UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ

Programando Multiprocessadores com OpenMP

Prof. João Marcelo Uchôa de Alencar

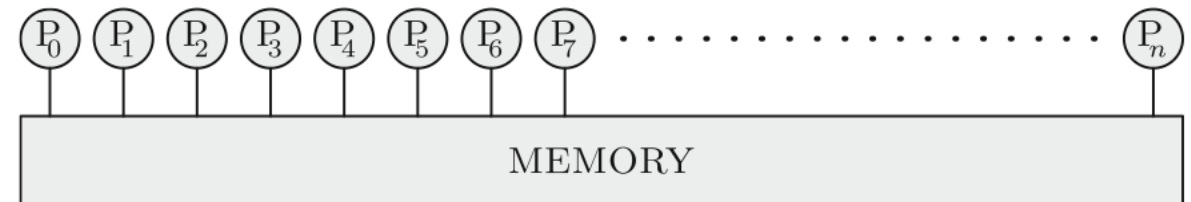
joao.marcelo@ufc.br

Introdução

- Computadores *Multicore* e com Memória Compartilhada.
  - A arquitetura da maioria dos computadores atuais.
  - As máquinas mais fáceis de programar com uma metodologia adequada.
  - OpenMP
    - Interface de programação adequada para criação de aplicações *multithread*.
    - Diretivas de Compilação + Funções de Biblioteca.
    - Foco na expressão do algoritmo paralelo.
- Em outros capítulos veremos:
  - MPI para memória compartilhada.
  - OpenCL para placas aceleradoras.

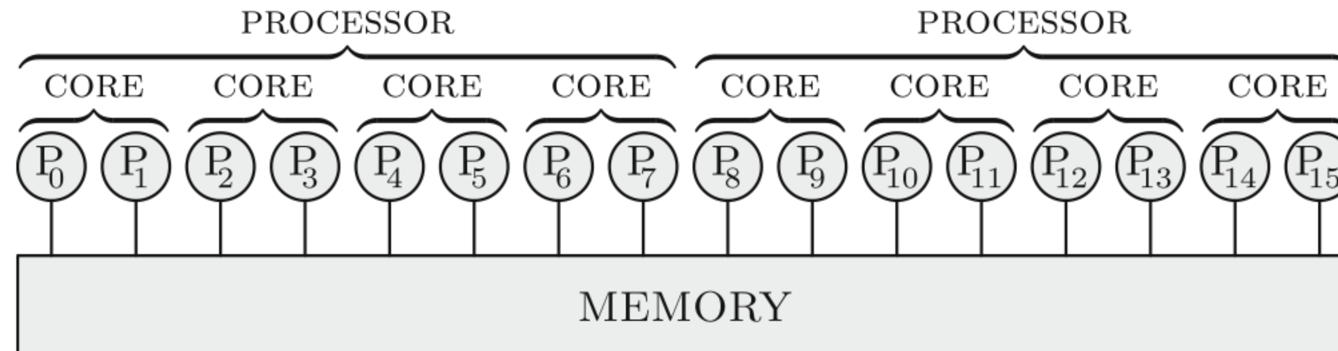
Modelo de Programação em Memória Compartilhada

- Processadores independentes compartilhando uma memória principal.
  - Cada processador pode acessar qualquer endereço.
  - A qualquer momento, diferentes processadores podem executar instruções diferentes em dados diferentes.
  - MIMD – *Multiple Instructions, Multiple Data.*
- É similar ao PRAM e variantes.
  - Mas não há garantia de acesso constante.
  - Mesmo assim, abstraímos detalhes como *cache*.



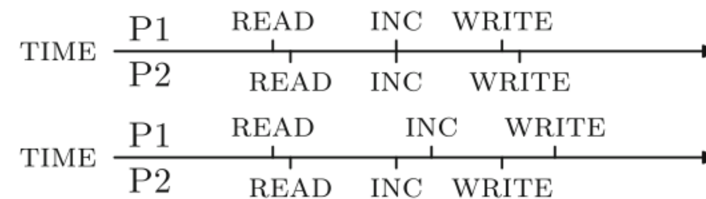
Modelo de Programação em Memória Compartilhada

- CPUs modernas são processadores *multicore*:
  - Cada *core* ou núcleo é uma unidade independente.
  - A duplicação de elementos do *pipeline* do *core* permite o *Simultaneous Multithreading* (SMT).
    - Cada *core* suporta mais de um *thread.*
    - Para o programador fica a ilusão de um *core* lógico.
  - O *core* lógico não chega a ser um *core* físico, pois há limitações de *cache* e barramento, mas facilita a programação pensar assim.

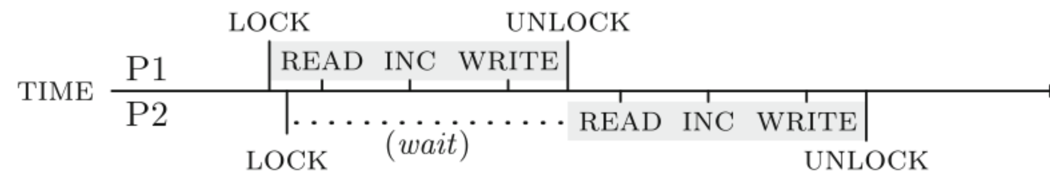


Modelo de Programação em Memória Compartilhada

- O Preço do Acesso Compartilhado.
  - Condição de Corrida: *threads* tentam acessar a mesma região na memória.



  - A solução é *locking* (travamento)
    - Cada *thread* precisa garantir o acesso exclusivo antes de operar na memória desejada.
    - Qualquer outro *thread* é bloqueado.



Modelo de Programação em Memória Compartilhada

- Para garantir a exclusão mútua, precisamos utilizar semáforos ou outras estruturas de bloqueio.
- O desenvolvimento de aplicações complexas se torna muito propenso a falhas de exclusão mútua.

Usando OpenMP para *Multithreading*

- No modelo de memória compartilhada um programa paralelo consiste de vários *threads*.
  - Cada *thread* executa em um *core* lógico.
  - Se há menos *threads* do que *cores,* alguns *cores* serão ociosos.
  - Se há mais *threads* do que *cores*, o sistema operacional aplica técnicas de escalonamento em busca do balanceamento de carga.
- Há várias bibliotecas e *frameworks* para paralelismo.
  - No mundo POSIX/UNIX, a biblioteca pthreads é a alternativa mais comum.
  - Os programas resultantes acabam sendo entulhados de detalhes de baixo nível.
  - Precisamos de algo mais “agradável”.

Usando OpenMP para *Multithreading*

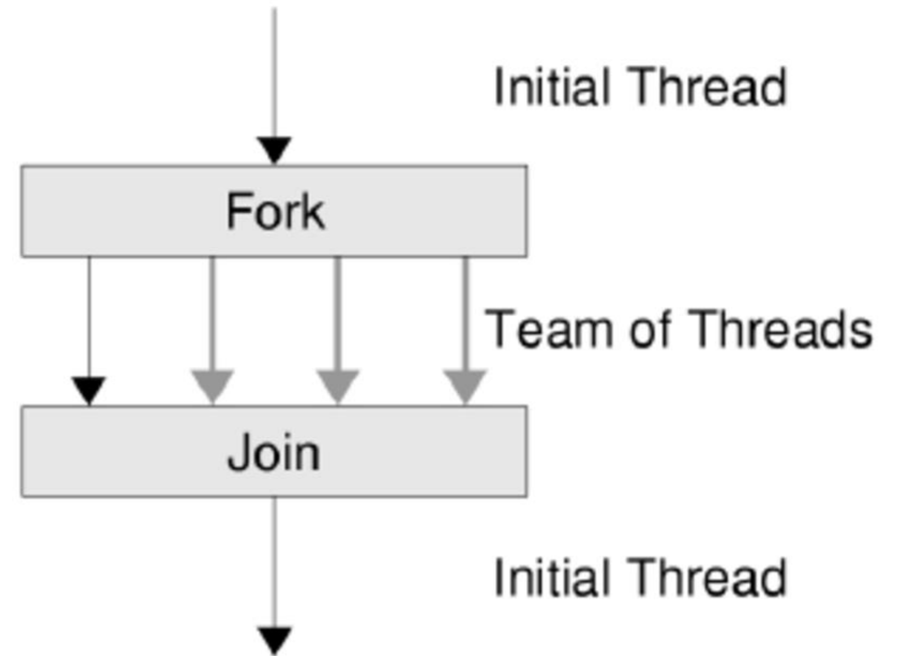
- OpenMP é uma extensão a linguagens existentes.
  - Fortran/C/C++ e outras.
  - A API do OpenMP consiste de :
    - Diretivas de compilação.
    - Funções Auxiliares.
    - Variáveis *shell*.
  - As diretivas informam ao compilador sobre seções do código com paralelismo em potencial e fornecem instruções de como gerar código paralelo.
  - *#pragmas*
  - As funções auxiliares permitem controlar o paralelismo.
  - As variáveis *shell* tem propósito semelhante às funções, mas podem ser alteradas sem exigir recompilação.

Usando OpenMP para *Multithreading*

```
#include <stdio.h>
#include <omp.h>

int main() {
        printf("Olá Mundo:");
        #pragma omp parallel
        printf(" %d", omp_get_thread_num());
        printf("\n");
        return 0;
}
```

Usando OpenMP para *Multithreading*

- Todo programa OpenMP inicia com uma única *thread*.
- Ao atingir uma diretiva *omp parallel*, vários threads são criados.
  - Um time ou gangue de *threads*.
  - Cada *thread* executa a sentença ou bloco imediatamente após a diretiva.
  - Após o fim da sentença ou bloco, um único *thread* executa o restante.



Usando OpenMP para *Multithreading*

- Compilando no GNU GCC C/C++
  - $ gcc –fopenmp –o olamundo olamundo.c
  - $ OMP_NUM_THREADS=8 ./olamundo
- Você está informando o compilador para usar o ambiente OpenMP e configurando quantos *threads* serão criados.
- Por padrão, se você não informar, será criado 1 *thread* para cada *core* lógico.

Usando OpenMP para *Multithreading*

- Regiões paralelas
  - Time de *threads.*
  - *Master Thread.*
  - Bloco estruturado.
    - Entrada e saídas únicas.
    - Barreira implícita.
  - Cláusulas.
    - num_threads(inteiro)
- O escalonamento das *threads* depende do SO e da implementação do OpenMP

```
#pragma omp parallel [cláusula [[,] cláusula] ... ]
        bloco estruturado
```

Usando OpenMP para *Multithreading*

- Configurando o número de *threads:*
  - Variáveis de ambiente:
    - OMP_NUM_THREADS: lista de valores inteiros separados por vírgula.
    - OMP_THREAD_LIMIT: limite total do número de *threads.*
  - Dentro do programa, podemos usar as funções:
    - void omp_set_num_threads(int)
    - int omp_get_num_threads()
    - int omp_get_max_threads()
    - int omp_get_thread_num()
- A vantagem de usar variáveis de ambiente é que o mesmo binário pode ser usado em diferentes configurações de *threads*.

Usando OpenMP para *Multithreading*

- Monitorando um Programa OpenMP
  - Abrir outra aba/sessão/terminal e executar o *top* enquanto o programa executa.
  - Invocar o programa com o comando *time* precedendo o nome do executável:
    - Intervalos *user* e *sys* correspondem ao tempo somado que cada *core* gastou.
    - Intervalo *real* diferença entre o fim e o início do programa.

Usando OpenMP para *Multithreading*

- Cálculo de Fibonnaci
  - Os *threads* realizam quantidade de trabalho diferente.
  - Inicia com um único *thread*.
  - Cria *threads* executa a região paralela.
  - Os *threads* finalizam em momentos diferentes.
- Usando o *top,* é possível ver a variação no uso dos *cores.*

```
#include <stdio.h>
#include <omp.h>

long fib(int n) {
   return (n < 2? 1 : fib(n-1) + fib(n-2));
}

int main() {
   int n = 45;
   #pragma omp parallel
   {
      int t = omp_get_thread_num();
      printf("%d: %ld\n", t, fib(n+t));
   }
   return 0;
}
```

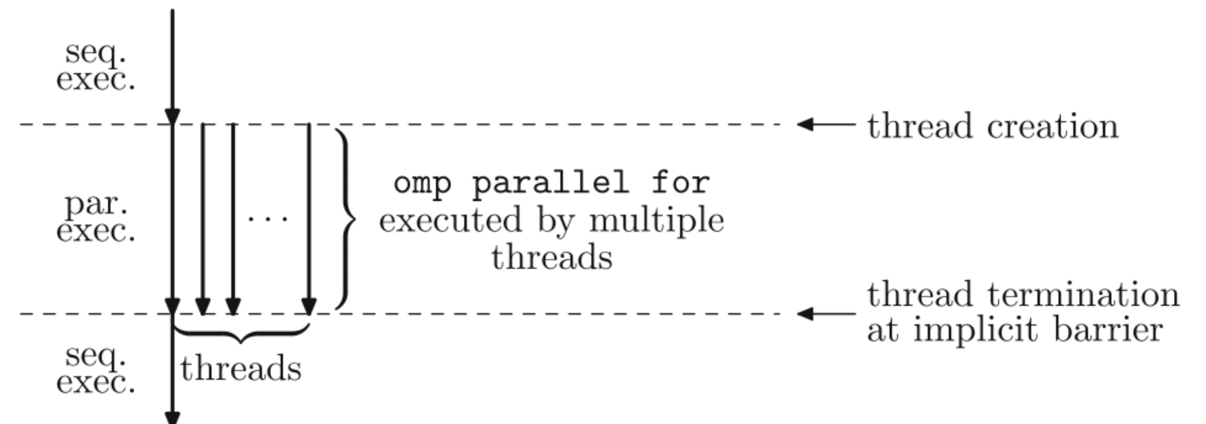
Paralelização de Laços

- A maioria dos programas limitados por CPU passa a maior parte do tempo de execução em laços.
  - Computação Científica e Engenharia.
  - Computação Técnica.
- OpenMP auxilia na paralelização de laços.
  - Eficiência.
  - Portabilidade.
  - Legibilidade.

Paralelização de Laços

- Laços com iterações independentes.
  - Exemplo: imprimir, em qualquer ordem, os números de 1 a *max*.
- *Master thread*: declara e lê o valor de *max*.
- Diretiva *parallel for*: cria *slave threads*.
- Tanto a *master* quanto as *slaves* cooperam na execução do laço:
  - Iterações do laço for são divididas entre as *threads*.
  - Barreira implícita ao final do laço.
- Em geral, existem <u>mais iterações do que *threads*</u>.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {
   int max;
   sscanf(argv[1], "%d", &max);
   #pragma omp parallel for
   for (int i = 1; i <= max; i++)
      printf("%d: %d\n", omp_get_thread_num(), i);
   return 0;
}
```



Paralelização de Laços

Veremos depois como o escalonamento de chunks pode ser controlado pelo programador

- O código anterior não determina como as iterações devem ser divididas entre os *threads*.
- A maioria das implementações do OpenMP opera da seguinte forma:
  - O total de iterações (no caso, valor inicial e final da variável *i*) é dividido em partições chamadas *chunks.*
  - Cada *chunk* tem um subintervalo independente do espaço total de iterações.
  - Uma *chunk* é atribuída a um *thread* para execução, um *thread* pode receber mais de uma *chunk*.
- As iterações dentro de uma *chunk* são executadas serialmente.
- Cada *thread* tem sua própria cópia da variável de controle *i*.
- A variável *max* pode ser compartilhada, pois é apenas lida.

Paralelização de Laços

- Diretiva deve ser usada dentro de uma região paralela.
  - *omp parallel for*
- Laços devem estar na forma canônica:
  - Variável de controle deve ser inteira ou ponteiro.
  - Variável de controle não pode ser modificada durante a execução.
  - A condição deve ser uma relação simples.
  - O incremento deve ser um aditivo constante.
  - O número de iterações deve ser conhecido de antemão.

```
#pragma omp for [cláusula [[,] cláusula] ... ]
        laço for
```

- Uma cláusula permite gerenciar o laço, exemplo:
  - *collapse(int)* indica quantos laços em um aninhamento devem ser paralelizados em conjunto.
  - *nowait* elimina a barreira implícita.
- Veremos mais cláusulas adiante.

Compartilhamento de Dados

- Existem várias cláusulas que definem como as variáveis devem ser compartilhadas entre as *threads* ao iniciar uma região paralela:
  - *shared (list):* variáveis listas são compartilhadas.
  - *private (list):* cada *thread* cria uma cópia local privata da variável.
  - *firstprivate(list):* é igual a *private*, mas a cópia é inicializada com o valor que a variável original possuía antes da região paralela.
  - *lastprivate(list):* a variável original é atualizada com o valor da última atualização antes do final da região paralela.
- Se não for especificado através de cláusulas, por padrão:
  - Variáveis declaradas fora de uma região paralela se tornam compartilhadas.
  - Variáveis declaradas dentro de uma região paralela são privadas.
  - Variáveis estáticas ou alocadas são compartilhadas.
- Ainda não falamos de <u>exclusão mútua</u>, necessária no caso de variáveis compartilhadas.

Exemplo: Adição de Vetores

```
double* vectAdd(double *c, double *a, double *b, int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
    return c;
}
```

- O resultado de uma iteração é totalmente independente das outras.
- Cada iteração acessa uma região de memória diferente.
- Não há <u>condição de corrida.</u>

Exemplo: Todos os Pares de Inteiros

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int max;
    sscanf(argv[1], "%d", &max);
    #pragma omp parallel for
    for (int i = 1; i <= max; i++)
        for (int j = 1; j <= max; j++)
            printf("%d: (%d, %d)\n", omp_get_thread_num(), i, j);
    return 0;
}
```

Exemplo: Todos os Pares de Inteiros

- Como é feito o paralelismo?
  - Se apenas o laço *for* for paralelizado, as suas iterações serão distribuídas.
  - Dentro de cada iteração há um laço *for*, então cada *thread* executa o laço interno por inteiro nas iterações que receber.
  - Se *max* = 6 e o número de threads for 4:
    - 6 iterações no laço externo para serem divididas por 4 *threads*.
    - 6 / 4 não é exata, então 2 *threads* ficam com 2 iterações, e 2 *threads* ficam com 1 iteração.
- Paralelizando apenas o laço externo, o balanceamento entre as *threads* não é adequado.

Entrando em Colapso

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int max;
    sscanf(argv[1], "%d", &max);
    #pragma omp parallel for collapse(2)
    for (int i = 1; i <= max; i++)
        for (int j = 1; j <= max; j++)
            printf("%d: (%d, %d)\n", omp_get_thread_num(), i, j);
    return 0;
}
```

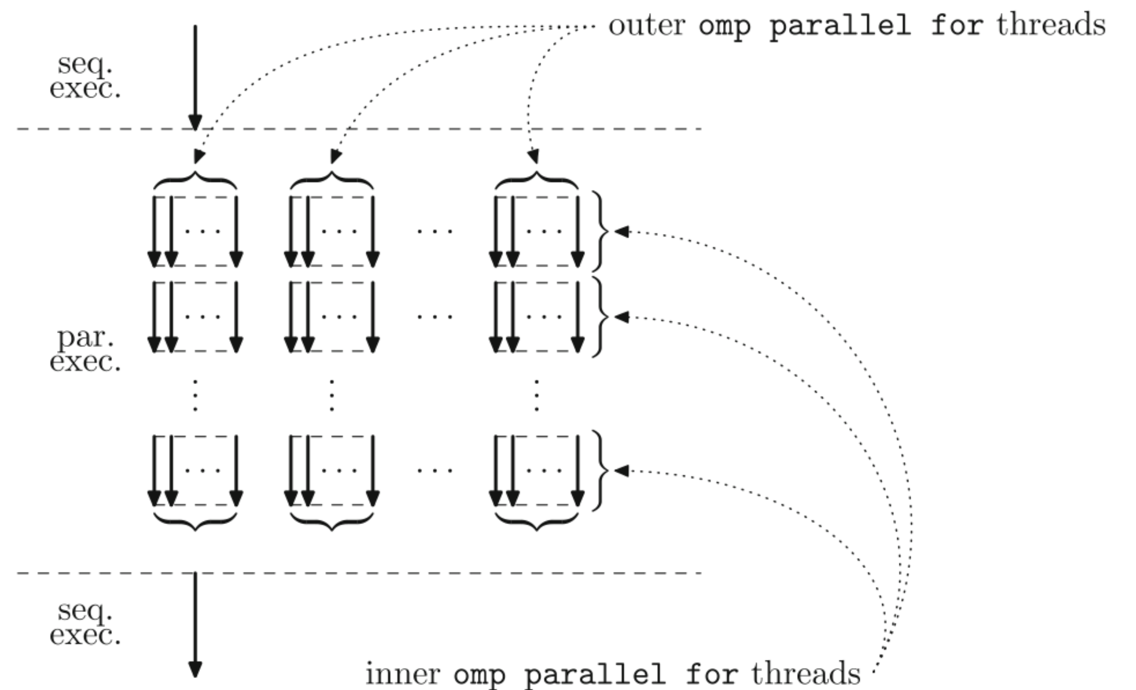
Entrando em Colapso

- O compilador unifica os dois laços aninhados em um só, distribuindo as iterações resultantes.
  - Imagine agora no lugar de 1 até *max*, 1 até max^2.
  - As max^2 iterações são distribuídas entre os *threads.*
  - Para *max* = 6, 36 iterações, 36 / 4 = 9 iterações por *thread*.
  - A distribuição de carga entre os *threads* é mais balanceada.
- Será que ainda temos outra opção?

Regiões Paralelas Aninhadas

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {
   int max;
   sscanf(argv[1], "%d", &max);
   #pragma omp parallel for
   for (int i = 1; i <= max; i++) {
      #pragma omp parallel for
      for (int j = 1; j <= max; j++) {
         printf("%d: (%d, %d)\n", omp_get_thread_num(), i, j);
       }
   }
   return 0;
}
```

- Para funcionar, precisamos habilitar o aninhamento de regiões paralelas
  - *omp_set_nested*(1)
  - OMP_NESTED=true



Regiões Paralelas Aninhadas

- São criados *threads* na primeira diretiva.
  - Os *threads* criados criarão mais *threads* na segunda diretiva.
  - O problema para entender os resultados é que *omp_get_thread_num* retorna o identificador relativo a última região paralela.
- OMP_NUM_THREADS=2,2
  - A primeira região paralela cria 2 *threads.*
  - Cada *thread* criado na primeira, ao encontra uma nova região paralela, cria mais 2 *threads.*

Opções de Paralelização dos Laços Aninhados

- Distribuições para *max* = 6 e 4 *threads*.

1,1	1,2	1,3	1,4	1,5	1,6
2,1	2,2	2,3	2,4	2,5	2,6
3,1	3,2	3,3	3,4	3,5	3,6
4,1	4,2	4,3	4,4	4,5	4,6
5,1	5,2	5,3	5,4	5,5	5,6
6,1	6,2	6,3	6,4	6,5	6,6

Apenas
uma região

1,1	1,2	1,3	1,4	1,5	1,6
2,1	2,2	2,3	2,4	2,5	2,6
3,1	3,2	3,3	3,4	3,5	3,6
4,1	4,2	4,3	4,4	4,5	4,6
5,1	5,2	5,3	5,4	5,5	5,6
6,1	6,2	6,3	6,4	6,5	6,6

Colapso

1,1	1,2	1,3	1,4	1,5	1,6
2,1	2,2	2,3	2,4	2,5	2,6
3,1	3,2	3,3	3,4	3,5	3,6
4,1	4,2	4,3	4,4	4,5	4,6
5,1	5,2	5,3	5,4	5,5	5,6
6,1	6,2	6,3	6,4	6,5	6,6

Regiões
Aninhadas

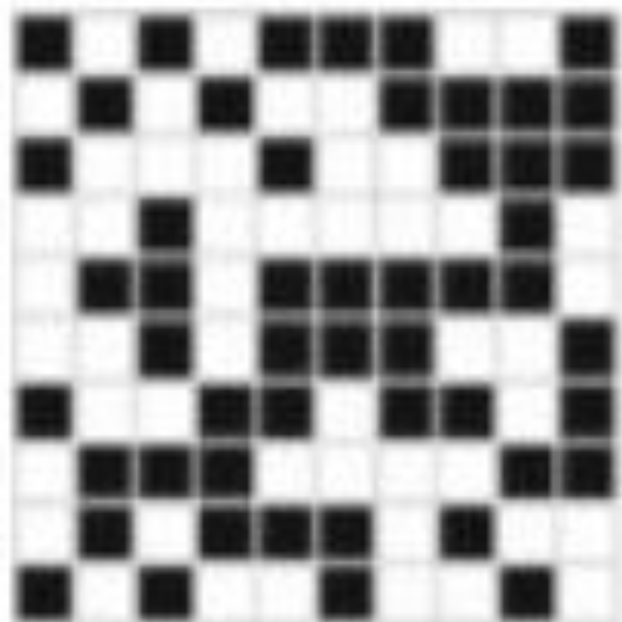
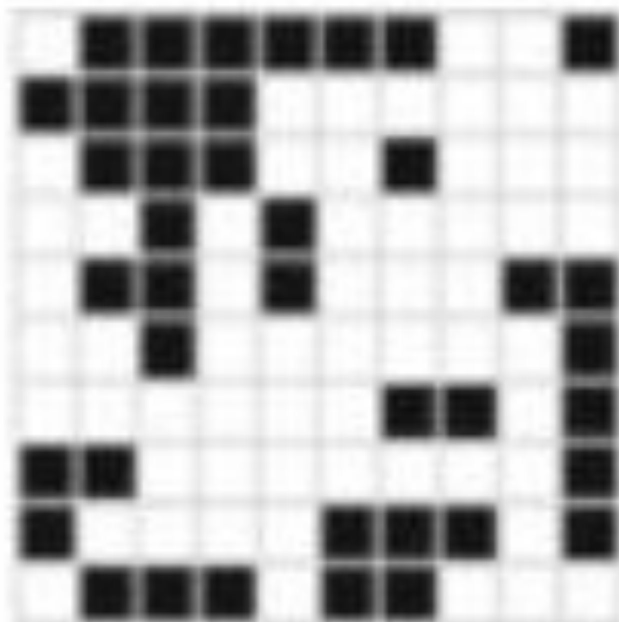
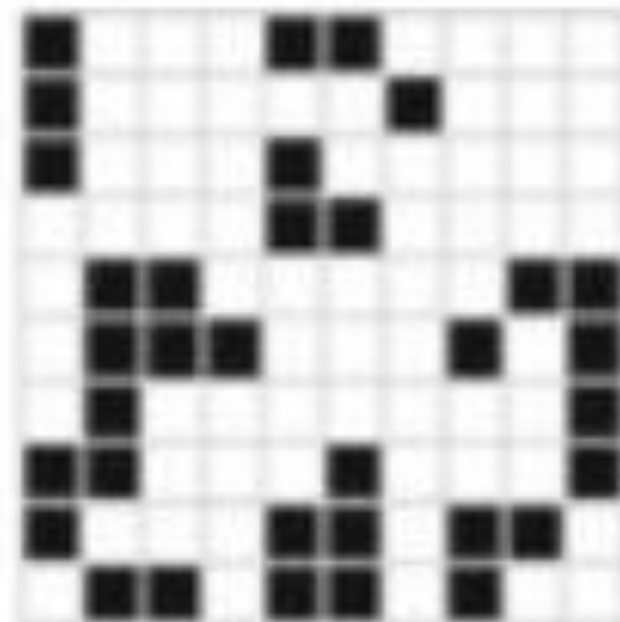
Exemplo: Multiplicação de Matrizes

```
double **mtxMul(double **c, double **a, double **b, int n) {
	#pragma omp parallel for collapse(2)
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++) {
			c[i][j] = 0.0;
			for (int k = 0; k < n; k++)
				c[i][j] = c[i][j] + a[i][k] * b[k][j];
		}
	return 0;
}
```

Exemplo: Jogo da Vida

- Um plano de células, cada uma viva ou morta.
- A cada iteração no tempo:
  - Uma célula com menos de dois vizinhos morre de solidão.
  - Uma célula com dois ou três vizinhos sobrevive.
  - Uma célula com mais de três vizinhos morre de inanição.
  - Cada célula morta, mas com três vizinhos, ressuscita.
- Cada célula tem 8 vizinhos.
- Dada uma geração inicial, qual é a configuração do ambiente no n-ésimo passo?

Exemplo: Jogo da Vida

Exemplo: Jogo da Vida

```
while (gens-- > 0) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int neighs = neighbors(plane, size, i, j);
            switch(plane[i][j]) {
                case 0:  aux_plane[i][j] = (neighs == 3);
                         break;
                case 1:  aux_plane[i][j] = (neighs == 2) || (neighs == 3);
                         break;
            }
        }
    }
    char *tmp_plane  = aux_plane;
    aux_plane = plane;
    plane = tmp_plane;
}
```

Exemplo: Jogo da Vida

- A variável *gens* contém o número de gerações a serem calculadas.
- A geração atual está na matriz *plane* contendo *size x size* células.
- A próxima geração é atualizada na matriz *aux_plane*, também com dimensões *size x size*.
- Tanto *plane* e *aux_plane* são alocados como um vetor de ponteiros para linhas.
- A função *neighbors* retorna o número de vizinhos da célula no plano especificado no primeiro argumento de tamanho especificado no segundo argumento, com posições da célula nos últimos argumentos.

Exemplo: Jogo da Vida

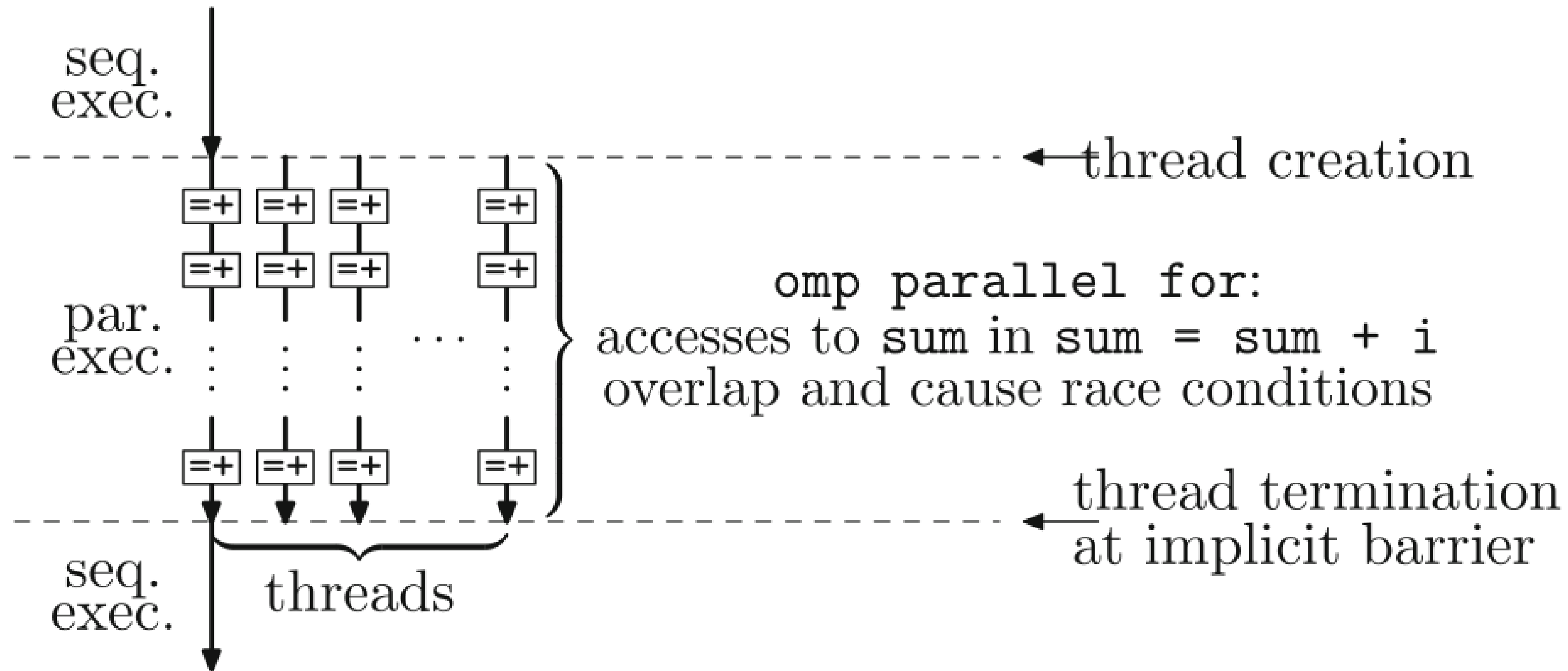
- A única mudança em relação ao código serial é a criação do *for* paralelo.
- Cada iteração do laço *while* irá disparar a criação de uma nova região paralela.
  - Cada iteração do laço *for* externo calcula uma linha do plano da próxima geração.
  - O laço interno atualiza uma única célula para a próxima geração.
- *plane* é apenas leitura, e cada iteração interna é a única a atualizar a posição (i, j) em *aux_plane*.
- Não há condição de disputa.

Combinando o Resultado de Iterações

- Nem todos os problemas paralelos atuam em elementos independentes de uma estrutura de dados compartilhada.
- Exemplo: somar todos os números até um limite.
  - Mesma variável *sum* compartilhada.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int max;
    sscanf(argv[1], "%d", &max);
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i <= max; i++)
        sum = sum + i;

    printf("%d\n", sum);
    return 0;
}
```

Combinando o Resultado de Iterações



Combinando o Resultado de Iterações

- A região crítica é o acesso *sum = sum + i*;
- A solução é colocar essa instrução em **seção crítica**.
  - O uso da diretiva *omp critical* dentro de uma região paralela.
  - O bloco estruturado em uma seção crítica só pode executada por um *thread* por vez.
  - O nome da seção pode ser usado para que tarefas implemente códigos distintos para a mesma seção (exemplo: produtor e consumidor).

```
#pragma omp critical (nome)
        bloco estruturado
```

Combinando o Resultado de Iterações

- Será que essa solução tem impacto no desempenho?
  - Há alguma diferença entre a execução serial e a paralela com a diretiva *critical*?
- Outra opção é utilizar a diretiva atômica.
  - *#pragma omp atomic*
  - Informa o compilador para usar instruções atômicas.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   int max;
   sscanf(argv[1], "%d", &max);
   int sum = 0;

   #pragma omp parallel for
   for (int i = 0; i <= max; i++)
      #pragma omp critical
      sum = sum + i;

   printf("%d\n", sum);
   return 0;
}
```

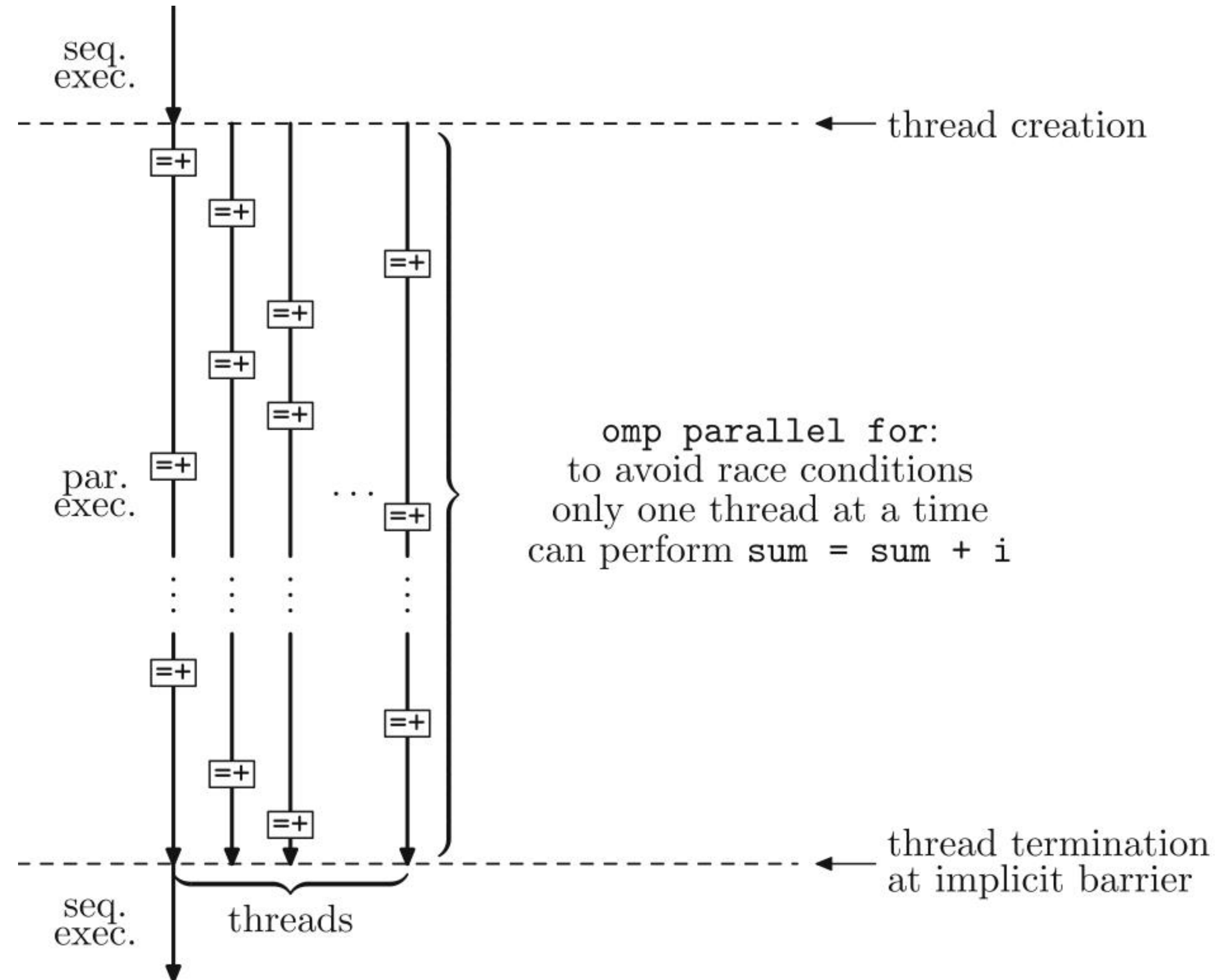
Combinando o Resultado de Iterações

- Três possíveis cláusulas:
  - *read*: expr = x;
  - *write*: x = expr;
  - *update*: ++x, x++, etc.
- A cláusula *seq_cst* força um *flush*.
- Ao contrário da *critical*, são expressões, não blocos.

```
#pragma omp atomic [seq_cst cláusulas]
        expressão
```

```
#pragma omp atomic [seq_cst]
        expressão
```

Combinando o Resultado de Iterações



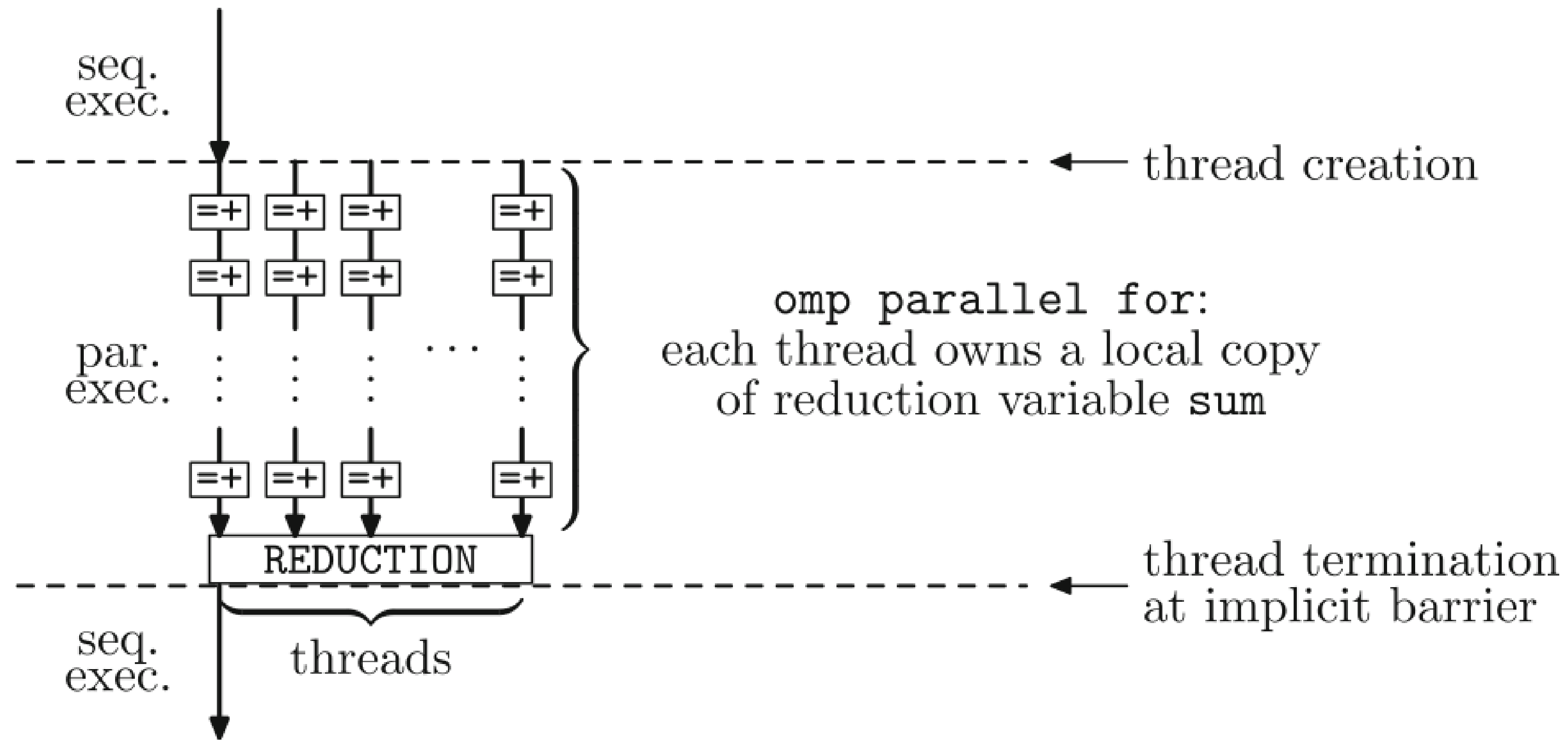
Redução

- Permite que cada *thread* trabalhe em uma cópia local, depois o valor é combinado.
- Possíveis operadores de redução: +, *, &, |, ^, &&, ||, min e max.
  - Para * e &&, o valor inicial é 1.
  - O valor inicial de *min* e *max* são os valores mínimos e máximos do tipo da variável.
  - Para todas outras operações, o valor inicial é 0.

reduction [operador: lista de identificadores]

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   int max;
   sscanf(argv[1], "%d", &max);
   int sum = 0;

   #pragma omp parallel for reduction (+:sum)
   for (int i = 0; i <= max; i++)
      sum = sum + i;

   printf("%d\n", sum);
   return 0;
}
```

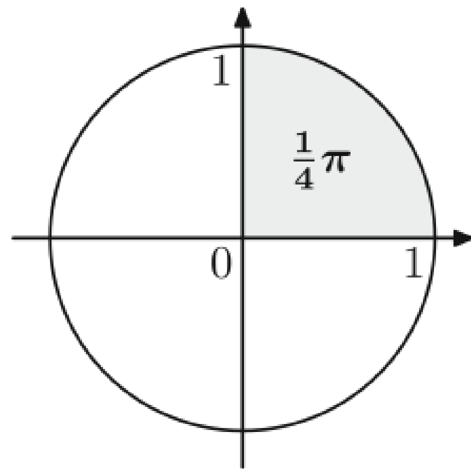
Redução

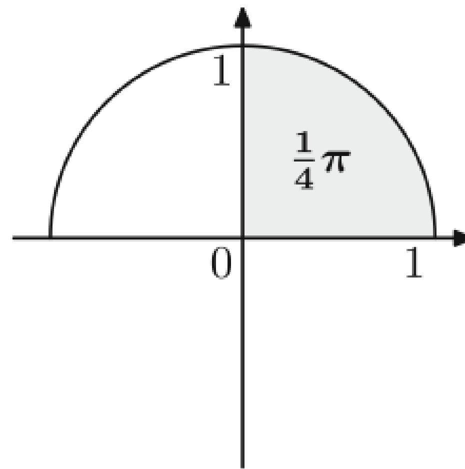


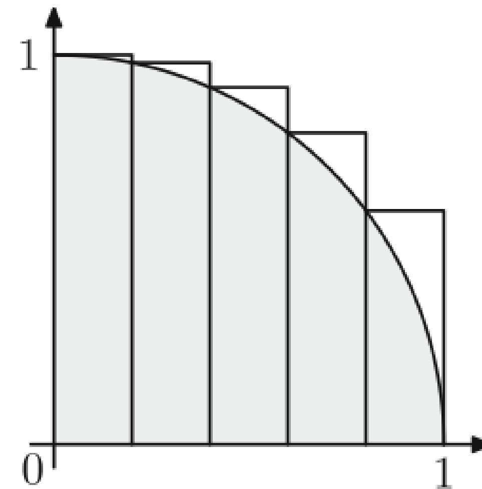
Exemplo: Calcular o Valor de π por Integração

- Calcular o valor de π usando a área do círculo $x^2 + y^2 = 1$.
- Reformulando a equação em $y = \sqrt{1 - x^2}$, teremos que:

$$\pi = 4 \int_0^1 \sqrt{1 - x^2}\, dx$$







Exemplo: Calcular o Valor de π por Integração

- O intervalo $[0, 1]$ é dividido em *N* intervalos $[\frac{1}{N}i, \frac{1}{N}(i+1)]$ onde $0 \leq i \leq (N-1)$, para um número escolhido de intervalos *N.*
- A área de cada retângulo é calculada como a largura do retângulo ($\frac{1}{N}$) vezes o valor $\sqrt{1-(\frac{1}{n})^2}$.

$$\int_0^1 \sqrt{1-x^2}\,dx \approx \sum_{i=0}^{N-1} (\frac{1}{N}\sqrt{1-(\frac{1}{N}i)^2})$$

Exemplo: Calcular o Valor de π por Integração

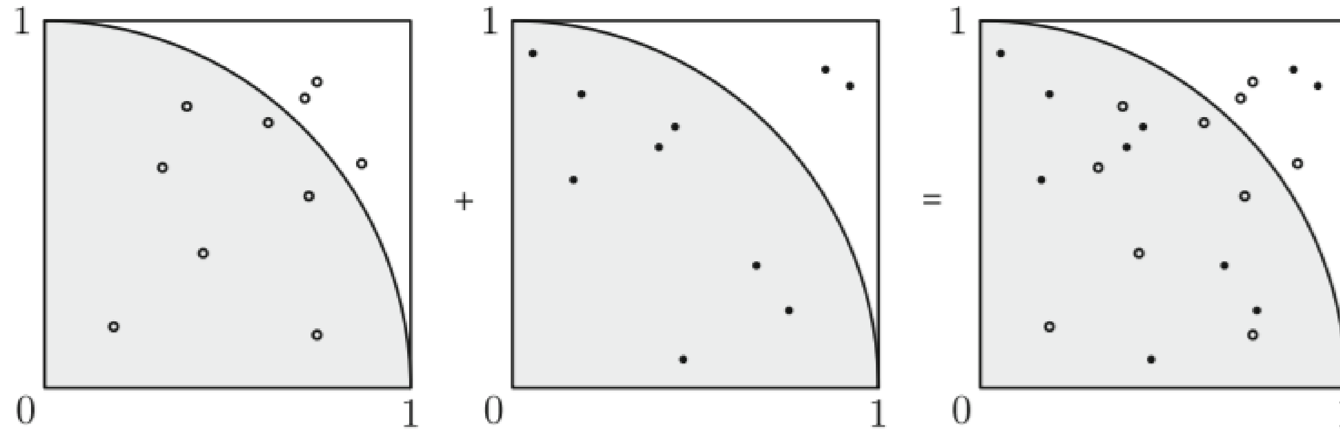
```
int intervals;
sscanf(argv[1], "%d", &intervals);

double integral = 0.0;
double dx = 1.0 / intervals;

#pragma omp parallel for reduction (+:integral)
for (int i = 0; i <= intervals; i++) {
   double x = i * dx;
   double fx = sqrt(1.0 - x * x);
   integral = integral + fx * dx;
}

double pi = 4 * integral;
```

Pode ser paralelizado.

```
int intervals;
sscanf(argv[1], "%d", &intervals);

double integral = 0.0;
double dx = 1.0 / intervals;

double x = 0.0;

#pragma omp parallel for reduction (+:integral)
for (int i = 0; i <= intervals; i++) {
   double fx = sqrt(1.0 - x * x);
   integral = integral + fx * dx;
   x = x + dx;
}

double pi = 4 * integral;
```

Funciona?

Exemplo: Calcular o Valor de π por Monte Carlo



- Quadrado [0,1] por [0,1];
- Gerar pontos aleatórios dentro desse quadro.

π = 4 * (pontos gerados dentro do círculo) / total de pontos gerados.

Exemplo: Calcular o Valor de π por Monte Carlo

- Só há um pequeno problema: os geradores de números aleatórios *rand* ou *random* não são reentrantes:
  - Cada invocação a essas funções acessam uma variável global que serve como base para a definição do número aleatório.
  - Se mais de uma *thread* executa o corpo de uma dessas funções, não há controle de seção crítica na variável global, o mesmo número "aleatório" é gerado duas vezes.
- Precisamos de uma função que use um número base para *seed* diferente em cada *thread*.
- Ainda não é um resultado aleatório verdadeiro, mas é melhor que usar *rand* ou *random*.

Exemplo: Calcular o Valor de π por Monte Carlo

```
double rnd (unsigned int * seed) {
   *seed = (1140671485 * (* seed) + 12820163) % (1 << 24);
   return ((double)(*seed)) / (1 << 24);
}
int main (int argc , char * argv[]) {
   int num_shots;
   sscanf (argv[1], "%d", & num_shots);
   unsigned int seeds[omp_get_max_threads()];
   for (int thread = 0; thread < omp_get_max_threads(); thread++)
      seeds[thread] = thread;
   int num_hits = 0;
   #pragma omp parallel for reduction(+:num_hits)
     for (int shot = 0; shot < num_shots; shot++) {
          int thread = omp_get_thread_num ();
          double x = rnd (&seeds[thread]);
          double y = rnd (&seeds[thread]);
          if (x * x + y * y <= 1) num_hits = num_hits + 1;
     }
   double pi = 4.0 * (double) num_hits / (double)num_shots;
   printf ("%20.18lf\n", pi);
   return 0;
}
```

Distribuindo Iterações entre *Threads*

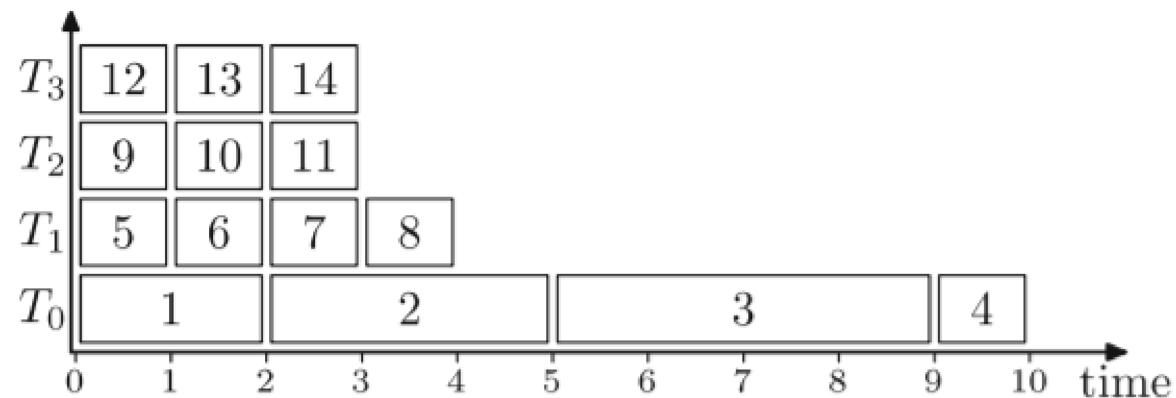
- Até agora, não explicamos como influenciar o escalonamento de iterações entre os *threads*.
- A diretiva *schedule* com valor *runtime* permite definir o escalonamento pela variável de ambiente OMP_SCHEDULE

```
#pragma omp parallel for reduction (+:sum) schedule (runtime)
for (int i = 1; i < max; i++) {
        printf("%2d @ %d\n", i, omp_get_thread_num());
        sleep (i < 4 ? i + 1 : 1);  /* As iterações iniciais "trabalharão" mais */
        sum = sum + i;
}
```

Distribuindo Iterações entre *Threads*

- Considere *max* = 14 e 4 *threads*.
  - *OMP_SCHEDULE=static* : divisão de iterações em *chunks* de tamanho aproximadamente igual.

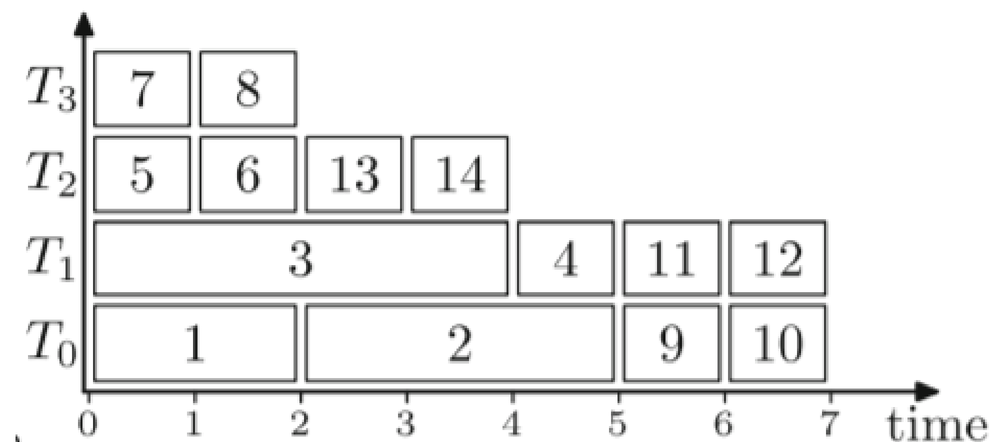
$$T_0:\ \underbrace{\{1, 2, 3, 4\}}_{\text{10 secs}} \quad T_1:\ \underbrace{\{5, 6, 7, 8\}}_{\text{4 secs}} \quad T_2:\ \underbrace{\{9, 10, 11\}}_{\text{3 secs}} \quad T_3:\ \underbrace{\{12, 13, 14\}}_{\text{3 secs}}$$



Distribuindo Iterações entre *Threads*

- Considere *max* = 14 e 4 *threads*.
  - *OMP_SCHEDULE=static,2* : o tamanho do *chunk* é 2, distribuídos em *round-Robin*

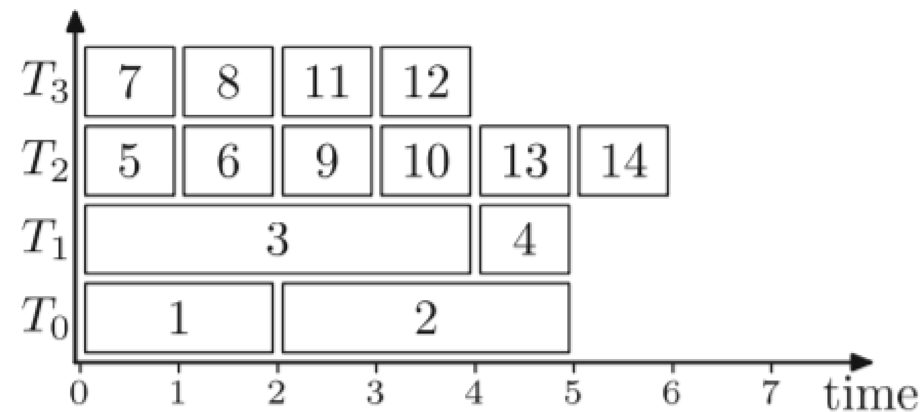
$T_0: \underbrace{\{1, 2; 9, 10\}}_{7 \text{ secs}} \quad T_1: \underbrace{\{3, 4; 11, 12\}}_{7 \text{ secs}} \quad T_2: \underbrace{\{5, 6; 13, 14\}}_{4 \text{ secs}} \quad T_3: \underbrace{\{7, 8\}}_{2 \text{ secs}}$



Distribuindo Iterações entre *Threads*

- Considere *max* = 14 e 4 *threads*.
  - *OMP_SCHEDULE=dynamic,2* : divisão de iterações em *chunks* de tamanho 2, colocados em uma *pool*.

$$T_0: \underbrace{\{1, 2\}}_{5 \text{ secs}} \quad T_1: \underbrace{\{3, 4\}}_{5 \text{ secs}} \quad T_2: \underbrace{\{5, 6; 9, 10; 13, 14\}}_{6 \text{ secs}} \quad T_3: \underbrace{\{7, 8; 11, 12\}}_{4 \text{ secs}}$$



Distribuindo Iterações entre *Threads*

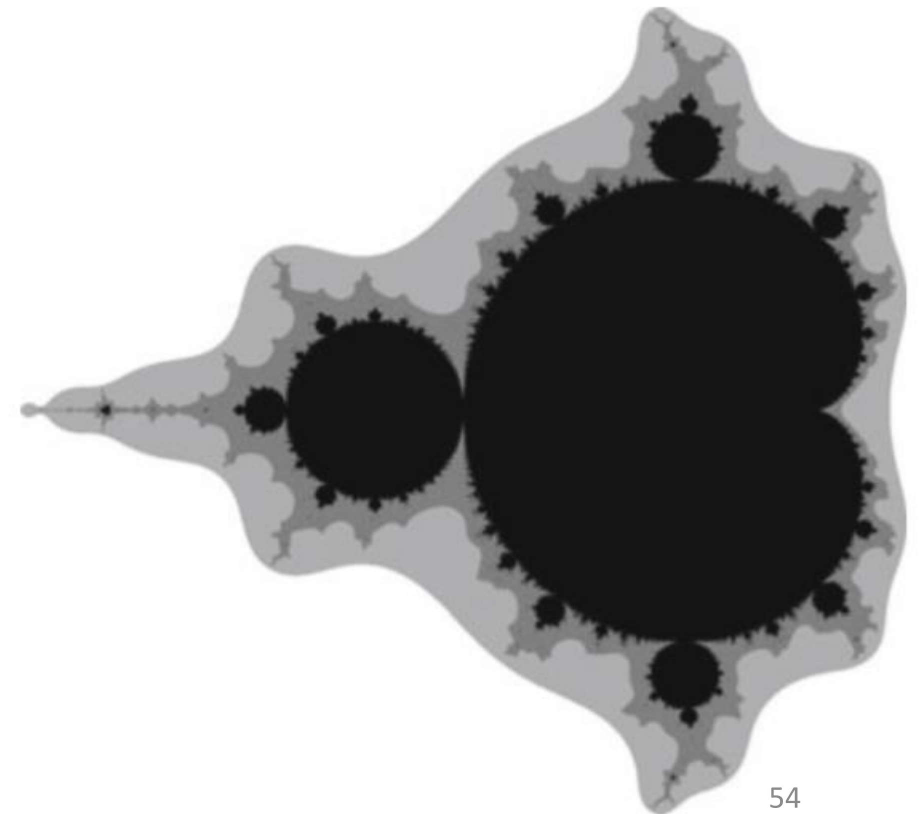
- Você também pode colocar o escalonamento direto no código
  - *schedule (static)*: *chunks* de tamanhos iguais, cada *thread* no máximo recebe um deles.
  - *schedule (static, chunk_size): chunks* de tamanhos *chunk_size*, divididos de maneira *round robin* entre os *threads*.
  - *schedule (dynamic, chunk_size): chunks* de tamanhos *chunk_size*, com uma pool de *chunks*.
  - *schedule (auto):* deixa o compilador decidir.
  - *schedule (runtime):* utilize a opção definida na variável OMP_SCHEDULE.
- Na ausência da diretiva, o comportamento é semelhante a *schedule (auto)*.

Exemplo: Conjunto de *Mandelbrot*

- O cálculo do conjunto de *Mandelbrot* consiste em definir quais pontos pertencem a um conjunto em $\mathbb{C}$. Temos a seguinte definição:

$$M = \{c \; ; \; \limsup_{n\to\infty} |z_n| \leq 2 \text{ where } z_0 = 0 \; \wedge \; z_{n+1} = z_n^2 + c\}$$

- Na equação, *c* são as coordenadas em $\mathbb{C}$.
- O nível de escuro na figura determina o número de iterações necessárias para definir se ponto pertence ou não ao conjunto.



Exemplo: Conjunto de *Mandelbrot*

```
#pragma omp parallel for collapse(2) schedule (runtime)
for (int i = 0; i < i_size; i++) {
        for (int j = 0; j < j_size; j++) {
                double c_re = min_re + i * d_re;
                double c_im = min_im + j *d_im;

                double z_re = 0.0;
                double z_im = 0.0;
                int iters = 0;
                while ((z_re * z_re + z_im * z_im < 4.0) && (iters < max_iters)) {
                        double new_z_re = z_re * z_re – z_im * z_im + c_re;
                        double new_z_im = 2 * z_re * z_im + c_im;
                        z_re = new_z_re;
                        z_im = new_z_im;
                        iters = iters + 1;
                }
                picture[i][j] = iters;
        }
}
```

Exemplo: Conjunto de *Mandelbrot*

- Usando *static,100* no lugar de apenas *static*, o *speedup* melhora em 30 %.
- O uso de *dynamic,100* melhora ainda mais o *speedup*.
- Isso ocorre porque cada (*i,j*) leva a uma quantidade de trabalho diferente.
- Mas... na prática?

Equivalência entre *parallel* e *parallel for*

- Você, em teoria, não precisa usar o for.
- Sabendo o número de iterações, poderia calcular manualmente o que cada *thread* irá fazer.
- Implementar escalonamento específico para sua aplicação.
- Mas o compilador costuma implementar operações otimizadas na redução, por exemplo.

Equivalência entre *parallel* e *parallel for*

```
int ts = omp_get_max_threads();
  if (max % ts != 0)
    return 1;
  int sums[ts];
  #pragma omp parallel
  {
    int t = omp_get_thread_num();
    int lo = (max / ts) * (t + 0) + 1;
    int hi = (max / ts) * (t + 1) + 0;
    sums[t] = 0;
    for (int i = lo; i <= hi; i++)
      sums[t] = sums[t] + i;
  }
  int sum = 0;
  for (int t = 0; t < ts; t++)
    sum = sum + sums[t];
```

Tarefas Paralelas

- A paralelização de laços é direta:
  - A região paralela cria as *threads*.
  - As iterações são divididas entre as *threads* para execução.
  - Você pode imaginar uma iteração como uma <u>tarefa</u>, uma requisição, que é designada para execução em uma *thread*.
- Se quisermos ter uma granularidade diferente nass tarefas, podemos cria-las sem utilizar um laço.

Tarefas Paralelas

```
#pragma omp parallel
{
  #pragma omp single
  for (int t = 0; t < tasks; t++){
    #pragma omp task
    {
      int local_sum = 0;
      int lo = (max / tasks) * (t + 0) + 1;
      int hi = (max / tasks) * (t + 1) + 0;

      printf("%d: %d..%d\n", omp_get_thread_num(), lo, hi);
      for (int i = lo; i <= hi; i++)
        local_sum = local_sum + i;
      #pragma omp atomic
        sum = sum + local_sum;
    }
  }
}
```

- *tasks* e *max* são fornecidos pelo usuário.
- Calculamos em *lo* e *hi* quais "iterações" cada *thread* cuidará.
- Toda vez que uma diretiva *#pragma omp task* é encontrada:
  - Uma nova tarefa é criada.
  - Tarefa != *thread*.
- A diretiva *#pragma omp single* garante que só uma *thread* executa o controle do laço, criando as tarefas.
- Cabe ao ambiente de execução alocar as tarefas a *threads.*
- Há uma barreira implícita para as tarefas ao final da região paralela.

Tarefas Paralelas

- Tarefas:
  - Cria uma tarefa que será designada a uma *thread*.
  - Cláusulas possíveis:
    - *final (expressão lógica)*: se a expressão for avaliada para verdadeiro, não são criadas novas tarefas.
    - *if (expressão lógica):* se a expressão for avaliada para verdadeiro, a tarefa sendo criada suspende a criadora.

#pragma omp task [*cláusulas*]
bloco-estruturado

- Execução única:
  - Garante que apenas uma *thread* executa o bloco.
  - Não há como dizer qual delas.
    - *private* (lista): específica variáveis privadas.
    - *nowait*: elimina a barreira.

#pragma omp single [*cláusulas*]
bloco-estruturado

Fibonacci em Tarefas

```
long fib(int n) {
   return (n < 2? 1 : fib(n-1) + fib(n-2));
}

int main(int argc, char *argv[]) {
   int max;
   sscanf(argv[1],"%d", &max);

   #pragma omp parallel
      #pragma omp single
      for (int n = 1; n <= max; n++)
         #pragma omp task
         printf("%d: %d %ld\n", omp_get_thread_num(), n, fib(n));
   return 0;
}
```

- Apenas para demonstrar que laços paralelos podem ser transformados em emissões de tarefas.
- Novamente, uma única *thread* criará as tarefas.
- Aqui as threads mais rápidas são criadas primeiro.
- Como fazer o contrário?

O cenário mais interessante para as tarefas é quando não sabemos o número de “iterações”.

Exemplo: Ordenação *QuickSort*

- A *QuickSort* é recursiva, não é natural representa-la como um laço.
- Dá para ter uma previsão do número de “iterações”, mas teríamos que abandonar a recursão para uma versão iterativa.
- Podemos usar tarefas no lugar do laço paralelo para manter a modelagem recursiva e ainda assim tirar proveito do paralelismo.

Exemplo: Ordenação *QuickSort*

```
void par_qsort(int *data, int lo, int hi, int (*compare)(int, int)) {
    if (lo > hi) return;
    int l = lo;
    int h = hi;
    int p = data[(hi + lo) / 2];
    while (l <= h) {
        while (compare(data[l], p) < 0) l++;
        while (compare(data[h], p) > 0) h--;
        if (l <= h) {
            int tmp = data[l];
            data[l] = data[h];
            data[h] = tmp;
            l++;
            h--;
        }
    }
    #pragma omp task final (h - lo < 1000)
        par_qsort(data, lo, h, compare);

    #pragma omp task final (hi - l < 1000)
        par_qsort(data, l, hi, compare);
}

int main(int argc, char *argv[]) {
 ...
 ...
  #pragma omp parallel
      #pragma omp single
          par_qsort(data, 0, size - 1, &compareInt);
...
...
}
```

Exemplo: Ordenação *QuickSort*

```
int par_qsort(char **data, int lo, int hi,  int (*compare)(const char *, const char *)) {
  if (lo > hi)  return 0;
  int l = lo;
  int h = hi;
  char *p = data[(hi + lo) / 2];
  int count = 0;
  while (l <= h)   {
    while (compare(data[l], p) < 0) l++;
    while (compare(data[h], p) > 0) h--;
    if (l <= h)        {
      count++;
      char *tmp = data[l];
      data[l] = data[h];
      data[h] = tmp;
      l++;    h--;
    }
  }
  int locount, hicount;
#pragma omp task shared(locount) final(h - lo < 1000)
  locount = par_qsort(data, lo, h, compare);
#pragma omp task shared(hicount) final(hi - l < 1000)
  hicount = par_qsort(data, l, hi, compare);
#pragma omp taskwait
  return count + locount + hicount;
}
```

Podemos combinar o resultado de tarefas como uma redução