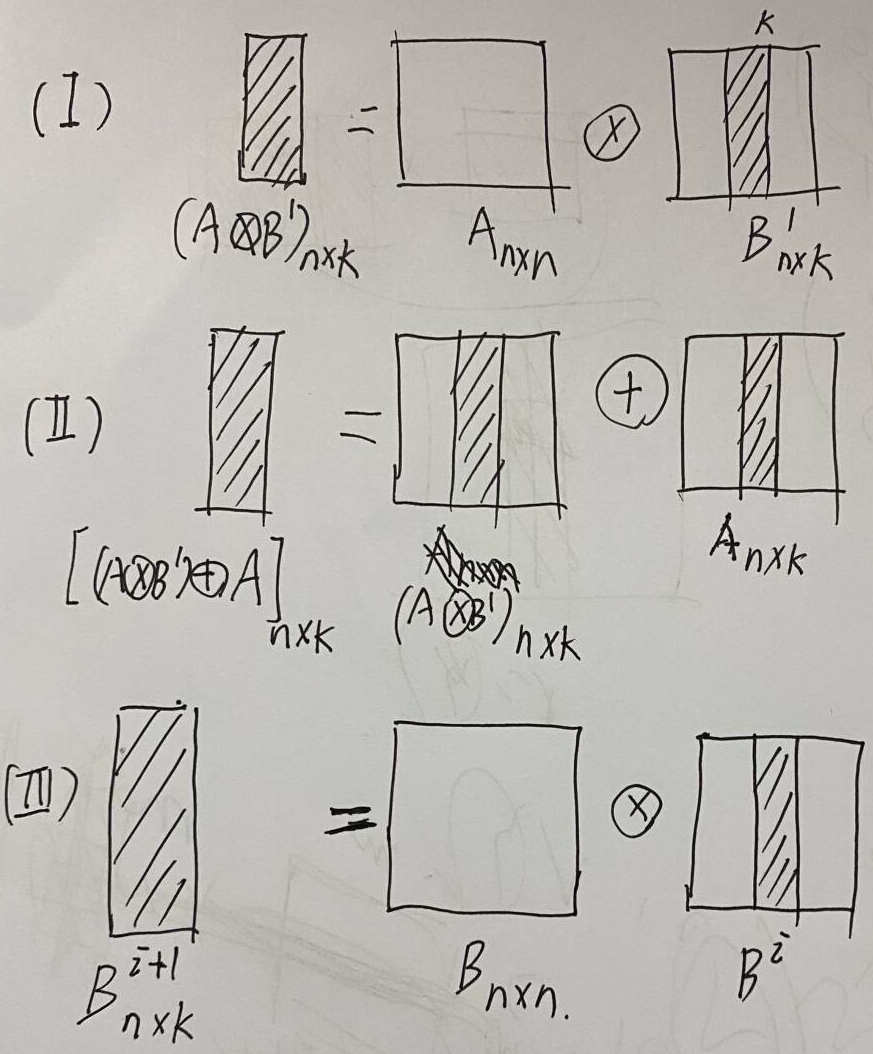
# 实验思路

## 并行方式

公式C由很多项模2矩阵加法组成，在每一项中，B的计算都依赖于前一项，因此不能根据加法项来划分进程。如果将A分块划分，每个进程都需要计算完整的B。所以，应对B进行划分，由于B在乘法右侧，因此对B按列进行划分。

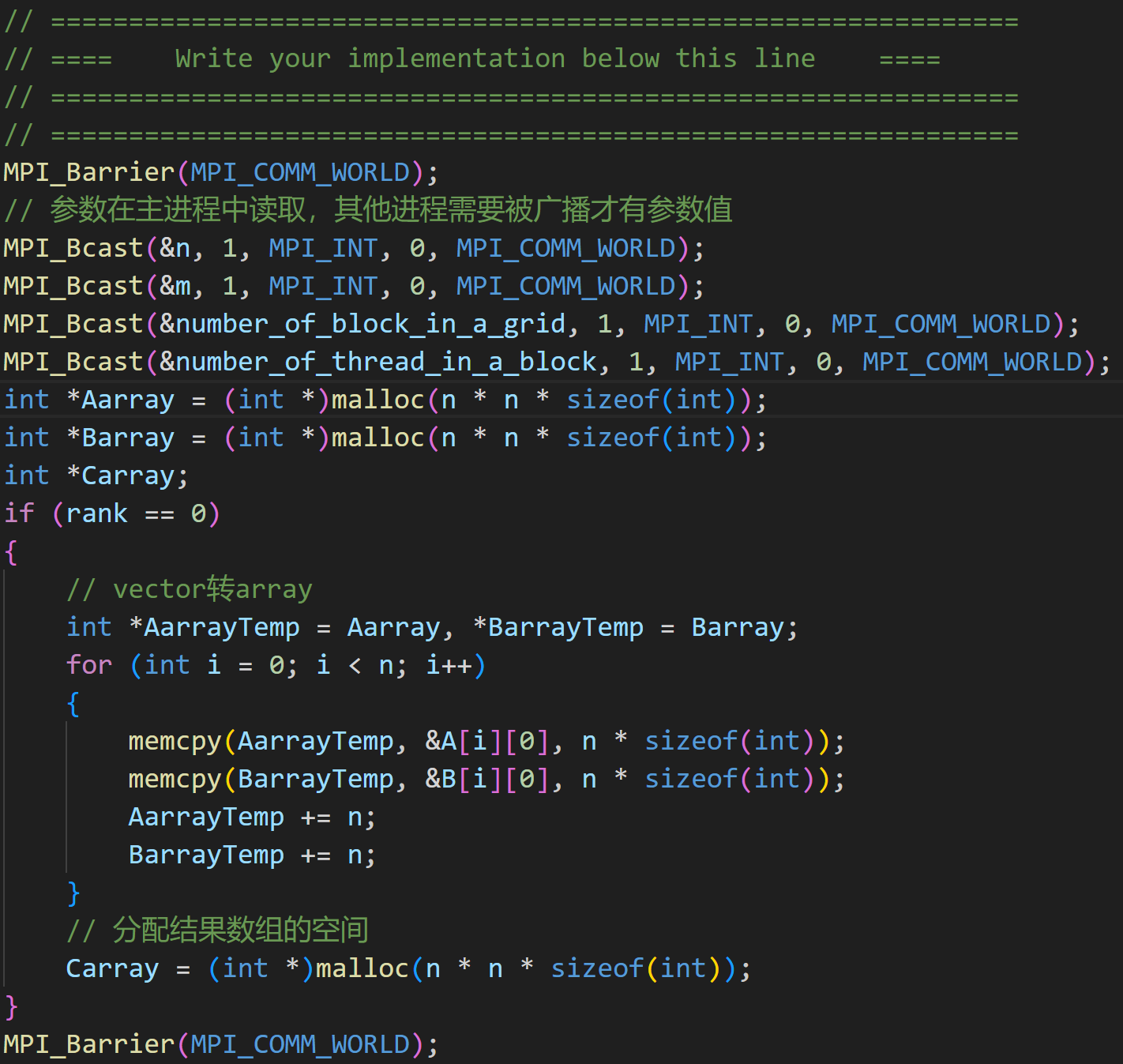
根据定义，又因为矩阵乘法满足交换律，所以，因此B的迭代可以使用列分块来进行。计算C的过程中，一共涉及到3种运算，如下图所示，每种运算都可以对运算符右侧进行列分块，因此我们可以对列进行划分并使用MPI来加速。



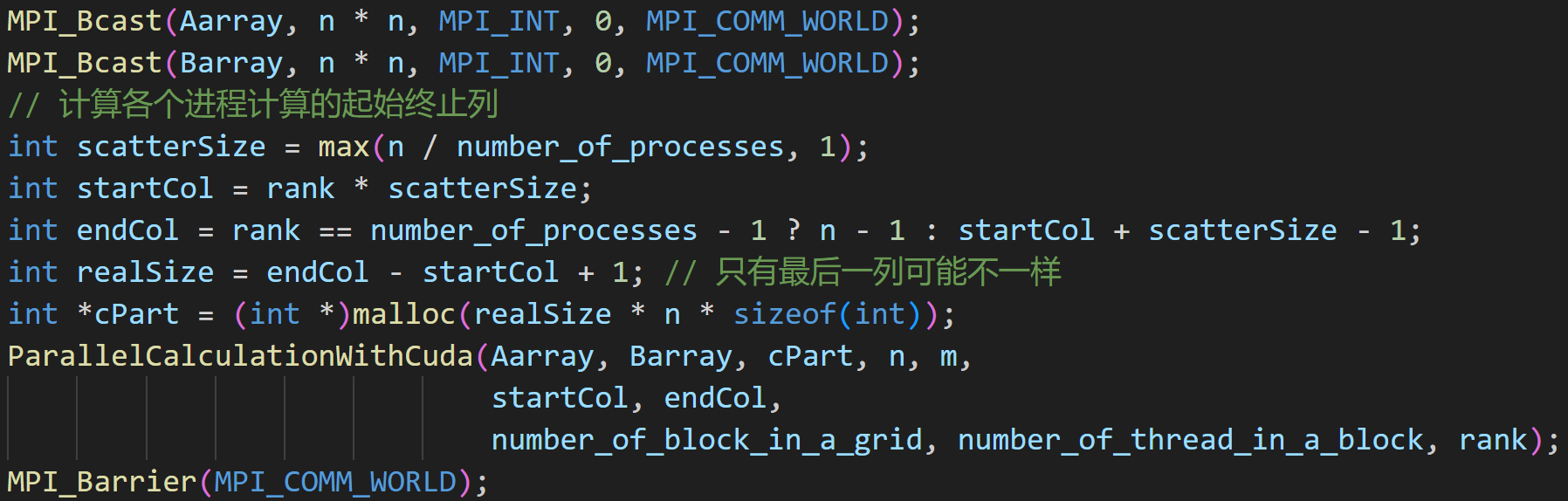
每个进程执行的每次运算都是模2的矩阵加法或乘法，因此可以使用GPU进行并行处理。程序读入的是一维参数，因此可以对运算符左侧矩阵的行进行划分。

## MPI并行

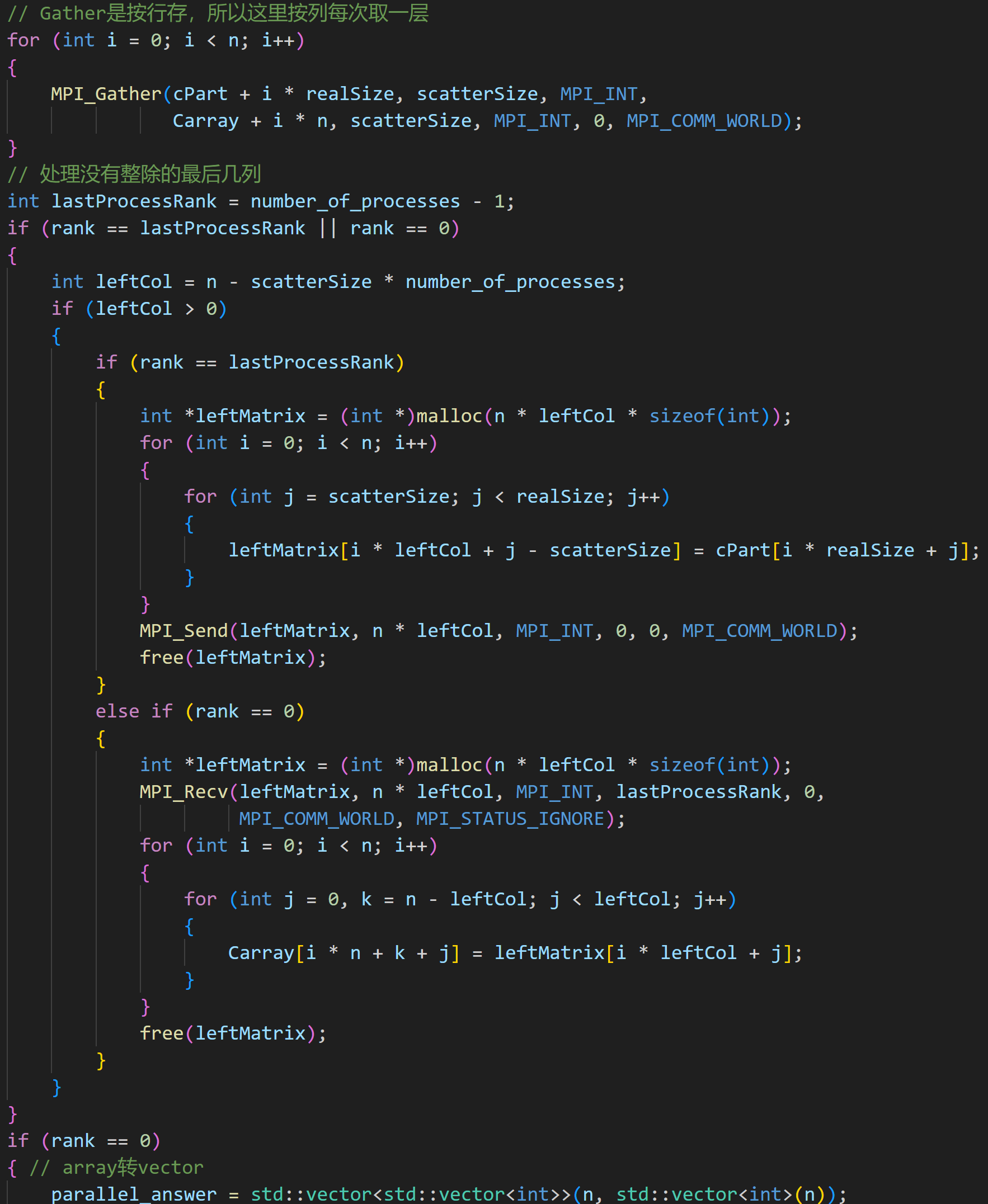
程序参数的读取以及矩阵数据的获取已经由原始代码完成。对于代码实现部分，首先让其他进程等待主进程读取数据，然后广播矩阵大小、迭代次数、每个grid的block数和每个block的thread数。vector无法直接使用MPI来发送，因此将其转化为普通数组。另外，主进程需要分配结果数组的空间，并且其他进程也需要等待此部分的数据处理结束。



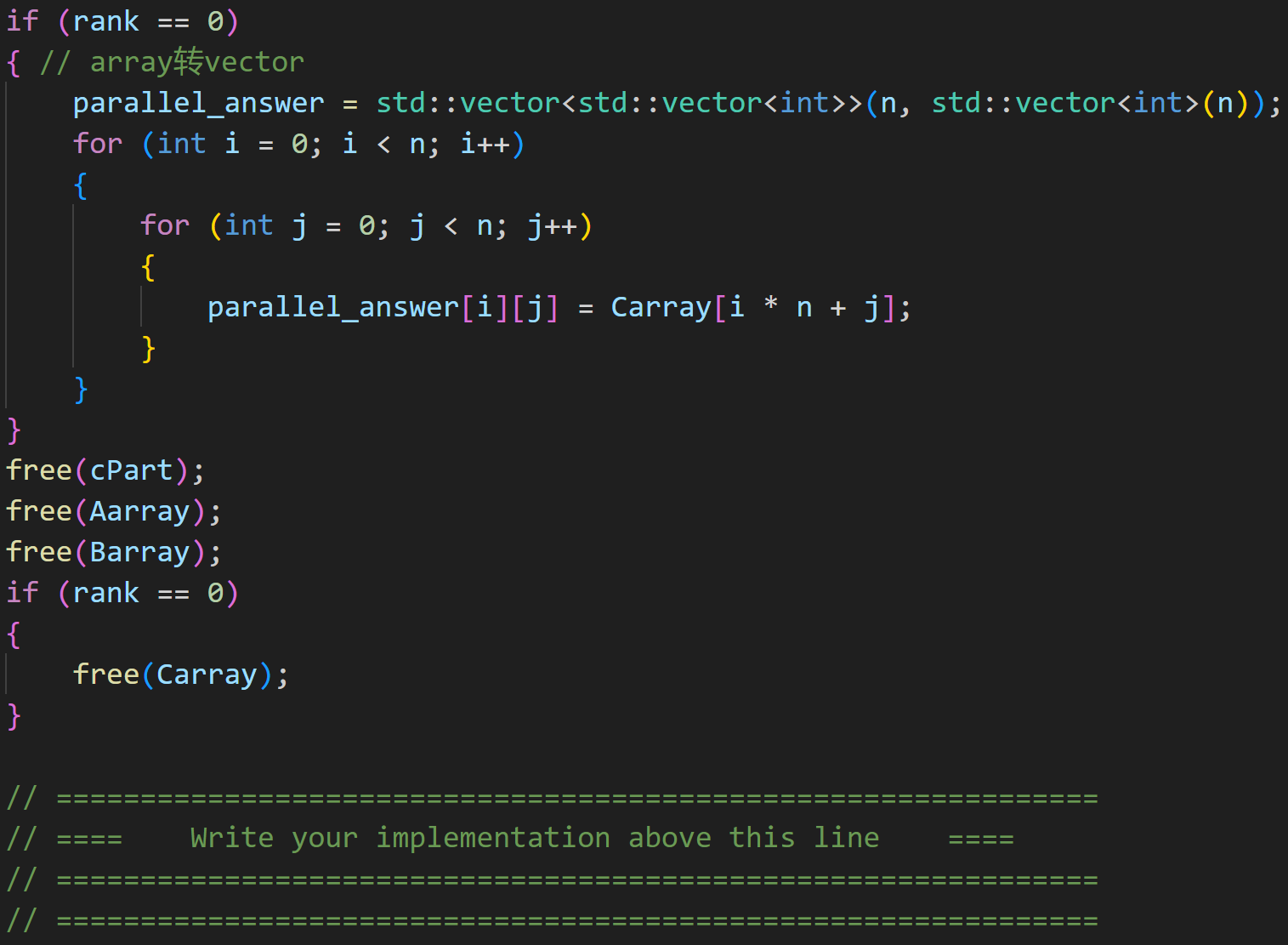
接下来，每个进程广播在计算的时候都需要用到矩阵A和B，计算在划分之后各个进程的列分块区间。当矩阵的列长度小于进程总数时，设置区间大小为1。进程数不一定总能整除矩阵大小，因此最后一个进程的终止列需要特殊处理，最后一个进程的处理区间可能比其他进程要大一些。然后，每个进程都调用计算函数，并等待其他进程计算完成。



MPI\_Gather将每个进程的数据进行聚集，但是我们最终得到的数据是按列存储的，直接使用这个函数会出现问题。这里调用MPI\_Gather函数n次，每次聚集同一行不同进程的计算结果。最后一个进程计算的列长度可能比其他进程多，因此这里必须访问cPart[realSize][\*]而不能是cPart[scatterSize][\*]。在聚集的时候，我们使用的是固定大小，因此需要将最后一个进程多出来的数据发送回主进程。这里对最后一个进程和主进程进行特殊判断，使用MPI\_Send和MPI\_Recv发送和接收数据。



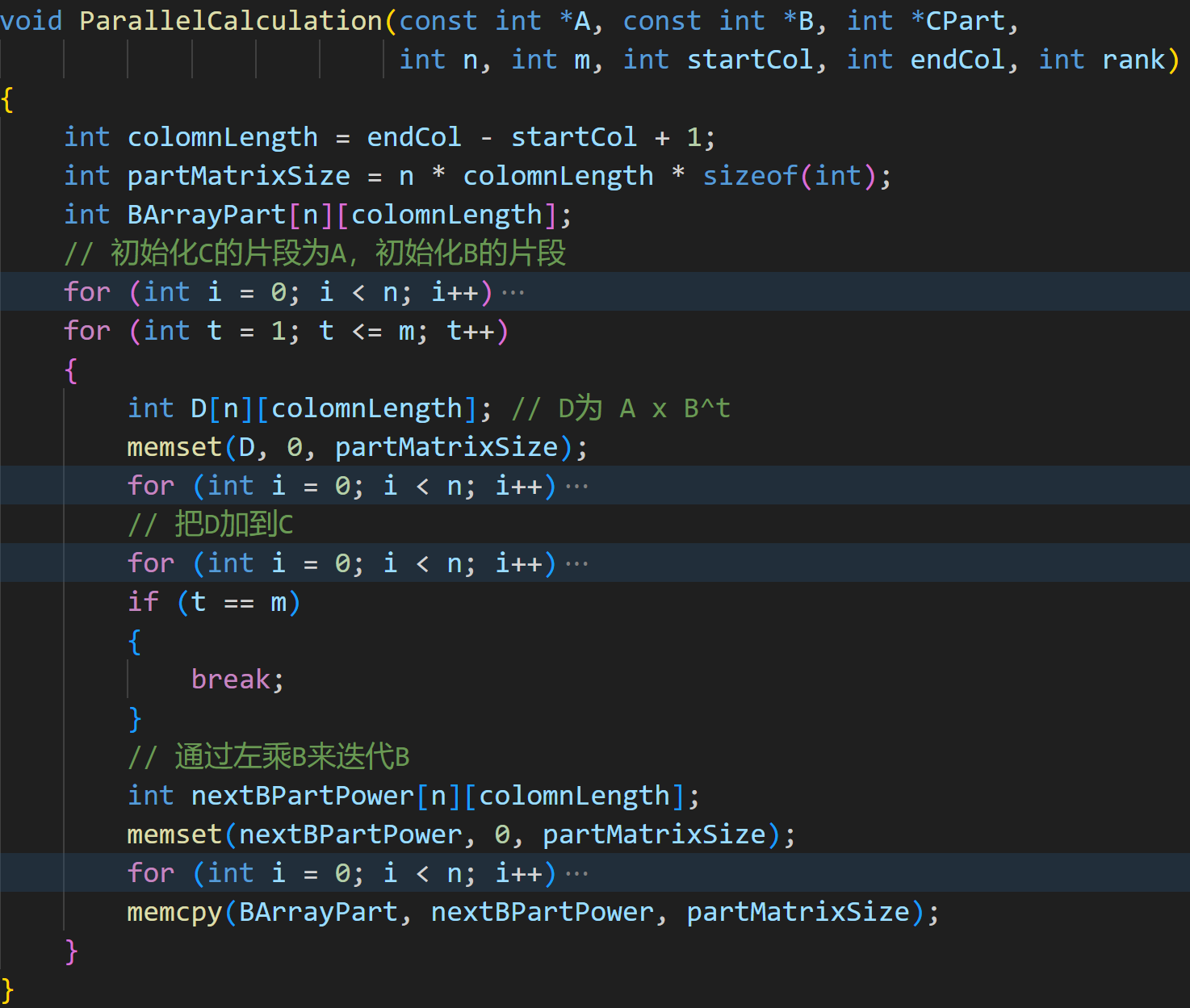
最后，在主进程中将计算后的数组转化为vector，并释放计算过程中malloc的内存，接着执行原始文件中的其他代码。



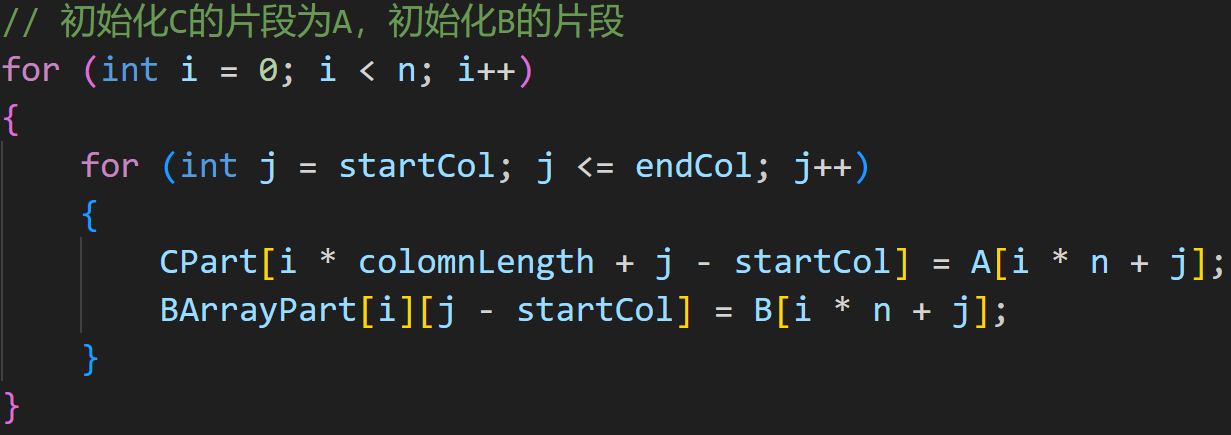
## CPU计算

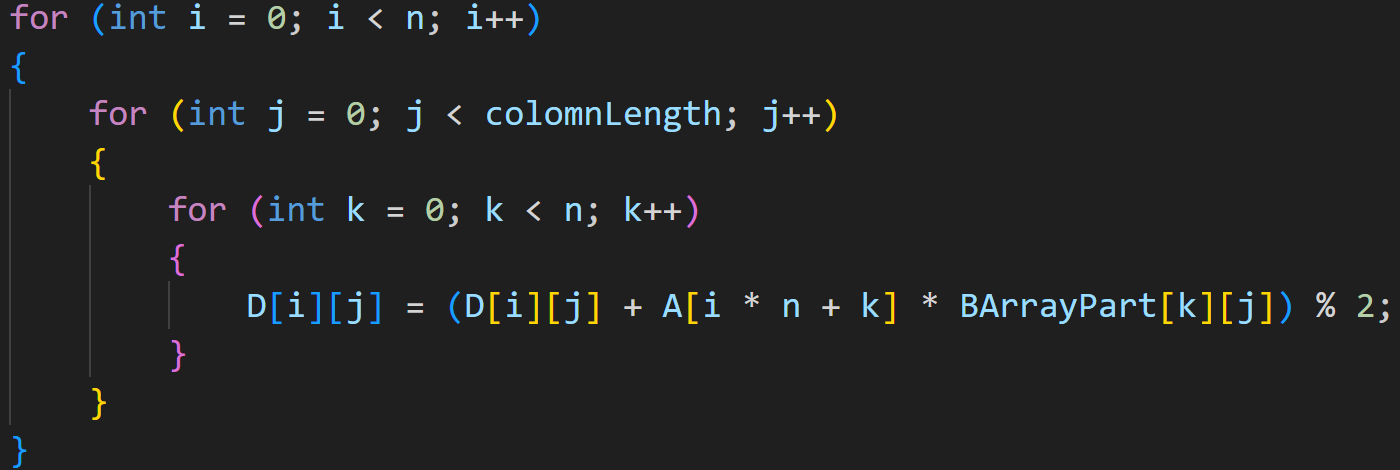
在编写计算部分的代码时，我首先编写的是CPU计算代码，完全测试通过后才编写GPU计算代码。

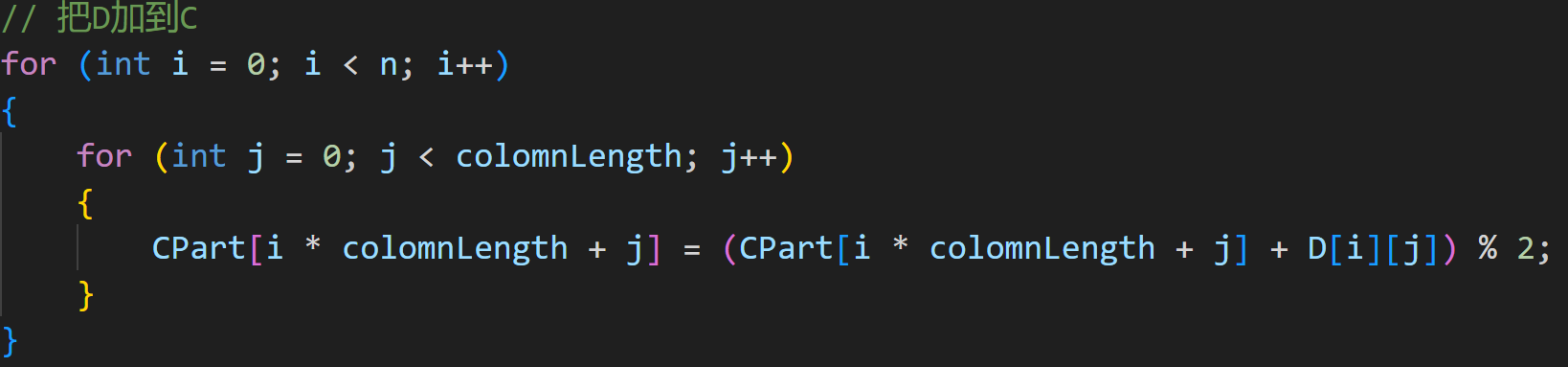
首先，从A中初始化C的片段，并从B中初始化B的片段，片段的列长度就是每个进程需要计算的列长度。然后，从1迭代到m，计算，，迭代。这里使用了memset和memcpy对矩阵进行快速赋值。

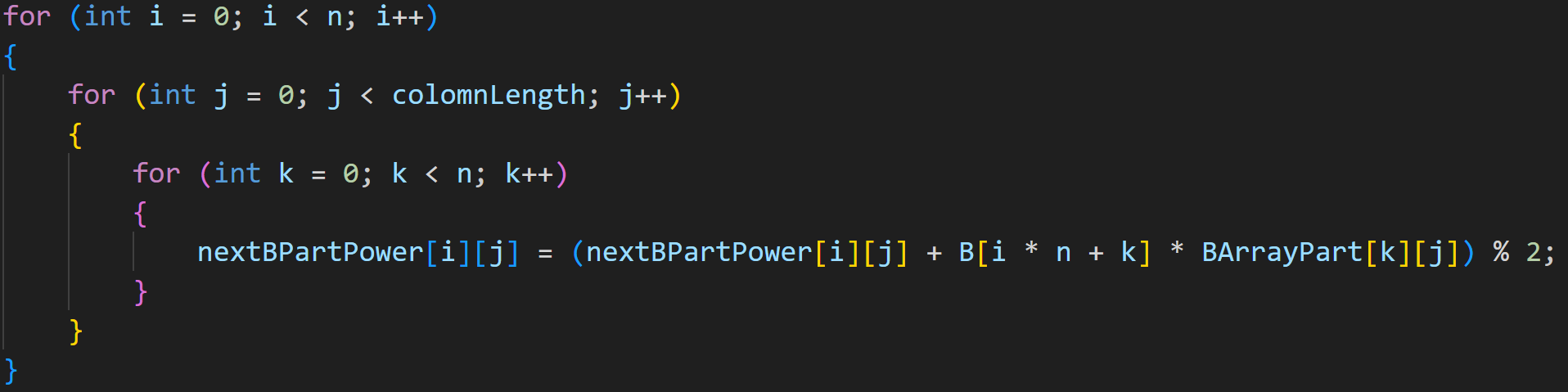


循环内的细节如下面四部分代码所示。



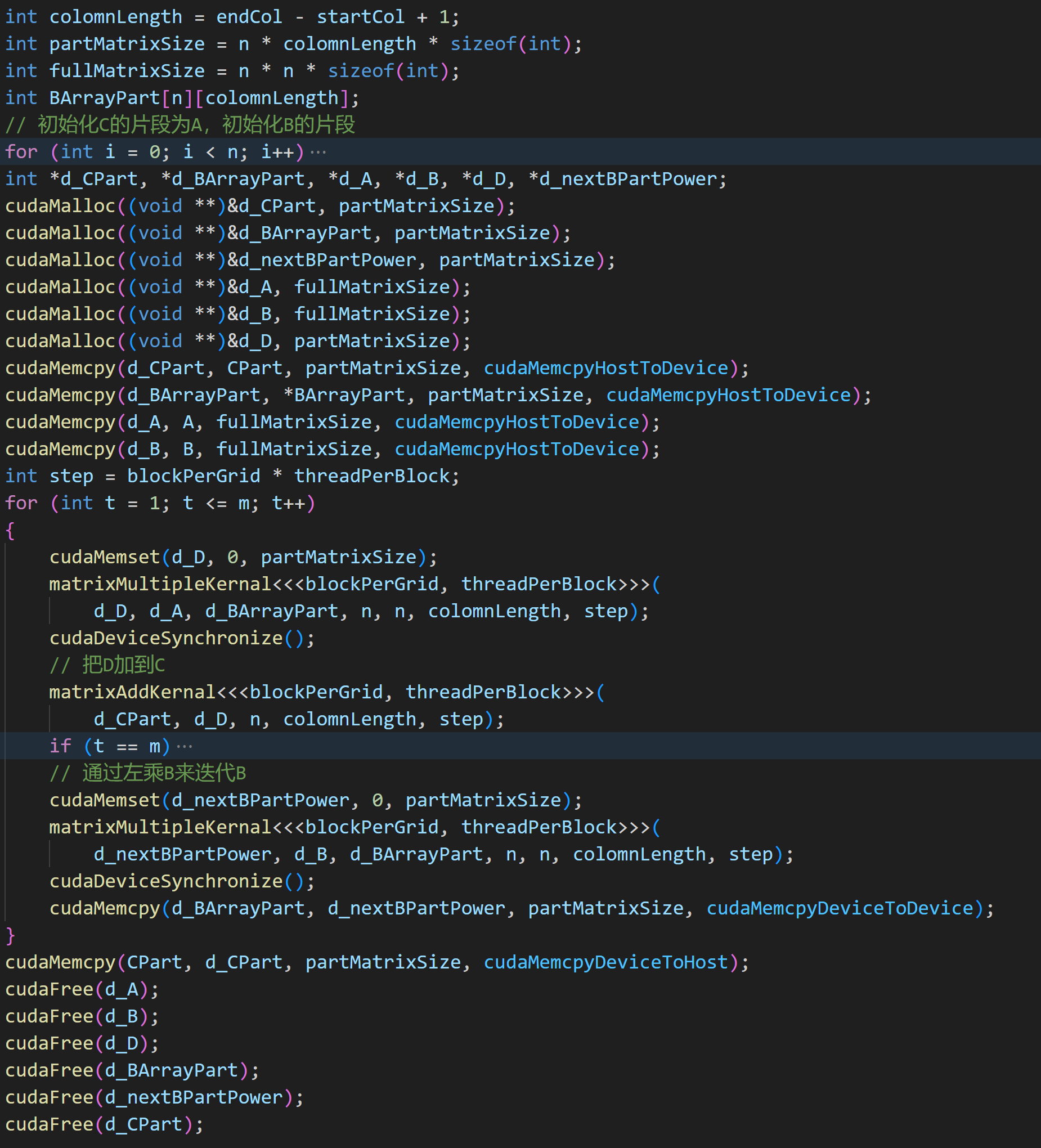




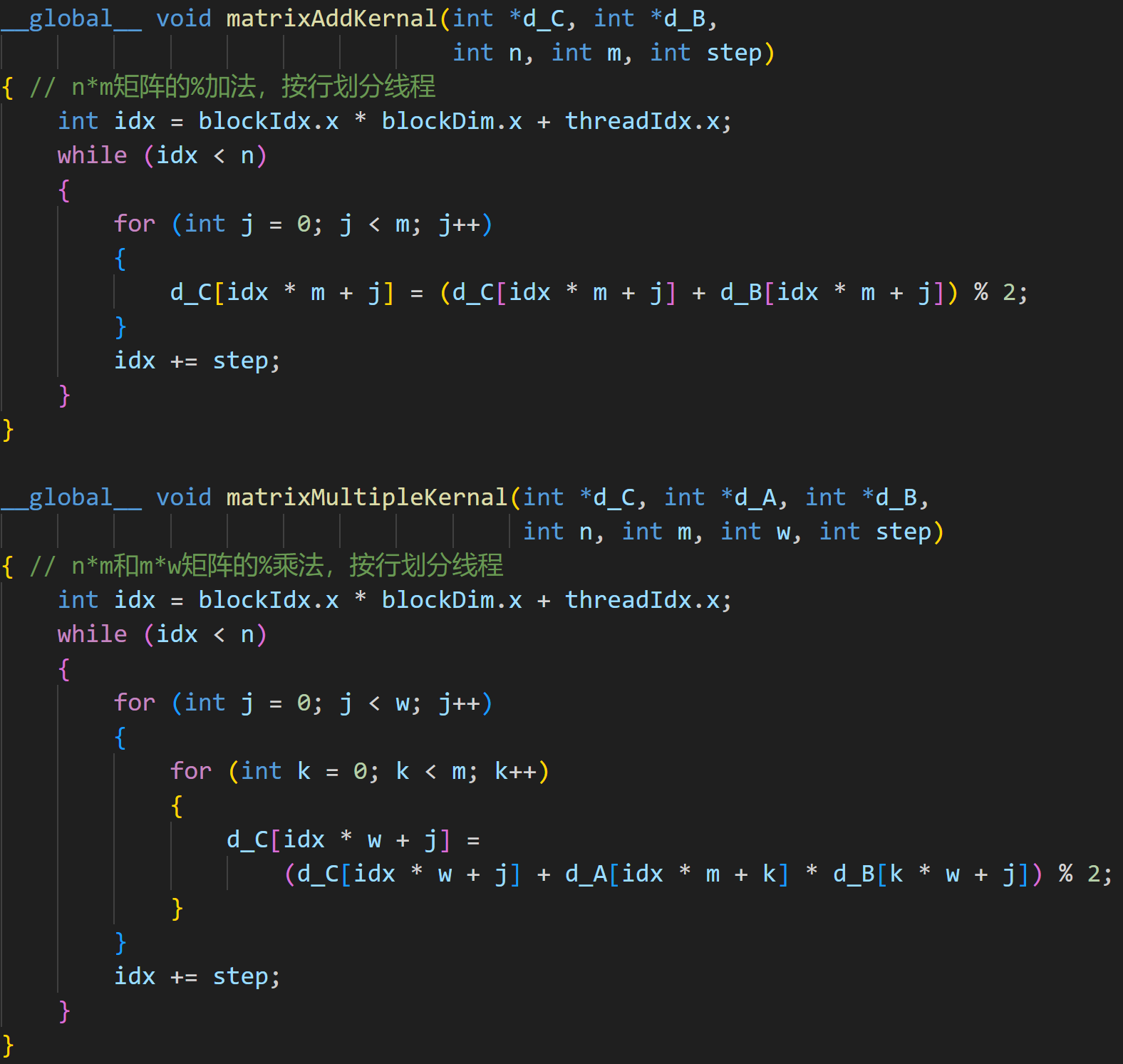


## GPU计算

在CPU版MPI并行计算代码的基础上，用cuda来进行GPU加速。首先，使用cudaMalloc在GPU中分配内存，对于每个进程来说，将矩阵A、矩阵B、列分块C和列分块B的值拷贝到device中。然后，将原来的for循环更改为调用kernal函数，函数调用后需要进行数据同步。最后，将GPU中的计算结果拷贝回host的内存中，并释放在GPU中申请的内存。



kernel函数分别是矩阵的模2乘法和加法。这里使用的是1维的下标计算，根据线程块id和线程id计算行的下标，然后对这一行的每个元素进行计算。计算后把下标增加gridDim\*blockDim，在循环中继续计算，防止线程数多于或少于矩阵的行数。

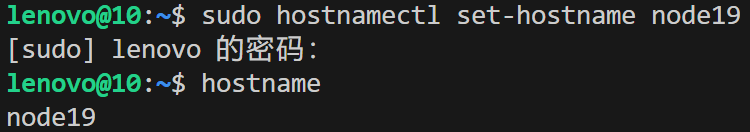


# 实验结果

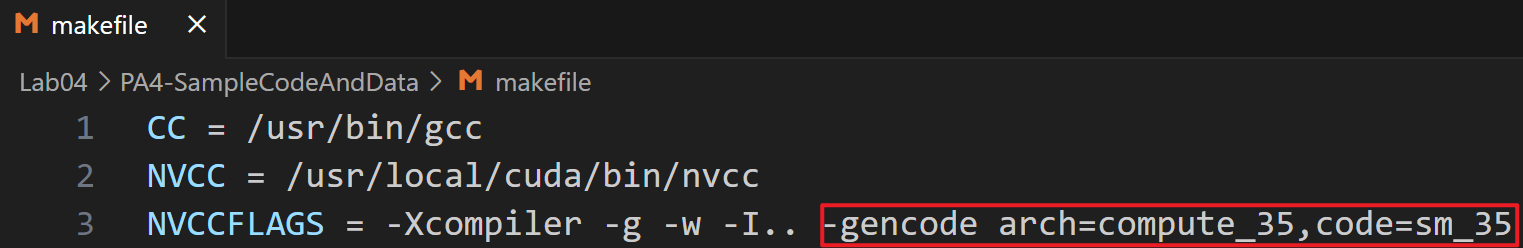
## 环境配置

在实验教室中使用了两台机器进行互联，设置好IP，ssh-copy-id发送公钥，测试ssh的连接。

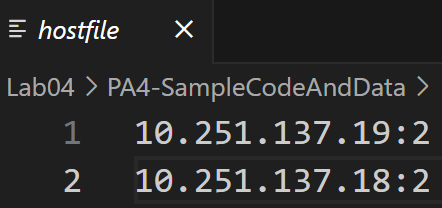
对于两台机器，可以设置hostname来区分主机。设置指令如下，重启终端即可看到效果。



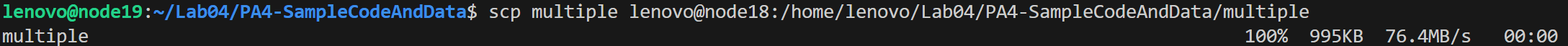
makefile中需要加入设置版本的参数，否则可能执行错误。



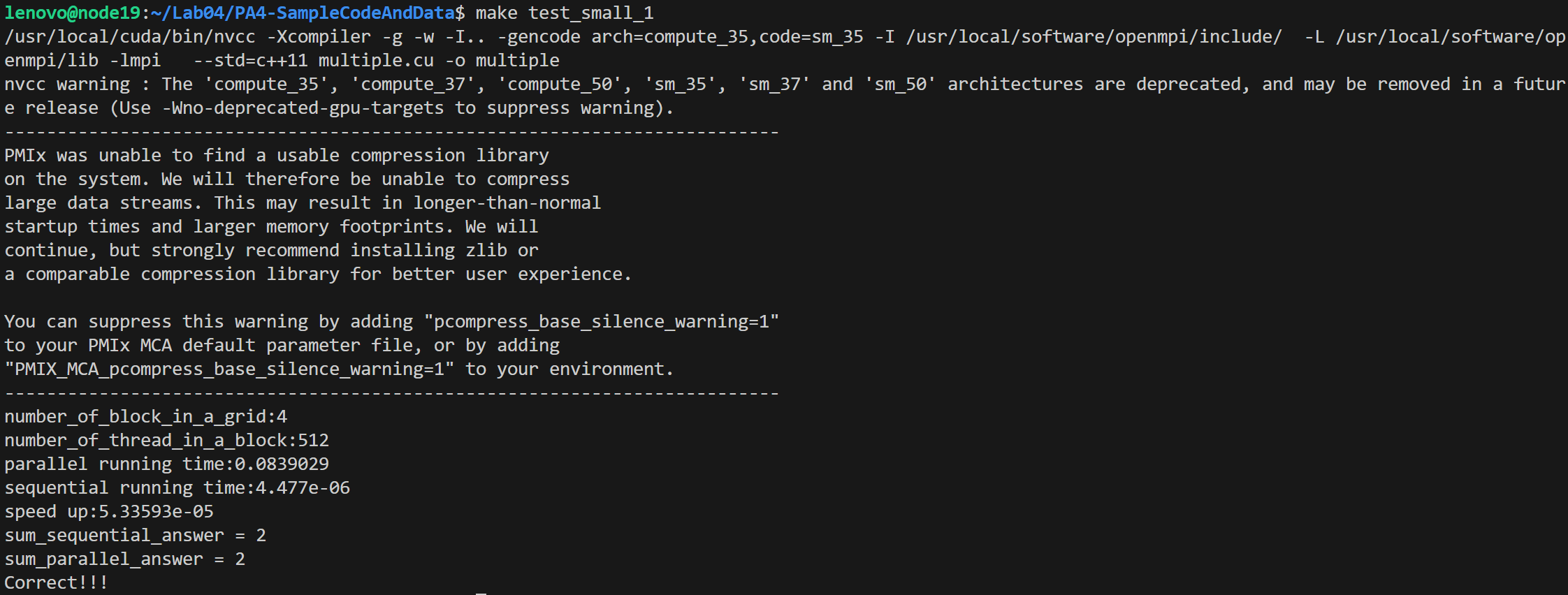
Hostfile文件内容如下，每个节点设置为最多2个进程。



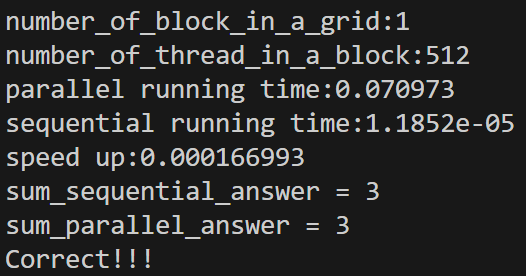
使用make命令编译程序，然后使用scp指令将程序发送给另一个节点，具体指令如下图所示。



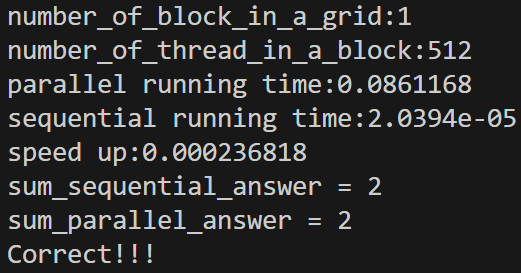
执行测试用例，可以看到程序正确计算出了结果，下图为test\_small\_1的计算结果。



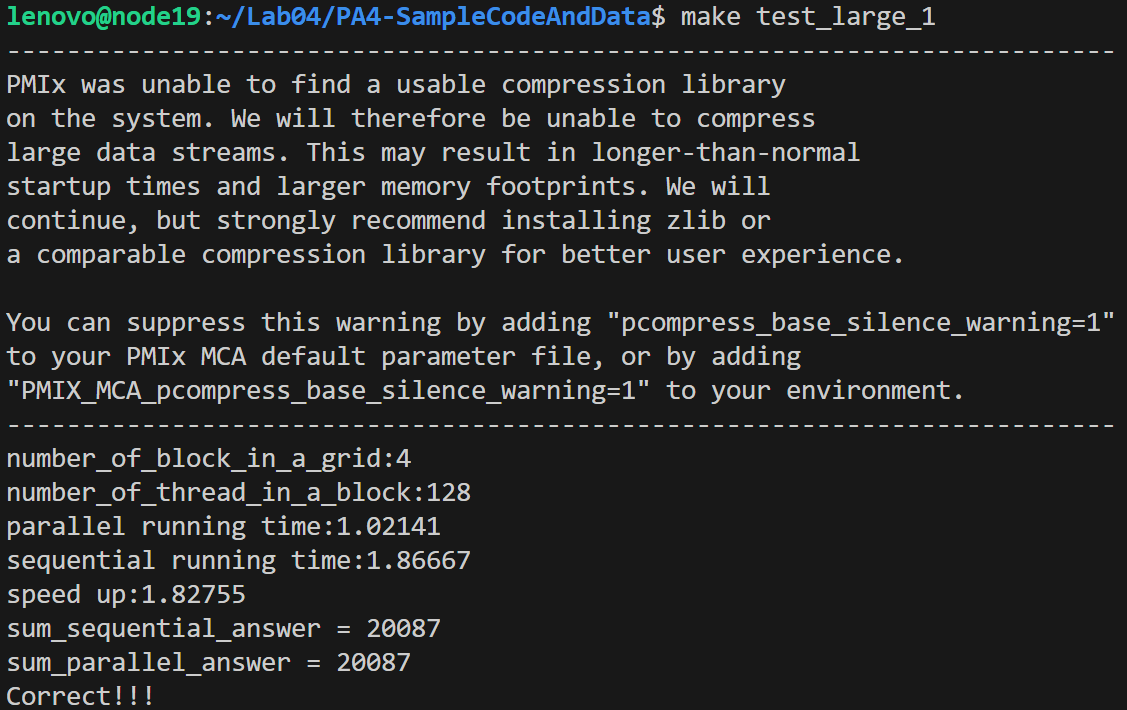
下图为test\_small\_2的计算结果。



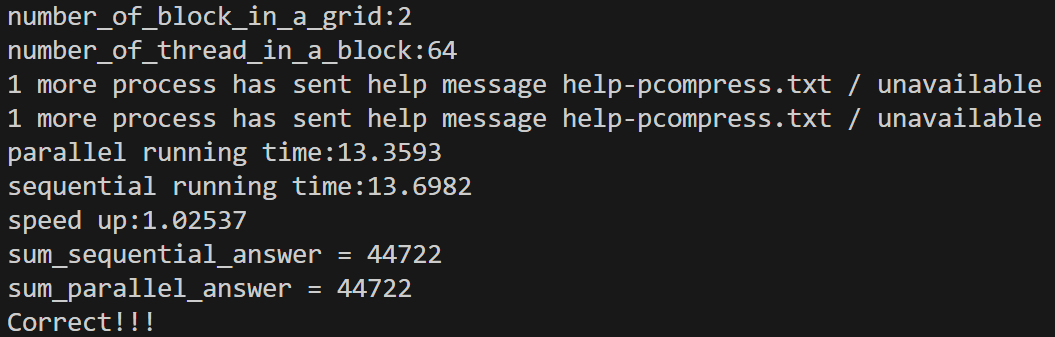
下图为test\_small\_3的计算结果。



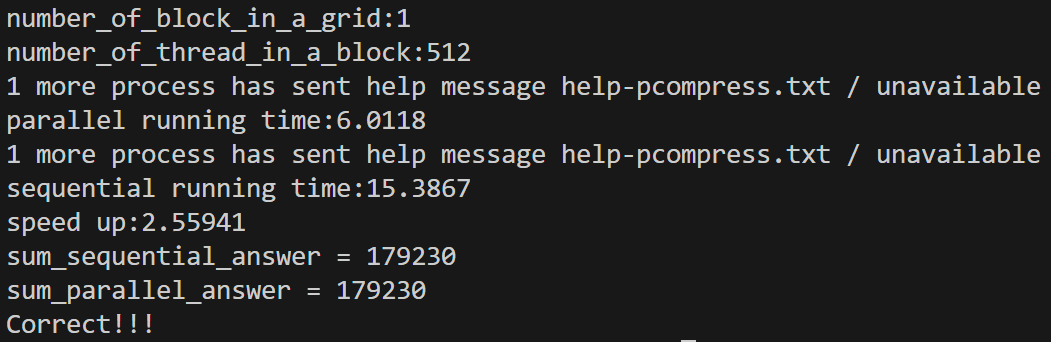
在较大的数据集上执行程序，程序计算的结果也是正确的，下图为test\_large\_1的计算结果。



下图为test\_large\_2的计算结果。



下图为test\_large\_3的计算结果。



可以看到在较大的数据集下，程序分别取得了1.83、1.03和2.56倍的速度提升。