Universidade Federal do ABC
Prova de MCZA020-13— Programação Paralela
Prova 2

Turma DAMCZA020-13SA
2018.Q3

Emilio Francesquini

Nome:

RA:

Questão	Pontos	Nota
1	1½	
2	3	
3	2½	
4	3	
Total:	10	

- A prova tem duração de **110 minutos**
- Não esqueça de preencher a caneta **todos** os campos na primeira página da folha de prova e seu nome e RA nesta folha de questões.
- Antes da resposta da questão i escreva "Questão i" em letras garrafais.
- As respostas às questões podem ser deixadas a lápis.
- Ao final da prova entregue tanto a folha de questões quanto a folha de prova.
- Em caso de fraude, **TODOS** os envolvidos:
  - **Receberão conceito final F (reprovado) na disciplina**
  - Serão **denunciados** à Comissão de Transgressões Disciplinares Discentes da Graduação e à Comissão de Ética da UFABC cuja punição pode resultar em **advertência, suspensão ou desligamento**, de acordo com os artigos 78-82 do Regimento Geral da UFABC e do artigo 25 do Código de Ética da UFABC.

Boa Prova!

Questão 1. Nesta disciplina exploramos em detalhes três ferramentas para a programação paralela: MPI, PThreads e OpenMP. Sobre essas três ferramentas responda:

(a) (½ ponto) Quais são as principais diferenças entre cada uma dessas ferramentas? Quais são os motivos que nos fariam escolher uma em detrimento das demais?

(b) (1 ponto) No primeiro projeto implementamos uma versão paralela do Crivo de Erastótenes. Essa paralelização foi feita utilizando MPI. Escolha PThreads ou OpenMP e descreva em detalhes como a paralelização no seu projeto poderia ser feita com esta outra ferramenta. Quais seriam as vantagens e as desvantagens da ferramenta escolhida com relação ao MPI?

Questão 2. (3 pontos) Um pipeline simples apropriado para um *media player* para *streaming* consiste de um thread monitorando a porta de rede para verificar o recebimento de dados; um thread para descomprimir os pacotes recebidos e gerar os frames de um vídeo; e um thread que faz o *rendering*, ou seja, que desenha os frames em intervalos pré-programados de tempo na tela. Os três threads de uma aplicação como esta precisam comunicar entre si através de buffers compartilhados na memória. Um buffer é usado entre o thread responsável pela rede e o thread descompressor, e outro buffer entre o descompressor e o thread que faz o rendering. Descreva **em detalhes** (preferencialmente em código) como usando PThreads (não é preciso dizer o nome exato das funções, mas é necessário utilizar os nomes corretos das estruturas sobre as quais elas operam e os nomes das operações) seria possível fazer a implementação de tal pipeline.

Questão 3. (2½ pontos) Considere o seguinte programa OpenMP e responda:

```
1  int x = 0;
2  int y = 0;
3
4  void foo1() {
5  #   pragma omp critical(x)
6      {
7          foo2();
8          x += 1;
9      }
10 }
11
12 void foo2() {
13 #   pragma omp critical(y)
14     y += 1;
15 }
16
17 void foo3() {
18 #   pragma omp critical(y)
19     {
20         y -= 1;
21         foo4();
22     }
23
24 }
25
26 void foo4() {
27 #   pragma omp critical(x)
28     x -= 1;
29 }
30
31 int main(void) {
32
33 #   pragma omp parallel private(i)
34     {
35         for (i = 0; i < 10; i++) {
36             foo1();
37             foo3();
38         }
39     }
40
41     printf("%d %d \n", x, y);
42 }
```

Assuma que dois threads executam este código e em dois cores de um processador multicore. É possível ocorrer um *deadlock* durante a execução? Caso positivo descreva a sequência de operações que deve ocorrer para produzir um *deadlock*. Caso negativo mostre porque um *deadlock* é impossível de ocorrer.

Questão 4. Count sort é um algoritmo de ordenação bem simples que pode ser implementado da seguinte maneira:

```
1  void Count sort (int a [], int n) {
2      int i, j, count;
3      int *temp = malloc (n * sizeof(int));
4
5      for (i = 0; i < n ; i ++) {
6          count = 0;
7          for (j = 0; j < n ; j ++)
8              if (a [j] < a [i])
9                  count ++;
10             else if (a[j] == a[i] && j < i)
11                 count ++;
12         temp[count] = a[i];
13     }
14     memcpy (a, temp, n * sizeof(int));
15     free (temp);
16 }
```

A ideia deste algoritmo é que para cada elemento `a[1]` no vetor `a`, contamos o número de elementos no vetor que são menores que `a[1]`. Inserimos então o elemento `a[1]` em uma lista temporária usando o índice determinado pela contagem. Existe um pequeno problema caso a lista contenha elementos repetidos já que eles poderiam ser associados ao mesmo índice na lista temporária. O código acima lida com este problema incrementando a contagem dos elementos baseado em seus índices. Se ambos `a[i] == a[j]` e `j < i`, então contamos `a[j]` como sendo menor que `a[i]`. Apos a execução do algoritmo sobrescrevemos o vetor original com o vetor temporário usando a função `memcpy`.

(a) (½ ponto) Caso queiramos paralelizar o laço da linha 5, quais variáveis devem ser marcadas como privadas (*private*) e quais devem ser compartilhadas (*shared*)?

(b) (½ ponto) Caso paralelizemos o laço do item anterior utilizando as regras de escopo definidas na sua resposta, há alguma dependência entre as iterações do laço (*loop-carried dependence*) que devamos considerar? Explique a sua resposta.

(c) (½ ponto) Seria possível paralelizar a chamada à função `memcpy`? Caso negativo, poderíamos modificar essa parte do código para tornar isto possível? Justifique sua resposta

(d) (1 ponto) Descreva claramente e em detalhes quais diretivas OpenMP você utilizaria e onde as utilizaria para paralelizar o código acima.

(e) (½ ponto) Critique a paralelização desenvolvida no item anterior. Dê a sua opinião sobre o desempenho que pode-se esperar da sua paralelização.