Executando OpenMP

Usaremos o Boot Remoto instalado na sala 6 do LIICT.

Alguns comando para Debian/Ubuntu:

Para saber se o pacote está disponível nos repositórios, abra um terminal e digite (use o comando abaixo):

- Debian/Ubuntu: **sudo apt-cache search <pacote>**

Para verificar se um pacote está instalado no Debian/Ubuntu:

- Debian/Ubuntu: **sudo dpkg -l | grep** `<pacote>`
- dpkg --get-selections | grep openmp

Instalando OpenMp no Ubuntu: **sudo apt-get install gcc-multilib**
Quando a instalação termina, você tem a API OpenMp instalado em sua máquina.

Em seu diretório home, crie um subdiretório para seus programas:

```
# cd /home

# mkdir openMP

# cd openMP
```

Então, copie a versão C dos arquivos de exercícios da versão OpenMP paralelo (fornecido no Tutorial), para seu subdiretório openMP

```
# cp /usr/global/docs/training/blaise/openMP/C/*
~/openMP
```

Conceitos OpenMP:

Sentinela: Em C/C++ #pragma omp

Em Fortran !$OMP

#pragma omp - é a funcionalidade básica de comunicar informação ao compilador de C/C++ de modo a este gerar código otimizado para o ambiente de execução do OpenMP.

Diretivas - Uma diretiva é uma linha especial de código fonte com significado especial apenas para determinados compiladores.

As diretivas do tipo #pragma permitem passar informação ao compilador **gcc** (GNU compiler).

Forma geral: #pragma omp **directive** [**clause**, ...]

A diretiva **parallel** é o construtor fundamental do OpenMP.

#pragma omp **parallel**

Estrutura Base de um Programa OpenMP

```
main() {
    ... // sequential region executed by master thread
    #pragma omp parallel // OpenMP parallel constructor
    { // master thread creates/launches the team of threads ...
      // parallel region executed by all threads
    } // team of threads sincronizes with master thread and terminates ...
    ... // sequential region executed by master thread
}
```

As definições da biblioteca OpenMP encontram-se em **omp.h** e a sua implementação em **libgomp.so**. Para compilar um programa com o OpenMP é necessário incluir o cabeçalho **#include** no início do programa e compilá-lo com a opção **-fopenmp**.

Listagem 1. Hello World com OpenMP

Supondo que estejamos usando um computador com um processador **Intel® Core i7**, com **quatro núcleos físicos** e **dois núcleos lógicos por núcleo físico**, a saída da Listagem 2, seguinte, parece adequada (8 encadeamentos = 8 núcleos lógicos).

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main ()
```

```
{

  /* Fork a team of threads */
  #pragma omp parallel
  {
    printf("Hello World!\n");
  }

}
```

Quando o código da **Listagem 1** é compilado e executado com **gcc**, com a opção **`-fopenmp`**. Mensagens **Hello, World!** devem ser exibidas no console.
A **Listagem 2** mostra a saída:

Listagem 2. Compilando e executando o código com o comando `-fopenmp`

```
$ gcc -fopenmp omp-hello.c
$ ./…     (aqui executa)
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

#pragma omp **parallel** [clause, ...]

clause - permite especificar informação adicional sobre uma diretiva.

Listagem 3 – Usa as cláusulas if, num_threads, default, private, shared

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

int main() {

   int n = NTHREADS, tid = -1;

   #pragma  omp  parallel if(n >= 1) num_threads(n) \
                        default(none)  private(tid)  shared(n)

   {
       // Obtem o identificador de thread
       tid = omp_get_thread_num();

       printf("Thread %d: Hello!\n", tid);
```

```
        if (n != omp_get_num_threads())
            printf("Error: NTHREADS\n");

    }

    printf("Thread %d: Bye!\n", tid); }

}
```

Se executarmos com **3** threads (NTHREADS) obtemos o seguinte output:

Thread 1: Hello!
Thread 0: Hello!
Thread 2: Hello!
Thread -1: Bye!

Explicando as cláusulas

if(expr) - Cláusula que significa executar em paralelo se a expressão expr for avaliada como verdade. Caso contrário, a execução é sequencial (apenas a master thread).

num_threads(expr) - Executa em paralelo com um número de threads igual ao resultado da avaliação da expressão expr.

private(list) - Cláusula que define que as variáveis definidas em list são duplicadas em cada thread e o seu acesso passa a ser local (privado) em cada thread. O valor inicial das variáveis privadas é indefinido (não é iniciado) e o valor final das variáveis originais (depois da região paralela) também é indefinido.

shared(list) – Cláusula que define sobre as variáveis definidas em list são partilhadas por todos os threads ficando à responsabilidade do programador garantir o seu correto manuseamento. Por omissão, as variáveis para as quais não é definido qualquer tipo são consideradas variáveis partilhadas.

default(none) - Cláusula que define que o tipo de todas as variáveis envolvidas na região paralela deve ser declarado explicitamente (sobrepõe-se à definição de que, por omissão, as variáveis são consideradas partilhadas).

`omp_get_thread_num()` – Função básica do OpenMP que retorna o identificador do thread corrente. Os N threads a executar numa região paralela são numerados de 0 a N-1 e o master thread é sempre identificado pelo número 0.

omp_get_num_threads() - Função básica do OpenMP que retorna o número de threads momentaneamente ativos. Se for chamada a partir duma região sequencial (executada apenas pelo master thread) retorna 1.

Listagem 4 - Exemplo OpenMP Hello World

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {

int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables
*/
#pragma omp parallel private(nthreads, tid)
 {

 /* Obtain thread number */
 tid = omp_get_thread_num();
 printf("Hello World from thread = %d\n", tid);

 /* Only master thread does this */
 if (tid == 0)
 {
  nthreads = omp_get_num_threads();
 printf("Number of threads = %d\n", nthreads);
 }

 } /* All threads join master thread and disband */

}
```

==========

Regiões Paralelas - Um código OpenMP que é executado por todas as threads dentro de uma região paralela. Ver em (3).

Seções Paralelas - Uma seção paralela OpenMP é um código OpenMP que permite blocos separados de código serem executados em paralelo (ex. diversas subrotinas independentes). Ver em (3).

Seções Críticas - Uma seção crítica é um bloco de código que só pode ser executado por uma thread por vez.

Encadeamento de Construtores - OpenMP deve suportar *nested parallelism, n*o caso de existirem vários níveis de encadeamento. Ver em (1).

Atributos de Dados Compartilhados

Em um programa OpenMP, os dados precisam receber um atributo ("labelled")

Há dois tipos básicos: **Shared** e **Private**

Shared V

- Só há uma instância do dado V.
- Todas as tarefas podem ler e escrever nesses dados concorrentemente ν
  - Exceções se construtores são usados ν.
- Todas as alterações são visíveis a todas as tarefas ν .
  - Mas não imediatamente a não ser se forçadas

Private ν

- Cada tarefa (thread) tem uma cópia do dado.
- Nenhuma outra tarefa (thread) pode acessar o dado
- Alterações são visíveis somente à tarefa (thread) proprietária do dado

==========

```
default(shared | none)
```

Todas as avariáveis em OpenMP são compartilhadas por default. Mas,se você deseja um conjunto de variáveis private, você necessitará especificar essas variáveis em uma diretiva pragma parallel em uma cláusula private.

Se você usa `#pragma omp parallel default(none )` você necessita especificar as variáveis privadas e as variáveis compartilhadas.

Por exemplo:

```
#pragma omp parallel default(none) private(i,j) shared(a,b)
```

==========

`firstprivate`:

Especifica que cada thread deve ter sua própria instância de uma variável e que a variável deve ser inicializada com o valor da variável, porque existe antes da construção paralela.

`lastprivate:`

Especifica que a versão do contexto de inclusão da variável é definida igual à versão privada de qualquer thread que executa a iteração final (construção for-loop) ou a última seção (seções #pragma).

```
#pragma omp parallel
{
   #pragma omp for lastprivate(i)
      for (i=0; i < n-1; i++)
         a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

Assim, neste exemplo, eu entendo que `lastprivate` permite para `i` ser retornada fora do loop, como o último valor que ela tinha.

==========

Aqui vamos considerar firstprivate e lastprivate. Lembre-se de uma das entradas anteriores sobre variáveis privadas. Quando uma variável é declarada como privada, cada thread recebe um endereço de memória exclusivo de onde armazenar valores para essa variável, enquanto na região paralela. Quando a região paralela termina, a memória é liberada e essas variáveis não existem mais. Considere o seguinte código como um exemplo:

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

int main(void) {
int i;
int x;
x=44;
#pragma omp parallel for private(x)
  for(i=0;i<=10;i++) {
  x=i;
  printf("Thread number: %d     x: %d\n", omp_get_thread_num(),x);
}
printf("x is %d\n", x);

}
```

Que acarreta ….

```
1
2    Thread number: 0     x: 0
3    Thread number: 0     x: 1
4    Thread number: 0     x: 2
5    Thread number: 3     x: 9
6    Thread number: 3     x: 10
```

```
Thread number: 2     x: 6
Thread number: 2     x: 7
Thread number: 2     x: 8
Thread number: 1     x: 3
Thread number: 1     x: 4
Thread number: 1     x: 5
x is 44
```

Você notará que x é exatamente o valor que era antes da região paralela.

Suponha que queremos manter o último valor de x após a região paralela. Isto pode ser conseguido com lastprivate. Substitua private (x) por lastprivate (x) e este é o resultado:

```
Thread number: 3     x: 9
Thread number: 3     x: 10
Thread number: 1     x: 3
Thread number: 1     x: 4
Thread number: 1     x: 5
Thread number: 0     x: 0
Thread number: 0     x: 1
Thread number: 0     x: 2
Thread number: 2     x: 6
Thread number: 2     x: 7
Thread number: 2     x: 8
x is 10
```

Observe que é 10 e não 8. Isso quer dizer, **é a última iteração que é mantida**, não a última operação. Agora, se substituirmos lastprivate (x) por firstprivate (x). O que você acha que vai fazer? Este:

```
Thread number: 3     x: 9
Thread number: 3     x: 10
Thread number: 1     x: 3
Thread number: 1     x: 4
Thread number: 1     x: 5
Thread number: 0     x: 0
Thread number: 0     x: 1
Thread number: 0     x: 2
Thread number: 2     x: 6
Thread number: 2     x: 7
Thread number: 2     x: 8
x is 44
```

Você poderia esperar obter o valor 0, ou seja, o valor de x na primeira iteração.

Firstprivate - Especifica que cada thread deve ter sua própria instância de uma variável e que a variável deve ser inicializada com o valor da variável, que existe antes da construção paralela.

Ou seja, cada thread recebe sua própria instância de x e essa instância é igual a 44.

==========

Redução

Mais uma cláusula: `reduction(op : list);`
usada para operações tipo "all-to-one":

- exemplo: op = '+'
- cada thread terá uma cópia da(s) variável(is) definidas em 'list' com a devida inicialização;
- ela efetuará a soma local com sua cópia;
- ao sair da seção paralela, as somas locais serão automaticamente adicionadas na variável

Exemplo de redução

Redução

```
#include <omp.h>
#define NUM_THREADS 4
void main( )
{
  int i, tmp, res = 0;
  #pragma omp  parallel  for reduction(+:res)  private(tmp)
     for (i=0 ; i< 10000 ; i++)
     {
       tmp = Calculo( );
       res += tmp ;
     }
     printf("O resultado vale %d´´, res) ; } Obs: os índices de
     laços sempre são privados.
}
```

Obs: Os índices de loops sempre são privados.

==========

Construtores de Work-Sharing (Ver em (1), Programação em Memória Partilhada com o OpenMP)

Os construtores de **work-sharing** permitem definir o modo de dividir trabalho entre os threads a executar numa região paralela. Os

construtores de **work-sharing** não criam novos threads, apenas definem o modo de execução de blocos específicos de código dentro duma região paralela (por este motivo, só faz sentido utilizá-los dentro de regiões paralelas). Todos os threads numa região paralela devem encontrar os mesmos construtores de work-sharing e pela mesma ordem.

Ao encontrarem um construtor de **work-sharing**, todos os threads podem desde logo começar a executar a parte de código que lhes diz respeito, isto é, à entrada dos construtores de **work-sharing** não existe qualquer barreira implícita de sincronização entre os threads. Ao completarem a região delimitada pelo construtor de **work-sharing**, por omissão, todos os threads sincronizam numa barreira implícita.

Existem 3 tipos de construtores de **work-sharing**:

- #pragma omp for
- #pragma omp sections
- #pragma omp single

Um exemplo prático #pragma omp for

O OpenMP usa técnicas de paralelização implícita, e é possível usar pragmas, funções explícitas e variáveis de ambiente para instruir o compilador. Vamos Considere o código da Listagem 6.

Listagem 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. Cada encadeamento (thread) pode ser executado em um núcleo diferente, o que reduz significativamente o tempo total de cálculo.

Seções críticas com OpenMP

```
#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 em um dado momento. Além disso, `optional section name` é um identificador global, e dois encadeamentos não podem executar seções críticas com o mesmo identificador global ao mesmo tempo.

Considere o código da Listagem 10.

Listagem 10. 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");
 }
```

Controle de baixa granularidade sobre a execução de tarefas

Todos os encadeamentos 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. Considere o código da Listagem 12.

Listagem 12. 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`, 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 individual. No entanto, as instruções individuais dentro do bloco de seção são sempre executadas em sequência. A Listagem 13 mostra a saída do código da Listagem 12.

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 oito encadeamento sendo criados inicialmente. Desses oito 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.

Terminogia:

chunks == pedaços de ...

spaw == gerar .... por exemplo,gerar uma região paralela.

mutex == para o objeto de exclusão mútua. Na programação de computadores, um mutex é um objeto de programa que permite que vários segmentos do programa possam compartilhar o mesmo recurso, tais como acesso a arquivos, mas não simultaneamente. Quando um programa é iniciado, um mutex é criado com um nome único. Após esta etapa, qualquer segmento que precisa o recurso deve bloquear a exclusão mútua de outros tópicos, enquanto ele está usando o recurso. O mutex é definido para desbloquear quando os dados não é mais necessário ou a rotina é concluída.