Lab4-report

- 实验内容

  使用MPI，实现`PSRS（Parallel sorting by regular sampling）`算法。

- 实验环境

  - 操作系统：Ubuntu 18.04 lts 64bits
  - 编译器
    - mpi 3.2.1 (for mpi)
  - CPU: Inte i5-8250U CPU & 3.40Ghz * 8
  - Memory： 8GB

- 算法设计与分析

  在本次实验中，要求用MPI实现Parallel Sorting by Regular Sampling算法，设待处理里序列长n，并行机上有p个处理器，在实现时待处理序列数组采用随机生成的策略，然后对该大小为n的随机数据进行MPI并行化和psrs算法的实现，其中psrs算法的具体流程分为正则采样、选择主元并划分、全局交换和归并排序四个步骤完成，具体算法的思想如下：

  - **1.**正则采样

    给定大小为n的数组后，根据线程数目p对全局数组进行均匀划分，每个线程(前p-1个线程)负责处理的局部数组大小为$[\frac{n}{p}]$，由于给定的数目n不一定恰好为p的整数倍，所以对于最后一个线程其处理的局部数组大小为$n-[\frac{n}{p}]\cdot(p-1)$。对原数组进行均匀划分之后，每个线程对其负责的局部数组进行局部的快速排序，然后进行正则采样。正则采样的思想是对于每一个线程的局部数组而言，从中均匀的（等步长）取出p个元素作为采样样本。

    具体代码实现时，设定每个线程设定`start`和`sub_array_size`两个变量记录其局部数组的数组起始点和长度，这样就确定了每个线程互不相交的局部数组，然后用串行排序`qsort()`对局部数组排序，最后按照等步长采样将p个样本存放在数组`sample_array`中。

  - **2.**选择主元并划分

    正则采样过程完成之后，首先要将每个线程的采样结果进行统计整合，指定rank=0的主线程负责收集每一个线程的采样数组，并且对接受到的总的采样结果进行局部的串行排序，对于排序之后的采样结果，可以从中等距的选出p-1个元素作为主元用于之后的划分。主线程将主元广播给其他线程，这样每一个线程都得到了全局数组的整体分布情况。线程就可以根据选择的p-1个主元对其局部数组进行划分，即按照主元作为分界点，可以将每一个局部数组划分为p个部分。

    具体代码实现时，通过MPI的`MPI_Gather()`函数进行采样数组的收集到主线程，然后判定rank为0的线程用`qsort()`串行算法对样本进行排序，然后得到主元放在数组`sample_pivots`中，并且将该数组通过`MPI_Bcast()`函数广播，具体在实现局部数组的划分时，在数组`partition_sizes`中保存每一个划分的数组长度，即遍历局部数组并且判断与每一个主元的大小关系从而进行截断，将每一部分的长度保存到该数组即可记录划分点。

- **3.全局交换**

  在完成上述过程的主元划分之后，由于每一个线程都有全局数组的副本，所以只需要将各自局部数组的记录划分点的划分数组分别发送给每一个线程，这样每一个线程就获取了新的局部数组的每一划分的长度，对局部数组的划分长度求和后可以得到新局部数组的大小，然后通过多线程的多对多通信将每一局部数组的对应划分块播送到对应的线程中，这样之后保证局部数组A_i种的任一元素都大于$A_j(j<i)$的任一元素。

  具体代码实现时，通过MPI的`MPI_Alltoall()`函数实现多对多通信，每一线程P_i恰好得到其他线程中第i块的划分大小，所以每个线程计算新的局部数组大小存放在遍历`new_sub_array_size`中，同时为新局部数组`new_sub_array`分配空间，最后通过`MPI_Alltoallv()`进行多次点对点通信，即可实现全局的交换过程。

- **4.归并排序**

  经过以上过程后，数组已经在全局上（线程级别）满足了排序关系，仅仅再对局部数组进行排序即可，如果直接按照`qsort()`进行局部排序会导致时间复杂度过大，因为当前局部数组中每一个划分都是满足大小关系的，所以按照归并排序的思想重新组合仅需要局部的$O(n_i)$的时间复杂度即可。

- **MPI实现**

  在用MPI实现并行时，首先要调用MPI函数`MPI_Init(..)`进入MPI环境并完成所有的初始化工作，然后，调用`MPI_Comm_rank`函数获得当前进程在指定通信域中的编号，将自身与其他程序区分。然后要获取指定通信域的进程数，调用`MPI_Comm_size`函数获取指定通信域的进程个数。如果返回的rank为0的话说明是主线程，从主线程接收命令行参数确定数组大小，然后通过`MPI_Send`将数组大下发送给其他线程，然后每个线程按照顺序调用以上四个阶段的处理函数即可。

  最后所有线程同步的模拟完成后rank为0的线程进行汇总，得到程序的运行时间和排序结果，并且由`check_result`函数进行排序结果的验证，从而保证结果的正确性。最后通过`MPI_Finalize`函数从MPI环境中退出。

- **算法时间复杂度及优化分析**

  在模拟时，按照以上分析的算法流程，每个线程并行的处理$\frac{n}{p}$个数组元素，局部进行快排，所以其时间复杂度为$O(\frac{n}{p}\cdot log(\frac{n}{p}))$。

核心代码

- **1.正则采样**

```
void regular_sample()
{
    // local qsort
    qsort(my_array + start, sub_array_size, sizeof(my_array[0]), cmp);
    // regular sample
    for(int i = 0; i < size; i++)
        sample_array[i] = my_array[start + (i * (array_size / (size * size)))];

}
```

- **2.**选择主元并划分

```
void pivots_partition()
{
    int *all_sample_array = (int *)malloc(size * size * sizeof(sample_array[0]));
    int *sample_pivots = (int*)malloc((size - 1) * sizeof(sample_array[0]));
    int index = 0;

    // Gather samples
    MPI_Gather(sample_array, size, MPI_INT, all_sample_array, size, MPI_INT, 0,
MPI_COMM_WORLD);
    if(rank == 0)
    {
        qsort(all_sample_array, size * size, sizeof(sample_array[0]), cmp);
        for(int i = 0; i < (size - 1); i++)
            sample_pivots[i] = all_sample_array[(((i + 1) * size) + (size / 2)) - 1];

    }
    // Bcast pivots
    MPI_Bcast(sample_pivots, size - 1, MPI_INT, 0, MPI_COMM_WORLD);
    // Partitions
    for (int i = 0; i < sub_array_size; i++)
    {
        if(my_array[start + i] > sample_pivots[index])
            index += 1;

        if (index == size)
        {
            // the last partition
            partition_sizes[size - 1] = sub_array_size - i + 1;
            break;
        }
        partition_sizes[index]++ ;
    }
    free(all_sample_array);
    free(sample_pivots);
}
```

- **3.**全局交换

```
void partitions_exchange()
{
    int new_sub_array_size = 0;
    int *sdispls = (int*)malloc(size * sizeof(int));
    int *rdispls = (int*)malloc(size * sizeof(int));

    MPI_Alltoall(partition_sizes, 1, MPI_INT, new_partition_sizes, 1, MPI_INT,
MPI_COMM_WORLD);
     // calculate the size of the new sub_arrray and allocate space.
    for(int i = 0; i < size; i++)
        new_sub_array_size += new_partition_sizes[i];

    new_sub_array = (int*)malloc(new_sub_array_size * sizeof(int));

    // calculate offsets
    sdispls[0] = 0;
    rdispls[0] = 0;
    for(int i = 1; i < size; i++)
    {
        sdispls[i] = partition_sizes[i - 1] + sdispls[i - 1];
        rdispls[i] = new_partition_sizes[i - 1] + rdispls[i - 1];
    }

    // communicate
    MPI_Alltoallv(&(my_array[start]), partition_sizes, sdispls, MPI_INT, new_sub_array,
new_partition_sizes, rdispls, MPI_INT, MPI_COMM_WORLD);
    free(sdispls);
    free(rdispls);
    return;
}
```

- **4.**归并排序

```
void partition_merge()
{
    int *sorted_sub_array;
    int *rdispls, *partitions_start, *partitions_end, *sub_array_sizes, sub_array_size;

    partitions_start = (int*)malloc(size * sizeof(int));
    partitions_end = (int*)malloc(size * sizeof(int));
    partitions_start[0] = 0;
    sub_array_size = new_partition_sizes[0];

    for(int i = 1; i < size; i++)
    {
        sub_array_size += new_partition_sizes[i];
        partitions_start[i] = partitions_start[i - 1] + new_partition_sizes[i - 1];
        partitions_end[i - 1] = partitions_start[i];
    }
    partitions_end[size - 1] = sub_array_size;

    sorted_sub_array = (int*)malloc(sub_array_size * sizeof(int));
    sub_array_sizes = (int*)malloc(size * sizeof(int));
    rdispls = (int*)malloc(size * sizeof(int));

    // merge sort locally
    for(int i = 0; i < sub_array_size; i++)
    {
        int min_value = INT_MAX;
        int min_index;
        for(int j = 0; j < size; j++)
        {
            if((partitions_start[j] < partitions_end[j]) &&
(new_sub_array[partitions_start[j]] < min_value))
            {
                min_value = new_sub_array[partitions_start[j]];
                min_index = j;
            }
        }
        sorted_sub_array[i] = min_value;
        partitions_start[min_index] += 1;
    }

    MPI_Gather(&sub_array_size, 1, MPI_INT, sub_array_sizes, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    // calculate offsets
    if(rank == 0)
    {
        rdispls[0] = 0;
        for (int i = 1; i < size; i++)
            rdispls[i] = sub_array_sizes[i - 1] + rdispls[i - 1];
    }



    MPI_Gatherv(sorted_sub_array, sub_array_size, MPI_INT, my_array, sub_array_sizes,
```

```
rdispls, MPI_INT, 0, MPI_COMM_WORLD);

    free(partitions_end);
    free(sorted_sub_array);
    free(partitions_start);
    free(sub_array_sizes);
    free(rdispls);
}
```

- **MPI执行部分**

```
    if(rank == 0)
    {
        array_size = atoi(argv[1]);

        wtime = MPI_Wtime();
        for(int j = 1; j < size; j++)
            MPI_Send(&array_size, 1, MPI_INT, j, 1, MPI_COMM_WORLD);
    }
    else
        MPI_Recv(&array_size, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);

    ...

    MPI_Barrier(MPI_COMM_WORLD);
    // start executing psrs sort
    regular_sample();
    if(size > 1)
    {
        pivots_partition();
        partitions_exchange();
        partition_merge();
    }
```

- 实验结果

取多个命令行参数n测试后，时间结果如下：

- 运行结果（**n=64**）

```
shuangli@ubuntu:~/Downloads/mpi$ mpirun -np 4 ./psrs_mpi 64
Initial array:
154 191 316 234 139 324 530 554 497 123 474 357 88 20 3 159 104 344 536 161 31 5
94 473 226 62 66 590 35 84 195 136 110 259 452 345 398 136 235 184 506 231 19 95
 319 39 98 350 15 314 247 177 345 201 522 571 135 460 521 170 544 588 306 15 207

after psrs, sorted array:
3 15 15 19 20 31 35 39 62 66 84 88 95 98 104 110 123 135 136 136 139 154 159 161
 170 177 184 191 195 201 207 226 231 234 235 247 259 306 314 316 319 324 344 345
 345 350 357 398 452 460 473 474 497 506 521 522 530 536 544 554 571 588 590 594

[INFO]time is 0.001089s.
[INFO]sorted result is right.
```

- 运行时间

规模\线程数	1	2	4	8
n=100w	0.235014s	0.177617s	0.1243461s	0.082461s
n=500w	1.619126s	1.122511s	0.753082s	0.530861s
n=1000w	2.928587s	1.863068s	1.210160s	0.851549s
n=2000w	5.928262s	3.581771s	2.124829s	1.535821s
n=4000w	12.471849s	7.877838s	4.227756s	3.019863s

- 加速比

规模\线程数	1	2	4	8
n=100w	1	1.33	1.89	2.85
n=500w	1	1.45	2.15	3.05
n=1000w	1	1.57	2.42	3.48
n=2000w	1	1.65	2.79	3.86
n=4000w	1	1.59	2.95	4.13

- 结果分析

  从实验结果来看，在n取值较大时，每一个规模下的psrs执行时间随着线程数目的增加而减少，即加速比增大，并且可以看到加速效果明显。

- ## 总结和收获

  此次实验，通过MPI实现了高效的多处理器上的psrs算法，并且在线程通信时使用了多对多的通信方式，通过实验可以看到并行化之后可以提高程序效率。需要注意的是，psrs算法尽管很高效，但是在进行主元划分时时可能会引起不均匀性，即位于某两个主元之间的元素可能要多一些，在算法中进行全局交换时，可能引起的直接后果是处理器负载不均匀，那么在归并排序中可能会引起排序时间的不均匀。