并行计算实验 2: MPI

傅申 PB20000051

1. 实验目的

使用 MPI 编写并行程序.

- 实现 PSRS 排序.
- N 个进程求 N 个数的全和, 要求每个处理器均保持全和.
  - 蝶式全和
  - 二叉树式全和

2. 实验环境

本次实验在我自己的笔记本上运行, 硬件参数如下:

CPU Intel i5-1035G1, 4 核 8 进程.
内存 8 GB DDR4 3200 MHz × 2.

相应的软件环境如下:

OS GNU/Linux 6.1.29-1-MANJARO x86_64.
MPI MPICH Version 4.1
编译器 mpicc
编译选项 `-Ofast -Wall`

3. 实验过程与结果

3.1. 实现 PSRS 排序

3.1.1. 程序的执行过程

程序的用法为 `mpirun 进程数 ./psrs [数组大小]`. 当没有提供数组大小时, 将默认使用课本上的 27 个元素的数组; 若提供了数组大小, 则会生成一个随机数组进行排序. 当最终的数组长度不能整除运行时的进程数时, 将会在数组末尾补上数据大小的上界以使得数组长度能够被进程数整除.

上述提到的数组都只会在主进程中生成, 在生成后, 只有数组的长度被广播到所有进程. 随后 PSRS 开始, 排序结果存储到一个新数组中. PSRS 结束后程序将对使用 `std::sort()` 对原始数组 (未被 PSRS 修改) 排序, 并将两次排序结果进行比对. 如果二者结果不相同, 则会输出错误信息.

最后, 程序将会输出两次排序的时间信息, 并计算其加速比并输出.

3.1.2. PSRS 实现

排序函数的定义为: `void psrs(int *array, unsigned int size, int *sorted)`. 其中 `array` 为输入数组, `size` 为数组的大小, `sorted` 为输出数组.

在进行排序前, 每个进程都会获得其编号 `id` 和进程数 `num_procs`.

PSRS 算法的实现如下:

(1) 均匀划分: 各个进程分配一段内存空间用于存储子数组, 主进程将数组散播.

```
int  subarray_size = size / num_procs;
int *subarray      = new int[subarray_size];
MPI_Scatter(array, subarray_size, MPI_INT, subarray, subarray_size, MPI_INT,
            0, MPI_COMM_WORLD);
```

(2) 局部排序: 各个进程调用 `std::sort()` 对子数组进行排序.

(3) 选取样本: 各个进程将选取的样本存储在 `samples` 中.

```
int samples[num_procs];
for (int i = 0; i < num_procs; i++) {
    samples[i] = subarray[i * subarray_size / num_procs];
}
```

主进程收集所有样本存储在 `sample_recv` 中.

```
int sample_num = num_procs * num_procs;
int sample_recv[sample_num];
MPI_Gather(samples, num_procs, MPI_INT, sample_recv, num_procs, MPI_INT, 0,
           MPI_COMM_WORLD);
```

(4) 样本排序: 主进程调用 `std::sort()` 对 `sample_recv` 进行排序.

(5) 选择主元: 主进程从 `sample_recv` 中选取 $p-1$ 个主元, 并广播给各个进程.

```
int pivots[num_procs - 1];
if (id == 0) {
    for (int i = 1; i < num_procs; i++) {
        pivots[i - 1] = sample_recv[i * num_procs];
    }
}
MPI_Bcast(pivots, num_procs - 1, MPI_INT, 0, MPI_COMM_WORLD);
```

(6) 主元划分: 各个进程根据 `pivots` 将子数组划分为 p 段. 这里程序只存储各个段的长度和起始元素的偏移量, 而不将元素拷贝到新的内存空间.

```
int partition_sizes[num_procs];
int offsets[num_procs];
int index  = 0;
offsets[0] = 0;
for (int i = 0; i < num_procs - 1; i++) {
    partition_sizes[i] = 0;
    while (index < subarray_size && subarray[index] <= pivots[i]) {
        partition_sizes[i]++;
        index++;
    }
    offsets[i + 1] = offsets[i] + partition_sizes[i];
}
partition_sizes[num_procs - 1] = subarray_size - offsets[num_procs - 1];
```

(7) 全局交换: 首先, 各个进程之间全局交换对应段的长度.

```
int partition_sizes_recv[num_procs];
MPI_Alltoall(partition_sizes, 1, MPI_INT, partition_sizes_recv, 1, MPI_INT,
             MPI_COMM_WORLD);
```

然后, 各个进程根据接收到的长度计算相应段的偏移量.

```
int new_subarray_size = 0;
int offsets_recv[num_procs + 1];
offsets_recv[0] = 0;
for (int i = 0; i < num_procs; i++) {
    new_subarray_size  += partition_sizes_recv[i];
    offsets_recv[i + 1] = offsets_recv[i] + partition_sizes_recv[i];
}
```

最后, 各个进程根据刚刚得到的长度和偏移量全局交换各个段的元素.

```
int *new_subarray = new int[new_subarray_size];
MPI_Alltoallv(subarray, partition_sizes, offsets, MPI_INT, new_subarray,
              partition_sizes_recv, offsets_recv, MPI_INT, MPI_COMM_WORLD);
```

(1) 归并排序: 因为各个数据段已经是有序的了, 所以直接归并各个有序段即可,

没有必要排序. 归并 p 个数组的过程不再赘述. 在归并完成后, 由主进程收集各个子数组到 `sorted` 中, 排序完成.

3.1.3. 程序其他内容

在 Makefile 中, 提供了两个模式 `psrs` 和 `debug`. `debug` 会输出更为细致的执行信息 (例如排序后的数组).

3.1.4. 运行结果

在默认情况下运行 `debug` 模式的编译结果, 输出如下:

```
$ mpirun -n 3 ./debug
[Debug] Using default array.
[Debug] The input array is: 15 46 48 93 39 6 72 91 14 36 69 40 89 61 97 12 2
1 54 53 97 84 58 32 27 33 72 20
[Debug] Starting psrs().
[Debug] psrs() done.
[Debug] The sorted array is: 6 12 14 15 20 21 27 32 33 36 39 40 46 48 53 54
58 61 69 72 72 84 89 91 93 97 97
[Info] The result is correct.
[Info] Time used by psrs():      5.3703e-05 s.
[Info] Time used by std::sort(): 7.82e-07 s.
[Info] Speedup: 0.0145616.
```

可以看到, PSRS 的运行结果没有问题.

运行 `psrs` 模式的编译结果, 可能的输出如下:

```
$ mpirun -n 4 ./psrs 10000000
[Info] The result is correct.
[Info] Time used by psrs():      0.231969 s.
[Info] Time used by std::sort(): 0.710032 s.
[Info] Speedup: 3.06089.
```

程序没有输出错误信息, 可以认为运行结果没有问题.

3.1.5. 运行时间分析

分别在不同的数据规模下以 4 进程运行 `./psrs` 十次, 统计平均运行时间, 如下:

数据规模	psrs()	std::sort()	加速比
1000	0.13 ms	0.04 ms	0.307
10000	0.66 ms	0.58 ms	0.885
100000	2.26 ms	5.71 ms	2.527
1000000	24.40 ms	65.25 ms	2.674
10000000	234.87 ms	715.46 ms	3.046

可以看到, 随着数据规模的增加, 并行算法的加速比越来越大, 性能提升越来越显著.

3.2. 多个进程求全和

3.2.1. 蝶式全和

在求和过程中, 各个进程选择其“邻居”, 与其交换各自存储的数字并相加, 重复这个过程 $\lg N$ 次, 最终得到全和.

不难注意到每轮迭代中, 两个互为“邻居”的进程编号只有一位不同, 且第 k 次迭代中不同的位恰好为第 k 位. 因此, 可以实现全和如下:

```
int butterfly(int number)
{
    int id, num_procs;

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    int        recv;
    MPI_Status status;
    for (int i = 1; i < num_procs; i <<= 1) {
        int tag  = i;
        int dest = id ^ tag;
        MPI_Send(&number, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, dest, tag, MPI_COMM_WORLD, &status);
        number += recv;
    }

    return number;
}
```

运行程序, 结果如下:

```
$ mpirun -n 8 ./butterfly
[Info] Proc #0's origin number is 1
[Info] Proc #1's origin number is 2
[Info] Proc #2's origin number is 3
[Info] Proc #3's origin number is 4
[Info] Proc #6's origin number is 7
[Info] Proc #5's origin number is 6
[Info] Proc #4's origin number is 5
[Info] Proc #7's origin number is 8
[Info] Proc #7's butterfly sum is 36.
[Info] Proc #2's butterfly sum is 36.
[Info] Proc #6's butterfly sum is 36.
[Info] Proc #0's butterfly sum is 36.
[Info] Proc #4's butterfly sum is 36.
[Info] Proc #3's butterfly sum is 36.
[Info] Proc #1's butterfly sum is 36.
[Info] Proc #5's butterfly sum is 36.
```

可以看到求到的全和是正确的, 并且每个进程都得到了全和.

3.2.2. 二叉树式全和

在求和过程中, 首先各个进程将自己的数字发送给“父进程”, 父进程将数字相加, 重复直到“根进程”得到了全和. 然后“根进程”再自上而下的将全和传递到各个进程.

不妨每次都选取进程号较小的进程作为两个“兄弟进程”的“父进程”. 这样, 较小进程号的进程只需要接收数据并与它自己的数据相加即可. 最后再自上而下的传递全和即可. 与蝶式全和相同, 第 k 次迭代中, 两个“兄弟进程”的进程号之间只有第 k 位不同.

```c
int tree(int number)
{
    int id, num_procs;

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    int        recv;
    MPI_Status status;

    /* Assuming smaller proc be the parent of larger proc */
    /* Bottom-up */
    for (int i = 1; i < num_procs; i <<= 1) {
        int tag = i;
        int sibling = id ^ tag;
        if (id & tag) {
            MPI_Send(&number, 1, MPI_INT, sibling, tag, MPI_COMM_WORLD);
        } else {
            MPI_Recv(&recv, 1, MPI_INT, sibling, tag, MPI_COMM_WORLD, &status);
            number += recv;
        }
```

```
    }

    /* Top-down */
    for (int i = num_procs; i >= 2; i >>= 1) {
        if ((id & (i - 1)) == 0) {  // id % i == 0
            MPI_Send(&number, 1, MPI_INT, id + i / 2, i + num_procs,
                     MPI_COMM_WORLD);
        } else if ((id & (i / 2 - 1)) == 0) {  // id % (i / 2) == 0
            MPI_Recv(&number, 1, MPI_INT, id - i / 2, i + num_procs,
                     MPI_COMM_WORLD, &status);
        }
    }

    return number;
}
```

运行程序, 结果如下:

```
$ mpirun -n 8 ./tree
[Info] Proc #2's origin number is 3
[Info] Proc #3's origin number is 4
[Info] Proc #1's origin number is 2
[Info] Proc #4's origin number is 5
[Info] Proc #0's origin number is 1
[Info] Proc #7's origin number is 8
[Info] Proc #6's origin number is 7
[Info] Proc #5's origin number is 6
[Info] Proc #0's tree sum is 36.
[Info] Proc #4's tree sum is 36.
[Info] Proc #5's tree sum is 36.
[Info] Proc #2's tree sum is 36.
[Info] Proc #1's tree sum is 36.
[Info] Proc #6's tree sum is 36.
[Info] Proc #7's tree sum is 36.
[Info] Proc #3's tree sum is 36.
```

可以看到求到的全和是正确的, 并且每个进程都得到了全和.

4. 实验总结

本次实验中, 我了解了 MPI 的通信模型, 并使用 MPI 实现了 PSRS 和求全和算法.