并行程序设计实验

实验 1 矩阵乘法运算

联系方式：NJU-TJL@outlook.com

目录

实验 1 矩阵乘法运算

一、 实验设计

1. 串行代码

通过 main 函数的参数获取到给定的矩阵阶数 n（未给定时默认值为 1000）。然后，在 init()函数中，初始化 A、B、C 三个 n 阶矩阵，主要是使用 malloc 函数分配空间以及初始化元素值为实验讲义给定的形式（如下所示）。

$$A=\begin{Bmatrix} 1 & 2 & 3 & 4 & \dots & n \\ 2 & 3 & 4 & 5 & \dots & n+1 \\ 3 & 4 & 5 & 6 & \dots & n+2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ n & n+1 & n+2 & n+3 & \dots & 2n-1 \end{Bmatrix} \quad B=\begin{Bmatrix} 1 & \dots & 1 \\ \vdots & \vdots & \vdots \\ 1 & \dots & 1 \end{Bmatrix}$$

之后，按照矩阵乘法公式（如下所示），使用三层 for 循环即可完成计算。

$$C[i,j]=\sum_{i=1}^{n} A[i,k]\times B[k,j]$$

2. OpenMP 代码

在串行代码基础上，只需添加如下一行的编译指示即可实现并行。

```
64      //计算C
65      #pragma omp parallel for
66      for (int i = 0; i < n; i++) {
67          for (int j = 0; j < n; j++) {
68              for (int k = 0; k < n; k++) {
69                  C[i][j] += A[i][k] * B[k][j];
70              }
71          }
72      }
```

此行代码将最外层 for 循环拆分到多个线程中，在矩阵乘法计算中，就相当于结果矩阵 C，以行为单位进行划分，各线程负责计算矩阵 C 不同的行。而在计算过程中各线程并行读取 A 矩阵与 B 矩阵内容，但是负责将计算结果写入矩阵 C 的不同行。

3. MPI 代码

每个进程都执行 init()函数，所以每个进程都会初始化矩阵 A、B、C，所以都可以在各自进程内访问到矩阵 A、B 的值。而计算任务划分，借鉴了 OpenMP 实现中的思路，把结果矩阵 C，以行为单位进行划分，各进程负责计算矩阵 C 不同的行，并且将这一行的计算结果发送到主进程（0 号进程）。而任务划分可以举一个例子来说明，比如 id 为 1 的进程负责 C 矩阵行下标为 1、1+n_threads、1+2*n_threads ... 等的行的计算。核心代码如下：

```
65      //各进程并行计算C：以行为单位进行划分计算任务
66      for (int i = myid; i < n; i += n_threads) {
67          for (int j = 0; j < n; j++) {
68              for (int k = 0; k < n; k++) {
69                  C[i][j] += A[i][k] * B[k][j];
70              }
71          }
72      }
73      //将计算结果C的第i行，发送到主进程
74      if(myid != 0){
75          for (int i = myid; i < n; i += n_threads) {
76              MPI_Send(C[i], n, MPI_INT, 0, i, MPI_COMM_WORLD);
77          }
78      }
```

二、 实验结果

1. 运行环境

物理机 CPU：Intel i5-7200U （双核），内存 16GB

虚拟机：CPU 分配 2 个核，内存 4GB

虚拟机系统环境：Ubuntu 16.04（64 位版本）、gcc version 5.4.0 20160609、mpicc for MPICH version 3.3.2

2. OpenMP 实现运行结果

使用如下命令编译源代码：

gcc -fopenmp -o MatrixMtp_OpenMP MatrixMtp_OpenMP.c

再按如下格式运行可执行文件：

./MatrixMtp_OpenMP　4　1000

（第 1 个参数表示线程个数、第 2 个代表方阵阶数；缺省值分别为 1 和 1000）

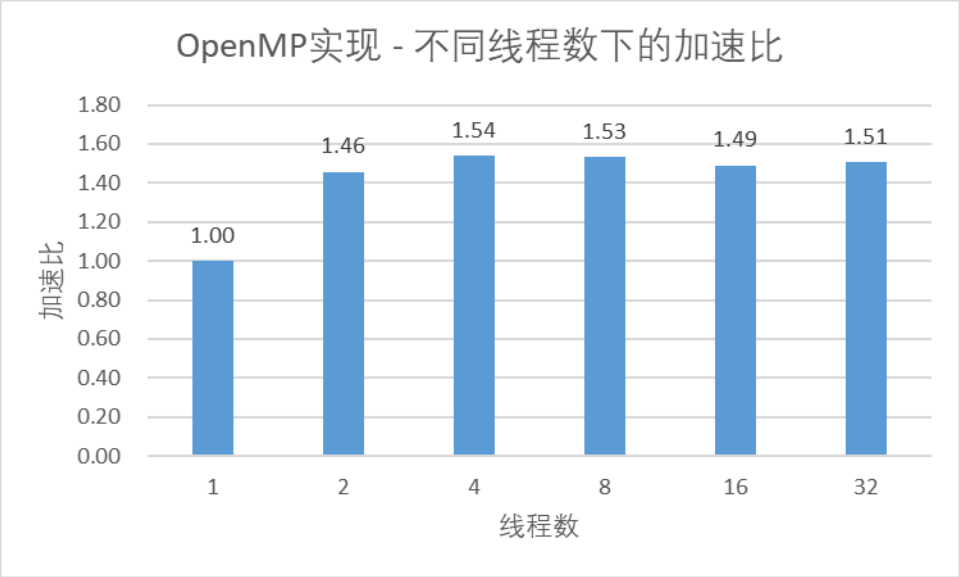
运行结果：

```
Sum of Matrix C:1000000000000
Time:7.103477 s
```

方阵阶定为 1000，线程数分别取 1、2、4、8、16 和 32 时，为减少误差每项实验进行 5 次。可得到数据如下表：

线程数	1	2	4	8	16	32
执行时间（单位：s）	9.63	6.01	5.85	5.89	5.96	6.03
	9.64	6.45	5.91	6.01	6.62	6.30
	9.02	6.47	5.95	5.90	6.36	6.10
	8.90	6.55	6.20	5.92	6.18	6.26
	9.21	6.35	6.19	6.57	6.00	6.10
平均时间	9.28	6.37	6.02	6.06	6.22	6.16
加速比	**1.00**	**1.46**	**1.54**	**1.53**	**1.49**	**1.51**

将加速比绘制成柱状图如下：



3. MPI 实现运行结果

使用如下命令编译源代码：

mpicc -o MatrixMtp_MPI MatrixMtp_MPI.c

再按如下格式运行可执行文件：

mpirun -np 1 ./MatrixMtp_MPI 1000

（“1”表示进程个数；“1000”代表方阵阶数，缺省值为 1000）

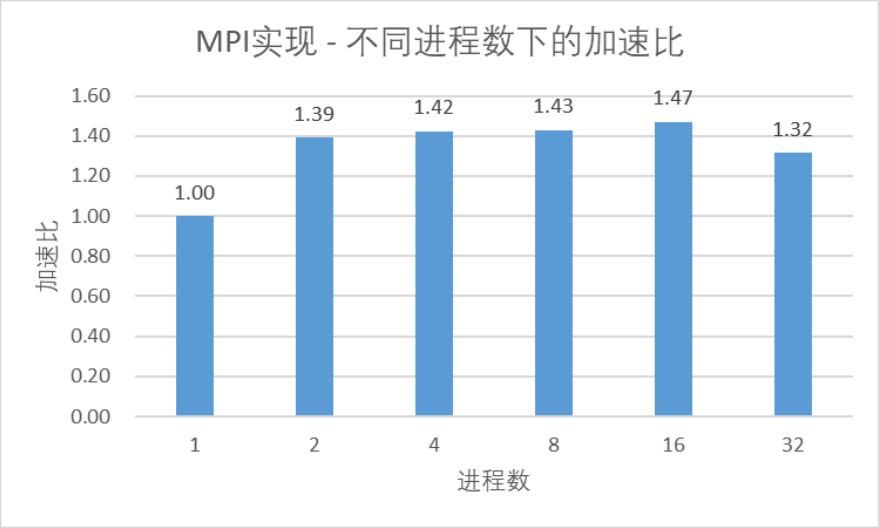
运行结果：

```
Sum of Matrix C:1000000000000
Time:9.324955 s
```

方阵阶定为 1000，进程数分别取 1、2、4、8、16 和 32 时，为减少误差每项实验进行 5 次。可得到数据如下表：

进程数	1	2	4	8	16	32
执行时间	10.86	7.15	7.53	6.95	7.33	7.74
	11.10	6.92	7.14	7.31	7.30	7.23
	10.37	7.43	6.66	7.06	6.62	7.38
	8.86	7.03	6.96	6.75	6.45	7.62
	8.87	7.45	6.87	6.91	6.34	8.06
平均时间	10.01	7.20	7.03	7.00	6.81	7.60
加速比	**1.00**	**1.39**	**1.42**	**1.43**	**1.47**	**1.32**

将加速比绘制成柱状图如下：



4. 结果分析

两种实现方式下，呈现出统一的趋势，即：随着线程/进程数增加，加速比逐渐增大到最大值，再随着进程数的增大，加速比逐渐有所下降。

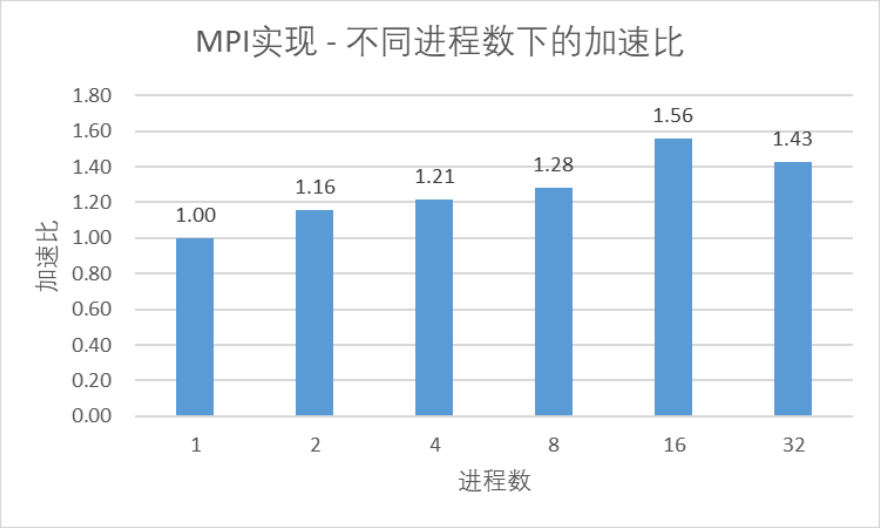
首先，一开始时线程/进程数增加，可以越来越充分利用其物理 CPU 核的并行资源，从而加速计算。而线程/进程数增加到一定程度，物理 CPU 核总是有限的，再增加线程/进程数也无法从物理上增加程序运行的并行程度，反而因为线程/进程数的增加，带来切换线程/进程、线程/进程间通信和管理等的额外开销，所以使得运行时间增加。

而在本次实验中，从 OpenMP 与 MPI 两种实现的对比中来看，OpenMP 实现更好。原因如下：① 实现简单，从串行代码到并写代码只需要考虑加入一行编译指示，而无需考虑任务划分、数据传送汇总等等一些系列问题；② 效果好，从加速比的柱状图可以看出，OpenMP 实现中，从 1 线程到 2 线程就基本到达了加速比上限，说明此实现中可以比较充分地利用好 CPU 物理上的并行计算资源。

三、 实验中遇到的问题及解决办法

1. 初版 MPI 实现效果不理想

初版 MPI 实现得出来的加速比结果如下图所示：



可以看出，刚开始进程数增加，带来的加速比提升很小，所以程序的并行效果并不理想。

解决：

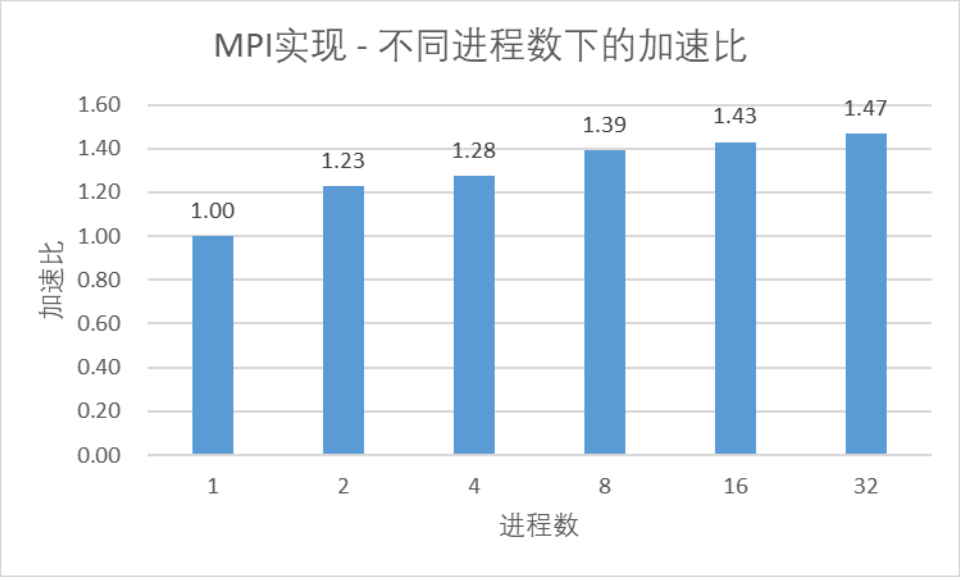
仔细分析原来的代码，最终发现了问题所在。初版实现的 MPI 核心代码如下：

```
62      //各进程并行计算C：以行为单位进行划分计算任务
63      for (int i = myid; i < n; i += n_threads) {
64          for (int j = 0; j < n; j++) {
65              for (int k = 0; k < n; k++) {
66                  C[i][j] += A[i][k] * B[k][j];
67              }
68          }
69          if(myid != 0)
70              //将计算结果C的第i行，发送到主进程
71              MPI_Send(C[i], n, MPI_INT, 0, i, MPI_COMM_WORLD);
72      }
```

在此代码中，每计算完矩阵 C 的一行，就 Send 发送，而 Send 是阻塞型的，需要等待 Recv 完毕才能返回。而主进程 Recv 的顺序如下，是按进程 id 的先后来接收，所以就导致 id 值大的进程会一直阻塞住，等 id 小的进程 Send 且 Recv 接收完了，才能继续。所以导致了并行效果不好。

```
//主进程接收其他进程的计算结果
for (int id = 1; id < n_threads; id++) {
    for (int i = id; i < n; i += n_threads) {
        MPI_Recv(C[i], n, MPI_INT, id, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

之后，我通过使用 Isend 函数（避免阻塞）验证了上述想法，得到了一个更为理想的结果：



当然，最根本的办法就是调整 Send 的时机：正确方式应该是全部计算完了之后开始逐一 Send，而不是算完一行就 Send 一行，因为主要的运行时间是花在计算矩阵 C 上，所以计算过程要提高并行程度，避免阻塞。这也就是最终我所采用的 MPI 实现方法。

四、 实验总结

1. 实验体会

- 使用 OpenMP 可以较为方便地从串行代码中修改得到并行代码，但是操作的灵活性不如 MPI，会有更多的局限性，适用面不如 MPI 广。
- MPI 程序设计需要仔细考量如何以及何时进行进程间数据传输，考虑到阻塞、缓冲区等等一系列细节问题。如果设计不当，则很可能并行效果不理想。

2. 可能的改进之处

- 在 MPI 实现中，如果最终仅仅只是需要计算矩阵 C 的元素之和，或许可以使用 reduce 操作，按累加和进行规约，直接得到各进程计算元素之和，这样实现会更加简单高效。