（a）

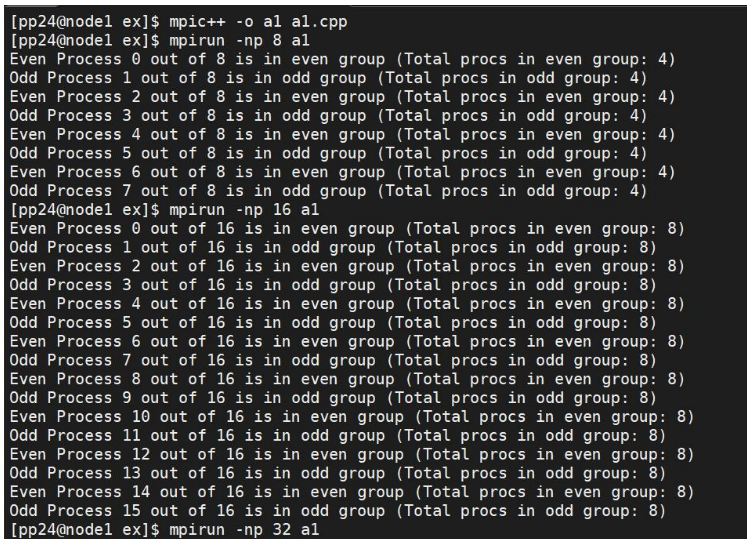
（1.1）写个将 MPI 进程按其所在节点分组的程序；（1.2）在 1.1 的基础上，写个广播程序，主要思想是：按节点分组后，广播的 root 进程将消息“发送”给各组的“0 号”，再由这些“0”号进程在其小组内执行 MPI_Bcast。

实现中，我采取的方法是将节点按照进程的秩的奇偶性进行分类

伪代码描述如下：

```
开始
    初始化 MPI 环境
    获取当前进程的 rank 和总进程数 size
    根据 rank 的奇偶性，将进程分配到对应的通信器
    如果 rank 是偶数
        将当前进程加入 even_comm
    否则
        将当前进程加入 odd_comm
    结束如果

    获取 even_comm 或 odd_comm 中的进程数 num_procs_in_comm
    打印信息：
        如果 rank 是偶数
            打印 "偶数进程 rank 总共 size 个进程，当前进程属于 even_comm 组，该组进程数为 num_procs_in_comm"
        否则
            打印 "奇数进程 rank 总共 size 个进程，当前进程属于 odd_comm 组，该组进程数为 num_procs_in_comm"
        结束如果

    释放 even_comm 或 odd_comm 通信器
    结束 MPI 环境
结束
```

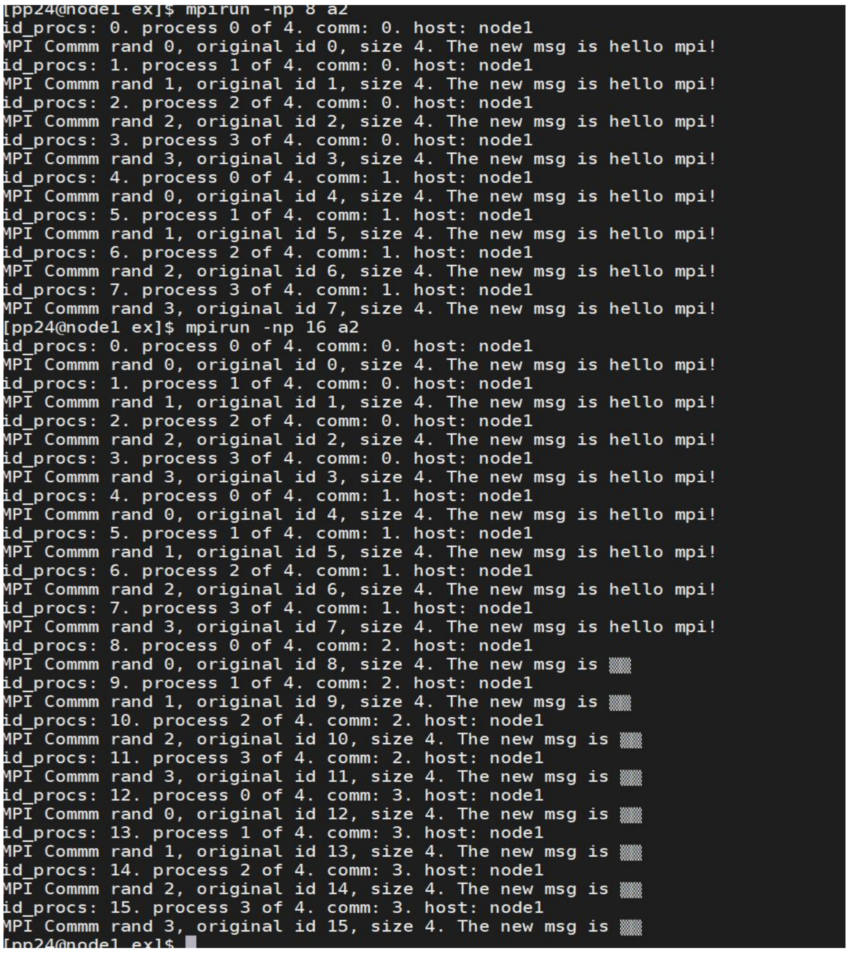
实验结果如下：



```
[pp24@node1 ex]$ mpic++ -o a1 a1.cpp
[pp24@node1 ex]$ mpirun -np 8 a1
Even Process 0 out of 8 is in even group (Total procs in even group: 4)
Odd Process 1 out of 8 is in odd group (Total procs in odd group: 4)
Even Process 2 out of 8 is in even group (Total procs in even group: 4)
Odd Process 3 out of 8 is in odd group (Total procs in odd group: 4)
Even Process 4 out of 8 is in even group (Total procs in even group: 4)
Odd Process 5 out of 8 is in odd group (Total procs in odd group: 4)
Even Process 6 out of 8 is in even group (Total procs in even group: 4)
Odd Process 7 out of 8 is in odd group (Total procs in odd group: 4)
[pp24@node1 ex]$ mpirun -np 16 a1
Even Process 0 out of 16 is in even group (Total procs in even group: 8)
Odd Process 1 out of 16 is in odd group (Total procs in odd group: 8)
Even Process 2 out of 16 is in even group (Total procs in even group: 8)
Odd Process 3 out of 16 is in odd group (Total procs in odd group: 8)
Even Process 4 out of 16 is in even group (Total procs in even group: 8)
Odd Process 5 out of 16 is in odd group (Total procs in odd group: 8)
Even Process 6 out of 16 is in even group (Total procs in even group: 8)
Odd Process 7 out of 16 is in odd group (Total procs in odd group: 8)
Even Process 8 out of 16 is in even group (Total procs in even group: 8)
Odd Process 9 out of 16 is in odd group (Total procs in odd group: 8)
Even Process 10 out of 16 is in even group (Total procs in even group: 8)
Odd Process 11 out of 16 is in odd group (Total procs in odd group: 8)
Even Process 12 out of 16 is in even group (Total procs in even group: 8)
Odd Process 13 out of 16 is in odd group (Total procs in odd group: 8)
Even Process 14 out of 16 is in even group (Total procs in even group: 8)
Odd Process 15 out of 16 is in odd group (Total procs in odd group: 8)
[pp24@node1 ex]$ mpirun -np 32 a1
```

对于第二问实现分组和广播
这里改为按照 4 个程序为一组，只需要判断进程是否是 0 号进程即可，如果在新的通信域中秩为 0，那么就进行广播。伪代码描述如下：

```
If newrank==0
    Bcast
Print newmessage
```



```
[pp24@node1 ex]$ mpirun -np 8 a2
id_procs: 0. process 0 of 4. comm: 0. host: node1
MPI Commm rand 0, original id 0, size 4. The new msg is hello mpi!
id_procs: 1. process 1 of 4. comm: 0. host: node1
MPI Commm rand 1, original id 1, size 4. The new msg is hello mpi!
id_procs: 2. process 2 of 4. comm: 0. host: node1
MPI Commm rand 2, original id 2, size 4. The new msg is hello mpi!
id_procs: 3. process 3 of 4. comm: 0. host: node1
MPI Commm rand 3, original id 3, size 4. The new msg is hello mpi!
id_procs: 4. process 0 of 4. comm: 1. host: node1
MPI Commm rand 0, original id 4, size 4. The new msg is hello mpi!
id_procs: 5. process 1 of 4. comm: 1. host: node1
MPI Commm rand 1, original id 5, size 4. The new msg is hello mpi!
id_procs: 6. process 2 of 4. comm: 1. host: node1
MPI Commm rand 2, original id 6, size 4. The new msg is hello mpi!
id_procs: 7. process 3 of 4. comm: 1. host: node1
MPI Commm rand 3, original id 7, size 4. The new msg is hello mpi!
[pp24@node1 ex]$ mpirun -np 16 a2
id_procs: 0. process 0 of 4. comm: 0. host: node1
MPI Commm rand 0, original id 0, size 4. The new msg is hello mpi!
id_procs: 1. process 1 of 4. comm: 0. host: node1
MPI Commm rand 1, original id 1, size 4. The new msg is hello mpi!
id_procs: 2. process 2 of 4. comm: 0. host: node1
MPI Commm rand 2, original id 2, size 4. The new msg is hello mpi!
id_procs: 3. process 3 of 4. comm: 0. host: node1
MPI Commm rand 3, original id 3, size 4. The new msg is hello mpi!
id_procs: 4. process 0 of 4. comm: 1. host: node1
MPI Commm rand 0, original id 4, size 4. The new msg is hello mpi!
id_procs: 5. process 1 of 4. comm: 1. host: node1
MPI Commm rand 1, original id 5, size 4. The new msg is hello mpi!
id_procs: 6. process 2 of 4. comm: 1. host: node1
MPI Commm rand 2, original id 6, size 4. The new msg is hello mpi!
id_procs: 7. process 3 of 4. comm: 1. host: node1
MPI Commm rand 3, original id 7, size 4. The new msg is hello mpi!
id_procs: 8. process 0 of 4. comm: 2. host: node1
MPI Commm rand 0, original id 8, size 4. The new msg is ▒
id_procs: 9. process 1 of 4. comm: 2. host: node1
MPI Commm rand 1, original id 9, size 4. The new msg is ▒
id_procs: 10. process 2 of 4. comm: 2. host: node1
MPI Commm rand 2, original id 10, size 4. The new msg is ▒
id_procs: 11. process 3 of 4. comm: 2. host: node1
MPI Commm rand 3, original id 11, size 4. The new msg is ▒
id_procs: 12. process 0 of 4. comm: 3. host: node1
MPI Commm rand 0, original id 12, size 4. The new msg is ▒
id_procs: 13. process 1 of 4. comm: 3. host: node1
MPI Commm rand 1, original id 13, size 4. The new msg is ▒
id_procs: 14. process 2 of 4. comm: 3. host: node1
MPI Commm rand 2, original id 14, size 4. The new msg is ▒
id_procs: 15. process 3 of 4. comm: 3. host: node1
MPI Commm rand 3, original id 15, size 4. The new msg is ▒
[pp24@node1 ex]$
```

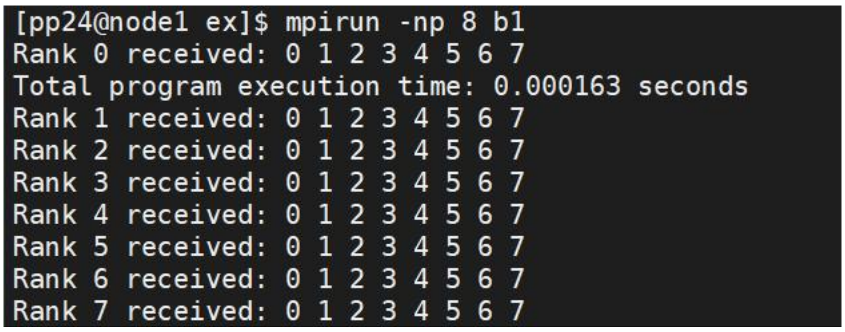
(2)

使用 MPI_Send 和 MPI_Recv 来模拟 MPI_Alltoall。将你的实验与相关 MPI 通信函数做评测和对比。

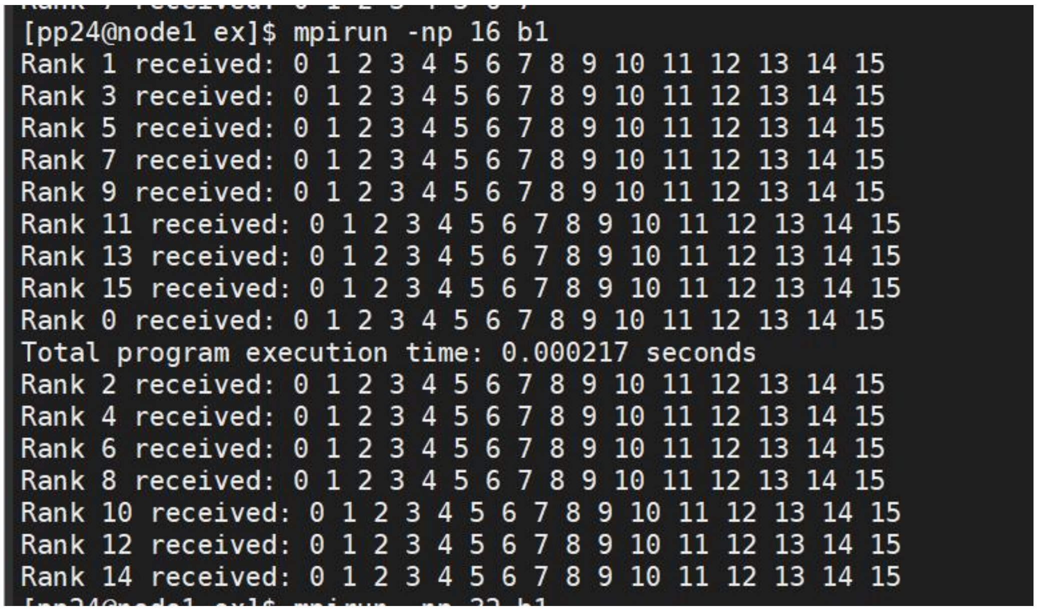
首先实现 mpi_send 和 mpi_recv

```
    对于 i 从 0 到 size-1
        如果 i 不等于当前进程的 rank
            // 发送数据给进程 i
            调用 MPI_Send(sendbuf, count, MPI_DATATYPE, i, tag, MPI_COMM_WORLD)

            // 接收来自进程 i 的数据
            调用 MPI_Recv(recvbuf[i * count : (i + 1) * count], count, MPI_DATATYPE, i, tag, MPI_COMM_WORLD, status)
        否则
            // 跳过自身，不需要发送和接收
        结束如果
    结束对于

    // 将接收到的数据复制到 recvbuf 的正确位置
    对于 i 从 0 到 size-1
        如果 i 不等于当前进程的 rank
            将 recvbuf[i * count : (i + 1) * count] 复制到 recvbuf[rank * count : (rank + 1) * count]
        否则
            // 本地数据已经在 recvbuf[rank * count : (rank + 1) * count] 中
        结束如果
    结束对于
结束
```

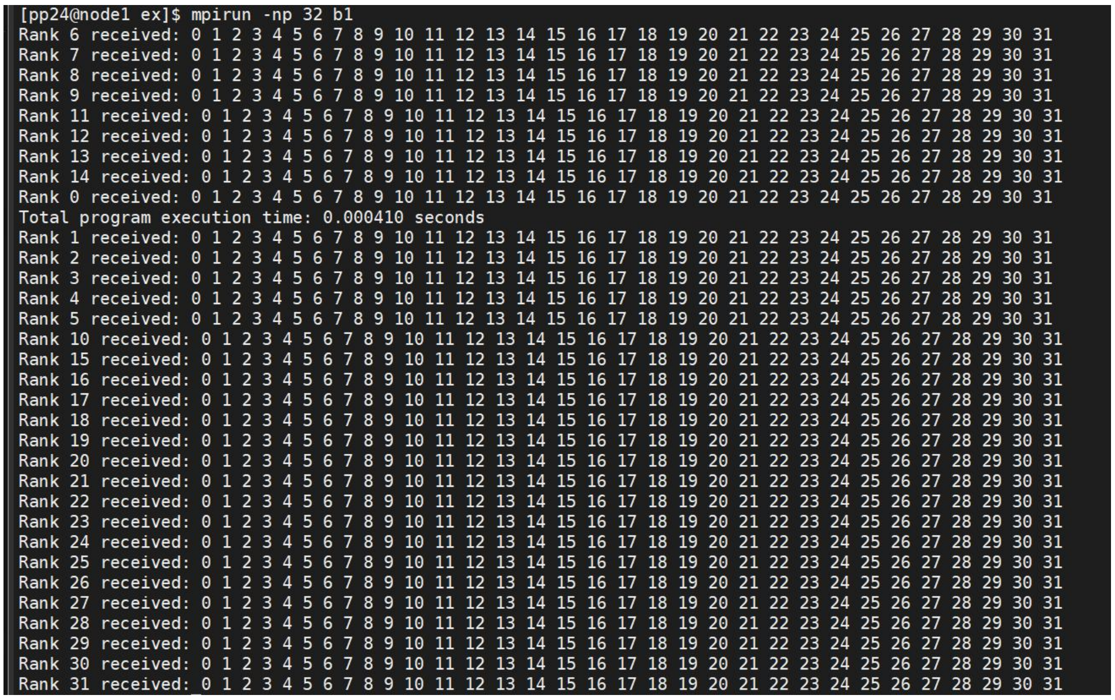
实验结果如下：分别使用 8 16 32



```
[pp24@node1 ex]$ mpirun -np 8 b1
Rank 0 received: 0 1 2 3 4 5 6 7
Total program execution time: 0.000163 seconds
Rank 1 received: 0 1 2 3 4 5 6 7
Rank 2 received: 0 1 2 3 4 5 6 7
Rank 3 received: 0 1 2 3 4 5 6 7
Rank 4 received: 0 1 2 3 4 5 6 7
Rank 5 received: 0 1 2 3 4 5 6 7
Rank 6 received: 0 1 2 3 4 5 6 7
Rank 7 received: 0 1 2 3 4 5 6 7
```

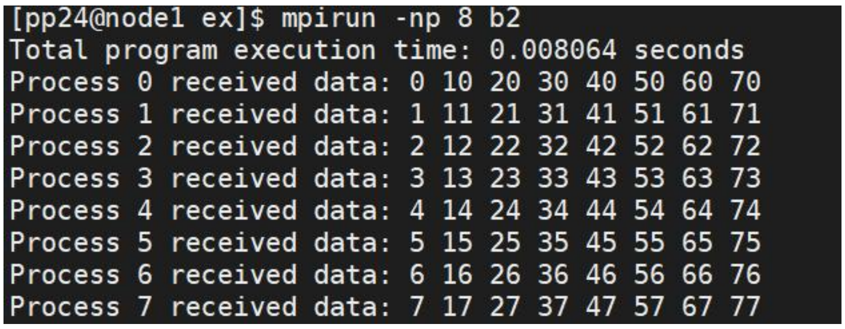


```
[pp24@node1 ex]$ mpirun -np 16 b1
Rank 1 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 3 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 5 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 7 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 9 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 11 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 13 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 15 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 0 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Total program execution time: 0.000217 seconds
Rank 2 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 4 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 6 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 8 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 10 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 12 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Rank 14 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

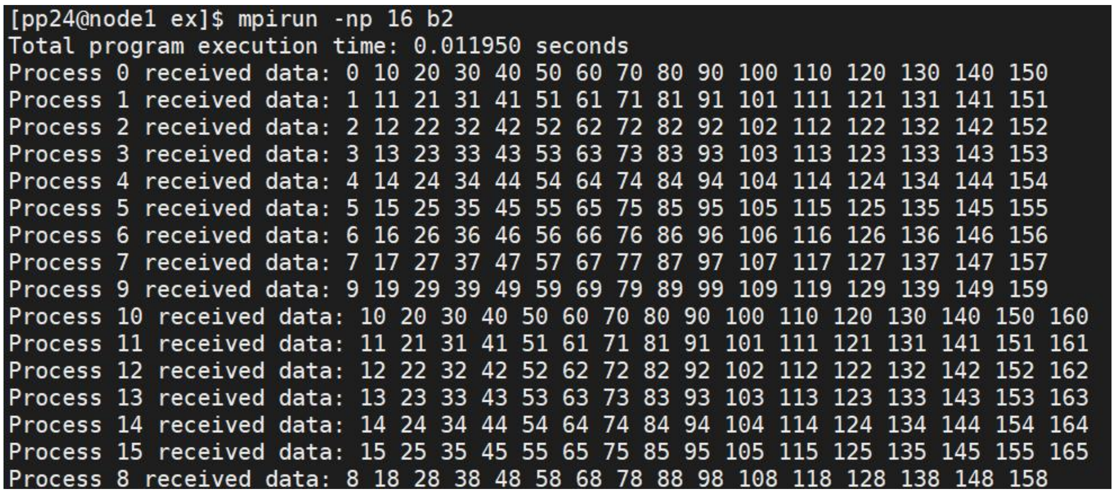


```
[pp24@node1 ex]$ mpirun -np 32 b1
Rank 6 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 7 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 8 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 9 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 11 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 12 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 13 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 14 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 0 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Total program execution time: 0.000410 seconds
Rank 1 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 2 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 3 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 4 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 5 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 10 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 15 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 16 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 17 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 18 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 19 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 20 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 21 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 22 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 23 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 24 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 25 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 26 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 27 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 28 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 29 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 30 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rank 31 received: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

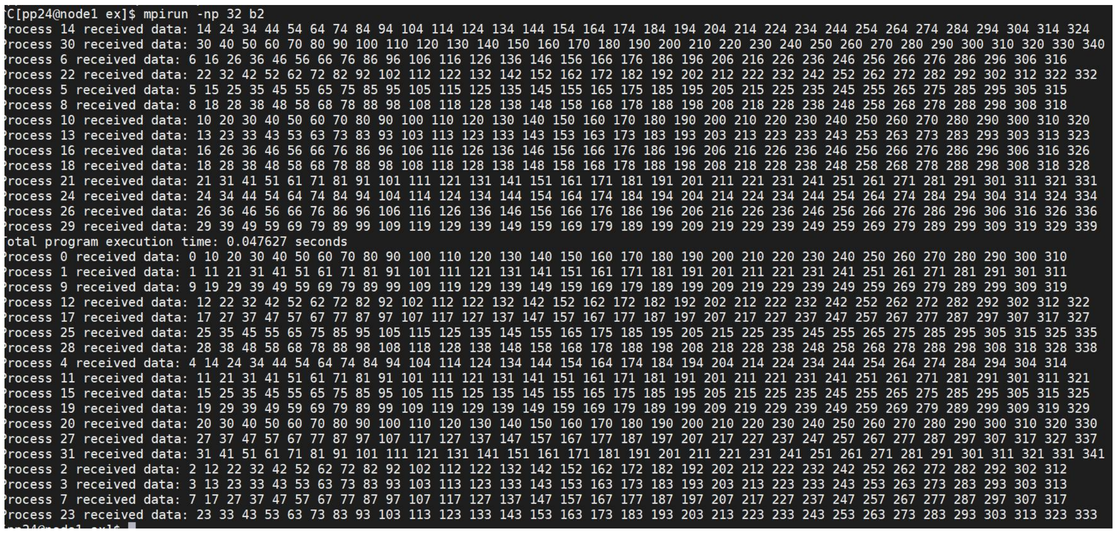
采用 ALLTOALL 实现



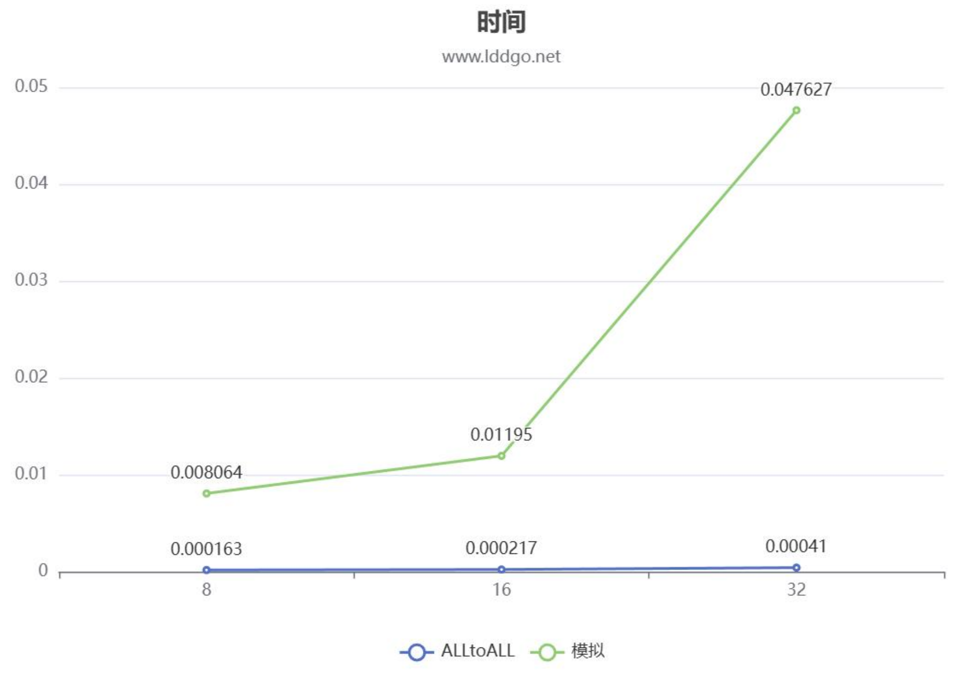
```
[pp24@node1 ex]$ mpirun -np 8 b2
Total program execution time: 0.008064 seconds
Process 0 received data: 0 10 20 30 40 50 60 70
Process 1 received data: 1 11 21 31 41 51 61 71
Process 2 received data: 2 12 22 32 42 52 62 72
Process 3 received data: 3 13 23 33 43 53 63 73
Process 4 received data: 4 14 24 34 44 54 64 74
Process 5 received data: 5 15 25 35 45 55 65 75
Process 6 received data: 6 16 26 36 46 56 66 76
Process 7 received data: 7 17 27 37 47 57 67 77
```



```
[pp24@node1 ex]$ mpirun -np 16 b2
Total program execution time: 0.011950 seconds
Process 0 received data: 0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150
Process 1 received data: 1 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151
Process 2 received data: 2 12 22 32 42 52 62 72 82 92 102 112 122 132 142 152
Process 3 received data: 3 13 23 33 43 53 63 73 83 93 103 113 123 133 143 153
Process 4 received data: 4 14 24 34 44 54 64 74 84 94 104 114 124 134 144 154
Process 5 received data: 5 15 25 35 45 55 65 75 85 95 105 115 125 135 145 155
Process 6 received data: 6 16 26 36 46 56 66 76 86 96 106 116 126 136 146 156
Process 7 received data: 7 17 27 37 47 57 67 77 87 97 107 117 127 137 147 157
Process 9 received data: 9 19 29 39 49 59 69 79 89 99 109 119 129 139 149 159
Process 10 received data: 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160
Process 11 received data: 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151 161
Process 12 received data: 12 22 32 42 52 62 72 82 92 102 112 122 132 142 152 162
Process 13 received data: 13 23 33 43 53 63 73 83 93 103 113 123 133 143 153 163
Process 14 received data: 14 24 34 44 54 64 74 84 94 104 114 124 134 144 154 164
Process 15 received data: 15 25 35 45 55 65 75 85 95 105 115 125 135 145 155 165
Process 8 received data: 8 18 28 38 48 58 68 78 88 98 108 118 128 138 148 158
```



```
C[pp24@node1 ex]$ mpirun -np 32 b2
rocess 14 received data: 14 24 34 44 54 64 74 84 94 104 114 124 134 144 154 164 174 184 194 204 214 224 234 244 254 264 274 284 294 304 314 324
rocess 30 received data: 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270 280 290 300 310 320 330 340
rocess 6 received data: 6 16 26 36 46 56 66 76 86 96 106 116 126 136 146 156 166 176 186 196 206 216 226 236 246 256 266 276 286 296 306 316
rocess 22 received data: 22 32 42 52 62 72 82 92 102 112 122 132 142 152 162 172 182 192 202 212 222 232 242 252 262 272 282 292 302 312 322 332
rocess 5 received data: 5 15 25 35 45 55 65 75 85 95 105 115 125 135 145 155 165 175 185 195 205 215 225 235 245 255 265 275 285 295 305 315
rocess 8 received data: 8 18 28 38 48 58 68 78 88 98 108 118 128 138 148 158 168 178 188 198 208 218 228 238 248 258 268 278 288 298 308 318
rocess 10 received data: 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270 280 290 300 310 320
rocess 13 received data: 13 23 33 43 53 63 73 83 93 103 113 123 133 143 153 163 173 183 193 203 213 223 233 243 253 263 273 283 293 303 313 323
rocess 16 received data: 16 26 36 46 56 66 76 86 96 106 116 126 136 146 156 166 176 186 196 206 216 226 236 246 256 266 276 286 296 306 316 326
rocess 18 received data: 18 28 38 48 58 68 78 88 98 108 118 128 138 148 158 168 178 188 198 208 218 228 238 248 258 268 278 288 298 308 318 328
rocess 21 received data: 21 31 41 51 61 71 81 91 101 111 121 131 141 151 161 171 181 191 201 211 221 231 241 251 261 271 281 291 301 311 321 331
rocess 24 received data: 24 34 44 54 64 74 84 94 104 114 124 134 144 154 164 174 184 194 204 214 224 234 244 254 264 274 284 294 304 314 324 334
rocess 26 received data: 26 36 46 56 66 76 86 96 106 116 126 136 146 156 166 176 186 196 206 216 226 236 246 256 266 276 286 296 306 316 326 336
rocess 29 received data: 29 39 49 59 69 79 89 99 109 119 129 139 149 159 169 179 189 199 209 219 229 239 249 259 269 279 289 299 309 319 329 339
otal program execution time: 0.047627 seconds
rocess 0 received data: 0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270 280 290 300 310
rocess 1 received data: 1 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151 161 171 181 191 201 211 221 231 241 251 261 271 281 291 301 311
rocess 9 received data: 9 19 29 39 49 59 69 79 89 99 109 119 129 139 149 159 169 179 189 199 209 219 229 239 249 259 269 279 289 299 309 319
rocess 12 received data: 12 22 32 42 52 62 72 82 92 102 112 122 132 142 152 162 172 182 192 202 212 222 232 242 252 262 272 282 292 302 312 322
rocess 17 received data: 17 27 37 47 57 67 77 87 97 107 117 127 137 147 157 167 177 187 197 207 217 227 237 247 257 267 277 287 297 307 317 327
rocess 25 received data: 25 35 45 55 65 75 85 95 105 115 125 135 145 155 165 175 185 195 205 215 225 235 245 255 265 275 285 295 305 315 325 335
rocess 28 received data: 28 38 48 58 68 78 88 98 108 118 128 138 148 158 168 178 188 198 208 218 228 238 248 258 268 278 288 298 308 318 328 338
rocess 4 received data: 4 14 24 34 44 54 64 74 84 94 104 114 124 134 144 154 164 174 184 194 204 214 224 234 244 254 264 274 284 294 304 314
rocess 11 received data: 11 21 31 41 51 61 71 81 91 101 111 121 131 141 151 161 171 181 191 201 211 221 231 241 251 261 271 281 291 301 311 321
rocess 15 received data: 15 25 35 45 55 65 75 85 95 105 115 125 135 145 155 165 175 185 195 205 215 225 235 245 255 265 275 285 295 305 315 325
rocess 19 received data: 19 29 39 49 59 69 79 89 99 109 119 129 139 149 159 169 179 189 199 209 219 229 239 249 259 269 279 289 299 309 319 329
rocess 20 received data: 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270 280 290 300 310 320 330
rocess 27 received data: 27 37 47 57 67 77 87 97 107 117 127 137 147 157 167 177 187 197 207 217 227 237 247 257 267 277 287 297 307 317 327 337
rocess 31 received data: 31 41 51 61 71 81 91 101 111 121 131 141 151 161 171 181 191 201 211 221 231 241 251 261 271 281 291 301 311 321 331 341
rocess 2 received data: 2 12 22 32 42 52 62 72 82 92 102 112 122 132 142 152 162 172 182 192 202 212 222 232 242 252 262 272 282 292 302 312
rocess 3 received data: 3 13 23 33 43 53 63 73 83 93 103 113 123 133 143 153 163 173 183 193 203 213 223 233 243 253 263 273 283 293 303 313
rocess 7 received data: 7 17 27 37 47 57 67 77 87 97 107 117 127 137 147 157 167 177 187 197 207 217 227 237 247 257 267 277 287 297 307 317
rocess 23 received data: 23 33 43 53 63 73 83 93 103 113 123 133 143 153 163 173 183 193 203 213 223 233 243 253 263 273 283 293 303 313 323 333
```



从性能分析图可以看出，模拟实现的时间较多，而采用库中 alltoall 的实现，时间较短，且随着进程数量的增加，模拟实现受进程数量影响较大，而库实现较小，时间增长少。

（3）蝶式求和：

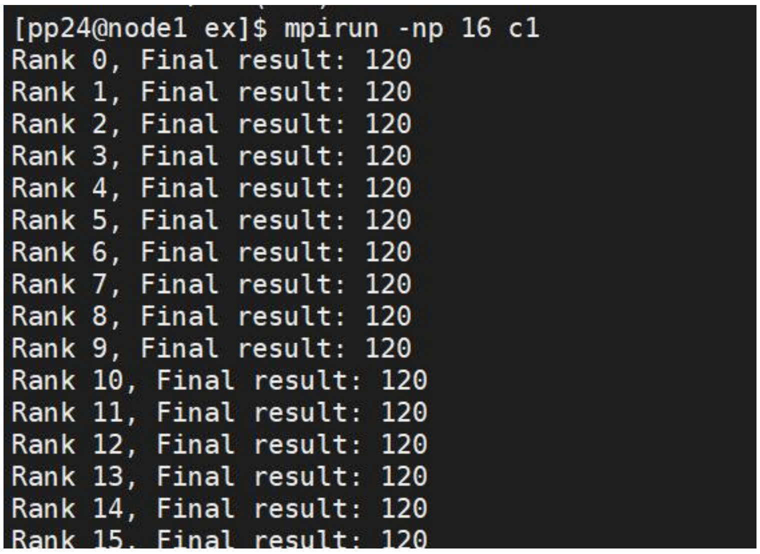
```
获取当前进程的秩（rank）
获取进程总数（nprocs）
初始化步长（step）为 1
当步长（step）小于进程总数（nprocs）时：
    计算通信伙伴的秩（partner），使用 XOR 运算确定
    如果通信伙伴的秩小于进程总数：
        如果当前进程的秩小于通信伙伴的秩：
            发送当前进程的数据给通信伙伴
            接收通信伙伴的数据
        否则：
```

接收通信伙伴的数据

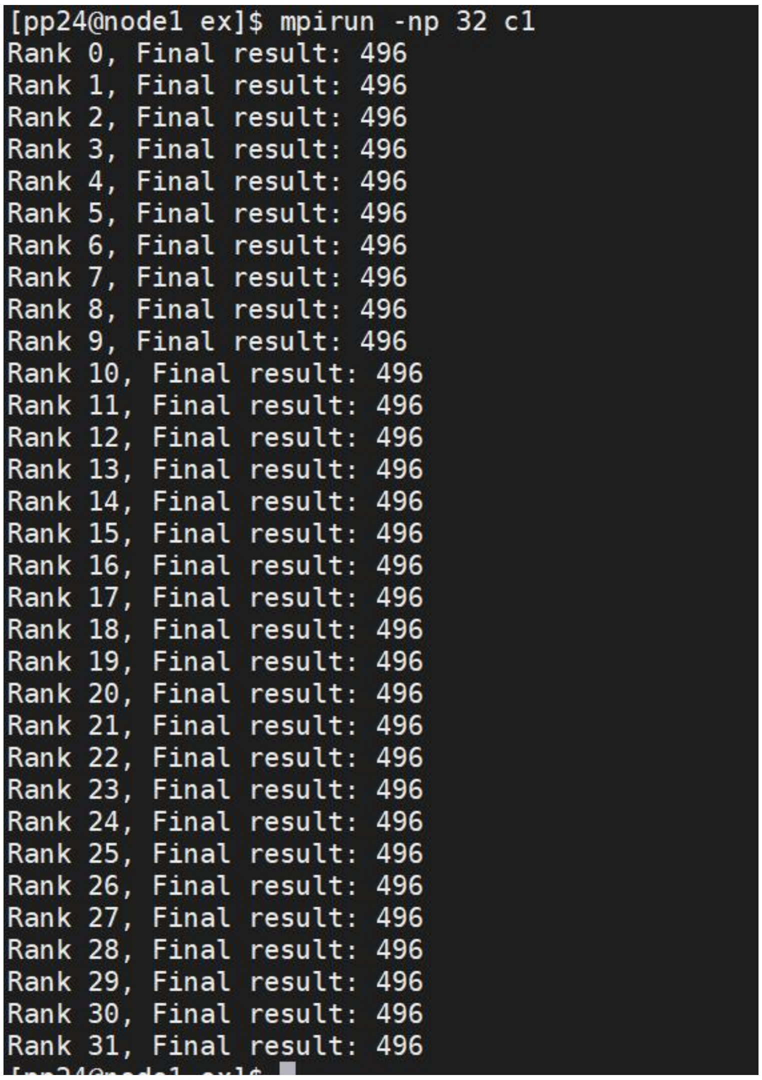
发送当前进程的数据给通信伙伴

执行归约操作（这里是加法），将接收到的数据加到当前进程的数据上

步长翻倍（step *= 2）



```
[pp24@node1 ex]$ mpirun -np 16 c1
Rank 0, Final result: 120
Rank 1, Final result: 120
Rank 2, Final result: 120
Rank 3, Final result: 120
Rank 4, Final result: 120
Rank 5, Final result: 120
Rank 6, Final result: 120
Rank 7, Final result: 120
Rank 8, Final result: 120
Rank 9, Final result: 120
Rank 10, Final result: 120
Rank 11, Final result: 120
Rank 12, Final result: 120
Rank 13, Final result: 120
Rank 14, Final result: 120
Rank 15, Final result: 120
```

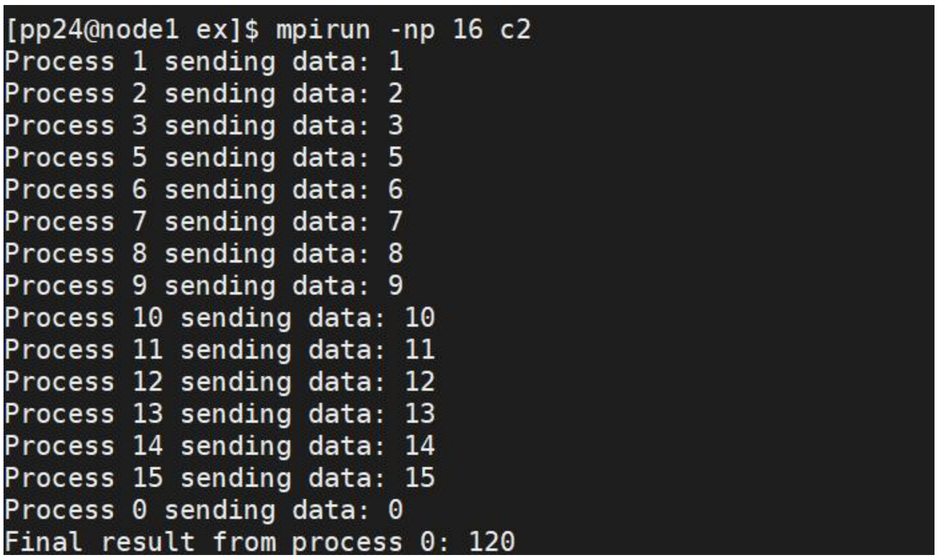

```
[pp24@node1 ex]$ mpirun -np 32 c1
Rank 0, Final result: 496
Rank 1, Final result: 496
Rank 2, Final result: 496
Rank 3, Final result: 496
Rank 4, Final result: 496
Rank 5, Final result: 496
Rank 6, Final result: 496
Rank 7, Final result: 496
Rank 8, Final result: 496
Rank 9, Final result: 496
Rank 10, Final result: 496
Rank 11, Final result: 496
Rank 12, Final result: 496
Rank 13, Final result: 496
Rank 14, Final result: 496
Rank 15, Final result: 496
Rank 16, Final result: 496
Rank 17, Final result: 496
Rank 18, Final result: 496
Rank 19, Final result: 496
Rank 20, Final result: 496
Rank 21, Final result: 496
Rank 22, Final result: 496
Rank 23, Final result: 496
Rank 24, Final result: 496
Rank 25, Final result: 496
Rank 26, Final result: 496
Rank 27, Final result: 496
Rank 28, Final result: 496
Rank 29, Final result: 496
Rank 30, Final result: 496
Rank 31, Final result: 496
```

二叉树求和：

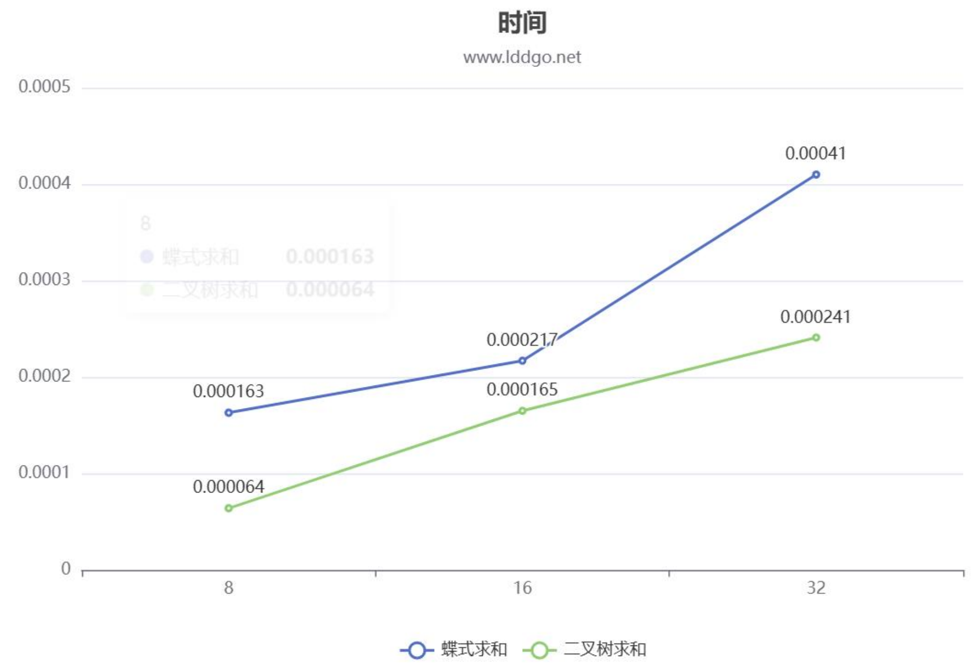
```
函数 group_wise_allreduce：
    输入参数：指向数据的指针（data），当前进程的秩（rank），进程总数（size），通信器（comm）

    初始化总数据（total_data）为当前进程的数据值

    对于每个步骤（step），从 1 开始，每次翻倍，直到小于进程总数：
        计算发送给其他进程的进程号（send_to）
        计算接收其他进程数据的进程号（recv_from）

        使用 MPI_Sendrecv 同时发送和接收数据：
            发送当前进程的总数据（total_data）给 send_to 进程
            接收 recv_from 进程发送的数据（recv_data）

        将接收到的数据（recv_data）加到总数据（total_data）上

    如果当前进程是根进程（rank == 0）：
```

输出最终结果

使用 MPI_Gather 将每个进程的最终结果发送到根进程

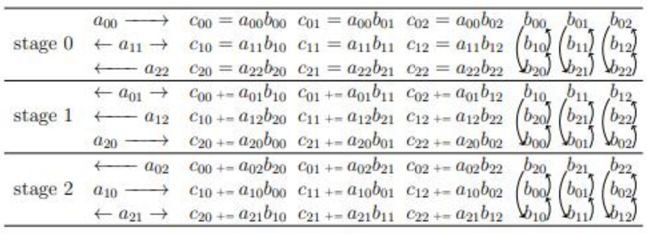


```
[pp24@node1 ex]$ mpirun -np 16 c2
Process 1 sending data: 1
Process 2 sending data: 2
Process 3 sending data: 3
Process 5 sending data: 5
Process 6 sending data: 6
Process 7 sending data: 7
Process 8 sending data: 8
Process 9 sending data: 9
Process 10 sending data: 10
Process 11 sending data: 11
Process 12 sending data: 12
Process 13 sending data: 13
Process 14 sending data: 14
Process 15 sending data: 15
Process 0 sending data: 0
Final result from process 0: 120
```



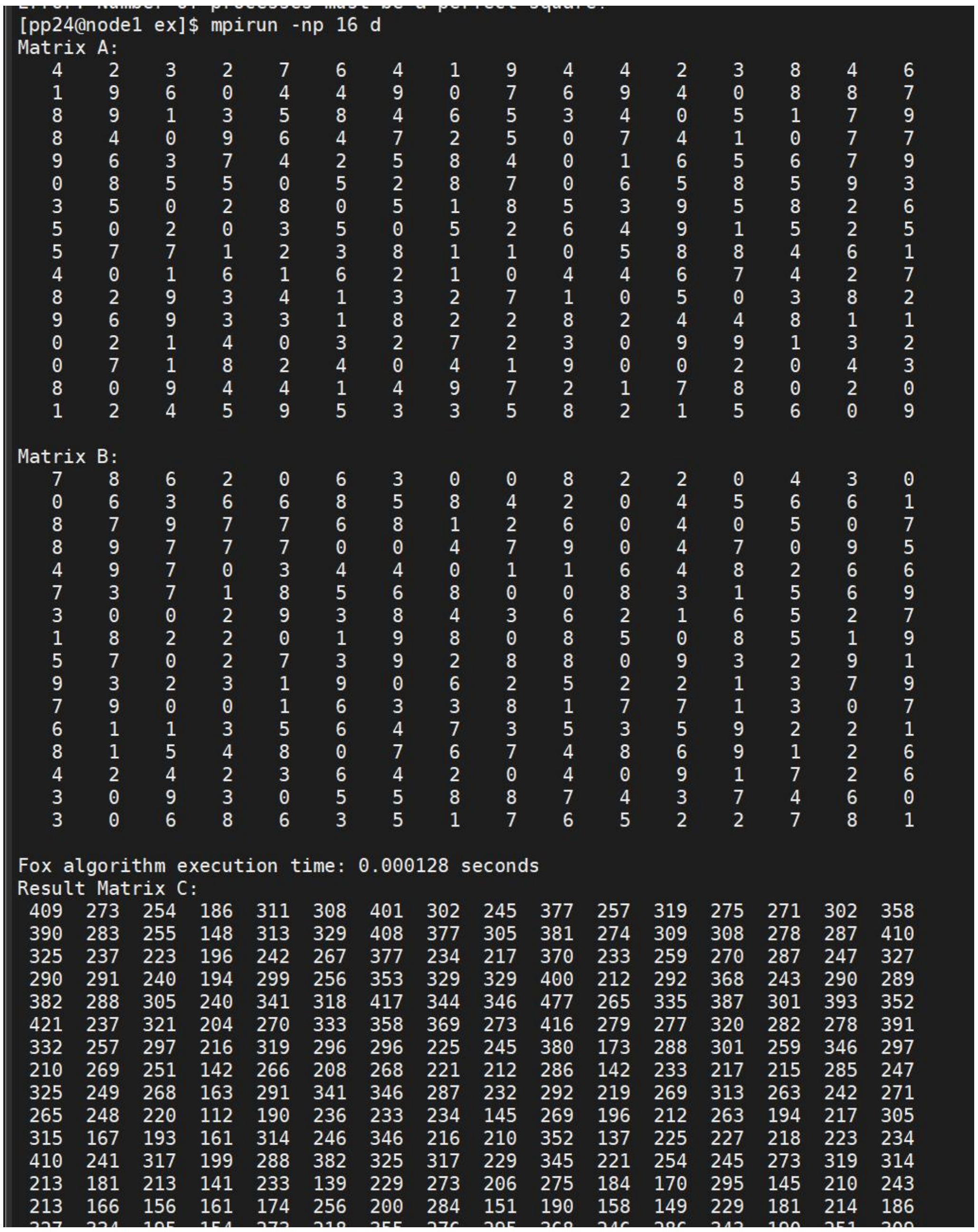
性能分析图如上，可以看出蝶式求和耗时较长，且随着进程数量增加，增长较大，猜测原因可能是因为蝶式求和的过程中进行的数据交换更多，而二叉树求和的数据交换量较少。

（4）FOX 矩阵乘法

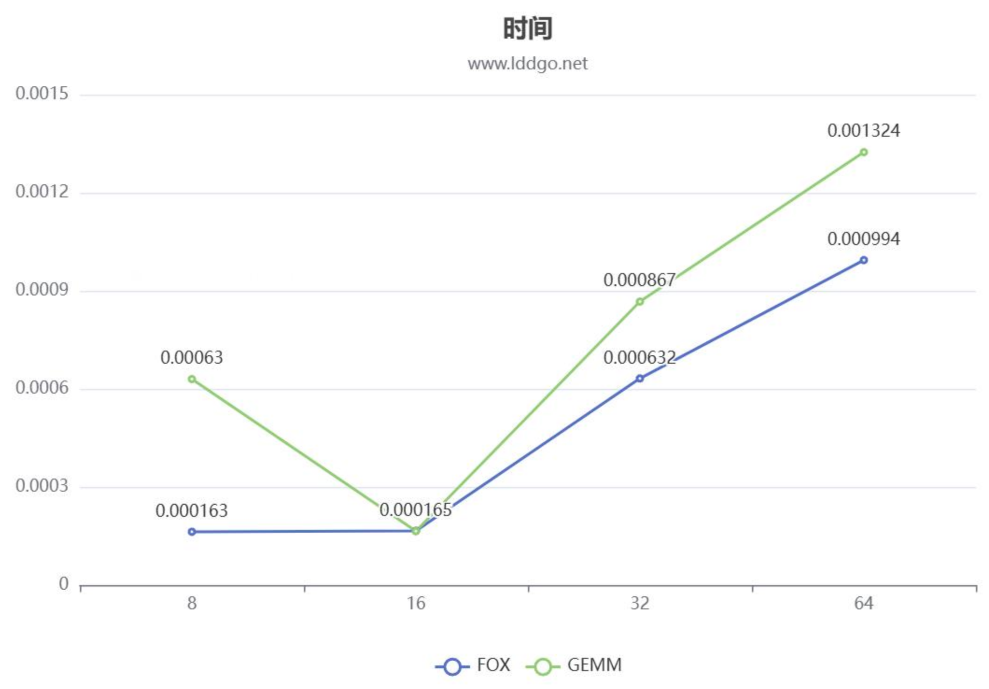


stage 0	$a_{00} \longrightarrow$	$c_{00} = a_{00}b_{00}$	$c_{01} = a_{00}b_{01}$	$c_{02} = a_{00}b_{02}$	b_{00}	b_{01}	b_{02}
	$\leftarrow a_{11} \rightarrow$	$c_{10} = a_{11}b_{10}$	$c_{11} = a_{11}b_{11}$	$c_{12} = a_{11}b_{12}$	b_{10}	b_{11}	b_{12}
	$\longleftarrow a_{22}$	$c_{20} = a_{22}b_{20}$	$c_{21} = a_{22}b_{21}$	$c_{22} = a_{22}b_{22}$	b_{20}	b_{21}	b_{22}
stage 1	$\leftarrow a_{01} \rightarrow$	$c_{00} += a_{01}b_{10}$	$c_{01} += a_{01}b_{11}$	$c_{02} += a_{01}b_{12}$	b_{10}	b_{11}	b_{12}
	$\longleftarrow a_{12}$	$c_{10} += a_{12}b_{20}$	$c_{11} += a_{12}b_{21}$	$c_{12} += a_{12}b_{22}$	b_{20}	b_{21}	b_{22}
	$a_{20} \longrightarrow$	$c_{20} += a_{20}b_{00}$	$c_{21} += a_{20}b_{01}$	$c_{22} += a_{20}b_{02}$	b_{00}	b_{01}	b_{02}
stage 2	$\longleftarrow a_{02}$	$c_{00} += a_{02}b_{20}$	$c_{01} += a_{02}b_{21}$	$c_{02} += a_{02}b_{22}$	b_{20}	b_{21}	b_{22}
	$a_{10} \longrightarrow$	$c_{10} += a_{10}b_{00}$	$c_{11} += a_{10}b_{01}$	$c_{12} += a_{10}b_{02}$	b_{00}	b_{01}	b_{02}
	$\leftarrow a_{21} \rightarrow$	$c_{20} += a_{21}b_{10}$	$c_{21} += a_{21}b_{11}$	$c_{22} += a_{21}b_{12}$	b_{10}	b_{11}	b_{12}

```
初始化 MPI 环境

为每个进程分配内存空间：
    a, b, c（局部矩阵块）
    如果 myrank 为 0，分配 A, B, C（全局矩阵）
如果 myrank 为 0：
    随机生成矩阵 A 和 B 的元素
    打印矩阵 A 和 B
对于每个进程：
    如果 myrank 为 0：
        将矩阵 A 和 B 划分为 sp * sp 个 n * n 的块
        将第 0 块的数据复制到 a 和 b
        将其他块的数据发送给对应的进程
    否则：
        接收对应的矩阵块 a 和 b
调用 Fox 算法函数：
    输入参数：a, b, c, sp, n, myrank
    在 Fox 算法开始前记录当前时间（start_time）

    对于每个轮次（round）：
        如果当前进程是本轮次的计算进程：
            将矩阵块 a 发送给其他进程
            将当前进程的矩阵块复制到临时变量 temp_a
        否则：
            从其他进程接收矩阵块 temp_a

        执行矩阵乘法，将结果存储在 c 中

        将矩阵块 b 发送给其他进程
        从其他进程接收矩阵块 temp_b
        将 temp_b 复制到 b
```



```
[pp24@node1 ex]$ mpirun -np 16 d
Matrix A:
   4    2    3    2    7    6    4    1    9    4    4    2    3    8    4    6
   1    9    6    0    4    4    9    0    7    6    9    4    0    8    8    7
   8    9    1    3    5    8    4    6    5    3    4    0    5    1    7    9
   8    4    0    9    6    4    7    2    5    0    7    4    1    0    7    7
   9    6    3    7    4    2    5    8    4    0    1    6    5    6    7    9
   0    8    5    5    0    5    2    8    7    0    6    5    8    5    9    3
   3    5    0    2    8    0    5    1    8    5    3    9    5    8    2    6
   5    0    2    0    3    5    0    5    2    6    4    9    1    5    2    5
   5    7    7    1    2    3    8    1    1    0    5    8    8    4    6    1
   4    0    1    6    1    6    2    1    0    4    4    6    7    4    2    7
   8    2    9    3    4    1    3    2    7    1    0    5    0    3    8    2
   9    6    9    3    3    1    8    2    2    8    2    4    4    8    1    1
   0    2    1    4    0    3    2    7    2    3    0    9    9    1    3    2
   0    7    1    8    2    4    0    4    1    9    0    0    2    0    4    3
   8    0    9    4    4    1    4    9    7    2    1    7    8    0    2    0
   1    2    4    5    9    5    3    3    5    8    2    1    5    6    0    9

Matrix B:
   7    8    6    2    0    6    3    0    0    8    2    2    0    4    3    0
   0    6    3    6    6    8    5    8    4    2    0    4    5    6    6    1
   8    7    9    7    7    6    8    1    2    6    0    4    0    5    0    7
   8    9    7    7    7    0    0    4    7    9    0    4    7    0    9    5
   4    9    7    0    3    4    4    0    1    1    6    4    8    2    6    6
   7    3    7    1    8    5    6    8    0    0    8    3    1    5    6    9
   3    0    0    2    9    3    8    4    3    6    2    1    6    5    2    7
   1    8    2    2    0    1    9    8    0    8    5    0    8    5    1    9
   5    7    0    2    7    3    9    2    8    8    0    9    3    2    9    1
   9    3    2    3    1    9    0    6    2    5    2    2    1    3    7    9
   7    9    0    0    1    6    3    3    8    1    7    7    1    3    0    7
   6    1    1    3    5    6    4    7    3    5    3    5    9    2    2    1
   8    1    5    4    8    0    7    6    7    4    8    6    9    1    2    6
   4    2    4    2    3    6    4    2    0    4    0    9    1    7    2    6
   3    0    9    3    0    5    5    8    8    7    4    3    7    4    6    0
   3    0    6    8    6    3    5    1    7    6    5    2    2    7    8    1

Fox algorithm execution time: 0.000128 seconds
Result Matrix C:
 409  273  254  186  311  308  401  302  245  377  257  319  275  271  302  358
 390  283  255  148  313  329  408  377  305  381  274  309  308  278  287  410
 325  237  223  196  242  267  377  234  217  370  233  259  270  287  247  327
 290  291  240  194  299  256  353  329  329  400  212  292  368  243  290  289
 382  288  305  240  341  318  417  344  346  477  265  335  387  301  393  352
 421  237  321  204  270  333  358  369  273  416  279  277  320  282  278  391
 332  257  297  216  319  296  296  225  245  380  173  288  301  259  346  297
 210  269  251  142  266  208  268  221  212  286  142  233  217  215  285  247
 325  249  268  163  291  341  346  287  232  292  219  269  313  263  242  271
 265  248  220  112  190  236  233  234  145  269  196  212  263  194  217  305
 315  167  193  161  314  246  346  216  210  352  137  225  227  218  223  234
 410  241  317  199  288  382  325  317  229  345  221  254  245  273  319  314
 213  181  213  141  233  139  229  273  206  275  184  170  295  145  210  243
 213  166  156  161  174  256  200  284  151  190  158  149  229  181  214  186
```



性能分析图如上，GEMM 代表不做任何优化的情况下执行矩阵乘法操作，在矩阵规模较小的情况下，此时计算量不大，使用并行算法反而增加了额外的通信成本，所以耗时较多，而在矩阵规模较大的情况下，将大规模的计算转化为小矩阵计算有利于增加访存效率，所以时间开始减少。

（5）

（e）　参数服务器系统的 MPI 模拟。

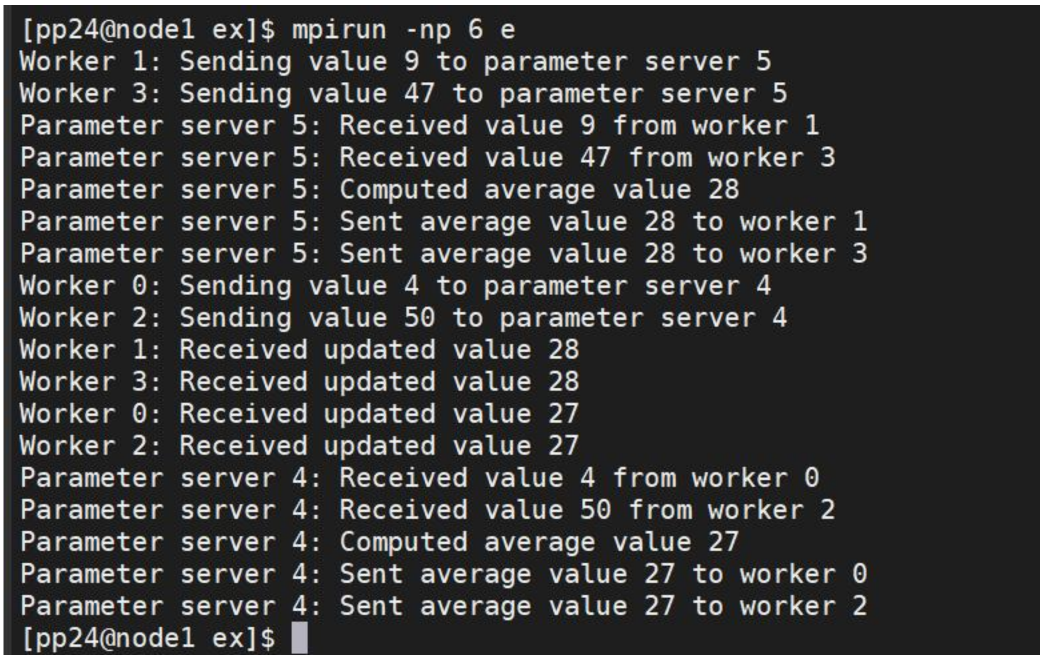
设系统中总计有 N 个进程，其中 P 个进程作为参数*服务器进程*，而 Q 个进程作为*工作进程*（N = P + Q，　且 0 < P << Q）。工作进程和服务器进程的互动过程如下：

1. **第 i 个工作进程首先产生一个随机数，发送给第 i%P 个参数服务器进程。然后等待并接收它对应的参数服务器进程发送更新后的数值，之后，再产生随机数，再发送……。**
2. **每个参数服务器进程等待并接收来自它对应的所有工作进程的数据，在此之后，经通信，使所有的参数服务器获得所有工作进程发送数据的平均值。**
3. **每个参数服务器发送该平均值给它对应的所有工作进程，然后再等待……。**

试给出上述互动过程的 MPI 程序实现。

工作者进程函数（worker_process）输入参数：当前进程的秩（rank），参数服务器的数量（num_servers）

```
计算对应的参数服务器进程的秩（server_rank）

初始化随机种子（使用当前时间和进程秩）

生成一个随机数（value）
打印工作者进程发送值的信息

向对应的参数服务器发送这个随机数
接收从参数服务器返回的更新后的值
打印工作者进程接收更新值的信息

参数服务器进程函数（parameter_server_process）：
复制
输入参数：当前进程的秩（rank），参数服务器的数量（num_servers）

计算分配给该参数服务器的工作者数量（num_workers）

分配内存以存储从工作者进程接收的值

对于每个分配给该参数服务器的工作者：
    从对应的工作者进程接收值
    打印参数服务器接收值的信息
    累加接收到的值

计算平均值（average_value）
打印参数服务器计算平均值的信息

对于每个分配给该参数服务器的工作者：
    向对应的工作者进程发送平均值
    打印参数服务器发送平均值的信息

释放分配的内存
```

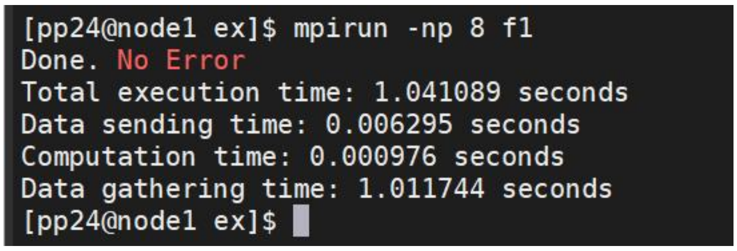


```
[pp24@node1 ex]$ mpirun -np 6 e
Worker 1: Sending value 9 to parameter server 5
Worker 3: Sending value 47 to parameter server 5
Parameter server 5: Received value 9 from worker 1
Parameter server 5: Received value 47 from worker 3
Parameter server 5: Computed average value 28
Parameter server 5: Sent average value 28 to worker 1
Parameter server 5: Sent average value 28 to worker 3
Worker 0: Sending value 4 to parameter server 4
Worker 2: Sending value 50 to parameter server 4
Worker 1: Received updated value 28
Worker 3: Received updated value 28
Worker 0: Received updated value 27
Worker 2: Received updated value 27
Parameter server 4: Received value 4 from worker 0
Parameter server 4: Received value 50 from worker 2
Parameter server 4: Computed average value 27
Parameter server 4: Sent average value 27 to worker 0
Parameter server 4: Sent average value 27 to worker 2
[pp24@node1 ex]$
```

（f）按行划分

```
初始化 MPI 环境

获取当前进程的秩（id_procs）和进程总数（num_procs）

定义常量 N 为 500，定义 IDX 宏以计算数组索引

分配内存空间：
    A（输入数组）
    B（输出数组）
    B2（用于验证的输出数组副本）

如果当前进程是最后一个进程（id_procs == num_procs - 1）：
    生成随机数组 A
    计算 B2 作为验证的正确结果

所有进程同步（MPI_Barrier）

如果当前进程是最后一个进程：
    将 A 数组分块发送给其他进程

否则：
    接收对应的数据块到 A 数组

如果当前进程不是最后一个进程：
    使用接收的数据块计算 B 数组的一部分
```

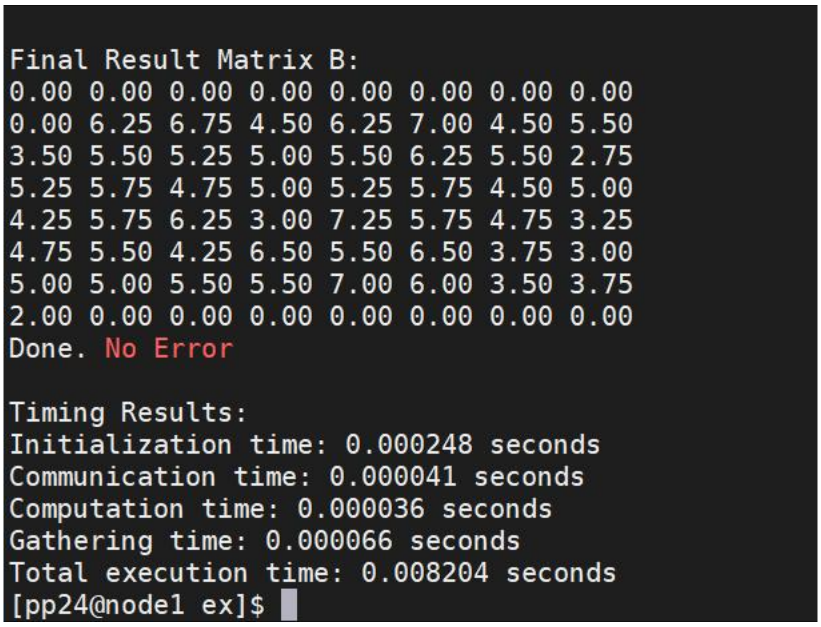
```
所有进程同步（MPI_Barrier）

如果当前进程是最后一个进程：
    收集其他进程计算的 B 数组的部分结果

否则：
    发送 B 数组的部分结果给最后一个进程

如果当前进程是最后一个进程：
    验证 B 数组和 B2 数组是否一致
    如果一致，打印"Done.No Error"
    否则，打印"Error Occured!"

释放内存空间：
    A
    B
    B2

结束 MPI 环境
```



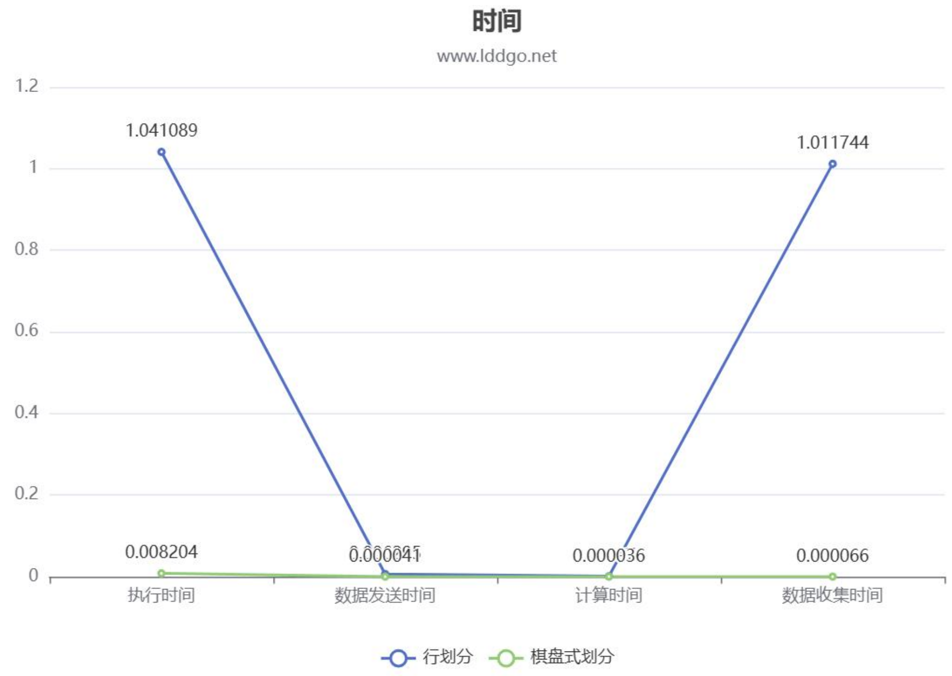
```
[pp24@node1 ex]$ mpirun -np 8 f1
Done. No Error
Total execution time: 1.041089 seconds
Data sending time: 0.006295 seconds
Computation time: 0.000976 seconds
Data gathering time: 1.011744 seconds
[pp24@node1 ex]$
```

```
按照棋盘式划分：初始化 MPI 环境

获取当前进程的秩（id_procs）和进程总数（num_procs）

定义常量 N 为 8，定义 IDX 宏以计算数组索引

定义函数：
    gen_rand_array：生成随机值填充矩阵
    compute：计算矩阵内部元素的平均值
    check_ans：检查结果矩阵是否正确
    print_matrix：打印矩阵

在主函数中：
    分配内存空间：
        A（输入矩阵）
        B（计算后的矩阵）
        B2（用于验证的矩阵副本）
```

```
如果当前进程是根进程（id_procs == 0）：
    使用 gen_rand_array 生成矩阵 A 的随机值
    使用 compute 计算矩阵 B2 作为验证的正确结果
    打印初始矩阵 A

所有进程同步（MPI_Barrier）

定义子矩阵类型（SubMat）用于广播
提交子矩阵类型（SubMat）

如果当前进程是根进程：
    广播矩阵 A 的子区域给其他进程
否则：
    接收根进程广播的子区域矩阵 A

计算子区域矩阵 B
打印计算后的子区域矩阵 B

定义子矩阵类型（SubMat_B）用于收集结果
提交子矩阵类型（SubMat_B）

如果当前进程是根进程：
    收集其他进程计算的子区域矩阵 B
否则：
    发送计算的子区域矩阵 B 给根进程

如果当前进程是根进程：
    打印最终结果矩阵 B
    使用 check_ans 验证 B 矩阵和 B2 矩阵是否一致
    如果一致，打印"Done. No Error"
    否则，打印"Error!"

结束 MPI 环境
```



```
Final Result Matrix B:
0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 6.25 6.75 4.50 6.25 7.00 4.50 5.50
3.50 5.50 5.25 5.00 5.50 6.25 5.50 2.75
5.25 5.75 4.75 5.00 5.25 5.75 4.50 5.00
4.25 5.75 6.25 3.00 7.25 5.75 4.75 3.25
4.75 5.50 4.25 6.50 5.50 6.50 3.75 3.00
5.00 5.00 5.50 5.50 7.00 6.00 3.50 3.75
2.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
Done. No Error

Timing Results:
Initialization time: 0.000248 seconds
Communication time: 0.000041 seconds
Computation time: 0.000036 seconds
Gathering time: 0.000066 seconds
Total execution time: 0.008204 seconds
[pp24@node1 ex]$
```



性能分析如下：可以看出行划分和棋盘式划分在数据发送和计算上的用时差不多，主要差距体现在数据收集的时候，此时棋盘式划分的用时明显更短，推测可能是因为棋盘式划分在通信传递信息的时候，进程网格在物理存储上相隔较近，所以耗时较少。