

SuperComputação

Aula 09 – Tarefas

2020 – Engenharia

Luciano Soares [<lpsoares@insper.edu.br>](mailto:lpsoares@insper.edu.br)

Igor Montagner [<igorsm1@insper.edu.br>](mailto:igorsm1@insper.edu.br)

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:

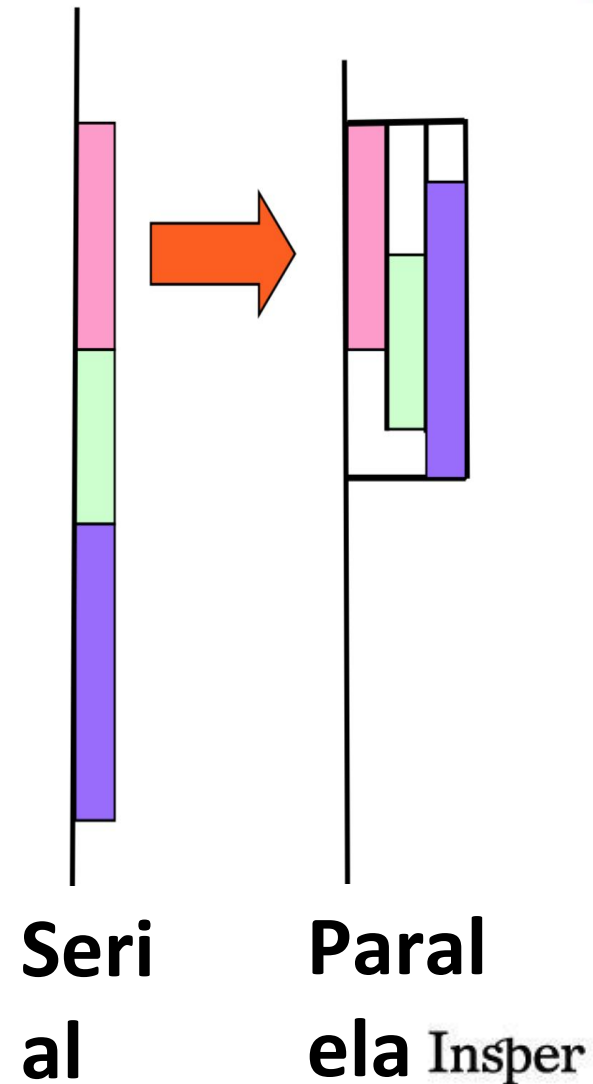
código para executar

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.

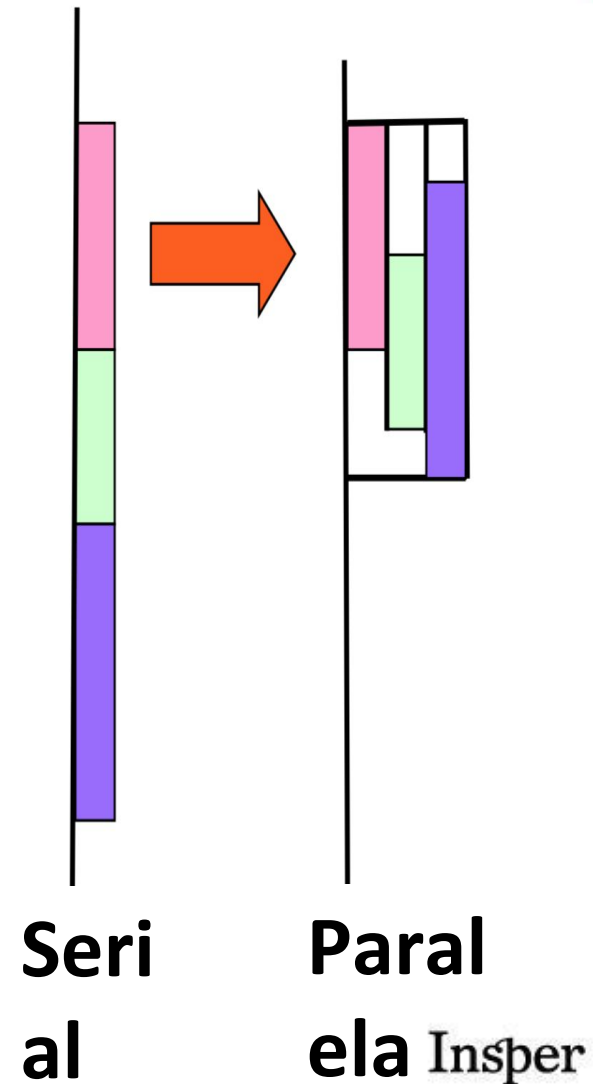


O quê são tarefas?

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



Tarefas em OpenMP

```
#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 seja liberada

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

```
#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 II

```
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

Exemplo II

```
#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

faz uma cópia de p quando
a tarefa é empacotada

Exemplo II - detalhes

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>
```

Exemplo III

```
#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

Tarefas - compartilhamento

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

Cuidado!

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]);
}
```

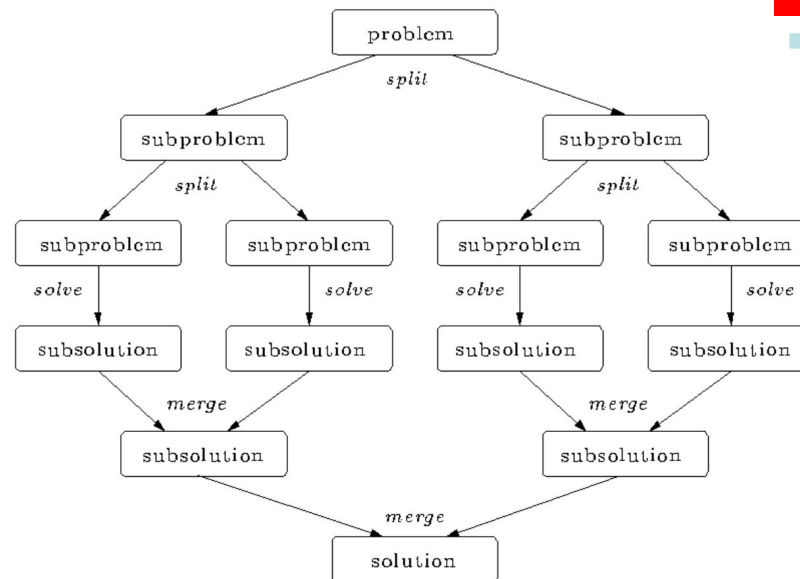
Cuidado! II

- O compartilhamento das variáveis pode ser confuso
 - compartilhamento padrão diferente de outras construções
 - 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

Divisão e conquista

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

Como paralelizar?



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.

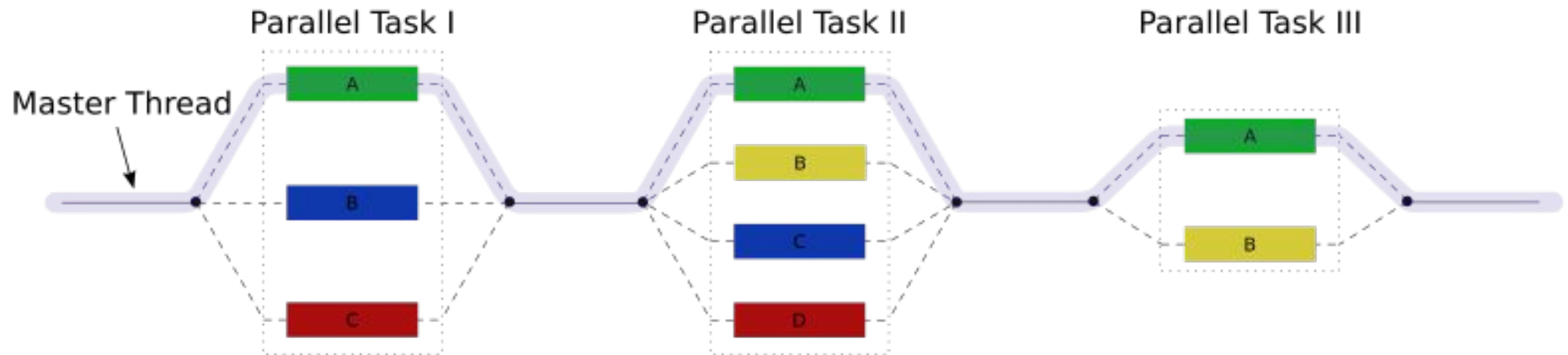
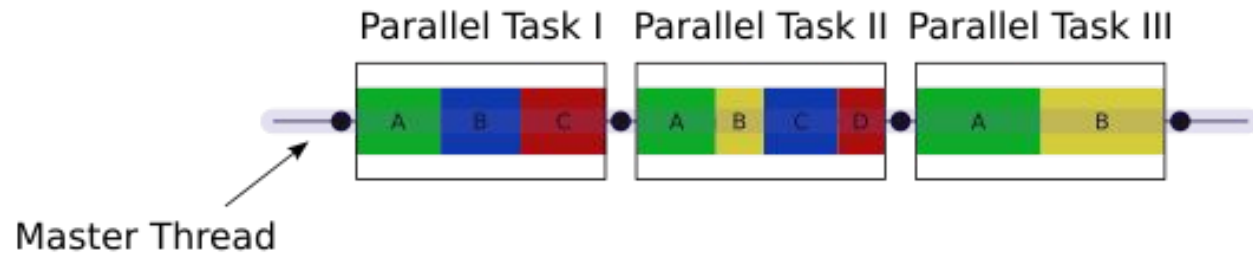
Internet:

<http://ieeexplore.ieee.org/abstract/document/4553700/>

<https://en.wikibooks.org/wiki/OpenMP/Tasks>

<http://openmp.org/wp-content/uploads/sc13.tasking.ruud.pdf>

Aulas passadas



Aulas passadas

Modelo fork-join:

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    long nt = omp_get_num_threads();
    double d = 0;
    for (long i = id; i < num_steps; i+=nt) {
        double x = (i-0.5)*step;
        d += 4.0/(1.0+x*x);
    }
    #pragma omp atomic
    sum += d;
}
```

Aulas passadas

Modelo fork-join:

```
#pragma omp parallel for reduction(+:sum)
for (long i = 0; i < num_steps; i++) {
    double x = (i-0.5)*step;
    sum += 4.0/(1.0+x*x);
}
return sum * step;
```

Hoje

- Comentários *mandel.c*
- Códigos com efeitos colaterais
- Mini-projeto: cálculo do pi usando sorteios aleatórios

Exercício *mandel.c* (solução)

```
# define NPOINTS 1000
# define MAXITER 1000

struct d_complex{
    double r;
    double i;
};

struct d_complex c;
int numoutside = 0;

void testpoint(struct d_complex);

int main(){
    int i, j;
    double area, error, eps = 1.0e-5;
    #pragma omp parallel for default(shared) private(c,j) firstprivate(eps)
        for (i=0; i<NPOINTS; i++) {
            for (j=0; j<NPOINTS; j++) {
                c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
                c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
                testpoint(c);
            }
        }

    area=5.625*(double)(NPOINTS*NPOINTS-numoutside)/(double)(NPOINTS*NPOINTS);
    error=area/(double)NPOINTS;
    printf("Area of Mandlebrot set = %12.8f +/- %12.8f\n",area,error);
    printf("Correct answer should be around 1.510659\n");
}
```

```
void testpoint(struct d_complex c){
    struct d_complex z;
    int iter;
    double temp;
    z=c;
    for (iter=0; iter<MAXITER; iter++){
        temp = (z.r*z.r)-(z.i*z.i)+c.r;
        z.i = z.r*z.i*2+c.i;
        z.r = temp;
        if ((z.r*z.r+z.i*z.i)>4.0) {
            #pragma omp atomic
            numoutside++;
            break;
        }
    }
}
```

Agora as respostas
são constantes e
corretas.

Exercício *mandel.c*

Qual a nota de vocês para

- organização do código?
- facilidade de leitura?
- boas práticas de programação?

Exercício *mandel.c*

Exercício 4 pedia para reorganizar o código.

Isso diminuiu os problemas de paralelização?

Exercício *mandel.c*

Efeitos colaterais: função que lê ou modifica o estado global do programa.

- 1) Seu resultado não depende somente dos argumentos passados;
- 2) A função escreve seus resultados em outros lugares além do seu valor de retorno.

Exercício *mandel.c*

Mundo ideal: nenhuma função modifica o estado global do programa, facilitando muito a paralelização

Mundo real: eliminar todos efeitos colaterais pode tornar o código menos claro, menos eficiente e muito menos legível

Exercício *mandel.c*

Mundo ideal: nenhum estado global do programa, paralelização

Linguagens funcionais

Mundo real: eliminar todos efeitos colaterais
pode tornar o código menos claro, menos
eficiente e muito menos legível

Exercício *mandel.c*

Mundo ideal: nenhuma função modifica o estado global do programa, facilitando muito a paralelização

Mundo real: **diminuir** efeitos colaterais na **parte paralela** do código pode

- 1) evitar problemas de compartilhamento de dados e de concorrência por recursos
- 2) melhorar organização do código

Exercício *mandel.c*

Objetivo é escrever código threadsafe:

manipula dados de modo que nenhuma thread interfira na execução de outra.

- Evitar estado compartilhado/efeitos colaterais
- Utilizar primitivas de sincronização para acessar os dados compartilhados

Atividade prática

- Cálculo do PI usando algoritmo probabilístico
- Diversas opções de paralelização

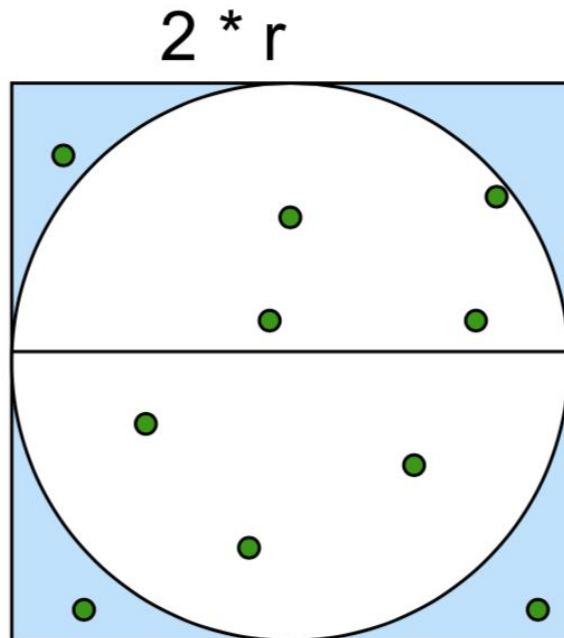
Geração de números aleatórios

- Em um gerador aleatório não é possível prever qual será o próximo número
- Podemos criar geradores pseudo-aleatórios: sequência depende de uma regra conhecida, mas possui propriedades parecidas com uma sequência aleatória de verdade
- A “aleatoriedade” da sequência depende do método e dos parâmetros usados
- Possibilidade de realizar simulações estatísticas

Método de Monte Carlo

Aproximar algum valor baseado em sorteios aleatórios

Exemplo:



Lance dardos aleatoriamente no quadrado
Probabilidade de cair no círculo é proporcional às áreas:

$$A_c = \pi * r^2$$

$$A_q = (2 * r) * (2 * r) = 4 * r^2$$

$$\text{Prob.} = A_c / A_s = \pi / 4$$

$$N = 10 \quad \pi = 2.8$$

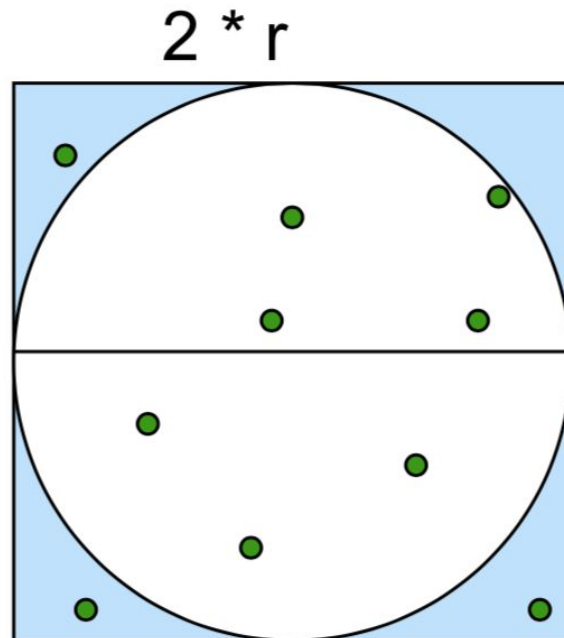
$$N = 100 \quad \pi = 3.16$$

$$N = 1000 \quad \pi = 3.148$$

Método de Monte Carlo

Aproximar algum valor baseado em sorteios aleatórios

Exemplo: Cálculo do PI



Define qualidade da aproximação!

Probabilidade de um ponto
cair dentro do círculo:

Método:

- 1) Sorteia N pontos
- 2) Conta pontos dentro (I) e fora (F)
- 3) $\pi = 4 * (I/F)$

Atividade prática

Cálculo do PI usando algoritmo probabilístico

Objetivo: explorar diferentes estratégias de paralelização

1. exclusão mútua
2. dividir tarefas paralelizáveis e não paralelizáveis
3. transformar em tarefas independentes

Referências

- Livros:
 - Hager, G. ; Wellein, G. **Introduction to High Performance Computing for Scientists and Engineers**. 1ª Ed. CRC Press, 2010.
- Artigos:
 - Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." *IEEE computational science and engineering* 5, no. 1 (1998): 46-55.
- Internet:
 - <https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG>
 - <http://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf>
 - http://extremecomputingtraining.anl.gov/files/2016/08/Mattson_830aug3_HandsOnIntro.pdf

Inspire

www.inspire.edubr