Open MP

Prof. MSc. Rodrigo D. Malara

Centro Universitário de Araraquara

Adaptado de "The OpenMP Crash Course" da Arctic Region Supercomputing Center

INTRODUÇÃO AO OPENMP

O que é o OpenMP ?

- O que é OpenMP?
- Modelo de execução Fork-Join
- Como funciona?
  - OpenMP versus Threads
  - OpenMP versus MPI
- Componentes do OpenMP
  - Diretivas de Compilação
  - Bibliotecas
  - Variáveis de Ambiente

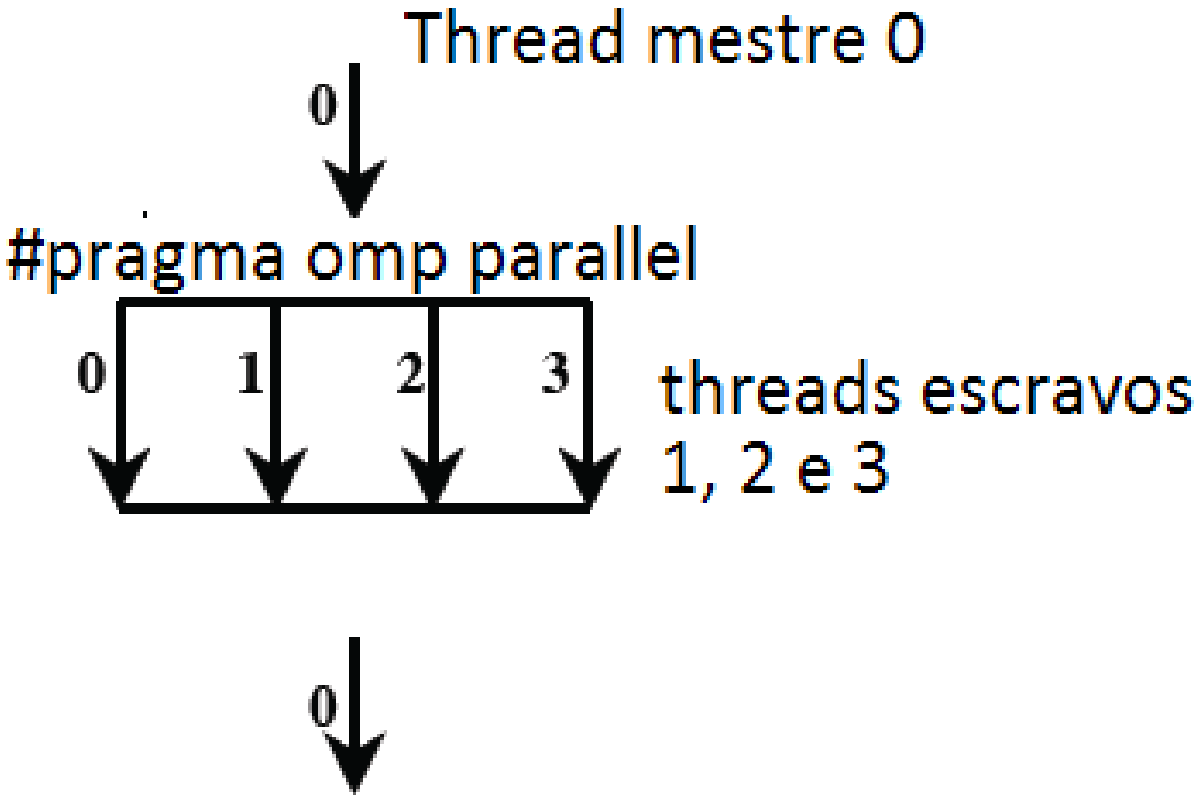
O que é o OpenMP ?

- Modelo de programação paralela de memória compartilhada padronizada
- Oferece portabilidade entre plataformas
- Só é útil para os sistemas de memória compartilhada
- Permite paralelismo incremental por diretivas
- Utiliza diretivas, variáveis de ambiente, chamadas à biblioteca própria

Modelo de Execução Fork-Join

- A execução começa em um thread mestre único
- Mestre gera threads para as regiões paralelas
  - Regiões paralelas executadas por vários threads
  - Threads escravos e mestre participam na região paralela
  - Escravos executam apenas dentro da região paralela
- Execução retorna para o thread mestre depois de uma região paralela

Como Funciona ?



Open MP versus Threads

- OpenMP e threads usam o mesmo paralelismo Fork-Join
- Threads
  - Explicitamente criar processos
  - Mais carga programador
- OpenMP
  - Cria threads implicitamente
  - É relativamente fácil de se programar

OpenMP versus MPI

- OpenMP
- Um processo, muitos threads
- Arquitetura compartilhada
- Mensagens implícitas
- Sincronização explícita
- Paralelismo Incremental
- Paralelismo de Granularidade Fina
- É relativamente fácil de programar

- MPI - Message Passing Interface
- Vários processos
- Arquitetura não-compartilhada
- Troca explícita de mensagens
- Sincronização Implícita
- Paralelismo tudo-ou-nada
- Paralelismo de Granularidade Grossa
- Relativamente difícil de se programar

Funções Utilitárias https://pastebin.com/rArLjp48

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Funcoes utilitarias
// Preenche um vetor de tamanho n com numeros aleatorios de 0 ate limite
void initRand(int* vetor, int n, int limite) {
        int i;
        float fatorMult = ((float)limite)/(float)RAND_MAX;
        srand(time(NULL));
        for (i = 0; i < n; i++) {
                vetor[i] = rand() * fatorMult;
        }
}

// Calcula o tempo entre duas tomadas de tempo em segundos
float calculaTempoSegundos(clock_t ini, clock_t fim) {
        return ((double)fim-ini)/CLOCKS_PER_SEC;
}

// Faz o processador aguardar ms milisegundos - Consome CPU !
void dormir(unsigned int ms){
    clock_t goal = ms + clock();
    while (goal > clock());
}
```

Funções Utilitárias https://pastebin.com/rArLjp48

```
// Imprime o vetor
void imprimeVetor(int v[], int tam) {
        int i;
        for (i = 0; i < tam; i++) {
                if (i > 0) printf(", ");
                printf("%d", v[i]);
        }
        printf("\n");
}

void insertionSort(int v[], int n) {
  int i, j, chave;
  for(j=1; j<n; j++) {
       chave = v[j];
       i = j-1;
       while(i >= 0 && v[i] > chave){
               v[i+1] = v[i];
               i--;
       }
        v[i+1] = chave;
     }
}
```

Funções Utilitárias https://pastebin.com/rArLjp48

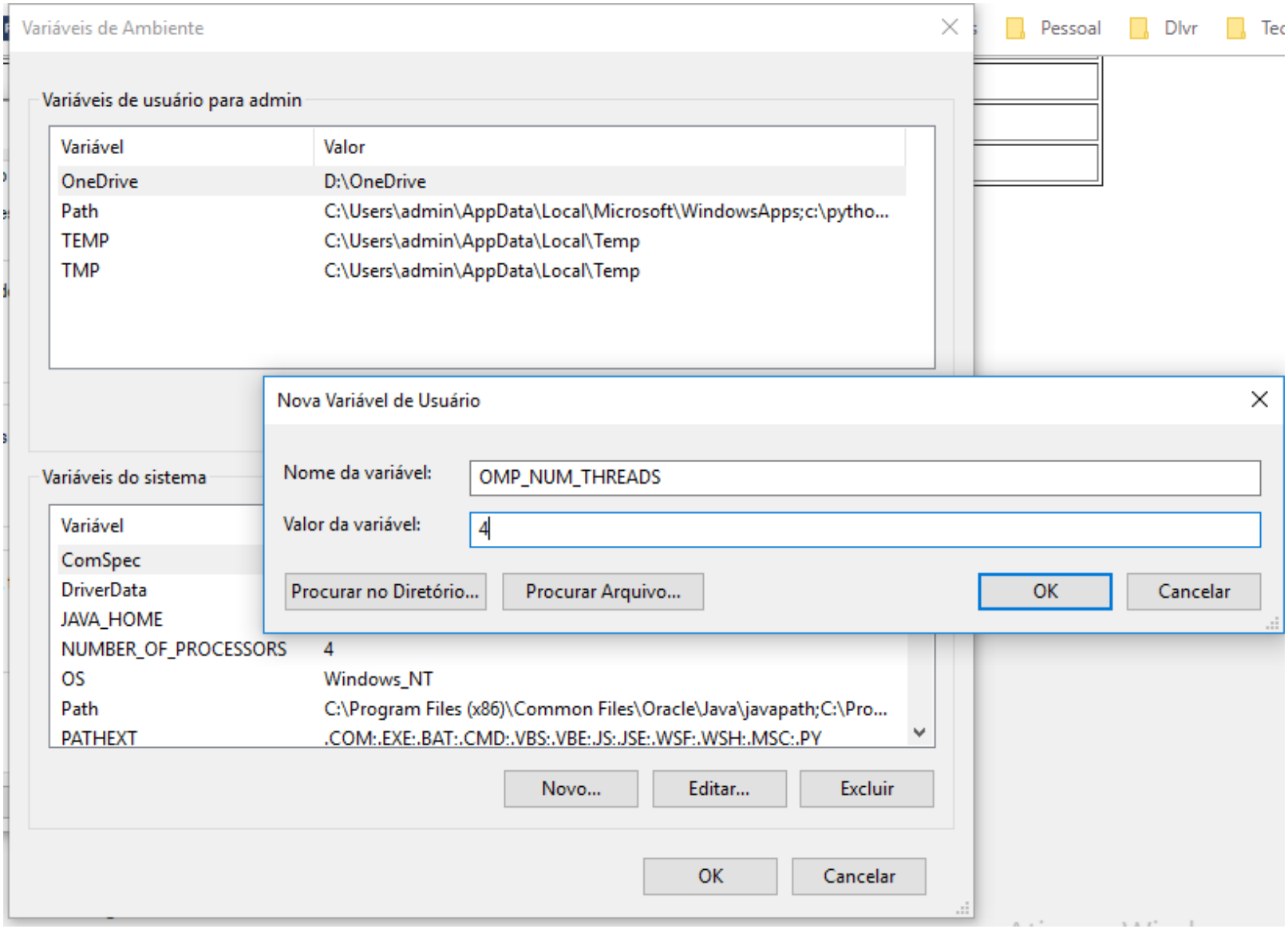
```
int main(int argc, char *argv[])
{
  // Definindo variaveis de tempo inicial e final
  clock_t ini, fim;  // Tempo de processamento (fim - ini)


  int vet[100000];

  printf("Insertion Sort\n");
  initRand(vet, 100000, 5000);

  imprimeVetor(vet, 20); // apenas 20 posicoes

  ini = clock();
  insertionSort(vet, 100000);
  fim = clock();

  imprimeVetor(vet, 20); // apenas 20 posicoes
  printf("Tempo Insertion Sort %f\n", calculaTempoSegundos(ini,fim));
}
```

CONFIGURAÇÃO DE AMBIENTE PARA O OPENMP

Variável de Ambiente

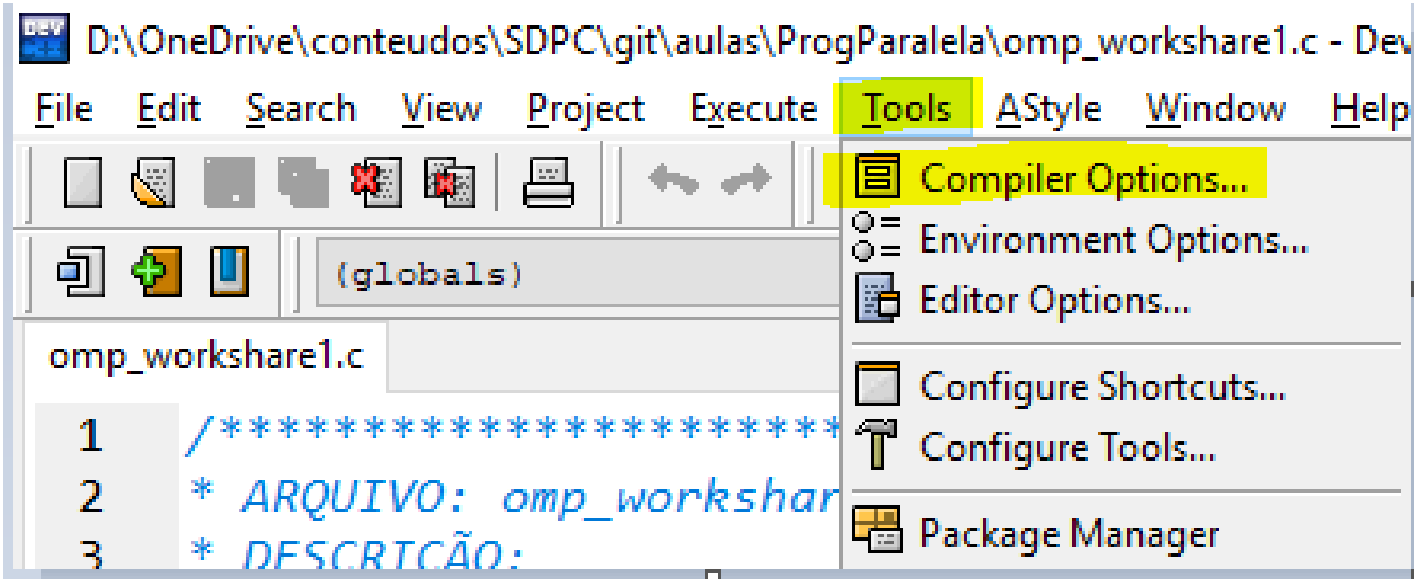
- Adicionar a variável de ambiente OMP_NUM_THREADS com o valor 4
- Faça isso em
  - Meu Computador
  - Propriedades
  - Configurações Avançadas
  - Variáveis de Ambiente

Variável de Ambiente

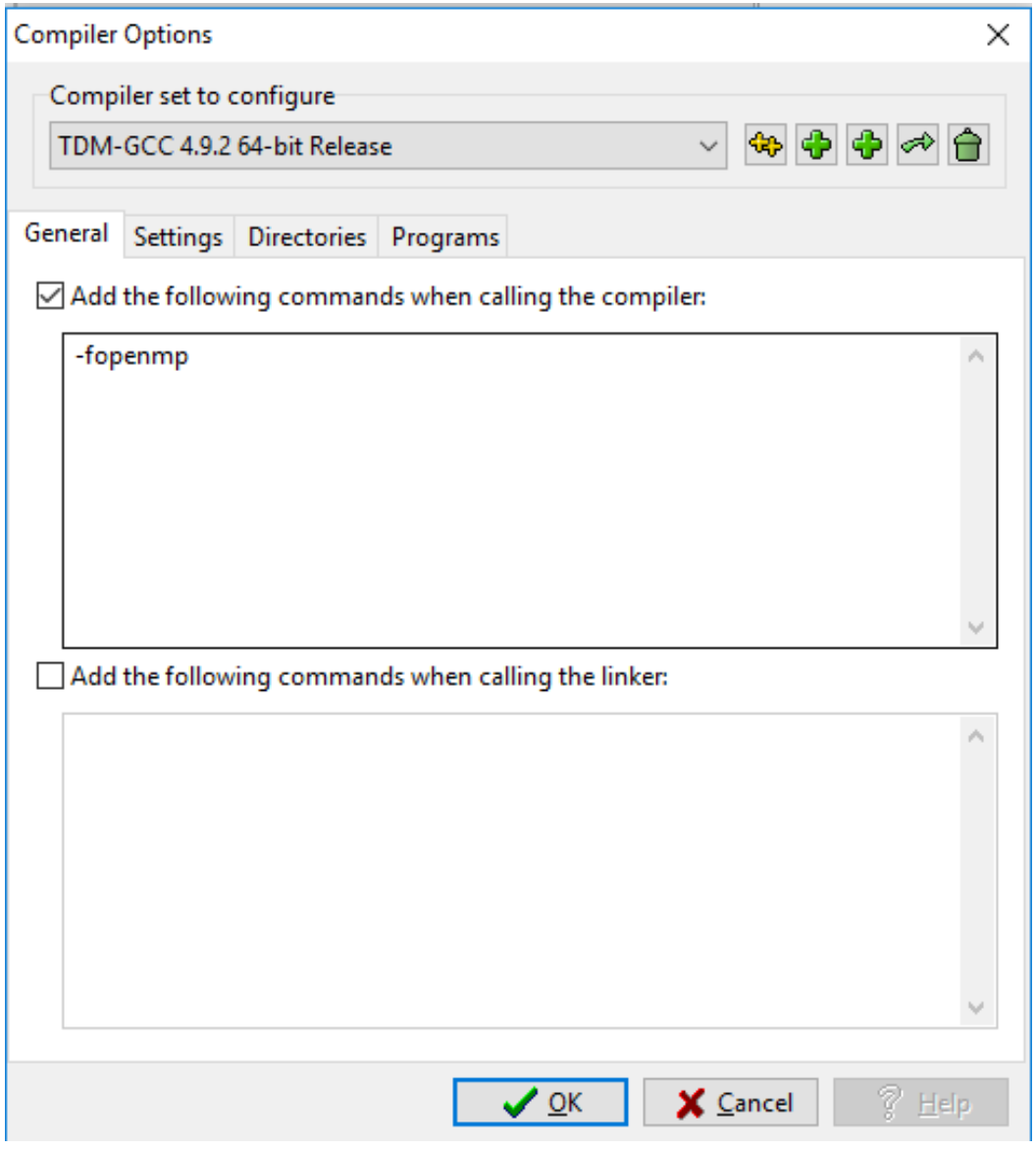


Configure o Dev C++

- Adicionar a opção -fopenmp nas opções de compilação
- Faça isso em



Configure o Dev C++



COMPONENTES DO OPENMP

Diretivas de Compilação

- Compilação baseada em diretivas
  - Compilador interpreta diretivas como comentários a menos que OpenMP esteja habilitado
- O mesmo código pode ser compilado como serial ou multitarefa
- Diretivas permitem
  - Compartilhamento de Trabalho
  - Sincronização de Dados
  - Definição de Escopo

Bibliotecas de Funções

- Rotinas de informação
  - omp_get_num_procs () - número de processadores no sistema
  - omp_get_max_threads () - número máximo de threads permitidos
  - omp_get_num_threads () - Obtém o número de threads ativos
  - omp_get_thread_num () - obtém número do thread atual
- Definir o número de threads
  - omp_set_num_threads (inteiro)
  - Configura número de threads
  - Ver OMP_NUM_THREADS
- Acesso a dados e sincronização
  - omp_ <*> _lock () – controla locks do OMP

Primeiro Exemplo

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

int main(int argc, char *argv[]) {
    int tid;
    printf("Hello world OMP\n");

    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf(" Thread nro %d\n", tid);
    }

    printf("Sou sequencial novamente\n");
    return 0;
}
```

Variáveis de Ambiente

- Controle do ambiente de execução
  - OMP_NUM_THREADS - número de threads a serem usados
  - OMP_DYNAMIC - ativar / desativar o ajuste segmento dinâmico
  - OMP_NESTED - ativar / desativar o paralelismo aninhado
- Controle do escalonamento de trabalho
  - OMP_SCHEDULE
    - especificar como será a distribuição de trabalho entre os threads
    - **static** – cada thread executa um número fixo de iterações
    - **dynamic** – as iterações são atribuídas aos threads em tempo de execução
    - **guided** - começa com pedaços grandes, o tamanho diminui exponencialmente
    - Ex Windows: *set OMP_SCHEDULE "dynamic, 4"*

CONTRUÇÕES DO OPENMP

Diretivas para Compilação Condicional

- C/C++

```
#ifdef _OPENMP
...
#endif
```

Formato das Diretivas

sentinela nome_da_diretiva [clausula [[,] clausula …]

- Diretivas devem ser em minúsculas em C
  - Em Fortran, isso não importa
- Cláusulas podem ser separadas por virgulas ou espaços em branco
- Sentinelas
  - C/C++: #pragma omp { … }
  - Fortran: !omp / comp / *$omp

Construção: Região Paralela

#pragma omp parallel [clausulas]

Bloco estruturado

- Todo o código do bloco abaixo a diretiva se repete em todos os threads
- Cada thread tem acesso a todos os dados definidos no programa
- Barreira implícita no final da região paralela

Loops Paralelizáveis

#pragma omp parallel for

- Combina construção paralela com compartilhamento de trabalho
- Conveniência
- A Construção mais comum do OpenMP é o *parallel for*

Exemplo: Comp. Paralelo

```
int main (int argc, char *argv[]) {
    int nthreads, tid, i;
    float a[N], b[N], c[N];

    /* Inicializacoes */
    for (i=0; i < N; i++) {
        a[i] = b[i] = i * 1.0;
    }

    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Restrições em loops paralelos

- O vetor deve ser de tamanho fixo
- While pode não servir
- Deve-se concluir todas as iterações do loop
  - não pode usar construções de saída (break, go to, continue)
    - exceção - a execução pode ser encerrada usando exit em C / C ++

Seções

```
#pragma omp sections [clausula]
{
   #pragma omp section
   {
        [codigo para a seção 1]
   }
   #pragma omp section
   {
        [codigo para a seção 2]
   }
}
```

- Seções concorrentes de código distribuidas entre os threads

Exemplo: Seções

```
int main (int argc, char *argv[]) {

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                printf("Thread executando secao 1\n");
            }
            #pragma omp section
            {
                printf("Thread executando secao 2\n");
            }
        }  /* fim das secoes */
    }
}
```

Simples

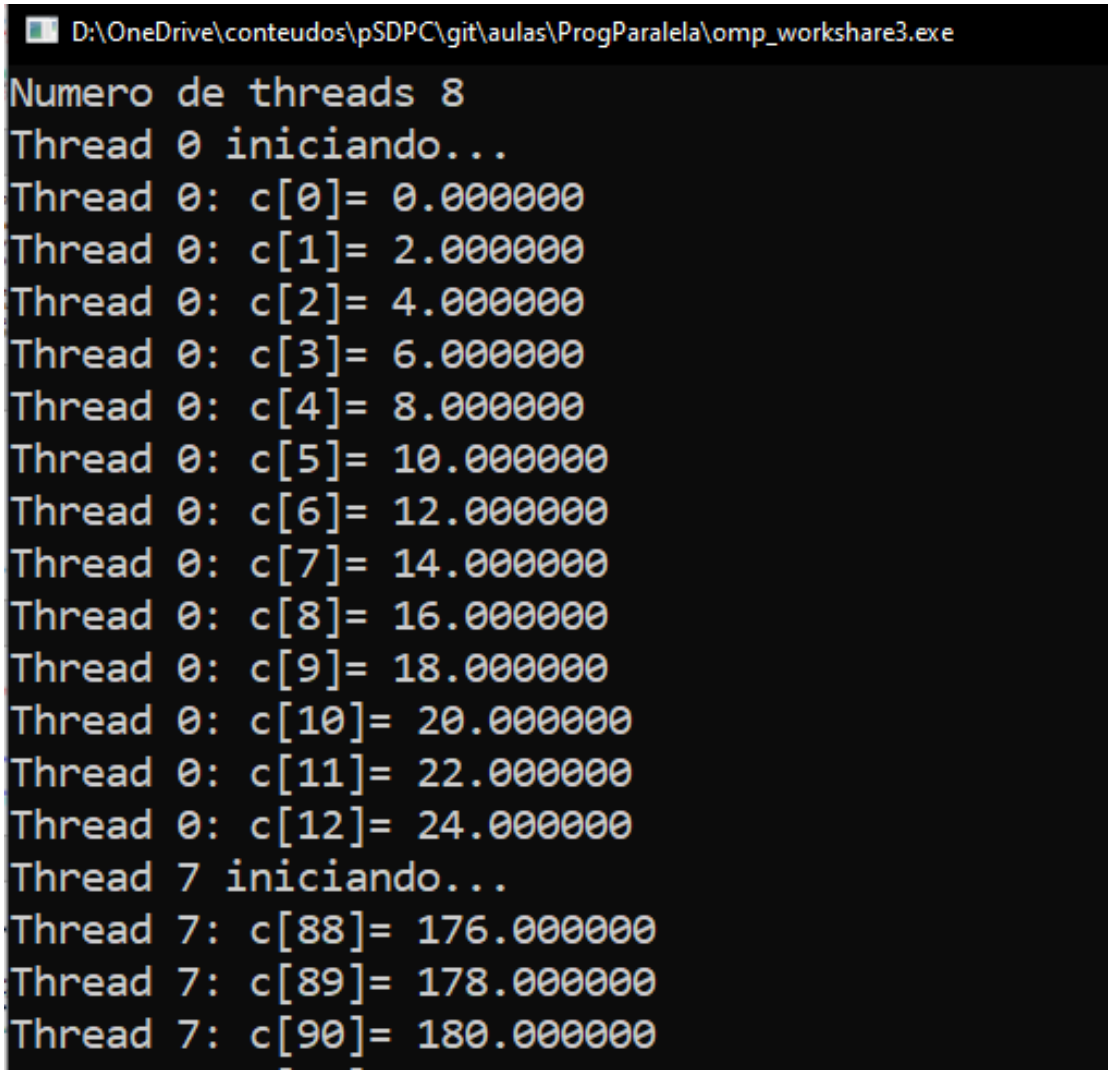
```
#pragma omp single [clausula] ...
{
   [codigo]
}
```

- Define o código serial em uma região paralela
- Um único thread (qualquer) executa o código
- Nenhuma barreira implícita no início da construção
- O uso comum é para a realização de operações de E / S

Exemplo: Simples

```
#pragma omp parallel private(tid)
{
    #pragma omp single
    {
        int numThreads = omp_get_num_threads();
        printf("Numero de threads %d\n", numThreads);
    }

    tid = omp_get_thread_num();
    printf("Thread %d iniciando...\n", tid);

    #pragma omp for
    for (i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
    }
}  /* Fim da secao paralela */
```

Exemplo: Simples



ESCOPO DE DADOS

Escopo de dados: O que é ?

- As fontes mais comum de erros na prog. de memória compartilhada são:
  - Compartilhamento de variáveis não-intencional
  - Privatização de variáveis que precisam ser compartilhadas
- Determinar se as variáveis devem ser compartilhadas ou privadas é a chave para a correção do programa paralelo
  e desempenho
  - Parte mais difícil da programação OpenMP

Escopo de dados

- Cada variável tem um escopo privado ou compartilhado
- As cláusulas de Escopo consiste em:
  - *private* e *shared* de variáveis específicas
  - *reduction* identifica explicitamente uma variável de redução

  - *firstprivate* ou *lastprivate* para inicialização e finalização das variáveis privatizadas
  - *default: muda* regras padrão quando as variáveis são implicitamente definidas

Escopo padrão

- Se o escopo uma variável em uma região paralela não está explicitamente definido então ela é compartilhada
  - Exceto para índices de laços **parallel for**
- O comportamento compartilhado é correto se a variável é apenas lida, mas não é modificada
- Se a variável for atribuída nas regiões paralelas, então é necessário
  - Acertar o escopo explicitamente
  - Adicionar sincronização

Exemplo de Escopo

INCONSISTENTE

```
#pragma omp parallel for
for (i=0; i<nl; i++)
{
    int x;
    x = func1(i);
    y =func2(i);
    a[i] = x*y;
}
```

CONSISTENTE

```
#pragma omp parallel for private(y)
for (i=0; i<nl; i++)
 {
    int x;
    x = func1(i);
    y =func2(i);
    a[i] = x*y;
 }
```

Exemplo de Escopo

RUIM

```
#pragma omp parallel for
for (i=0; i<nl; i++)
{
    int x;
    x = func1(i);
    y =func2(i);
    a[i] = x*y;
}
```

Concorrência gerando potencial inconsistência na variável y

OK pois i é gerenciada pelo OpenMP

BOM

```
#pragma omp parallel for private(y)
for (i=0; i<nl; i++)
{
    int x;
    x = func1(i);
    y =func2(i);
    a[i] = x*y;
}
```

Solução cada thread terá a sua própria variável y

Regras de Escopo

- Uma variável individual pode aparecer no máximo em uma única cláusula de escopo
- Variável a ter seu escopo definido deve se referir a um objeto inteiro, e não uma parte de um objeto
  - Não é possível definir o escopo de um elemento de uma matriz ou um campo de uma estrutura
- Válido para estruturas C ou classes C++

Cláusulas de Escopo

- Compartilhado: shared (varlist)
  - Declara variáveis que têm uma localização de memória única e comum para todos os threads
  - Todos os threads acessam a mesma posição de memória, assim o acesso deve ser cuidadosamente controlado

Cláusulas de Escopo

- Privado: private (varlist)
  - Declara variáveis para que cada thread tenha sua própria cópia
  - Variáveis privadas só existem na região paralela
  - O valor é indefinido no início da região paralela
  - O valor é indefinido após o final da região paralela
  - Obs: Deve ser possível determinar o tamanho da variável antes de entrar na região paralela
    - Tipos primitivos ou vetores alocados estaticamente
      - Nada de malloc (C) ou new (C++)

Cláusulas de Escopo

- Redução (redn_oper: varlist)
  - Declara que uma variável escalar estará envolvida em uma operação de redução
  - Em Fortran, os operadores de redução podem ser +, *, -, AND, OR, EQV, NEQV, MAX, MIN, I e IOR, IEOR...
  - Em C / C + +, os operadores de redução pode ser +, *, -, &, |, ^, & &, | |

Exemplo: Redução

```
!$omp parallel do reduction(+:sum)
    do i=1,ndata
          a(i) = a(i) * a(i)
          sum = sum + a(i)
    end do
```

```
#pragma omp parallel for reduction(+:sum)
   for (i=0; i<ndata; i++){ a[i]=a[i]*a[i]; sum+=a[i];}
```

Cláusulas de Escopo

- Primeira privada: firstprivate (varlist)
  - Declara variáveis para que cada thread tenha a sua própria cópia
  - A cópia de cada thread é inicializada com o valor do thread mestre antes de entrar no loop
  - Pode ser mais eficiente do que declarar algumas variáveis a serem compartilhadas

Clausulas de Escopo

- Última privada: lastprivate (varlist)
  - Carrega variáveis do thread mestre com o último valor definido pela última iteração do loop
  - Se você não faz a última iteração, o comportamento é indefinido

Cláusulas de Escopo

- **DEFAULT:**
  - Sintaxe: default(shared|none)
- Não pode ter default (private) devido a requisitos das funções do próprio OpenMP
  - Muda o escopo padrão de variáveis cujo escopo não foi definido através das cláusulas acima
  - default(none) ajuda encontrar problemas de escopo forçando definição explícita do escopo

Cláusulas de Escopo

- Distribuição: schedule (tipo, bloco)
  - Controla distribuição de iterações entre threads em laços for
  - Tipo pode ser:
    - Static: blocls atribuídos sequencialmente aos threads
    - Dynamic: blocos são determinados dinâmicamente
    - Guided: Primeiro distribui blocos grandes e depois blocos menores entre threads
    - Runtime: distribuição é determinada pela variável de ambiente OMP_SCHEDULE
  - Bloco é um inteiro positivo

Guia para Atributos de Escopo

- Considere o código abaixo:

```
int x, temp, i, k, n;
float a[N], b[N], y[N], soma;

float soma = 0.0;
#pragma omp parallel for shared (??) private (??) ??
for(i = 0; i < n; i++) {
  a[i] = x;
  b[i] = y[k];
  temp = a[i] * b[i];
  soma = soma + temp;
}
printf("soma: %f\n", soma);
```

Guia para Atributos de Escopo

1 – Faça uma lista de todas as variáveis no loop:

soma, i, n, a, x, b, y, k, temp

2 – variáveis indexadas pelo índice do parallel for devem ser SHARED. Veja A e B

a) O indice só pode ser a variável do for (i) se tiver um a[i -1] pode impedir paralelização

b) Variáveis podem aparecer dos dois lados de uma atribuição

c) Cada thread OpenMP acessa elementos diferentes baseado na distribuição (schedule).

Os threads não estarão nunca na mesma iteração

Ou seja, no mesmo i

Guia para Atributos de Escopo

3 - Variáveis não indexadas pelo loop paralelizado

a) Variáveis do lado direito de atribuições são SHARED
   - Ex: n, x, y e k
   - Cada thread acessa a mesma variável. Como o acesso é de leitura não gera condição de disputa.

b) Variáveis usadas em reduções são compartilhadas
   - Ex: soma
   - Variável é inicializada antes do loop e usada após

Guia para Atributos de Escopo

c) Variáveis definidas dentro do loop são PRIVATE

- ex: temp
- A variável não é usada antes ou após o loop
- Compilador cria uma variável para cada thread

4 – O índice do loop é controlado pelo compilador portanto não deve aparecer no SHARED ou no PRIVATE. O compilador irá ignorar e fazer o que é certo

ex: i

Guia para Atributos de Escopo

- Portanto a diretiva correta fica:

```
#pragma omp parallel for shared
  (a, b, n, x, y, k, soma)
  private (temp)
  reduction(+:soma)
```

- Já que o SHARED é padrão:

```
#pragma omp parallel for
  private (temp)
  reduction(+:soma)
```

- Se não for possível usar o exemplo acima o loop precisa de uma análise mais profunda ou pode não ser paralelizável

SINCRONIZAÇÃO

Construções

#pragma omp master

#pragma omp critical [(name)]

#pragma omp barrier

#pragma omp atomic

#pragma omp flush [(list)]

#pragma omp ordered

MASTER

```
#pragma omp master
    bloco estruturado
```

- O bloco é executado apenas pelo thread mestre
- Não há uma barreira antes ou após essa diretiva

CRITICAL

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

- Apenas um thread por vez pode executar o bloco
- Não há como determinar a ordem de entrada na região crítica pelos threads

BARRIER

```
#pragma omp barrier
```

- Sincroniza todos os threads num time
- Para a execução do programa até que todos os threads se cheguem nesse ponto de execução

ATOMIC

```
#pragma omp atomic
    expressão
```

- Caso especial do CRITICAL
- Protege uma variável escalar para que não seja modificada por múltiplos threads ao mesmo tempo
- Funciona como um mutex

FLUSH

```
#pragma omp flush [(list)]
```

- Obriga escrita na memória com dados que estejam em cache ou bufferizadas pelo OpenMP

ORDERED

```
#pragma omp ordered
    bloco estruturado
```

- Ordena um laço for ou while
- É como se estivesse executando em serialmente (1 processador)

ASPECTOS PRÁTICOS

Sugestões de Uso

- Não use cláusulas se possível – ao invés disso
  - Declare variáveis privadas dentro das regiões paralelas
- Não utilize a sincronização, se possível – em especial a “flush"
- Não use travas se possível - se tiver que usar, certifique-se em destravá-las

Sugestões de Uso

- Use apenas construções paralelas para for e do
  - As outras situações provavelmente irão causar lentidão
- Distribuição
  - Use STATIC a menos que o balanceamento de carga seja um problema real
    • Se o balanceamento de carga é um problema, use STATIC com BLOCOS
  - Usar DYNAMIC ou GUIDED exigem uma grande sobrecarga

Overheads Típicos

- Exibir o nro do thread: 10-50
- Do/For estático: 100-200 (STATIC)
- Barreira: 200-500
- Do/For dinâmico: 1000-2000 (DYNAMIC)
- Comando ordenado: 5000-10000

Sugestões de Uso

- Conheça sua aplicação
- Use um profiler para encontrar gargalos

Problemas Comuns

- Paralelismo de granularidade fina demais
- Excesso de sincronização
- Desequilíbrio de carga
- Compartilhamento verdadeiro
  - Evitar ping pong em cache
- Compartilhamento Falso
  - Não usa cache
- Condições de Disputa
- Deadlocks