Apostila de Introdução ao OpenMP

INE5645 – Programação Paralela e Distribuída

Prof. João Bosco M. Sobral

Março de 2017

Quando você executa um programa sequencial

As instruções são executadas em **1 núcleo** e **os outros núcleos ficam ociosos**.

Existe perda de recursos processamento disponíveis.

Dependendo da aplicação (jogos, por exemplo), o melhor é que todos os núcleos do processador sejam usados para programa. Ou seja, o programa seja implementado em paralelo.

COMO?

Utilizando a **API padrão para memória compartilhada** para implementar Aplicativos Paralelos em C, C ++ e Fortran. A API do OpenMP implementa programas paralelos.

A API do OpenMP consiste em:

- ✓ Diretivas do compilador
- ✓ Funções de tempo de execução
- ✓ Variáveis de ambiente

Aprendendo a usar OpenMP

A estrutura **Open Multiprocessing** (**OpenMP**) é extremamente eficiente, que ajuda a aproveitar os benefícios de sistemas com mais de um núcleo por processador, para aplicativos C, C++ e Fortran.

Aqui é explicado como usar os recursos do OpenMP em código C/C++ e traz alguns exemplos que podem ajudar o leitor a começar a usar o OpenMP.

A estrutura OpenMP é uma maneira eficiente de fazer programação paralela em C, C++ e também na linguagem Fortran (IBM,1957), uma linguagem muito usada na Engenharia.

O GNU Compiler Collection (GCC) versão 4.2 tem suporte para o padrão OpenMP 2.5, enquanto o GCC v4.4 tem suporte para o padrão OpenMP 3, mais recente.

Outros compiladores, como o Microsoft® Visual Studio, também têm suporte para OpenMP. Agora você, aprenderá a usar *pragmas* de compilador da estrutura OpenMP, a encontrar suporte para algumas das interfaces de programação de aplicativos (APIs) que ela fornece e a testá-la com alguns algoritmos paralelos. Esta apostila usa GCC 4.2 como o compilador preferencial.

O seu primeiro programa de OpenMP

Vamos começar com um simples programa para exibir a frase **Hello, World!**, que inclui um pragma adicional. A Listagem 1 mostra o código.

Listagem 1. Hello World com OpenMP

```
#include <iostream>
int main()
{
  #pragma omp parallel
  {
    std::cout << "Hello World!\n";
  }
}
```

Quando o código da Listagem 1 é compilado e executado

com g++ (um compilador de C++), uma única mensagem **Hello, World!** deve ser exibida no console.

Agora, recompile o código com a opção `-fopenmp`.

A Listagem 2 mostra a saída.

Listagem 2. Compilando e executando o código com o comando com o g++ e a opção `-fopenmp`

```
tintin$ g++ test1.cpp -fopenmp
tintin$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Um ótimo recurso do OpenMP é que não é necessário nada além da instalação padrão do gcc. Programas com OpenMP devem ser compilados com a opção `-fopenmp`.

Listagem 3. Compilando e executando o código com o comando com o gcc e a opção `-fopenmp`

```
tintin$ gcc omp_hello.c -fopenmp -o hello
tintin$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

O que aconteceu?

O pragma OpenMP

```
#pragma omp parallel
```

funciona apenas quando a opção `-fopenmp` do compilador é especificada.

Internamente, durante a compilação, o gcc (GNU Compiler Collection) gera o código para criar o máximo número de encadeamentos (*threads*) possível no tempo de execução, com base no hardware e na configuração do sistema operacional.

A rotina de início de cada encadeamento (thread) é o código no bloco após o *pragma*. Esse comportamento chama-se ***paralelização implícita***.

Na sua essência, OpenMP consiste em um conjunto de *pragmas* eficientes que livram o desenvolvedor da obrigação de incluir muitos códigos repetidos.

Para comparação, confira com o que você conhece em **pthreads**. os encadeamentos da Interface de Sistema Operacional Portátil (POSIX).)

Como eu estou usando um computador com um processador Intel® Core i7, com 4 núcleos físicos e 2 núcleos lógicos por núcleo físico, a saída da Listagem 3 parece adequada (**8 encadeamentos = 8 núcleos lógicos**).

Núcleos físicos

Os processadores possuem núcleos internos, que são os responsáveis por processar as informações. Os núcleos físicos, como o próprio nome já diz, existem fisicamente dentro do processador, o que o torna o processador uma unidade de processamento com núcleos reais.

Núcleos lógicos

Diferentemente dos núcleos físicos, os núcleos lógicos não existem fisicamente dentro do processador. Na realidade um núcleo físico é “dividido” por uma forma lógica de processamento, emulando assim um núcleo lógico.

O uso de núcleos lógicos, apesar de dividir de certa forma o processamento de um núcleo físico, permite o aumento de desempenho do processador, pois a capacidade de processamento de cada núcleo (não importando se ele é físico ou lógico) é “independente”, sendo assim o processamento de diversas tarefas acaba ganhando desempenho.

A Intel tem uma tecnologia chamada *Hyper-threading*, onde ele emula um segundo nucleo. Pelo menos o processamento fica em torno de 30% a 40% mais rápido.

Todos os computadores (Dual core, quad core, six core, etc...) vem apenas com 1 núcleo ativado, ou seja, você tem que ativar os outros núcleos. Veja o tutorial a seguir se precisar ativar os núcleos restantes.

[Tutorial] Como ativar todos os núcleos do processador
http://www.clubedohardware.com.br/forums/topic/1136929-tutorial-como-ativar-todos-os-núcleos-do-processador-cpu/

Um encadeamento é uma forma de estrutura computacional em termos de valores e sequências de computação usando estes valores. Ela permite a definição de *building blocks* que são sequências de computação, além disso ela determina como combinar computações e formar uma nova, esse encadeamento pode ser uma **composição**, onde a saída de uma computação é a entrada de uma outra. Mas, **no contexto do OpenMP os encadeamentos podem ser entendidos como as threads.**

Threads OpenMP e Cores (Núcleos)

Thread é uma sequência independente de execução de código de programa. Bloco de código com uma entrada e uma saída Para nossos propósitos um conceito mais abstrato, sem relação com Cores / CPUs (núcleos). **Threads OpenMP são mapeadas em núcleos físicos**. É possível mapear mais de 1 thread em um núcleo. Na prática é melhor ter um-para-um mapeamento.

Agora vamos ver mais detalhes sobre *pragmas* paralelos.

Usando com OpenMP paralelo

Para controlar o número de encadeamentos, basta usar o argumento `num_threads` do pragma.

Aqui está novamente o código da Listagem 1 com o número de encadeamentos disponíveis especificados como 5 (como mostra a Listagem 4).

Listagem 4. Controlando o número de encadeamentos com `num_threads`

```
#include <iostream>
int main()
{
  #pragma omp parallel num_threads(5)
  {
    std::cout << "Hello World!\n";
  }
}
```

Em vez do atributo num_threads, aqui está uma outra alternativa para alterar o número de encadeamentos (threads) executando o código.

Agora você terá contato com a primeira **API do OpenMP** que você irá usar: omp_set_num_threads. Essa função é definida no arquivo de cabeçalho omp.h.

Não é necessário vincular outras bibliotecas para que o código na Listagem 4 funcione — apenas a opção do compilador -fopenmp.

Listagem 4. Uso de omp_set_num_threads para ajuste fino da criação de encadeamentos (*threads*)

```
#include <omp.h>
#include <iostream>
int main()
{
  omp_set_num_threads(5);
  #pragma omp parallel
  {
    std::cout << "Hello World!\n";
  }
}
```

O OpenMP também usa **variáveis de ambiente externas** para controlar seu comportamento.

É possível alterar o código na Listagem 2 para imprimir apenas **Hello World!** seis vezes, definindo a variável OMP_NUM_THREADS para 6. A Listagem 5 mostra a execução.

No ambiente LINUX/bash, no prompt de comando colocar:

```
export OMP_NUM_THREADS = <número de threads>
```

Listagem 5. Usando variáveis de ambiente para ajustar o comportamento do OpenMP

```
tintin$ export OMP_NUM_THREADS=6
```

```
tintin$ export OMP_NUM_THREADS=6
tintin$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Você já viu sobre as três facetas do OpenMP:

- ✓ pragmas do compilador (#pragma omp)
- ✓ APIs do tempo de execução (omp_set_num_threads)
- ✓ variáveis de ambiente

export OMP_NUM_THREADS = <número de threads>

O que acontece se usarmos a variável de ambiente e a API do tempo de execução? **A API tem precedência mais alta.**

Um exemplo prático

O OpenMP usa técnicas de paralelização implícita, e é possível usar pragmas, funções explícitas e variáveis de ambiente para instruir o compilador.

Vamos examinar um exemplo no qual OpenMP pode ser de grande ajuda. Considere o código da Listagem 6.

Lista 6. Processamento sequencial em um loop `for`

```
int main( )
{
int a[1000000], b[1000000];
// ... some initialization code for
populating arrays a and b;

int c[1000000];

for (int i = 0; i < 1000000; ++i)
  c[i] = a[i] * b[i] + a[i-1] * b[i+1];

// ... now do some processing with array c
 }
```

Claramente, é possível **dividir o loop `for` e executar em mais de um núcleo**.

O cálculo de qualquer **`c[k]`** é independente dos outros elementos do array **`c`**.

A Listagem 7 mostra como o OpenMP ajuda a fazer isso.

Listagem 7. Processamento paralelo em um loop `for` com o pragma `parallel for`

```
int main( )

{

int a[1000000], b[1000000];

// ... some initialization code for
populating arrays a and b;

int c[1000000];

#pragma omp parallel for
for (int i = 0; i < 1000000; ++i)
  c[i] = a[i] * b[i] + a[i-1] * b[i+1];
```

```
// ... now do some processing with array c

}
```

O #pragma parallel for ajuda a dividir a carga de trabalho do loop for em mais de um encadeamento (thread). Cada encadeamento (thread) pode ser executado em um núcleo diferente, o que reduz significativamente o tempo total de cálculo. A Listagem 8 prova isso.

Listagem 8. Entendendo omp_get_wtime

```
#include <omp.h>
#include <math.h>
#include <time.h>
#include <iostream>
int main(int argc, char *argv[])
{
   int i, nthreads;
   clock_t clock_timer;
   double wall_timer;
   double c[1000000];
   for (nthreads = 1; nthreads <= 8; ++nthreads)
    {
     clock_timer = clock();
     wall_timer = omp_get_wtime();
     #pragma omp parallel for private(i) num_threads(nthreads)
      for (i = 0; i < 1000000; i++)
          c[i] = sqrt(i * 4 + i * 2 + i);
      std::cout << "threads: " << nthreads <<  " time on clock(): " <<
          (double) (clock() - clock_timer) / CLOCKS_PER_SEC
        << " time on wall: " <<  omp_get_wtime() - wall_timer << "\n";
    }
}
```

Na Listagem 8, o código aumenta continuamente o número de encadeamentos para medir quanto tempo o loop `for` interno leva para ser executado.

A API `omp_get_wtime` retorna o tempo em segundos a partir de algum ponto arbitrário, porém consistente.

Portanto, (`omp_get_wtime()-wall_timer`) **retorna o tempo real para execução do loop `for`.**

A chamada do sistema `clock()` é usada para estimar o tempo de uso do processador para todo o programa — ou seja, **o tempo de uso de processador dos encadeamentos (threads) individuais** é somado antes de a chamada informar o número final.

No meu computador Intel Core i7, a Listagem 9 mostra as informações exibidas.

Listagem 9. Números para a execução do loop `for` interno

```
threads: 1 time on clock(): 0.015229 time on wall: 0.0152249
threads: 2 time on clock(): 0.014221 time on wall: 0.00618792
threads: 3 time on clock(): 0.014541 time on wall: 0.00444412
threads: 4 time on clock(): 0.014666 time on wall: 0.00440478
threads: 5 time on clock(): 0.015940 time on wall: 0.00359988
threads: 6 time on clock(): 0.015069 time on wall: 0.00303698
threads: 7 time on clock(): 0.016365 time on wall: 0.00258303
threads: 8 time on clock(): 0.016780 time on wall: 0.00237703
```

Embora o tempo de processador seja quase o mesmo em todas as execuções (como deveriam ser, exceto por algum tempo adicional para criar os encadeamentos e o comutador de contexto), o que nos interessa é o tempo real. Este é reduzido progressivamente à medida que o número de encadeamentos aumenta, dando a entender que os dados estão sendo calculados pelos núcleos em paralelo.

Uma nota final sobre a sintaxe do pragma:

`#pragma parallel for private(i)`

significa que **a variável de loop `i` deve ser tratada como um armazenamento local de encadeamento** (thread), com cada encadeamento tendo uma cópia da variável. A variável local do encadeamento não é inicializada.

Controle de baixa granularidade sobre a execução de tarefas

Você já viu que todos os encadeamentos (threads) executam o bloco de códigos após `#pragma omp parallel` em paralelo.

É possível categorizar ainda mais o código dentro desse bloco para ser executado por encadeamentos selecionados (seções paralelas). Considere o código da Listagem 12.

Listagem 10. Aprendendo a usar o pragma de seções paralelas

```
int main( )
{
  #pragma omp parallel
  {
    cout << "All threads run this\n";

    #pragma omp sections
    {
      #pragma omp section
      {
        cout << "This executes in parallel\n";
      }
      #pragma omp section
      {
        cout << "Sequential statement 1\n";

        cout << "This always executes after statement 1\n";
      }
      #pragma omp section
      {
        cout << "This also executes in parallel\n";
      }
    }
  }
}
```

O código que vem antes de `#pragma omp sections`, mas logo após `#pragma omp parallel`

```
cout << "All threads run this\n";
```

é executado por todos os encadeamentos em paralelo.

O bloco que vem depois de `#pragma omp sections` é classificado ainda mais em subseções individuais usando `#pragma omp section`.

Cada bloco `#pragma omp section` está disponível para ser executado por um encadeamento (thread) individual. No entanto, as instruções individuais dentro do bloco de seção são sempre executadas em sequência.

A Listagem 11 mostra a saída do código da Listagem 10.

Listagem 13. Saída do código da Listagem 12

```
tintin$ ./a.out
All threads run this
All threads run this
All threads run this
All threads run this
All threads run this
All threads run this
All threads run this
All threads run this
This executes in parallel
Sequential statement 1
This also executes in parallel
This always executes after statement 1
```

Na Listagem 13, temos novamente 8 encadeamentos sendo criados inicialmente.

Desses 8 encadeamentos, há trabalho suficiente para apenas três deles no bloco `#pragma omp sections`.

Na segunda seção, especificamos a ordem na qual as instruções de impressão são executadas. É esse o motivo para usar o `#pragma sections`. Se for necessário, é possível especificar a ordem dos blocos de códigos.

Outro Exemplo com Seções Paralelas

```
******************************************************************************
#include
#include
#include
#define N 50 int
main (int argc, char *argv[])
{ int i, nthreads, tid;
  float a[N], b[N], c[N], d[N];

/* Some initializations */

for (i=0; i<N; i++) {

    a[i] = i * 1.5;

    b[i] = i + 22.35;

    c[i] = d[i] = 0.0;

 }

#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)

{

   tid = omp_get_thread_num();

   if (tid == 0)

  {

       nthreads = omp_get_num_threads();

       printf("Number of threads = %d\n", nthreads);

   }

   printf("Thread %d starting...\n",tid);
```

```
#pragma omp sections nowait
{
  #pragma omp section
  {
    printf("Thread %d doing section 1\n",tid);
    for (i=0; i<N; i++)
     {
       c[i] = a[i] + b[i];
       printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
     }
  }
  #pragma omp section
  {
       printf("Thread %d doing section 2\n",tid);
       for (i=0; i<N; i++)
       {
         d[i] = a[i] * b[i];
         printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
       }
  }
} /* end of sections */
 printf("Thread %d done.\n",tid);
```

```
  } /* end of parallel section */

}
```

Seções críticas (regiões críticas) com OpenMP

Em OpenMP não é necessário criar explicitamente uma *mutex*, mas ainda é preciso especificar a seção crítica.

Aqui está a sintaxe:

```
#pragma omp critical (optional section name)
{
// no 2 threads can execute this code block concurrently
}
```

O código que vem depois de `#pragma omp critical` pode apenas ser executado por um único encadeamento (única thread) em um dado momento.

Além disso, um `optional` *`section name`* é um identificador global, e **dois encadeamentos (duas threads) não podem executar seções críticas com o mesmo identificador global ao mesmo tempo**.

Considere o código da Listagem 14.

Listagem 14. Mais de uma seção crítica com o mesmo nome

```
#pragma omp critical (section1)
{
  myhashtable.insert("key1", "value1");
}

// ... other code follows
```

```
#pragma omp critical (section1)
{
  myhashtable.insert("key2", "value2");
}
```

Com base nesse código, podemos supor com segurança que as duas inserções de *hashtable* nunca acontecerão simultaneamente, pois **os nomes da seção crítica são os mesmos**.

Isso é um pouco diferente da maneira com que você está acostumado a lidar com seções críticas usando *pthreads*, que são, em grande parte, caracterizadas pelo uso (ou abuso) de bloqueios.

Bloqueios e mutexes com OpenMP

Exclusão mútua. ... Exclusão mútua (também conhecida pelo acrônimo **mutex** para mutual exclusion, o termo em inglês) **é** uma técnica usada em programação concorrente para evitar que dois processos ou threads tenham acesso simultaneamente a um recurso compartilhado, acesso esse denominado por seção crítica.

Mutexes são mecanismos utilizados para implementar exclusão mútua em threads.

Um algoritmo de exclusão mútua serve para garantir que regiões críticas de código não sejam executadas simultaneamente, protegendo estruturas de dados compartilhadas de modificações simultâneas.

pthread_mutex_t lock;

pthread_mutex_init (&lock, NULL);

```
pthread_mutex_lock (&lock);
    /* Executa região crítica */
pthread_mutex_unlock(&lock);
```

Curiosamente, OpenMP tem suas próprias versões de **mutexes** (então, não se trata apenas de pragmas).

Seja bem-vindo ao `omp_lock_t`, definido como **parte do arquivo de cabeçalho omp.h**.

As operações de **mutex** no estilo *pthread* são iguais — até os nomes das APIs são semelhantes.

Há cinco funções de APIs que o desenvolvedor deve conhecer:

- `omp_init_lock()`: Essa deve ser a primeira API a acessar `omp_lock_t`. É usada para inicialização. Observe que, logo após a inicialização, considera-se que o bloqueio não foi definido.

- `omp_destroy_lock()`: Essa API destrói o bloqueio. O bloqueio deve estar no estado não definido para que essa API seja chamada, o que significa que não é possível chamar `omp_set_lock` e depois fazer uma chamada para destruir o bloqueio.

- `omp_set_lock()`: Essa função API define `omp_lock_t`— ou seja, o *mutex* é adquirido. Se um encadeamento não puder definir o bloqueio, ele continuará a aguardar até que possa.

- `omp_test_lock()`: Essa função API tenta bloquear, se o bloqueio estiver disponível, retorna 1 em caso de sucesso e 0 em caso de fracasso. Essa é uma *API sem bloqueio* —

ou seja, essa função não faz o encadeamento (a thread) aguardar para definir o bloqueio.

- `omp_unset_lock()`: Essa função API libera o bloqueio.

A Listagem 11 mostra uma implementação trivial de uma fila legada de um encadeamento, estendida para lidar com multiencadeamento usando bloqueios do OpenMP.

Observe que isso pode não ser o ideal para todas as situações. O exemplo é apenas uma ilustração rápida.

Listagem 11. Usando OpenMP em C++ para estender uma fila de um encadeamento

```
#include <openmp.h>
#include "myqueue.h"

class omp_q : public myqueue<int> {
public:

   typedef myqueue<int> base;
   omp_q( ) {
      omp_init_lock(&lock);
   }
   ~omp_q() {
       omp_destroy_lock(&lock);
   }
   bool push(const int& value) {
      omp_set_lock(&lock);
      bool result = this->base::push(value);
      omp_unset_lock(&lock);
      return result;
   }
   bool trypush(const int& value)
   {
       bool result = omp_test_lock(&lock);
       if (result) {
          result = result && this-
>base::push(value);
          omp_unset_lock(&lock);
       }
       return result;
```

```
    }
    // likewise for pop
private:
    omp_lock_t lock;
};
```

Lembrar:

Um semáforo é uma extensão de um mutex. Um mutex permite que um segmento no interior da secção crítico, um semáforo permite tópicos N numa seção crítica (quando o número n é dada como um parâmetro no inicialização). Um semáforo é útil quando um recurso tem mais de uma instância, e um mutex pode ser implementado por inicializar um semáforo com o valor 1.

Uma descrição sobre o semáforo e mutex: "Quando os semáforos são usados para exclusão mútua de manutenção de vários processos de execução dentro de uma seção crítica simultaneamente, o seu valor será inicialmente definido para 1 um semáforo pode ser. realizada apenas por um único processo ou thread a qualquer momento. Um semáforo usado neste modo é chamado às vezes um mutex, que é, naturalmente, uma abreviação de "exclusão mútua." Quase todos os semáforos encontrados no kernel do Linux são usados para exclusão mútua. "

EXAMPLE – Multiplicação de Matrizes

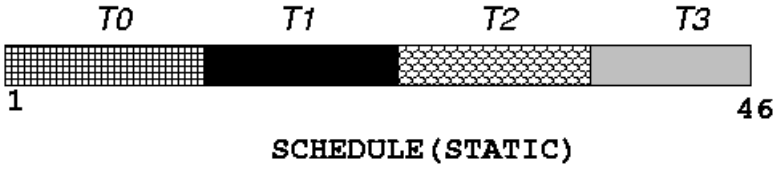
```
/******************************************************************
* FILE: omp_mm.c Matrix multiply
* DESCRIPTION:
* OpenMp Example - Matrix Multiply - C Version
* Demonstrates a matrix multiply using OpenMP. Threads share row
iterations
* according to a predefined chunk size.
* AUTHOR: Blaise Barney
* LAST REVISED: 06/28/05
******************************************************************
/
#include <omp.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#define NRA 62 /* number of rows in matrix A */
#define NCA 15 /* number of columns in matrix A */
#define NCB 7 /* number of columns in matrix B */
int main (int argc, char *argv[])
{
int tid, nthreads, i, j, k, chunk;
double a[NRA][NCA], /* matrix A to be multiplied */
b[NCA][NCB], /* matrix B to be multiplied */
c[NRA][NCB]; /* result matrix C */
chunk = 10; /* set loop iteration chunk size */
/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
 {
    tid = omp_get_thread_num();
    if (tid == 0)
    {
       nthreads = omp_get_num_threads();
       printf("Starting matrix multiple example with %d threads\n",nthreads);
       printf("Initializing matrices...\n");
     }
 /*** Initialize matrices ***/
 #pragma omp for schedule (static, chunk)
 for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
          a[i][j]= i+j;
 #pragma omp for schedule (static, chunk)
 for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
          b[i][j]= i*j;
```

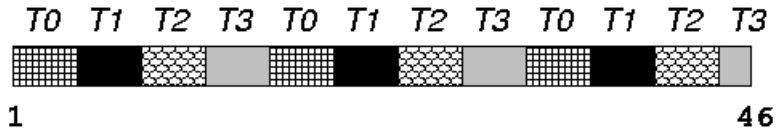
```
 #pragma omp for schedule (static, chunk)
 for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
       c[i][j]= 0;
/*** Do matrix multiply sharing iterations on outer loop ***/
/*** Display who does which iterations for demonstration purposes ***/
printf("Thread %d starting matrix multiply...\n",tid);
 #pragma omp for schedule (static, chunk)
 for (i=0; i<NRA; i++)
   {
     printf("Thread=%d did row=%d\n",tid,i);
     for(j=0; j<NCB; j++)
       for (k=0; k<NCA; k++)
       c[i][j] += a[i][k] * b[k][j];
    }
 } /*** End of parallel region ***/
/*** Print results ***/
printf("******************************************************\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
   {
     for (j=0; j<NCB; j++)
     printf("%6.2f ", c[i][j]);
     printf("\n");
   }
printf("******************************************************\n");
printf ("Done.\n");
}
```

SCHEDULE - Descreve como as iterações do “loop” serão divididas por entre as “threads”:

STATIC - O “loop” será dividido em pedaços de tamanho “chunk” e distribuído estaticamente para cada “thread” (por ordem de “thread”). Se o parâmetro “chunk” não for definido, o compilador irá dividir o “loop” de maneira contínua e em partes iguais (quando possível), para cada “thread”.



SCHEDULE(STATIC)



SCHEDULE(STATIC,4)

Observações

1- Se o atributo **nowait** (C/C++) ou NO WAIT (Fortran) for especificado, as “threads” não serão sincronizadas no final do “loop” paralelo, continuando a execução para o próximo comando.

2- Não é possível desviar para for a de um “loop” associado a diretiva DO/for.

O que segue está em: https://pt.wikipedia.org/wiki/OpenMP

schedule (type chunk): É útil se a construção de partilha for um *do-loop* ou *for-loop*. A iteração, ou as iterações, na construção de partilha são atribuídas para as threads de acordo com o método programado definido nesta cláusula. Os três tipos de programação para *type* são:

1. *static*: Aqui, a todos os threads são atribuídas as iteração antes de executar o laço de repetição. As interações são divididas igualmente entre os threads por padrão. No entanto, especificando um inteiro para um parâmetro "chunk" fixará um número "chunk" de iterações sequenciadas a uma determinada lista de threads.

2. *dynamic*: Aqui, algumas das iterações são atribuídas a um número menor de threads. Após o término da iteração atribuída a um thread em particular, ele retorna para buscar uma das iterações restantes. O parâmetro "chunk" define o número de iterações sequenciais que são atribuídas a um thread por vez.

3. *guided*: Um grande "chunk" de iterações sequenciadas são atribuídos a cada thread dinamicamente (como acima). O tamanho do "chunk" diminui exponencialmente com cada atribuição sucessiva até um tamanho mínimo especificado no parâmetro *chunk*.

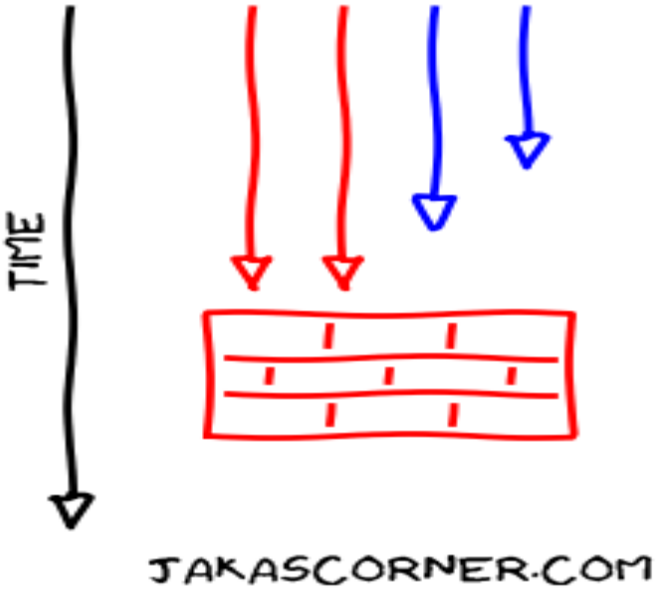
OpenMP: Barrier

http://jakascorner.com/blog/2016/07/omp-barrier.html

O que é uma barreira?

É um ponto na execução de um programa onde os threads esperam um para o outro. Nenhuma thread é permitido continuar até que todos os segmentos em uma equipe (team) alcançar a barreira.

Basicamente, uma barreira é um ponto de sincronização em um programa. Podemos visualizá-lo com uma parede.



Na figura, os fios vermelhos estão esperando na parede para os fios azuis. Os fios vermelhos não podem ir além da parede. Eles só podem prosseguir quando todos os fios atingirem a parede.

Prós e contras

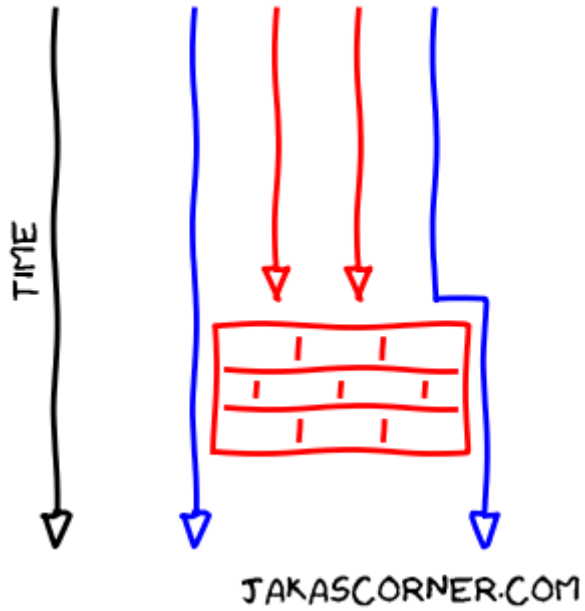
A principal razão para uma barreira em um programa é evitar corridas de dados e garantir a correção do programa.

Claro que existem algumas desvantagens. Cada sincronização é uma ameaça para o desempenho. Quando um segmento espera por outros threads, ele não faz nenhum trabalho útil e gasta recursos valiosos.

Outro problema pode ocorrer se não estivermos cuidadosamente inserindo barreiras. Assim que um fio atingir a barreira, então todos os fios da equipe devem alcançar a barreira. Caso contrário, os segmentos esperando na barreira vão esperar para sempre

(exceto se usarmos uma construção de cancelamento, mas este é um tópico para outro artigo).

A figura a seguir mostra como um par de fios azuis evita a barreira. Neste caso, os fios vermelhos esperarão para sempre os fios azuis.



Como adicionar uma barreira a um programa?

Podemos inserir explicitamente uma barreira em um programa adicionando a construção de barreira:

```
#pragma omp barrier
```

Esta é uma maneira explícita de adicionar uma barreira.

Há também muitas outras situações, onde um compilador insere uma barreira em vez de nós. Isso acontece porque muitas construções OpenMP implicam uma barreira. Por exemplo, a construção paralela implica uma barreira na extremidade da região paralela. A construção de loop implica uma barreira na

extremidade do loop. A construção única implica uma barreira no final da única região.

No entanto, existem também construções OpenMP que não implicam uma barreira. A construção mestre é tal exemplo. Essa construção é muito semelhante à **construção única**: **o código dentro da construção mestre é executado por apenas um thread (mestre)**. Mas a diferença é que a construção mestre não implica uma barreira enquanto a construção única faz.

Como podemos descobrir quais construções implicam uma barreira e quais não?

Temos de olhar para a especificação OpenMP. A descrição de cada construção contém as informações sobre a existência da barreira.

Evitando as barreiras implícitas

Uma questão natural que surge é: Podemos omitir as barreiras implícitas?

Isso depende das construções. Algumas construções suportam a remoção de uma barreira, enquanto as outras não suportam tal recurso. Novamente, a especificação do OpenMP pode nos dizer se uma construção suporta esse recurso.

A construção de um loop suporta a remoção de uma barreira. Um programador pode então, omitir a barreira, adicionando nowait cláusula para a construção de loop.

```
#pragma omp parallel
{
    #pragma omp for nowait
```

```
    for (...)
    {
        // for loop body
    }

    // next instructions
}
```

Neste caso, a thread que termina de forma cedo prossegue direto para a próxima instrução e não espera pelos outros threads na equipe.

Usar a cláusula nowait pode melhorar o desempenho de um programa. Mas devemos ter cuidado, porque remover uma barreira pode introduzir uma corrida de dados.

Um Exemplo

No artigo sobre a **construção única**, apresentamos vários programas que acumulam os salários de todos os funcionários em duas empresas. A terceira versão era a seguinte:

```
#pragma omp parallel shared(salaries1, salaries2)
{
    #pragma omp for reduction(+: salaries1)
    for (int employee = 0; employee < 25000; employee++)
    {
        salaries1 += fetchTheSalary(employee, Co::Company1);
    }
```

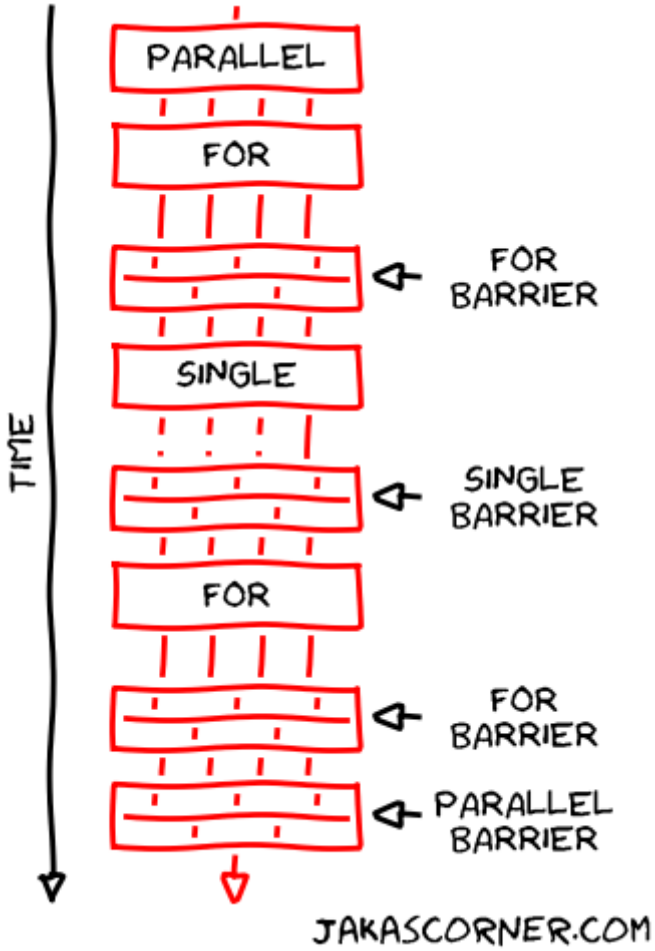
```
    #pragma omp single
    {
        std::cout << "Salaries1: " << salaries1 << std::endl;
    }

    #pragma omp for reduction(+: salaries1)
    for (int employee = 0; employee < 25000; employee++)
    {
        salaries2 += fetchTheSalary(employee, Co::Company2);
    }
}
std::cout << "Salaries2: " << salaries2 << std::endl;
```

Esta exemplo sofre de *oversynchronization* (lido como: tem muitas barreiras). Portanto, agora explicamos o problema com o programa e diferentes soluções para o problema.

A chave é perceber onde estão as **barreiras implícitas**. Eles são

- Na extremidade da região paralela,
- No final do primeiro for loop,
- No final da construção única,
- No final do segundo for loop.



Vamos analisar cada barreira.

A primeira barreira está no final do primeiro loop for. Se omitimos a barreira lá, poderíamos introduzir uma corrida de dados. Isso ocorre porque a próxima instrução após o loop for, acessa a variável de redução: **salaries1**. Sem a barreira, um segmento poderia acessar **salários1** para impressão, enquanto algum outro segmento ainda pode atualizar o valor dos **salários1**. Portanto, não devemos adicionar **nowait** cláusula para o primeiro para loop.

A segunda barreira está na extremidade da construção única. Na construção única, o programa imprime o valor dos **salários1**. Esta é a última vez que o programa lê/grava **salários1**. As próximas instruções já calculam **salários2**.

Devido a esta independência, podemos remover com segurança a barreira no final da construção única. Podemos fazer isso inserindo a cláusula **nowait**.

Há também outra opção. Podemos substituir a construção única pela construção principal. A construção mestre é muito semelhante à construção única. As principais diferenças são que a construção mestre é executada pelo thread principal e que a construção mestre não implica uma barreira.

Existem mais duas barreiras. Ambos estão no final da região paralela. A construção paralela não suporta a cláusula **nowait**. Assim, a única possibilidade de eliminar a barreira está no final do segundo ciclo. A eliminação não introduz uma corrida de dados, porque existe a barreira da construção paralela, que sincroniza os fios. Portanto, é seguro omitir a barreira implícita no final do segundo ciclo. Observe que um compilador pode fazer isso automaticamente.