并行程序设计与算法实验2

实验	MPI集合通信矩阵乘法	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazp@mail2.sysu.edu.cn	完成日期	2024/04/05

1. 实验目的

> 改进上次实验中的MPI并行矩阵乘法(MPI-v1)，并讨论不同通信方式对性能的影响。
>
> 1. 采用MPI集合通信实现并行矩阵乘法中的进程间通信；使用mpi_type_create_struct聚合MPI进程内变量后通信；尝试不同数据/任务划分方式（选做）。
> 2. 对于不同实现方式，调整并记录不同线程数量（1-16）及矩阵规模（128-2048）下的时间开销，填写表格，并分析其性能及扩展性。

2. MPI集合通信并行矩阵乘法实现

并行矩阵乘法的大体思路是用A的部分行与整体B相乘，得到结果C的部分行。因此，在矩阵乘法中，我们可以用MPI_Scatter散射矩阵A到通信子中的各个进程；用MPI_Bcast广播矩阵B到通信子中的各个进程。在各个进程完成自己部分的矩阵乘法后，再通过MPI_Gather将各个进程的localC收集到进程0.

代码详见github

2.1 MPI_Scatter

核0(master core)分发矩阵至[0 to comm_sz - 1]核：

```
int avg_rows = N / comm_sz;
...
// scatter A
MPI_Scatter(A, N * avg_rows, MPI_FLOAT, localA, N * avg_rows, MPI_FLOAT, 0,
MPI_COMM_WORLD);
```

根据MPI_Scatter函数，进行参数的解释：

```
int MPI_Scatter(
    void*         send_buf_p      /* in  */,
    int           send_count      /* in  */,
    MPI_Datatype  send_type       /* in  */,
    void*         recv_buf_p      /* out */,
    int           recv_count      /* in  */,
    MPI_Datatype  recv_type       /* in  */,
    int           src_proc        /* in  */,
    MPI_Comm      comm            /* in  */);
```

将矩阵A（一维数组实现）分成comm_sz份，第一份给0号进程，第二份给1号进程，以此类推。每份的大小为N * avg_rows，即为send_count；数据类型为MPI_FLOAT。接收buffer为localA，同样的，接收大小为N * avg_rows，接收数据类型为MPI_FLOAT，0号进程为源程序，通信子为MPI_COMM_WORLD。

2.2 MPI_Bcast

整个矩阵B广播给所有进程：

```
// broadcast B
MPI_Bcast(B, N * N, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

根据MPI_Scatter函数，进行参数的解释：

```
int MPI_Bcast(
    void*         data_p        /* in/out */,
    int           count         /* in     */,
    MPI_Datatype  datatype      /* in     */,
    int           source_proc   /* in     */,
    MPI_Comm      comm          /* in     */);
```

广播的数据即为矩阵B，数据个数为N * N，数据类型为MPI_FLOAT，源进程为0，处于MPI_COMM_WORLD通信子下。

2.3 MPI_Gather

当各个进程计算结束后，MPI_Gather收集localC，汇聚到矩阵C。

```
// gather c
MPI_Gather(localC, avg_rows * N, MPI_FLOAT, C, avg_rows * N, MPI_FLOAT, 0,
MPI_COMM_WORLD);
```

同样，根据MPI_Gather函数，进行参数的解释：

```
int MPI_Gather(
    void*         send_buf_p      /* in  */,
    int           send_count      /* in  */,
    MPI_Datatype  send_type       /* in  */,
    void*         recv_buf_p      /* out */,
    int           recv_count      /* in  */,
    MPI_Datatype  recv_type       /* in  */,
    int           dest_proc       /* in  */,
    MPI_Comm      comm            /* in  */);
```

数据源为localC，发送数据大小为N * avg_rows，发送数据类型为MPI_FLOAT。矩阵C接收，接收数据大小为N * avg_rows，接收数据类型为MPI_FLOAT，目标进程即为0号进程。处于MPI_COMM_WORLD通信子内。

3. 实验结果

N = 128

```
PP2 ) mpiexec -n 1 mpicol.out
Cost time: 0.005921
PP2 ) mpiexec -n 2 ./mpicol.out
Cost time: 0.002949
PP2 ) mpiexec -n 4 ./mpicol.out
Cost time: 0.001551
PP2 ) mpiexec -n 8 ./mpicol.out
Cost time: 0.000946
PP2 ) mpiexec -n 16 ./mpicol.out
Cost time: 0.000482
```

N = 256

```
PP2 ) mpiexec -n 1 mpicol.out
Cost time: 0.048979
PP2 ) mpiexec -n 2 ./mpicol.out
Cost time: 0.025063
PP2 ) mpiexec -n 4 ./mpicol.out
Cost time: 0.015384
PP2 ) mpiexec -n 8 ./mpicol.out
Cost time: 0.006214
PP2 ) mpiexec -n 16 ./mpicol.out
Cost time: 0.003992
```

N = 512

```
PP2 ) mpiexec -n 1 mpicol.out
Cost time: 0.391363
PP2 ) mpiexec -n 2 ./mpicol.out
Cost time: 0.197304
PP2 ) mpiexec -n 4 ./mpicol.out
Cost time: 0.101385
PP2 ) mpiexec -n 8 ./mpicol.out
Cost time: 0.062479
PP2 ) mpiexec -n 16 ./mpicol.out
Cost time: 0.085388
```

N = 1024

```
PP2 ) mpiexec -n 1 mpicol.out
Cost time: 3.582811
PP2 ) mpiexec -n 2 ./mpicol.out
Cost time: 1.892759
PP2 ) mpiexec -n 4 ./mpicol.out
Cost time: 1.070382
PP2 ) mpiexec -n 8 ./mpicol.out
Cost time: 0.643595
PP2 ) mpiexec -n 16 ./mpicol.out
Cost time: 0.661593
```

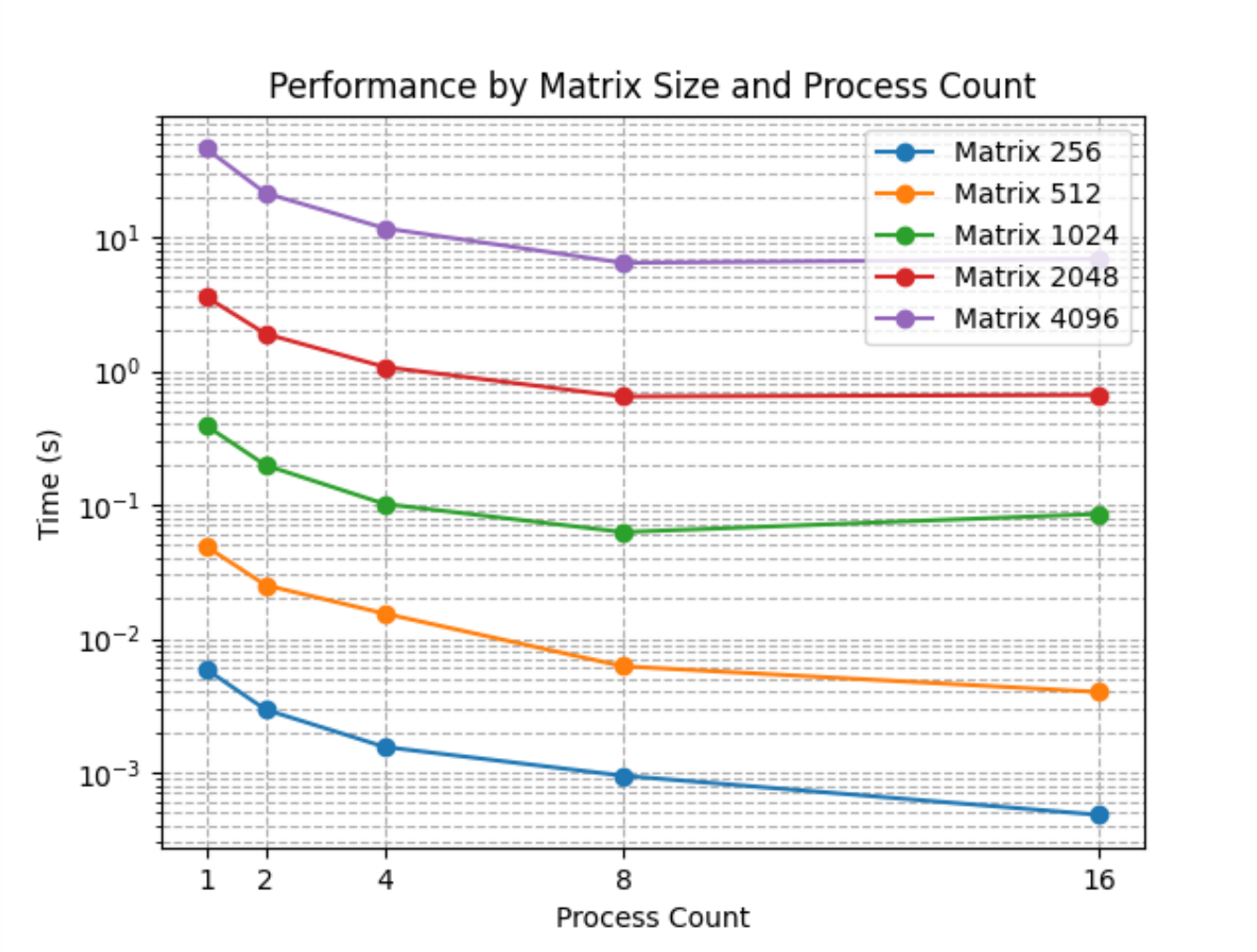
N = 2048

```
PP2 ) mpiexec -n 1 mpicol.out
Cost time: 45.888383
PP2 ) mpiexec -n 2 mpicol.out
Cost time: 21.289783
PP2 ) mpiexec -n 4 mpicol.out
Cost time: 11.637980
PP2 ) mpiexec -n 8 mpicol.out
Cost time: 6.418614
PP2 ) mpiexec -n 16 mpicol.out
Cost time: 6.849250
```

根据如上实验结果，得到表格：

P\N	128	256	512	1024	2048
1	0.005921	0.048979	0.391363	3.582811	45.888383
2	0.002949	0.025063	0.197304	1.892759	21.289783
4	0.001551	0.015384	0.101385	1.070382	11.637980
8	0.000946	0.006214	0.062479	0.643595	6.418614
16	0.000482	0.003992	0.085388	0.661593	6.849250

- 横向来看，对于给定的线程数，随着矩阵规模的增加，执行时间增加。这是因为工作量随着矩阵规模的增长呈指数增加。
- 纵向来看，对于给定的矩阵规模，增加线程数通常会减少执行时间，这体现了并行计算的优势。
- 而由于我的虚拟机只分配了8核，因此，使用超过8个线程可能不会给性能带来太多改善，甚至可能会因为线程上下文切换的开销而降低性能。
- 对于较小的矩阵（例如128x128），线程数的增加对执行时间的改善不大。但对于较大的矩阵（2048x2048），增加线程数可以显著降低执行时间，直到达到物理核心的数量。



根据表格，绘制图片。通过图片，我们可以更加直观地看到不同矩阵规模，以及不同进程数的加速效果。

4. 分块实现

4.1 分块矩阵乘法原理

借鉴自知乎

矩阵乘法的原始定义其实可以换个角度理解，把矩阵 A 看成 m 个 n 维行向量，矩阵 B 看成 s 个 n 维列向量。在我们实验中，可以将A矩阵拆成p个子矩阵，对应于(block_row = N / p)行N列的一个矩阵；将B矩阵拆成p个子矩阵，对应于(block_col = N / p)列N行的一个矩阵，即：

$$AB = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix} [B_0 \quad B_1 \quad \cdots \quad B_{p-1}] = \begin{bmatrix} A_0B_0 & A_0B_1 & \cdots & A_0B_{p-1} \\ A_1B_0 & A_1B_1 & \cdots & A_1B_{p-1} \\ \vdots & & \ddots & A_0B_{p-1} \\ A_{p-1}B_0 & A_{p-1}B_1 & \cdots & A_{p-1}B_{p-1} \end{bmatrix} \tag{1}$$

实现方式：

将可用于计算的进程数（comm_sz）分解为a*b，然后将全体行划分为a个部分，全体列划分为b个部分，从而将整个矩阵划分为size相同的（comm_sz）个块。每个子进程负责计算最终结果的一块，只需要接收A对应范围的行和B对应范围的列，而不需要把整个矩阵传过去。主进程负责分发和汇总结果。

注意：由于要用到集合通信的方式实现，我将B设计为列主序，这样就可以直接scatter去散射矩阵B了。此外，最后的gather C矩阵，由于分块的原因。不能一行一行的gather。需要有特殊额外的处理放到合适的位置。由于本实验主要目的是“并行”，不在我们的考虑之中，故没有实现。

4.2 代码实现

首先，我根据进程数的不同，划分矩阵A和矩阵B的方式也不同。

```
if (comm_sz == 2) {
    block_rows = N / 2;
}
else if (comm_sz == 4) {
    block_rows = N / 2;
    block_cols = N / 2;
}
else if (comm_sz == 8) {
    block_rows = N / 4;
    block_cols = N/ 2;
}
else if (comm_sz == 16) {
    block_rows = N / 4;
    block_cols = N / 4;
}
```

block_rows代表分块平均行数，block_cols代表分块平均列数。

将矩阵A和矩阵B分发：

```
// scatter A
MPI_Scatter(A, N * block_rows, MPI_FLOAT, localA, N * block_rows, MPI_FLOAT, 0,
MPI_COMM_WORLD);
// scatter B
MPI_Scatter(B, N * block_cols, MPI_FLOAT, localB, N * block_cols, MPI_FLOAT, 0,
MPI_COMM_WORLD);
```

计时：

```
MPI_Barrier(MPI_COMM_WORLD);
local_start = MPI_Wtime();
...
local_end = MPI_Wtime();
local_elapsed = local_end - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

与课本类似。

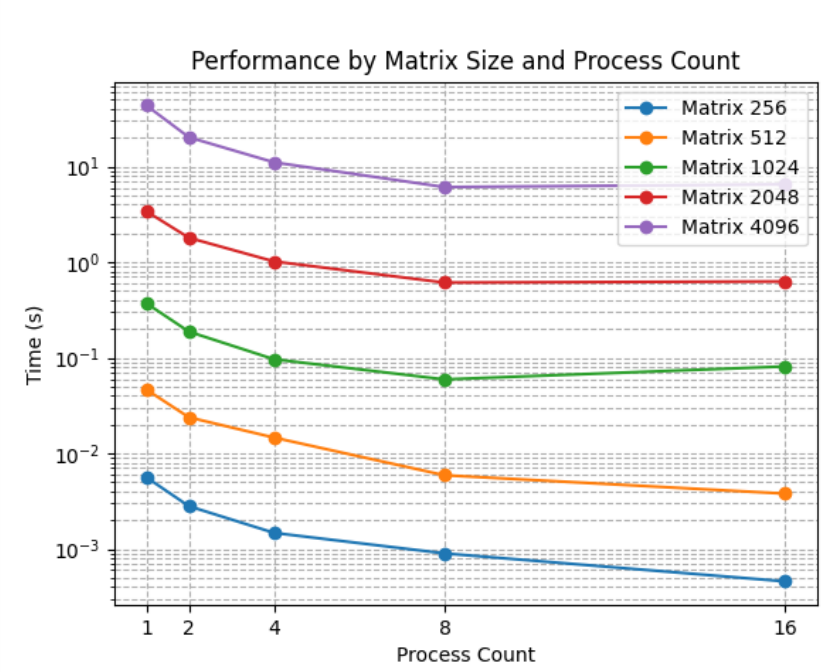
汇集：

```
// gather c
MPI_Gather(localC, block_rows * block_cols, MPI_FLOAT, C, block_rows * block_cols,
MPI_FLOAT, 0, MPI_COMM_WORLD);
```

4.3 实验结果

P \ N	128	256	512	1024	2048
1	0.005625	0.046530	0.371795	3.403670	43.593964
2	0.002802	0.023810	0.187439	1.798121	20.225294
4	0.001473	0.014615	0.096316	1.016863	11.056081
8	0.000899	0.005903	0.059355	0.611415	6.097683
16	0.000458	0.003792	0.081119	0.628513	6.506788

根据表格，绘制图片：



发现与本实验的曲线图，基本一致，但是时间略微减少。
分析：可能是由于空间局部性原理。在将矩阵分块之后，我们不需要将矩阵B整个放入到cache中。相反，只需要把部分的B放入到cache，这样，在读localA矩阵时，read miss就会减少。从而可以提高矩阵乘法运行效率。

5. 实验感想

在这次的并行程序设计与算法实验中，我们通过MPI集合通信实现了并行矩阵乘法，并进一步探索了分块实现的方法。整个实验过程不仅加深了我对并行计算概念的理解，也让我体会到了理论与实践结合的重要性。

首先，通过MPI集合通信的方法实现矩阵乘法，让我对进程间通信有了更直观的认识。利用MPI_Scatter、MPI_Bcast和MPI_Gather等函数，我成功地在多个进程之间分配任务并收集结果，这个过程中的数据分配和同步对于并行计算的效率至关重要。实验结果表明，随着进程数的增加，程序的执行时间明显减少，这验证了并行计算在处理大规模数据时的优势。

此外，分块实现的探索过程更是让我受益匪浅。通过将矩阵分块，我们不仅减少了通信的开销，还通过提高数据的空间局部性来增加了计算效率。这一部分的实验不仅让我理解了分块矩阵乘法的原理，也让我深刻感受到了优化算法对于提高程序性能的重要性。

在整个实验过程中，我也遇到了一些挑战，比如对于不同大小的矩阵，如何选择最优的进程数和分块策略。这需要我们不断地尝试和调整，才能找到最佳的解决方案。此外，虽然增加进程数可以减少执行时间，但当进程数超过物理核心数时，性能的提升会受到限制，这也提醒我们在并行计算中要考虑到硬件的限制。

总体来说，这次实验不仅让我对并行计算有了更深刻的认识，也锻炼了我的问题解决能力和优化思维。我相信这些知识和技能将在我的未来学习和研究中发挥重要作用。