### 团队项目

#### 项目组成员

|  |  |  |  |
| --- | --- | --- | --- |
| 学号 | 姓名 | 项目分工 | 贡献占比 |
| 202030441031 | 黄耀熙 | 找并行算法资料、学习并行共轭梯度算法、测试代码运行结果、写大作业文档团队部分 | 40% |
| 202030443103 | 夏志勇 | 学习oneAPI技术、编写代码、展示汇报 | 30% |
| 202030445428 | 鄢志宇 | 学习oneAPI技术、编写代码、写大作业文档团队部分 | 30% |

#### 项目简介

|  |  |
| --- | --- |
| 项目名称 | 并行共轭梯度算法的开发 |
| 项目摘要或简介 | 共轭梯度算法也叫共轭斜量法。从理论上讲它是属于直接法，但在实际计算过程中，由于不可避免地会出现误差，所以常作为迭代法使用。该方法的最大特点是，当方程组的阶数很高时，往往只要经过比阶数小得多的迭代次数，就能得到满足精度要求的近似解。  基本思想：共轭梯度法是属于最小化类的迭代方法。为了求解Ax = b这样的一次函数，可以先构造一个二次函数  然后利用最小化原理对其求导，再令其为零。这样求解Ax = b的值可以转换为求解f(x)的最小值。  使用openMP共轭梯度算法的串行以及并行版本，并统计在不同数据规模以及不同的并行线程数下，共轭梯度算法完成运算所耗费的时间并比较。 |

#### 项目背景

* **共轭梯度法**

共轭梯度法（Conjugate Gradient）是介于最速下降法与牛顿法之间的一个方法，它仅需利用一阶导数信息，但克服了最速下降法收敛慢的缺点，又避免了牛顿法需要存储和计算Hesse矩阵并求逆的缺点，共轭梯度法不仅是解决大型线性方程组最有用的方法之一，也是解大型非线性最优化最有效的算法之一。 在各种优化算法中，共轭梯度法是非常重要的一种。其优点是所需存储量小，具有步收敛性，稳定性高，而且不需要任何外来参数。

共轭梯度法最早是由Hestenes和Stiefle提出来的，在这个基础上，Fletcher和Reeves 首先提出了解非线性最优化问题的共轭梯度法。由于共轭梯度法不需要矩阵存储，且有较快的收敛速度和二次终止性等优点，现在共轭梯度法已经广泛地应用于实际问题中。

共轭梯度法是一个典型的共轭方向法，它的每一个搜索方向是互相共轭的，而这些搜索方向d仅仅是负梯度方向与上一次迭代的搜索方向的组合，因此，存储量少，计算方便。

* **当作为直接法时**

假设两个非零向量u和v是共轭的（相对于A），如果

由于A是对称和正定的，所以左侧定义了内积：

当且仅当它们相对于该内积正交时，两个向量是共轭的。 共轭是一个对称关系：如果u与v共轭，则v与u共轭。假设

是一组n维共轭向量。那么P 组成了的基，在此基础上我们表述Ax=b解为：

基于这个扩展，我们计算：

其中：

这给出了求解方程的以下方法：Ax = b：找到n个共轭方向的序列，然后计算系数。

* **当作为迭代法时**

如果仔细选择共轭向量，那么可能不需要所有这些来获得解x\*的好的近似值。因此，将共轭梯度法视为迭代法。这也能够大致解决n大到非常大的系统，因为直接方法花费太多时间。

假设是的初始值，我们可以不失一般性的假设 (否则，考虑用系统代替)。

从 开始搜索解决方案，在每次迭代中，需要一个指标来说明是否更接近解决方案。

该度量来自于以下事实：解也是以下二次函数的唯一最小值：

这表明在处采用第一基矢量为f的梯度的负值。f的梯度等于Ax-b。 从“猜测的解决方案”x0开始（如果没有理由猜测其他任何东西，则总是猜测），这意味着我们取。

在此基础上的其他向量将与梯度共轭，因此称为共轭梯度法。

让成为第k步的残差：

注意，是处的f的负梯度，因此梯度下降法将是沿方向移动。 在这里，坚持方向彼此共轭。还要求下一个搜索方向由当前的残差和所有先前的搜索方向构成，这在实践中是足够合理的。

共轭约束是一种正交类型约束，因此该算法与Gram-Schmidt正交归一化相似。

给出以下表达式：

按照这个方向，下一个最佳位置由下式给出：

其中

由于和是共轭的，所以最后一个等式成立。

* **openMP编程**

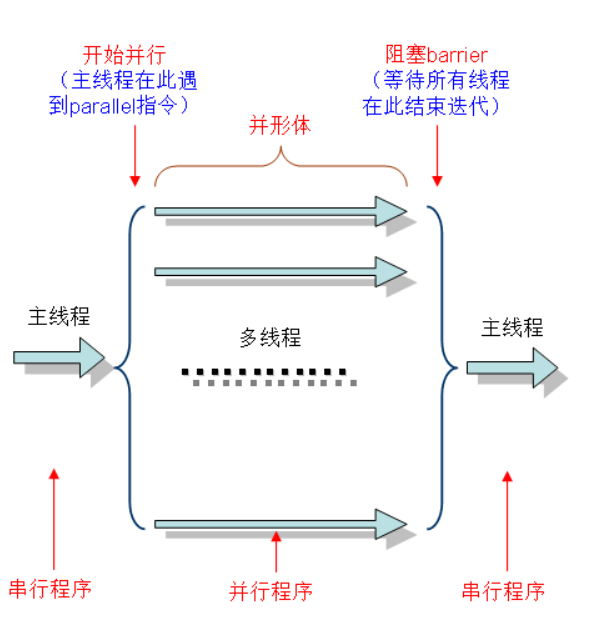
OpenMp是由OpenMP Architecture Review Board牵头提出的，并已被广泛接受的，用于共享内存并行系统的多线程程序设计的一套指导性的编译处理方案(Compiler Directive)。OpenMP支持的编程语言包括C语言、C++和Fortran;而支持OpenMp的编译器包括Sun Compiler，GNU Compiler和Intel Compiler等。OpenMp提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的pragma来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。当选择忽略这些pragma，或者编译器不支持OpenMp时，程序又可退化为通常的程序(一般为串行)，代码仍然可以正常运作，只是不能利用多线程来加速程序执行

OpenMp提供的这种对于并行描述的高层抽象降低了并行编程的难度和复杂度，这样程序员可以把更多的精力投入到并行算法本身，而非其具体实现细节。对基于数据分集的多线程程序设计，OpenMP是一个很好的选择。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。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。

* **OpenMP执行模式**

OpenMP采用fork-join的执行模式。开始的时候只存在一个主线程，当需要进行并行计算的时候，派生出若干个分支线程来执行并行任务。当并行代码执行完成之后，分支线程会合，并把控制流程交给单独的主线程。

一个典型的fork-join执行模型的示意图如下：



OpenMP编程模型以线程为基础，通过编译制导指令制导并行化，有三种编程要素可以实现并行化控制，他们分别是编译制导、API函数集和环境变量。

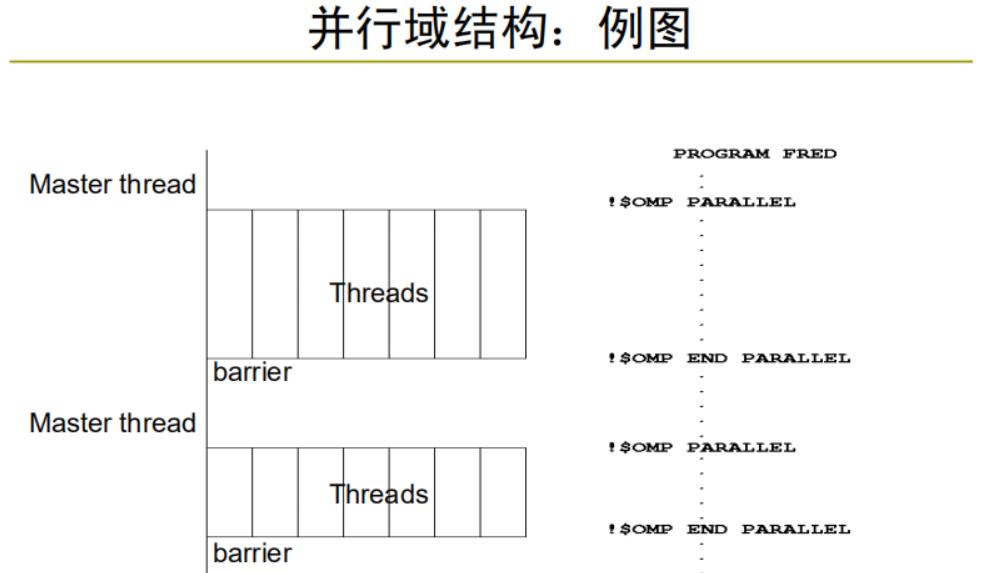
* **编译制导**

OpenMP的并行化是通过使用嵌入到C/C++或Fortran源代码中的编译制导语句来实现. 通过对串行程序添加制导语句实现并行化。支持并行区域、工作共享、同步等。支持数据的共享和私有化。支持增量并行。

编译制导指令以#pragma omp 开始，后边跟具体的功能指令，格式如：**#pragma omp 指令[子句[,子句] …]**。

并行域制导一个并行域就是一个能被多个线程并行执行的程序段

在并行域结尾有一个隐式同步（barrier）。子句（clause）用来说明并行域的附加信息。C/C++子句间用空格分开。



**常用的功能指令如下：**

1. **parallel**：用在一个结构块之前，表示这段代码将被多个线程并行执行；
2. **for**：用于for循环语句之前，表示将循环计算任务分配到多个线程中并行执行，以实现任务分担，必须由编程人员自己保证每次循环之间无数据相关性；
3. **parallel** **for**：parallel和for指令的结合，也是用在for循环语句之前，表示for循环体的代码将被多个线程并行执行，它同时具有并行域的产生和任务分担两个功能；
4. **sections**：用在可被并行执行的代码段之前，用于实现多个结构块语句的任务分担，可并行执行的代码段各自用section指令标出（注意区分sections和section）；
5. **parallel** **sections**：parallel和sections两个语句的结合，类似于parallel for；
6. **single**：用在并行域内，表示一段只被单个线程执行的代码；
7. **critical**：用在一段代码临界区之前，保证每次只有一个OpenMP线程进入；
8. **flush**：保证各个OpenMP线程的数据影像的一致性；
9. **barrier**：用于并行域内代码的线程同步，线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行；
10. **atomic**：用于指定一个数据操作需要原子性地完成；
11. **master**：用于指定一段代码由主线程执行；
12. **threadprivate**：用于指定一个或多个变量是线程专用，后面会解释线程专有和私有的区别。

**常用子句：**

1. **private**：指定一个或多个变量在每个线程中都有它自己的私有副本；
2. **firstprivate**：指定一个或多个变量在每个线程都有它自己的私有副本，并且私有变量要在进入并行域或任务分担域时，继承主线程中的同名变量的值作为初值；
3. **lastprivate**：是用来指定将线程中的一个或多个私有变量的值在并行处理结束后复制到主线程中的同名变量中，负责拷贝的线程是for或sections任务分担中的最后一个线程；
4. **reduction**：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归约运算，并将结果返回给主线程同名变量；
5. **nowait**：指出并发线程可以忽略其他制导指令暗含的路障同步；
6. **num\_threads**：指定并行域内的线程的数目；
7. **schedule**：指定for任务分担中的任务分配调度类型；
8. **shared**：指定一个或多个变量为多个线程间的共享变量；
9. **ordered**：用来指定for任务分担域内指定代码段需要按照串行循环次序执行；
10. **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；
11. **copyin**：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；
12. **default**：用来指定并行域内的变量的使用方式，缺省是shared。

**API函数：**

1. **omp\_in\_parallel**: 判断当前是否在并行域中；
2. **omp\_get\_thread\_num**: 返回线程号；
3. **omp\_set\_num\_threads**: 设置后续并行域中的线程格式；
4. **omp\_get\_num\_threads**: 返回当前并行域中的线程数；
5. **omp\_get\_max\_threads** :获得并行域中可用的最大线程数
6. **omp\_get\_num\_procs**: 返回系统中处理器的个数
7. **omp\_get\_dynamic** :判断是否支持动态改变线程的数目
8. **omp\_set\_dynamic**: 启动或者关闭线程数目的动态改变
9. **omp\_get\_nested:** 判断系统是否支持并行嵌套
10. **omp\_set\_nested**: 启动或者关闭并行嵌套
11. **omp\_init\_lock** 初始化一个简单锁
12. **omp\_set\_lock** 上锁操作
13. **omp\_unset\_lock**解锁操作，要omp\_set\_lock函数配对使用。
14. **omp\_destroy\_lock**， omp\_init\_lock函数的配对操作函数，关闭一个锁

**环境变量：**

1. **OMP\_SCHEDULE**：用于for循环并行化后的调度，它的值就是循环调度的类型；
2. **OMP\_NUM\_THREADS**：用于设置并行域中的线程数；
3. **OMP\_DYNAMIC**：通过设定变量值，来确定是否允许动态设定并行域内的线程数；
4. **OMP\_NESTED**：指出是否可以并行嵌套

#### 项目内容和方案

矩阵是数值代数中的一个基本概念，许多科学计算问题往往归结为对矩阵的操作，如数值天气预报、三维图像处理、地震分析等，矩阵乘法，求解线性方程组和矩阵特征值等问题是矩阵计算最基本的内核。由于矩阵的运算，特别是大规模矩阵相乘，单处理机已经无法承受，因此有效地实现大型的矩阵并行算法在实际应用中是非常重要的。在海量数据的时代，数据处理越来越大，计算越来越复杂，并行模型也越来越丰富，比如：Map/Reduce、MPI、openMP、Pthread等等。在此我们选择openMP作为我们实践并行处理解决矩阵运算问题的接口。

项目内容：

开发矩阵运算代码（包括矩阵加法运算、矩阵乘法运算openmp）、开发并行共轭梯度算法

**矩阵相加：**

oneAPI程序的例子vectorAdd\_dpcpp.cpp，其功能为计算两个一维向量的相加。编译器使用dpcpp，具体编译命令为：dpcpp vectorAdd\_dpcpp.cpp -o vectorAdd\_dpcpp

#include <CL/sycl.hpp>

using namespace sycl;

static const size\_t numElements = 50000;

void work(queue &q) {

std::cout << "Device : "

<< q.get\_device().get\_info<info::device::name>()

<< std::endl;

float vector1[numElements] ;

float vector2[numElements] ;

float vector3[numElements];

auto R = range(numElements);

for (int i = 0; i < numElements; ++i) {

vector1[i] = rand()/(float)RAND\_MAX;

vector2[i] = rand()/(float)RAND\_MAX;

}

//创建vector1、vector2、vector3向量的SYCL缓冲区；

buffer vector1\_buffer(vector1,R);

buffer vector2\_buffer(vector2,R);

buffer vector3\_buffer(vector3,R);

//向Device提交工作（定义了访问缓冲区内存的accessor；）

q.submit([&](handler &h) {

accessor v1\_accessor (vector1\_buffer,h,read\_only);

accessor v2\_accessor (vector2\_buffer,h,read\_only);

accessor v3\_accessor (vector3\_buffer,h);

//调用oneAPI的核函数在Device上完成指定的运算；

h.parallel\_for (range<1>(numElements), [=](id<1> index) {

//核函数部分，若单独写一个函数，直接使用函数名调用即可

if (index < numElements)

v3\_accessor [index] = v1\_accessor [index] + v2\_accessor [index];

});

}).wait();

//排队等待

// 将SYCL缓冲区的数据读到Host端，检查误差

host\_accessor h\_c(vector3\_buffer,read\_only);

for (int i = 0; i < numElements; ++i) {

if (fabs(vector1[0] + vector2[0] - vector3[0] ) > 1e-8 ) {

fprintf(stderr, "Result verification failed at element %d!\n", i);

exit(EXIT\_FAILURE);

}

}

}

int main() {

try {

queue q;

work(q);

} catch (exception e) {

std::cerr << "Exception: " << e.what() << std::endl;

std::terminate();

} catch (...) {

std::cerr << "Unknown exception" << std::endl;

std::terminate();

}

}

DPC++程序设计大致可分为以下5个步骤：

（1）申请Host内存

（2）创建SYCL缓冲区并为其定义访问缓冲区内存的方法

设备(device)和主机(host)可以共享物理内存或具有不同的内存。当内存不同时，卸载计算需要在主机和设备之间复制数据。而通过创建缓冲区(buffer)和访问器(accessor)的方式，DPC++就不需要管理数据复制，其能够确保数据可供主机和设备使用，而无需介入。DPC++ 还允许明确地显式控制数据移动，以实现最佳性能

（3）创建队列以向Device（包括Host）提交工作（包括选择设备和排队）

可以通过选择器(selector)选择 CPU、GPU、FPGA和其他设备。使用默认的 q，这意味着 DPC++ 运行时会使用默认选择器（default selector）选择功能最强大的设备

（4）调用oneAPI的核函数在Device上完成指定的运算

（5）将SYCL缓冲区的数据读到Host端

**矩阵乘法openmp：**

从串行到并行：

一个正常的C语言程序，从main函数开始，以return 0为结束，往往是串行的。而openmp程序从串行区开始，中间并行区派生出多个线程处理工作，最后可能再回到串行区。所以说，OpenMP程序通常采用串行→并行→串行→并行→串行，而不是整个程序都采用并行机制去运行。

通常For循环的并行化：

//设置启用3个线程

omp\_set\_num\_threads(3);

// 并行编译指导语句，代表着并行区域的开始

#pragma omp paralle

//获取当前的进程号  
tid=omp\_get\_thread\_num();

//获取总进程数  
nthreads=omp\_get\_num\_threads();

循环的并行化，是比较独立和重要的，也是最简单有效的并行。类比于如果循环100次，我们采用1个线程跑100次，和4个线程每个跑25次，需要的时间肯定是后者要短，而且要短的多。

#pragma omp for

通过上述指令，就可以对循环进行并行化，来达到加速程序的目的

入门学习openmp：

通过一个简单的openmp一维数组加法程序来入门学习openmp：

要求：要计算c[i]数组的值，c[i]=a[i]+b[i]  
如果我们设置3个线程，i=3，则1个线程做1次加法

#include<iostream>

#include<omp.h>

using namespace std;

int main()

{

int tid; //表示当前线程号

int nthreads; //表示所有的线程数

omp\_set\_num\_threads(3); //设置启用3个线程

int a[3]={1,2,3};

int b[3]={1,2,3};

int c[3];

/\*并行区域开始\*/

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

{

#pragma omp for

for(int i=0;i<3;i++)

{

tid=omp\_get\_thread\_num(); //获取当前的进程号

nthreads=omp\_get\_num\_threads(); //获取总进程数

c[i]=a[i]+b[i];

cout<<"线程数："<<nthreads<<"，线程号："<<tid;

cout<<",c["<<i<<"]="<<c[i]<<endl;

}

}

/\*并行区域结束\*/

return 0;

}

输出结果：

线程数：3，线程号：2,c[2]=6  
线程数：3，线程号：0,c[0]=2  
线程数：3，线程号：1,c[1]=4

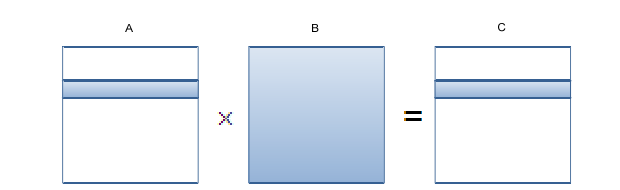
分析：

由于多个线程的执行顺序是随机的，因此结果可能是乱序。程序从串行区开始，设置了3个线程，到并行区域，派生出3个线程，分别执行c[i]=a[i]+b[i]加法运算。当遇见了},也就是并行语句块结束，继续执行后续代码，当然后续可能是串行区，也可能是并行区。

事实上，如果把数组改换为[二维数组](https://so.csdn.net/so/search?q=%E4%BA%8C%E7%BB%B4%E6%95%B0%E7%BB%84&spm=1001.2101.3001.7020)，二维数组的加法运算也是在for循环下的，而实际上，二维数组就是矩阵，也就是说，我们其实用openmp实现了矩阵的加法运算。

下面通过改进来实现矩阵的乘法运算：

已知两个方阵A[n][n],B[n][n]（n>=1000），C[n][n]=A[n][n]\* B[n][n]。矩阵相乘的关键是相乘的两个元素的下标要满足一定的要求(即对准)，计算C 的任务需要访问 A 的第 i 行和 B 的第 j 列。基于行的划分方式中,把A按行均分为n等份，然后把每一份各自传到n个线程中去分别计算，对于B，因为A中每一行都要和B中各列分别计算，所以B广播到各个线程中



#include<omp.h>

#include<stdio.h>

#include<time.h>

#define NN 2000

int a[NN][NN], b[NN][NN];

long long c[NN][NN];

void getTime(int n, int num\_thread)

{

int i, j, k, temp;

clock\_t startTime, endTime;

double totalTime = 0;

long long sum = 0;

omp\_set\_num\_threads(num\_thread);

for (i = 0; i < n; i++)//初始化

{

temp = i + 1;

for (j = 0; j < n; j++)

{

a[i][j] = temp++;

b[i][j] = 1;

}

}

// for (i = 0; i < n; i++)//输出矩阵

// {

// for (j = 0; j < n; j++)

// {

// printf("%d,%d\n", a[i][j], b[i][j]);

// }

//}

startTime = clock();

#pragma omp parallel shared(a,b,c) private(i,j,k)

{

#pragma omp for schedule(dynamic)

//矩阵乘法实现

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

c[i][j] = 0;

for (k = 0; k < n; k++)

{

c[i][j] += a[i][k] \* b[k][j];

}

}

}

}

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

sum += c[i][j];

endTime = clock();

totalTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

printf("sum=%lld time = %.21fs\n", sum, totalTime);

}

int main()

{

int n, num\_thread;

while (scanf\_s("%d",&num\_thread) != EOF)

{

n = 1000;

getTime(1000, num\_thread);

}

return 0;

}

**共轭梯度算法：**

完成矩阵的加法和乘法运算入门openmp后，进行并行共轭梯度算法的开发。

共轭梯度算法也叫共轭斜量法。从理论上讲它是属于直接法，但在实际计算过程中，由于不可避免地会出现误差，所以常作为迭代法使用。

理解：将共轭性与最快速下降法结合，利用已知点处的梯度构造一组共轭向量，并沿这组方向进行搜索，求出目标函数的极小点

性质：1.共轭梯度法中的搜索方向d0、d1…是一组共轭方向。

2.对于任意初始点x0，都能在n次迭代内收敛到唯一的全局极小点。

因此该方法的最大特点是，当方程组的阶数很高时，往往只要经过比阶数小得多的迭代次数，就能得到满足精度要求的近似解

基本思想：共轭梯度法是属于最小化类的迭代方法。为了求解Ax = b这样的一次函数，可以先构造一个二次函数

然后利用最小化原理对其求导，再令其为零。这样求解Ax = b的值可以转换为求解f(x)的最小值。

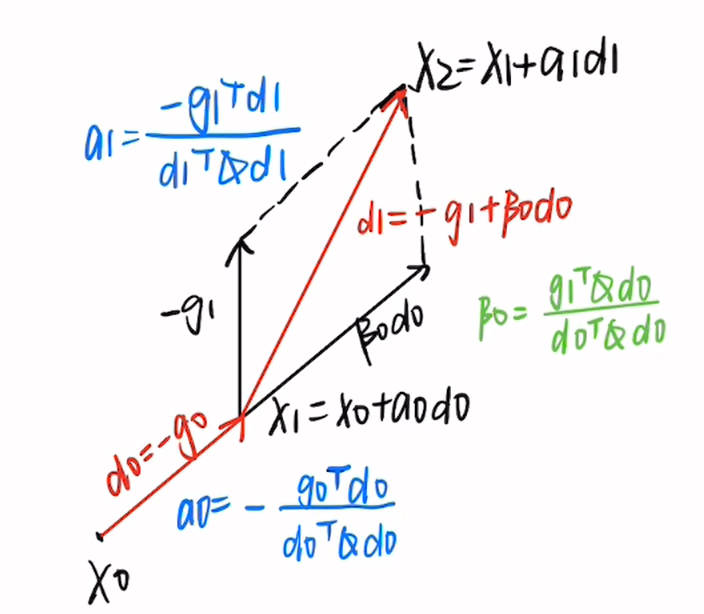
共轭梯度算法的步骤：

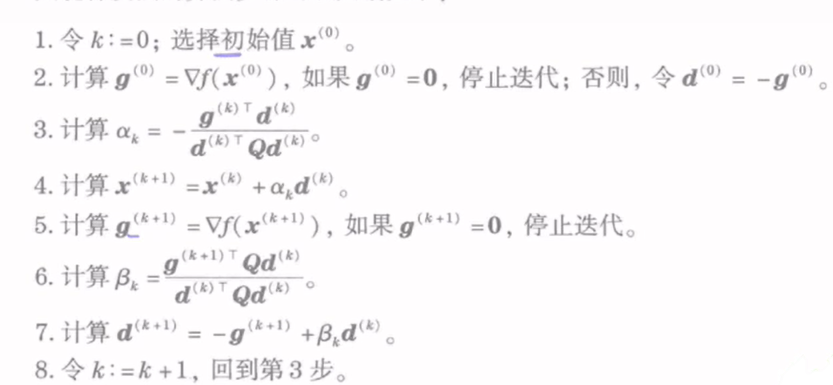
初始化,x(k)表示第k次迭代的解向量。d(k)表示第k次迭代的方向向量。r(k)表示第k次迭代的残差向量。

接下来进行k此迭代，主要分为四个步骤

1. 计算残差向量：
2. 计算方向向量：
3. 计算步长：
4. 更新解向量：

算法步骤图示化：





共轭梯度算法伪代码：

输入：A(n\*n矩阵)，b(n\*1)矩阵

输出：x(n\*1)矩阵

x = [0,0...0],d = [0,0...0],r = -b;

for (i = 1:n){ //n次迭代

denom1 = r^T\*r //计算r的转置乘以r

r = A\*x-b //计算残差r = Ax-b

num1 = r^T\*r; //计算r的转置乘以r

if(num1 < 精度)

break;

//计算方向向量d

d = -r + num1/denom1\*d;//d和r都是向量

num2 = d^T\*r;//计算d的转置乘以r

denom2 = d^T\*(A\*d) //计算d的转置乘以A乘以d最后

a = -num2/denom2; //计算步长

x = x+Ad // 修正x

}

print(x);

## 开发串行共轭梯度算法：

源码如下：

#define WIN32

#include <time.h>

#ifdef WIN32

#include <windows.h>

#else

#include <sys/time.h>

#endif

#ifdef WIN32

int gettimeofday(struct timeval\* tp, void\* tzp)

{

time\_t clock;

struct tm tm;

SYSTEMTIME wtm;

GetLocalTime(&wtm);

tm.tm\_year = wtm.wYear - 1900;

tm.tm\_mon = wtm.wMonth - 1;

tm.tm\_mday = wtm.wDay;

tm.tm\_hour = wtm.wHour;

tm.tm\_min = wtm.wMinute;

tm.tm\_sec = wtm.wSecond;

tm.tm\_isdst = -1;

clock = mktime(&tm);

tp->tv\_sec = clock;

tp->tv\_usec = wtm.wMilliseconds \* 1000;

return (0);

}

#endif

#include<iostream>

#include<vector>

#include<cmath>

#include<stdio.h>

#include "omp.h"

using namespace std;

int N = 1000;

//初始化A为一个N\*N的矩阵的对称矩阵

vector<vector<double> > A(N, vector<double>(N, 0));

//初始化数组b

vector<double> b(N, 1);

//初始化残差r,结果x,计算方向向量d

vector<double> r(N, -1);

vector<double> d(N, 0);

vector<double> x(N, 0);

void displayA(vector<vector<double> >& a) {

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

for (int j = 0; j < N; j++) {

cout << a[i][j] << " ";

}

cout << endl;

}

}

void displayb(vector<double>& b) {

for (int i = 0; i < b.size(); i++) {

cout << b[i] << " ";

}

cout << endl;

}

//计算内积，也就是模的平方

double INNER\_PRODUCT(vector<double>& a, vector<double>& b) {

double res = 0;

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

res += a[i] \* b[i];

}

return res;

}

//更新残差 r = A\*x-b

vector<double>& MATRIX\_VECTOR\_PRODUCT(vector<vector<double> >& a, vector<double>& x, vector<double>& b) {

double temp = 0;

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

temp = 0;

for (int j = 0; j < N; j++) {

temp += a[i][j] \* x[j];

}

r[i] = temp - b[i];

}

return r;

}

//计算dtAd

double MATRIX\_PRODUCT(vector<vector<double> >& a, vector<double>& d) {

double res = 0;

double temp = 0;

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

temp = 0;

for (int j = 0; j < N; j++) {

temp += d[j] \* A[i][j];

}

res += temp \* d[i];

}

return res;

}

int main() {

//初始化A

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

for (int j = 0; j < N; j++) {

if (i == j) {

A[i][j] = 2;

}

if (abs(i - j) == 1) {

A[i][j] = -1;

}

}

}

//开始迭代

timeval start, end;

gettimeofday(&start, NULL);

int count = 0;

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

count++;

//计算r^Tr,

double denom1 = INNER\_PRODUCT(r, r);

r = MATRIX\_VECTOR\_PRODUCT(A, x, b);

double num1 = INNER\_PRODUCT(r, r);

//退出循环条件

if (num1 < 0.000001) {

break;

}

double temp = num1 / denom1;

//计算方向向量d

for (int j = 0; j < N; j++) {

d[j] = -r[j] + temp \* d[j];

}

double num2 = INNER\_PRODUCT(d, r);

double denom2 = MATRIX\_PRODUCT(A, d);

//计算步长

double length = -num2 / denom2;

//修正x

for (int j = 0; j < N; j++) {

x[j] = x[j] + length \* d[j];

}

}

gettimeofday(&end, NULL);

double run\_time = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_usec - start.tv\_usec);

run\_time = run\_time / 1000000;

cout << "run\_time: " << run\_time << " s" << endl;

cout << "迭代次数: " << count << endl;

/\* cout<<"结果"<<endl;

displayb(x);\*/

return 0;

}

串行执行源代码包括：输出矩阵的函数、计算内积的函数、更新残差 r = A\*x-b函数、计算dtAd的函数，不断迭代进行残差的计算、方向向量的计算、搜索步长的计算、更新解向量，直到满足退出循环条件、停止迭代。

开发并行共轭梯度算法openmp版本：

源码如下：

#include<iostream>

#include<vector>

#include<cmath>

#include<stdio.h>

#include<omp.h>

#include<sys/time.h>

using namespace std;

int ThreadSize = 2;

int N =1000;

//初始化A为一个N\*N的矩阵的对称矩阵

vector<vector<double> > A(N,vector<double>(N,0));

//初始化数组b

vector<double> b(N,1);

//初始化残差r,结果x,计算方向向量d

vector<double> r(N,-1);

vector<double> d(N,0);

vector<double> x(N,0);

void displayA(vector<vector<double> > &a){

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

for(int j=0;j<N;j++){

cout<<a[i][j]<<" ";

}

cout<<endl;

}

}

void displayb(vector<double> &b){

for(int i=0;i<b.size();i++){

cout<<b[i]<<" ";

}

cout<<endl;

}

//计算内积，也就是模的平方

double INNER\_PRODUCT(vector<double> &a,vector<double>&b){

double res = 0;

#pragma omp parallel for reduction(+:res) num\_threads(ThreadSize)

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

res+=a[i]\*b[i];

}

return res;

}

//更新残差 r = A\*x-b

vector<double>& MATRIX\_VECTOR\_PRODUCT(vector<vector<double> > &a,vector<double> &x,vector<double>& b){

double temp = 0;

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

temp = 0;

#pragma omp parallel for reduction(+:temp) num\_threads(ThreadSize)

for(int j=0;j<N;j++){

temp += a[i][j]\*x[j];

}

r[i] = temp - b[i];

}

return r;

}

//计算dtAd

double MATRIX\_PRODUCT(vector<vector<double> >&a,vector<double> &d){

double res = 0;

double temp = 0;

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

temp = 0;

#pragma omp parallel for reduction(+:temp) num\_threads(ThreadSize)

for(int j = 0;j<N;j++){

temp += d[j]\*A[i][j];

}

res += temp\*d[i];

}

return res;

}

int main(int argv,char \*\*args){

string s = args[1];

ThreadSize = stoi(s);

timeval start,end;

gettimeofday(&start,NULL);

//初始化A

#pragma omp parallel for num\_threads(ThreadSize)

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

for(int j =0;j<N;j++){

if(i==j){

A[i][j] = 2;

}

if(abs(i-j) == 1){

A[i][j] = -1;

}

}

}

//displayA(A);

//displayb(b);timeval

//开始迭代

int count = 0;

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

count++;

//计算r^Tr,

double denom1 = INNER\_PRODUCT(r,r);

r = MATRIX\_VECTOR\_PRODUCT(A,x,b);

double num1 = INNER\_PRODUCT(r,r);

if(num1 < 0.000001){

break;

}

double temp = num1/denom1;

//计算方向向量d

#pragma omp parallel for num\_threads(ThreadSize)

for(int j = 0;j<N;j++){

d[j] = -r[j]+temp\*d[j];

}

double num2 = INNER\_PRODUCT(d,r);

double denom2 = MATRIX\_PRODUCT(A,d);

//计算步长

double length = -num2/denom2;

//修正x

#pragma omp parallel for num\_threads(ThreadSize)

for(int j=0;j<N;j++){

x[j] = x[j]+ length\*d[j];

}

}

gettimeofday(&end,NULL);

double run\_time = (end.tv\_sec-start.tv\_sec)\*1000000+(end.tv\_usec-start.tv\_usec);

run\_time = run\_time/1000000;

cout<<"ThreadSize: "<<ThreadSize<<endl;

cout<<"run\_time: "<<run\_time<<" s"<<endl;

cout<<"迭代次数: "<<count<<endl;

//cout<<"ThreadSize: "<<ThreadSize<<endl;

//displayb(x);

return 0;

}

openmp源代码包括：输出矩阵的函数、计算内积的函数、更新残差 r = A\*x-b函数、计算dtAd的函数，不断迭代进行残差的计算、方向向量的计算、搜索步长的计算、更新解向量，直到满足条件退出循环、停止迭代。

#pragma omp for

通过上述指令，就可以对循环进行并行化，来达到加速程序的目的，同时有些地方使用reduction子句去避免多个线程同时操作同一个数据的情况。

#### 项目创新点

开发矩阵运算代码来入门（包括dpcpp矩阵加法运算、openmp矩阵乘法运算）

共轭梯度算法的实现（包括串行执行、openmp并行执行加速两个版本）

学习使用reduction子句解决了并发程序的数据竞争问题，避免多个线程同时操作同一个数据的情况：

举例从并发求和开始：

任务是两个线程同时对一个变量 data 进行 ++操作，执行 10000 次，代码如下：

int main() {

#pragma omp parallel num\_threads(2) // 使用两个线程同时执行上面的代码块

{

for(int i = 0; i < 10000; i++) {

data++;

usleep(10);

}

// omp\_get\_thread\_num 函数返回线程的 id 号 这个数据从 0 开始，0, 1, 2, 3, 4, ...

printf("data = %d tid = %d\n", data, omp\_get\_thread\_num());

}

printf("In main function data = %d\n", data);

return 0;

}

在上面的代码当中，我们开启了两个线程并且同时执行pragma 下面的代码块，但是上面的程序有一个问题，就是两个线程可能同时执行 data++ 操作，但是同时执行这个操作的话，就存在并发程序的数据竞争问题，在 OpenMP 当中默认的数据使用方式就是线程之间是共享的比如下面的执行过程：

首先线程 1 和线程 2 将 data 加载到缓存当中，当前的两个线程得到的 data 的值都是 0 。

线程 1 和线程 2 对 data 进行 ++ 操作，现在两个线程的 data 的值都是 1。

线程 1 将 data 的值写回到主存当中，那么主存当中的数据的值就等于 1 。

线程 2 将 data 的值写回到主存当中，那么主存当中的数据的值也等于 1 。

但是上面的执行过程是存在问题的，因为我们期望的是主存当中的 data 的值等于 2，因此上面的代码是存在错误的。

联想在课堂上估计π的值的程序中，我们使用数组巧妙解决并发程序当中的数据竞争问题。使用了一个函数 omp\_get\_thread\_num 这个函数可以返回线程的 id 号，我们可以根据这个 id 来解决。

static int data;

static int tarr[2];

int main() {

#pragma omp parallel num\_threads(2)

{

int tid = omp\_get\_thread\_num();

for(int i = 0; i < 10000; i++) {

tarr[tid]++;

usleep(10);

}

printf("tarr[%d] = %d tid = %d\n", tid, tarr[tid], tid);

}

data = tarr[0] + tarr[1];

printf("In main function data = %d\n", data);

return 0;

}

在上面的程序当中我们额外的使用了一个数组 tarr 用于保存线程的本地的和，然后在最后在主线程里面讲线程本地得到的和相加起来，这样的话得到的结果就是正确的了

tarr[1] = 10000 tid = 1

tarr[0] = 10000 tid = 0

In main function data = 20000

需要知道的是，只有当并行域当中所有的线程都执行完成之后，主线程才会继续执行并行域后面的代码，因此主线程在执行代码

data = tarr[0] + tarr[1];

printf("In main function data = %d\n", data);

之前，OpenMP 中并行域中的代码全部执行完成，因此上面的代码执行的时候数组 tarr 中的结果已经计算出来了，因此上面的代码最终的执行结果是 20000

除了上面的方法避免多个线程同时操作同一个数据（数据竞争）的情况，在开发并行共轭梯度算法时我们使用 了reduction 子句去解决这个问题：

#include <stdio.h>

#include <omp.h>

#include <unistd.h>

static int data;

int main() {

#pragma omp parallel num\_threads(2) reduction(+:data)

{

for(int i = 0; i < 10000; i++) {

data++;

usleep(10);

}

printf("data = %d tid = %d\n", data, omp\_get\_thread\_num());

}

printf("In main function data = %d\n", data);

return 0;

}

在上面的程序当中我们使用了一个子句 reduction(+:data) 在每个线程里面对变量 data 进行拷贝，然后在线程当中使用这个拷贝的变量，这样的话就不存在数据竞争了，因为每个线程使用的 data 是不一样的，在 reduction 当中还有一个加号+，这个加号表示如何进行规约操作，所谓规约操作简单说来就是多个数据逐步进行操作最终得到一个不能够在进行规约的数据。

例如在上面的程序当中我们的规约操作是 + ，因此需要将线程 1 和线程 2 的数据进行 + 操作，即线程 1 的 data 加上 线程 2 的 data 值，然后将得到的结果赋值给全局变量 data，这样的话我们最终得到的结果就是正确的。

如果有 4 个线程的话，那么就有 4 个线程本地的 data（每个线程一个 data）。那么规约（reduction）操作的结果等于：

(((data1 + data2) + data3) + data4)

其中 datai 表示第 i 个线程的得到的 data

redcution 子句的语法格式如下：

reduction(操作符:变量)

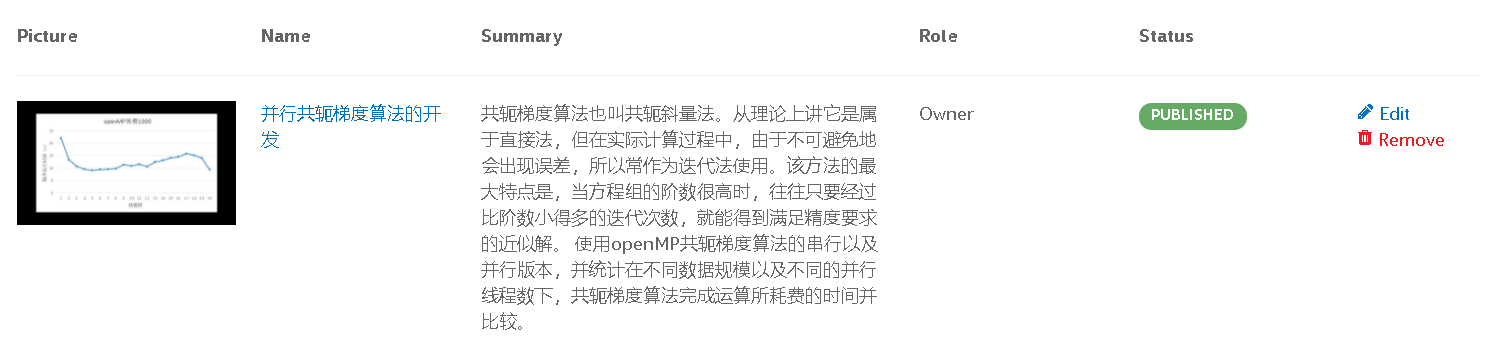
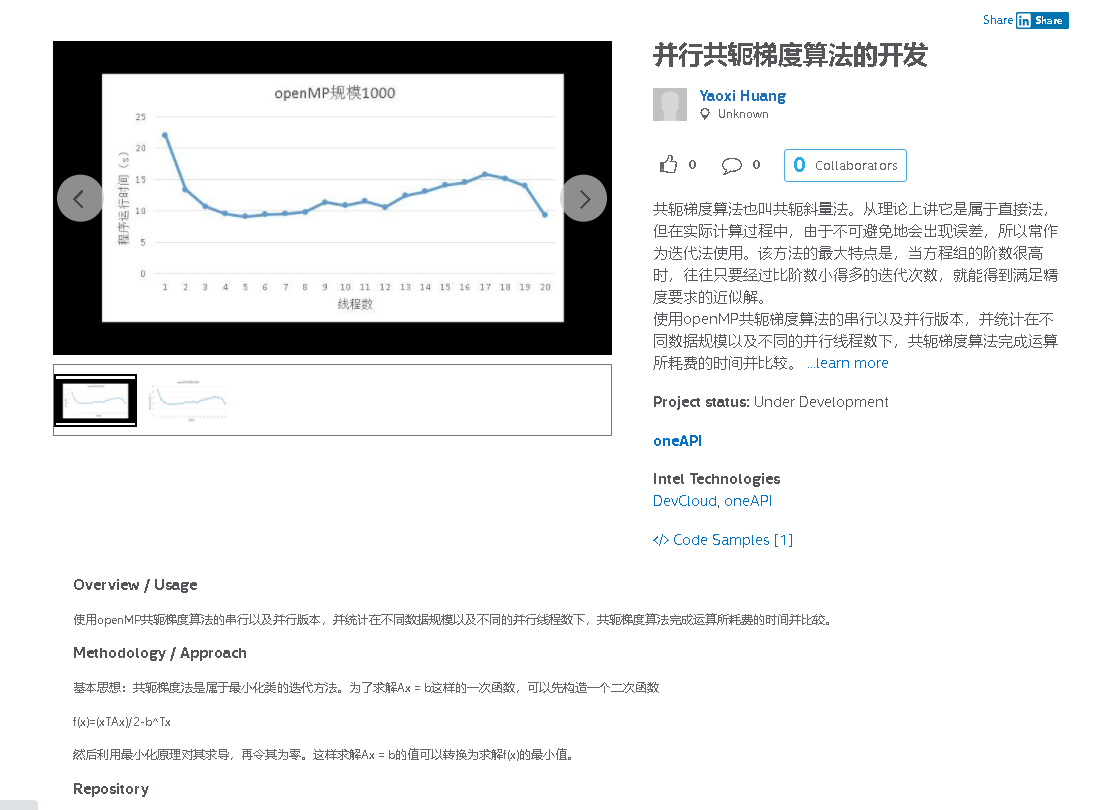
我们使用 reduction 子句的时候线程使用的是与外部变量同名的变量，同名变量的初始值具体设置规则，比如加法运算初始符设为0、乘法运算初始符设置为1

我们在写多线程程序的时候可能会存在这种需求，每个线程都会得到一个数据的结果，然后在最后需要将每个线程得到的数据进行求和，相乘，或者逻辑操作等等，在这种情况下我们可以使用 reduction 子句进行操作。

#### 项目成果

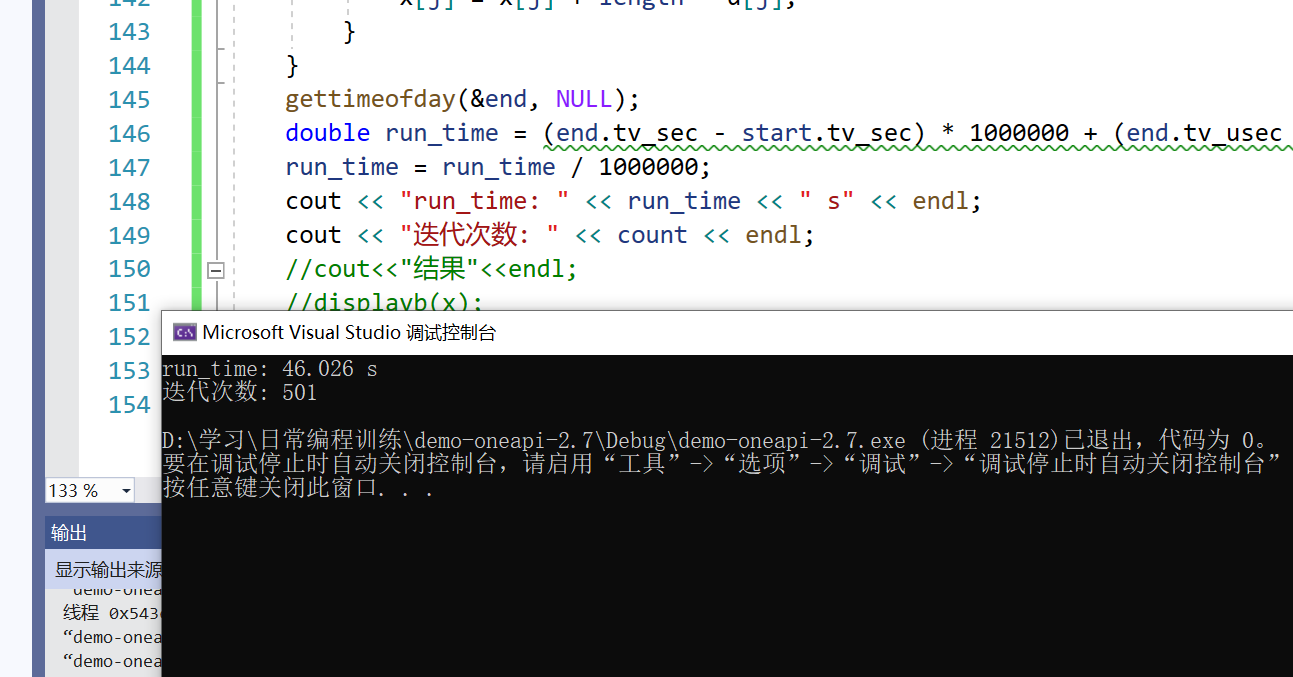
DevMesh地址：

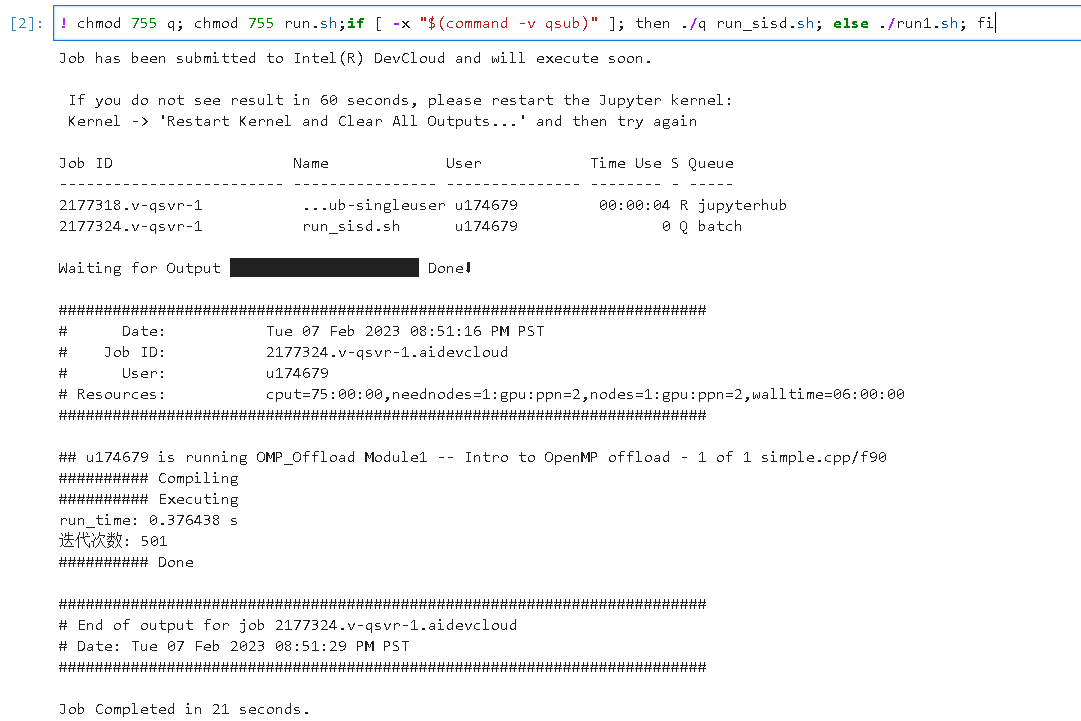
https://devmesh.intel.com/projects/1718ee

DevMesh截图：

项目的主要结果或截图展示：

串行源代码运行截图（本地运行、devcloud）：





Openmp源代码运行截图：



不同线程数以及不同规模下的统计数据：

|  |  |  |
| --- | --- | --- |
| **规模**  **线程数** | **openMP（1000）** | **openMP（2000）** |
| **1** | **22.027** | **156.465** |
| **2** | **13.368** | **80.335** |
| **3** | **10.714** | **64.001** |
| **4** | **9.559** | **55.784** |
| **5** | **9.119** | **54.979** |
| **6** | **9.410** | **62.447** |
| **7** | **9.569** | **49.246** |
| **8** | **9.819** | **64.584** |
| **9** | **11.325** | **75.283** |
| **10** | **10.852** | **83.081** |
| **11** | **11.523** | **69.815** |
| **12** | **10.627** | **96.143** |
| **13** | **12.445** | **62.664** |
| **14** | **13.074** | **70.822** |
| **15** | **14.076** | **92.931** |
| **16** | **14.570** | **93.755** |
| **17** | **15.849** | **88.482** |
| **18** | **15.183** | **89.584** |
| **19** | **14.006** | **83.105** |
| **20** | **9.325** | **86.217** |

通过上图统计中可以看到，使用openMP实现并行共轭梯度算法，在线程开启的较少时，程序执行时间会下降，但是当程序线程更多时，程序的执行时间反而会略有上升，说明此时通信的开销更大。