nesse método você pode alterar a mesma linha em outras iterações. Col-1 só deve ser acessado quando Col for resolvida.

(d) Determine se o laço interno do algoritmo orientado a colunas pode ser paralelizado.

Pode sim.

(e) Escreva um programa OpenMP para cada um dos loops que você determinou que poderiam ser paralelizados. Você pode achar a diretiva single útil - quando um bloco de código está sendo executado em paralelo e um sub-bloco deve ser exe cutado por apenas uma *thread*, o sub-bloco pode ser modificado por uma diretiva #pragma omp single. As *threads* serão bloqueadas no final da diretiva até que to das as threads a tenha concluído.

```c
/* Resolve o sistema triangular superior Ax = b paralelo
   A: matriz triangular superior
   b: lado direito do sistema
   x: vetor solução
*/
float *triangular_superior_colunas_p(float **A, float *B, int n) {
  float *x = (float *)malloc(n * sizeof(float));

#pragma omp for schedule(runtime)
  for (int lin = 0; lin < n; lin++)
    x[lin] = B[lin];

  for (int col = n - 1; col >= 0; col--) {
    x[col] /= A[col][col];

#pragma omp for schedule(runtime)
    for (int lin = col - 1; lin >= 0; lin--)
      x[lin] -= A[lin][col] * x[col];
  }

  return x;
}
```

```c
/* Resolve o sistema triangular superior Ax = b paralelo
   A: matriz triangular superior
   b: lado direito do sistema
   x: vetor solução
*/
float *triangular_superior_linhas_p(float **A, float *B, int n) {
  float *x = (float *)malloc(n * sizeof(float));
  int col, lin;
  float soma;

  for (lin = n - 1; lin >= 0; lin--) {
    soma = 0.0F;

#pragma omp parallel for default(none) reduction(+ : soma) shared(A,B,x,n,lin) private(col) schedule(runtime)
    for (col = lin + 1; col < n; col++) {
      soma += A[lin][col] * x[col];
    }

    x[lin] = (B[lin] - soma) / A[lin][lin];
  }

  return x;
}
```

(f) Modifique seu laço paralelo com uma cláusula schedule(runtime) e teste o pro grama com vários escalonamentos. Se o seu sistema triangular superior tiver 10.000 variáveis, qual escalonamento oferece o melhor desempenho?

```
Executando Colunas com 4 threads e escalonador static
Tempo de execução: 0.299430 segundos
Tempo de execução Paralelo: 0.289315 segundos

Executando Linhas com 4 threads e escalonador static
Tempo de execução: 0.128743 segundos
Tempo de execução paralelo: 0.042883 segundos

Executando Colunas com 4 threads e escalonador dynamic
Tempo de execução: 0.293913 segundos
Tempo de execução Paralelo: 1.320095 segundos

Executando Linhas com 4 threads e escalonador dynamic
Tempo de execução: 0.132060 segundos
Tempo de execução paralelo: 1.156573 segundos

Executando Colunas com 4 threads e escalonador guided
Tempo de execução: 0.307996 segundos
Tempo de execução Paralelo: 0.301823 segundos

Executando Linhas com 4 threads e escalonador guided
Tempo de execução: 0.124268 segundos
Tempo de execução paralelo: 0.057974 segundos
```

O escalonador `static` apresentou o melhor desempenho geral, especialmente na operação de linhas.

Já o escalonador `dynamic` foi o pior, apresentando tempos paralelos até 10 vezes maiores que os seriais, o que pode indicar sobrecarga na gerência dinâmica de tarefas em loops com granulação muito fina

4. Use OpenMP para implementar um programa que faça eliminação gaussiana (veja o pro blema anterior). Você pode assumir que o sistema de entrada não precisa de nenhuma troca de linha.

```
(base) felipehidequel@la-maquina:~/Documents/maratona_pcd/reducao_gaussiana$ ./run.sh 9999
gcc -fopenmp -O3 -Wall -o main main.c -lm
Tempo Sequencial: 106.540640
Tempo Paralelo: 87.097444
Resultado: CORRETO
(base) felipehidequel@la-maquina:~/Documents/maratona_pcd/reducao_gaussiana$
```

```c
void eliminacao_gaussiana_p(float **A, float *B, int n) {
 int pivot, col, lin;

 for (pivot = 0; pivot < n - 1; pivot++) {
   if (A[pivot][pivot] == 0.0) {
     printf("Erro: pivô nulo encontrado na linha %d.\n", pivot);
     exit(1);
   }

#pragma omp parallel for default(none) shared(A, B, n, pivot) private(lin, col)
   for (lin = pivot + 1; lin < n; lin++) {
     float fator = A[lin][pivot] / A[pivot][pivot];

#pragma omp simd
     for (col = pivot; col < n; col++) {
       A[lin][col] -= fator * A[pivot][col];
     }

     B[lin] -= fator * B[pivot];
   }
 }
}

void eliminacao_gaussiana(float **A, float *B, int n) {
 for (int pivot = 0; pivot < n - 1; pivot++) {
   if (A[pivot][pivot] == 0.0) {
     printf("Erro: pivô nulo encontrado na linha %d.\n", pivot);
     exit(1);
   }

   for (int lin = pivot + 1; lin < n; lin++) {
     float fator = A[lin][pivot] / A[pivot][pivot];

     for (int col = pivot; col < n; col++) {
       A[lin][col] -= fator * A[pivot][col];
     }

     B[lin] -= fator * B[pivot];
   }
 }
}

void validacao(float **A, float **B, int n) {
 int ok = 1;

 for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
     if (fabs(A[i][j] - B[i][j]) > EPSILON) {
```

```
        ok = 0;
        break;
      }
    }
  }

  printf("Resultado: %s\n", (ok == 0) ? "ERRADO" : "CORRETO");
  return;
}
```

5. Use OpenMP para implementar um programa produtor-consumidor no qual algumas *th reads* são produtoras e outras são consumidoras. As produtoras leem o texto de uma coleção de arquivos, um por produtor. Elas inserem linhas de texto em uma única fila compartilhada. Os consumidores pegam as linhas do texto e as tokenizam. *Tokens* são "palavras" separadas por espaço em branco. Quando uma consumidora encontra um *token*, ela o grava no stdout.

Fila.h:

```
// Fila.h
#ifndef FILA_H
#define FILA_H

typedef struct no {
 char *linha;
 struct no *proximo;
} No;

typedef struct fila {
 No *inicio;
 No *fim;
 int tamanho;
} Fila;

Fila *cria_fila();

void insere_fila(Fila *f, char *linha);

char *remove_fila(Fila *f);
```

```
void libera_fila(Fila *f);

#endif // FILA_H
```

Fila.c:

```
#include "fila.h"
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

Fila *cria_fila() {
 Fila *f = (Fila *)malloc(sizeof(Fila));
 f->inicio = NULL;
 f->fim = NULL;
 f->tamanho = 0;
 return f;
}

void insere_fila(Fila *f, char *linha) {
 No *novo = (No *)malloc(sizeof(No));
 novo->linha = linha;
 novo->proximo = NULL;

#pragma omp critical
 {
   if (f->tamanho == 0) {
     f->inicio = novo;
     f->fim = novo;
   } else {
     f->fim->proximo = novo;
     f->fim = novo;
   }
   f->tamanho++;
 }
}

char *remove_fila(Fila *f) {
 No *removido = NULL;
 char *linha = NULL;

#pragma omp critical
 {
   if (f->tamanho != 0) {
     removido = f->inicio;
     linha = removido->linha;

     f->inicio = removido->proximo;
```

```
    f->tamanho--;

    if (f->tamanho == 0) {
      f->fim = NULL;
    }
  }
 }

 if (removido != NULL) {
   free(removido);
 }

 return linha;
}

void libera_fila(Fila *f) {
 No *atual = f->inicio;
 while (atual != NULL) {
   No *proximo = atual->proximo;
   free(atual);
   atual = proximo;
 }
 free(f);
}
```

main.c:

```
#include "fila.h"
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void produtor(Fila *fila, int *produtores, int rank) {
 FILE *arquivo;
 char nome[15];
 char *linha = NULL, *lido;
 size_t tamanho = 0;

 snprintf(nome, sizeof(nome), "input-%d.txt", rank);
 arquivo = fopen(nome, "rt");

 if (arquivo == NULL) {
   fprintf(stderr, "Erro ao abrir arquivo: %s", nome);
```

```c
#pragma omp atomic
   (*produtores)--;

   return;
 }

 while (getline(&linha, &tamanho, arquivo) != -1) {
   lido = strdup(linha);
   if (lido != NULL) {
     insere_fila(fila, lido);
   }
 }

 free(linha);
 fclose(arquivo);

#pragma omp atomic
 (*produtores)--;
}

void consumidor(Fila *fila, int *n_tokens, int rank, int *produtores) {
 char *linha = NULL;

 while (linha || *produtores != 0) {
   linha = remove_fila(fila);

   if (linha != NULL) {
     char *token, *ptr;
     token = strtok_r(linha, " \n", &ptr);

     while (token != NULL) {
       printf("Thread %d consumiu o token: %s\n", rank, token);

#pragma omp atomic
       (*n_tokens)++;

       token = strtok_r(NULL, " \n", &ptr);
     }
     free(linha);
   }
 }
}

int main(int argc, char *argv[]) {
 if (argc < 2) {
   printf("Usage:\n");
   printf("./run.sh <quantidade_ranks>");
 }
```

```c
 int n_ranks, n_tokens = 0, produtores, rank;
 Fila *fila;
 n_ranks = atoi(argv[1]);
 fila = cria_fila();

 produtores = (n_ranks / 2) + (n_ranks & 1);

#pragma omp parallel num_threads(n_ranks) default(none)                         \
   shared(produtores, fila, n_tokens) private(rank)
 {
   rank = omp_get_thread_num();

   if (rank < produtores)
     produtor(fila, &produtores, rank);
   else
     consumidor(fila, &n_tokens, rank, &produtores);
 }

 printf("Tokens lidos: %d\n", n_tokens);

 libera_fila(fila);

 return 0;
}
```

Referências

Peter S. Pacheco and Matthew Malensek. *An Introduction to Parallel Programming*. El sevier, 2 edition, 2022. ISBN 9780128046050. doi: 10.1016/C2015-0-01650-1. URL https://linkinghub.elsevier.com/retrieve/pii/C20150016501.