第三次作业

元培学院 黄道吉 1600017857

前期准备

概述PSRS算法

Parallel Sorting by Regular Sampling(以下简称PSRS)算法, 首先将输入数据平均分配给各个核, 在各个核中排序, 收集各自局部的枢轴值(pivot), 再取出枢轴值的枢轴值(全局的pivot), 按全局pivot将各个核的数组分类, 每一个核处理两个枢轴值之间的数字(这里的实现取[pivot[i - 1], pivot[i]), 即左闭右开的区间划分), 最后根线程收集数组.

具体的算法参考, 这一篇论文

MPI用法

关于MPI函数的用法和参数, 参考了这里的文档中对各个函数的介绍.

结果评测

以单线程的std::sort作为baseline, -O2优化, 排序1e8的数组所需时间约12s

下面列出在进程数$n = 4$, 数组大小变化, 排序算法采用merge_sort(二路归并), 编译选项-O1时的时间消耗

数组大小	时间
1000000	1.07 +- 0.02s
10000000	9.42 +- 0.01s
50000000	55.875 +- 0.015s
100000000	112.35 +- 0.05s

发现时间大致按照线性增长, 符合复杂度预期($O(\frac{n}{p}\log\frac{n}{p})$), 算法实现没有问题

下面分别列出在psrs内使用std::sort排序和使用multi_merge排序, 使用不同的进程数量, 编译选项-O2, 比较数组大小100000000时的排序时间

进程数	std::sort	multi_merge
1	16.67s	15.15s
4	11.94 +- 0.08s	6.98 +- 0.12s
8	9.47 +- 0.05s	6.65 +- 0.07s
16	10.5 +- 1.5s	8.5 +- 0.7s
20	12.2 +- 1.2s	10.4 +- 1.5s

对比进程数为1的和使用std::sort()的时间, 估计通信时间大致为3s; 并行的效率在$n = 4$时达到最大, 加速比在$n = 8$时达到最大, 故进一步的调试采用$n = 8$.

取merge_sort, $n = 8$, -O3优化, 时间一般达到6.47 +- 0.02s(如果考虑最短时间, 可以达到6.42s)

取多次运行的平均时间为最终时间, 为6.47s(若取最短时间, 可以是6.42s)

代码位置: (mc:~) ./users/Huang.Daoji/3/my_sort_mergesort.cpp, ./users/Huang.Daoji/3/* 为各种已用mpi编译好程序, 最终结果采用

```
mpirun -n 8 ./my_sort_mergesort
```

的结果

代码实现

计时部分&随机数

完全按照助教的示例代码, 计时的时间段是从MPI_Init()到MPI_Finalize(), 即整个排序阶段, 不包含生成数组和验证阶段.

算法的代码

首先各个核分到自己的数据, 自己排序

```
// scatter original array to each thread

    vector<int> this_array(size);
    MPI_Scatter(tovoid(v), size, MPI_INT, tovoid(this_array), size, MPI_INT, 0, MPI_COMM_WORLD);

    // and each thread sort its own part, use std::sort here because it is fastest
    std::sort(this_array.begin(), this_array.end());
```

选出自己的枢轴值, 选出全局的枢轴

```
// choose pivot!

    vector<int> pivot(size_of_threads);
    vector<int> pivot_tot(size_of_threads * size_of_threads);
    for(int i = 0; i < size_of_threads; i += 1){
        pivot[i] = this_array[i * size / size_of_threads];
    }
    // and gather them
    MPI_Gather(tovoid(this_array), size_of_threads, MPI_INT, tovoid(pivot_tot), size_of_threads,
MPI_INT, 0, MPI_COMM_WORLD);


// select pivot!

    if(rank == 0){
```

```
        vector<int*> s(size_of_threads);
        vector<int> l(size_of_threads);
        for(int i = 0; i < size_of_threads; i += 1){
            s[i] = &pivot_tot[i * size_of_threads];
            l[i] = size_of_threads;
        }
        vector<int> pivot_sorted(size_of_threads * size_of_threads);
        // seems to be faster, i do not choose loser tree for its conplexity
        multi_merge(&s[0], &l[0], size_of_threads, toint(pivot_sorted), size_of_threads *
size_of_threads);

        for(int i = 0; i < size_of_threads - 1; i += 1){
            pivot[i] = pivot_sorted[(i + 1) * size_of_threads];
        }
    }
    // and send global pivot to all processes
    MPI_Bcast(tovoid(pivot), size_of_threads - 1, MPI_INT, 0, MPI_COMM_WORLD);
```

每个核分到按照新的枢轴值分的数组

```
// class partition

    vector<int> class_start(size_of_threads);
    vector<int> class_length(size_of_threads);
    int idx = 0;
    for(int class_idx = 0; class_idx < size_of_threads - 1; class_idx += 1){
        class_start[class_idx] = idx;
        class_length[class_idx] = 0;
        while(idx < size && this_array[idx] < pivot[class_idx]){
            class_length[class_idx] += 1;
            idx += 1;
        }
    }
    // the last class
    class_start[size_of_threads - 1] = idx;
    class_length[size_of_threads - 1] = size - idx;

// sort each part again!

    vector<int> received;
    vector<int> received_length(size_of_threads);
    vector<int> received_start(size_of_threads);

    for(int thread = 0; thread < size_of_threads; thread += 1){
        // how long should i send?
        MPI_Gather(&class_length[thread], 1, MPI_INT, tovoid(received_length), 1, MPI_INT,
thread, MPI_COMM_WORLD);
        if(rank == thread){
            received_start[0] = 0;
            for(int i = 1; i < size_of_threads; i += 1){
                received_start[i] = received_start[i - 1] + received_length[i - 1];

            }
```

```
            // how long should i receive?
            received.resize(received_start[size_of_threads - 1] +
received_length[size_of_threads - 1]);
        }
        // send & receive them!
        MPI_Gatherv(&this_array[class_start[thread]], class_length[thread], MPI_INT,
tovoid(received), toint(received_length), toint(received_start), MPI_INT, thread,
MPI_COMM_WORLD);
    }

    vector<int*> s(size_of_threads);
    for(int i = 0; i < size_of_threads; i += 1){
        s[i] = toint(received) + received_start[i];
    }
    // sort my part
    multi_merge(&s[0], toint(received_length), size_of_threads, toint(v), n);

    // how long should i send to root process?
    int this_send_length = received_start[size_of_threads - 1] + received_length[size_of_threads
- 1];
```

rank=0的线程回收所有数据

```
// then collect them!

    vector<int> send_length(size_of_threads);
    vector<int> send_starts(size_of_threads);

    // tell root process how long i will send
    MPI_Gather(&this_send_length, 1, MPI_INT, tovoid(send_length), 1, MPI_INT, 0,
MPI_COMM_WORLD);

    if(rank == 0){
        send_starts[0] = 0;
        for(int i = 1; i < size_of_threads; i += 1){
            send_starts[i] = send_starts[i - 1] + send_length[i - 1];
        }
    }


// collect parts, now the original array is already sorted

    MPI_Gatherv(tovoid(v), this_send_length, MPI_INT, tovoid(v), toint(send_length),
toint(send_starts), MPI_INT, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
```