高性能并行计算

填空

1. `gcc` 编译 OpenMP 程序
    - ```
      gcc -fopenmp filename.c # 编译

      export OPM_NUM_THREADS=4 # 设置环境变量 线程数
      ./a.out # 运行可执行文件
      ```
2. MPI 的概念 `PPT238`
    - Massage Passing Interface
        - 是消息传递函数库的标准规范, 由 MPI 论坛开发, 支持 Fortran 和 C
            - 一种新的库描述, 不是一种语言, 共有上百个函数调用接口, 在 Fortran 和 C 语言中可以直接对这些函数进行调用.
            - MPI 是一种标准或规范的代表, 而不是特指某一个对它的具体实现
            - MPI 是一种消息传递编程模型, 并称为这种编程模型的代表和事实上的标准.
3. 日本的超算
    - 日本曾经排名第一的超级计算机 「地球模拟器」 用于气象预报
    - 中国「神威」「银河 III」「曙光」在中央气象台
    - 中国国家超级计算中心
        - 天津 天河一号
        - 深圳 曙光 6000
        - 长沙 天河一号
        - 济南 神威蓝光
        - 广州 天河二号
        - 无锡 神威 太湖之光
4. 并行效率计算
      ```

并行效率指标

- “以万核级为基准的并行效率”是指以程序在1万核下的执行时间为基准，考 查并行规模增长到 N 万核时，程序性能的增长比例。其计算公式如下:

$$以万核为基准的并行效率 = \frac{T_1}{T_N \times N} \times 100\%$$

上式中，T1 为程序在并行规模 1 万核时的执行时间，TN 为程序在并行规模 N 万核时的执行时间。

并行效率计算示例

- 假定某程序在 1 万核上执行时间为 1000 秒:
- - 如其在 10 万核上执行时间 100 秒，则并行效率为 100%
- - 如其在 10 万核上执行时间 200 秒，则并行效率为 50%
- - 如其在 10 万核上执行时间 500 秒，则并行效率为 20%
- - 如其在 60 万核上执行时间 16.667 秒，则并行效率为 100%
- - 要达到 60 万核并行效率≥30%，运行时间需≤55.555 秒

- 加速比
  - 加速比常用来衡量程序并行执行后的性能提升效果，其原理计算公式如下:

$$Speedup = \frac{T_{串行}}{T_{并行}}$$

上式中，T 串行为程序的串行执行时间，T 并行为程序的并行执行时间。加速比结果表述为“**倍”，或简写为“**x”。

$$Speedup = \frac{T_{单处理器}}{T_{多处理器}} \qquad Speedup = \frac{T_{单节点}}{T_{多节点}} \qquad Speedup = \frac{T_{少量进程}}{T_{大量进程}}$$

5. OpenMP
    - OpenMP 应用编程接口 API 是在共享存储体系结构上的一个编程模型
    - 包含三个基本 API
        - 编译制导(Compiler Directive)
        - 运行库例程 (Runtime Library)
        - 环境变量(Environment Variables)
    - 支持增量并行化(Incremental Parallelization)
    - 是 C/C++ 和 Fortran 等的应用编程接口
    - OpenMP 不包含的性质
        - 不是建立在分布式存储系统上的
        - 不是所有的环境下都是一样的
        - 不能保证让多数共享存储器均能有效利用
6. 云计算服务
    - 云计算是通过网络按需提供可动态伸缩的廉价计算服务。
    - 云计算是一种商业计算模型。它将计算任务分布在大量计算机构成的资源池上，使各种应用系统能够根据需要获取计算力、存储空间和信息服务。
    - 分为三类
        - 软件即服务 SaaS
        - 平台即服务 PaaS
        - 硬件即服务 IaaS
7. 最早集群
    - 贝奥武夫机群（Beowulf cluster），又称贝奥伍尔夫集群，是一种高性能的并行计算机集群结构，特点是使用廉价的个人电脑硬件组装以达到最优的性能/价格比。得名于古英语著名史诗《貝奥武夫》。于1994年最早在NASA为Donald Becker等人开发，是目前科学计算中流行的一类并行计算机。
    - 集群
        - 量产的商业处理器和内存
            - 处理器的性能至关重要
        - 同样有远程内存访存延迟
        - 需要消息传递
            - 最小化通讯开销
        - 全部是量产的商业部件
        - 各机器可以有独立的 OS
        - 各机器有独立 IO

8. 进程/线程
    - 进程
        - 一段程序的执行过程
        - 是一个具有一定独立功能的程序关于某次数据集合的一次运行活动，它是操作系统分配资源的基本单元
        - 进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程中调用的指令和本地变量。
        - 进程是一个「执行中的程序」。程序是一个没有生命的实体，只有处理器赋予程序生命时，它才能成为一个活动的实体，我们称其为进程。
    - 线程
        - 通常在一个进程中可以包含若干个线程，当然一个进程中至少有一个线程，不然没有存在的意义。
        - 线程可以利用进程所拥有的资源，在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位，由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统多个程序间并发执行的程度。
    - 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。
9. 编译模式区别
    - 编译指导指令
        - `critical`：每次允许一个线程执行
        - `single`：只由一个线程执行一次
        - `atomic`：功能类似critical，不过所属指令为简单的表达式，这样减少开销
        - `master`：仅由主线程执行
10. OpenMP
    1. OpenMP 的基本指令形式为 `#pragma omp 指令 [子句[子句]…]`
        - 指令

    - `parallel` 用在一个代码段之前，表示这段代码将被多个线程并行执行
    - `for` 用于 for 循环之前，将循环分配到多个线程中并行执行，必须保证每次循环之间无相关性。
    - `parallel for` 是 `parallel` 和 `for` 语句的结合，也是用在一个for循环之前，表示for循环的代码将被多个线程并行执行。
    - `sections` 用在可能会被并行执行的代码段之前
    - `parallel sections` 是 `parallel` 和 `sections` 两个语句的结合
    - `critical` 用在一段代码临界区之前 每次允许一个线程执行
    - `single` 用在一段只被单个线程执行的代码段之前，表示后面的代码段将被单线程执行。
    - @没学不考 `flush` 用来保证线程的内存临时视图和实际内存保持一致，即各个线程看到的共享变量是一致的
    - `barrier` 用于并行区内代码的线程同步，所有线程执行到 `barrier` 时要停止，直到所有线程都执行到 `barrier` 时才继续往下执行。
    - `atomic` 用于指定一块内存区域被制动更新 功能类似 `critical`，不过所属指令为简单的表达式，这样减少开销
    - `master` 用于指定一段代码块仅由主线程执行
    - @没学不考 `ordered` 用于指定并行区域的循环按顺序执行
    - `threadprivate` 用于指定一个变量是线程私有的
    - `copyprivate` 配合 `single` 指令，将指定线程的专有变量广播到并行域内其他线程的同名变量中；
    - `copyin n` 用来指定一个 `threadprivate` 类型的变量需要用主线程同名变量进行初始化
    - `default` 用来指定并行域内的变量的使用方式，缺省是 `shared`
- 库函数
    - `omp_get_num_procs` 返回运行本线程的多处理机的处理器个数。
    - `omp_get_num_threads` 返回当前并行区域中的活动线程个数。
    - `omp_get_thread_num` 返回线程号
    - `omp_set_num_threads` 设置并行执行代码时的线程个数
    - `omp_init_lock` 初始化一个简单锁
    - `omp_set_lock` 上锁操作
    - `omp_unset_lock` 解锁操作，要和 `omp_set_lock` 函数配对使用
    - `omp_destroy_lock` 是 `omp_init_lock` 函数的配对操作函数，关闭一个锁

    - > `OMP_NUM_THREADS`：环境变量级别的线程总数限定，优先级最低，`omp_set_num_threads` 是程序级别的，优先级较高；子句 `num_threads(i)` 是block级别，优先级最高

  - 子句
    - `private` 指定每个线程都有它自己的变量私有副本。
    - `firstprivate` 指定每个线程都有它自己的变量私有副本，并且变量要被继承主线程中的初值。
    - `lastprivate` 主要是用来指定将线程中的私有变量的值在并行处理结束后复制回主线程中的对应变量。
    - `reduce` 用来指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的运算。
    - `nowait` 忽略指定中暗含的等待
    - `num_threads` 指定线程的个数
    - `schedule` 指定如何调度 for 循环迭代
    - `shared` 指定一个或多个变量为多个线程间的共享变量
    - `ordered` 用来指定 for 循环的执行要按顺序执行
    - `copyprivate` 用于 `single` 指令中的指定变量为多个线程的共享变量
    - `copyin` 用来指定一个 `threadprivate` 的变量的值要用主线程的值进行初始化。
    - `default` 用来指定并行处理区域内的变量的使用方式，缺省是 `shared`

11. 计算单位

计算单位	大小
Kilo	10^3
Mega	10^6
Giga	10^9
Tera	10^{12}
Peta	10^{15}
Exa	10^{18}
Zetta	10^{21}

选择

1. MPI greeting

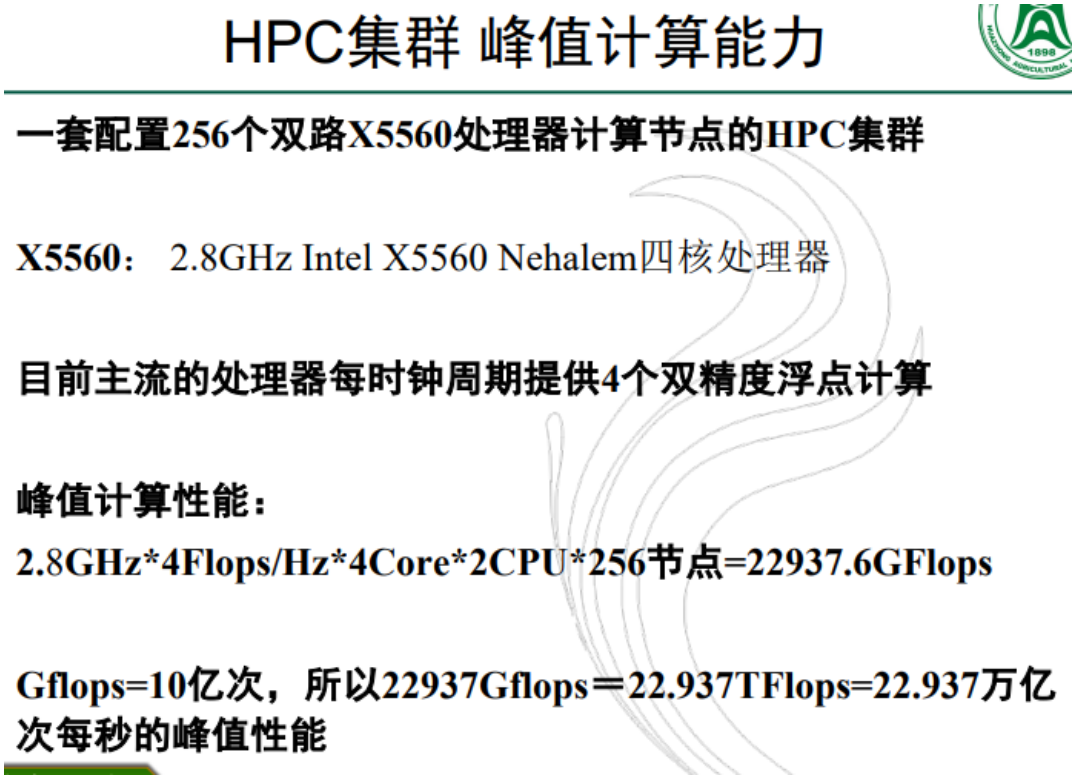
   ```c
   #include <stdio.h>
   #include "mpi.h"

   main (int argc, char* argv[]) {
       int numprocs, myid, source;
       MPI_Status status;
       char message[100];

       // 启动 MPI 环境, 标志并代码开始
       MPI_Init(&argc, &argv);
       // 获得当前通信域中的进程数目，数目保存在变量 myid 中。
       MPI_Comm_rank(MPI_COMM_WORLD, &myid);
       // 获取进程数目
       MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

       if (myid != 0) {
           strcpy(message, "Hello World!");

           // 发送消息
           MPI_Send(&message, strlen(message) + 1, MPI_CHAR, 0, 99, MPI_COMM_WORLD);

           for (source = 1; source < numprocs; source++) {
               // 接收消息
               MPI_Recv(&message, 100, MPI_CHAR, source, 99, MPI_COMM_WORLD, &status);
               printf("%d\n", message);
           }
       }
       // 从 MPI 环境中退出
       MPI_Finalize();
   }
   ```

   - MPI 初始化 `int MPI_Init(int *argc, char **argv)`
     - MPI 程序的第一个调用, 完成 MPI 程序的所有初始化工作, 所有 MPI 程序的第一条可执行语句.
     - 启动 MPI 环境, 标志并行代码开始.
     - 并行代码前, 除 `MPI_Initialize()` 的第一个 MPI 函数
     - 要求 `main` 函数必须带参数运行, 否则出错
   - `MPI_COMM_WORLD` 通信空间（通信子）

- 发送消息 `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - `buf` 要发送内容的首地址
  - `count` 发送的元素数目
  - `datatype` 要发送的数据类型
    - C 语言中的数据类型，会与 MPI 的数据类型绑定
MPI(C 语言绑定)	C 语言中的数据类型
`MPI_BYTE`	
`MPI_DOUBLE`	`double`
`MPI_FLOAT`	`float`
`MPI_INT`	`int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_CHAR`	`signed char`
  - 消息信封 `<源/目标地址(source/dest)，消息标签(tag)，通信空间(MPI_Comm)>` 表示接收/发送消息的地址
    - `source` / `dest` 指源/目标进程的进程号, 即从函数 `MPI_Comm_rank` 获得的 `rank` 值
- 接收消息 `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - `status` 返回状态
    - 在 C 语言中, 状态变量至少包括
      - `MPI_SOURCE`
      - `MPI-TAG`
      - `MPI_ERROR`
    - `status` 通过 `MPI_Status status;` 语句生成
- MPI 结束 `int MPI_Finalize(void)`
  - MPI 程序的最后一个调用, 结束 MPI 程序的运行, 是 MPI 程序的最后一条可执行语句
  - 标志并行代码的结束, 结束除主进程以外的其他进程
  - 之后的代码仍然可以在主进程(rank = 0)运行

2. MPI 编程初始化

    - 见上一条
3. 指令集 Intel
    - 复杂指令集
        - X86
            - Intel
            - AMD
    - 精简指令集
        - ARM
        - RISC-V
        - Alpha
4. 并行计算的用处
    - 天气预报
    - 核武器模拟
    - 国防安全
    - 天体物理模拟
    - 地球物理勘探
    - 物质结构分析
    - 生物信息学
5. 峰值计算的单位换算



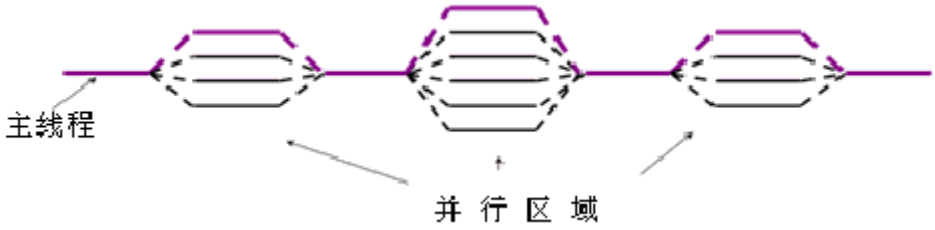
简答

1. OpenMP 的结构, 并行 1-100 的求和 for 循环

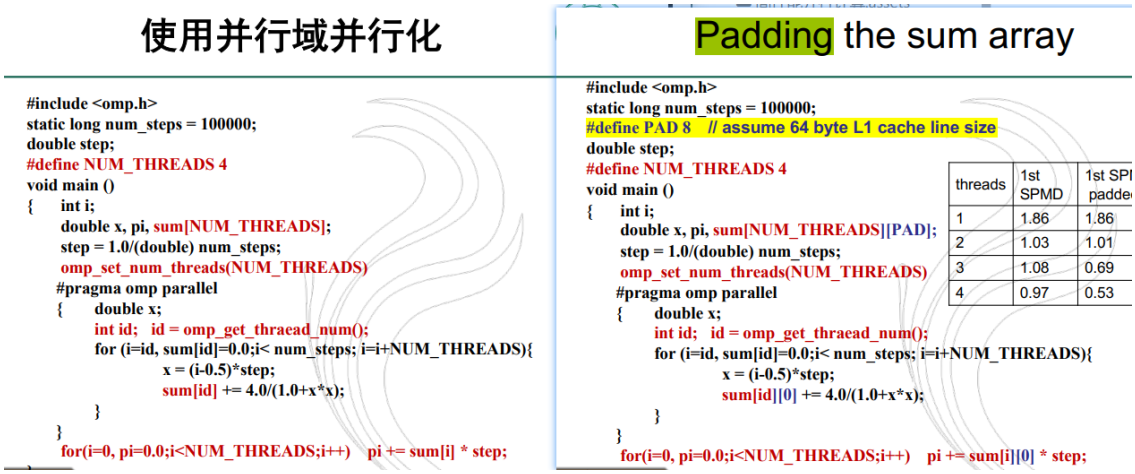
```c
#include <stdio>
#include <omp.h>

void main() {
    int sum;
    // 需要使用 reducition 规约
    // 将最终的 sum 加起来 防止多线程冲突
    #pragma omp parallel for reducion(+:sum)
    for(int i = 0; i <= 100; i++) {
        sum += i;
    }

    printf("sum=%d", sum);
}
```

> 在每个线程中，都创建了一个私有的`sum`，把该线程操作的数累加，之后每个线程再汇集其私有的`sum`，得到最终的`sum`，这样每个线程的私有`sum`互不干涉，就防止数据冲突。

2. 并行域的数值分析 `private` `lastprivate` `firstprivate`
    - `private`
        - 声明其列表中的变量对每个线程都是私有的
        - 从申明的并行区域开始，为每个线程声明一个相同类型的新对象，对原始对象的所有引用都将替换为对新对象的引用
        - ```c
          // 并行域外声明了 i 和 a
              int i = 0;
              float a = 512.3;
          #pragma omp parallel private(i,a)
              {
                  // 并行域内的 i 和 a 与并行域外无关
                  // 是两个没有初始化的变量，值为对应类型默认值
                  // i 为 int 类型默认值 0
                  // j 为 float 类型默认值 0.000000
                  // 每个线程之间 i a 都是私有的 (独立的)
                  i = i + 1;
                  printf("thread %d i = %d a= %f\n",
          omp_get_thread_num(), i, a);
              }
              printf("out of parallel i = %d", i);
              }
          ```
    - `firstprivate`
          ```

- 声明其列表中的变量对每个线程都是私有的
- 但是初始化的值为进入并行域之前的值
- ```
    int i = 1;
    float a = 512.3;
#pragma omp parallel firstprivate(i) private(a)
    {
        // i 是 firstprivate
        // 对于每个线程, i 是私有独立的
        // 但根据并行域之前定义的 i = 1 对其赋值
        i = i + 1; // 允许完这个语句之后 i = 2
        // a 是 private 所以不会根据并行域外的 a 对其赋值, 这里为类型默认值
        printf("thread %d i = %d a= %f\n", omp_get_thread_num(), i, a);
    }
    printf("out of parallel i = %d", i);
```

- `lastprivate`
  - 指定的变量不仅是 `private` 作用范围，同时会将最后一次迭代或最后一个 `section` 执行的值复制回原来的变量
  - 即在并行域外定义的变量, 在并行域内对其值进行修改后, 会将修改后的值赋值给并行域外的原变量
  - ```
    int i = 0;
    float a = 512.3;
#pragma omp parallel
    {
    #pragma omp sections lastprivate(i) private(a)
        {
            // 2 个 section
            // 运行时一个 thread num 为 0 (主线程)
            // 另一个为 1
        #pragma omp section
            {
                i = i + omp_get_thread_num();
                printf("thread %d i = %d a= %f\n", omp_get_thread_num(), i, a);
            }
        #pragma omp section
            {
                i = i + omp_get_thread_num();
                printf("thread %d i = %d a= %f\n", omp_get_thread_num(), i, a);
    ```

```
            }
        }
    }
    // 在并行域外打印
    printf("out of parallel i = %d", i); // i 输出 1

```

- 其他的数据作用域
  - `shared`
    - 表示线程之间共享被 `shared` 指定的变量
    - 需要注意并发冲突 同时需要用 `critical` 等指令
    - 会将并行域内变量的修改赋值到原变量

3. OpenMP Fork-Join 并行执行模型
   - OpenMP在并行执行程序时，采用的是fork/join式并行模式，共享存储式并行程序就是使用fork/join式并行的。在开始时，只有一个叫做主线程的运行线程存在 。在运行过程中，当遇到需要进行并行计算的时候，派生出（Fork）线程来执行并行任务 。在并行代码结束执行，派生线程退出或挂起，控制流程回到单独的主线程中（Join）。
   - 

4. 点对点

代码和改错

1. 并行 pi (并行域), 代码改错

```
#include <omp.h>

static long num_steps = 100000;
double step;
#define NUM_THREADS 4

void main() {
    double pi, sum = 0.0;
    step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        double x;
```

```
        int i;
#pragma omp for reduction(+:sum)
        for (i = 0; i < num_steps; i++) {
            x = (i - 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
        pi = step * sum;
    }
}
```

2. padding 解决伪共享



3. MPI 的编译和运行

- ```
  mpif77 hello.f #两个都可以, 默认生成 a.out 的可执行文件
  mpicc hello.c
  ```

  ```
  mpif77 -o hello hello.f #两个都可以, 生成 hello 可执行文件
  mpicc -o hello hello.c
  ```

  ```
  mpirun -np 4 a.out #运行, 指定线程数为4
  ```

4. 改错

```
#include <stdio.h>
#include "mpi.h" // 必须包含 mpi.h

main (int argc, char *argv[]) { // main 函数必须有参数
    MPI_Init(&argc, &argv); // mpi 第一个调用
    printf("Hello, world!\n");
    MPI_Finalize();
}
```
```

讨论

1. 广播的作用效果及实现 3x3 矩阵分享 (用广播, 1号发给其他线程)
2. 高性能的理解, 前景在生信和大数据的应用关系, 以及是否量子计算取代传统计算