

# Supercomputação

Luciano Soares

Igor Montagner

# Aula 12

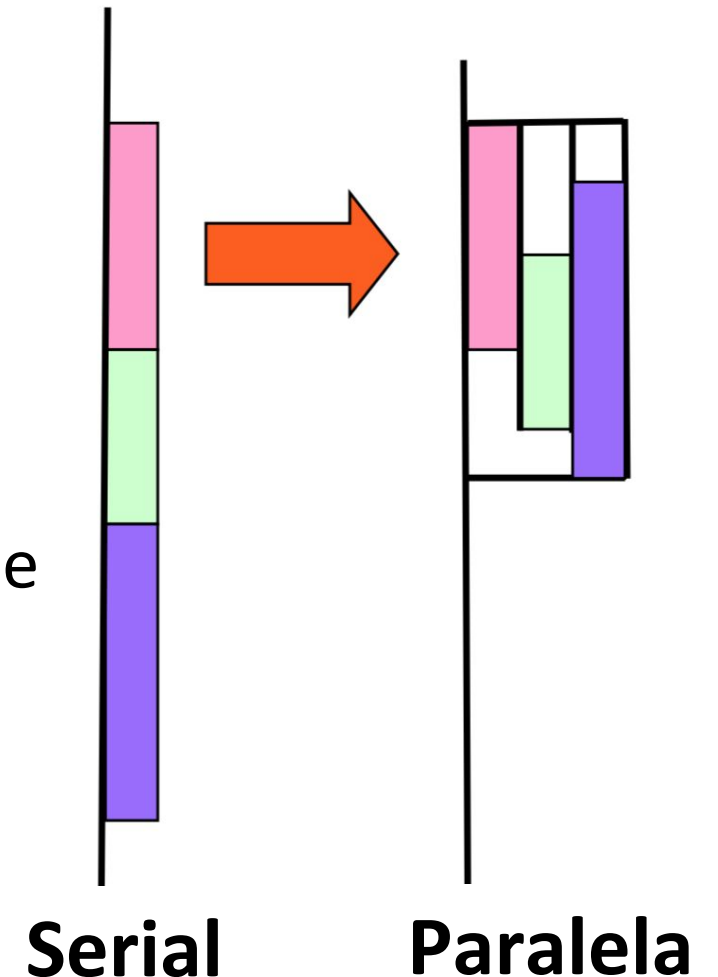
## Tarefas em Ambientes Paralelos

# Objetivos de Aprendizagem

- Lançar tarefas em ambientes paralelos
- Desenvolver algoritmos de divisão e conquista
- Avaliar o custo relativo ao número de tarefas disparada

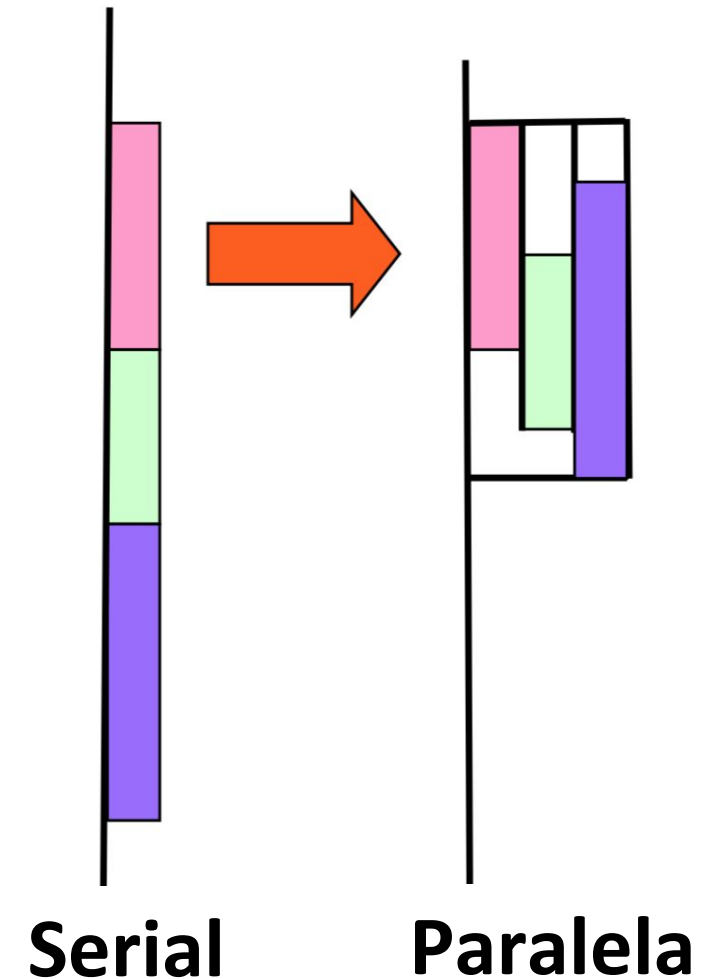
# Tarefas em Programação Paralela

- As tarefas (*tasks*) são unidades de trabalho independentes
- As tarefas são compostas por:
  - código para executar
  - dados para calcular
- Threads são atribuídas para executar o trabalho de cada tarefa (*tasks*).
  - A thread que encontrar a tarefa pode executar ela imediatamente.
  - As threads podem adiar a execução para mais tarde.



# Como são as tarefas (*tasks*)

- A tarefa é definida em um bloco estruturado de código
- Dentro de uma região paralela, a thread que encontra uma construção de tarefa irá empacotar todo o bloco de código e seus dados para execução
- Tarefas podem ser aninhadas: isto é, uma tarefa pode gerar novas tarefas



# Diretivas para Tarefas (Task)

`#pragma omp task[clauses]`

```
#pragma omp parallel  
{  
  #pragma omp master  
  {  
    #pragma omp task  
    huguinho();  
    #pragma omp task  
    zezinho();  
    #pragma omp task  
    luizinho();  
  }  
}
```

Crie um conjunto de threads

Thread 0 organiza as tarefas

Tarefas executadas por alguma thread em alguma ordem

Todas as tarefas devem ser concluídas antes que esta barreira

# Atividade 1: Tarefas Simples

- Escreva um programa, usando tarefas (tasks), que irá aleatoriamente gerar uma das duas cadeias de caracteres:
  - I think race cars are fun
  - I think car races are fun
- Dica: use tarefas para imprimir a parte indeterminada da saída (ou seja, as palavras "race" ou "cars").
- Isso é chamado de "Condição da corrida". Ocorre quando o resultado de um programa depende de como o sistema operacional escalona as threads.
- Você pode usar:
  - #pragma omp parallel
  - #pragma omp task
  - #pragma omp master
  - #pragma omp single

# Quando as tarefas finalizam?

- Nas barreiras das threads (explícitas ou implícitas)
  - aplica-se a todas as tarefas (tasks) geradas na região paralela até a barreira
- Na diretriz taskwait
  - aguarde até que todas as tarefas disparadas na tarefa atual terminem.

#pragma omp taskwait

- Nota: aplica-se apenas a tarefas geradas na tarefa atual, e não a "descendentes".
- No final de uma região do grupo de tarefas

# pragma omp taskgroup


  - aguarde até que todas as tarefas criadas no grupo de tarefas tenham concluído, inclusive os "descendentes"



# Exemplo com taskwait

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        huguinho();
        #pragma omp task
        zezinho();
        #pragma omp taskwait
        #pragma omp task
        luizinho();
    }
}
```

huguinho() e zezinho()  
precisam completar antes  
de iniciar luizinho()



# Exemplo de traversal de uma lista ligada

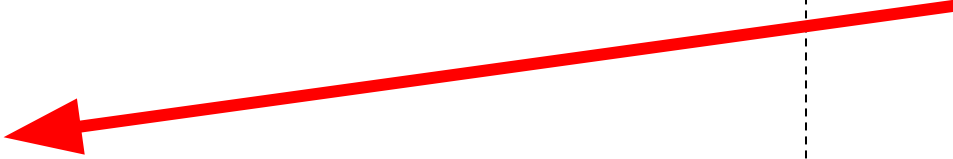
```
p = listhead;  
while(p) {  
    process(p);  
    p = next(p);  
}
```

- Percurso clássico de lista ligada
- Em cada item da lista faz alguma coisa
- Assume que os itens podem ser processados de forma independente
- Não é possível usar uma diretiva OpenMP de loop

# Traversal de uma lista ligada em paralelo

```
#pragma omp parallel
{
    #pragma omp master
    {
        p = listhead;
        while(p) {
            #pragma omp task firstprivate(p)
            {
                process(p);
            }
            p = next(p);
        }
    }
}
```

Somente uma thread que organiza e cria as tarefas



faz uma cópia de p quando a tarefa é empacotada



# Traversal de uma lista ligada em paralelo

Thread 0:

```
p = listhead;
while (p) {
    <package up task>
    p=next (p);
}

while (tasks_to_do) {
    <execute task>
}

<barrier>
```


outras threads:

```
while (tasks_to_do) {
    <execute task>
}

<barrier>
```

# Traversal de múltiplas listas ligadas em paralelo

```
#pragma omp parallel
{
    #pragma omp for private(p)
    for(int i=0; i<numlists; i++) {
        p = listheads[i];
        while(p) {
            #pragma omp task firstprivate(p)
            {
                process(p);
            }
            p=next(p);
        }
    }
}
```



Loop em paralelo para  
empacotar as tarefas

# Padrões para Tarefas

- O comportamento padrão desejado para o compartilhamento das variáveis das tarefas geralmente é firstprivate, porque a tarefa pode demorar a ser executada (e as variáveis podem ter saído do escopo)
  - Variáveis que são privadas quando a construção da tarefa é encontrada viram firstprivate por padrão

```
#pragma omp parallel shared(A) private(B)
{
    ...
    #pragma omp task
    {
        int C;
        compute(A,B,C);
    }
}
```



A é shared  
B é firstprivate  
C é private

# Atividade 2: Fibonacci

Rode o seguinte código e depois faça ele paralelo por tarefas:

```
#include <iostream>

int fib(int n) {
    int x,y;
    if(n<2) return n;
    x=fib(n-1);
    y=fib(n-2);
    return(x+y);
}

int main() {
    int NW=45;
    int f=fib(NW);
    std::cout << f << std::endl;
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Muito ineficiente  $O(2^n)$  !

**Use recursos como:**

```
#pragma omp task
#pragma omp taskwait
#pragma omp parallel
#pragma omp master
```

# Cuidado: Task switching

- Mudança de execução de tarefas são lentas
  - Muitas tarefas geradas podem deixar o código mais lento
  - Ao gerar tarefa o sistema terá que suspender a execução por um tempo

```
#pragma omp single
{
    for(i=0;i<UMZILHAO;i++)
        #pragma omp task
            process(item[i]);
}
```



# Dependência de Tarefas (Tasks)

- Em certas situações o resultado de uma tarefa pode ser necessário para outra tarefa. Para isso as tasks em OpenMP tem a clausula *depend* que gerencia esse dependência:
- Exemplo
  - As duas primeiras tarefas podem ser executadas em paralelo
  - A terceira tarefa não pode começar até que as duas primeiras sejam completas

```
#pragma omp task depend(out:a)  
{...} // grava em a  
  
#pragma omp task depend(out:b)  
{...} // grava em b  
  
#pragma omp task depend(in:a,b)  
{...} // lê a e b
```

# Usando tarefas (tasks)

- O compartilhamento das variáveis pode ser confuso
  - regras de compartilhamento padrão diferentes de outras construções
  - como de costume, usar `default(none)` é uma boa ideia
- Não use tarefas para coisas já otimizadas pelo OpenMP
  - por exemplo loops for
  - a sobrecarga para executar tarefas acaba sendo maior
- Não espere milagres do tempo de execução
  - melhores resultados geralmente obtidos quando o usuário controla o número e a granularidade das tarefas

# Atividade 3: Pi com tarefas

- Considere o programa "pi\_recur.cpp". Este programa implementa uma versão de algoritmo recursivo do programa para calcular pi
  - Paralelize este programa usando tarefas OpenMP
  - Use recursos como:

```
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp master
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num()
int omp_get_num_threads()
```

# Código do pi recursivo

```
#include <omp.h>
#include <iostream>
static long num_steps = 1024*1024*1024;
#define MIN_BLK 1024*1024*256

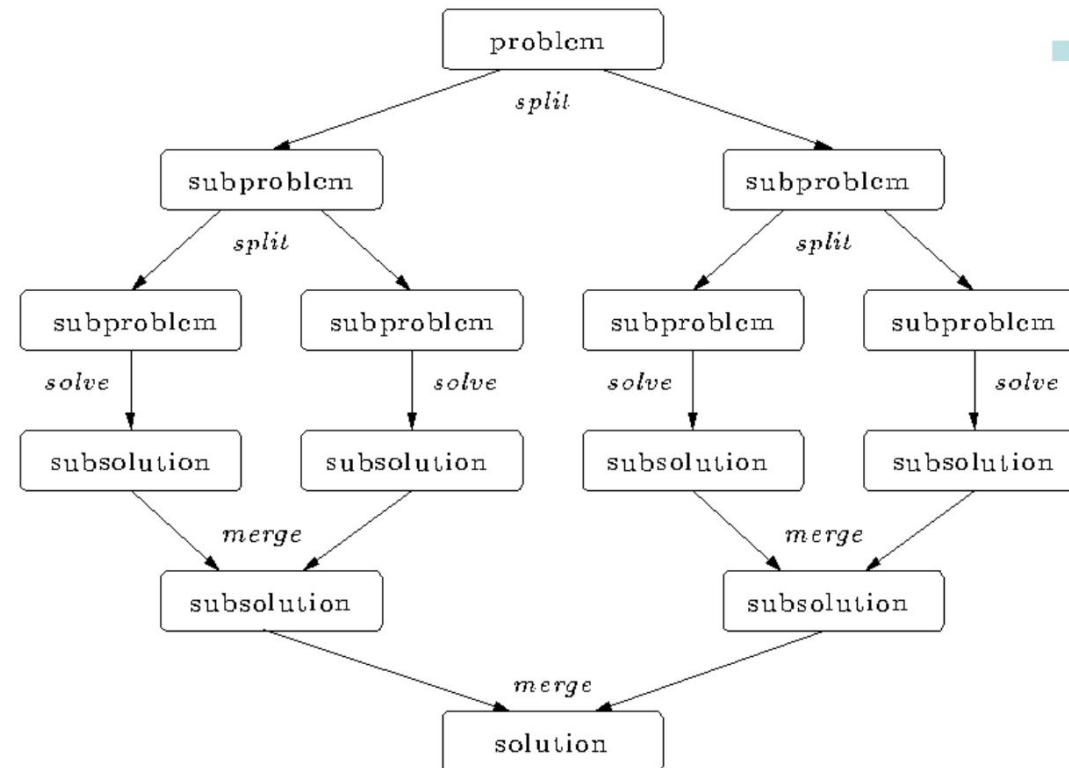
int main () {
    int i;
    double step, pi, sum;
    double init_time, final_time;
    step = 1.0/(double) num_steps;

    init_time = omp_get_wtime();
    sum = pi_comp(0,num_steps,step);
    pi = step * sum;
    final_time = omp_get_wtime() - init_time;
    std::cout << "for " << num_steps << " steps
pi = " << pi << " in " << final_time << "
secs\n";
}
```

```
double pi_comp(int Nstart,int Nfinish,double step) {
    int i,iblk;
    double x, sum = 0.0, sum1, sum2;
    if (Nfinish-Nstart < MIN_BLK){
        for (i=Nstart;i< Nfinish; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    else{
        iblk = Nfinish-Nstart;
        sum1 = pi_comp(Nstart,Nfinish-iblk/2,step);
        sum2 = pi_comp(Nfinish-iblk/2,Nfinish,step);
        sum = sum1 + sum2;
    }
    return sum;
}
```

# Divisão e conquista

- Divida o problema em subproblemas menores; continue até onde os subproblemas podem ser resolvidos diretamente



# Referências

- Livros:

- Hager, G. ; Wellein, G. **Introduction to High Performance Computing for Scientists and Engineers**. 1ª Ed. CRC Press, 2010.

- Artigos:

- Duran, Alejandro, Julita Corbalán, and Eduard Ayguadé. "Evaluation of OpenMP task scheduling strategies." In *International Workshop on OpenMP*, pp. 100-110. Springer, Berlin, Heidelberg, 2008.

- Internet:

- <http://ieeexplore.ieee.org/abstract/document/4553700/>
- <https://en.wikibooks.org/wiki/OpenMP/Tasks>
- <http://openmp.org/wp-content/uploads/sc13.tasking.ruud.pdf>