Universidade Federal do Rural do Semi-Árido
Centro Multidisciplinar de Pau dos Ferros
Departamento de Engenharias e Tecnologia
PEX1272 - Programação Concorrente e Distribuída
Docente: Ítalo Assis

Lista de Exercícios

Programação de memória compartilhada com OpenMP

Atenção: as questões 1, 4, 6, 7 e 13 devem ser executadas no supercomputador Santos Dumont incluindo execuções com 8 *threads* ou mais. O *script* utilizado para execução do programa deve ser incluído na resposta.

1. Baixe o arquivo `omp_trap_1.c` do site do livro e exclua a diretiva `critical`. Compile e execute o programa com cada vez mais *threads* e valores cada vez maiores de n.

   (a) Quantas *threads* e quantos trapézios são necessários antes que o resultado esteja incorreto?

   (b) Como o aumento do número de trapézios influencia nas chances do resultado ser incorreto?

   (c) Como o aumento do número de *threads* influencia nas chances do resultado ser incorreto?

2. Baixe o arquivo `omp_trap1.c` do site do livro. Modifique o código para que

   - ele use o primeiro bloco de código da página 222 do livro e
   - o tempo usado pelo bloco paralelo seja cronometrado usando a função OpenMP `omp_get_wtime()`. A sintaxe é

     ```
     double omp_get_wtime(void)
     ```

     Ele retorna o número de segundos que se passaram desde algum tempo no passado. Para obter detalhes sobre cronometragem, consulte a Seção 2.6.4. Lembre-se também de que o OpenMP possui uma diretiva de barreira:

     ```
     # pragma omp barrier
     ```

     Agora encontre um sistema com pelo menos dois núcleos e cronometre o programa com

   - uma *thread* e um grande valor de n, e
   - duas *threads* e o mesmo valor de n.

   (a) O que acontece?

   (b) Baixe o arquivo `omp_trap2b.c` do site do livro. Como seu desempenho se compara?

   Explique suas respostas.

3. Suponha que no incrível computador Bleeblon, variáveis com tipo float possam armazenar três dígitos decimais. Suponha também que os registradores de ponto flutuante do Bleeblon possam armazenar quatro dígitos decimais e que, após qualquer operação de ponto flutuante, o resultado seja arredondado para três dígitos decimais antes de ser armazenado. Agora suponha que um programa C declare um *array* a da seguinte forma:

```
float a[] = {4.0, 3.0, 3.0, 1000.0};
```

   (a) Qual é a saída do seguinte bloco de código se ele for executado no Bleeblon? Justifique sua resposta.

```
int i ;
float sum = 0.0;
for (i = 0; i < 4; i++)
  sum += a[i];
printf ("sum = %4.1f\n", sum );
```

   (b) Agora considere o seguinte código:

```
int i;
float sum = 0.0;
#pragma omp parallel for num threads (2) reduction (+:sum)
for (i = 0; i < 4; i++)
  sum += a[i];
printf("sum = %4.1f\n", sum );
```

   Suponha que o sistema operacional atribua as iterações $i = 0, 1$ à *thread* 0 e $i = 2, 3$ à *thread* 1. Qual é a saída deste código no Bleeblon? Justifique sua resposta.

4. Escreva um programa OpenMP que determine o escalonamento padrão de laços `for` paralelos. Sua entrada deve ser o número de iterações e quantidade de *threads* e sua saída deve ser quais iterações de um laço `for` paralelizado são executadas por qual *thread*. Por exemplo, se houver duas *threads* e quatro iterações, a saída poderá ser:

```
Thread 0: Iterações 0 -- 1
Thread 1: Iterações 2 -- 3
```

   (a) De acordo com a execução do seu programa, qual é o escalonamento padrão de laços `for` paralelos de um programa OpenMP? Porque?

5. Considere o seguinte laço:

```
a[0] = 0;
for ( i = 1; i < n ; i++)
  a[i] = a[i-1] + i;
```

   Há claramente uma dependência no laço já que o valor de `a[i]` não pode ser calculado sem o valor de `a[i-1]`. Sugira uma maneira de eliminar essa dependência e paralelizar o laço.

6. Modifique o programa da regra do trapézio que usa uma diretiva `parallel for` (`omp_trap_3.c`) para que o `parallel for` seja modificado por uma cláusula `schedule(runtime)`. Execute o programa com várias atribuições à variável de ambiente `OMP_SCHEDULE` e determine quais iterações são atribuídas a qual *thread*. Isso pode ser feito alocando um *array* `iteracoes` de n `int`'s e, na função `Trap`, atribuindo `omp_get_thread_num()` a `iteracoes[i]` na i-ésima iteração do laço `for`. Qual é o escalonamento padrão de iterações em seu sistema? Como o escalonamento `guided` é determinado?

7. Lembre-se de que todos os blocos estruturados modificados por uma diretiva `critical` formam uma única seção crítica. O que acontece se tivermos um número de diretivas `atomic` nas quais diferentes variáveis estão sendo modificadas? Todas elas são tratadas como uma única seção crítica?

   Podemos escrever um pequeno programa que tente determinar isso. A ideia é fazer com que todas as *threads* executem simultaneamente algo como o código a seguir:

   ```
   int i;
   double minha_soma = 0.0;
   for (i = 0; i < n; i++)
     #pragma omp atomic
     minha_soma += sin(i);
   ```

   Podemos fazer isso modificando o código com uma diretiva `parallel`:

   ```
   # pragma omp parallel num threads(thread_count)
   {
     int i;
     double minha_soma = 0.0;
     for (i = 0; i < n; i ++)
       #pragma omp atomic
       minha_soma += sin( i);
   }
   ```

   Observe que já que `minha_soma` e `i` são declaradas no bloco paralelo, cada *thread* possui sua própria cópia privada. Agora, se medirmos o tempo desse código para um `n` grande com `thread_count = 1` e também quando `thread_count > 1`, contanto que `thread_count` seja menor que o número de núcleos disponíveis, o tempo de execução para a execução de *thread* única deveria ser aproximadamente o mesmo que o tempo para a execução com múltiplas *threads* se as diferentes execuções de `minha_soma += sin(i)` são tratadas como diferentes seções críticas. Por outro lado, se as diferentes execuções de `minha_soma += sin(i)` são todas tratadas como uma única seção crítica, a execução com múltiplas *threads* deve ser muito mais lenta que a execução de *thread* única. Escreva um programa OpenMP que implemente este teste. Sua implementação do OpenMP permite a execução simultânea de atualizações para diferentes variáveis quando as atualizações são protegidas por diretivas `atomic`?

8. Baixe o arquivo `omp_mat_vect_rand_split.c` no site do livro. Encontre um programa que faça o perfilamento de *cache* (por exemplo, Valgrind) e compile o programa de acordo

com as instruções na documentação do perfilador de *cache* (por exemplo, com Valgrind e compilador gcc você desejará uma tabela de símbolos e otimização completa, ou seja, `gcc -g -O2 . . .`). Execute o programa de acordo com as instruções na documentação do perfilador de *cache* usando a entrada $k \times (k \times 10^6)$, $(k \times 10^3) \times (k \times 10^3)$ e $(k \times 10^6) \times k$. Escolha k tão grande que o número de falhas de *cache* de nível 2 seja da ordem 10^6 para pelo menos uma das entradas conjuntos de dados.

(a) Quantas falhas de escrita no cache de nível 1 ocorrem com cada uma das três entradas?

(b) Quantas falhas de escrita no cache de nível 2 ocorrem com cada uma das três entradas?

(c) Onde ocorre a maioria das falhas de escrita? Para quais dados de entrada o programa apresenta mais falhas de escrita? Você pode explicar por quê?

(d) Quantas falhas de leitura de cache de nível 1 ocorrem com cada uma das três entradas?

(e) Quantas falhas de leitura de cache de nível 2 ocorrem com cada uma das três entradas?

(f) Onde ocorre a maioria das falhas de leitura? Para quais dados de entrada o programa apresenta mais falhas de leitura? Você pode explicar por quê?

(g) Execute o programa com cada uma das três entradas, mas sem usar o perfilador de *cache*. Com qual entrada o programa é mais rápido? Com qual entrada o programa é mais lento? Suas observações sobre falhas de *cache* podem ajudar a explicar as diferenças? Como?

9. Lembre-se do exemplo de multiplicação de matrizes e vetores com a entrada 8000×8000. Assuma que uma linha de *cache* contém 64 *bytes* ou 8 `double`s.

   (a) Suponha que a *thread* 0 e a *thread* 2 sejam atribuídas a processadores diferentes. É possível que ocorra um falso compartilhamento entre as *threads* 0 e 2 para alguma parte do vetor `y`? Por que?

   (b) E se a *thread* 0 e a *thread* 3 forem atribuídas a processadores diferentes? É possível que ocorra um falso compartilhamento entre elas para alguma parte de `y`?

10. Embora `strtok_r` seja *thread-safe*, ele tem a propriedade bastante infeliz de modificar a *string* de entrada. Escreva um método para gerar *tokens* que seja *thread-safe* e não modifique a *string* de entrada.

11. Utilizando OpenMP, implemente o programa paralelo do histograma discutido no Capítulo 2.

Questões extra

12. 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

$$\frac{qtd\_no\_circulo}{num\_lancamentos} = \frac{\pi}{4} \tag{1}$$

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 aleatórios:

```
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 int`s 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.

13. *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 elementos 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?

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

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

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

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

14. Lembre-se de que quando resolvemos um grande sistema linear, frequentemente usamos a eliminação gaussiana seguida de substituição regressiva. A eliminação gaussiana converte um sistema linear $n \times n$ em um sistema linear triangular superior usando "operações 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

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

    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 "orientada 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.

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

(c) Determine se o (segundo) laço externo do algoritmo orientado a colunas pode ser paralelizado.

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

(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 executado 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 todas as threads a tenha concluído.

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

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

16. Use OpenMP para implementar um programa produtor-consumidor no qual algumas *threads* 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`.

Referência

- PACHECO, Peter S. An introduction to parallel programming. Amsterdam Boston: Morgan Kaufmann, c2011. xix, 370 p. ISBN: 9780123742605.