实验：用 OpenMP 实现线程级并行

实验目标：

掌握在多核处理器上实现共享内存的线程级并行编程

资源：

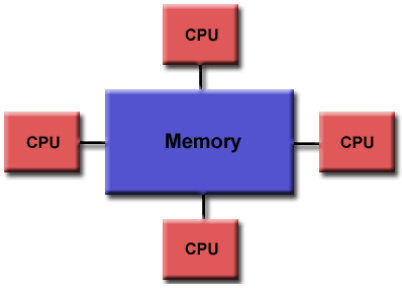
- 在线教程:
  - OpenMP Tutorial | High Performance Computing (llnl.gov)
  - iwomp2005_tutorial_openmp_rvdp.pdf (uoregon.edu)
- 官方教程网址：Tutorials & Articles - OpenMP
- 请自行百度中文教程,例如：OpenMP 中文教程 - 简书 (jianshu.com)

实验准备：

下载文件包并解压： `$ tar xvf lab06.tar`

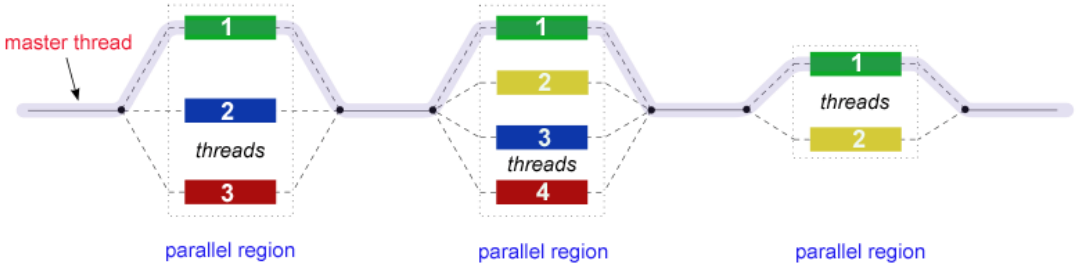
OpenMP 简介：

OpenMP 是一个应用程序接口（API），它提供一种已被广泛接受、用于共享内存并行多处理器程序设计的一套指导性编译处理方案。OpenMP 支持的编程语言包括 C、C++和 Fortran；



共享内存的多处理器模型

OpenMP 使用并行执行的 fork-join 模型



- 程序开始于一个主线程，主线程顺序执行，直到遇到一个并行区域结构（parallel region construct）；

- FORK： 主线程创建一组并行线程；
- 并行区域结构中封装的语句在各个线程中并行执行；
- JOIN： 当该组中每一个并行线程都完成并行区域结构中的语句时，它们进行同步并终止，只留下主线程；
- 并行区域的数量和组成并行区域的线程数目是任意的。

OpenMP 提供对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的指导性指令 pragma 来表明自己的意图，编译器根据程序员给出的指导，自动将程序进行并行化，并在必要之处加入同步互斥以及通信。下面举例说明 OpenMP 中的编译指导语句。

考虑以下 Hello World 的实现 (hello.c):

```
int main() {
  #pragma omp parallel
  {
    int thread_ID = omp_get_thread_num();
    printf(" hello world %d\n", thread_ID);
  }
}
```

- #pragma 告诉编译器，本行的后面部分是一个编译指导指令. 本例中的指导指令是：omp parallel.
- omp 告诉编译器，这是一个 OpenMP 程序
- parallel 告诉编译器，接下来的代码块 ({ }中的代码) 需要多线程并行执行。

尝试编译和运行程序：

```
$ make hello
$ ./hello
```

你会看到会有多个线程输出“hello world”. 缺省情况下，线程的数目由 OpenMP 设置环境变量来决定，一般会将线程数量设置为处理器个数。你亦可以将编译指导指令改写为：

```
#pragma omp parallel num_threads(NUM_THREAD)
```

其中，将 NUM_THREAD 置为你希望的线程个数。例如：NUM_THREAD 值为 8，就代表将线程数目设置为 8 个。多次运行 hello 程序，由于多个线程是并行执行，你会发现多个线程并没有固定的执行顺序。变量 thread_ID 是每一个线程都拥有的独立的局部变量。所以，在 omp parallel 代码块以外定义的变量，是被所有线程共享的全局变量，而在 omp parallel 代码块内定义的变量，是每个线程都拥有一份的私有变量。

Exercise 1: 向量加法 Vector Addition

编译和运行程序：

```
$ make v_add
$ ./v_add
```

阅读 v_add.c 程序，其中的 v_add() 函数 的作用是调用不同的方法，完成将向量（vector）x 和向量（vector）y 的元素两两相加，并存入结果向量（vector）z 中。 不输入参数时，会使用 method_0 函数中的方法进行计算:

```
void method_0(double* x, double* y, double* z) {
  #pragma omp parallel
  {
    for(int i=0; i<ARRAY_SIZE; i++)
      z[i] = x[i] + y[i];
  }
}
```

运行结果显示：增加线程的个数，反而运行时间更慢了。因为在 omp parallel 块中的语句（parallel region:并行区域），每个线程都会执行一遍，所以，如果有八个线程，向量加法就重复了 8 次。

继续阅读 v_add.c 程序，观察 method_1() 和 method_2() 两个函数，你可以通过运行时输入不同的参数，观察 method_1() 和 method_2()的运行结果。

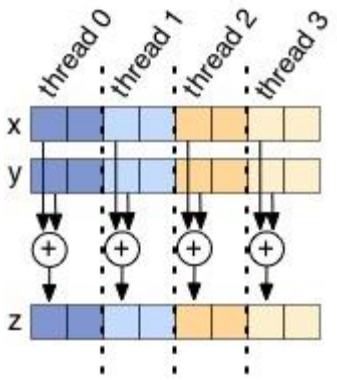
```
$ ./v_add 1       # 显示调用 method_1()的执行结果

$ ./v_add 2       # 显示调用 method_2()的执行结果
```

回答问题：

1) method_1() 和 method_2() 都实现了并行向量加法，它们和 method_0() 区别在于：不同的线程只计算一部分工作量。method_1()通过手工编程把任务分解给不同的线程，method_2()通过编译指导：#pragma omp for 实现在多个线程间 work sharing， 运行程序，比较它们的运行时间。为什么它们的执行时间不一样？method_1() 的运行时间受了什么因素的影响？（提示： 假共享问题， false sharing）

接下来，改写 method_3() 函数，手动分割任务量，使得不同的线程，分工处理不同部分的向量加法，分工如下图所示，避免多个线程并行处理的数据是相邻的，以防止因为“false sharing” 导致并行处理性能下降。



推荐两个函数供你使用:

```
int omp_get_num_threads();
int omp_get_thread_num();
```

改写完 method_3 后，编译并测试它的性能：

```
$ make v_add
$ ./v_add 3
```

回答问题：
2） 你的 method_3 达到 method_2 同等的性能了吗？ 贴出你的实现代码。

Exercise 2: Dot Product

接下来计算向量点积（点积_百度百科 (baidu.com), dot product ） 这个问题的难点在于，如何将部分点积的结果，累加到一个共享变量 global_sum. (reduction).

一种解决方案是使用临界区（critical section）. 代码如下所示 (dotp.c):

```
double dotp_1(double* x, double* y) {
  double global_sum = 0.0;
  #pragma omp parallel
  {
    #pragma omp for
    for(int i=0; i<ARRAY_SIZE; i++)
      #pragma omp critical
        global_sum += x[i] * y[i];
  }
  return global_sum;
}
```

回答问题：

1) 如果删除函数 dotp_1(double* x, double* y) 中的 #pragma omp critical，你会发现点积计算的结果是不正确的。所以，访问共享变量 global_sum 时，设置临界区是非常必要的。但临界区设置的不恰当，还是会大大影响程序的性能。**编译和运行程序 (make dotp and ./dotp). 观察一下，是不是线程的数目越多，反而性能越差? 分析原因？**

2) **修改程序，让各个线程在计算部分点积时，不要将结果直接写入 global_sum, 而是写入各自的私有变量 local_sum, 最后再通过临界区，汇总到 global_sum。 在函数 dotp_2(double* x, double* y)中给出你改写的代码，并对比修改前后的性能。**

3) 进一步的，你可以使用 OpenMP 的归约操作，程序中的 dotp_3(double* x, double* y) 函数给出了示意，形式如下：

```
double dotp_3(double* x, double* y) {
  double global_sum = 0.0;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:global_sum)
    for(int i=0; i<ARRAY_SIZE; i++){
       global_sum += x[i] * y[i];
    }
  return global_sum;
}
```

解释一下 reduction 语句的作用，并测试使用归约语句改写后的并行点积计算的性能，对比它与 dotp_1 以及 dotp_2 的性能差别。