



南开大学
Nankai University

《并行程序设计》实验报告

(2023~2024 学年第一学期)

实验名称: MPI 编程

学 院: 软件学院

姓 名: 陈高楠

学 号: 2112966

指导老师: 孙永谦

2023 年 12 月 19 日

目录

1 梯形积分法	1
1 问题描述	1
2 算法设计与分析	1
2.1 静态分配	1
2.2 动态分配	3
2.3 动态分配粗粒度	5
3 结果统计与分析	8
3.1 结果正确性	8
3.2 梯形个数对不同分配方式运行时间影响	9
3.3 粒度粗细对运行时间的影响	10
2 “多个数组排序”的任务不均衡案例	1
1 问题描述	1
2 算法设计与分析	1
2.1 静态分配	1
2.2 动态分配	3
2.3 动态粗粒度分配	7
3 结果统计与分析	10
3.1 不同分配方式对计算结果影响	10
3.2 不同分配粒度对计算结果影响	11
3 高斯消元	13
1 问题描述	13
2 算法设计与分析	13
2.1 静态分配	13
2.2 动态分配	15
3 结果统计与分析	18
3.1 MPI 任务分配方式运行时间比较	18
3.2 MPI 与 OpenMP, Pthread 的比较	18

实验一 梯形积分法

1 问题描述

实现第 5 章课件中的梯形积分法的 MPI 编程熟悉并掌握 MPI 编程方法，规模自行设定，可探讨不同规模对不同实现方式的影响。

2 算法设计与分析

2.1 静态分配

采用静态分配来进行梯形积分的运算，将积分分段平均分配给每一个进程。相关代码如下：

```
1 // 计算局部和
2 double Trap(double a, double b, int count, double h) {
3     double my_result, x;
4     int i;
5     my_result = (f(a) + f(b)) / 2.0;
6     // 局部求和，避免过多通信
7     for (i = 1; i < count; i++) {
8         x = a + i * h;
9         my_result += f(x);
10    }
11    my_result *= h;
12    return my_result;
13 }
14 // 主函数
15 int main(int argc, char* argv[])
16 {
17     int my_rank, comm_sz, n = atoi(argv[1]);
18     double a = 1, b = 100;
```

```
19     double h = (b - a) / n;
20     double global_result;
21     double start, end;
22     double time;
23
24     MPI_Init(NULL, NULL);
25     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
26     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
27     //printf("%li \n",my_rank);
28     //0号进程开始计时
29     if (my_rank == 0) {
30         start = MPI_Wtime();
31     }
32     int local_n = n / comm_sz;
33     double local_a = a + my_rank * local_n * h;
34     double local_b = local_a + local_n * h;
35     //将未除尽的任务数分配给最后一个进程
36     if (my_rank == comm_sz - 1) {
37         local_n = n - (comm_sz - 1) * local_n;
38         local_b = b;
39     }
40     //每个进程计算自身分配的任务的局部和
41     double local_result = Trap(local_a, local_b, local_n, h);
42     //printf("%.15e\n",local_result);
43     //其他进程将局部和发送给0号进程
44     if (my_rank != 0) {
45         MPI_Send(&local_result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
46     }
47     //0号进程收集局部和并求全局和
48     else {
49         global_result = local_result;
50         for (int source = 1; source < comm_sz; source++)
51         {
52             MPI_Recv(&local_result, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD..
53                     , MPI_STATUS_IGNORE);
54             global_result += local_result;
55         }
```

```
55         //完成计算，0号进程停止计时
56         end = MPI_Wtime();
57         time = end - start;
58     }
59     //0号进程打印出结果
60     if (my_rank == 0) {
61         printf("划分成小梯形的块数: %d\n", n);
62         printf("计算结果是: %.15e\n", global_result);
63         printf("总共耗时: %f s\n ", time);
64     }
65
66     MPI_Finalize();
67     return 0;
68 }
```

2.2 动态分配

采用动态分配来进行梯形积分的运算，每一次分配一个任务给一个进程，进程计算完成后则继续分配。相关代码如下：

```
1 double Trap(double a, double b, int count, double h) {
2     double my_result = (f(a) + f(b)) / 2.0;
3     for (int i = 1; i < count; i++) {
4         double x = a + i * h;
5         my_result += f(x);
6     }
7     return my_result * h;
8 }
9
10 int main(int argc, char* argv[]) {
11     int my_rank, comm_sz, n = atoi(argv[1]);
12     double a = 1, b = 100;
13     double h = (b - a) / n;
14     double global_result = 0.0;
15     double start, end;
16 }
```

```
17 MPI_Init(NULL, NULL);
18 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
19 MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
20
21 if (my_rank == 0) {
22     start = MPI_Wtime();
23     int tasks_completed = 0;
24     double local_result;
25
26     // 分发初始任务
27     for (int i = 1; i < comm_sz; i++) {
28         MPI_Send(&tasks_completed, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
29         tasks_completed++;
30     }
31
32     while (tasks_completed < n) {
33         MPI_Status status;
34         MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ...
35                 MPI_ANY_TAG, MPI_COMM_WORLD, &status);
36         global_result += local_result;
37
38         MPI_Send(&tasks_completed, 1, MPI_INT, status.MPI_SOURCE, 0, ...
39                 MPI_COMM_WORLD);
40         tasks_completed++;
41     }
42
43     // 告诉所有进程任务已完成
44     for (int i = 1; i < comm_sz; i++) {
45         int stop_signal = -1;
46         MPI_Send(&stop_signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
47     }
48
49     // 收集最后的任务
50     for (int i = 1; i < comm_sz; i++) {
51         MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ...
52                 MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
53         global_result += local_result;
```

```
51     }
52
53     end = MPI_Wtime();
54 } else {
55     while (1) {
56         int task;
57         MPI_Recv(&task, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, ...
58                 MPI_STATUS_IGNORE);
59         if (task == -1) break; // 停止信号
60
61         double local_a = a + task * h;
62         double local_b = local_a + h;
63         double local_result = Trap(local_a, local_b, 1, h);
64         MPI_Send(&local_result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
65     }
66
67     if (my_rank == 0) {
68         printf("划分成小梯形的块数: %d\n", n);
69         printf("计算结果是: %.15e\n", global_result);
70         printf("总共耗时: %f s\n", end - start);
71     }
72
73     MPI_Finalize();
74     return 0;
75 }
```

2.3 动态分配粗粒度

采用动态粗粒度分配来进行梯形积分的运算，每一次分配若干个任务给一个进程，进程计算完成后则继续分配。这里每次分配的任务数可指定。相关代码如下：

```
1 double Trap(double a, double b, int count, double h) {
2     double my_result = (f(a) + f(b)) / 2.0;
3     for (int i = 1; i < count; i++) {
4         double x = a + i * h;
```

```
5         my_result += f(x);
6     }
7     return my_result * h;
8 }
9
10 int main(int argc, char* argv[]) {
11     int my_rank, comm_sz, n = atoi(argv[1]);
12     double a = 1, b = 100;
13     double h = (b - a) / n;
14     double global_result = 0.0;
15     double start, end;
16     int TASK_SIZE = atoi(argv[2]);
17     MPI_Init(NULL, NULL);
18     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
19     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
20
21     if (my_rank == 0) {
22         start = MPI_Wtime();
23         int tasks_completed = 0;
24         double local_result;
25
26         // 分发初始任务
27         for (int i = 1; i < comm_sz; i++) {
28             MPI_Send(&tasks_completed, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
29             tasks_completed += TASK_SIZE;
30         }
31
32         while (tasks_completed < n) {
33             MPI_Status status;
34             MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ...
35                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
36             global_result += local_result;
37             if (tasks_completed + TASK_SIZE > n) {
38                 MPI_Send(&tasks_completed, 1, MPI_INT, status.MPI_SOURCE, 0, ...
39                         MPI_COMM_WORLD);
38                 tasks_completed = n; // 避免超出范围
39             } else {
```



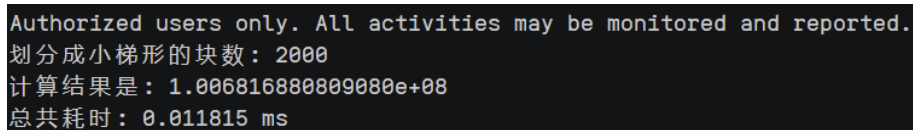
```
40         MPI_Send(&tasks_completed, 1, MPI_INT, status.MPI_SOURCE, 0,...
               MPI_COMM_WORLD);
41         tasks_completed += TASK_SIZE;
42     }
43 }
44
45 // 告诉所有进程任务已完成
46 for (int i = 1; i < comm_sz; i++) {
47     int stop_signal = -1;
48     MPI_Send(&stop_signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
49 }
50
51 // 收集最后的任务
52 for (int i = 1; i < comm_sz; i++) {
53     MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ...
               MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
54     global_result += local_result;
55 }
56
57 end = MPI_Wtime();
58 } else {
59     while (1) {
60         int task;
61         MPI_Recv(&task, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, ...
               MPI_STATUS_IGNORE);
62         if (task == -1) break; // 停止信号
63
64         double local_a = a + task * h;
65         double local_b = local_a + h*TASK_SIZE;
66         if (local_b >= b) {
67             local_b = b;
68             double local_result = Trap(local_a, local_b, (local_b - local_a) / h...
               + 1, h);
69             MPI_Send(&local_result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
70         }
71         else {
72             double local_result = Trap(local_a, local_b, TASK_SIZE, h);
```

```
73         MPI_Send(&local_result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);}
74     }
75 }
76
77 if (my_rank == 0) {
78     printf("划分成小梯形的块数: %d\n", n);
79     printf("计算结果是: %.15e\n", global_result);
80     printf("总共耗时: %f s\n", end - start);
81 }
82
83 MPI_Finalize();
84 return 0;
85 }
```

3 结果统计与分析

3.1 结果正确性

将积分函数设为 $f(x) = 4x^3 + 2x^2 + 3x$, 划分一共 2000 个梯形, 积分区间从 1 到 100, 测试结果正确性。



```
Authorized users only. All activities may be monitored and reported.
划分成小梯形的块数: 2000
计算结果是: 1.006816880809080e+08
总共耗时: 0.011815 ms
```

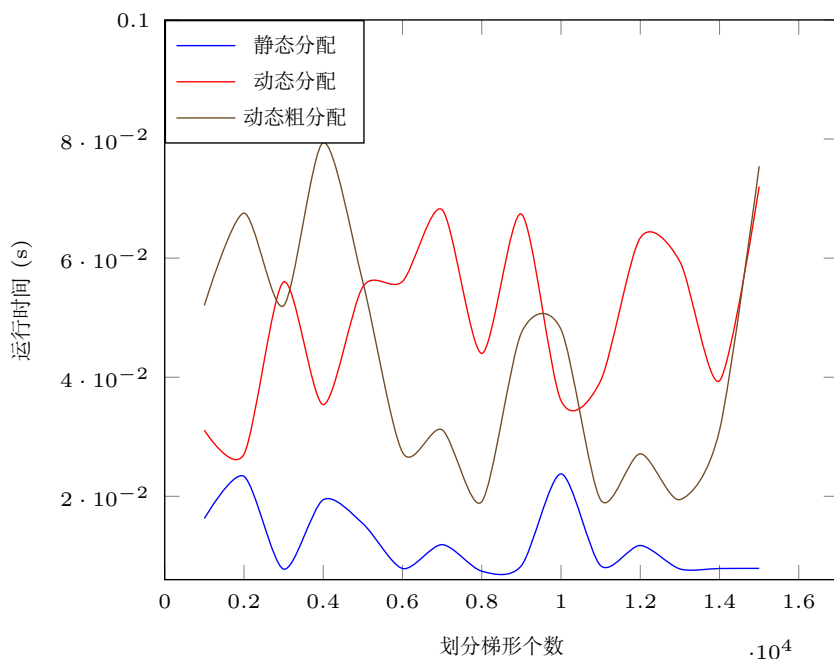
图 1.1: 结果正确性

经过验证, 实验结果正确。

3.2 梯形个数对不同分配方式运行时间影响

划分梯形个数	静态分配	动态分配	动态粗分配
1000	0.016254	0.031094	0.052052
2000	0.023353	0.027063	0.06755
3000	0.007772	0.055996	0.051994
4000	0.019421	0.035392	0.079391
5000	0.015418	0.055183	0.055981
6000	0.007873	0.056073	0.027398
7000	0.011881	0.068118	0.031169
8000	0.007422	0.043994	0.019089
9000	0.008351	0.067402	0.047442
10000	0.023764	0.036062	0.04802
11000	0.008354	0.039419	0.0194
12000	0.011737	0.063405	0.027147
13000	0.007800	0.059412	0.01944
14000	0.007876	0.039399	0.031095
15000	0.007895	0.072012	0.075421

表 1.1: 运行时间比较



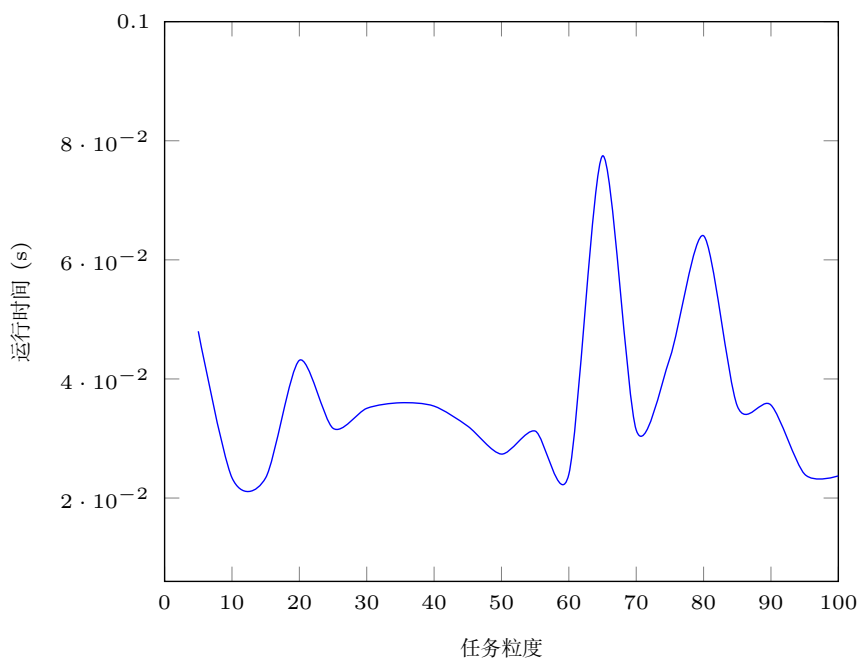
从图中可知，划分梯形的大小与计算时间的长短并没有十分严格的比例关系，这是因为在并行计算中，通信和协调的开销可能对总体计算时间有显著影响。即使计算任务本身不是很复杂，但进程间的通信和数据同步可能导致额外的时间开销。

尽管如此，我们仍然可以很清晰地看出，采用动态分配所耗费的时间明显大于静态分配

所耗费的时间，这是因为动态分配耗费了更多的时间在通信上，导致运行时间变大。而采用粗粒度分配比细粒度分配更好一点，因为它减少了通信的开销。

3.3 粒度粗细对运行时间的影响

这里以分 1000 个梯形为例探讨任务动态分配时分配任务粒度粗细对运行时间的影响。



从图中可知，当粒度分配合理时执行效率会变快，当任务分配粒度为 15 时能够达到比较好的效果。推测当粒度分配合理时能够减少不同主机之间通信的开销，从而加快计算速率。

实验二 “多个数组排序”的任务不均衡案例

1 问题描述

对于课件中“多个数组排序”的任务不均衡案例进行 MPI 编程实现，规模可自己设定、调整。

2 算法设计与分析

2.1 静态分配

采用静态分配来进行多个数组排序的运算，将数组平均分配给每一个进程，然后每个进程分别进行计算。相关代码如下：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <mpi.h>
5
6 #define ARR_NUM 10000
7 #define ARR_LEN 10000
8 #define PROCESS_NUM 6
9 #define SEG (ARR_NUM / PROCESS_NUM)
10
11 int arr[ARR_NUM][ARR_LEN];
12
13 void init_2(void) {
14     int i, j, ratio;
15     srand((unsigned)time(NULL));
16     for (i = 0; i < ARR_NUM; i++) {
17         if (i < SEG) ratio = 0;
18         else if (i < SEG * 2) ratio = 32;
```

```
19     else ratio = 128;
20
21     if ((rand() & 127) < ratio) {
22         for (j = 0; j < ARR_LEN; j++) {
23             arr[i][j] = ARR_LEN - j;
24         }
25     } else {
26         for (j = 0; j < ARR_LEN; j++) {
27             arr[i][j] = j;
28         }
29     }
30 }
31 }
32
33 int compare_ints(const void* a, const void* b) {
34     int arg1 = *(const int*)a;
35     int arg2 = *(const int*)b;
36
37     if (arg1 < arg2) return -1;
38     if (arg1 > arg2) return 1;
39     return 0;
40 }
41
42 int main() {
43     int my_rank, comm_sz;
44     double start, end;
45     double time;
46
47     // Initialize the array
48     init_2();
49
50     MPI_Init(NULL, NULL);
51     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
52     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
53
54     // Allocate tasks based on process number
55     int startTask = my_rank * SEG;
```

```
56     int endTask = startTask + SEG;
57
58     // Sorting
59     start = MPI_Wtime();
60     for (int i = startTask; i < endTask && i < ARR_NUM; i++) {
61         qsort(arr[i], ARR_LEN, sizeof(int), compare_ints);
62     }
63     end = MPI_Wtime();
64     time = end - start;
65
66     // Other processes send their runtime to process 0
67     if (my_rank != 0) {
68         MPI_Send(&time, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
69     } else { // Process 0 prints the runtime of all processes
70         printf("各进程运行时间如下:\n");
71         printf("0号进程: %fs\n", time);
72         for (int source = 1; source < comm_sz; source++) {
73             MPI_Recv(&time, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, ...
74                     MPI_STATUS_IGNORE);
75             printf("%d号进程: %fs\n", source, time);
76         }
77
78     MPI_Finalize();
79     return 0;
80 }
```

2.2 动态分配

采用动态分配来进行多个数组排序的运算，每次分配一个数组给每一个进程进行排序，然后每个进程分别进行计算。计算完后重复分配任务一直到数组分配完毕。相关代码如下：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <mpi.h>
```

```
5
6 #define ARR_NUM 10000
7 #define ARR_LEN 10000
8 #define PROCESS_NUM 6
9 #define SEG (ARR_NUM / PROCESS_NUM)
10 int arr[ARR_NUM][ARR_LEN];
11
12 void init_2(void) {
13     int i, j, ratio;
14     srand((unsigned)time(NULL));
15     for (i = 0; i < ARR_NUM; i++) {
16         if (i < SEG) ratio = 0;
17         else if (i < SEG * 2) ratio = 32;
18         else ratio = 128;
19
20         if ((rand() & 127) < ratio) {
21             for (j = 0; j < ARR_LEN; j++) {
22                 arr[i][j] = ARR_LEN - j;
23             }
24         } else {
25             for (j = 0; j < ARR_LEN; j++) {
26                 arr[i][j] = j;
27             }
28         }
29     }
30 }
31
32 int compare_ints(const void* a, const void* b) {
33     int arg1 = *(const int*)a;
34     int arg2 = *(const int*)b;
35
36     if (arg1 < arg2) return -1;
37     if (arg1 > arg2) return 1;
38     return 0;
39 }
40
41 int main() {
```



```
42     int my_rank, comm_sz;
43     double start, end;
44     double time;
45
46     // Initialize the array
47     init_2();
48
49     MPI_Init(NULL, NULL);
50     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
51     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
52
53     if (my_rank == 0) {
54         start = MPI_Wtime();
55         int tasks_completed = 0;
56         double local_result=0;
57         // 分发初始任务
58         for (int i = 1; i < comm_sz; i++) {
59             MPI_Send(&tasks_completed, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
60             tasks_completed++;
61         }
62
63         while (tasks_completed < ARR_NUM) {
64             MPI_Status status;
65             MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ...
66                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
67             MPI_Send(&tasks_completed, 1, MPI_INT, status.MPI_SOURCE, 0, ...
68                     MPI_COMM_WORLD);
69             tasks_completed++;
70         }
71
72         // 告诉所有进程任务已完成
73         for (int i = 1; i < comm_sz; i++) {
74             int stop_signal = -1;
75             MPI_Send(&stop_signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
76         }
77
78         // 收集最后的任务
```

```
77     for (int i = 1; i < comm_sz; i++) {
78         MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ...
79             MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
80     }
81     end = MPI_Wtime();
82     time=end-start;
83     // printf("各进程运行时间如下:\n");
84     printf("0号进程: %f s\n", time);
85     // for (int source = 1; source < comm_sz; source++) {
86     //     MPI_Recv(&time, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, ...
87         MPI_STATUS_IGNORE);
88     //     printf("%d号进程: %f s\n", source, time);
89     // }
90 }
91 else {
92     double start1=MPI_Wtime();
93     while (1) {
94         int task;
95         double local_result=0;
96         MPI_Recv(&task, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, ...
97             MPI_STATUS_IGNORE);
98         if (task == -1) break; // 停止信号
99         qsort(arr[task], ARR_LEN, sizeof(int), compare_ints);
100         MPI_Send(&local_result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
101     }
102     double end1 = MPI_Wtime();
103     double time1=end1-start1;
104     printf("%d号进程: %f s\n", my_rank, time1);
105     // MPI_Send(&time1, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
106 }
107
108 MPI_Finalize();
109 return 0;
110 }
```

2.3 动态粗粒度分配

采用动态分配来进行多个数组排序的运算，每次分配若干个数组给每一个进程进行排序，然后每个进程分别进行计算。计算完后重复分配任务一直到数组分配完毕。相关代码如下：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <mpi.h>
5
6 #define ARR_NUM 10000
7 #define ARR_LEN 10000
8 #define PROCESS_NUM 6
9 #define SEG (ARR_NUM / PROCESS_NUM)
10
11 int arr[ARR_NUM][ARR_LEN];
12
13 void init_2(void) {
14     int i, j, ratio;
15     srand((unsigned)time(NULL));
16     for (i = 0; i < ARR_NUM; i++) {
17         if (i < SEG) ratio = 0;
18         else if (i < SEG * 2) ratio = 32;
19         else ratio = 128;
20
21         if ((rand() & 127) < ratio) {
22             for (j = 0; j < ARR_LEN; j++) {
23                 arr[i][j] = ARR_LEN - j;
24             }
25         } else {
26             for (j = 0; j < ARR_LEN; j++) {
27                 arr[i][j] = j;
28             }
29         }
30     }
31 }
```

```
32
33 int compare_ints(const void* a, const void* b) {
34     int arg1 = *(const int*)a;
35     int arg2 = *(const int*)b;
36
37     if (arg1 < arg2) return -1;
38     if (arg1 > arg2) return 1;
39     return 0;
40 }
41
42 int main(int argc, char* argv[]) {
43
44     int my_rank, comm_sz;
45     double start, end;
46     double time;
47     int TASK_SIZE=atoi(argv[1]);
48     // Initialize the array
49     init_2();
50
51     MPI_Init(NULL, NULL);
52     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
53     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
54
55     if (my_rank == 0) {
56         start = MPI_Wtime();
57         int tasks_completed = 0;
58         double local_result=0;
59         // 分发初始任务
60         for (int i = 1; i < comm_sz; i++) {
61             MPI_Send(&tasks_completed, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
62             tasks_completed+=TASK_SIZE;
63         }
64
65         while (tasks_completed < ARR_NUM) {
66             MPI_Status status;
67             MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ...
                    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

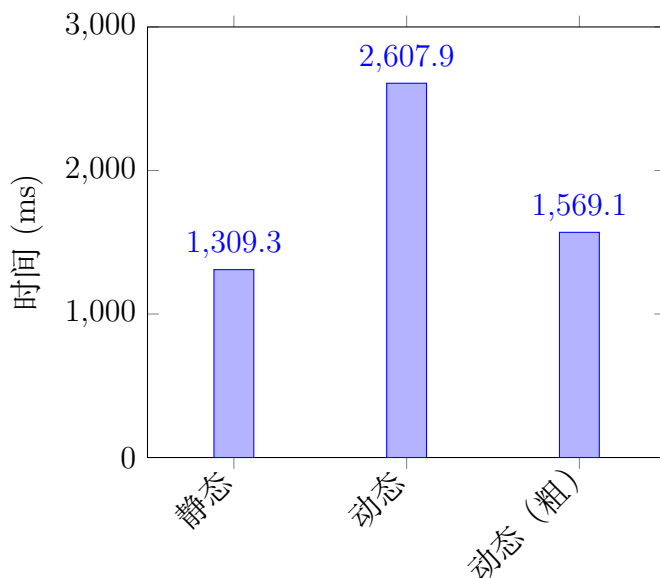
```
68         if (tasks_completed + TASK_SIZE > ARR_NUM) {
69             MPI_Send(&tasks_completed, 1, MPI_INT, status.MPI_SOURCE, 0, ...
70                 MPI_COMM_WORLD);
71             tasks_completed = ARR_NUM; // 避免超出范围
72         }
73         else {
74             MPI_Send(&tasks_completed, 1, MPI_INT, status.MPI_SOURCE, 0,...
75                 MPI_COMM_WORLD);
76             tasks_completed+=TASK_SIZE;
77         }
78
79         // 告诉所有进程任务已完成
80         for (int i = 1; i < comm_sz; i++) {
81             int stop_signal = -1;
82             MPI_Send(&stop_signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
83         }
84
85         // 收集最后的任务
86         for (int i = 1; i < comm_sz; i++) {
87             MPI_Recv(&local_result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ...
88                 MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
89         }
90
91         end = MPI_Wtime();
92         time=end-start;
93         // printf("各进程运行时间如下:\n");
94         printf("0号进程: %f s\n", time);
95         // for (int source = 1; source < comm_sz; source++) {
96         //     MPI_Recv(&time, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, ...
97         //         MPI_STATUS_IGNORE);
98         //     printf("%d号进程: %f s\n", source, time);
99         // }
100     }
101     else {
102         double start1=MPI_Wtime();
```

```
101     while (1) {
102
103         int task;
104         double local_result=0;
105         MPI_Recv(&task, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, ...
106                 MPI_STATUS_IGNORE);
107         if (task == -1) break; // 停止信号
108         for (int i=task; i<task+TASK_SIZE&& i<ARR_NUM; i++)
109             qsort(arr[i], ARR_LEN, sizeof(int), compare_ints);
110         MPI_Send(&local_result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
111     }
112     double end1 = MPI_Wtime();
113     double time1=end1-start1;
114     printf("%d号进程: %f s\n", my_rank, time1);
115     // MPI_Send(&time1, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
116 }
117
118 MPI_Finalize();
119 return 0;
120 }
```

3 结果统计与分析

3.1 不同分配方式对计算结果影响

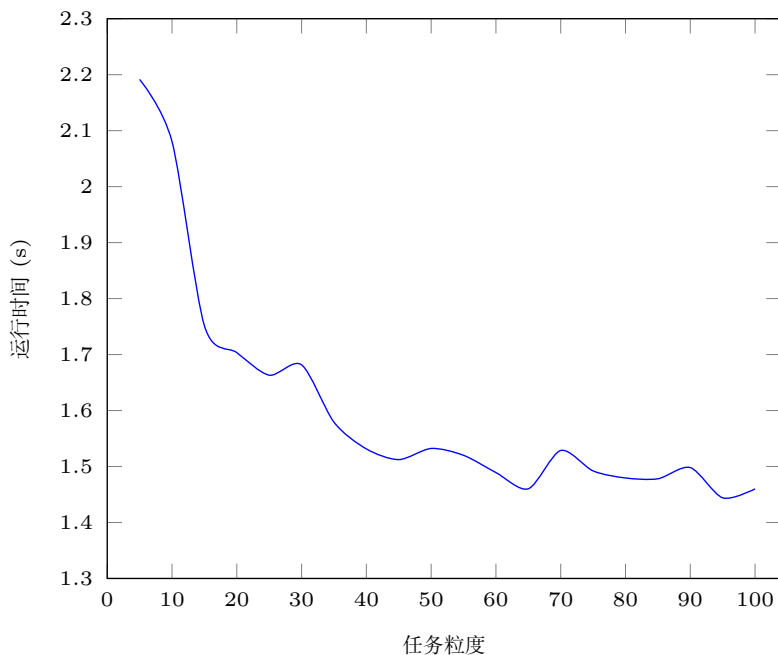
下面比较采用静态分配，动态分配，动态粗粒度分配时的运行时间比较。



可以看出，静态分配运行的时间远小于动态分配和动态粗粒度分配，而动态粗粒度分配又优于动态分配。这是因为静态分配能够显著减少不同主机间通信的开销，动态粗粒度分配相比于动态分配也能显著减少不同主机之间通信的开销。

3.2 不同分配粒度对计算结果影响

下面比较不同任务分配粒度下运行时间的影响



从图中可以看出，随着任务分配粒度增大，运行时间呈下降趋势。这是因为随着任务分

配粒度增大，进程间通信次数会变少，降低了运行时通信的时间开销，从而运行时间下降。从图中可以看出分配粒度为 65 是比较好的。

实验三 高斯消元

1 问题描述

实现高斯消去法解线性方程组的 MPI 编程，与 SSE (或 AVX) 编程结合，并与 Pthread、OpenMP (结合 SSE 或 AVX) 版本对比，规模自己设定。

2 算法设计与分析

2.1 静态分配

采用静态分配来进行高斯消元的运算，每一个进程固定分配高斯消元的块，分别进行计算。相关代码如下：

```
1 double LU_mpi_static(int argc, char* argv[]) {
2     MPI_Init(&argc, &argv);
3     double start_time = 0;
4     double end_time = 0;
5     int total = 0;
6     int rank = 0;
7     int i = 0;
8     int j = 0;
9     int k = 0;
10    MPI_Status status;
11
12    MPI_Comm_size(MPI_COMM_WORLD, &total);
13    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14
15    int begin = N / total * rank;
16    int end = (rank == total - 1) ? N : N / total * (rank + 1);
17
18    if (rank == 0) {
```

```
19     A_init();
20     for (j = 1; j < total; j++) {
21         int b = j * (N / total), e = (j == total - 1) ? N : (j + 1) * (N...
           / total);
22         for (i = b; i < e; i++) {
23             MPI_Send(&A[i][0], N, MPI_FLOAT, j, 1, MPI_COMM_WORLD);
24         }
25     }
26 }
27 else {
28     A_initAsEmpty();
29     for (i = begin; i < end; i++) {
30         MPI_Recv(&A[i][0], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
31     }
32 }
33
34 MPI_Barrier(MPI_COMM_WORLD);
35 start_time = MPI_Wtime();
36
37 for (k = 0; k < N; k++) {
38     if ((begin ≤ k && k < end)) {
39         for (j = k + 1; j < N; j++) {
40             A[k][j] /= A[k][k];
41         }
42         A[k][k] = 1.0;
43         for (j = 0; j < total; j++) {
44             if (j != rank)
45                 MPI_Send(&A[k][0], N, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
46         }
47     }
48     else {
49         int src = (k < N / total * total) ? k / (N / total) : total - 1;
50         MPI_Recv(&A[k][0], N, MPI_FLOAT, src, 0, MPI_COMM_WORLD, &status...
           );
51     }
52
53     for (i = max(begin, k + 1); i < end; i++) {
```

```
54         for (j = k + 1; j < N; j++) {
55             A[i][j] -= A[i][k] * A[k][j];
56         }
57         A[i][k] = 0;
58     }
59 }
60
61 MPI_Barrier(MPI_COMM_WORLD);
62
63 if (rank == 0) {
64     end_time = MPI_Wtime();
65     printf("静态块划分耗时: %.4lf s\n", end_time - start_time);
66 }
67
68 MPI_Finalize();
69
70 return end_time - start_time;
71 }
```

2.2 动态分配

采用动态分配来进行高斯消元的运算，每一次循环都重新分配任务，分别进行计算。相关代码如下：

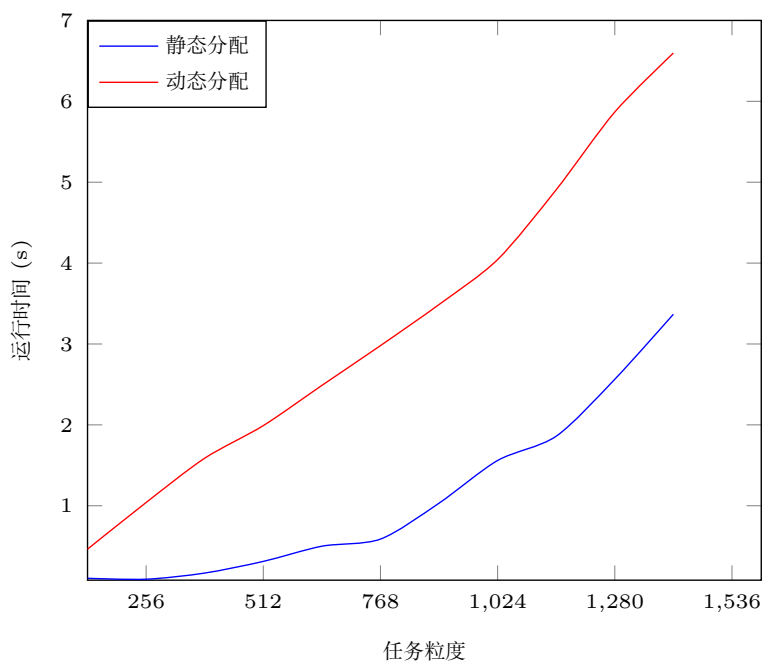
```
1 double LU_mpi_dynamic(int argc, char* argv[]) {
2     double start_time = 0;
3     double end_time = 0;
4     MPI_Init(&argc, &argv);
5     int total = 0;
6     int rank = 0;
7     int i = 0;
8     int j = 0;
9     int k = 0;
10    MPI_Status status;
11    MPI_Comm_size(MPI_COMM_WORLD, &total);
12    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
13
14     if (rank == 0) {
15         A_init();
16         for (j = 1; j < total; j++) {
17             for (i = j; i < N; i += total) {
18                 MPI_Send(&A[i][0], N, MPI_FLOAT, j, 1, MPI_COMM_WORLD);
19             }
20         }
21     }
22     else {
23         A_initAsEmpty();
24         for (i = rank; i < N; i += total) {
25             MPI_Recv(&A[i][0], N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
26         }
27     }
28
29     MPI_Barrier(MPI_COMM_WORLD);
30     start_time = MPI_Wtime();
31
32     for (k = 0; k < N; k++) {
33         if (k % total == rank) {
34             for (j = k + 1; j < N; j++) {
35                 A[k][j] /= A[k][k];
36             }
37             A[k][k] = 1.0;
38
39             for (j = 0; j < total; j++) {
40                 if (j != rank)
41                     MPI_Send(&A[k][0], N, MPI_FLOAT, j, 0, MPI_COMM_WORLD);
42             }
43         }
44         else {
45             int src = k % total;
46             MPI_Recv(&A[k][0], N, MPI_FLOAT, src, 0, MPI_COMM_WORLD, &status...
47                 );
48         }
49     }
```

```
49     int begin = k;
50     while (begin % total != rank)
51         begin++;
52     for (i = begin; i < N; i += total) {
53         for (j = k + 1; j < N; j++) {
54             A[i][j] -= A[i][k] * A[k][j];
55         }
56         A[i][k] = 0;
57     }
58 }
59
60 MPI_Barrier(MPI_COMM_WORLD);
61
62 if (rank == 0) {
63     end_time = MPI_Wtime();
64     printf("动态块划分耗时: %.4lf s\n", end_time - start_time);
65 }
66
67 MPI_Finalize();
68
69 return end_time - start_time;
70 }
```

3 结果统计与分析

3.1 MPI 任务分配方式运行时间比较



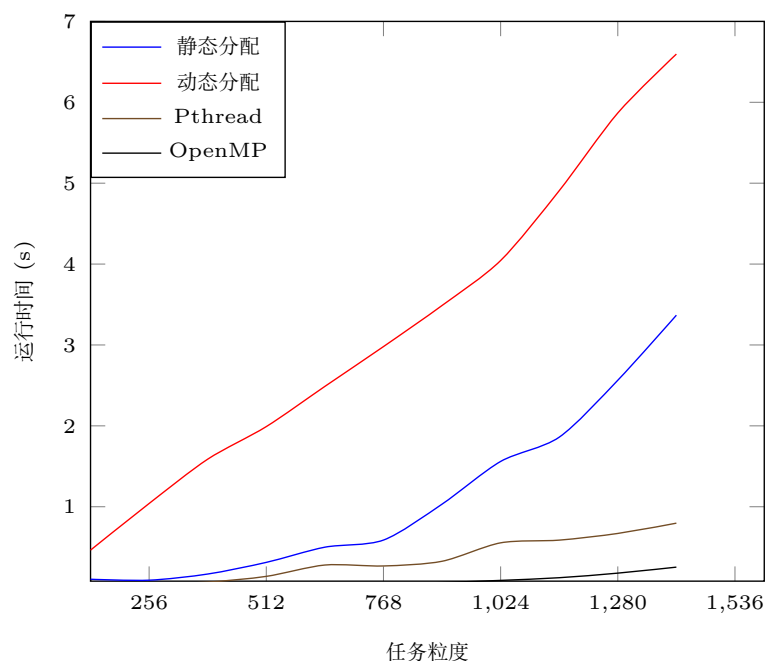
从图中可以看出，采用静态分配来分配任务远远优于使用动态分配来分配任务。因为静态分配避免了重复分配任务的开销，使得运行时间变短。

3.2 MPI 与 OpenMP, Pthread 的比较

以下是 pthread, OpenMP, MPI 运行不同矩阵规模的运行时间。

矩阵规模	静态分配 (s)	动态分配 (s)	Pthread(s)	OpenMP(s)
128	0.1036	0.4636	0.0225829	0.00295702
256	0.0919	1.0402	0.0452168	0.00611565
384	0.168	1.588	0.0681525	0.0120125
512	0.3121	1.9908	0.139116	0.0192517
640	0.5	2.4878	0.279277	0.0287804
768	0.5878	2.9798	0.267719	0.0423381
896	1.032	3.488	0.325799	0.0633175
1024	1.56	4.044	0.553773	0.0893394
1152	1.86	4.9071	0.587159	0.122548
1280	2.5638	5.868	0.669795	0.17918
1408	3.3678	6.5962	0.79703	0.253331

表 3.1: 不同矩阵规模下不同分配策略的 LU 分解执行时间比较



从图中可以看出，采用 Pthread 和 OpenMP 来进行高斯消元并行计算要远远优于使用 MPI 进行计算，这是因为 Pthread 和 OpenMP 这两种技术用于共享内存系统。它们允许所有线程或任务访问相同的内存空间，这对于高斯消元这类需要频繁访问和修改相同数据集的任务非常有效。而 MPI 是为分布式内存系统设计的，适合于多个处理器或计算节点上的并行计算。对于需要大量数据交换的任务，如高斯消元，MPI 的通信开销可能会变得相对较大。

实验结论及心得体会

在这次实验中，我不仅学会了 MPI 编程，而且深入探讨了它与 pthread 和 OpenMP 的相似之处和差异。通过对 MPI 的学习，我了解到这是一种适用于并行计算的通信协议，尤其在分布式内存系统中表现出色。但是本次实验的三个实验并没有体现出 MPI 编程的优势，这是因为本次实验在计算规模上并不算特别大，同时通信也比较频繁。在三个实验中，可以很清晰地看到静态分配任务远远优于动态分配任务，这一点是不同于 pthread 编程和 OpenMP 编程的，这也能体现出 MPI 编程与他们的不同。虽然在这三个实验中，MPI 编程并没有体现出它的优势，但这不代表 MPI 编程没有用。在需要大规模的计算与数据交换时 MPI 编程便能体现出它的优越性。