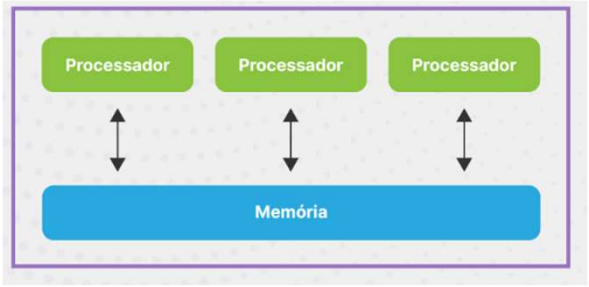
Capítulo 3

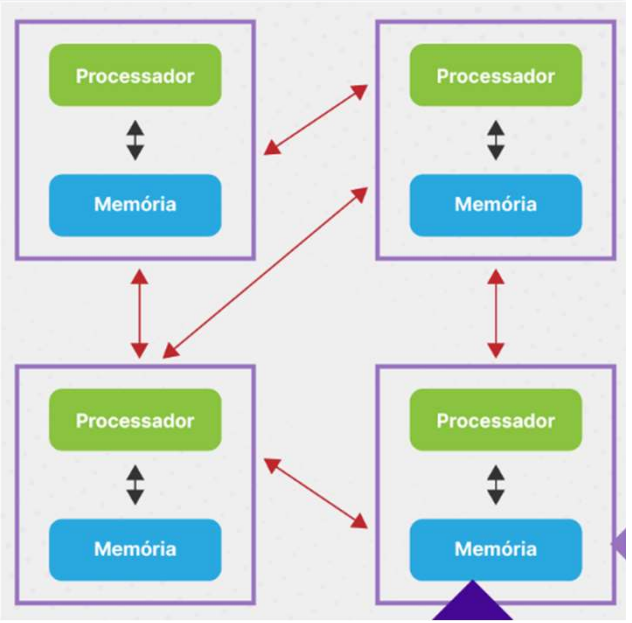
Processamento paralelo, Arquiteturas paralelas

[e ... Multithreading]

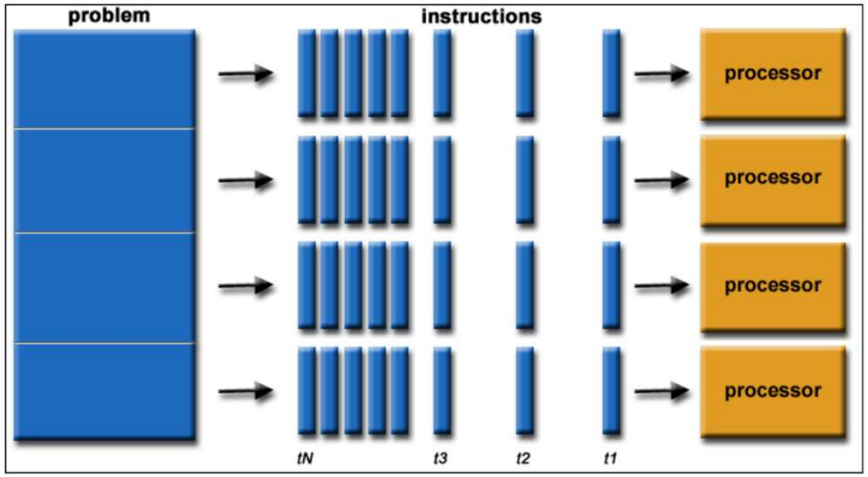
Introdução: visão geral

- Paralelismo tem duas arquiteturas
  - Multicore x Sistemas distribuídos

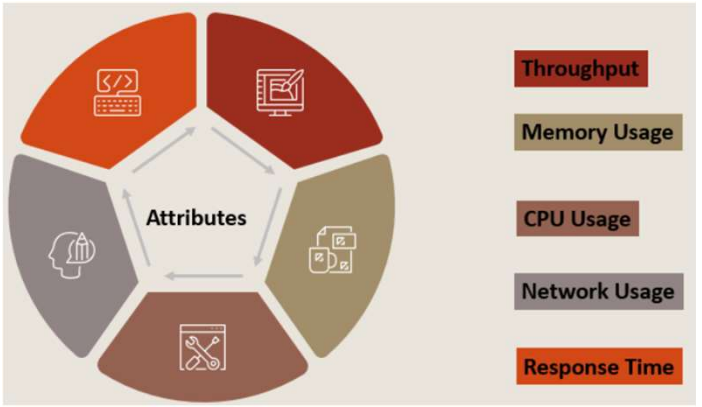




  - Visão do paralelismo local:



  - Questão a discutir:
    - paralelizar algoritmo
    - paralelizar código
    - Paralelismo x STRs



Terminologias

- **Speedup** ([1], pág. 2)**:**
  - quociente do tempo gasto usando um único processador (T (1)) ao relação ao mesmo código sendo executado em p processadores/core e gastando T (p)

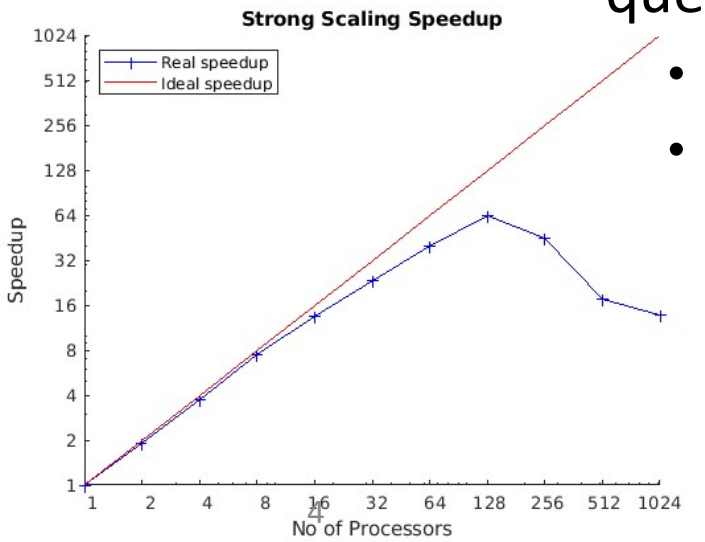
$$S = \frac{T(1)}{T(p)}$$

- **Eficiência e custo** (*super-linear speedups* - [1], pág. 2)**:**
  - Associando S ao número de processadores

$$E = \frac{S}{p} = \frac{T(1)}{T(p) \times p}$$

Terminologias

- **Escalabilidade** ([1], pág. 3)**:**
  - Associando S ao número de processadores
  - Eficiência determinado por número de processadores
  - Escalabilidade é variável o número de processadores (p = 1, 2, 4, 8, 16, 32, etc)
  - Indica o comportamento de um programa paralelo quando o número de processadores aumenta.
  - O tamanho dos dados de entrada é outro parâmetro que pode querer variar:
    - Weak = entrada varia (duplicar p duplica tamanho entrada)
    - Strong = entrada constante



Terminologias

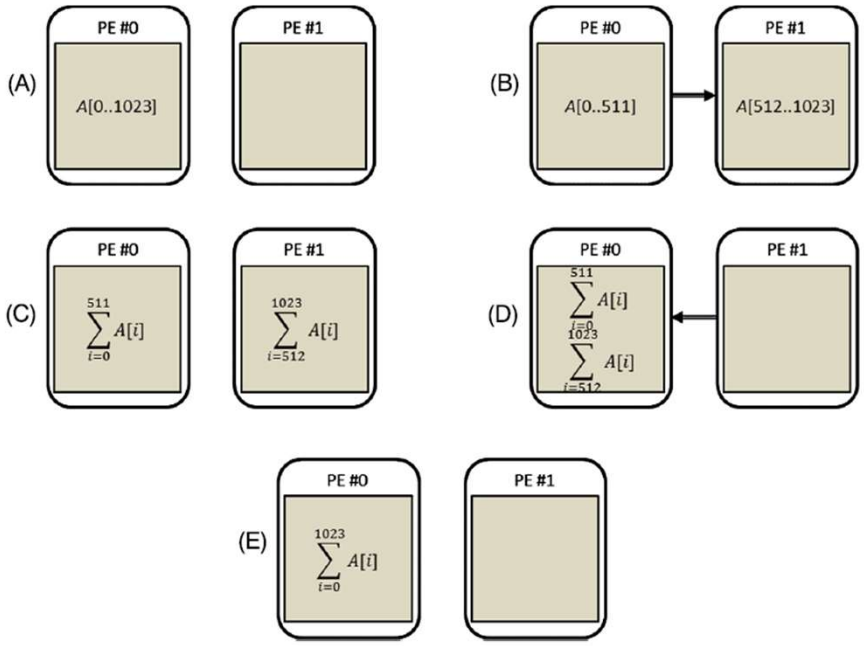
- **PE – processing elements:**
  - Número de processadores ou núcleos ou elementos aptos a processar no seu sistema.

- **Coeficinte computation-to-communication** ([1], pág. 3)**:**
  - tempo gasto calculando dividido pelo tempo gasto na comunicação de mensagens entre processadores.
  - Exemplo:

> The example we now want to look at is a simple *summation*; i.e., given an array A of n numbers we want to compute $\sum_{i=0}^{n-1} A[i]$. We parallelize this problem using an array of *processing elements* (PEs). We make the following (not necessarily realistic) assumptions:
>
> - **Computation.** Each PE can add two numbers stored in its local memory in one time unit.
> - **Communication.** A PE can send data from its local memory to the local memory of any other PE in three time units (independent of the size of the data).
> - **Input and output.** At the beginning of the program the whole input array A is stored in PE #0. At the end the result should be gathered in PE #0.
> - **Synchronization.** All PEs operate in lock-step manner; i.e. they can either compute, communicate, or be idle. Thus, it is not possible to overlap computation and communication on this architecture.

Métricas

- **Exemplo 1**: somatório de n = 1024 números em p = 2 PEs



A. inicialmente o PE #0 armazena todos os dados de entrada localmente;
B. PE #0envia metade da entrada para PE #1 (leva tempo 3);
C. Cada PE soma seus 512 números (leva o tempo 511)
D. PE #1 envia sua soma parcial de volta para PE #0 (leva tempo 3);
E. Para finalizar o cálculo, PE #0 adiciona as duas somas parciais (leva 1 tempo)

Assim, o tempo de execução total é T (2, 1024) = 3+511 +3+1 = 518.

Métricas

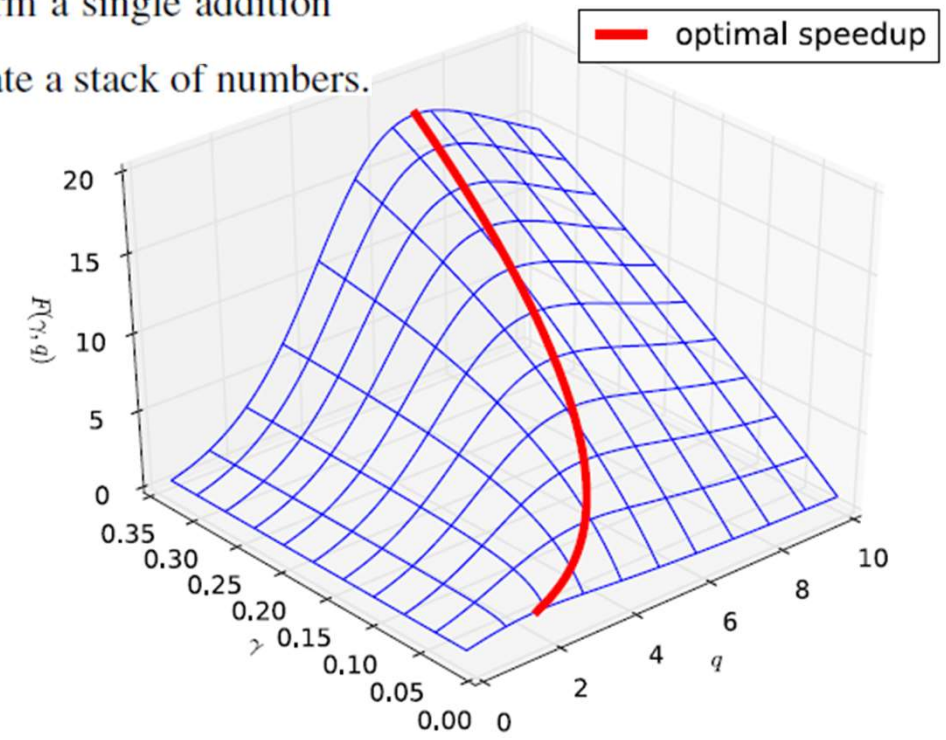
- **Exemplo 1 (continuação):**
  - Otimização de elementos segundo o problema $\sum_{i=0}^{n-1} A[i]$.

$p = 2^q$ PEs and $n = 2^k$ input numbers

- **Data distribution time:** $3 \times q$.
- **Computing local sums:** $n/p - 1 = 2^{k-q} - 1$.
- **Collecting partial results:** $3 \times q$.
- **Adding partial results:** q.

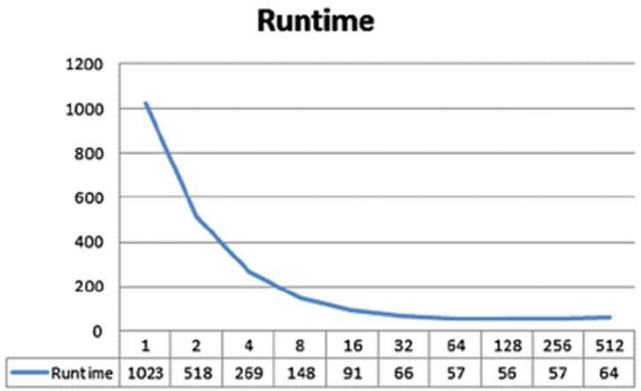
let $\alpha > 0$ be the time needed to perform a single addition and $\beta > 0$ be the time to com-municate a stack of numbers.

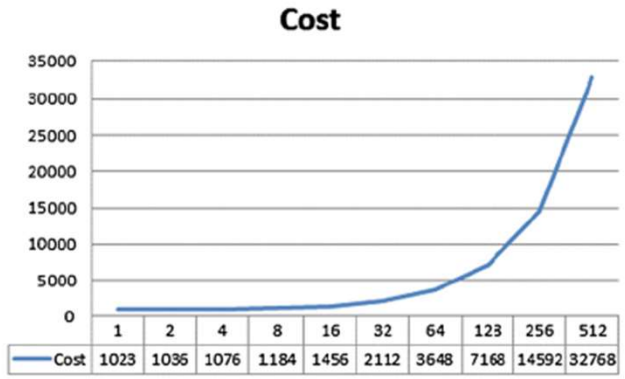
$\gamma = \frac{\alpha}{\beta}$

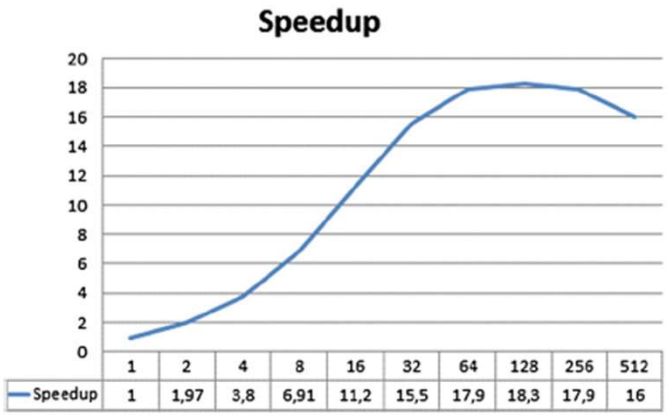


Métricas

- **Exemplo 1 (continuação):**
  - Ainda no problema $\sum_{i=0}^{n-1} A[i]$.







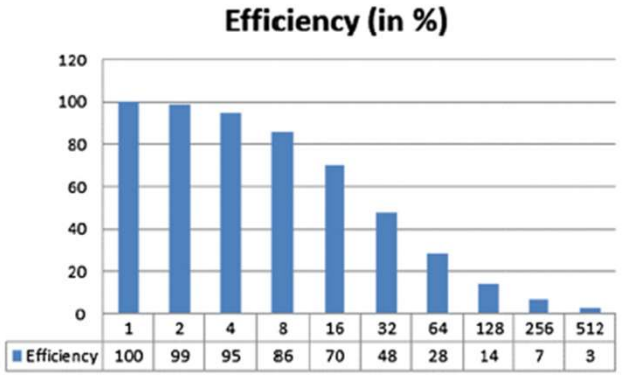


FIGURE 1.4

Strong scalability analysis: runtime, speedup, cost, and efficiency of our parallel summation algorithm for adding $n = 1024$ numbers on a varying number of PEs (ranging from 1 to 512).

Métricas

- Considerações na hora de paralelizar programas:
  - **Particionamento**: quebrar o problema em 'pedaços'. Pode ser feito das maneiras:
    - paralelismo de dados
    - paralelismo de tarefas
    - paralelismo do modelo (ver exemplo 2)
  - **Comunicação**: com base no esquema de particionamento, determina a quantidade e os tipos de recursos necessários de comunicação entre processos ou threads.
  - **Sincronização**: para cooperar de maneira apropriada, threads ou processos podem precisar ser sincronizado.
  - **Balanceamento de carga**: a quantidade de trabalho precisa ser dividida igualmente entre threads ou processos para minimizar tempos de inatividade

Modelo PRAM

- 'Parallel Random Access Machine' (**PRAM**)
  - n processadores idênticos P_i, i =0, . . . , n−1,

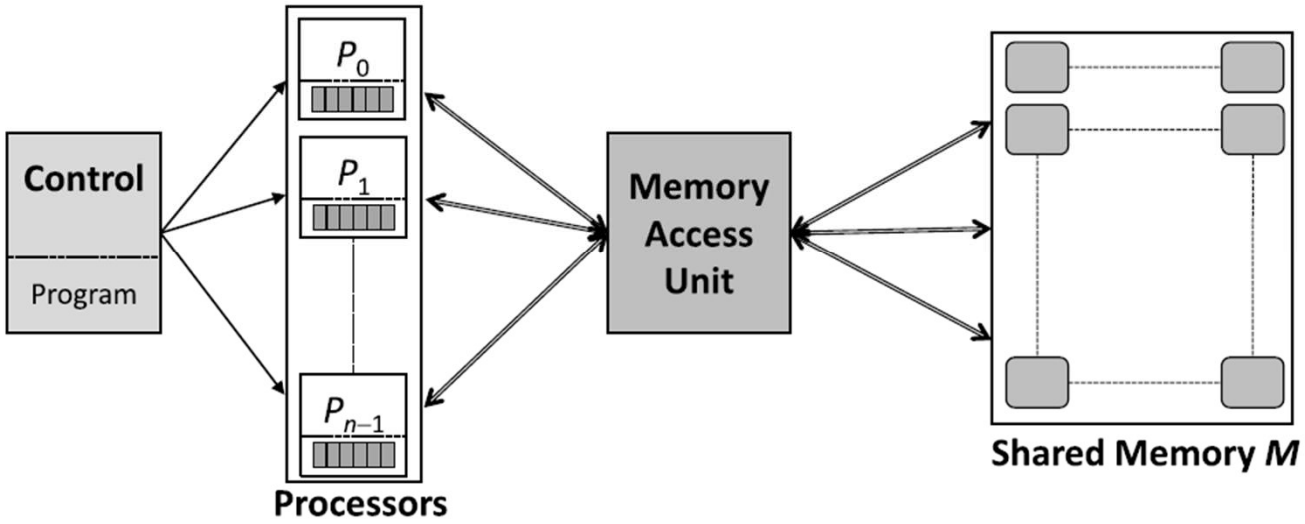
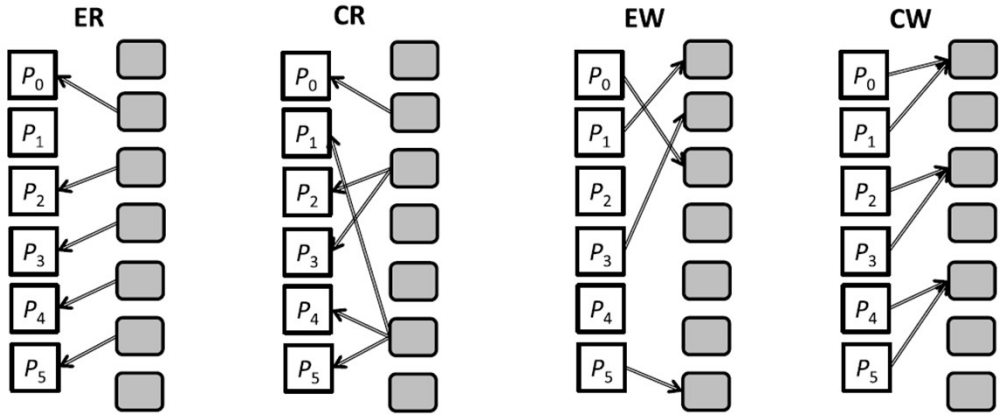


FIGURE 2.1

Important features of a PRAM: n processors $P_0, \ldots, P_{n-1}$ are connected to a global shared memory M, whereby any memory location is uniformly accessible from any processor in constant time. Communication between processors can be implemented by reading and writing to the globally accessible shared memory.

Modelo PRAM

- O compartilhamento de memória pode levar a estes cenários:



- ✓ ER (exclusive read) || CR (concurrent read) || EW (exclusive write) || CW (concurrent write).

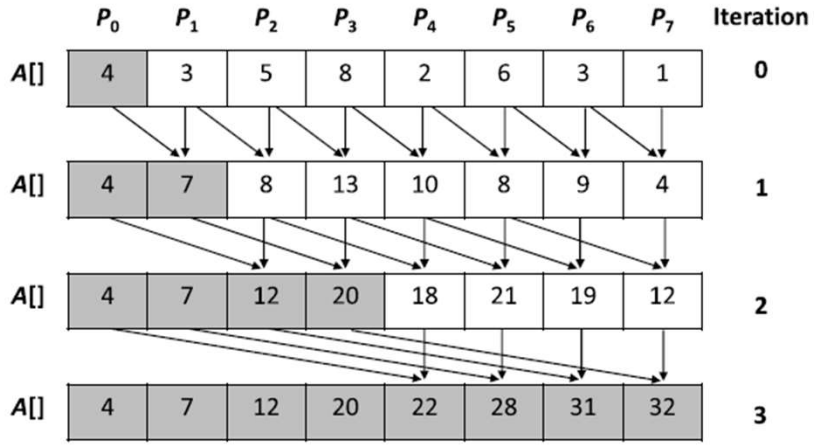
- ✓ Outras nomeclaturas:
  - ✓ *Exclusive Read Exclusive Write (**EREW**):* só um processador pode ler/escrever ao mesmo tempo em uma célula de memória.
  - ✓ *Concurrent Read Exclusive Write (**CREW**):* vários processadores podem ler dados simultameamente de uma mesma célula mas só um pode escrever
  - ✓ Concurrent Read Concurrent Write (**CRCW**): permite leituras e escritas simultâneas. Neste último caso, pode haver prioridade ou arbitrariedade

PRAM

- **Exemplo 2** (de paralelização usando PRAM):

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

$p = n$ processors




```
1  // each processor copies an array entry to a local register
2  for (j=0; j<n; j++) do_in_parallel
3      reg_j = A[j];
4
5  // sequential outer loop
6  for (i=0; i<ceil(log(n)); i++) do
7      // parallel inner loop performed by Processor j
8      for (j = pow(2,i); j<n; j++) do_in_parallel {
9          reg_j += A[j-pow(2,i)];  // perform computation
10         A[j] = reg_j; // write result to shared memory
11     }
```

FIGURE 2.3

Parallel prefix summation of an array A of size 8 on a PRAM with eight processors in three iteration steps based on recursive doubling.

Leis de AMDAHL

([1], pág. 32)

- Objetivo: determinar máximo ganho alcançável ao paralelizar um determinado programa sequencial.

- Um programa sequential (T (1)) em um único PE é compost por:

$$T(1) = T_{\text{ser}} + T_{\text{par}}$$

Paralelizando com p processadores:

$$T(p) \geq T_{\text{ser}} + \frac{T_{\text{par}}}{p}$$

Estimando o speedup S(p)

$$S(p) = \frac{T(1)}{T(p)} \leq \frac{T_{\text{ser}} + T_{\text{par}}}{T_{\text{ser}} + \frac{T_{\text{par}}}{p}}$$

O termo f denota a trafção de Tser em relação a T(1). Logo

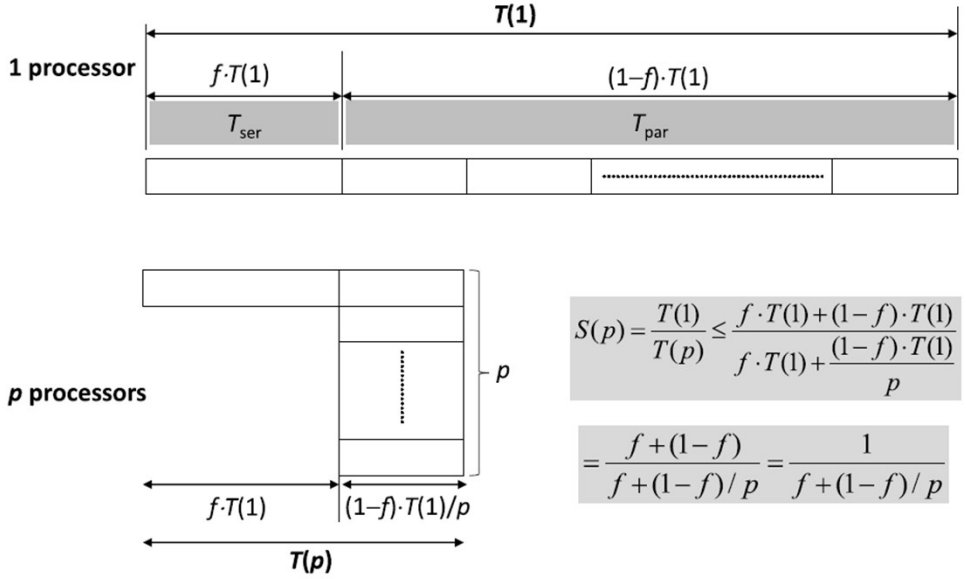
$$T_{\text{ser}} = f \cdot T(1) \text{ and } T_{\text{par}} = (1 - f) \cdot T(1) \qquad (0 \leq f \leq 1)$$

Lei de AMDAHL

- Lei de AMDAHL

$$S(p) = \frac{T(1)}{T(p)} \leq \frac{T_{ser} + T_{par}}{T_{ser} + \frac{T_{par}}{p}} = \frac{f \cdot T(1) + (1-f) \cdot T(1)}{f \cdot T(1) + \frac{(1-f) \cdot T(1)}{p}} = \frac{f + (1-f)}{f + \frac{(1-f)}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$

Ilustrando:



$$S(p) = \frac{T(1)}{T(p)} \leq \frac{f \cdot T(1) + (1-f) \cdot T(1)}{f \cdot T(1) + \frac{(1-f) \cdot T(1)}{p}}$$

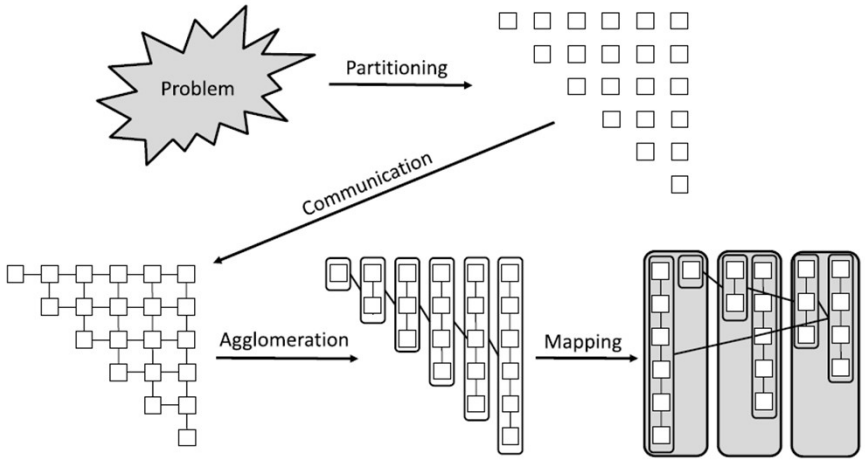
$$= \frac{f + (1-f)}{f + (1-f)/p} = \frac{1}{f + (1-f)/p}$$

Exemplo: 10% do tempo de execução de um programa é gasto em código inerentemente sequencial. Qual é o limite de aceleração alcançável?

$$S(\infty) \leq \lim_{p \to \infty} \frac{1}{0.1 + \frac{(0.9)}{p}} = 10$$

Algoritmo de paralelização de Foster

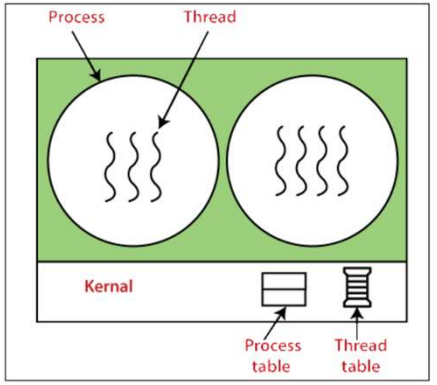
- Foco: (i) particionamento e (ii) comunicação (iii) sincronização e (iv) balanceamento carga
- Tanto cluster quanto multicore
- Processo metodológico:
    1) Um determinado problema é particionado em pequenas tarefas.
    2) A comunicação entre tarefas é especificada através da ligação entre elas
    3) Pequenas tarefas são aglomeradas em grandes tarefas (reduzir sobrecarga de comunicação)
        - "data locality" E "communication overhead"
    4) Tarefas são mapeadas em processadores para reduzir o tempo geral de execução. "
        - (i) minimizar comunicação entre processadores, (ii) melhorar concorrência atribuindo tarefas diferentes processadores e (iii) Balanço de cargas entre processadores

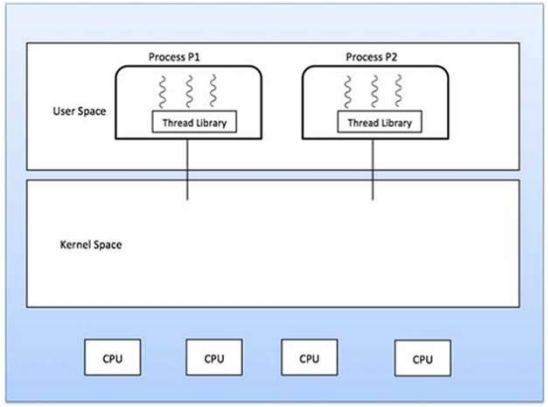


Neste exemplo, 7 tarefas são mapeadas em três processadores para obter um bom equilíbrio de carga (estático).

Multithreading [1]

- Mecanismos:
  - **Multiprocessamento**: paraleliza um programa em múltiplas unidades de computação (núcleos de CPU) para explorar recursos redundantes (registradores, ULA, CPUs)
  - **Multithreading**: compartilha os recursos de hardware, como caches e RAM de um único núcleo ou múltiplos núcleos para evitar ociosidade de recursos não utilizados.
- Tipos:
  - 'Físicos':
    - hardware thread ou hyperThread (Intel)
  - De software (SO):
    - User Level thread (ULT)
    - Kernel Level Thread (KLT)
  - 'Master' thread e derivadas
- Número de threads:
  - Número de núcleos x oversubscription





Multithreading

- Fluxo de threads

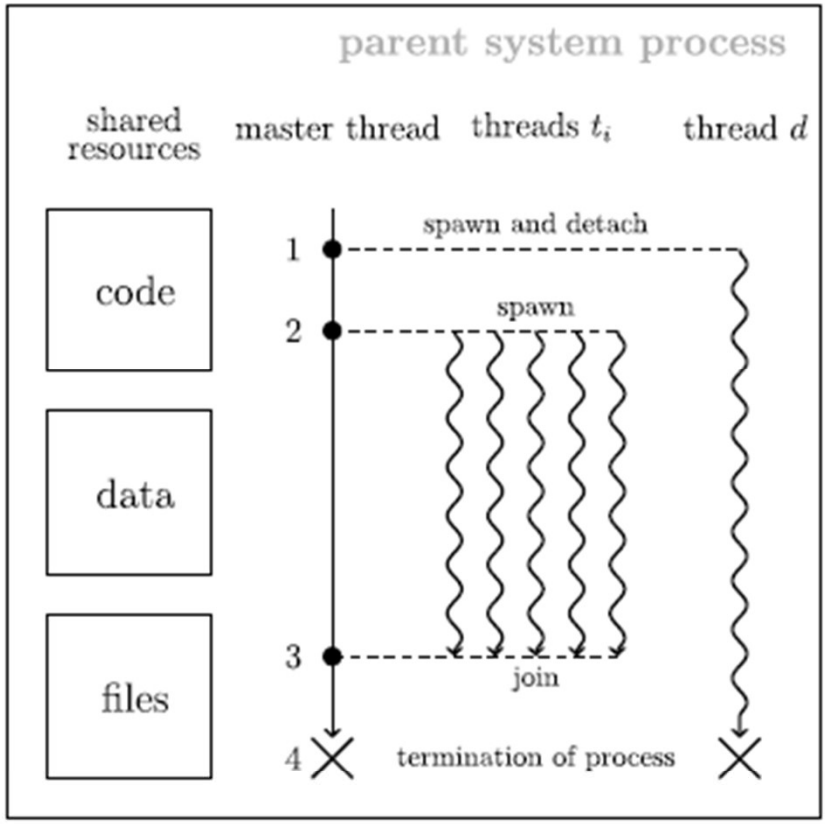


FIGURE 4.1

Exemplary workflow of a multithreaded program. First, we spawn a thread d and detach it immediately. This thread executes some work until finished or being terminated at the end of our program (4). Second, five threads t_i are spawned concurrently processing some data. Third, the master thread waits for them to finish by joining them. Fourth, we reach the end of the program resulting in the termination of the master thread and all detached threads. Code, data, and file handles are shared among all threads within their respective scopes.

Thread* class

- Exemplo:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define THREADS_MAX     4

void *function(void *param)
{
        int id = *((int *)(param));
        int i, loops = 10;
        for(i = 0; i < loops; i++)
        {
                printf("thread %d: loop %d\n", id, i);
        }
        pthread_exit(NULL);
}

int main(void)
{
        pthread_t threads[THREADS_MAX];
        int thread_args[THREADS_MAX];
        int i;
        printf("pre-execution\n");
        for (i = 0; i < THREADS_MAX; i++)
        {
                thread_args[i] = i;
                pthread_create(&threads[i], NULL, function, (void *) &thread_args[i]);
        }
        printf("mid-execution\n");

        for (i = 0; i < THREADS_MAX; i++)
        {
                pthread_join(threads[i], NULL);
        }
        printf("post-execution\n");
        return EXIT_SUCCESS;
}
```

* Obs.: também chamada de Pthread quando a classe segue o padrão POSIX de SOs

Thread class

- Classe “Thread” usada para gerenciar, junto SO, as thread
  - Métodos mais comuns:
    - **Suspend**(): suspende a execução de uma Thread (até o método Resume() seja chamado)
    - **Resume**(): reinicia uma thread suspensa.
    - **Sleep**(): uma thread pode suspender a si mesma utilizando esse método que espera um valor em milisegundos para especificar esse tempo de pausa.
    - **Join**(): chamado por uma thread, faz com que outras threads esperem por ela até que ela acabe sua execução.
    - **CurrentThread**(): retorna uma referência à thread em execução

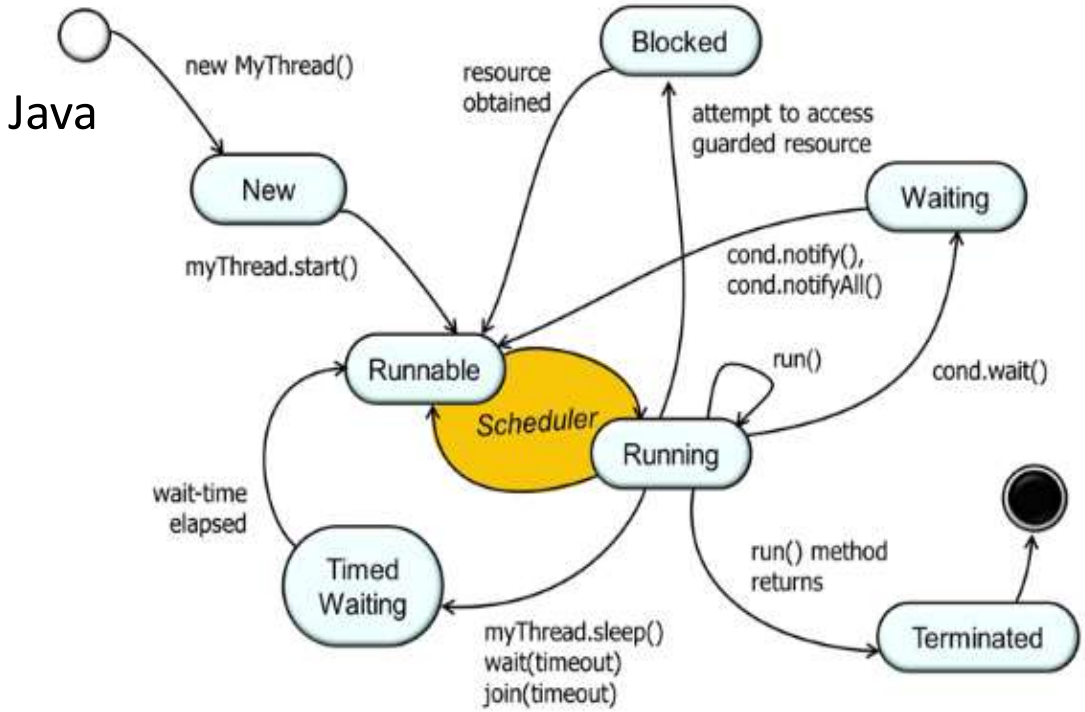
Thread class

- Classe “Thread” (continuação ...)
  - Estados de uma thread mais comuns (*enumeration ThreadState*):
    - **Aborted**: já abortada;
    - **AbortRequested**: quando uma thread chama o método Abort();
    - **Background**: rodando em background;
    - **Running**: Rodando depois que outra thread chama o método start();
    - **Stopped**: depois de terminar o método run() ou Abort();
    - **Suspended**: suspendida depois de chamar o método Suspend();
    - **Unstarted**: criada mas não iniciada.
    - **WaitSleepJoin**: quando uma thread chama Sleep() ou Join(), ou quando uma outra thread chama join();

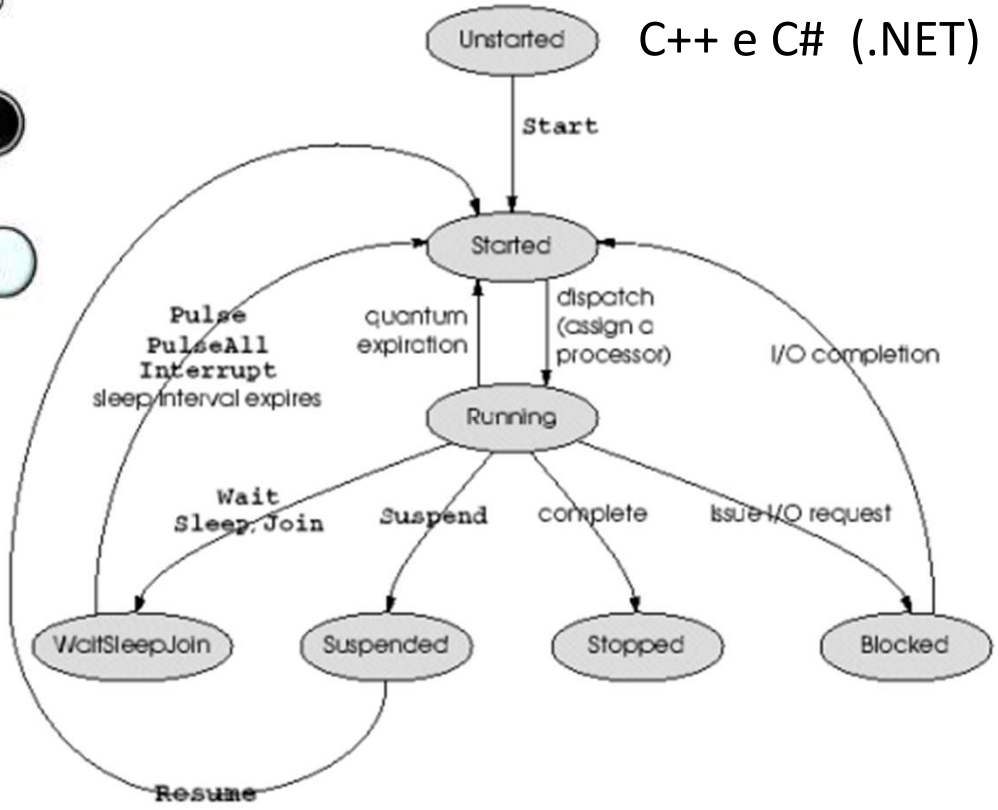
Thread class

- Exemplo dos estados da “Thread”

Java

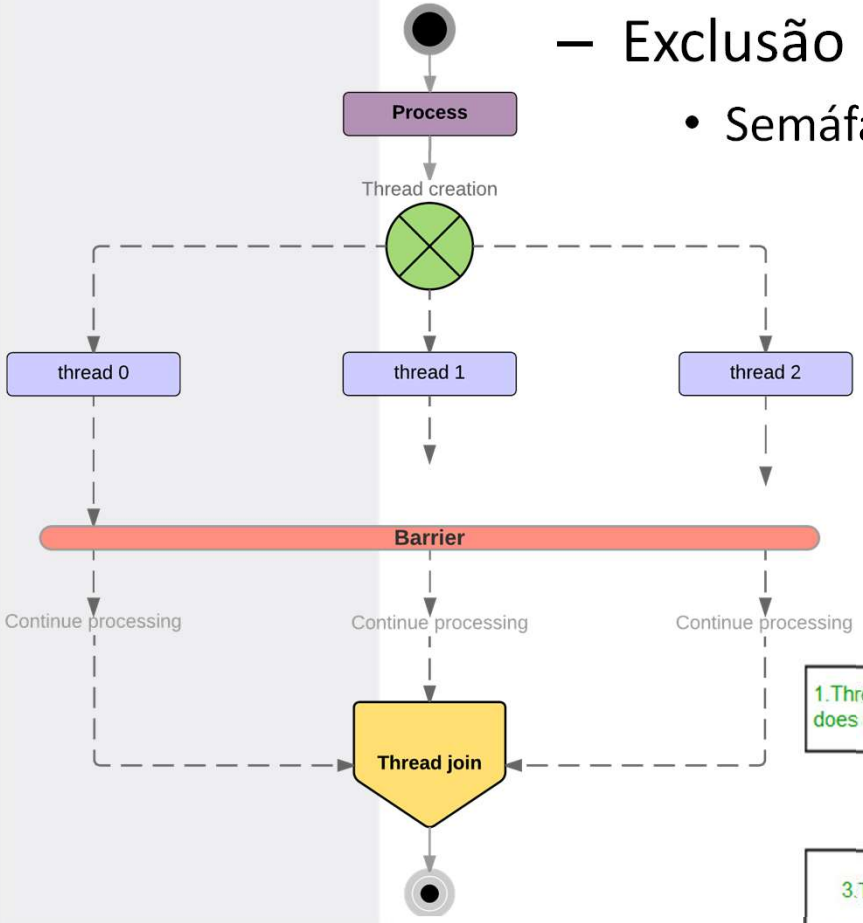


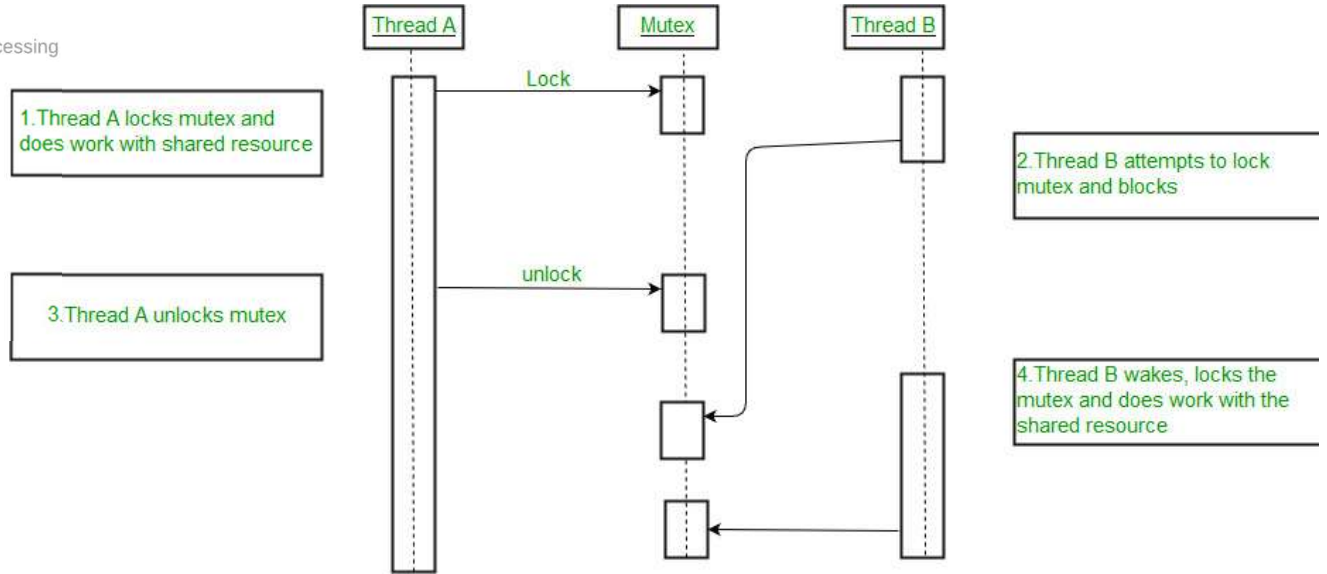
C++ e C# (.NET)



Thread class

- Compartilhamento de recursos de “Threads”
  - Barreiras
  - Exclusão mútua
    - Semáfaro -> mutex





Thread class

- Classe “Thread” (continuação ...)
  - Propriedades:
    - **Name**: retorna o nome da thread;
    - **ThreadState**: retorna o estado da thread;
    - **Priority**: retorna a prioridade da thread que pode ser:
      - Highest
      - AboveNormal
      - Normal
      - BelowNormal
      - Lowest
    - **IsAlive**: retorna um valor booleano indicando se uma thread está “viva” ou não;
    - **IsBAckground**: retorna um booleano indicando se a thread está rodando em background ou foreground.

Thread class

- Classe “Thread” (continuação ...)
  - ‘Regras’ para evitar problemas:
    1) Cuidado com compartilhamento de recursos (variáveis, arquivos, I/Os, etc)
    2) Cada thread só pode ser Joined ou concluída uma vez (não podem ser reutilizados)
    3) Todos os threads devem ser joined ou terminadas dentro do escopo de sua declaração

Exemplo 1 – criação c++

STR

```
std::thread * threads = new std::thread[num_threads];
```

```
1  #include <cstdint>    // uint64_t
2  #include <vector>     // std::vector
3  #include <thread>     // std::thread
4
5  // this function will be called by the threads (should be void)
6  void say_hello(uint64_t id) {
7      std::cout << "Hello from thread: "  << id << std::endl;
8  }
9
10 // this runs in the master thread
11 int main(int argc, char * argv[]) {
12
13     const uint64_t num_threads = 4;
14     std::vector<std::thread> threads;
15
16     // for all threads
17     for (uint64_t id = 0; id < num_threads; id++)
18         // emplace the thread object in vector threads
19         // using argument forwarding, this avoids unnecessary
20         // move operations to the vector after thread creation
21         threads.emplace_back(
22             // call say_hello with argument id
23             say_hello, id
24         );
25
26     // join each thread at the end
27     for (auto& thread: threads)
28         thread.join();
29 }
```

```
threads.push_back(std::thread(say_hello, id));
```

```
g++ -O2 -std=c++11 -pthread hello_world.cpp -o hello_world
```

Exemplo 2 – Fibonacci

$$a_n = a_{n-1} + a_{n-2} \quad \text{with initial conditions} \quad a_0 = 0, a_1 = 1$$

```cpp
#include <iostream>        // std::cout
#include <cstdint>         // uint64_t
#include <vector>          // std::vector
#include <thread>          // std::thread

template <
    typename value_t,
    typename index_t>
void fibo(
    value_t n,
    value_t * result) {    // <- here we pass the address

    value_t a_0 = 0;
    value_t a_1 = 1;

    for (index_t index = 0; index < n; index++) {
        const value_t tmp = a_0; a_0 = a_1; a_1 += tmp;
    }

    *result = a_0;         // <- here we write the result
}
```

```cpp
// this runs in the master thread
int main(int argc, char * argv[]) {

    const uint64_t num_threads = 32;
    std::vector<std::thread> threads;

    // allocate num_threads many result values
    std::vector<uint64_t> results(num_threads, 0);

    for (uint64_t id = 0; id < num_threads; id++)
        threads.emplace_back(
            // specify template parameters and arguments
            fibo<uint64_t, uint64_t>, id, &(results[id])
        );

    // join the threads
    for (auto& thread: threads)
        thread.join();

    // print the result
    for (const auto& result: results)
        std::cout << result << std::endl;
}
```

Exemplo 2 – Fibonacci

- Mecanismo de “promise” do C++ para troca valores entre threads

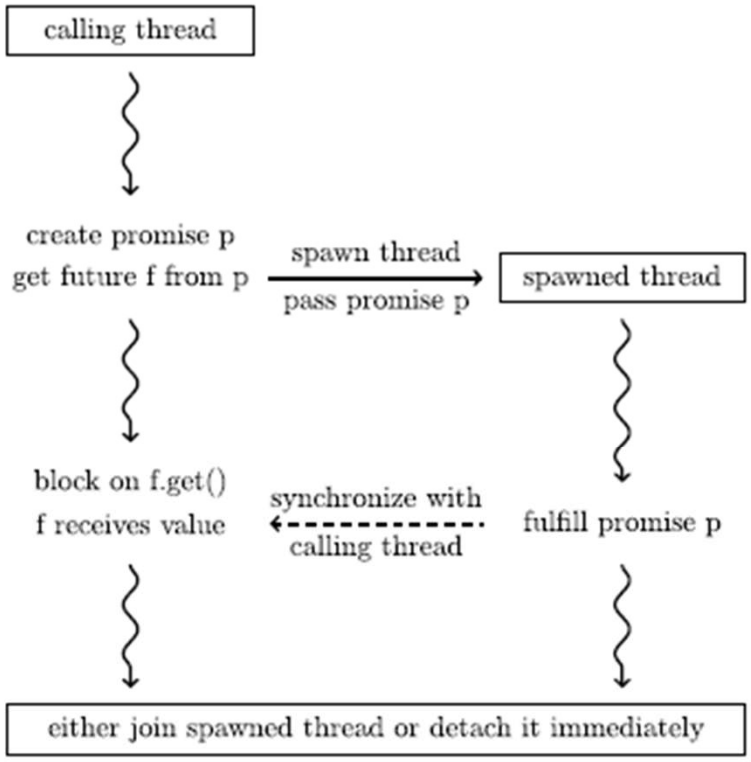


FIGURE 4.2

Synchronization of two threads using a promise and its associated future.

Exemplo 2 – Fibonacci

com *promise* e *future*


```
4  #include <thread>                      // std::thread
5  #include <future>                      // std::promise/future
6
7  template <
8      typename value_t,
9      typename index_t>
10 void fibo(
11     value_t n,
12     std::promise<value_t> && result) {  // <- pass promise
13
14     value_t a_0 = 0;
15     value_t a_1 = 1;
16
17     for (index_t index = 0; index < n; index++) {
18         const value_t tmp = a_0; a_0 = a_1; a_1 += tmp;
19     }
20
21     result.set_value(a_0);              // <- fulfill promise
22 }
23
24 int main(int argc, char * argv[]) {
25
26     const uint64_t num_threads = 32;
27     std::vector<std::thread> threads;
28
29     // storage for futures
30     std::vector<std::future<uint64_t>> results;
31
32     // for each thread
33     for (uint64_t id = 0; id < num_threads; id++) {
34
35         // define a promise and store the associated future
36         std::promise<uint64_t> promise;
37         results.emplace_back(promise.get_future());
38
39         // emplace the spawned thread
40         threads.emplace_back(
41             // move the promise to the spawned thread
42             // note that promise is now moved elsewhere
43             // and cannot be accessed safely anymore
44             fibo<uint64_t, uint64_t>, id, std::move(promise)
45         );
46     }
47
48     // read the futures resulting in synchronization of threads
49     // up to the point where promises are fulfilled
50     for (auto& result: results)
51         std::cout << result.get() << std::endl;
52
53     // this is mandatory since threads have to be either
54     // joined or detached at the end of our program
55     for (auto& thread: threads)
56         thread.join();
57 }
```

Exemplo 2 – Fibonacci

Com modo *asynchronous*


```cpp
#include <iostream>  // std::cout
#include <cstdint>   // uint64_t
#include <vector>    // std::vector
#include <future>    // std::async

// traditional signature of fibo without syntactic noise
uint64_t fibo(uint64_t n) {

    uint64_t a_0 = 0;
    uint64_t a_1 = 1;

    for (uint64_t index = 0; index < n; index++) {
        const uint64_t tmp = a_0; a_0 = a_1; a_1 += tmp;
    }

    return a_0;
}
```

```cpp
int main(int argc, char * argv[]) {

    const uint64_t num_threads = 32;
    std::vector<std::future<uint64_t>> results;

    // for each thread
    for (uint64_t id = 0; id < num_threads; id++)
        // directly emplace the future
        results.emplace_back(
            std::async(
                std::launch::async, fibo, id
            )
        );

    // synchronization of spawned threads
    for (auto& result: results)
        std::cout << result.get() << std::endl;
}
```

Exemplo 3 – DMV

- DMV = dense matrix vector products $b_i = \sum_{j=0}^{n-1} A_{ij} \cdot x_j \quad \text{for all} \quad i \in \{0, \ldots, m-1\}$
- Versão sequencial:

```
6  #include "../include/hpc_helpers.hpp" // custom timers
7
8  // initialize A as lower triangular matrix
9  // simulating prefix summation and vector x
10 // with consecutive values (0, 1, 2, 3, ...)
11 template <
12     typename value_t,
13     typename index_t>
14 void init(
15     std::vector<value_t>& A,
16     std::vector<value_t>& x,
17     index_t m,
18     index_t n) {
19
20     for (index_t row = 0; row < m; row++)
21         for (index_t col = 0; col < n; col++)
22             A[row*n+col] = row >= col ? 1 : 0;
23
24     for (index_t col = 0; col < n; col++)
25         x[col] = col;
26 }
```

```
28 // the sequential matrix vector product
29 template <
30     typename value_t,
31     typename index_t>
32 void sequential_mult(
33     std::vector<value_t>& A,
34     std::vector<value_t>& x,
35     std::vector<value_t>& b,
36     index_t m,
37     index_t n) {
38
39     for (index_t row = 0; row < m; row++) {
40         value_t accum = value_t(0);
41         for (index_t col = 0; col < n; col++)
42             accum += A[row*n+col]*x[col];
43         b[row] = accum;
44     }
45 }
```

```
47 int main(int argc, char* argv[]) {
48
49     const uint64_t n = 1UL << 15;
50     const uint64_t m = 1UL << 15;
51
52     TIMERSTART(overall)
53
54     TIMERSTART(alloc)
55     std::vector<uint64_t> A(m*n);
56     std::vector<uint64_t> x(n);
57     std::vector<uint64_t> b(m);
58     TIMERSTOP(alloc)
59
60     TIMERSTART(init)
61     init(A, x, m, n);
62     TIMERSTOP(init)
63
64     TIMERSTART(mult)
65     sequential_mult(A, x, b, m, n);
66     TIMERSTOP(mult)
67
68     TIMERSTOP(overall)
69
70     // check if summation is correct
71     for (uint64_t index = 0; index < m; index++)
72         if (b[index] != index*(index+1)/2)
73             std::cout << "error at position "
74                       << index << std::endl;
75
76 }
```

Exemplo 3 – DMV

Com distribuição por ‘blocos’

- Estratégia:

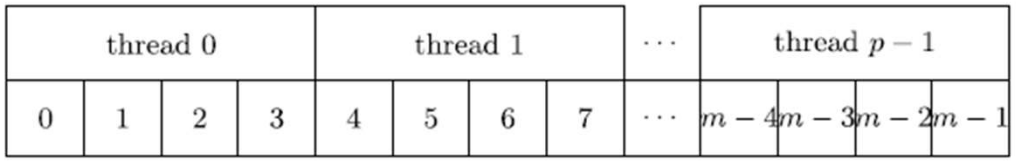


FIGURE 4.3

An example of a static block distribution assigning $c = 4$ consecutive tasks to each of the p threads in order to concurrently process $p \cdot c = m$ overall tasks.

- Acesso a variáveis por ‘função lambda’

```
auto add_one = [] (const uint64_t& v) { return v+1; };
```

```
uint64_t w = 1;
// ERROR: w is not declared within the body of add_w!
auto add_w = [] (const uint64_t& v) { return v+w; };
```

```
uint64_t w = 1;  // w will be accessed inside lambdas

// capture w by value
auto add_w_0 = [w]  (const uint64_t& v) { return v+w; };

// capture w by reference
auto add_w_1 = [&w] (const uint64_t& v) { return v+w; };

// capture everything accessed in add_w_2 by reference
auto add_w_2 = [&]  (const uint64_t& v) { return v+w; };

// capture everything accessed in add_w_3 by value
auto add_w_3 = [=]  (const uint64_t& v) { return v+w; };
```

Exemplo 3 – DMV

Com distribuição por 'blocos'

STR

```
1  template <
2      typename value_t,
3      typename index_t>
4  void block_parallel_mult(
5      std::vector<value_t>& A,
6      std::vector<value_t>& x,
7      std::vector<value_t>& b,
8      index_t m,
9      index_t n,
10     index_t num_threads=8) {
11
12     // this function is called by the threads
13     auto block = [&] (const index_t& id) -> void {
14         //        ^-- capture whole scope by reference
15
16         // compute chunk size, lower and upper task id
17         const index_t chunk = SDIV(m, num_threads);
18         const index_t lower = id*chunk;
19         const index_t upper = std::min(lower+chunk, m);
20
21         // only computes rows between lower and upper
22         for (index_t row = lower; row < upper; row++) {
23             value_t accum = value_t(0);
24             for (index_t col = 0; col < n; col++)
25                 accum += A[row*n+col]*x[col];
26             b[row] = accum;
27         }
28     };
29
30     // business as usual
31     std::vector<std::thread> threads;
32
33     for (index_t id = 0; id < num_threads; id++)
34         threads.emplace_back(block, id);
35
36     for (auto& thread : threads)
37         thread.join();
38 }
```

Exemplo 3 – DMV

Com distribuição cíclica

- Estratégia:

threads	0	1	2	$\cdots$	$p-1$	0	1	2	$\cdots$	$p-1$	$\cdots$
tasks	0	1	2	$\cdots$	$p-1$	p	$p+1$	$p+2$	$\cdots$	$2p-1$	$\cdots$

FIGURE 4.4

An example of a static cyclic distribution assigning c tasks to each thread in a round-robin fashion using a stride of p.

```
1  template <
2      typename value_t,
3      typename index_t>
4  void cyclic_parallel_mult(
5      std::vector<value_t>& A,
6      std::vector<value_t>& x,
7      std::vector<value_t>& b,
8      index_t m,
9      index_t n,
10     index_t num_threads=8) {
11
12     // this function is called by the threads
13     auto cyclic = [&] (const index_t& id) -> void {
14
15         // indices are incremented with a stride of p
16         for (index_t row = id; row < m; row += num_threads) {
17             value_t accum = value_t(0);
18             for (index_t col = 0; col < n; col++)
19                 accum += A[row*n+col]*x[col];
20             b[row] = accum;
21         }
22     };
23
24     // business as usual
25     std::vector<std::thread> threads;
26
27     for (index_t id = 0; id < num_threads; id++)
28         threads.emplace_back(cyclic, id);
29
30     for (auto& thread : threads)
31         thread.join();
32 }
```

Referências

- [1] Parallel programming concepts and practice.
- [2] An introduction to parallel programming