Introduction to High Performance Computing
PA1
Report

计 86　任一　2018011423

2020 年 3 月 27 日

实验环境	
操作系统:	Windows10 家庭版 18362.72 Windows Subsystem for Linux
mpicc 版本:	gcc version 7.5.0

1 Ex3.13

1.1 解题思路

本题主要任务是解决 MPI_Scatter 和 MPI_Gather 在数组长度无法被进程数整除时不能使用的问题，因此在本题中，我使用了 MPI_Scatterv 和 MPI_Gatherv 进行数组的分发和收集。

在 MPI_Scatterv 和 MPI_Gatherv 这两个函数中，与 MPI_Scatter 和 MPI_Gather 较为不同的两个参数是 displacement 和 datacount，这两个参数都是整型数组类型的。displacement 指各进程接收到的数据在数组中的开始位置，datacount 则是每个进程所接收的数据量，通过这两个参数的指定，MPI_Scatterv 和 MPI_Gatherv 即可实现每个进程分配指定数量的数据，因而更为灵活一些。

由于第一次书面作业的 Exercise1.1 中涉及到了此题的理论基础，即在数组长度不能被进程数整除时，各进程的元素分配计算，因此只需参考第一次作业 Exercise1.1 题的结果，即可完成此题中 displacement 和 datacount 参数的计算。不过还有一种特殊情况是，数组长度小于进程数。我对于这种情况的处理是，优先进程编号小的进程获得数据，即进程编号小于数组长度的进程分配一个元素，其他进程不分配元素也不参与运算。但是涉及到其他无元素进程的内存分配问题还值得商榷，例如出现 malloc(0) 这样的情况，这种情况的可靠性还有待商榷，不过在本实验的测试中，这种方法并没有问题。

1.2 测试

本并行程序的测试通过与串行程序的对拍进行。在本地测试中，对拍 1000 组数据没有产生错误。下面是部分对拍测试截图。在集群上，只需在本题文件夹下输入 make run 命令即可运行本程序 (makefile 中设置了默认使用 4 进程)，输入 make check 命令即可进行对拍测试，更多测试功能详见本题 makefile.

```
nmren@DESKTOP-NV1R0OE:/mnt/d/Tsinghua/2020Spring/HPC/materials/ipp-source/ipp-source-use/HW/PA1/Ex3.13$ make check
python check.py ./serial "mpiexec -n 4 parallel" "python datamaker.py"
Running Case #1 ... OK
Running Case #2 ... OK
Running Case #3 ... OK
Running Case #4 ... OK
Running Case #5 ... OK
Running Case #6 ... OK
Running Case #7 ... OK
Running Case #8 ... OK
Running Case #9 ... OK
Running Case #10 ... OK
Running Case #11 ... OK
Running Case #12 ... OK
Running Case #13 ... OK
Running Case #14 ... OK
Running Case #15 ... OK
Running Case #16 ... OK
Running Case #17 ... OK
Running Case #18 ... OK
Running Case #19 ... OK
Running Case #20 ... OK
```

图 1: 并行程序与对拍程序部分结果图

```
Running Case #1298 ... OK
Running Case #1299 ... OK
Running Case #1300 ... OK
Running Case #1301 ... OK
Running Case #1302 ... OK
Running Case #1303 ... OK
Running Case #1304 ... OK
Running Case #1305 ... OK
Running Case #1306 ... OK
Running Case #1307 ... OK
Running Case #1308 ... OK
Running Case #1309 ... OK
Running Case #1310 ... OK
Running Case #1311 ... OK
Running Case #1312 ... OK
Running Case #1313 ... OK
```

图 2: 并行程序与对拍程序部分结果图

2 Exercise 3.11

2.1 解题思路

本题使用 MPI_Scan 函数，对 p 个进程，每个进程有 10 个元素的数组求了前缀和。我的思路是，首先对每个进程求局部和，然后利用 MPI_Scan 函数，使得每个进程 i 都有进程 0 至 i 这 i+1 个进程的元素总和，接着在每个进程里，通过用 MPI_Scan 得到的总和，从后向前累计减去当前进程中的元素，即可所求的全局前缀和。最后将这些全局前缀和通过 MPI_Gather 函数，汇总到 0 号进程统一输出。

2.2 测试

为了测试，我编写了串行求前缀和的程序，与并行程序进行对拍。通过本机测试得到，对拍 1000 组长度为 40 的前缀求和，没有发生错误。

2.3 运行方式

为了能够满足题目中" 每个进程随机生成的 10 个数" 以及与串行程序对拍的需求，我参考了 ch3 中 mpi_odd_even.c 中实现的命令行参数控制数据输入方式。

具体来说，在集群上运行 srun -n <p> parallel -<g|i>, 其中 p 代表进程数，-g 代表使用自动随机生成的数据，-i 代表使用用户从命令行输入的 10p 个数据。通过-i 的命令设置，我能够从文件中重定向输入，从而完成与串行程序的对拍。在集群上，只需在本题文件夹下输入 make run 命令即可运行本程序 (makefile 中设置了默认使用 4 进程)，输入 make check 命令即可进行对拍测试，更多测试功能详见本题 makefile.

3 Ex 3.1

3.1 解题思路

本题需要补充 Find_bins 和 Which_bins 函数。在 Find_bins 函数中，通过调用 Which_bins 函数统计当前进程所负责的元素在哪个 bin 中，得到局部的 bin 计数之后，使用 MPI_Reduce 函数将结果快速求和到 0 号进程。

在 Which_bins 函数中，由于每个 bin 的宽度都已知，是 $bin\_maxes[0] - min\_meas$, 因此 $data$ 所在的 bin 编号即为 $(int)((data - min\_meas)/(bin\_maxes[0] - min\_meas))$

3.2 测试

本程序在本地与集群上测试均表现正常。在集群上，只需在本题文件夹下输入 make run 命令即可运行本程序 (makefile 中设置了默认使用 4 进程).

4 Ex3.5

4.1 解题思路

在本题中，我使用了附上的 prog3.5_mpi_mat_vect_col.c 程序完成。在这份代码中，已经基本实现了 0 号进程并行按列分块然后分发到不同进程的功能。其中关键的函数是 Build_derived_type 中的 MPI_Type_vector 和 MPI_Type_create_resized.

MPI_Type_vector 接受 5 个参数，前三个较为关键，分别是 Number of blocks, Number of elements in blocks, Number of elements between start of each block. [1] 如果设矩阵是 $m * n$ 阶，进程数是 $comm\_sz$，那么这三个参数分别应该设为 $m, n/comm\_sz, n$，这样就可以满足把待计算的矩阵根据进程数按列分块，每个进程得到 $m * (n/comm\_sz)$ 阶的矩阵进行计算。最后再使用 MPI_Reduce 得到最终结果。而 MPI_Type_create_resized 共接收 4 个参数，本题中较为关键的是第 2 个参数和第 3 个参数，含义分别为 New lower bound of data type, New extent of data type. [2] 这两个参数在本题中分别设置为 $0, n/comm\_sz$. 这样可以理解为，把新生成的数据类型的宽度调整为 $n/comm\_sz$，这样就可以在本题中方便地实现 Scatter.

下图示意了本题如何按列分块。[3]

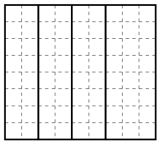

图 3: 通过 MPI_Type_vector 和 MPI_Type_create_resized 实现矩阵按列分块读取

[1] 此处参考https://www.open-mpi.org/doc/current/man3/MPI_Type_vector.3.php

[2] 此处参考https://www.open-mpi.org/doc/current/man3/MPI_Type_create_resized.3.php

[3] 此图来源https://pdfs.semanticscholar.org/afa8/9673d416faf99cfd6b4353ab810e8418f5d7.pdf

4.2 运行方式

在集群上，只需在本题文件夹下输入 make run 命令即可运行本程序 (makefile 中设置了默认使用 4 进程)，在运行 make run 后输入 python check.py 即可自动测试本报告中的数据，并且重定向到 result 文件夹下进行输出。

4.3 测试结果

本题进行的所有测试，并行计算结果与串行计算结果均相同 (差的二范数为 0)，从而保证了并行程序的正确性。

4.3.1 并行计算时间和串行计算时间图表分析

表 1: Ex3.5 并行计算时间 (s)

	Matrix Order						
Processors		1024	2048	4096	8192	16384	32768
	1	0.006984	0.018833	0.051525	0.232303	0.82091	3.714691
	2	0.003527	0.009607	0.025758	0.116467	0.410803	1.863362
	4	0.001778	0.004934	0.014211	0.056639	0.233671	0.873432
	8	0.000901	0.002473	0.007457	0.029537	0.117692	0.470057
	16	0.00047	0.001307	0.003816	0.014908	0.059785	0.236177
	32	0.000258	0.000531	0.002265	0.008542	0.033146	0.120416

表 2: Ex3.5 串行计算时间 (s)

	Matrix Order						
Processors		1024	2048	4096	8192	16384	32768
	1	0.006748	0.017442	0.051249	0.262236	0.819512	3.715776
	2	0.006644	0.018268	0.051153	0.232251	0.821548	3.714187
	4	0.006783	0.018551	0.056124	0.224963	0.931101	3.280052
	8	0.006998	0.018533	0.051223	0.232268	0.929077	3.715156
	16	0.006745	0.018167	0.058115	0.23203	0.901695	3.714686
	32	0.007508	0.014606	0.05868	0.234549	0.936246	4.624938

本部分中展示了不同进程数下，并行和串行计算时间随着矩阵阶数的变化表格和折线图。从图 4 中可以清晰看出，并行计算时间与矩阵阶数成正相关。由于矩阵元素数量与矩阵阶数成二次关系，图中的耗时与矩阵阶数也可以近似看作二次关系。同时可以看出，进程数越多，并行计算时间越少，这也充分体现了并行计算在计算时的优越性。

从图 5 中可以看出，串行计算时间与矩阵阶数近似为二次关系。不过矩阵阶数一定，不同的进程数下串行计算时间基本相同，这也基本符合串行计算在

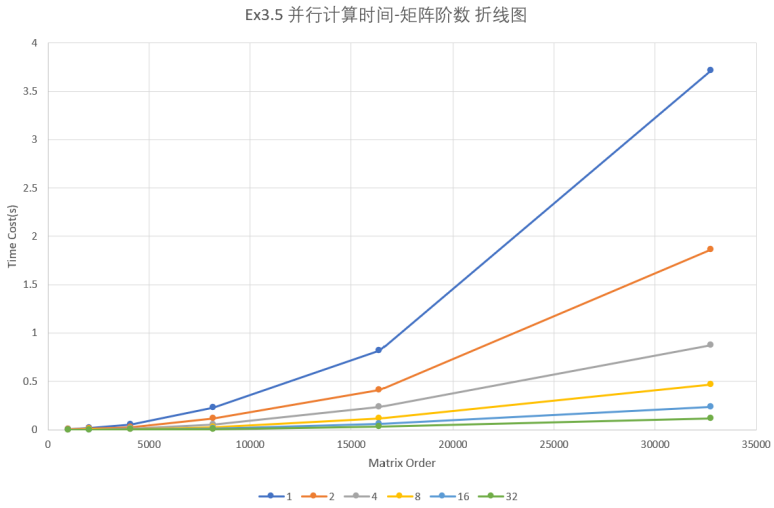

图 4: Ex3.5 并行计算时间-矩阵阶数折线图

本实验中只由 0 号进程计算的事实。同时从图中和表格中都可以看到，在矩阵阶数一定时，并行计算的时间要远低于串行计算，这再一次表明了并行程序在计算方面的优越性。

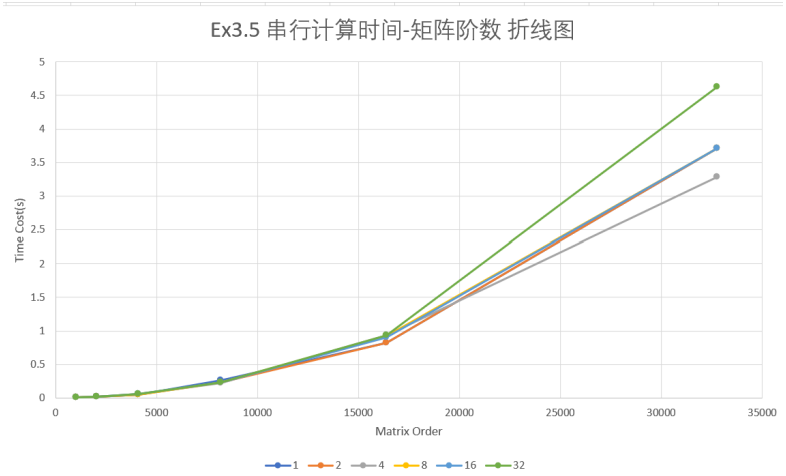

图 5: Ex3.5 串行计算时间-矩阵阶数折线图

4.3.2 并行分配时间与总时间图表分析

表 3: Ex3.5 并行分配时间 (s)

	Matrix Order						
Processors		1024	2048	4096	8192	16384	32768
	1	0.003748	0.011979	0.040306	0.165983	0.65822	2.654574
	2	0.034063	0.106454	0.304867	1.104077	3.911728	16.8321
	4	0.034214	0.108905	0.315048	1.101445	4.37147	15.888
	8	0.03481	0.108111	0.29167	1.121542	4.309148	17.13832
	16	0.036225	0.106354	0.276049	1.11216	4.273796	17.19085
	32	0.115958	0.254139	6.009296	3.238718	95.0812	52.53218

表 4: Ex3.5 并行总时间 (s)

	Matrix Order						
Processors		1024	2048	4096	8192	16384	32768
	1	0.010732	0.030812	0.091831	0.398286	1.47913	6.369265
	2	0.03759	0.116061	0.330625	1.220544	4.322531	18.69546
	4	0.035992	0.113839	0.329259	1.158084	4.605141	16.76143
	8	0.035711	0.110584	0.299127	1.151079	4.42684	17.60837
	16	0.036695	0.107661	0.279865	1.127068	4.333581	17.42703
	32	0.116216	0.25467	6.011561	3.24726	95.11434	52.6526

本部分展示了不同进程数下，并行分配时间和并行计算总时间与矩阵阶数的关系图表。从图 6 中可以看出，并行分配时间与矩阵阶数成正相关关系，不考虑进程数为 32 的图线的情况下，并行分配时间仍可近似认为与矩阵阶数成二次关系。

但是进程数为 32 时，当矩阵阶数是 16384 时，分配时间达到了 95s，这个结果较为反常。于是我对这组数据又进行了几次测试，包括对结果正确性的检验。在额外的这几次测试中，我得到的并行分配时间和并行计算时间均在 13-14s 左右，符合图线趋势，也和串行计算的结果完全相同。于是我认为，图表中出现的这组异常数据，并非由于我的程序错误导致，而是 OS 状态等原因造成的偶然情况。

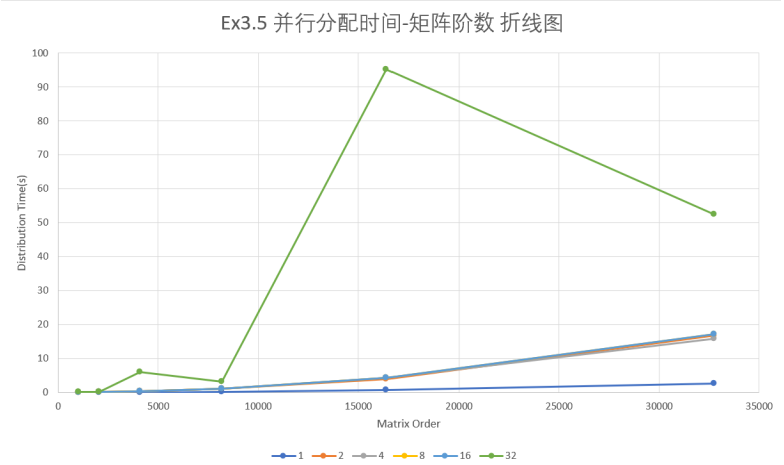

图 6: Ex3.5 并行分配时间-矩阵阶数折线图

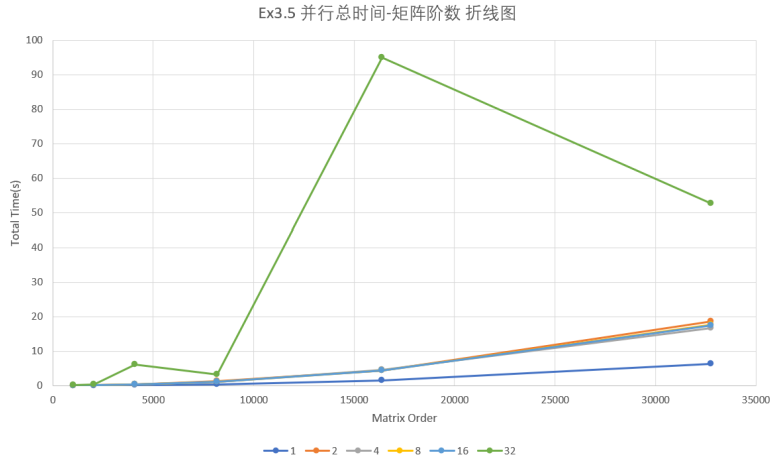

图 7: Ex3.5 并行总时间-矩阵阶数折线图

4.3.3 并行计算时间加速比与并行总时间加速比

表 5: Ex3.5 并行计算时间加速比

	Matrix Order						
Processors		1024	2048	4096	8192	16384	32768
	1	0.966208	0.92614	0.994643	1.128853	0.998297	1.000292
	2	1.883754	1.90153	1.985907	1.994136	1.999859	1.993272
	4	3.814961	3.75983	3.949335	3.971875	3.984666	3.75536
	8	7.766926	7.494137	6.869116	7.863629	7.894139	7.903629
	16	14.35106	13.89977	15.2293	15.56413	15.08229	15.7284
	32	29.10078	27.50659	25.90728	27.45832	28.24612	38.408

表 6: Ex3.5 并行总时间加速比

	Matrix Order						
Processors		1024	2048	4096	8192	16384	32768
	1	0.628774	0.566078	0.55808	0.658411	0.55405	0.583392
	2	0.176749	0.1574	0.154716	0.190285	0.190062	0.198668
	4	0.188459	0.162958	0.170455	0.194254	0.202187	0.19569
	8	0.195962	0.167592	0.171242	0.201783	0.209874	0.210988
	16	0.183813	0.168743	0.207654	0.20587	0.208072	0.213157
	32	0.064604	0.057353	0.009761	0.07223	0.009843	0.087839

本部分展示了并行计算时间加速比与并行总时间加速比的图表。由图 8 可以看出，并行计算时间加速比基本稳定在进程数上下，这充分体现了并行程序在计算层面的优越性。在这里我也思考了，为什么进程数一定时，加速比没有随着矩阵阶数明显增加。我认为这是由于进行试验时，最小的矩阵阶数是 1024，因此最小的矩阵元素个数也有 10^8 之多，这个数量已经非常大了，因此在某一进程数下，计算时间加速比基本已经稳定在进程数上下。此外我们也可以注意到，当进程数达到 32 时，加速比随着矩阵阶数增加较为明显，这也能在一定程度上印证上面的说法。

从图 9 中可以看出，考虑到分配时间、计算时间等时间的并行总时间加速比并不高，并且该加速比随着进程数的增加而降低，随着矩阵阶数的增加变化不明显。这也提醒了我们，并行程序虽然计算速度非常快，但是数据分发效率不高。这可能是由于数据的分发、汇总等都是由 1 个进程即 0 号进程进

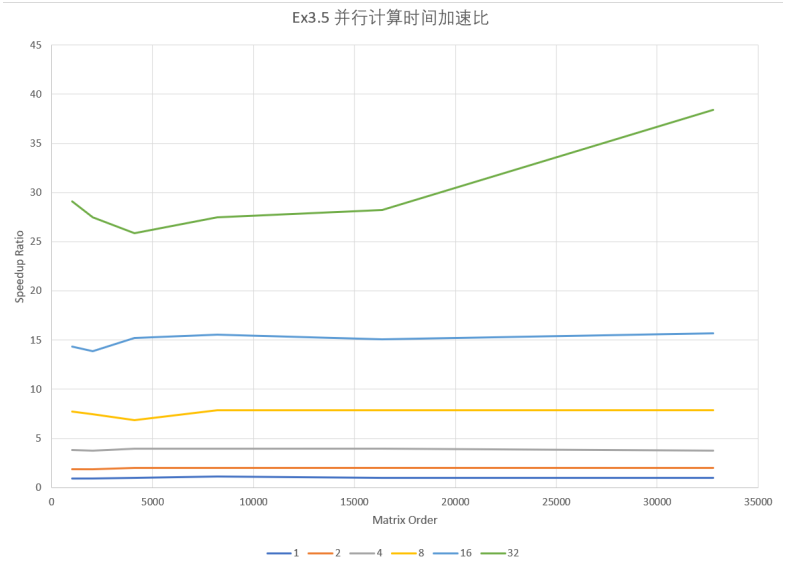

图 8: Ex3.5 并行计算时间加速比

行的，这导致了效率的低下。一种可能的改进方法是，如果能够让多个进程同时读取到应得的数据，这样就可以大大缓解这个问题。

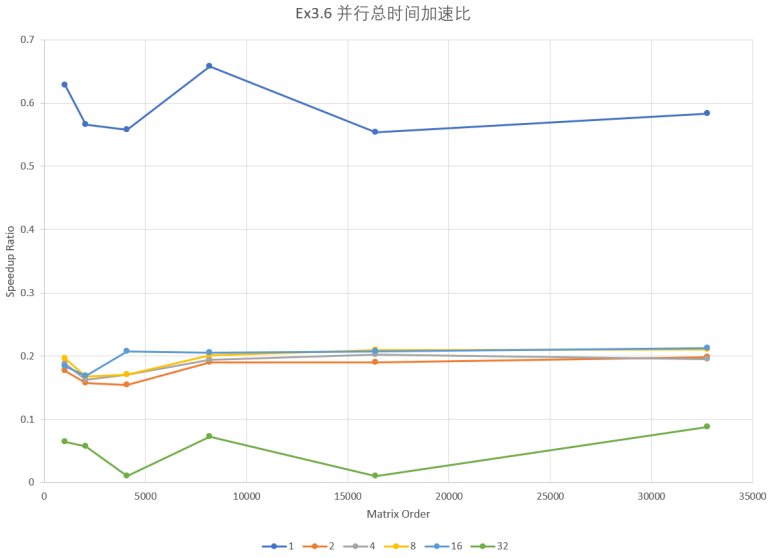

图 9: Ex3.5 并行总时间加速比

5 Ex3.6

5.1 解题思路

在本题中，我使用了附上的 prog3.5_mpi_mat_vect_col.c 程序完成。在这份代码中，已经基本实现了 0 号进程并行按列分块然后分发到不同进程的功能。我通过修改一些参数和发送方式，完成了本题对矩阵进行分块然后再计算的要求。我主要进行修改的地方是函数 Build_derived_type 中的 MPI_Type_vector 和 MPI_Type_create_resized，以及实现了通过编写不同的通信域实现最终结果的 Reduce 和 Gather.

MPI_Type_vector 和 MPI_Type_create_resized 的解释已经在 Ex3.5 的解题思路当中详细说明了，在此不再重复。本题中通过这两个函数的使用，成功实现了将矩阵进行分块并且发送，原理与 Ex3.5 相同，只不过参数进行了一些调整。与 Ex3.5 不同，本题使用了 Scatterv 的发送方法，这是因为必须要指定发送的起始位置，才可以保证分块矩阵没有重叠地发送到各个进程中。

在编写本题时，另一个难点是如何将分块的计算结果，进行 Reduce 到某些核并最终 Gather 到 0 号核。在这个问题上，我参考了`https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/` 这篇文章中提到的方法。在 Reduce 时，把每一行的分块矩阵编入一个通信域，使得每行的分块矩阵 Reduce 到最左边的分块矩阵。在 Gather 时，令每一行最左边的分块矩阵 Gather 到 0 号核，即可完成计算。

下面的图 10 与图 11 示意了本题的简要流程。[4] [5]

[4]图 10`https://pdfs.semanticscholar.org/afa8/9673d416faf99cfd6b4353ab810e8418f5d7.pdf`

[5]图 11`https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/`

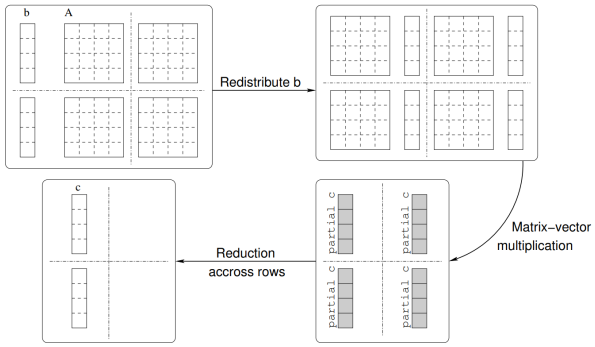

图 10: Ex3.6 简要思路

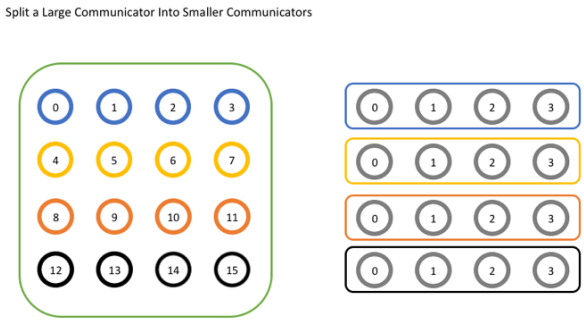

图 11: Ex3.6 Reduce 时通讯子分组

5.2 运行方式

在集群上，只需在本题文件夹下输入 make run 命令即可运行本程序 (makefile 中设置了默认使用 4 进程)，在运行 make run 后输入 python check.py 即可自动测试本报告中的数据，并且重定向到 result 文件夹下进行输出。

5.3 测试结果

本题进行的所有测试，并行计算结果与串行计算结果均相同 (差的二范数为 0)，从而保证了并行程序的正确性。

5.3.1 并行计算时间和串行计算时间图表分析

表 7: Ex3.6 并行计算时间 (s)

	Matrix Order					
Processor		1080	2160	4320	7200	14400
	1	0.0077	0.021366	0.057277	0.158988	0.634204
	4	0.001955	0.005366	0.015299	0.042381	0.169856
	9	0.000874	0.002466	0.007333	0.020316	0.081537
	16	0.000479	0.001363	0.004174	0.011507	0.046374
	25	0.000383	0.000829	0.002663	0.007618	0.029684

表 8: Ex3.6 串行计算时间 (s)

	Matrix Order					
Processors		1080	2160	4320	7200	14400
	1	0.0077	0.021366	0.057277	0.158988	0.634204
	4	0.001955	0.005366	0.015299	0.042381	0.169856
	9	0.000874	0.002466	0.007333	0.020316	0.081537
	16	0.000479	0.001363	0.004174	0.011507	0.046374
	25	0.000383	0.000829	0.002663	0.007618	0.029684

本部分中展示了不同进程数下，并行和串行计算时间随着矩阵阶数的变化表格和折线图。从图 12 中可以清晰看出，与 Ex3.5 类似，并行计算时间与矩阵阶数成正相关。由于矩阵元素数量与矩阵阶数成二次关系，图中的耗时与矩阵阶数也可以近似看作二次关系。同时可以看出，进程数越多，并行计算时间越少，这也充分体现了并行计算在计算时的优越性。

从图 13 中可以看出，串行计算时间与矩阵阶数近似为二次关系。不过矩阵阶数一定，不同的进程数下串行计算时间基本相同，这也基本符合串行计算在本实验中只由 0 号进程计算的事实。同时从图中和表格中都可以看到，在矩阵阶数一定时，并行计算的时间要远低于串行计算，这再一次表明了并行

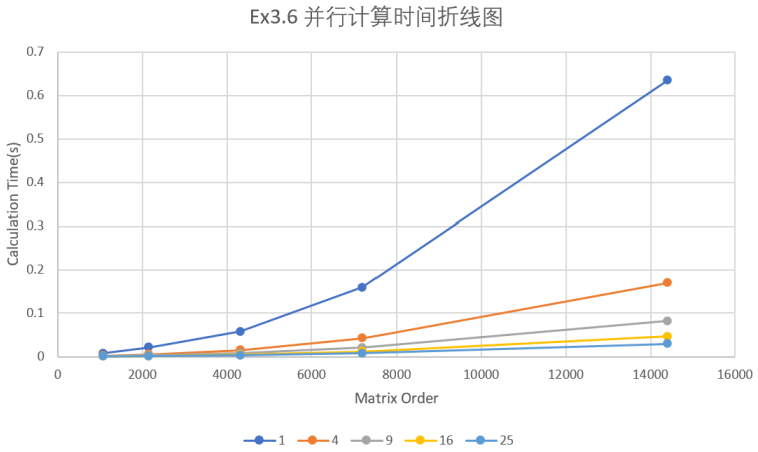

图 12: Ex3.6 并行计算时间-矩阵阶数折线图

程序在计算方面的优越性。

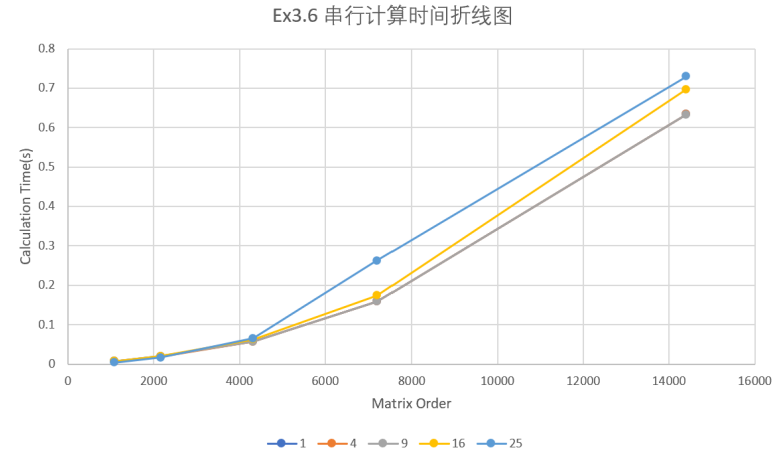

图 13: Ex3.6 串行计算时间-矩阵阶数折线图

5.3.2 并行分配时间与总时间图表分析

表 9: Ex3.6 并行分配时间 (s)

	Matrix Order					
Processors		1080	2160	4320	7200	14400
	1	0.003992	0.01379	0.045249	0.127489	0.502016
	4	0.038476	0.113324	0.324967	0.796448	3.001446
	9	0.038288	0.116956	0.324326	0.814763	3.085259
	16	0.039415	0.119221	0.351533	0.869724	3.335967
	25	0.089289	0.278221	0.965618	2.596891	10.076753

表 10: Ex3.6 并行总时间 (s)

	Matrix Order					
Processes		1080	2160	4320	7200	14400
	1	0.011692	0.035156	0.102526	0.286477	1.13622
	4	0.040431	0.11869	0.340266	0.838829	3.171302
	9	0.039162	0.119422	0.331659	0.835079	3.166796
	16	0.039894	0.120584	0.355707	0.881231	3.382341
	25	0.089672	0.27905	0.968281	2.604509	10.106437

本部分展示了不同进程数下，并行分配时间和并行计算总时间与矩阵阶数的关系图表。从图 14 中可以看出，并行分配时间与矩阵阶数成正相关关系，并行分配时间可近似认为与矩阵阶数成二次关系。

由于并行的分配时间在一定程度上远大于并行的计算时间，因此对比图 14 图 15 可以看出，并行总时间的变化趋势与并行分配时间的变化区时基本一致。

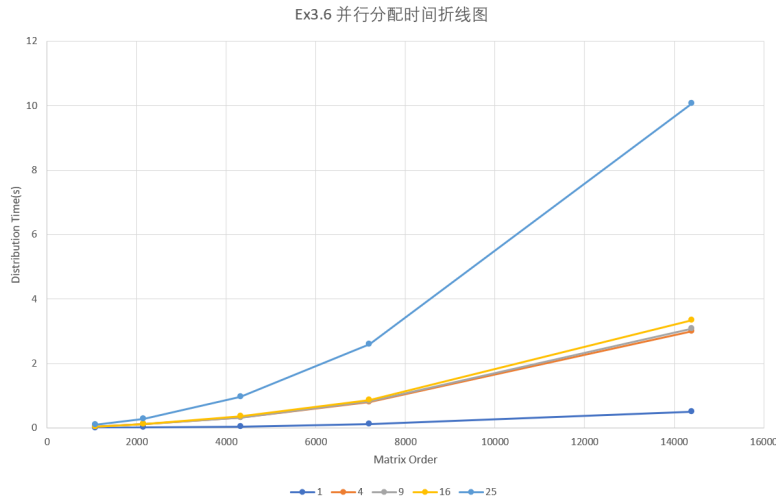

图 14: Ex3.6 并行分配时间-矩阵阶数折线图

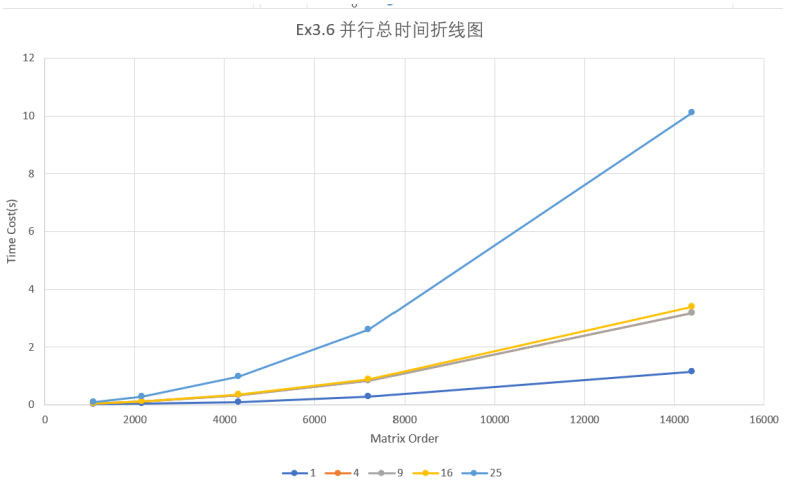

图 15: Ex3.6 并行总时间-矩阵阶数折线图

5.3.3 并行计算时间加速比与并行总时间加速比

表 11: Ex3.6 并行计算时间加速比

	Matrix Order					
Processors		1080	2160	4320	7200	14400
	1	0.947143	0.93382	1.000017	0.99844	0.999227
	4	3.742711	3.569884	3.737434	3.741512	3.732085
	9	8.354691	8.025953	7.798309	7.800354	7.760685
	16	15.14823	14.42406	15.02827	15.13261	15.03198
	25	10.62141	19.75513	24.39392	34.41139	24.59089

表 12: Ex3.6 并行总时间加速比

	Matrix Order					
Processors		1080	2160	4320	7200	14400
	1	0.62376	0.567528	0.558668	0.554111	0.557739
	4	0.180975	0.161395	0.168042	0.189036	0.199892
	9	0.186456	0.165732	0.172421	0.189769	0.199818
	16	0.181882	0.16304	0.176347	0.1976	0.206098
	25	0.045365	0.058688	0.067089	0.100651	0.072227

本部分展示了并行计算时间加速比与并行总时间加速比的图表。由图 16 可以看出，并行计算时间加速比基本稳定在进程数上下，这充分体现了并行程序在计算层面的优越性。此处进程数一定时，加速比没有随着矩阵阶数明显增加的原因与 Ex3.5 中类似，即实验中所采用的矩阵阶数都已经较大了

图 16 中，进程数为 25 时，加速比似乎有些不太正常，体现为波动较大。因此我对于进程数为 25 时的实验进行了额外的测试，发现的确加速比会有很大的振荡，从 10-30 左右的加速比均有可能出现。因此我认为，分块矩阵这种计算方法，可能由于计算的过程较为复杂，会产生一些波动，这也可能与 OS 的状态有关。

从图 17 中可以看出，与 Ex3.5 类似，考虑到分配时间、计算时间等时间的并行总时间加速比并不高，并且该加速比随着进程数的增加而降低，随着矩

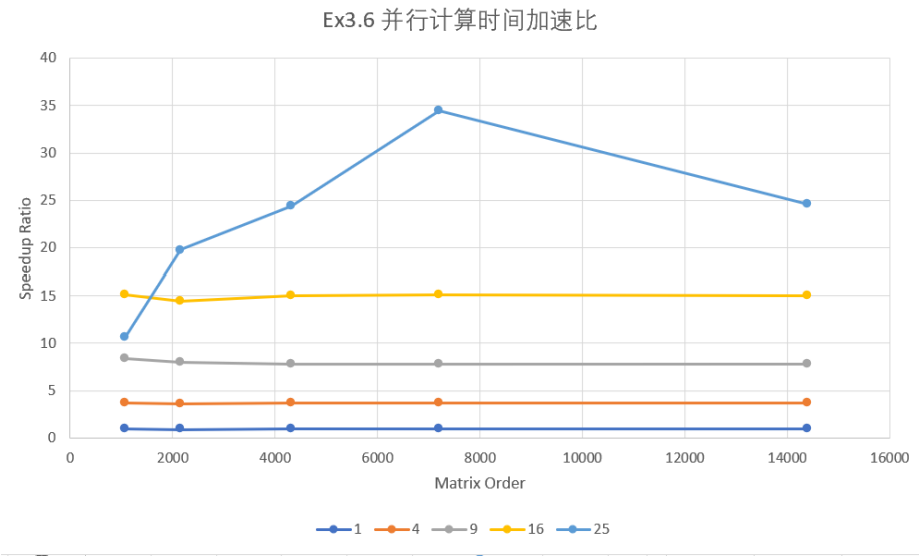

图 16: Ex3.6 并行计算时间加速比

阵阶数的增加变化不明显。这里与 Ex3.5 中得到的结论基本相同: 并行程序由于分发较为缓慢，在数据量不足够大时，效率并不一定高。

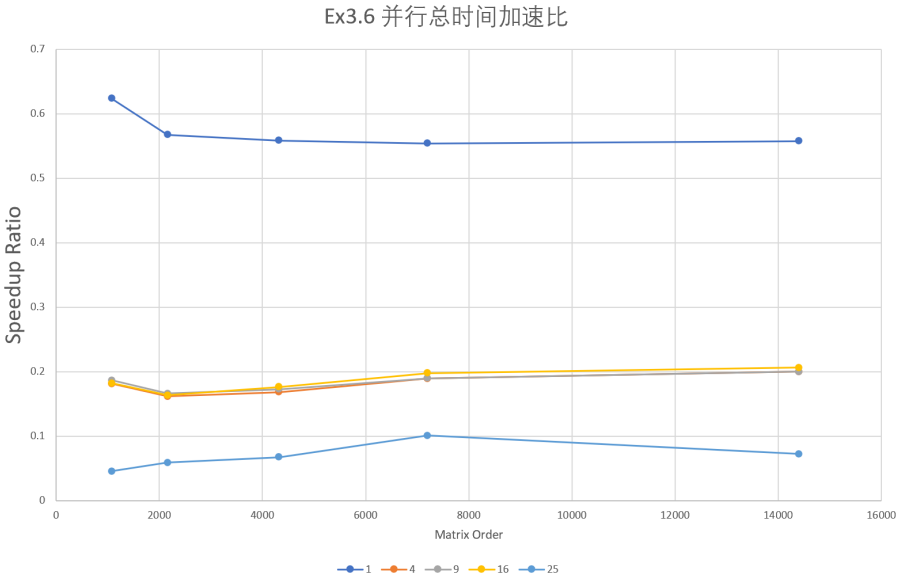

图 17: Ex3.6 并行总时间加速比

6 总结

在本次实验中，我锻炼了编写 MPI 并行程序的能力，也通过对运行结果的充分评估，加强了实验分析和思考的能力。同时，我也在本次实验中锻炼了使用服务器集群的能力。

感谢老师和助教在本次实验中给予我们的悉心指导！