OpenMP

OpenMP 是一种跨平台共享内存的应用程序接口，通过编译器制导语句实现半自动并行方法，提高串行代码性能，和 C++ 多线程编程模型比，OpenMP 语法更加简约，**OpenMP 理想地适用于研究串行代码的并行潜力，或者简单的在短时间内快速加速算法并行。**

优点：

对基于松耦合的数据集的多线程程序设计，OpenMP是一个很好的选择，例如两个向量加减，两个矩阵运算，此类原有串行代码可以在几乎不修改代码的情况下，改为并行代码，方便易用。并且对于上述代码，OpenMP 执行效率相比使用基础 c++ thread 开发的代码执行效率高。同时，使用OpenMP也提供了更强的灵活性，可以较容易的适应不同的并行系统配置，OpenMP的一个优势就是跨平台，不依赖本地系统API (早期的 c++ 多线程编程依赖系统API，比如 windows 的API函数 CreateThread，linux 创建线程函数 pthread_create，这样编写的代码不可移植，不能跨平台，但是 C11的线程库解决了跨平台的问题)。此外线程粒度和负载平衡等是传统多线程程序设计中的难题，但在OpenMP中，OpenMP库从程序员手中接管了部分这两方面的工作，使得程序员可以更多关心并行算法逻辑，而非具体实现细节。

缺点：

OpenMP 作为一种高层抽象，**OpenMP 并不适合需要复杂的线程间同步和互斥的场合**。例如一个读写队列缓存，一些线程读数据放入队列，一些线程取出数据处理，这样的场景需要频繁的上锁，且当队列为空时，数据处理线程应该等待， 当队列满时，数据放入线程应该等待，这样细颗粒度的线程控制逻辑，在 OpenMP 中几乎无法实现，或者实现的成本大于传统线程编程。 此外，原有业务逻辑线程封装的复杂，例如被类包裹，OpenMP 很难改造。管理线程之间的复杂交互或依赖于对线程函数的紧密操作的程序将需要使用传统线程API。

此外和传统线程api编程相比，OpenMP 对线程优先级缺乏控制力，无法对每个线程调整优先级，在其它方面也有缺陷，例如缺乏线程信号量控制等。OpenMP 的内存模型是统一/共享的内存模型 (unified/shared memory)，比如笔记本的电脑CPU有4个核心但只有一个共同的内存，各个核心通过内存来分享交换数据。那么显然，OpenMP的另一个缺点是不能在非共享内存系统(如计算机集群)上使用。在这样的系统上，MPI使用较多。

> openmp 更多用于快速 优化/并行 算法， 而不是并行业务逻辑

基本使用

使用 gcc 编译器，编译时需加上选项 -fopenmp， 例如 `g++ -O2 -std=c++14 -fopenmp helloworld.cc`

> gcc 版本对 OpenMP的支持 详见， 本文例子均使用 gcc-10.2.0 编译

Hello World

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

int main() {
    #pragma omp parallel
```

```
    printf("Hello from thread %d, nthreads %d\n", omp_get_thread_num(),
omp_get_num_threads());  // 线程编号从 0 开始
    return 0;
}

---------
Hello from thread 1, nthreads 12
Hello from thread 3, nthreads 12
Hello from thread 5, nthreads 12
Hello from thread 2, nthreads 12
Hello from thread 6, nthreads 12
Hello from thread 4, nthreads 12
Hello from thread 7, nthreads 12
Hello from thread 8, nthreads 12
Hello from thread 10, nthreads 12
Hello from thread 9, nthreads 12
Hello from thread 0, nthreads 12
Hello from thread 11, nthreads 12
```

> OpenMP 在运行时确定使用的默认线程个数等于操作系统的逻辑cpu核数，这可以通过 OMP_NUM_THREADS 环境变量修改，也可以在代码中使用 set_num_threads() 或者特殊编译器制导语句指定， 当前制导语句作用的线程，称之为一个组 team

parallel for 并行 for 循环

循环的并行化是利用OpenMP进行程序并行计算的最关键部分。因为很多科学计算程序，尤其是涉及矩阵求解运算的程序，将大量的计算时间消耗在对循环计算的处理上，因此循环的并行化在OpenMP程序中是一个相对独立并且十分重要的组成部分

```
#pragma omp parallel for
for(...) {
  ...
}

#pragma omp parallel for if(bool) // 条件判断语句，可根据某变量决定是否启用并行模式
for(...) {
  ...
}

// #pragma omp parallel for 制导语句的完整形式如下：
#pragma omp parallel
{
  #pragma omp for   // 此语句尤为关键，如果忽略此语句，每个线程将完整的执行一遍 for 循环，这句话将分隔 for 循环为 n 段索引大小为 n/num_threads 的块，然后启动线程去执行这些块
  for(...) {
    ...
  }
```

```
}
```

> OpenMP 是一个工作在共享内存上的API，也就是说我们必须保证代码数据之间是独立的，不存在依赖关系。 例如如果 for 循环内的代码是 a[i+2] = a[i+1] + a[i]; 那么 openmp 并行后的代码，结果是不正确的，数据之间存在依赖

举例1: 简单使用

```
int main() {
  omp_set_num_threads(4); //设置线程数，一般设置的线程数不超过CPU核心数，这里开4个线程执行并行代码段
  #pragma omp parallel for
  for (int i = 0; i < 8; i++)  // 4个线程，将 i=[0, 8) 切成连续的四块，分别执行
      printf("i = %d, I am Thread %d\n", i, omp_get_thread_num());
}
-----------------
i = 0 from thread 0
i = 1 from thread 0
i = 2 from thread 1
i = 3 from thread 1
i = 6 from thread 3
i = 7 from thread 3
i = 4 from thread 2
i = 5 from thread 2
```

举例2：向量加法 (已知 x 向量的值从 1 到 n， y 向量的值从 n-1 到 0，求两向量之和 z 向量)

```
#include <vector>
#include <iostream>
#include <chrono>
#include <omp.h>

using namespace std;

// 辅助计时器
#define TIMERSTART(label)                                                      \
        std::chrono::time_point<std::chrono::system_clock> a##label, b##label;
\
        a##label = std::chrono::system_clock::now();

#define TIMERSTOP(label)                                                       \
        b##label = std::chrono::system_clock::now();
\
        std::chrono::duration<double> delta##label = b##label-a##label;
\
```

```
        std::cout << "# elapsed time ("<< #label <<"): "
\
                  << delta##label.count()  << "s" << std::endl;

int main(){
    TIMERSTART(alloc)
    const uint64_t  num_entries = 1UL << 30;
    vector<uint64_t> x(num_entries);
    vector<uint64_t> y(num_entries);
    vector<uint64_t> z(num_entries);
    TIMERSTOP(alloc)

    TIMERSTART(init) // 初始化
    for (uint64_t i = 0; i < num_entries; ++i) {
        x[i] = i;
        y[i] = num_entries - i;
    }
    TIMERSTOP(init)

    TIMERSTART(seq)  // 串行代码计算两向量相加
    for (uint64_t i = 0; i < num_entries; ++i) {
        z[i] = x[i] + y[i];
    }
    TIMERSTOP(seq)

    TIMERSTART(openmp) // openmp 并行计算两向量相加
    #pragma omp parallel for
    for (uint64_t i = 0; i < num_entries; ++i) {
        z[i] = x[i] + y[i];
    }
    TIMERSTOP(openmp)

    TIMERSTART(check) // 检验计算结果是否正确
    for (uint64_t i = 0; i < num_entries; ++i) {
        if(z[i] - num_entries) {
            cout << "error at position: " << i << endl;
        }
    }
    TIMERSTOP(check)
}
---------------
# elapsed time (alloc): 15.2817s
# elapsed time (init): 12.5398s
# elapsed time (seq): 22.8436s
# elapsed time (openmp): 10.5254s  // 使用 openmp 并行代码后的时间
# elapsed time (check): 10.8684s
```

举例3：矩阵乘法，已知矩阵 A 为 m*n维的实数矩阵， x 是一个 n 维向量， 那么矩阵 A 将向量 x 从 n 维空间映射到 m 维向量空间，即求向量 b = A * x

```
#include <iostream>
#include <cstdint>
#include <vector>
#include <chrono>

// 辅助计时器
#define TIMERSTART(label)                                                      \
        std::chrono::time_point<std::chrono::system_clock> a##label, b##label;
\
        a##label = std::chrono::system_clock::now();

#define TIMERSTOP(label)                                                       \
        b##label = std::chrono::system_clock::now();
\
        std::chrono::duration<double> delta##label = b##label-a##label;
\
        std::cout << "# elapsed time ("<< #label <<"): "
\
                  << delta##label.count()  << "s" << std::endl;

// 初始化代码
template <typename value_t,
          typename index_t>
void init(std::vector<value_t>& A,
          std::vector<value_t>& x,
          index_t m,
          index_t n) {

    for (index_t row = 0; row < m; row++)
        for (index_t col = 0; col < n; col++)
            A[row*n+col] = row >= col ? 1 : 0; // 矩阵的对角线(包含对角线)一下填充为
1，以上填充为0

    for (index_t col = 0; col < m; col++)
        x[col] = col; // x 为简单的递增向量
}

template <typename value_t,
          typename index_t>
void mult(std::vector<value_t>& A,
          std::vector<value_t>& x,
          std::vector<value_t>& b,
          index_t m,
          index_t n,
          bool parallel) {
```

```
    #pragma omp parallel for if(parallel)  // openmp 制导语句，可切换串行，并行模式
    for (index_t row = 0; row < m; row++) { // 连续计算 A 的第 i 行和向量 x 的标量积
        value_t accum = value_t(0);
        for (index_t col = 0; col < n; col++)
            accum += A[row*n+col]*x[col];
        b[row] = accum;
    }
}

int main() {
    const uint64_t n = 1UL << 15;
    const uint64_t m = 1UL << 15;

    TIMERSTART(overall)
    TIMERSTART(alloc)
    std::vector<uint64_t> A(m*n);
    std::vector<uint64_t> x(n);
    std::vector<uint64_t> b(m);
    TIMERSTOP(alloc)

    // 初始化矩阵 A,  向量 x
    TIMERSTART(init)
    init(A, x, m, n);
    TIMERSTOP(init)

    // 串行计算 A * x = b  三次
    for (uint64_t k = 0; k < 3; k++) {
        TIMERSTART(mult_seq)
        mult(A, x, b, m, n, false);
        TIMERSTOP(mult_seq)
    }
    // 并行计算 A * x = b 三次
    for (uint64_t k = 0; k < 3; k++) {
        TIMERSTART(mult_par)
        mult(A, x, b, m, n, true);
        TIMERSTOP(mult_par)
    }
    TIMERSTOP(overall)

    // 检验结果是否正确
    for (uint64_t index = 0; index < m; index++)
        if (b[index] != index*(index+1)/2)
            std::cout << "error at position " << index
                      << " " << b[index] << std::endl;
}
------------------
# elapsed time (alloc): 4.47038s
# elapsed time (init): 5.78912s
# elapsed time (mult_seq): 0.950984s
```

```
# elapsed time (mult_seq): 0.846955s
# elapsed time (mult_seq): 0.848383s
# elapsed time (mult_par): 0.279757s
# elapsed time (mult_par): 0.288949s
# elapsed time (mult_par): 0.292137s
# elapsed time (overall): 13.7698s
```

隐含同步

```
#pragma omp parallel for
for(...) {
  ...
}
...
#pragma omp parallel for
for(...) {
  ...
}
...

// 上面代码有一个缺点，每个 for 循环开始时，会生成线程组，结束时，销毁，那么可不可以在整个算法逻辑中只生成一次线程组，作用于所有 for 循环，从而提高效率呢？答案是可以的，但是此时要注意隐含同步问题

#pragma omp parallel
{ //此时线程组生成
    #pragma omp for
    for (uint64_t i = 0; i < num_entries; ++i) {
        x[i] = i;
        y[i] = num_entries - i;
    }
    // implicit barrier 隐式屏障(同步)，空闲线程会等待，直到所有该for循环的线程任务完成

    #pragma omp for
    for (uint64_t i = 0; i < num_entries; ++i) {
        z[i] = x[i] + y[i];
    }
    // implicit barrier

    #pragma omp for
    for (uint64_t i = 0; i < num_entries; ++i) {
        if(z[i] - num_entries) {
            cout << "error at position: " << i << endl;
        }
    }
    // implicit barrier
} // join all threads
```

```
// final implicit barrier
```

如果想要破坏隐式同步，从而让多个 for循环并行执行，可以使用 `nowait` 语句显示移除屏障，这能优化运行时间

```
#pragma omp parallel
{ //此时线程组生成
    #pragma omp for nowait
    for (...) {...}
    // 没有屏障
    #pragma omp for nowait
    for (...) {...}
} // 此时，两个 for 循环能够并行执行
```

> 需要强制同步的地方也可以使用 显示同步屏障制导语句 `#pragma omp barrier` ， 线程执行到此会被阻塞，直到所有线程都同步于此

私有变量

```
#1 正确，每个线程会持有独立的 j 变量
int main(){
  #pragma omp parallel for
  for(int i=0; i < 10; i++)
    for(int j=0; j < 10; j++)
      do_something(i, j);
}

#2 错误，每个线程操作共享的 j 变量
int main(){
  int i, j;
  #pragma omp parallel for
  for(i=0; i < 10; i++)
    for(j=0; j < 10; j++)
      do_something(i, j);
}

#3 正确，每个线程持有独立的 j 变量
int main(){
  int i, j;
  #pragma omp parallel for private(j) // 声明 j 为每个线程的私有变量
  for(i=0; i < 10; i++)
    for(j=0; j < 10; j++)
      do_something(i, j);
}
```

```
#4 初始化线程私有变量
int main(){
  int i=0;
  #pragma omp parallel  private(i)
  do_something(i); // 错误，线程私有变量必须在并行区域中初始化，否则会产生为定义行为
}

int main(){
  int i=0;
  #pragma omp parallel  private(i)
  {
    i = 10; // 正确，线程私有变量 i 在并行区域中初始化
    do_something(i);
  }
}

#5 拷贝初始化
int main(){
  int i=10;
  #pragma omp parallel for firstprivate(i)   //正确，会拷贝 i 到每个线程
  {
      do_something(i);
  }
}

#6 捕获线程私有变量，假如想在并行区域外，访问到线程的私有变量的值，可以通过吧线程私有变量写入到全局数组中实现
int main(){
  const int num = omp_get_max_threads();
  int* aux = new int[num];
  int i = 1;

  #pragma omp parallel firstprivate(i) num_threads(num)
  {
    const int j = omp_get_thread_num(); // 获得线程编号
    i += j;
    aux[j] = i; // 写回全局数组
  }
  for(int j=0; j<num; j++)
    printf("%d ", aux[j]);'

  delete[] aux;
}
-------
1 2 3 4 5 6...
```

锁

```
#include <iostream>
#include <omp.h>

static omp_lock_t lock; // 定义锁
void putMes(int i)
{
    std::cout << i << ":AA" << std::endl;
    omp_set_lock(&lock); //获得锁
    std::cout << i << ":BB1" << std::endl;
    std::cout << i << ":BB2" << std::endl;
    omp_unset_lock(&lock); //释放锁
}

int main()
{
    omp_init_lock(&lock); // 初始化互斥锁，必须
    #pragma omp parallel for
    for (int i = 0; i < 4; ++i)
    {
        putMes(omp_get_thread_num()); // 获取当前执行的线程编号
    }
    omp_destroy_lock(&lock); //销毁互斥器
    return 0;
}

--------------------
3:AA
3:BB1
3:BB2
0:AA
1:AA
2:AA
1:BB1
1:BB2
2:BB1
2:BB2
0:BB1
0:BB2
```

临界区

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

int main()
{
    int sum = 0;
    #pragma omp parallel for
    for (int i = 0; i < 10000; ++i)
    {
    #pragma omp critical  // 临界区，一个时刻只能有一个线程执行 {} 内的逻辑
        {
          sum = sum + i%7;
        }
    }
    printf("Sum: %d\n" ,sum);
    return 0;
}
```

原子操作

```
#pragma omp atomic
count = count+10;
```

> openmp 中锁，临界区，原子操作耗时比例约为 3.5 : 7 : 1 (非官方数据)

高级使用

数据同步问题

场景： 外部有一全局变量 sum， 并行区域代码做一些运算，最后汇总到 sum 上， 解决这种数据同步问题在 基础使用方法中已经给出了两种解决方案， 第一种：维护一个全局数组， 各个线程运算的结果写回数组，最后统计数组的和即为 sum；第二种：上锁或使用临界区，原子操作；这里给出第三种方案，利用 openmp 原语 `#pragma omp for reduction` ，reduction 原语指定一个变量，与相应的操作符，然后每个线程都会创建一份私有变量(初始值为 0)，在并行区域区域结束后，各个线程的私有拷贝通过 reduction 指定的操作符进行运算，最后赋值给原始变量。

```
int main()
{
  int sum = 100;
  #pragma omp parallel for num_threads(4) reduction(+:sum) // 归约运算符 +
  for(int i = 0; i < 8; i++)
```

```
  {
    int id = omp_get_thread_num();
    sum += id;
    printf("%d->%d\n",id, sum);
  }
  // 0-3 号线程 sum值分别为 0 2 4 6,  加上主线程值为 112
  printf("Sum: %d\n",sum);

  return 0;
}
```

归约运算符只支持 +, *, -, &, |, ^, &&, ||， 八种，不支持除法(因为除法不满足交换律)， 虽然减法也不满足交换律，但是减法可以转换为加法，此时要注意 reduction(-:sum)本质是在做加法，所以不能想当然，例子如下：

```
int main()
{
  int sum = 100;
  #pragma omp parallel for num_threads(4) reduction(-:sum) // 归约运算符 -
  for(int i = 0; i < 8; i++)
  {
    int id = omp_get_thread_num();
    sum += id;
    printf("%d->%d\n",id, sum);
  }
  // 0-3 号线程 sum 值分别为 0 2 4 6,  此时 sum 应该是 100 - 12 吗?  不对依然是 100 + 12 为 112,  正确的写法要将线程私有的 sum 计算成负数
  printf("Sum: %d\n",sum);
  return 0;
}
```

> 若 reduction 里调用其它自定义数值处理函数，记得传指针或者引用
>
> OpenMP 4.0 支持自定义归约操作符

合并 for 循环

如下的 for 循环，当 parallel for 制导语句作用于外层 for 循环，若外层循环数很小，而内层很大，则会导致启用的线程数很少，每个线程的负载很大，运算效率不高。 若 parallel for 制导语句作用于内层 for 循环，且外层循环数很大，而内层很小，则会导致线程数过多，每个线程负载过小，致线程的计算时间相比线程的建立、销毁时间不够长，合并 for 循环制导语句 `collapse` 可以一定程度解决负载均衡问题

```
for(int i = 0; i < outer_loop; i++) {
 for(int j = 0; j < inner_loop; j++) {
  do_something(i, j);
 }
}

#pragma omp parallel for collapse(2) //collapse(2)表示同时线程化接下来的两层循环，这样循环次数就变成了 outer_loop × inner_loop，这既改善了负载均衡性，又增加了每个线程的计算量。
for(int i = 0; i < 3; i++){
   for(int j = 0; j < num; j++){
    do_something(i, j);
 }
}
```

线程任务分配

```
#pragma omp parallel
{
    #pragma omp master  // 有且只有主线程能执行 {} 中的代码
    {
        cout << "thread_id: " << omp_get_thread_num() << "->";
        for (int i = 0; i < 10; i++) {
            cout << i << endl;
        }
    }
    cout << "sub threads is running" << endl; // 子线程执行此处代码
}

// 静态创建任务
#pragma omp parallel sections
{
    #pragma omp section // 一个 section 只会被一个线程执行一次， section 之间是并行的
    {
        printf("section 1,threadid=%d\n",omp_get_thread_num());
    }

    #pragma omp section
    {
        printf("section 2,threadid=%d\n",omp_get_thread_num());
    }

    #pragma omp section
    {
        printf("section 3,threadid=%d\n",omp_get_thread_num());
    }
```

```
}

// 然而上述代码有缺陷， 如下，当任务过多时，产生的代码量太长且重复，考虑动态创建任务
int main()
{
  int a[3]{0};
  #pragma omp parallel
  {
      #pragma omp sections
      {
      #pragma omp section
            do_something(a[0]);
      #pragma omp section
            do_something(a[1]);
      #pragma omp section
            do_something(a[2]);
      }
  }
  return 0;
}

// 动态创建任务
#pragma omp parallel
{
    #pragma omp single  // 表示只有一个线程会执行 {} 的语句
    {
      for(int i = 0; i < N; i++)
      {
        #pragma omp task  // 动态创建 tasks ，空闲线程会去执行 task
        do_something(a[i]);
      }
    }
}
```

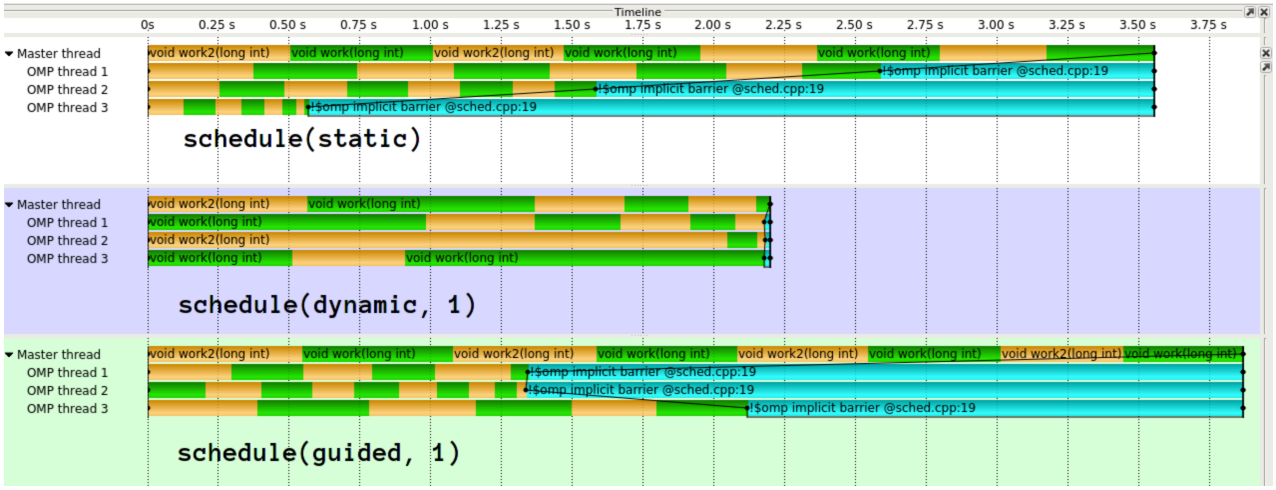
线程调度机制

OpenMP 提供了开箱即用的线程调度模式，简要说明如下：

- static 模式(默认)：所有的迭代(for 循环的范围)粗略分成 m/c 块，每个块在 m 个索引上执行 c 次顺序运算， 这些大小为 c 的块组合以循环方式顺序分配给线程，假如一个线程完成了分配给它的块任务时，它将处于空闲状态，等待其他所有线程执行完毕。 块大小 c 默认等于 m/num_of_threads，通俗来说就是几个线程几个块
- dynamic 模式：所有的迭代也再次划分成相等大小的块，并且一个接一个的分配给等待工作的执行线程，因此，空闲只能发生在当一个线程执行的是最后一个块任务时，假如没有规定大小，块的大小默认为1，通俗来说就是多个块，动态分配给线程组
- guided 模式：所有的迭代划分成大小递减的块(用户可以自定义最小的大小)，块的分配和动态分配机制一样，假如用户没有自定义最小块的大小，默认值为 m/num_of_threads

- auto 模式：编译器或者运行时操作系统将自动选择上述某一种调度模式
- reuntime 模式：按照环境变量 OMP_SCHEDULE ，运行时操作系统将决定采用哪种调度模式

在执行的任务大小不一致的时候，才需要使用线程调度模式，例如外层 for 循环为 0-100， 默认 4 个线程执行，那么每个线程执行 25 次迭代，若 for 循环中的子任务和循环次数挂钩，例如 do_something(i) { for(0 ... i) ... } 那么此时此刻，采用默认调度，会导致最后一个线程执行的是编号 75-99 的迭代，而子任务会变的很大，那么该线程的执行时间就会变长，与之相对的是第一个线程，执行的是 0-24 迭代，此线程执行时间较短，会存在大量空闲时间，整个线程组负载不均衡。 此时此刻可以考虑使用 static 指定小块， 或者 dynamic 模式， guided 模式优化，具体情况具体分析，此处仅仅抛砖引玉



举例：计算内积函数

```
#define MODE dynamic

template <typename value_t,
          typename index_t>
void inner_product(value_t * data,
                   value_t * delta,
                   index_t num_entries,
                   index_t num_features,
                   bool    parallel) {

    #pragma omp parallel for schedule(MODE) if(parallel)
    for (index_t i = 0; i < num_entries; i++)
        for (index_t j = i; j < num_entries; j++) {
            value_t accum = value_t(0);
            for (index_t k = 0; k < num_features; k++)
                accum += data[i*num_features+k] *
                         data[j*num_features+k];
            delta[i*num_entries+j] =
            delta[j*num_entries+i] = accum;
        }
}

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

```
    // run parallelized when any command line argument given
    const bool parallel = argc > 1;

    std::cout << "running "
              << (parallel ? "in parallel" : "sequentially")
              << std::endl;

    // the shape of the data matrices
    const uint64_t num_features = 28*28;
    const uint64_t num_entries = 65000;

    TIMERSTART(alloc)
    // memory for the data matrices and all-pair matrix
    std::vector<float> input(num_entries*num_features);
    std::vector<float> delta(num_entries*num_entries);
    TIMERSTOP(alloc)

    TIMERSTART(read_data)
    // get the images and labels from disk
    load_binary(input.data(), input.size(), "./data/X.bin");
    TIMERSTOP(read_data)

    TIMERSTART(inner_product)
    // start computing task
    inner_product(input.data(), delta.data(),
                  num_entries, num_features, parallel);
    TIMERSTOP(inner_product)
}
```

分别采取以下调度策略：

- **SPB**： #define MODE static （静态默认块）
- **SPC**： #define MODE static,1（静态小块）
- **SBC**： #define MODE static,32（静态大块）
- **DPC**：#define MODE dynamic （动态默认）
- **DBC**：#define MODE dynamic,32 （动态大块）

性能	SPB	SPC	SBC	DPC	DBC
时间(s)	71.0	35.4	36.5	33.8	34.1
加速比	26.4	53.9	51.3	55.4	55.0

> 注：此为在 32核 64线程机器执行结果，串行执行耗时约 30min

参考资料

并行程序设计-概念与实践 (Parallel Programming: Concepts and Practice)

Choosing between OpenMP and Explicit Threading Methods

Guide into OpenMP: Easy multithreading programming for C++

OpenMP: For & Scheduling

源码链接

OpenMP_Tutorial