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

Lista de Exercícios

Sumário

1 Porque computação paralela?

1. Suponha que precisamos calcular n valores e somá-los. Suponha que também tenhamos p núcleos e p seja muito menor que n. Então cada núcleo pode calcular uma soma parcial de aproximadamente valores n/p da seguinte maneira:

```
minha_soma = 0;
meu_pri_i = . . . ;
meu_ult_i = . . . ;
for (meu_i = meu_pri_i; meu_i < meu_ult_i; meu_i+=meu_desl) {
  meu_x = Compute_prox_valor(. . .);
  minha_soma += meu_x;
}
```

   Aqui o prefixo $meu\_$ indica que cada núcleo está usando suas próprias variáveis privadas e cada núcleo pode executar este bloco de código independentemente dos outros núcleos.

   Supondo que cada chamada para $Compute\_prox\_valor$ requer aproximadamente a mesma quantidade de trabalho, elabore fórmulas para calcular meu_pri_i, meu_ult_i e meu_desl. Lembre-se de que cada núcleo deve receber aproximadamente o mesmo número de elementos de computação no loop.

   Dica: primeiro considere o caso em que n é divisível por p. A partir daí, elabore fórmulas para o caso em que n não é divisível por p.

2. Suponha que precisamos calcular n valores e somá-los. Suponha também que tenhamos p núcleos e p seja muito menor que n. Então cada núcleo pode calcular uma soma parcial de aproximadamente n/p valores da seguinte maneira:

```
minha_soma = 0;
meu_pri_i = . . . ;
meu_ult_i = . . . ;
for (meu_i = meu_pri_i; meu_i < meu_ult_i; meu_i+=meu_desl) {
  meu_x = Compute_prox_valor(. . .);
  minha_soma += meu_x;
}
```

   Aqui o prefixo $meu\_$ indica que cada núcleo está usando suas próprias variáveis privadas e cada núcleo pode executar este bloco de código independentemente dos outros núcleos.

   Considerando que a chamada com $i = k$ requer $k+1$ vezes mais trabalho que a chamada com $i = 0$ (se a chamada com $i = 0$ requer 2 milissegundos, a chamada com $i = 1$ requer 4, a chamada com $i = 2$ requer 6...), elabore fórmulas para calcular meu_pri_i, meu_ult_i e meu_desl. Comparado com a distribuição em blocos contíguos de iterações, sua fórmula deve reduzir a diferença do tempo de execução entre os núcleos.

3. Escreva um pseudocódigo para uma soma global estruturada em árvore. Suponha que o número de núcleos seja uma potência de dois.

4. Podemos usar os operadores bit a bit de C para implementar uma soma global estruturada em árvore. Implemente este algoritmo em pseudocódigo usando o operador OU EXCLUSIVO e o operador *shift* para esquerda.

5. Escreva um pseudocódigo para uma soma global estruturada em árvore. Considere que o número de núcleos pode não ser uma potência de dois.

6. Elabore fórmulas para o número de recebimentos e adições que o núcleo 0 realiza usando

   (a) uma soma global onde apenas o núcleo 0 realiza as adições;

   (b) uma soma global estruturada em árvore.

   Faça uma tabela mostrando os números de recebimentos e adições realizados pelo núcleo 0 quando as duas somas são usadas com 2, 4, 8, $\cdots$, 1024 núcleos.

7. Descreva um problema de pesquisa em sua área (Tecnologia de Informação, Engenharia de Software...) que se beneficiaria com o uso da computação paralela. Forneça um esboço de como o paralelismo seria usado. Você usaria paralelismo de tarefas ou de dados? Por quê?

2 *Hardware* e *software* paralelos

8. Quando discutimos a adição de ponto flutuante, partimos da suposição simplificadora de que cada uma das unidades funcionais levava o mesmo tempo. Suponha que buscar (*fetch*) e armazenar (*store*) levem 2 nanossegundos cada e que as operações restantes levem 1 nanossegundo cada.

   (a) Quanto tempo leva uma adição de ponto flutuante com essas suposições?

   (b) Quanto tempo levará uma adição sem pipeline de 1.000 pares de *floats* com essas suposições?

   (c) Quanto tempo levará uma adição em pipeline de 1.000 pares de *floats* com essas suposições?

   (d) O tempo necessário para busca e armazenamento pode variar consideravelmente se os operandos/resultados forem armazenados em diferentes níveis da hierarquia de memória. Suponha que uma busca de um cache de nível 1 leve dois nanossegundos, enquanto uma busca de um cache de nível 2 leve cinco nanossegundos e uma busca da memória principal leve cinquenta nanossegundos. O que acontece com o pipeline quando há uma falha de cache de nível 1 na busca de um dos operandos? O que acontece quando há uma falha de nível 2?

9. Explique como uma fila implementada em hardware na CPU poderia ser usada para melhorar o desempenho de um *cache write-through*.

10. Dado o exemplo a seguir envolvendo leituras de cache de um array bidimensional.

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
// Initializa A e x; y = 0
. . .
// Primeiro par de loops/
for (i = 0; i < MAX ; i++)
  for (j = 0; j < MAX; j++)
    y [i] += A[i][j]x[j];
. . .
// y = 0
. . .
// Segundo par de loops
for (j = 0; j < MAX ; j++)
  for (i = 0; i < MAX; i++)
    y [i] += A[i][j]x[j];
```

   (a) Como o aumento do tamanho da matriz afetaria o desempenho dos dois pares de loops aninhados?

   (b) Como o aumento do tamanho da cache afetaria o desempenho dos dois pares de loops aninhados?

   (c) Quantas falhas ocorrem nas leituras de A no primeiro par de loops aninhados?

(d) Quantas falhas ocorrem no segundo par?

Suponha que uma linha de cache consiga armazenar quatro elementos de A.

11. Responda as questões abaixo. Explique o por quê de cada resposta.

    (a) A adição de cache e memória virtual a um sistema von Neumann altera sua designação como sistema SISD?

    (b) E quanto à adição de pipeline?

    (c) Multiple issue?

    (d) Hardware multithreading?

12. Suponha que um programa deva executar 10^{12} instruções para resolver um problema específico. Suponha ainda que um sistema de processador único possa resolver o problema em 10^6 segundos (cerca de $11,6$ dias). Portanto, em média, o sistema de processador único executa 10^6 instruções por segundo. Agora suponha que o programa tenha sido paralelizado para execução em um sistema de memória distribuída. Suponha também que o programa paralelo usa p processadores e, assim, cada processador executará $10^{12}/p$ instruções. Além disso, cada processador deverá enviar $10^9(p-1)$ mensagens. Por fim, suponha que não haja *overhead* adicional na execução do programa paralelo. Ou seja, o programa será concluído depois que cada processador tiver executado todas as suas instruções e enviado todas as suas mensagens e não haverá atrasos devido a coisas como espera por mensagens.

    (a) Suponha que demore 10^{-9} segundos para enviar uma mensagem. Quanto tempo levará para o programa ser executado com 1000 processadores se cada processador for tão rápido quanto o processador único no qual o programa serial foi executado?

    (b) Suponha que demore 10^{-3} segundos para enviar uma mensagem. Quanto tempo levará para o programa rodar com 1000 processadores?

13. (a) Suponha que um sistema de memória compartilhada use coerência de cache (*snooping*) e caches *write-back*. Suponha também que o núcleo 0 tenha a variável x em seu cache e execute a atribuição $x = 5$. Finalmente, suponha que o núcleo 1 não tenha x em seu cache e, após a atualização do núcleo 0 para x, o núcleo 1 tente executar $y = x$. Qual valor será atribuído a y? Por que?

    (b) Suponha que o sistema de memória compartilhada da parte anterior utilize um protocolo baseado em diretório. Qual valor será atribuído a y? Por que?

    (c) Como os problemas encontrados nas duas primeiras partes podem ser resolvidos?

14. Sendo n o tamanho do problema e p o número de *threads* (ou processos):

    (a) Suponha que o tempo de execução de um programa sequencial seja dado por $T_{sequencial} = n^2$, onde as unidades do tempo de execução estão em microssegundos. Suponha que uma paralelização deste programa tenha tempo de execução $T_{paralelo} = n^2/p + log_2(p)$. Escreva um programa que encontre as acelerações e eficiências deste programa para vários valores de n e p. Execute seu programa com $n = 10, 20, 40, \cdots, 320$ e $p = 1, 2, 4, \cdots, 128$.

- O que acontece com os *speedups* e eficiências à medida que p aumenta e n é mantido fixo?
- O que acontece quando p é fixo e n é aumentado?

(b) Suponha que $T_{paralelo} = T_{sequencial}/p + T_{overhead}$. Suponha também que mantemos p fixo e aumentamos o tamanho do problema.

- Mostre que, se $T_{overhead}$ crescer mais lentamente que $T_{sequencial}$, a eficiência paralela aumentará à medida que aumentarmos o tamanho do problema.
- Mostre que se, por outro lado, $T_{overhead}$ crescer mais rápido que $T_{sequencial}$, a eficiência paralela diminuirá à medida que aumentamos o tamanho do problema.

15. Às vezes, diz-se que um programa paralelo que obtém um *speedup* maior que p (o número de processos ou *threads*) tem *speedup* superlinear. No entanto, muitos autores não consideram os programas que superam as "limitações de recursos"como tendo aceleração superlinear. Por exemplo, um programa que deve usar armazenamento secundário para seus dados quando é executado em um sistema de processador único pode ser capaz de acomodar todos os seus dados na memória principal quando executado em um grande sistema de memória distribuída. Dê outro exemplo de como um programa pode superar uma limitação de recursos e obter *speedups* maiores que p.

16. Sendo n o tamanho do problema e p o número de *threads* (ou processos), suponha $T_{serial} = n$ e $T_{parallel} = n/p + log_2(p)$, onde os tempos estão em microssegundos. Se aumentarmos p por um fator de k, encontre uma fórmula para quanto precisaremos aumentar n para manter a eficiência constante.

    (a) Quanto devemos aumentar n se dobrarmos o número de processos de 8 para 16?

    (b) O programa paralelo é escalável?

17. Um programa que obtém *speedup* linear é fortemente escalável? Explique sua resposta.

18. Bob tem um programa que deseja cronometrar com dois conjuntos de dados, *input_data1* e *input_data2*. Para ter uma ideia do que esperar antes de adicionar funções de temporização ao código de seu interesse, ele executa o programa com os dois conjuntos de dados e o comando shell Unix *time*:

```
$ time ./bobs_prog < input_data1
real 0m0.001 s
user 0m0.001 s
sys  0m0.000 s

$ time ./bobs_prog < input_data2
real 1m1.234 s
user 1m0.001 s
sys  0m0.111 s
```

    A função de temporização que Bob está usando tem resolução de milissegundos.

    (a) Bob deveria usá-la para cronometrar seu programa com o primeiro conjunto de dados? Por que?

(b) E quanto ao segundo conjunto de dados? Por que?

19. Escreva um pseudocódigo para a soma global estruturada em árvore para somar vetores.

    (a) Primeiro considere como isso pode ser feito em um ambiente de memória compartilhada.

        • Quais variáveis são compartilhadas e quais são privadas?

    (b) Depois considere como isso pode ser feito em um ambiente de memória distribuída.

3 Programação de memória compartilhada com OpenMP

20. 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?

21. Baixe o arquivo `omp_trap_1.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:

      ```
      # barreira pragma omp
      ```

      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_trap_2.c` do site do livro. Como seu desempenho se compara?

    Explique suas respostas.

22. 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.

23. 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?

24. 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.

25. 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?

26. Lembre-se de que todos os blocos estruturados modificados por uma diretiva `critical` formam uma única secção 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`?

27. 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?

28. 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`?

29. 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.

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

3.1 Questões extra

31. 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.

32. *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`?

33. 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?

34. 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.

35. 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`.

4 Programação de memória distribuída com MPI

36. Modifique a regra trapezoidal para que ela estime corretamente a integral mesmo que `comm_sz` não divida n uniformemente Você ainda pode assumir que $n \geq comm\_sz$.

37. Modifique o programa que apenas imprime uma linha de saída de cada processo (`mpi_output.c`) para que a saída seja impressa na ordem de classificação do processo: processe a saída de 0 primeiro, depois processe 1 e assim por diante.

38. Suponha que um programa seja executado com `comm_sz` processos e que x seja um vetor com n componentes. Como os componentes de x seriam distribuídos entre os processos em um programa que usasse uma distribuição:

    (a) em bloco?

    (b) cíclica?

    (c) bloco-cíclica com tamanho de bloco b?

    Suas respostas devem ser genéricas para que possam ser usadas independentemente dos valores de `comm_sz` e n. Ao mesmo tempo, as distribuições apresentadas nas respostas devem ser "justas", de modo que, se q e r forem dois processos quaisquer, a diferença entre o número de componentes atribuídos a q e a r seja a menor possível.

39. Escreva um programa MPI que receba do usuário dois vetores e um escalar, todos lidos pelo processo 0 e distribuídos entre os processos. O primeiro vetor deve ser multiplicado pelo escalar. Para o segundo vetor, deve-se calcular o produto interno. Os resultados calculados devem ser coletados no processo 0, que os imprime. Você pode assumir que n, a ordem dos vetores, é divisível por `comm_sz`.

40. Encontrar somas de prefixos é uma generalização da soma global. Em vez de simplesmente encontrar a soma de n valores,

$$x_0 + x_1 + \cdots + x_{n-1}, \tag{2}$$

    as somas dos prefixos são as n somas parciais

$$x_0, x_0 + x_1, x_0 + x_1 + x_2, \cdots, x_0 + x_1 + \cdots + x_{n-1}. \tag{3}$$

    (a) Elabore um algoritmo serial para calcular as n somas de prefixos de um vetor com n elementos.

    (b) Paralelize seu algoritmo serial para um sistema com n processos, cada um armazenando um dos elementos de x.

    - Sem utilizar `MPI_Scan`

    (c) Suponha $n = 2^k$ para algum inteiro positivo k. Crie um algoritmo paralelo que exija apenas k fases de comunicação.

    - Sem utilizar `MPI_Scan`

(d) O MPI fornece uma função de comunicação coletiva, `MPI_Scan`, que pode ser usada para calcular somas de prefixos:

```
int MPI Scan(
void* sendbuf p /* in */,
void* recvbuf p /* out */,
int count /* in */,
MPI Datatype datatype /* in */,
MPI Op op /* in */,
MPI Comm comm /* in */);
```

Ela opera em arrays com `count` elementos; `sendbuf_p` e `recvbuf_p` devem se referir a blocos de `count` elementos do tipo `datatype`. O argumento `op` é igual ao `op` para o `MPI_Reduce`. Escreva um programa MPI que gere um vetor aleatório de `count` elementos em cada processo MPI, encontre as somas dos prefixos e imprima os resultados.

41. Uma alternativa para um `allreduce` estruturado em borboleta é uma estrutura de passagem em anel. Em uma passagem de anel, se houver p processos, cada processo q envia dados para o processo $q + 1$, exceto que o processo $p - 1$ envia dados para o processo 0. Isso é repetido até que cada processo tenha o resultado desejado. Assim, podemos implementar `allreduce` com o seguinte código:

```
sum = temp_val = my_val;
for (i = 1; i < p; i++) {
  MPI_Sendrecv_replace(&temp_val, 1, MPI INT, dest,
                       sendtag, source, recvtag, comm, &status);
  sum += temp_val;
}
```

(a) Escreva um programa MPI que implemente esse algoritmo para o `allreduce`. Como seu desempenho se compara ao `allreduce` estruturado em borboleta?

(b) Modifique o programa MPI que você escreveu na primeira parte para que ele implemente somas de prefixos.

42. As funções `MPI_Scatter` e `MPI_Gather` têm a limitação de que cada processo deve enviar ou receber o mesmo número de itens de dados. Quando este não for o caso, devemos utilizar as funções `MPI_Gatherv` e `MPI_Scatterv`. Consulte a documentação dessas funções e modifique seu programa da questão 39 para que ele possa lidar corretamente com o caso quando n não é divisível por `comm_sz`.

43. Suponha que $comm\_sz = 8$ e o vetor $x = (0, 1, 2, \cdots, 15)$ tenha sido distribuído entre os processos usando uma distribuição em bloco. Desenhe um diagrama ilustrando as etapas de uma implementação borboleta da função `allgather` de x.

44. A função `MPI_Type_contiguous` pode ser usada para construir um tipo de dados derivado de uma coleção de elementos contíguos em uma matriz. Sua sintaxe é

```
int MPI_Type_contiguous(
    int count /* in */,
    MPI Datatype old_mpi_t /* in */,
    MPI Datatype* new_mpi_t_p /* out *);
```

Modifique as funções `Read_vector` e `Print_vector` (`mpi_vector_add.c`) para que elas usem um tipo de dados MPI criado por uma chamada para `MPI_Type_contiguous` e um argumento de contagem de 1 nas chamadas para `MPI_Scatter` e `MPI_Gather`.

45. A função `MPI_Type_indexed` pode ser usada para construir um tipo de dados derivado de elementos arbitrários de um vetor. Sua sintaxe é

```
int MPI_Type_indexed(
    int count /* in */,
    int array_of_blocklengths[] /* in */,
    int array_of_displacements[] /* in */,
    MPI_Datatype old_mpi_t /* in */,
    MPI_Datatype* new_mpi_t_p /* out */);
```

Ao contrário da função `MPI_Type_create_struct`, os deslocamentos são medidos em unidades de `old_mpi_t` - não em bytes. Use a função `MPI_Type_indexed` para criar um tipo de dados derivado que corresponda à parte triangular superior de uma matriz quadrada. Por exemplo, na matriz 4 x 4

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

a parte triangular superior são os elementos 0, 1, 2, 3, 5, 6, 7, 10, 11, 15. O processo 0 deve ler uma matriz $n \times n$ como um vetor unidimensional, criar o tipo de dados derivado e enviar a parte triangular superior com uma única chamada de `MPI_Send`. O processo 1 deve receber a parte triangular superior com uma única chamada ao `MPI_Recv` e depois imprimir os dados recebidos.

46. As funções `MPI_Pack` e `MPI_Unpack` fornecem uma alternativa aos tipos de dados derivados para agrupar dados. O `MPI_Pack` copia os dados a serem enviados, um bloco por vez, em um *buffer* fornecido pelo usuário. O *buffer* pode então ser enviado e recebido. Após o recebimento dos dados, `MPI_Unpack` pode ser usado para descompactá-los do *buffer* de recebimento. A sintaxe do `MPI_Pack` é

```
int MPI_Pack(
    void* in_buf /* in */,
    int in_buf_count /* in */,
    MPI_Datatype datatype /* in */,
    void* pack_buf /* out */,
    int pack_buf_sz /* in */,
    int* position_p /* in/out */,
    MPI_Comm comm /* in */);
```

Poderíamos, portanto, empacotar os dados de entrada para o programa da regra dos trapézios com o seguinte código:

```
char pack_buf[100];
int position = 0;
MPI_Pack(&a, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&b, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&n, 1, MPI_INT, pack_buf, 100, &position, comm);
```

A chave é o argumento da `position`. Quando `MPI_Pack` é chamado, a posição deve referir-se ao primeiro slot disponível no `pack_buf`. Quando `MPI_Pack` retorna, ele se refere ao primeiro slot disponível após os dados que acabaram de ser compactados, portanto, após o processo 0 executar este código, todos os processos podem chamar `MPI_Bcast`:

```
MPI_Bcast(pack_buf, 100, MPI_PACKED, 0, comm);
```

Observe que o tipo de dados MPI para um *buffer* compactado é `MPI_PACKED`. Agora os outros processos podem descompactar os dados usando: `MPI_Unpack`:

```
int MPI Unpack(
    void* pack_buf /* in */,
    int pack_buf_sz /* in */,
    int* position_p /* in/out */,
    void* out_buf /* out */,
    int out_buf_count /* in */,
    MPI_Datatype datatype /* in */,
    MPI_Comm comm /* in */);
```

`MPI_Unpack` pode ser usado "invertendo"as etapas do `MPI_Pack`, ou seja, os dados são descompactados um bloco por vez, começando em $position = 0$.

Escreva outra função `Get_input` para o programa da regra dos trapézios. Este deve usar `MPI_Pack` no processo 0 e `MPI_Unpack` nos demais processos.

47. Cronometre a implementação do livro da regra dos trapézios que usa `MPI_Reduce` para diferentes números de trapézios e processos,n e p, respectivamente. Lembre-se de medir o tempo de execução ao menos 5 vezes para cada par (n, p).

    (a) Qual critério você utilizou para escolher n?

    (b) Como os tempos mínimos se comparam aos tempos médios e medianos?

    (c) Quais são os *speedups*?

    (d) Quais são as eficiências?

    (e) Com base nos dados que você coletou, você diria que a regra dos trapézios é escalável?

48. Embora não conheçamos os detalhes da implementação do `MPI_Reduce`, podemos supor que ele usa uma estrutura semelhante à árvore binária que discutimos. Se for esse o caso, esperaríamos que seu tempo de execução crescesse aproximadamente à taxa de $\log_2(p)$ ($p = comm\_sz$), uma vez que existem aproximadamente $\log_2(p)$ níveis na árvore. Como o tempo de execução da regra dos trapézios serial é aproximadamente proporcional a n, o número de trapézios, e a regra dos trapézios paralela simplesmente aplica a regra serial a n/p trapézios em cada processo, com nossa suposição sobre `MPI_Reduce`, obtemos uma fórmula para o tempo de execução geral da regra dos trapézios paralela que se parece com

$$T_{parallel}(n, p) \approx a \times \frac{n}{p} + b \log_2(p) \tag{4}$$

onde a e b são constantes.

Use a fórmula, os tempos que você mediu no Exercício 47 e seu programa favorito para fazer cálculos matemáticos (por exemplo, o MATLAB®) para obter uma estimativa de mínimos quadrados dos valores de a e b. Comente sobre a qualidade dos tempos de execução previstos usando a fórmula.

49. Encontre os *speedups* e as eficiências da ordenação ímpar-par paralela (`mpi_odd_even.c`).

    (a) O programa obtém *speedups* lineares?

    (b) É escalável?

    (c) É fortemente escalável?

    (d) É fracamente escalável?

50. Modifique a ordenação ímpar-par paralela (`mpi_odd_even.c`) para que as funções `Merge` simplesmente troquem os ponteiros do vetor após encontrar os elementos menores ou maiores. Que efeito essa mudança tem no tempo de execução geral?

5 Referência

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