

# Introdução à Computação em Física

INTEGRAÇÃO NUMÉRICA  
PROF. WALBER

Refs.:

Cálculo numérico, aspectos teóricos e computacionais (2nd edição), M. A. G. Ruggiero, V. L. da Rocha Lopes  
Numerical Python, Second Edition, Robert Johansson

## Ideia principal

---

Queremos obter o valor de uma integral definida  $I$ , de uma função contínua  $f(x)$  no intervalo  $[a,b]$

$$I = \int_a^b f(x)dx$$

Iremos estudar métodos numéricos que são aplicados em casos onde  $f(x)$  é uma função de difícil integração ou função é conhecida apenas para um conjunto de pontos, por exemplo.

Ideia é substituir  $f(x)$  por um polinômio interpolador, que irá facilitar a obtenção de  $I$ , de tal maneira que

$$\int_a^b f(x)dx = \sum_{i=0}^n A_i f(x_i)$$

Quadratura numérica

## Fórmulas de Newton-Cotes

---

Vamos considerar que  $f(x)$  pode ser escrita em termos do polinômio interpolador de Lagrange:

$$p_n(x) = \sum_{i=0}^n f(x_i) L_i(x)$$

$$L_k(x) = \frac{\prod_{j=0, j \neq k}^n (x - x_j)}{\prod_{j=0, j \neq k}^n (x_k - x_j)}$$

Com isso, podemos escrever:

$$\int_a^b f(x) dx = \int_a^b \sum_{i=0}^n f(x_i) L_i(x) dx + \int_a^b \prod_{i=0}^n (x - x_i) \frac{f^{(n+1)}(\xi_x)}{(n+1)!} dx$$


Aproximação da integral


Erro gerado pela interpolação

## A regra dos trapézios

---

Considerando  $x_0 = a$  e  $x_1 = b$  ( $h = b - a$ ), e utilizando o polinômio de grau um de Lagrange:

$$p_1(x) = \frac{(x - x_1)}{(x_0 - x_1)} f(x_0) + \frac{(x - x_0)}{(x_1 - x_0)} f(x_1)$$


$$\int_a^b f(x) dx = \int_a^b \left[ \frac{(x - x_1)}{(x_0 - x_1)} f(x_0) + \frac{(x - x_0)}{(x_1 - x_0)} f(x_1) \right] dx + E$$


$$I = \int_a^b f(x) dx = \frac{(x_1 - x_0)}{2} [f(x_0) + f(x_1)] + E$$

## A regra dos trapézios

Considerando  $x_0 = a$  e  $x_1 = b$  ( $h = b - a$ ), e utilizando o polinômio de grau um de Lagrange:

$$I = \int_a^b f(x) dx = \frac{(x_1 - x_0)}{2} [f(x_0) + f(x_1)] + E$$

área do trapézio



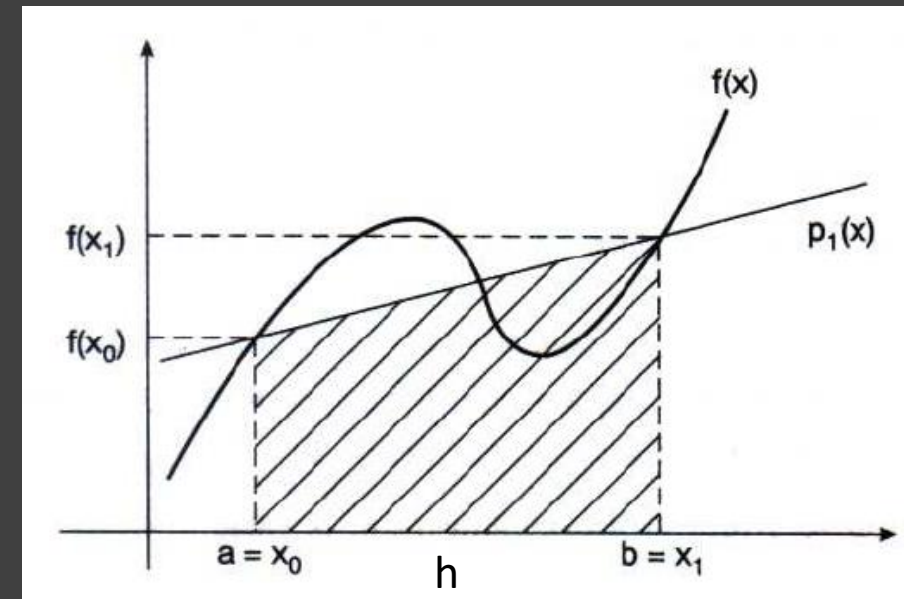
$$I \approx \int_a^b f(x) dx = \frac{h}{2} [f(x_0) + f(x_1)]$$

Pode-se mostrar que o erro:

$$|E| \leq \frac{h^3}{12} \max |f''(\xi)|$$

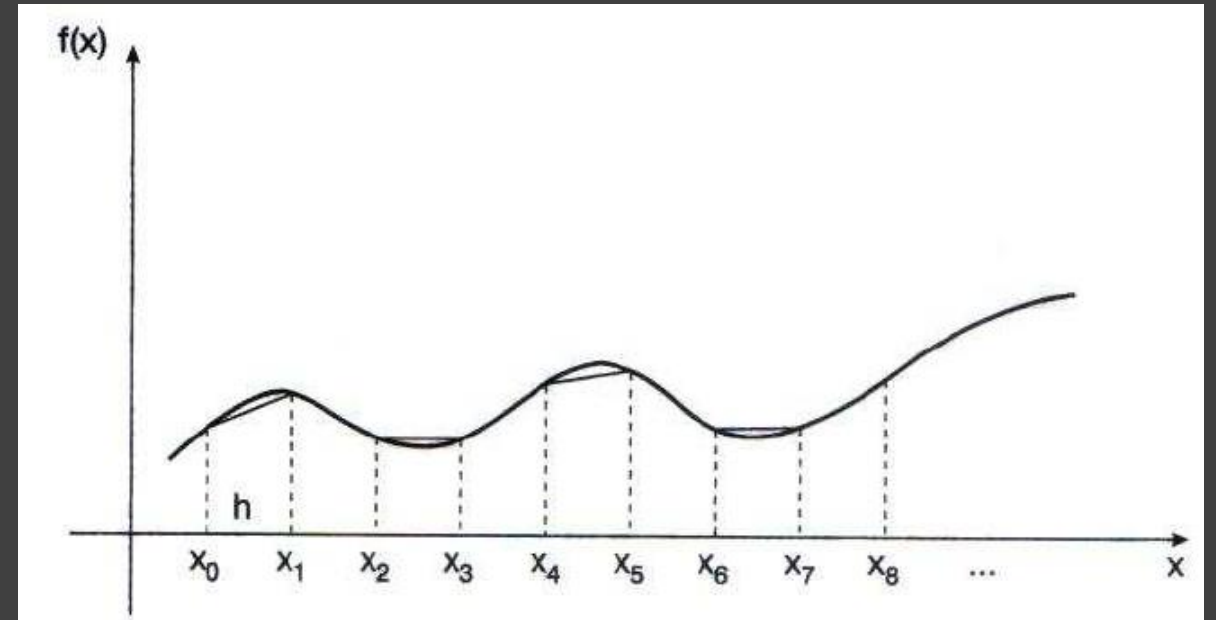
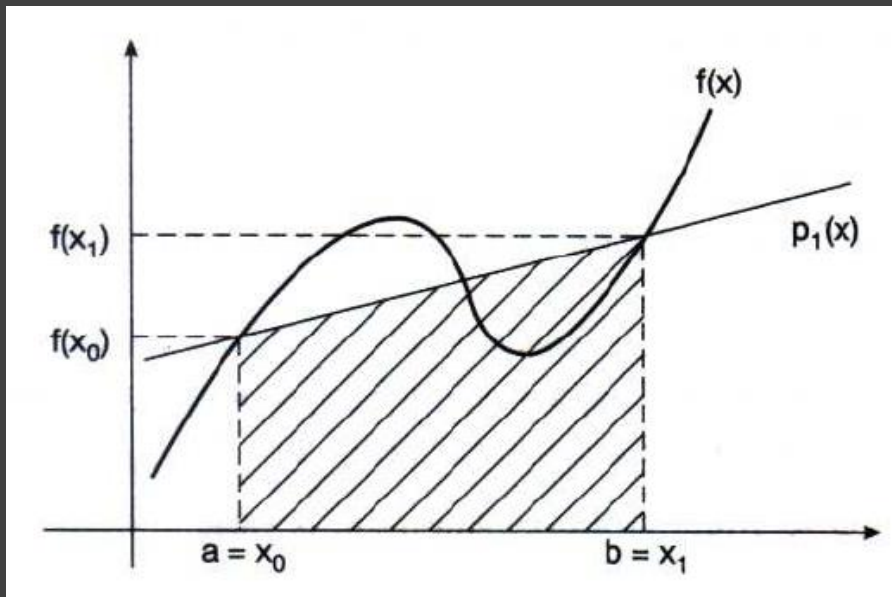
Onde:

$$\xi \in (a, b)$$



## A regra dos trapézios repetida

A ideia é diminuir o tamanho dos intervalos  $h$ , e com isso obter resultados mais próximos do valor da integral exata



## A regra dos trapézios repetida

---

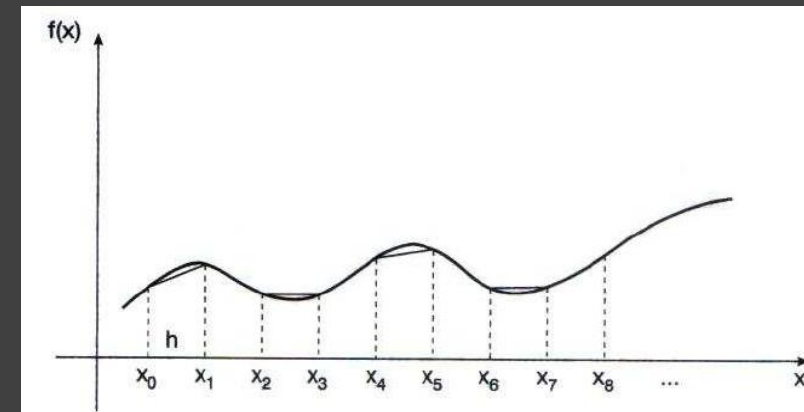
Assim sendo, podemos escrever

$$h = x_{i+1} - x_i$$

$$\int_a^b f(x)dx = \sum_{i=0}^{m-1} \int_{x_i}^{x_{i+1}} f(x)dx \approx \sum_{i=0}^{m-1} \left[ \frac{h}{2} (f(x_i) + f(x_{i+1})) \right]$$



$$\int_a^b f(x)dx \approx \frac{h}{2} [f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{m-1}) + f(x_m)]$$



## Comparação:

---

Regra do trapézio simples:

$$I \approx \int_a^b f(x)dx = \frac{h}{2}[f(x_0) + f(x_1)]$$

$$|E| \leq \frac{h^3}{12} \max |f''(\xi)|$$

$$\xi \in (a, b)$$

Regra do trapézio repetida:

$$\int_a^b f(x)dx \approx \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{m-1}) + f(x_m)]$$

$$|E_{TR}| \leq \frac{mh^3}{12} \max |f''(\xi)|$$

No caso da regra do trapézio repetida o erro está associado com o número de divisões do intervalo  $[a,b]$ .



Exemplo:

---

Vamos calcular a integral abaixo através da regra dos trapézios repetida:

$$I = \int_0^1 e^x dx$$

Dividindo o intervalo entre 0 a 1, em pontos  $x_i = 0.1 \times i$  ( $i = 0, 1, 2, \dots, 10$ ) ( $h = 1/10$ ):

$$I \approx \frac{0.1}{2} (e^0 + 2e^{0.1} + 2e^{0.2} + \dots + 2e^{0.8} + 2e^{0.9} + e) = 1.719713$$

Valor do erro pode ser obtido como:

$$|E_{TR}| \leq \frac{10 \times 0.1^3}{12} \max |e^x| = 0.002265$$

$$x \in [0, 1]$$

Exemplo:

---

Sendo:

$$I = \int_0^1 e^x dx$$

$$I \approx \frac{0.1}{2} (e^0 + 2e^{0.1} + 2e^{0.2} + \dots + 2e^{0.8} + 2e^{0.9} + e) = 1.719713$$

$$|E_{TR}| \leq \frac{10 \times 0.1^3}{12} \max |e^x| = 0.002265$$

$$x \in [0, 1]$$

Podemos estimar o número de divisões para uma erro menor que  $10^{-3}$ :

$$mh = x_m - x_0 = 1 - 0 = 1$$



$$|E_{TR}| \leq \frac{h^2 e}{12}$$



$$h^2 < \frac{12 \times 10^{-3}}{e} \approx 0.00441$$

$$m = \frac{1}{h} \geq 16$$

## Regra 1/3 de Simpson

---

Vamos agora considerar o polinômio de Lagrange de segundo grau. Assim sendo, temos que:

$$p_2(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}f(x_0) + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}f(x_1) + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}f(x_2)$$

Sendo  $x_0 = a$ ,  $x_2 = b$ ,  $x_1 = a+h$ , ( $h = b-a/2$ ):

$$I = \int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)]$$

$$|E_S| \leq \frac{h^5}{90} \max |f^{(4)}(x)| \quad x \in [0, 1]$$

Erro agora vai com  $h^5$ , que nos mostra que há um aumento considerável da precisão.

## Regra 1/3 de Simpson repetida

Ideia é aplicar a regra 1/3 de Simpson para cada subintervalo. Nesse caso vamos considerar que  $x_0, x_1, \dots, x_m$  são pontos igualmente espaçados ( $h = x_{i+1} - x_i$ ), e  $m$  é par (condição necessária pois cada polinômio usa três pontos).

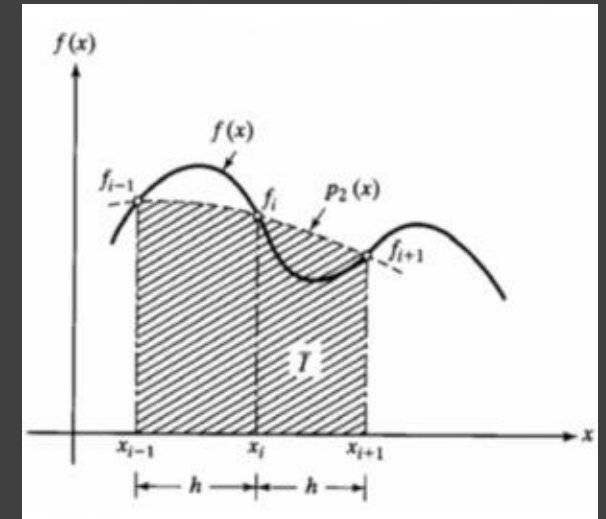
Para cada par de subintervalos:

$$\int_{x_{2k-2}}^{x_{2k}} f(x) dx \approx \frac{h}{3} [f(x_{2k-2}) + 4f(x_{2k-1}) + f(x_{2k})]$$

$$k = 1, \dots, \frac{m}{2}$$

Com isso,

$$I = \int_{x_0}^{x_m} f(x) dx \approx \frac{h}{3} \{ [f(x_0) + 4f(x_1) + f(x_2)] + [f(x_2) + 4f(x_3) + f(x_4)] + \dots + [f(x_{m-2}) + 4f(x_{m-1}) + f(x_m)] \}$$



## Regra 1/3 de Simpson repetida

---

$$I = \int_{x_0}^{x_m} f(x)dx \approx \frac{h}{3} \{ [f(x_0) + 4f(x_1) + f(x_2)] + [f(x_2) + 4f(x_3) + f(x_4)] + \cdots + [f(x_{m-2}) + 4f(x_{m-1}) + f(x_m)] \}$$



$$I \approx \frac{h}{3} \{ [f(x_0) + f(x_m)] + 4[f(x_1) + f(x_3) + \cdots + f(x_{m-1})] + 2[f(x_2) + f(x_4) + \cdots + f(x_{m-2})] \}$$

Onde o erro é dado por

$$|E_{SR}| \leq \frac{mh^5}{180} \max |f^{(4)}(x)|$$

$$x \in [x_0, x_m]$$

## Comparação entre os diferentes métodos

---

Regra trapézio:

$$I \approx \int_a^b f(x)dx = \frac{h}{2}[f(x_0) + f(x_1)]$$

$$|E| \leq \frac{h^3}{12} \max |f''(\xi)|$$

$$\xi \in (a, b)$$

Regra do trapézio repetida:

$$\int_a^b f(x)dx \approx \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{m-1}) + f(x_m)]$$

$$|E_{TR}| \leq \frac{mh^3}{12} \max |f''(\xi)|$$

Regra 1/3 de Simpson:

$$I = \int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)]$$

$$|E_S| \leq \frac{h^5}{90} \max |f^{(4)}(x)|$$

$$x \in [x_0, x_m]$$

Regra 1/3 de Simpson repetida:

$$I \approx \frac{h}{3}\{[f(x_0) + f(x_m)] + 4[f(x_1) + f(x_3) + \cdots + f(x_{m-1})] + 2[f(x_2) + f(x_4) + \cdots + f(x_{m-2})]\}$$

$$|E_{SR}| \leq \frac{mh^5}{180} \max |f^{(4)}(x)|$$

## Atividade prática

---

1. Implementar código utilizando o método dos trapézios repetidos para calcular a seguinte integral:

$$\int_0^{\pi/4} e^{3x} \operatorname{sen}(2x) dx$$

Considere 10, 50, 100 e 200 pontos.

2. Implementar código utilizando a regra 1/3 de Simpson repetida para calcular a mesma integral acima.
3. Propor um exemplo de aplicação em Física (trazer na próxima aula).

# Integração numérica com SciPy

---



## Integração via SciPy

---

Integração simples pode ser feita através da biblioteca SciPy,

```
from scipy import integrate
```

```
...
```

```
inte, eroint = integrate.quad(fint, 0.0, np.pi/4.0)
```

$$\int_0^{\pi/4} e^{3x} \operatorname{sen}(2x) dx$$

Retorna valor da integral  
(inte) e erro associado  
(eroint)

Mais informações em:

<https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html>

# Prelúdio de computação paralela



## Computação de alto desempenho (HPC)

---

- Vários problemas em Física requerem a solução de problemas complexos, que por sua vez, exigem intensa carga computacional (workload).
- É nesse sentido, que podemos apontar a chamada computação de alto desempenho (high performance computing – HPC) como ferramenta para solução de problemas em diversas áreas da Física.
- Podemos pensar então, que o HPC envolve uma parte de hardware onde o uso de supercomputadores, ou até mesmo, de agregados (clusters) de computadores são utilizados para se atingir alta performance computacional.
- Nesse sentido, os códigos numéricos devem ser capazes de explorar a alta disponibilidade de processamento desses sistemas. [Computação paralela](#).

# Tarefas em serial

---

Execução de tarefas em serial:

Linha de montagem Ford

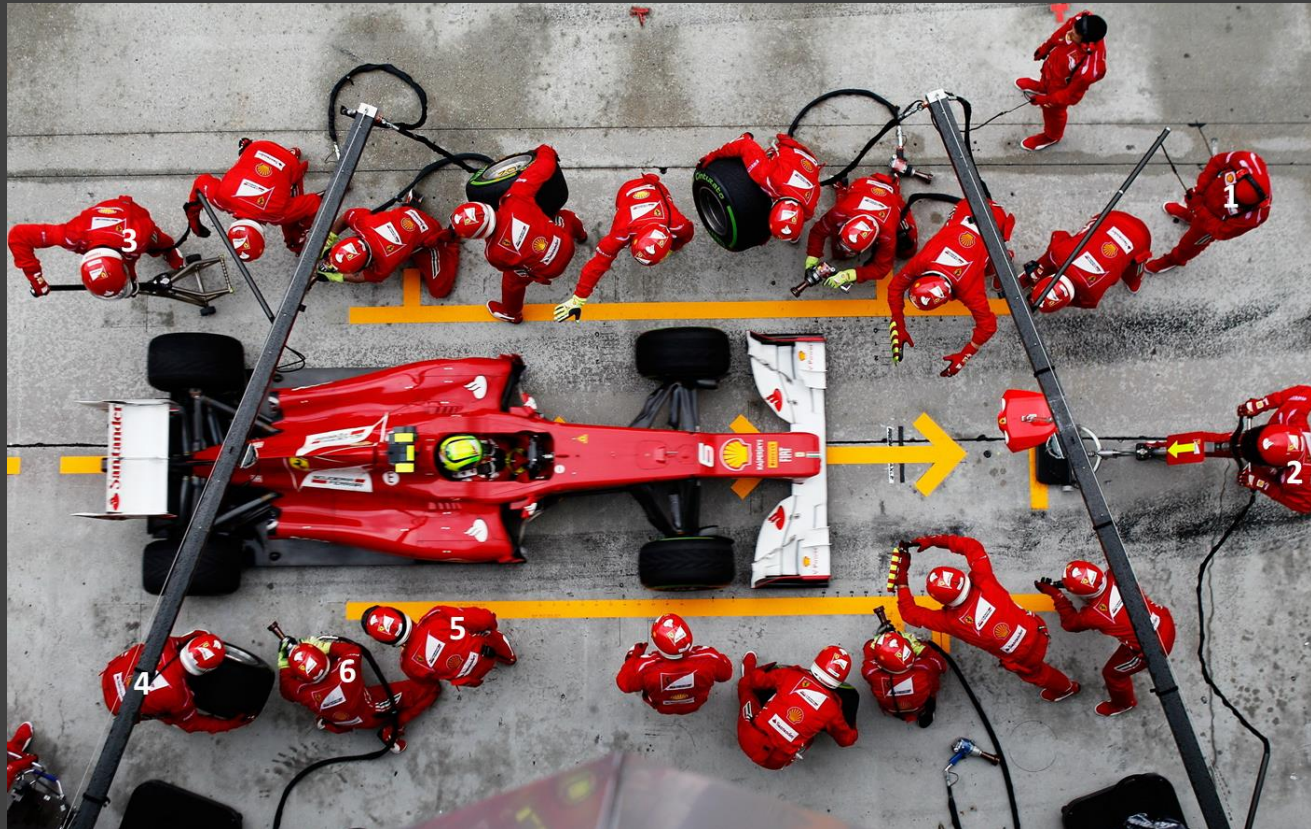




# Tarefas em paralelo

---

Execução de tarefas em paralelo (Pit stop formula 1)

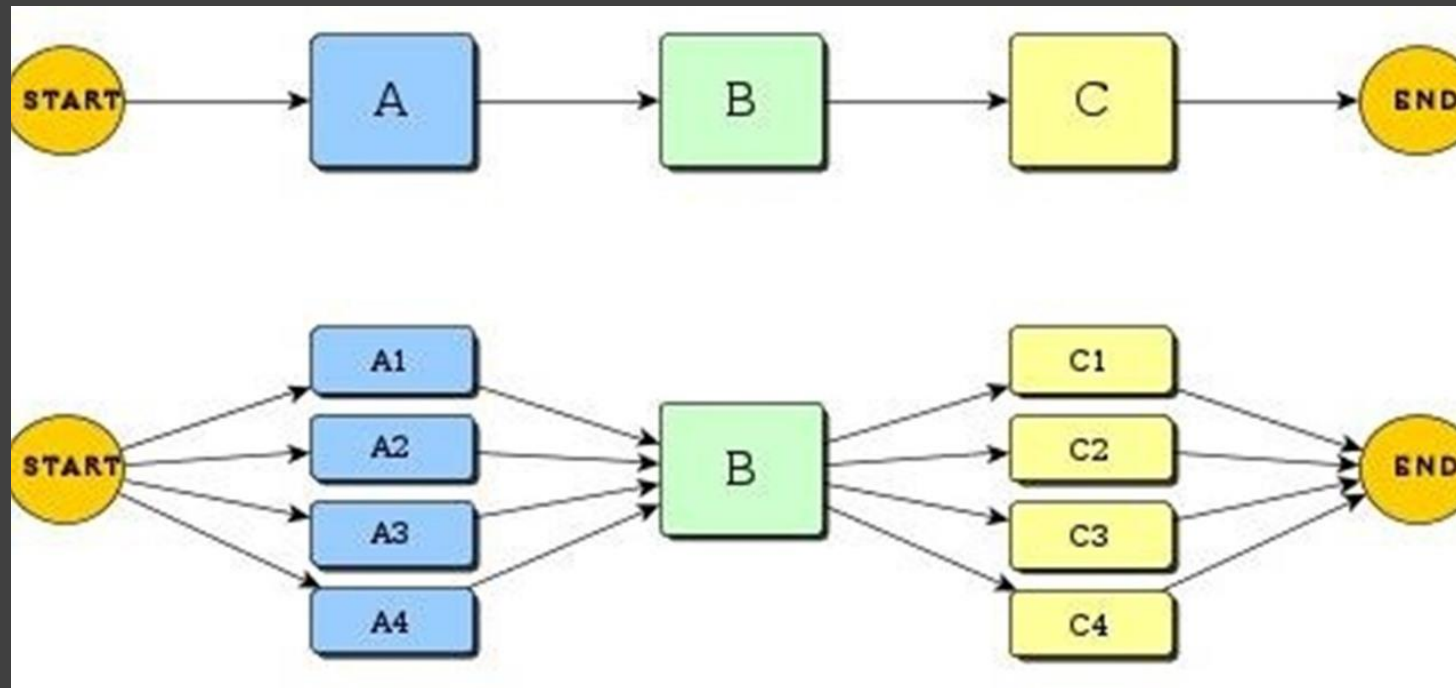


# Computação paralela

---

A grosso modo, utiliza-se um conjunto de CPUs (Central Processing unit) interligadas (comunicação rápida) para se dividir uma dada tarefa computacional

Serial



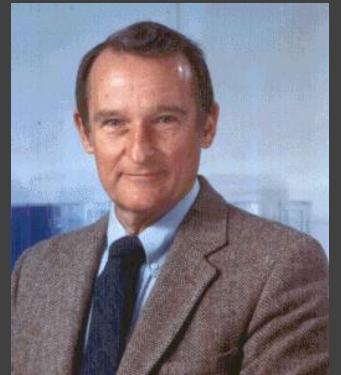
Paralelo

## Arquitetura: ponto de vista histórico

---

Historicamente o conceito de computação paralela é inicialmente empregado em supercomputadores, que eram destinados a solução de problemas específicos. Um dos mais conhecidos eram os desenvolvidos pela Cray (em alusão a Seymour Cray)

Cray 1  
(Los Alamos  
National lab) - 1975





## Arquitetura: ponto de vista histórico

---

Atualmente são fabricados por algumas empresas (Atos, Cray/HPE, IBM, etc.), com as mais variadas configurações e níveis de otimização:

Fujitsu- Fugaku (RIKEN, Japão)



IBM – Summit (Oak Ridge National lab, EUA)





# Supercomputadores Nacionais

---

Santos Dumont - Laboratório Nacional de Computação Científica (LNCC)



Mais informações em:  
<https://sdumont.lncc.br/>

# Supercomputadores Nacionais

---

## Santos Dumont - LNCC



O SDumont possui um total de 36.472 núcleos de CPU, distribuídos em 1.134 nós computacionais, dos quais são compostos, na sua maioria, exclusivamente por CPUs com arquitetura multi-core. Há, no entanto, quantidade adicional significativa de nós que, além das mesmas CPUs multi-core, contém tipos de dispositivos com a chamada arquitetura many-core: GPU e MIC. O SDumont é dotado de um nó diferenciado, o MESCA2, com número elevado de núcleos (240) e arquitetura de memória compartilhada de grande capacidade (6 Tb em um único espaço de endereçamento). Além disso, existe um nó especialmente projetado para aplicações de Inteligência Artificial (Deep Learning) que dispõe de 8 GPUs NVIDIA Tesla V100-16Gb com Nvlink, totalizando 40.960 CUDA-core e 5120 Tensor-core. Uma

<https://sdumont.lncc.br/machine.php?pg=machine#>

# Supercomputadores Nacionais

---

Santos Dumont - LNCC



LCC-UFMG



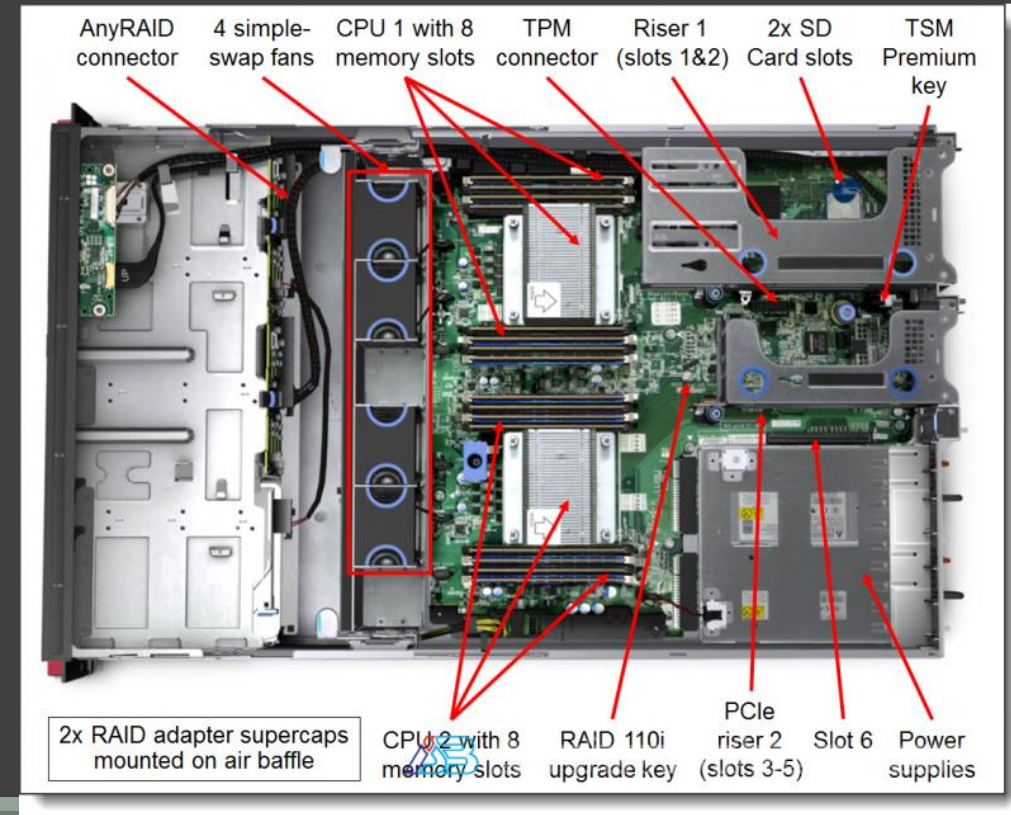


## Arquitetura: ponto de vista histórico

Compostos por vários sublocos (nós) que possuem em geral mais de um processador em uma mesma placa mãe:



Servidor Lenovo



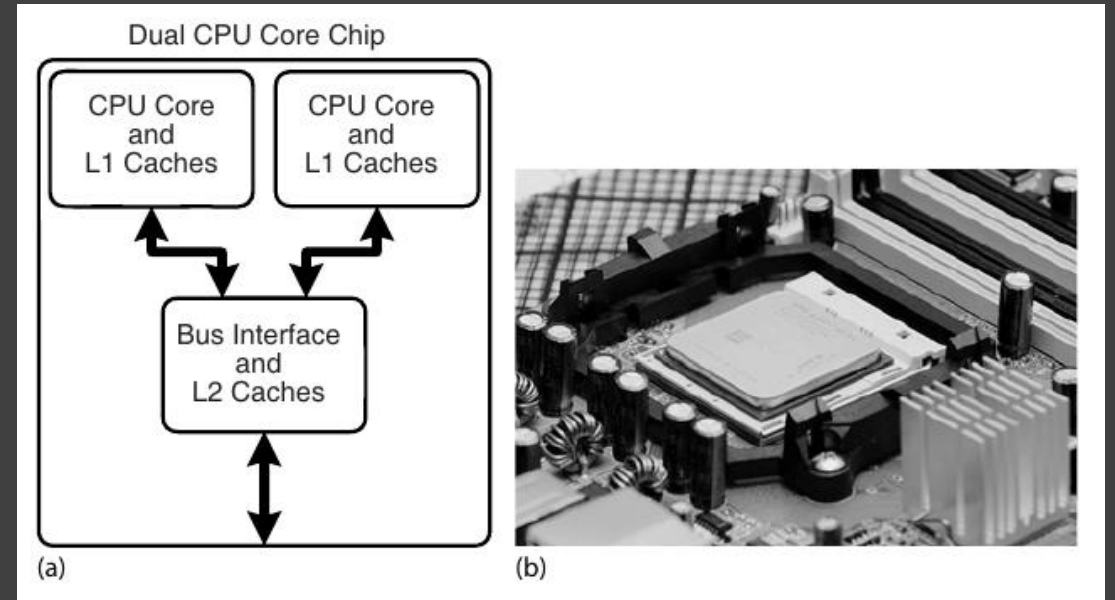
# Arquitetura: atualmente PC's

---

Atualmente os processadores multicore, presentes em PC's, podem também serem vistos como máquinas para processamento paralelo:



## Dual core



# Arquitetura: atualmente PC's

---

## Intel i9-9900KF



Processador Intel® Core™ i9-9900K

Cache de 16M, até 5,00 GHz

### Especificações da CPU

Número de núcleos ?	8
Nº de threads ?	16
Frequência turbo max ?	5.00 GHz
Tecnologia Intel® Turbo Boost frequência 2.0† ?	5.00 GHz
Frequência baseada em processador ?	3.60 GHz
Cache ?	16 MB Intel® Smart Cache
Velocidade do barramento ?	8 GT/s
TDP ?	95 W

# Classificação

---

Podemos classificar os diferentes tipos de paralelismo da seguinte forma:

1. Shared memory systems : Sistemas com múltiplas unidades de processamento atreladas a uma única unidade de memória. Ex.: PC's
2. Distributed systems: Sistemas que consistem de várias unidades de processamento, cada uma com sua unidade de memória e CPU.
3. Graphic processor units: unidades gráficas utilizadas como co-processadores para se resolver problemas numéricos.

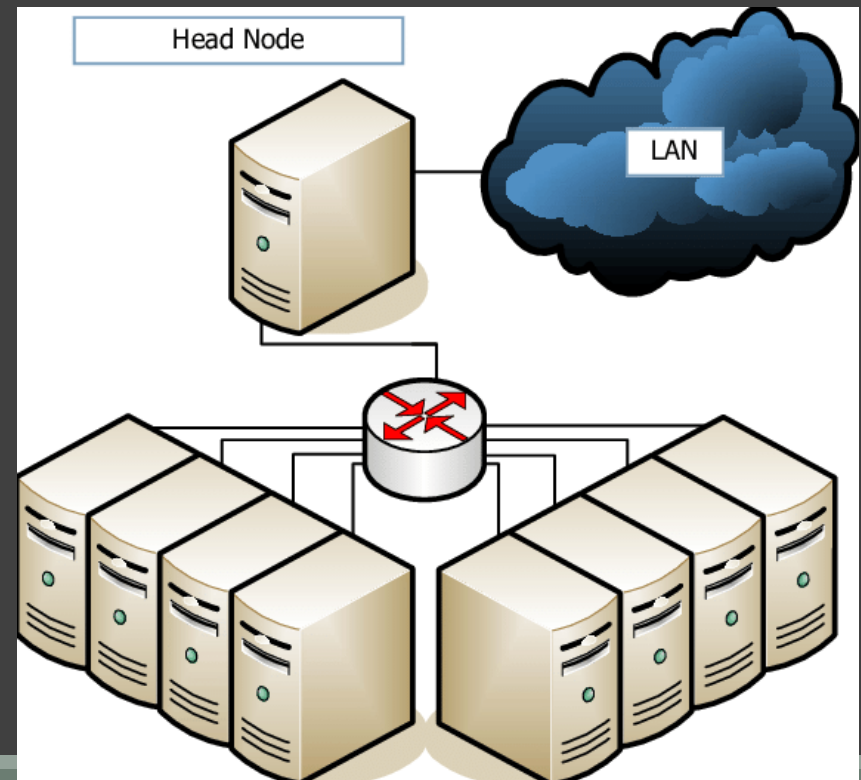


# Distributed systems

---

Exemplo de distributed systems são clusters para computação paralela:

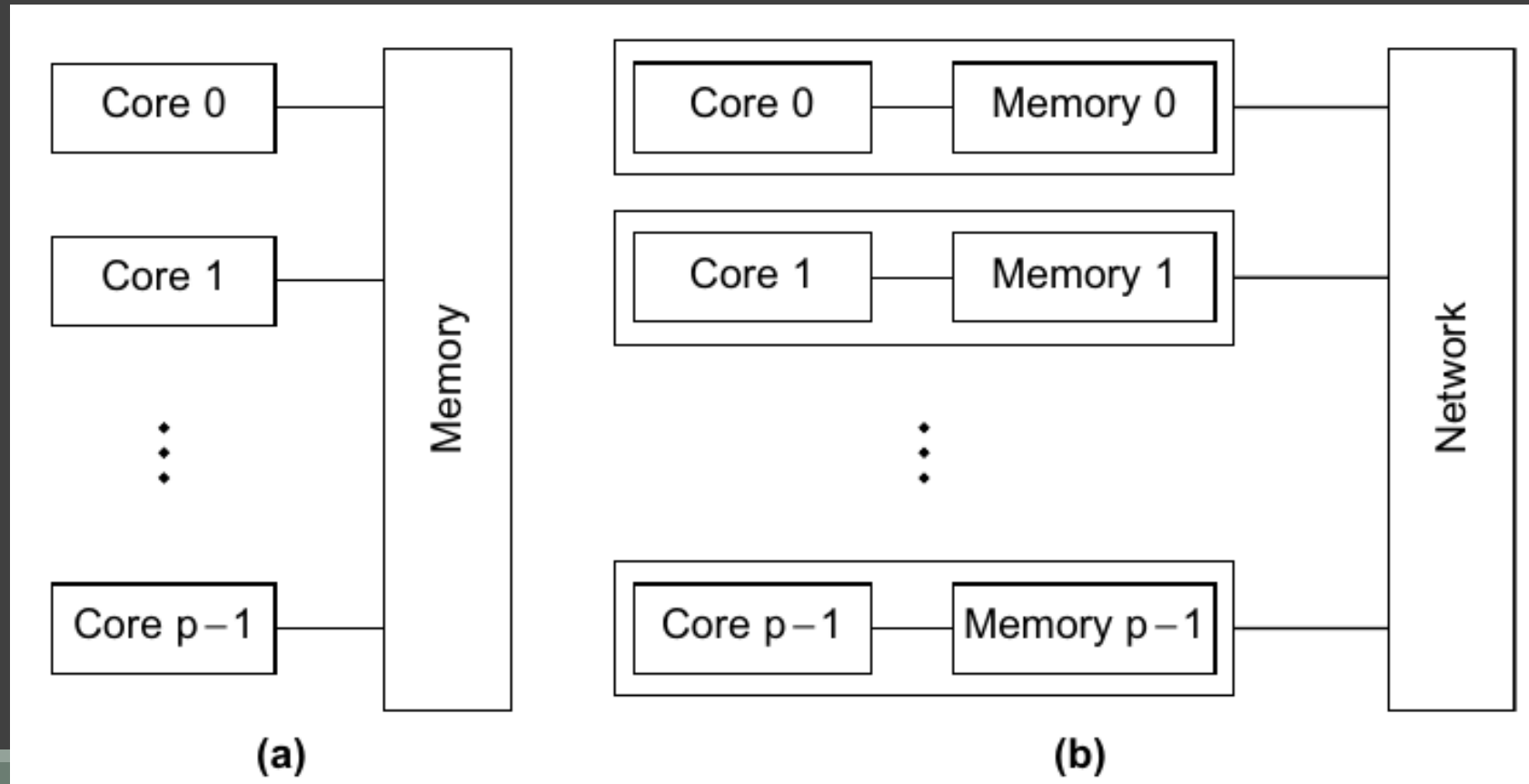
Bewolf cluster





# Shared-memory vs. distributed systems

Diferentes tipos de máquinas de processamento paralelo:



# Performance

---

Para entender melhor a performance e outros conceitos é necessário definir algumas outras grandezas. Vamos considerar que executamos um programa com  $p$  cores, e que o tempo de processamento serial seja  $T_{serial}$  e paralelo  $T_{paralelo}$ . Assim sendo, no caso idealizado

$$T_{paralelo}^{ideal} = \frac{T_{serial}}{p}$$

Limitações associadas as partes seriais , ao acesso de memória e comunicação entre cpus, introduz gargalos.

Definimos o speedup (S) :

$$S = \frac{T_{serial}}{T_{paralelo}}$$

No caso ideal:

$$S^{ideal} = \frac{pT_{serial}}{T_{serial}} = p$$

Speedup linear

## Eficiência e Speedup

---

Eficiência:

$$E = \frac{S}{p} = \frac{T_{serial}}{pT_{paralelo}}$$

No caso ideal:

$$E^{ideal} = \frac{S^{ideal}}{p} = 1$$

Podemos escrever:  $E \leq 1$

Sendo agora **b** a fração do programa que é paralelizada. Com isso, a lei de Amdahl estabelece que o speedup máximo para um processo em  $p$  cores:

$$S(p) = \frac{1}{(1 - b) + \frac{b}{p}}$$

Speedup máximo

(1-b): Fração serial do programa

# Speedup e lei de Amdahl

## Lei de Amdahl

$$S(p) = \frac{1}{(1 - b) + \frac{b}{p}}$$

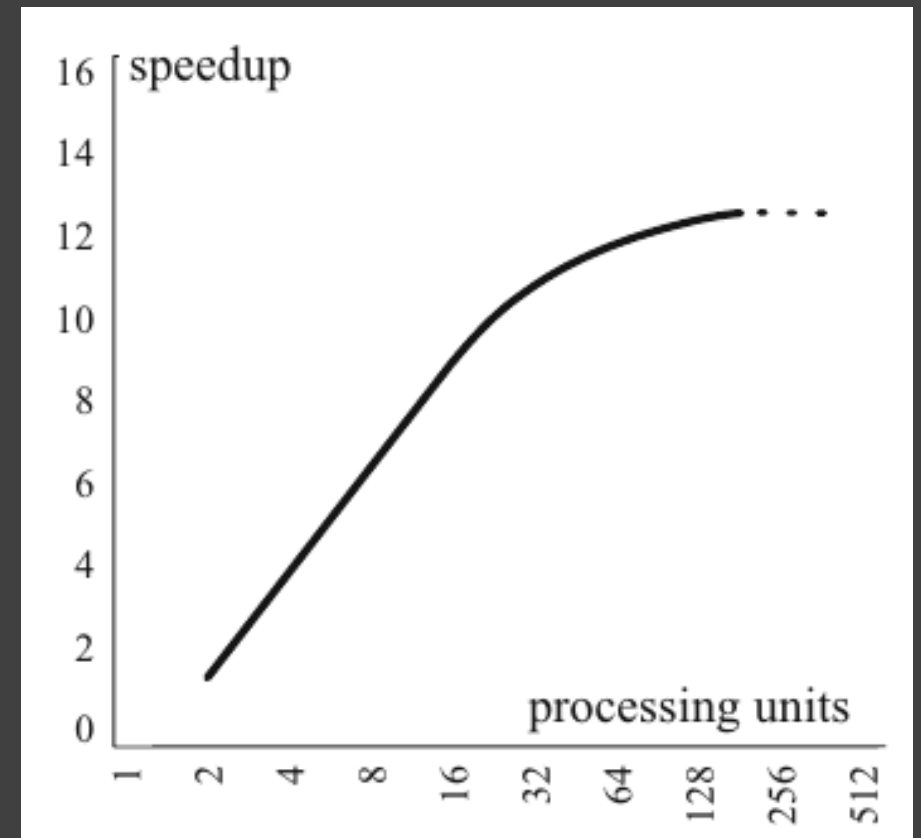
Ex.:  $b = 0.7$  e  $(1-b) = 0.3$ ;

Speedup máximo:

$$S = \frac{1}{0.3 + \frac{17}{16}} = 2.91$$

$$S = \frac{1}{0.3 + \frac{17}{32}} = 3.11$$

$$S = \frac{1}{0.3 + \frac{17}{128}} = 3.27$$



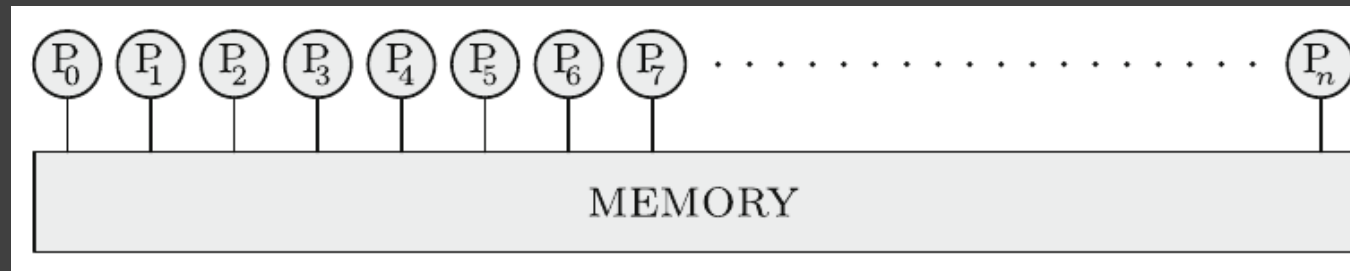
## Interface para programação paralela (OpenMP)

---

## Modelos de programação paralela

---

Para máquinas de memória compartilhada (shared memory), a ideia é utilizar os diferentes cores como cpus distintas acopladas a uma única memória



Podemos separar a execução de um dado código também nas tarefas(threads) do sistema (processador).

Para programação utilizando essas vantagens, iremos fazer uso de uma application programming interface (API) conhecida como OpenMP;

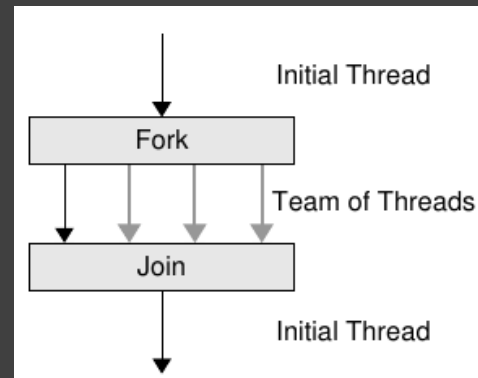
# OpenMP

---

OpenMP consiste em um conjunto de instruções (diretivas) que permitem ao usuário definir quais operações serão executadas em paralelo e como a distribuição entre as tarefas será feita.

Não é uma nova linguagem de programação, mas sim uma interface feita para funcionar em conjunto com compiladores para linguagens C/C++ e Fortran.

Ideia:



Mais informações em: <https://www.openmp.org/>

## Pragmas e instruções básicas

---

Em C/C++ as instruções OpenMP, a parte paralela fica sendo escrita com o uso de `#pragma` no código fonte.

`#pragma omp parallel` : início da parte paralela, criação das threads

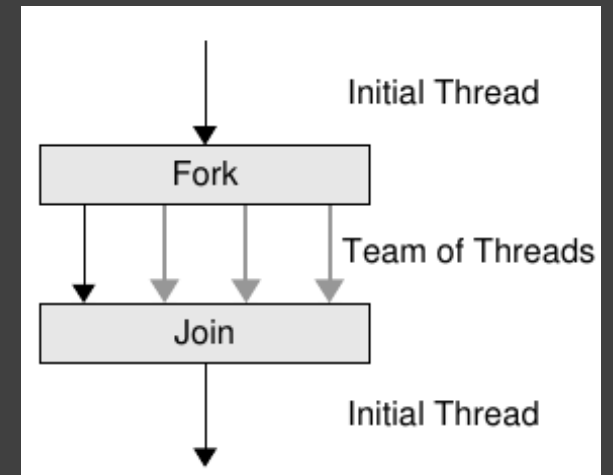
O que fica dentro do pragma acima fica sendo como um bloco que será executado por cada thread. Fica implícito uma "barreira" no final , onde todos threads terminam a execução e o resultado é gerado pelo thread master

Outras instruções básicas:

`omp_get_max_threads()` : obtêm o número máximo de threads;

`omp_get_thread_num()`: obtêm o número (ID) de uma dada thread;

`omp_get_num_threads()`: obtêm o número de threads ;

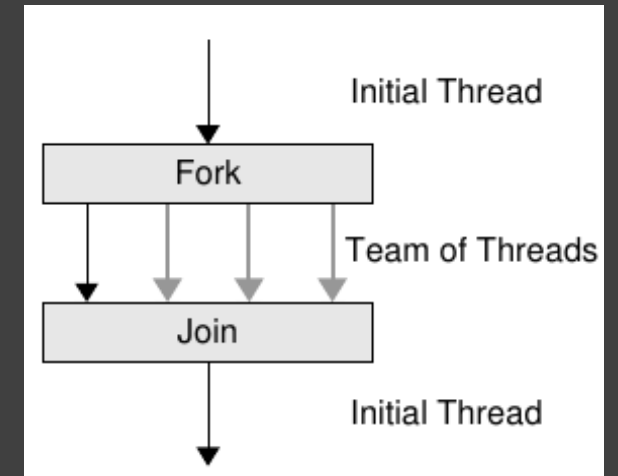




# Exemplo (Olá mundo!)

```
#include <iostream>
#include <omp.h> // incluindo suporte ao OpenMP

int main(int argc, char* argv[])
{
    printf("Olá mundo: ");
    /* Inicio bloco paralelo */
    #pragma omp parallel
        /* Obtem o ID de cada thread */
        printf(" %d ", omp_get_thread_num());
    printf("\n");
}
```



Saída:

>> Olá mundo: 0 1 7 4 5 6 3 2

Compilação GNU compiler: `g++ -fopenmp -o exemplo1.x exemplo.cpp`

# Paralelização de loops

---

Uma das principais vantagens das paralelizações é a aplicação em loops, que são blocos comuns de vários dos programas para solução de problemas científicos.

Vamos tomar como exemplo inicial a escrita de números inteiros até um valor máximo max:

```
#include <iostream>
#include <omp.h>
int main(int argc, char* argv[])
{
    int max = 15;
    /* Inicio do loop paralelo */
    #pragma omp parallel for
    /* Loop a ser executado */
    for (int i = 1; i <= max; i++)
    {
        printf(" %d: %d\n ", omp_get_thread_num(), i);
    }
    return 0;
}
```

Saída:

0: 1	3: 7
0: 2	3: 8
4: 9	6: 13
4: 10	6: 14
5: 11	1: 3
5: 12	1: 4
7: 15	
2: 5	
2: 6	

# Paralelização de loops

---

```
#include <iostream>
#include <omp.h>
int main(int argc, char* argv[])
{
    int max = 15;
    /* Inicio do loop paralelo */
    #pragma omp parallel for
    /* Loop a ser executado */
    for (int i =1; i <= max; i++)
    {
        printf(" %d: %d\n ", omp_get_thread_num(), i);
    }
    return 0;
}
```

Cria-se um grupo de threads (uma sendo a master);

As iterações do loop paralelo são divididas entre as threads;

Cada thread executa o que foi atribuído a ela.

Uma vez que as iterações terminam , todas threads são sincronizadas (barreira implícita no final) e todas threads escravas são finalizadas.

## Exemplo: Multiplicação matricial

---

Através da paralelização de loops, podemos paralelizar operações como a multiplicação de matrizes quadradas;

Vale ressaltar que nesses casos as subdivisões das threads são feitas internamente

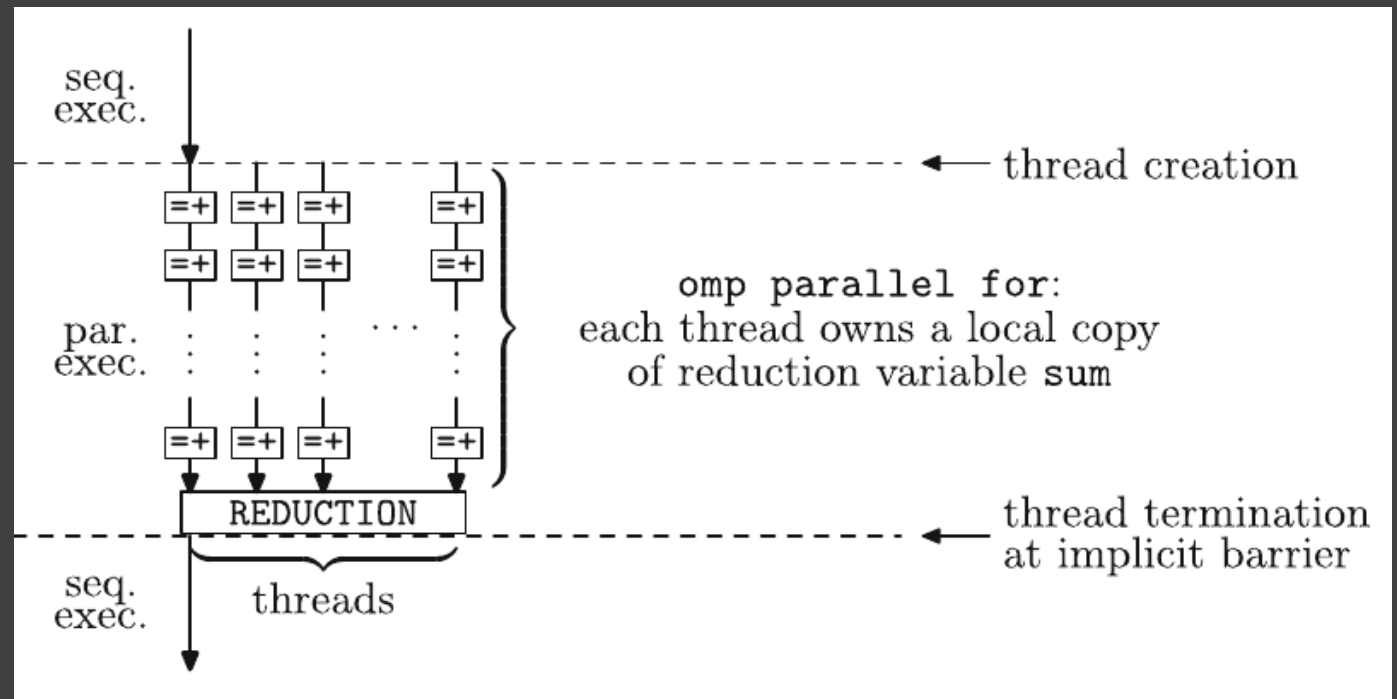
```
#include <iostream>
#include <omp.h>
using namespace std;

void mtzMult(double am[][3], double bm[][3], double cm[][3],
int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        #pragma omp parallel for
        for (int j = 0; j < n; j++)
        {
            for (int k = 0; k < n; k++)
            {
                cm[i][j] = cm[i][j] + (am[i][k]*bm[k][j]);
            }
        }
    }
}
```

## OpenMP: reduction

Uma outra instrução importante consiste no loop paralelo com a instrução reduction;  
Cada thread irá fazer sua parte na soma, que ao final será coletada uma única vez.

Faz uso de variáveis privadas;

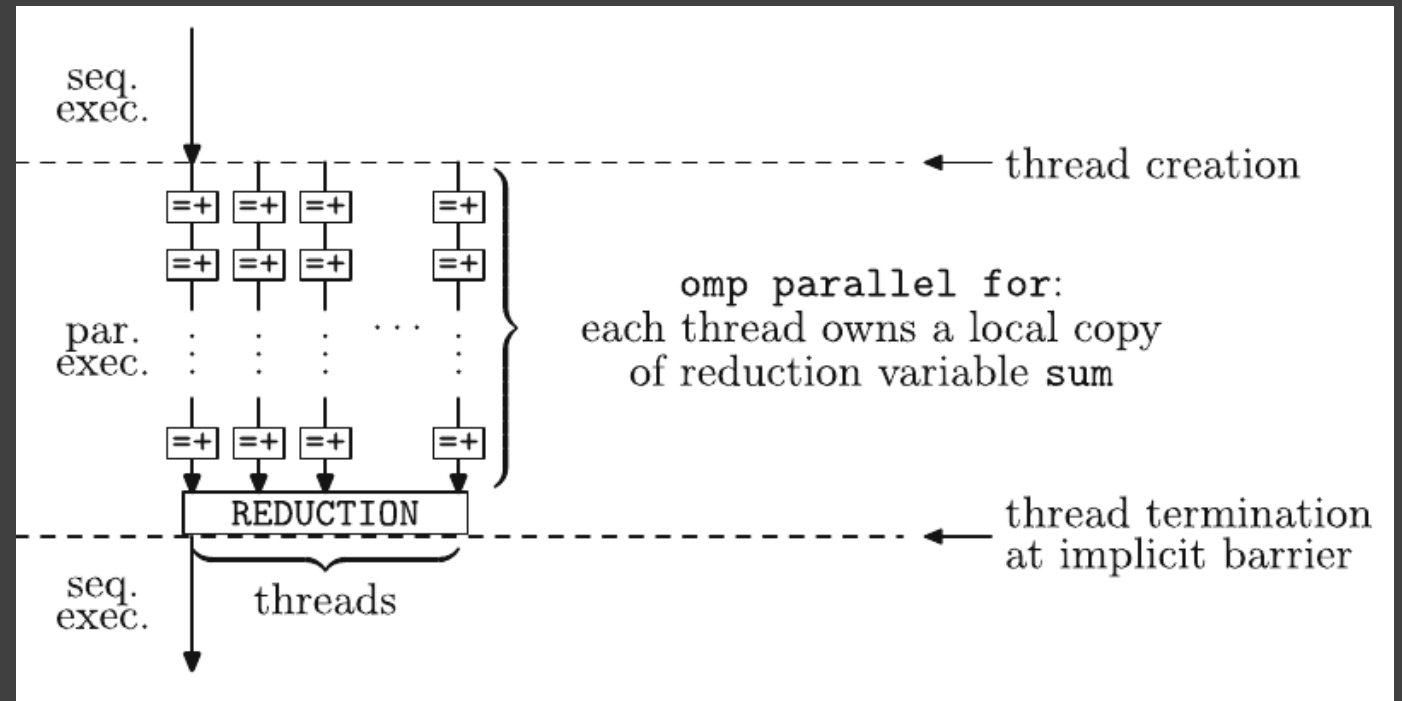


# Exemplo: Soma de números números inteiros

Seja a soma de inteiros em um intervalo até o inteiro max;

```
#include <iostream>
#include <omp.h>
using namespace std;

int main(int argc, char* argv[])
{
    int max = 8;
    int sum = 0;
    #pragma omp parallel for reduction (+:sum)
    for (int i = 1; i <= max; i++)
    {
        sum = sum + i;
    }
    cout << "Valor da soma = " << sum << "\n";
}
```



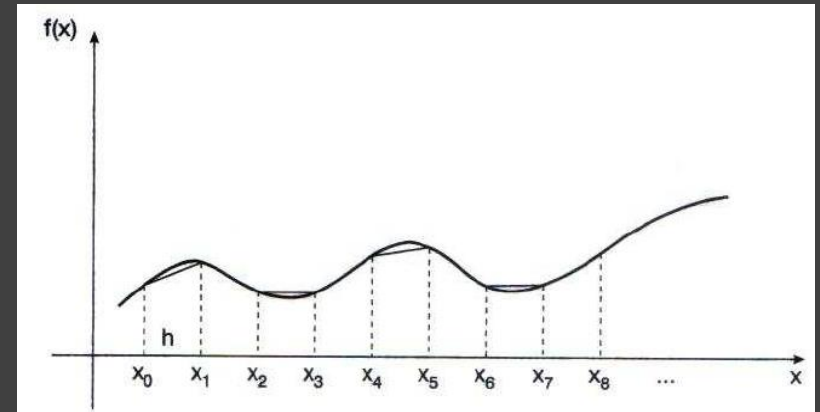
## Exemplo: A regra dos trapézios repetida

---

Lembrando que

$$h = x_{i+1} - x_i$$

$$\int_a^b f(x)dx = \sum_{i=0}^{m-1} \int_{x_i}^{x_{i+1}} f(x)dx \approx \sum_{i=0}^{m-1} \left[ \frac{h}{2} (f(x_i) + f(x_{i+1})) \right]$$



Na aula anterior implementamos a versão serial do programa para

$$\int_0^{\pi/4} e^{3x} \operatorname{sen}(2x) dx$$

## Exemplo: A regra dos trapézios repetida

---

Código serial em C++

$$\int_0^{\pi/4} e^{3x} \text{sen}(2x) dx$$

```
#include <iostream>
```

```
#include <math.h>
```

```
#define pi 3.14159265
```

```
#define fun(x) exp(3*x)*sin(2*x)
```

```
using namespace std;
```

```
int main(){
```

```
    double x0 = 0.0;
```

```
    double x1 = pi/4.0;
```

```
    int N = 100;
```

```
    double h = (x1-x0)/N;
```

```
    cout << "h = " << h << "\n";
```

```
    double integral = fun(x0)+fun(x1);
```

```
    for (int i = 1; i <= N-1; i++)
```

```
    {
```

```
        double k = x0 + (i*h);
```

```
        integral = integral + (2*fun(k));
```

```
    }
```

```
    integral = (h/2.0)*integral;
```

```
    cout << "Valor da integral = " <<  
    integral << "\n";
```

```
}
```



# Performance

---

Serial (python) :  $t = 3\text{min} + 1.13\text{ segundos}$

Serial (C++):  $t = 1.91\text{ segundos}$

Paralelo (C++ / OpenMP):

# Threads	Tempo (segundos)
1	1.924
2	1.002
3	0.689
4	0.515
5	0.416
6	0.351
7	0.299
8	0.262

