
Alessandra Chan - *aleclipper*

Luis Felipe Martins Linhares - *pascal*

Murilo Carneiro Rodrigues - *figura*

Rafael Soares Camargo - *brás*

# Tópicos

- Alternativas para a para Programação Paralela
- Memória Compartilhada
- Thread
- Métodos de paralelismo SMP
- O que é OpenMP?
- Histórico
- Objetivos
- Visão Geral
- OpenMP C/C++
- Overhead
- Usando OpenMP
- OpenMP vs. MPI
- Exemplos
- Referências

# *Alternativas para Programação Paralela*

- Uso de uma nova linguagem
- Uso de uma linguagem seqüencial modificada para lidar com paralelismo
- Uso de um compilador para "paralelização"
- Uso de rotinas de biblioteca/diretivas de compilação em uma linguagem seqüencial existente
  - Memória compartilhada (OpenMP) x Memória distribuída (MPI)

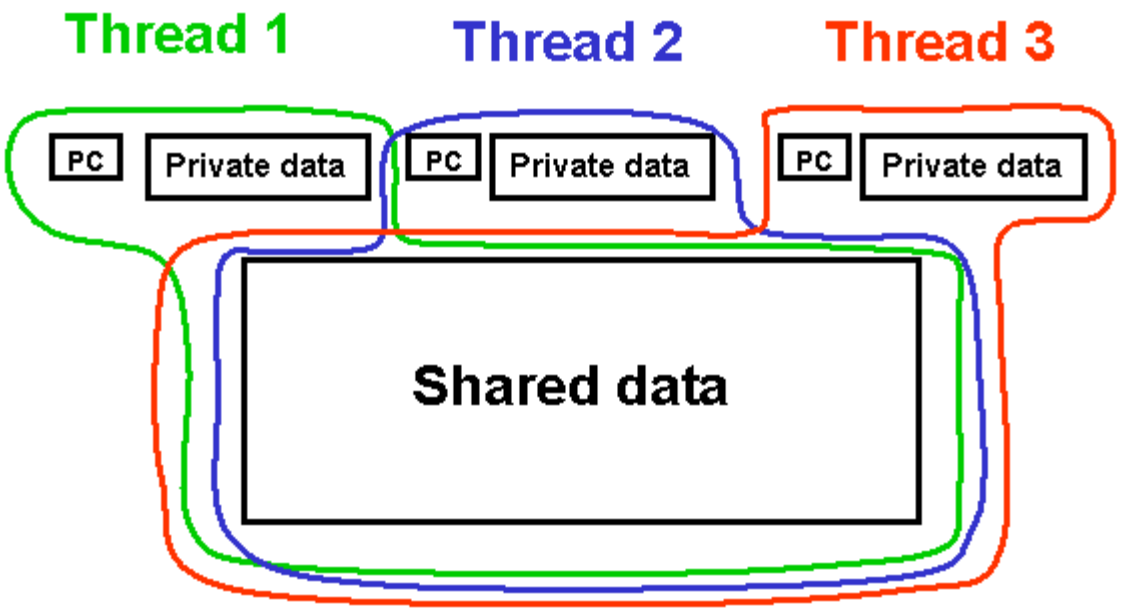
# *Memória Compartilhada*

- Todos os processadores podem acessar toda a memória em um sistema paralelo
- Tempo de acesso pode não ser igual para todos os processadores
- Paralelização em memória compartilhada não reduz o tempo de CPU
- Execução paralela é obtida pela geração de múltiplos threads que executam em paralelo
- Número de threads é independente do número de processadores

# *Threads*

- Todos os threads criados pelo mesmo processo compartilham o mesmo espaço de endereçamento
- Baixo custo de criação e destruição
- Podem ser criados e associados a múltiplos processadores
  - Base de paralelismo SMP (Shared-Memory Programming)

# Threads


Thread 1
Thread 2
Thread 3
PC
Private data
PC
Private data
PC
Private data
Shared data

# *Métodos de Paralelismo SMP*

- Uso explícito de threads
  - Pthreads
- Uso de um compilador paralelizador e suas diretivas para gerar threads em um nível mais alto de abstração
  - Diretivas vendor-specific (ex. !SMP$)
  - Diretivas industry-standard (ex. !$OMP)

OpenMP

# O que é OpenMP?

- API usada para paralelismo de memória compartilhada multi-thread
- Conjunto de diretivas de compilação, biblioteca de rotinas dinâmicas e variáveis de ambiente
- Portabilidade
  - API especificada para Fortran e C/C++
  - Implementado em várias plataformas: plataformas Unix e a Windows NT

# O que é OpenMP?

- Desenvolver programas paralelos para arquiteturas baseadas em memória compartilhada
- Fácil programação
  - Não exige o detalhe da paralelização
  - Vantagem em relação ao uso de threads
  - Mais atraente que MPI (para máquinas com memória compartilhada)

# *Histórico*

- Baseado no ANSI X3H5 (1994)
  - Não portável
- Criado em conjunto por várias empresas de hardware e software em 1997
- Padronização
  - Fabricantes de hardware e software parceiros (Compaq, HP, IBM, Intel, Sun,...)
  - Espera-se tornar um padrão ANSI

# Objetivos

- Prover um padrão para diversos tipos de arquiteturas baseadas em memória compartilhada
- Estabelecer um conjunto simples e limitado de diretivas para a programação paralela
- Prover a capacidade de paralelizar um programa serial de forma incremental
- Suporta paralelismo de granulosidade fina ou grossa
- Suportar Fortran(77, 90 e 95) e C/C++

# *Visão Geral*

- Diretivas e sentinelas
- Biblioteca de rotinas
- Variáveis de ambiente
- Regiões paralelas
- Dados compartilhados e privados
- Loops paralelos
- Sincronização
- Reduções

# *Diretivas e sentinelas*

- Dão inúmeros recursos ao programador
- Diretivas são linhas especiais de código
- As diretivas são distinguidas por um sentinela no início da linha
  - !$OMP → Fortran
  - #pragma omp → C/C++

# Diretivas

| Clause | Directive | | | | | |
|---|---|---|---|---|---|---|
| | PARALLEL | DO/for | SECTIONS | SINGLE | PARALLEL DO/for | PARALLEL SECTIONS |
| IF | ● | | | | ● | ● |
| PRIVATE | ● | ● | ● | ● | ● | ● |
| SHARED | ● | ● | | | ● | ● |
| DEFAULT | ● | | | | ● | ● |
| FIRSTPRIVATE | ● | ● | ● | ● | ● | ● |
| LASTPRIVATE | | ● | ● | | ● | ● |
| REDUCTION | ● | ● | ● | | ● | ● |
| COPYIN | ● | | | | ● | ● |
| SCHEDULE | | ● | | | ● | |
| ORDERED | | ● | | | ● | |
| NOWAIT | | ● | ● | ● | | |

- Diretivas sem cláusula
  - MASTER
  - CRITICAL
  - BARRIER
  - ATOMIC
  - FLUSH
  - ORDERED
  - THREADPRIVATE

# Biblioteca de rotinas

- OMP_SET_NUM_THREADS
- OMP_GET_NUM_THREADS
- OMP_GET_MAX_THREADS
- OMP_GET_THREAD_NUM
- OMP_GET_NUM_PROCS
- OMP_IN_PARALLEL
- OMP_SET_DYNAMIC
- OMP_GET_DYNAMIC
- OMP_SET_NESTED
- OMP_GET_NESTED
- OMP_INIT_LOCK
- OMP_DESTROY_LOCK
- OMP_SET_LOCK
- OMP_UNSET_LOCK
- OMP_TEST_LOCK

# *Variáveis de ambiente*

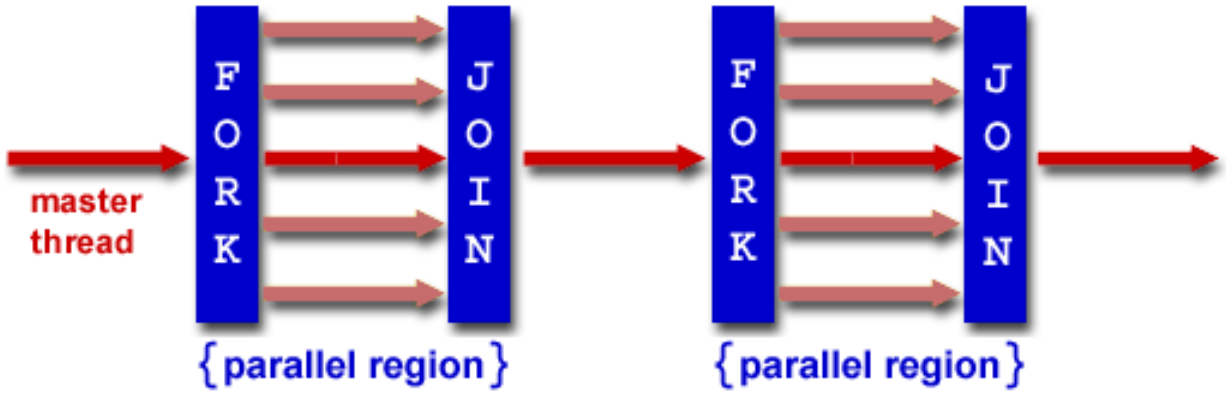
- OMP_SCHEDULE
  - Aplica-se apenas para *for e parallel for* que possuem a cláusula *schedule* definida como RUNTIME
  - O valor dessa variável determina como as iterações do loop são escalonadas nos processadores
    - Exemplo: setenv OMP_SCHEDULE “guided, 4”
- OMP_NUM_THREADS
  - Define o número máximo de threads utilizados durante a execução
    - Exemplo: setenv OMP_NUM_THREADS 10
- OMP_DYNAMIC
  - Habilita ou desabilita o ajuste do número de threads disponíveis para a execução em regiões paralelas
    - Exemplo: setenv OMP_DYNAMIC TRUE
- OMP_NESTED
  - Habilita ou desabilita o paralelismo aninhado
    - Exemplo: setenv OMP_NESTED FALSE

# *Regiões paralelas*

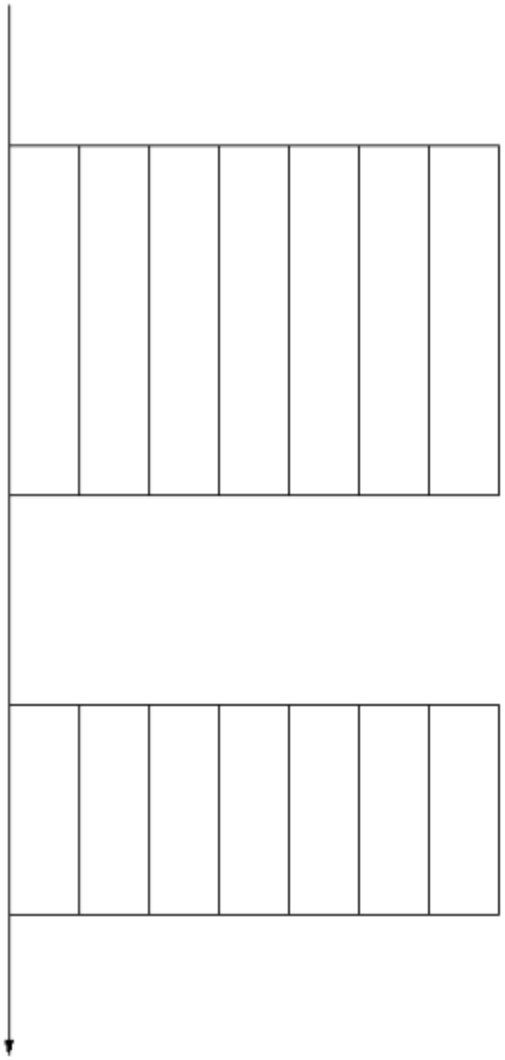
- Construção paralela básica em OpenMP
- Define uma seção de um programa
- O programa inicia com a execução de apenas um thread
- Quando uma região paralela é encontrada este cria um conjunto de threads (modelo fork/join)
- Cada thread executa o código da região
- Ao final, o thread principal espera pelo término da execução do conjunto de threads para continuar

# Regiões paralelas

- Master thread → thread principal
- Fork → cria um conjunto de threads
- Parallel region → código executado em paralelo pelo conjunto de threads
- Join → sincronização e término do conjunto de threads



# Regiões paralelas - Exemplo



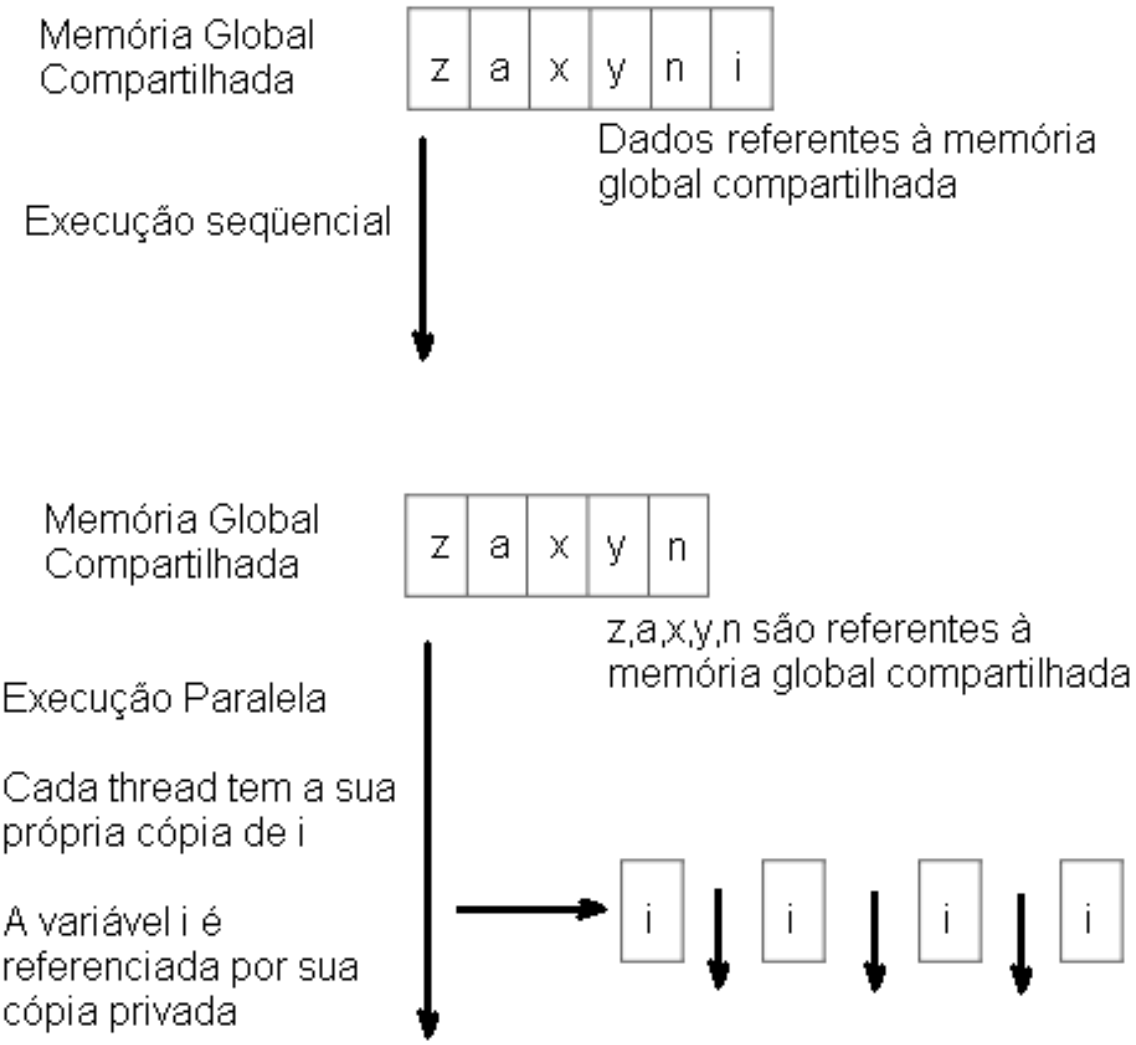
```
      PROGRAM FRED
        .
        .
!$OMP PARALLEL
        .
        .
        .
        .
        .
        .
        .
        .
        .
!$OMP END PARALLEL
        .
        .
        .
        .
        .
!$OMP PARALLEL
        .
        .
        .
        .
        .
!$OMP END PARALLEL
        .
        .
        .
```

# *Dados compartilhados e privados*

- Em uma região paralela, variáveis podem ser compartilhadas ou privadas
- Todos os threads enxergam a mesma cópia das variáveis compartilhadas e podem ler/escrever nas mesmas
- Cada thread tem a sua própria cópia das variáveis privadas que podem ser escrita/lida pelo próprio thread, sendo invisível aos outros threads

# Dados compartilhados e privados Exemplo

```
void exemplo()
{
    int i, n;
    float z[n], a, x[n], y;
   #pragma omp parallel \
    shared(z,a,x,y,n) private(i)
    {
       #pragma omp parallel for
       for (i=1; i<n; ++i)
            z[i] = a * x[i] + y;
       return;
    }
}
```



# *Dados compartilhados e privados*
# *Divisão de trabalho*

**n=40, 4 threads**

**Memória global compartilhada**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| z(1) | | z(10) | z(11) | | z(20) | z(21) | | z(30) | z(31) | | z(40) | a |
| x(1) | | x(10) | x(11) | | x(20) | x(21) | | x(30) | x(31) | | x(40) | y |
| | | | | | | | | | | | | n |

**Memória local privada**

| | | | |
|---|---|---|---|
| i = 1, 10 | i = 11, 20 | i = 21, 30 | i = 31, 40 |

# *Loops paralelos*

- Loops são a principal fonte de paralelismo
- Se as iterações de um loop são independentes (podem ser feitas em qualquer ordem), então pode-se compartilhar as iterações entre threads
- Exemplo:

```
for (i=0; i<=100; ++i)
  a[i] += b[i];
```

Usando dois threads: iteração entre 1 a 50 e 51 a 100 nos threads

# *Sincronização*

- Necessidade de assegurar que ações em variáveis compartilhadas ocorrem na ordem correta
  - Exemplo: thread 1 deve escrever variável A antes do thread 2 a leia
- Atualizações em variáveis compartilhadas são não atômicas
  - Devem ser usados artifícios para contornar esse fato.
    - Exemplo: #pragma omp critical

# Reduções

- Uma redução produz um único valor derivado de operações associativas como adição, multiplicação, max, min, and, or,...
- Exemplo:

```
#pragma omp parallel for default(shared) private(i) /
  schedule(static,chunk) reduction(+:result)
for (i=0; i < n; i++)
    result += (a[i] * b[i]);
```

- Ao permitir que apenas um thread atualiza a variável result por vez, o paralelismo é removido
- Então, cada thread faz a sua própria acumulação, e no final estas são somadas

# *OpenMP em C/C++*

- Escopo de variáveis:
  - Quando #pragma omp parallel é definido, todas as variáveis visíveis são compartilhadas por padrão
- Variáveis estáticas declaradas dentro de uma região paralela também são compartilhadas
- A memória alocada utilizando *malloc* é compartilhada, porém o ponteiro pode ser privado

# *Overhead*

- A sobrecarga na paralelização utilizando OpenMP é grande
  - Exemplo: em 16 processadores SGI Origin 2000, são consumidos 8000 ciclos utilizando a diretiva *parallel do*
  - O tamanho do código paralelo construído deve ser significante para compensar o overhead criado
  - Regra: são necessários 10 kFLOPS para amortizar o overhead

# *Usando OpenMP*

- Geralmente utiliza-se para parelelizar loops
  - Encontra-se os loops mais custosos
  - Divide-se os mesmos entre vários threads
- Melhores resultados podem ser obtidos usando regiões paralelas de OpenMP, porém essa não é uma tarefa trivial

# OpenMP vs. MPI

- Apenas para computadores de memória compartilhada
- Fácil para o paralelismo incremental
  - Maior dificuldade para escrever programas altamente escalares
- Pequena API baseada em diretivas e rotinas limitadas
- O mesmo programa pode ser usado para execução seqüencial e paralela
- Variáveis compartilhadas e privadas podem causar confusão

- Portável para todas plataformas
- Paraleliza tudo ou nada
- Vasta coleção de rotinas
- Difícil de usar o mesmo código para execução seqüencial e paralela
- Variáveis são locais para cada processador

# Exemplos

```
#include <omp.h>
main ()
{
   int i, n, chunk;
   float a[100], b[100], result;
   n = 100;
   chunk = 10;
   result = 0.0;
   for (i=0; i < n; i++)
   {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
   }
   #pragma omp parallel for        \
        default(shared) private(i)\
        schedule(static,chunk)     \
        reduction(+:result)
   for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
   printf("Resultado final= %f\n",result);
}
```

# Exemplos

```
#include <omp.h>
main()
{
   int x;
   x = 0;

   #pragma omp parallel shared(x)
   {
        #pragma omp critical
        x = x + 1;
   }
}
```

# Exemplos

```
#include <omp.h>
#define N 1000
main ()
{
   int i;
   float a[N], b[N], c[N];

   for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
   #pragma omp parallel shared(a,b,c) private(i)
   {
        #pragma omp sections nowait
        {
                #pragma omp section
                for (i=0; i < N/2; i++)
                        c[i] = a[i] + b[i];
                #pragma omp section
                for (i=N/2; i < N; i++)
                        c[i] = a[i] + b[i];
        }
   }
}
```

# *Referências*

- www.openmp.org
- www.llnl.gov/computing/tutorials/openMP
- www.mcc.ac.uk/HPC/OpenMP/
- www.inf.ufrgs.br/~elgio/trabs-html/Nucleo/openMP.html