中山大学数据科学与计算机学院

《高性能程序设计基础》实验 6

（2018-2019 学年秋季学期）

学　　号：16337113

姓　　名：劳马东

教学班级：教务 2 班

专　　业：超算

一、实验题目

1、 完成正则采样排序 PSRS 的 MPI 算法；

2、 按要求使用 MPI 集合通信。

二、正则采样排序概述

快速排序算法的效率相对较高，并行算法在理想的情况下时间复杂度可达到 $O(n)$，但它有一个严重的问题：会造成严重的负载不平衡，最差情况下算法的复杂度可达 $O(n^2)$。并行正则采样排序克服了这一缺点，是一种基于均匀划分的负载平衡的并行排序算法。

（一）算法流程

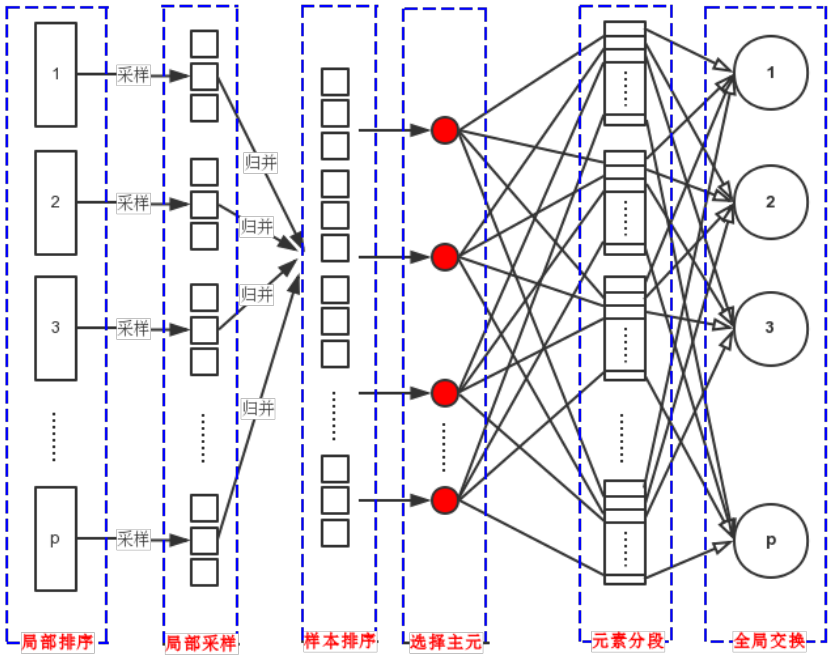

图 1. 正则采样排序示意图

假设待排序的元素 n 个，处理器 p 个，算法大体流程如下。

1、 首先将这 n 个元素均匀的分成 p 部分，每部分包含 $\frac{n}{p}$ 个元素（如图 1 矩形）。每个处理器负责其中的一部分，并对其进行局部排序；

2、 为确定局部有序序列在整个序列中的位置，每个处理器从各自的局部有序序列中选取几个代表元素（如图 1 第一列正方形）；

3、 将这些代表元素进行排序后选出 p-1 个主元（如图 1 红色圆形）；

4、 每个处理器根据这 p-1 个主元将自己的局部有序序列分成 p 段；

5、 然后通过全局交换的方式，将 p 段有序序列分发给对应的处理器（如图 1 大圆），使第 i 个处理器都拥有各个处理器的第 i 段，共 p 段有序序列。

6、 每个处理器对这 p 段有序序列进行排序。最后，将各个处理器的有序段按顺序汇合起来，就是全局有序序列了。

三、实验过程

（一）元素划分与采样

出于方便和性能考虑，每个进程直接从文件中并行读取元素。并行读取最重要的是解决文件指针定位的问题，即每个进程该从什么位置开始读取多少个元素？假设 p 个进程分别编号为 0 1...$p-2$ $p-1$，则显然进程 i 开始读的位置是前 $i-1$ 个进程所读元素的和，即：

$$f(i) = \begin{cases} f(i-1) + balance(i-1) & i = 1, 2, ..., p-2, p-1 \\ 0 & i = 0 \end{cases} \tag{1}$$

其中 *balance* 函数返回对应进程的负载，即读取多少个元素。

那么如何相对均衡地分配负载呢？方法是先给每个进程分配 $\lfloor \frac{n}{p} \rfloor$ 个元素，然后将剩下的 r 个元素分配给编号 0 到 $r-1$ 的进程。

```
vector<uint_fast64_t> divide_read_directly(istream& in,
                                           uint_fast64_t n,
                                           MPI_Comm comm)
{
    Comm_Info info(comm);
    // 获取每个进程对应的负载（均衡）
    vector<uint_fast64_t> balance = get_v<uint_fast64_t>(n, info.comm_size);
    // 得到自己的负载的元素个数
    uint_fast64_t my_balance = balance[info.rank];
    vector<uint_fast64_t> local(my_balance);
    // 计算负载的前缀和，每个数代表对应进程开始读的位置
    vector<uint_fast64_t> prefixes = get_prefix_sum<uint_fast64_t>(balance);
    // 每个数8个字节，指针定位时乘上8（加1是为了跳过开头表示元素总数的那个数）
    in.seekg((prefixes[info.rank] + 1) * 8, ios::beg);
    for (auto& x: local)
        read(in, x);
    return local;
}
```

代码清单 1. 并行读取元素

1、 本地数据排序

排序原则上采用任何排序算法都可以，但是由于数据量比较大，如果使用空间复杂度较大的算法（如快速排序和归并排序），就很容易超出内存限制，导致程序崩溃。因此，实验中使用了 STL 的 stable_sort，它使用桶排序算法对序列原地排序，并且时间复杂度是 $O(n \log n)$。

2、 按进程数 p 等间隔采样

每个进程从其局部有序序列中选取 p 个样本，因此总共有 p^2 个样本，取数间隔为 $\frac{n}{p^2}$。

```
stable_sort(local.begin(), local.end());

int num_samples = info.comm_size * info.comm_size;
vector<uint_fast64_t> sample = copy_every_n(local, n / num_samples);
```

代码清单 2. 排序与采样

（二）划分主元

1、 收集样本：一个进程（0 号）用 MPI_Gatherv 收集样本并对所有样本进行排序；

2、 采样获得主元：按进程数 p 对全体样本等间隔采样；

3、 用 MPI_Bcast 广播主元。

```
vector<uint_fast64_t> global_sample;     // 存储全部样本
// 收集每个进程的样本
Gather(sample, global_sample, MPI_UINT64_T, 0, MPI_COMM_WORLD);
// 存储主元，p-1个
vector<uint_fast64_t> pivots(info.comm_size - 1);
if (info.rank == 0) {
    // 对所有样本排序，使用归并排序
    auto it = global_sample.begin();
    for (int j = 0; j < info.comm_size - 1; ++j) {
        it += info.comm_size;
        inplace_merge(global_sample.begin(), it, it + info.comm_size);
    }

    // 从第p个开始，等间隔p采样主元，最终采得p-1个
    pivots = copy_every_n(global_sample, info.comm_size, info.comm_size);
}
// 广播主元
Bcast(pivots, MPI_UINT64_T, 0, MPI_COMM_WORLD);
```

代码清单 3. 划分主元并广播

（三）交换数据

1、 本地数据分块

将有序数组 a 中的元素，以数组 d 中的元素为分界线，划分成多个段。算法首先变量 d 中的每个分界点 x，将小于或等于 x 的元素存储在一维数组 seg 中，这个 seg 数组就是一段，最终所有的 seg 数组组成一个二维数组。

```
template <typename T>
vector<vector<T>> divide_seg(const vector<T>& a, const vector<T>& d) {
    vector<vector<T>> res;
    auto it = a.begin();
    for (T x: d) {
        vector<T> seg;
        while (it != a.end() && *it <= x) {
            seg.push_back(*it);
            ++it;
        }
        res.push_back(seg);
    }
    // 处理最后由于it==a.end()退出的情况
    res.push_back(vector<T>(it, a.end()));
    return res;
}
```

代码清单 4. 数组分段

2、 全交互

该过程每个进程将自己局部序列的 p 个段按顺序发给 p 个进程，使用一个循环来将 p 个段 Gather 给对应进程，并返回各个进程第 i 段的长度，用数组 seg_length 记录，它的作用是在之后的归并排序中计算每个有序子段的范围。

```
vector<vector<uint_fast64_t>> local_blocks = divide(local, pivots);
// 存储全交互结果
vector<uint_fast64_t> local2;
// 存储从各个进程收到的段长度
vector<int> seg_length;
for (int i = 0; i < info.comm_size; ++i) {
    // 向进程i聚集段i，并返回从各个进程收到的元素个数（段长度）
    vector<int> tmp = Gatherv(local_blocks[i], local2, MPI_UINT64_T,
                              i, MPI_COMM_WORLD);
    if (info.rank == i)
        seg_length = tmp;
}
```

代码清单 5. 全交互

（四）归并排序

由于从各个进程收集到的子序列都是有序的，因此可以充分利用这一特性进行归并排序，而不是对整个序列应用其他排序算法。此外，为了降低空间复杂度，使用 STL 的原地归并算法，而不是普通归并（利用第三个数组存储归并中间结果）。

```
auto it = local2.begin();
for (int j = 0; j < info.comm_size - 1; ++j) {
    it += seg_length[j];
    inplace_merge(local2.begin(), it, it + seg_length[j + 1]);
}
```

代码清单 6. 有序子段归并

四、实验结果及分析