Programação Concorrente (ICP-361)
Cap. 2: Construindo algoritmos concorrentes*

Prof. Silvana Rossetto

[1]Instituto de Computação (IC)
Universidade Federal do Rio de Janeiro (UFRJ)
Agosto de 2023
silvana@ic.ufrj.br

1. Introdução

Tipicamente aprendemos a programar desenvolvendo algoritmos que modelam uma única sequência de passos consecutivos (algoritmo sequencial). Em alguns casos, essa ordem de execução é estritamente necessária, ou seja, não podemos executar o passo 2 junto com o passo 1 ou não podemos executar o passo 5 antes de termianr o passo 3. Entretanto, há vários casos em que a execução de dois ou mais passos pode ocorrer ao mesmo tempo, ou seja, não existe dependências entre eles. Nesse caso, podemos transformar nosso algoritmo sequencial em um **algoritmo concorrente**, definindo duas ou mais sequências consecutivas de passos que podem ser executadas ao mesmo tempo dentro do programa, ou começar a executar um outro passo sem precisar esperar que o anterior termine. Dessa forma teremos um **programa concorrente**.

Para ilustrar essa ideia, considere que temos uma médica que precisa atender os 5 pacientes que estão na sala de espera. O algoritmo para a médica realizar essa tarefa é atender um paciente de cada vez, de acordo com a ordem de chegada. Esse é um exemplo de uma sequência consecutiva de passos (algoritmo sequencial) para executar uma tarefa determinada. Entretanto, se outra médica for alocada para esse atendimento também, podemos dividir a tarefa entre elas, determinando que cada médica atenderá o próximo paciente da fila. Isso é possível porque não existe dependência em relação à ordem em que os pacientes são atendidos e não há restrição em atender dois pacientes ao mesmo tempo, em salas distintas. Se todos os atendimentos demandarem o mesmo tempo médio, o tempo total para realizar todos os atendimentos será aproximadamente a metade do tempo requerido com uma médica.

A programação concorrente engloba o uso de técnicas para dividir uma tarefa em subtarefas menores, e de mecanismos para coordenar a execução dessas subtarefas. Dependendo das características do problema, e do ambiente de execução concorrente (por exemplo, CPU multinúcleo, sistemas distribuídos ou aceleradores como GPUs), o programador deve selecionar as alternativas mais adequadas.

Neste capítulo descrevemos em que consiste escrever um programa concorrente e apresentamos alguns exemplos iniciais. Em seguida, apresentamos métricas de desempenho das aplicações, as quais são usadas para avaliar os ganhos obtidos em relação ao código sequencial e o grau de escalabilidade dos algoritmos concorrentes, ou seja, como o algoritmo se comporta com o aumento da carga de trabalho e das unidades de processamento disponíveis.

*Texto baseado no livro "Introdução à programação paralela em GPU para a implementação de métodos numéricos", D. Alfaro e S. Rossetto, Notas em Matemática Aplicada, vol 84, 2016, disponível em: https://www.sbmac.org.br/wp-content/uploads/2022/08/livro_84.pdf

2. Exemplos de algoritmos concorrentes

Para introduzir a programação concorrente, vamos começar tomando como referência dois exemplos de problemas que podem ser implementados de forma concorrente: **soma de vetores** e **multiplicação de matrizes**.

Soma de vetores Considere a operação SAXPY (*Single-Precision A·X Plus Y*) [1] que consiste em calcular `C = A * k + B`, sendo A, B e C vetores de tamanho N e k um número. Essa é uma operação que aparece com frequência em problemas de álgebra linear computacional.

Para encontrar o vetor de saída C, precisamos calcular cada elemento desse vetor fazendo: `C[i] = A[i] * k + B[i], 0<=i<N.` Na programação sequencial, executamos a sentença acima uma vez para cada posição do vetor C, como ilustrado no programa abaixo (escrito na linguagem C).

```
#define N 1000000 //N igual ao número de elementos do vetor
//calcula C = k * A + B
void somaVetoresSeq(const float a[], const float b[],
  float c[], float k, int n) {
   int i;
   for(i=0; i<n; i++) {
      c[i] = a[i] * k  + b[i];
   }
}
void main() {
   float a[N], b[N], c[N];
   //inicializa os vetores a e b
   ...
   somaVetoresSeq(a, b, c, 2.0, N);
}
```

Esse programa será executado por um único processador que calculará os elementos do vetor de saída C na ordem crescente das suas posições.

Para escrever um algoritmo concorrente que calcule o vetor `C = A * k + B`, vamos pensar também em termos da menor tarefa do problema, que é o cálculo de uma posição do vetor de saída. Como cada posição do vetor é independente das demais, podemos computar todos os valores ao mesmo tempo, desde que tenhamos um número de unidades de processamento igual ao número de elementos do vetor. Nesse caso, ao invés de usarmos o comando de repetição (*for*), definimos uma função que calcula o valor de um único elemento e disparamos N fluxos de execução distintos que executarão essa função uma única vez, informando a posição que deverá ser calculada.

No código abaixo, mostramos como implementar esse algoritmo. Vamos assumir por ora que na linha 15 usaremos funções especiais providas pela linguagem C para criar e disparar fluxos independentes de execução. O que fizemos foi reduzir a complexidade de processamento de $O(N)$ para $O(1)$, assumindo que N fluxos de execução poderão estar ativos ao mesmo tempo. Entretanto, nem sempre dispomos de tantas unidades de processamento simultâneas (como ocorre com as CPUs). Nesse caso, podemos variar o tamanho da tarefa processada por cada fluxo de execução, por exemplo, atribuindo para cada um deles o cálculo de um subconjunto de elementos do vetor de saída.

```
#define N 1000000 //N igual ao número de elementos do vetor

//calcula C[i] = k * A[i] + B[i]
void calculaElementoVetor(const float a[], const float b[],
  float c[], float k, int pos) {
   c[pos] = k * a[pos] + b[pos];
}

void main() {
   float a[N], b[N], c[N];
   int i;
   //inicializa os vetores a e b
   ...
   //faz C = k * A + B
   for(i=0; i<N; i++) {
      //dispara um fluxo de execução f para executar:
      //calculaElementoVetor(a, b, c, 2.0, i)
   }
}
```

Assumindo que vamos criar um fluxo de execução paralela para cada unidade de processamento da máquina, uma alternativa é dividir o número total de elementos do vetor (N) pelo número de unidades de processamento (P), e deixar para a última unidade o cálculo dos elementos adicionais (no caso da divisão anterior não ser exata), como ilustrado no programa abaixo.

```
#define N 1000000 //N igual ao número de elementos do vetor
#define P 4 //P igual ao número de unidades de processamento

//calcula uma parte de C = k * A + B
void calculaElementosConsVetor(const float a[], const float b[],
  float c[], float k, int inicio, int fim) {
   int i;
   for(i=inicio; i<fim; i++) {
      c[i] = k * a[i] + b[i];
   }
}

void main() {
   float a[N], b[N], c[N];
   int i, inicio, fim;
   //inicializa os vetores a e b
   ...
   //faz C = k * A + B
   for(i=0; i<P; i++) {
      inicio = i * (N/P);
      //o último fluxo trata os elementos restantes
      if (i<P-1) {
         fim = inicio + (N/P);
      } else {
         fim = N;
      }
      //dispara um fluxo de execução f para executar:
      //calculaElementosConsVetor(a, b, c, 2.0, inicio, fim);
   }
}
```

Outra alternativa para dividir a tarefa completa (N elementos) entre as unidades de processamento (P) é alternar o elemento que deve ser calculado por cada unidade. Nesse caso, não é necessário lidar explicitamente com o caso do número de elementos não ser divisível pelo número de unidades de processamento pois os elementos restantes serão automaticamente divididos entre as unidades de processamento.

O programa seguinte mostra como implementar essa alternativa. Cada fluxo de execução deverá usar como índice inicial o seu identificador único. A cada passo do algoritmo salta-se P elementos. Em todos os fluxos, a repetição termina quando o valor do índice alcança o valor N.

```
#define N 1000000 //N igual ao número de elementos do vetor
#define P 4 //P igual ao número de unidades de processamento
//calcula uma parte de C = k * A + B
void calculaElementosInterVetor(const float a[], const float b[],
 float c[], float k, int inicio, int salto, int n) {
   int i;
   for(i=inicio; i<n; i=i+salto) {
      c[i] = k * a[i]  + b[i];
   }
}
void main() {
   float a[N], b[N], c[N];
   int i;
   //inicializa os vetores a e b
   ...
   //faz C = k * A + B
   for(i=0; i<P; i++) {
      //dispara um fluxo de execução f para executar:
      //calculaElementosInterVetor(a, b, c, 2.0, i, P, N);
   }
}
```

Uma preocupação importante quando projetamos um algoritmo concorrente é garantir que todas as unidades de processamento recebam a mesma carga de trabalho para aproveitar ao máximo todos os recursos de processamento disponíveis. Se uma unidade termina sua execução muito antes das demais, esse recurso ficará ocioso e não será aproveitado pela aplicação. Todas as alternativas de paralelização mostradas acima garantem o balanceamento de carga entre as unidades de processamento. Entretanto, o padrão de acesso à memória varia de uma alternativa para a outra, o que poderá fazer com que o desempenho de cada uma delas — em termos de tempo de acesso à memória — varie de forma significativa.

Multiplicação de matrizes Consideremos agora o problema de multiplicação de matrizes. Dadas duas matrizes de entrada, A e B, queremos calcular a matriz `C = A * B`. A parte central do algoritmo sequencial para resolver esse problema está implementada no código abaixo. Por simplicidade, assume-se que as matrizes são quadradas de dimensão $N \times N$.

```
...
for(i=0; i<N; i++) {
   for(j=0; j<N; j++) {
      soma = 0.0;
      for(k=0; k<N; k++) {
         soma = soma + a[i][k] * b[k][j];
      }
      c[i][j] = soma;
   }
}
...
```

De que forma podemos paralelizar esse algoritmo? Uma estratégia é isolarmos o cálculo de cada elemento da matriz de saída (linhas 4 a 7 do código) e tratar esse subproblema como um tarefa básica que poderá ser executada por fluxos de execução paralelos. O cálculo de cada elemento da matriz de saída consiste em uma sequência de operações de multiplicação e soma, completamente independente do cálculo dos demais elementos. Para calcular o elemento (i, j) da matriz de saída C, precisamos apenas dos elementos da linha i da matriz A e dos elementos da coluna j da matriz B. O resultado da computação é armazenado em uma posição única da matriz de saída. Por exemplo, o valor do elemento $C_{(1,1)}$ da matriz de saída será computado da seguinte forma: $C_{(1,1)} = A_{(1,0)} * B_{(0,1)} + A_{(1,1)} * B_{(1,1)} + A_{(1,2)} * B_{(2,1)}$

A linha i da matriz A também é usada para calcular outros elementos da matriz C (por exemplo, os elementos $C_{(i,j-1)}$ e $C_{(i,j+1)}$), assim como a coluna j da matriz B é usada mais de uma vez para calcular elementos diferentes da matriz de saída. Essa reutilização de valores de entrada (que não ocorria no exemplo da soma de vetores) não afeta a paralelização do problema, pois dois fluxos de execução podem ler a mesma variável de entrada ao mesmo tempo.

O programa seguinte mostra o pseudo-código de um algoritmo paralelo para multiplicar duas matrizes quadradas. Por simplicidade, deixamos as matrizes como variáveis globais. Nos exercícios no final deste capítulo discutiremos outras alternativas de paralelização desse problema.

```
#define N 1000 //N igual à dimensão da matriz
float a[N][N], b[N][N], c[N][N];
void calculaElementoMatriz(int dim, int i, int j) {
   int k, soma = 0.0;
   for(k=0; k<dim; k++) {
      soma = soma + a[i][k] * b[k][j];
   }
   c[i][j] = soma;
}
void main() {
   int i, j;
   //inicializa as matrizes a e b (...)
   //faz C = A * B
   for(i=0; i<N; i++) {
      for(j=0; j<N; j++) {
         //dispara um fluxo de execução f para executar:
 //calculaElementoMatriz(N, i, j);
      }
   } }
```

3. Tipos de problemas concorrentes e etapas de desenvolvimento de algoritmos concorrentes

De acordo com [2], um problema P_D com domínio D é dito paralelizável se for possível decompor P_D em k subproblemas:

$$D = d_1 \oplus d_2 \oplus \cdots \oplus d_k = \sum_{i=1}^{k} d_i \tag{1}$$

Dependendo das características do problema, há diferentes formas de realizar essa decomposição.

Ainda de acordo com a definição de Navarro et al. [2], dizemos que o problema P_D é um problema com **paralelismo de dados** se D é composto de elementos de dados e é possível aplicar uma função $f(\cdots)$ para todo o domínio:

$$f(D) = f(d_1) \oplus f(d_2) \oplus \cdots \oplus f(d_k) = \sum_{i=1}^{k} f(d_i) \tag{2}$$

Os problemas apresentados na seção anterior são exemplos de problemas com paralelismo de dados.

Por outro lado, se D é composto por funções e a solução do problema consiste em aplicar cada função sobre um fluxo de dados comum S, dizemos que o problema P_D é um problema com **paralelismo de tarefas**:

$$D(S) = d_1(S) \oplus d_2(S) \oplus \cdots \oplus d_k(S) = \sum_{i=1}^{k} d_i(S) \tag{3}$$

Projetar e implementar algoritmos concorrentes não é uma tarefa simples e não há uma regra geral para desenvolver algoritmos concorrentes perfeitos. Foster [3] sugere um método que se divide em quatro etapas: particionamento, comunicação, aglomeração e mapeamento.

1. **Particionamento**: a tarefa que deve ser executada e o conjunto de dados associado a ela são decompostos em tarefas menores. Nessa etapa, o objetivo é identificar oportunidades de execução paralela. Para isso, caracterizar o domínio do problema (com "paralelismo de dados" ou "paralelismo de tarefas") é fundamental para escolher uma boa estratégia de particionamento.
2. **Comunicação**: nessa etapa, determina-se a comunicação requerida para coordenar a execução da tarefa e define-se as estruturas e algoritmos de comunicação mais apropriados.
3. **Aglomeração**: as subtarefas e estruturas de comunicação definidas nas etapas anteriores são avaliadas com respeito aos requisitos de desempenho e custos de implementação e, se necessário, as subtarefas são combinadas em tarefas maiores para melhorar o desempenho ou reduzir os custos de desenvolvimento.
4. **Mapeamento**: nessa última etapa, cada tarefa é designada para uma unidade de processamento de uma forma que satisfaça a meta de maximizar o uso da capacidade de processamento paralela disponível e minimizar os custos de comunicação

e gerência. O mapeamento pode ser feito de forma estática ou determinado em tempo de execução. Para essa etapa é fundamental conhecer as características principais da arquitetura alvo.

Além das questões e desafios relacionados em como "pensar" de forma concorrente para projetar algoritmos com vários fluxos de execução independentes, a programação concorrente requer que se leve em conta também detalhes sobre a arquitetura dos dispositivos de processamento. Em cada caso (multiprocessadores, aceleradores, sistemas distribuídos) é preciso compreender quais são os padrões de acesso à memória, como os fluxos são escalonadas para execução, entre outros, para implementar os algoritmos de forma a extrair o máximo de desempenho do sistema utilizado.

4. Métricas de desempenho

As métricas de desempenho são usadas para quantificar a qualidade de um algoritmo. Os algoritmos sequenciais são normalmente avaliados e comparados por meio de duas métricas: tempo de execução e espaço de memória requerido. Para os algoritmos concorrentes, métricas adicionais são normalmente usadas, entre elas **aceleração** e **eficiência**. Nesta seção, apresentaremos essas métricas.

4.1. Tempo de execução

Dado um problema de tamanho n, vamos denotar por $T_p(n, p)$ o tempo de execução de um algoritmo concorrente usando p processadores e por $T_s(n)$ o tempo de execução de um algoritmo sequencial. Medimos o **tempo de execução** (T) de um programa, ou de parte dele, tomando o instante de tempo inicial (imediatamente antes do início do trecho do programa que queremos avaliar) e o instante de tempo final (imediatamente após a execução do trecho de código avaliado). Em seguida, calculamos o intervalo de tempo decorrido.

Para que essa tomada de tempo seja realizada de forma correta, precisamos usar funções que meçam o tempo de forma contínua, isto é, sem a interferência de alterações ou ajustes no relógio do sistema. Em ambientes Linux, por exemplo, podemos calcular o tempo de execução de um trecho de um programa escrito em C usando a função `clock_gettime`, definida na biblioteca `time.h`. Essa função retorna uma estrutura de dados que contém dois campos: um contador de segundos e um contador de nanosegundos cujos valores são preenchidos com o tempo transcorrido desde uma data predefinida.

O código abaixo mostra um exemplo de uso da função `clock_gettime` para calcular o tempo gasto para executar um determinado trecho do programa. A função auxiliar `tempoAtual` chama a função `clock_gettime` e em seguida processa a estrutura de dados retornada para gerar uma medida de tempo em segundos.

```
#include <time.h>
//retorna o instante de tempo atual em segundos
double tempoAtual() {
   struct timespec tempo;
   clock_gettime(CLOCK_MONOTONIC_RAW, &tempo);
   return tempo.tv_sec + tempo.tv_nsec/1000000000.0;
}
```

```
void main() {
   double inicio, fim, delta;
   inicio = tempoAtual();
   //...trecho do programa que queremos medir o tempo
   fim = tempoAtual();
   //tempo consumido para executar o trecho do programa
   delta = fim - inicio;
}
```

Normalmente, quando já dispomos de uma implementação sequencial do problema e queremos paralelizá-la, o primeiro passo consiste em identificar quais partes do programa demandam maior tempo de execução e então avaliar as possibilidades de paralelização dessas partes.

4.2. Aceleração

Dado um problema de tamanho n, a **aceleração** (A) (do termo em inglês *speedup*) é a razão entre o tempo de execução da melhor versão sequencial do algoritmo ($T_s(n)$) e o tempo de execução da versão paralela ($T_p(n,p)$) usando p processadores:

$$A(n,p) = T_s(n)/T_p(n,p) \tag{4}$$

Caso a implementação sequencial do algoritmo não esteja disponível, calculamos a aceleração substituindo o tempo do algoritmo sequencial ($T_s(n)$) pelo tempo do algoritmo paralelo usando um único processador ($T_p(n,1)$). Nesse caso temos:

$$A(n,p) = T_p(n,1)/T_p(n,p) \tag{5}$$

Quando paralelizamos um problema, o ideal seria conseguir dividir a tarefa igualmente entre os fluxos de execução paralela sem adicionar carga de trabalho extra. Nesse caso, teríamos $T_p(n,p) = T_s(n)/p$ e a aceleração seria **perfeitamente linear**, ou seja, com p processadores o algoritmo concorrente executaria p vezes mais rápido que o algoritmo sequencial. Teoricamente esse é o valor máximo de aceleração de um algoritmo concorrente.

Entretanto, há vários fatores que impossibilitam uma curva de aceleração perfeitamente linear: o algoritmo concorrente requer uma carga de processamento adicional, necessária para gerenciar a execução dos fluxos de execução; nem sempre é possível paralelizar o problema inteiro ou dividir a tarefa igualmente entre os fluxos de execução; a complexidade de gerência dos fluxos de execução e a competição pelo acesso à memória pode crescer com o aumento do número de processadores. Nesses casos, a curva da aceleração é dita **sublinear** e o principal desafio para os programadores é tentar minimizar a degradação de desempenho causada por esses fatores.

Há ainda situações em que a curva da aceleração ultrapassa o seu limite teórico, sendo chamada de **superlinear**. Diferentes fatores relacionados com o hardware da máquina podem levar a essa situação, entre eles o seu modelo de organização e acesso à memória. Pode ocorrer, por exemplo, do volume de dados acessado pela aplicação sequencial ultrapassar o tamanho da memória cache do processador, ocasionando uma baixa taxa de acerto de cache. Ao particionar os dados da aplicação entre diferentes fluxos de execução que são alocados para diferentes processadores, a cache de cada processador

pode agora ser suficiente para a quantidade de dados manipulados, permitindo uma alta taxa de acerto da cache. Outro exemplo são as aplicações que fazem acesso a dados no disco (memória persistente) e não dispõem de espaço suficiente na memória RAM (memória temporária) para realizar a carga completa dos dados. Dividindo-se a execução da aplicação em várias máquinas torna-se possível carregar todos os dados de uma vez para a memória RAM de cada máquina, reduzindo os custos do acesso ao disco [4].

4.3. Eficiência

A medida de **eficiência** (E) é usada para mostrar como a aceleração varia com o aumento do número de processadores, e é dada pela seguinte equação:

$$E(n, p) = A(n, p)/p$$

A é a aceleração, p é o número de processadores e n é o tamanho da entrada.

Essa métrica permite avaliar, em particular, como a carga de trabalho extra do algoritmo concorrente cresce quando o número de processadores aumenta. Quando a aceleração é linear, temos eficiência igual a 1 (valor ideal). Como normalmente a aceleração é menor que o número de unidades de processamento, o valor da eficiência fica abaixo de 1, sendo desejável se buscar o valor mais próximo de 1 possível.

5. Lei de Amdahl

Em algumas situações, quando já temos alguma informação sobre o tempo de execução de uma aplicação sequencial, queremos estimar qual será o ganho de desempenho de uma possível versão concorrente dessa aplicação. Assumindo que o tempo para um processador executar um dado programa seja de 1 unidade de tempo e que p seja a fração do programa que pode ser executada em paralelo, a *Lei de Amdahl* [5] estima a aceleração teórica (A_t) que poderá ser obtida por uma versão concorrente desse programa usando n processadores da seguinte forma:

$$A_t = \frac{1}{(1-p) + (p/n)}$$

Esse tipo de análise é muito importante para a computação concorrente. O ganho de tempo na execução de uma tarefa complexa é limitado pela quantidade de código que deve continuar sendo executado sequencialmente. Por outro lado, para alguns problemas, o percentual da quantidade de código que executa sequencialmente diminui com o crescimento da entrada. Por exemplo, em um problema de multiplicação de matrizes, o percentual de tempo necessário para carregar as matrizes de entrada, comparado com o tempo de processamento dessas matrizes, diminui quando a dimensão das matrizes aumenta.

Exercício

1. No programa de multiplicação de matrizes mostramos uma forma de paralelizar o algoritmo de multiplicação de matrizes criando um fluxo de execução independente para calcular cada um dos elementos da matriz de saída. Proponha outra solução onde a tarefa de cada fluxo de execução seja calcular uma linha inteira da matriz de saída.

2. Para arquiteturas de hardware com poucas unidades de processamento (como é o caso das CPUs *multicores*) geralmente é melhor criar uma quantidade de fluxos de execução igual ao número de unidades de processamento. Altere a solução do exercício anterior fixando o número de fluxos de execução e dividindo o cálculo das linhas da matriz de saída entre eles.
3. A série mostrada abaixo pode ser usada para estimar o valor da constante π. A função `piSequencial()` implementa o cálculo dessa série de forma sequencial. Proponha um algoritmo paralelo para resolver esse problema dividindo a tarefa de estimar o valor de π entre M fluxos de execução independentes.

$$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots)$$

```
double piSequencial (long long n) {
   double soma = 0.0, fator = 1.0;
   long long i;
   for (i = 0; i < n; i++) {
      soma = soma + fator/(2*i+1);
      fator = -fator;
   }
   return 4.0 * soma;
}
```

4. A série infinita mostrada abaixo estima o valor de `log(1+x)` ($-1 < x <= 1$):

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots$$

Dois programas foram implementados para calcular o valor dessa série (um programa sequencial e outro paralelo) usando N termos. Após a implementação, foram realizadas execuções dos dois programas, obtendo as medidas de tempo apresentadas na Tabela 1. A coluna N informa o número de elementos da série, a

N	p	T_s (s)	T_p (s)	A
1×10^6	1	0,88	0,89	
1×10^6	2	0,88	0,50	
1×10^7	1	8,11	8,34	
1×10^7	2	8,11	4,44	
2×10^7	1	16,21	16,41	
2×10^7	2	16,21	8,84	

Table 1. Medidas de tempo para calcular log(1+x).

coluna p informa o número de processadores, e as colunas T_s e T_p informam os tempos de execução do programa sequencial e do programa paralelo, respectivamente.

(a) Complete a coluna A com os valores de aceleração.
(b) Avalie os resultados obtidos para essa métrica. Considere os casos em que a carga de dados aumenta junto com o número de processadores e os casos isolados onde apenas a carga de trabalho ou o número de processadores aumenta.

5. Considere uma aplicação na qual 20% do tempo total de execução é comprometido com tarefas sequenciais e o restante, 80%, pode ser executado de forma concorrente.
   (a) Se dispusermos de uma máquina com 4 processadores, qual será a aceleração teórica (de acordo com a lei de Amdahl) que poderá ser alcançada em uma versão concorrente aplicação?
   (b) Se apenas 50% atividades pudessem ser executadas em paralelo, qual seria a aceleração teórica considerando novamente uma máquina com 4 processadores?

References

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 4 edition, 2013.

[2] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, 15:285–329, 2014.

[3] Ian Foster. *Designing and building parallel programs : concepts and tools for parallel software engineering*. Addison-Wesley, Boston, MA, USA, 1 edition, 1995.

[4] Gabriel P. Silva, Calebe P. Bianchini, and Evaldo B. Costa. *Programação Paralela e Distribuída com MPI, OpenMP e OpenACC para computação de alto desempenho*. Casa do Código, 2022.

[5] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.