并行计算 Lab2: MPI

1. 实验简介

MPI(Message Passing Interface)是目前最重要的一个基于消息传递的并行编程工具,它具有移植性好、功能强大、效率高等许多优点,而且有多种不同的免费、高效、实用的实现版本,几乎所有的并行计算机厂商都提供对它的支持,成为了事实上的并行编程标准。

Ubuntu 下配置: Ubuntu上MPI运行环境的配置

Windows 下配置相对简单,但仍需命令行运行,命令为: mpiexec.exe -n 3 .\PSRS.exe

2. PSRS 排序

2.1 思想

(1) 均匀划分: 将 n 个元素 A[1..n] 均匀划分成 p 段,每个 pi 处理 A[(i-1)n/p+1..in/p]

(2) 局部排序: pi 调用串行排序算法对 A[(i-1)n/p+1..in/p] 排序

(3) 选取样本: pi 从其有序子序列 A[(i-1)n/p+1..in/p] 中选取 p 个样本元素

(4) 样本排序:用一台处理器对 p^2 个样本元素进行串行排序

(5) 选择主元: 用一台处理器从排好序的样本序列中选取 p-1 个主元, 并播送给其他 pi

(6) **主元划分**: pi 按主元将有序段 A[(i-1)n/p+1..in/p] 划分成 p 段

(7) 全局交换: 各处理器将其有序段按段号交换到对应的处理器中

(8) 归并排序: 各处理器对接收到的元素进行归并排序

2.2 核心代码

step 4. 采样排序 & step 5. 选择主元

与 OpenMP 不同的是,每个线程选取了样本后,需要通过消息传递发送给线程 0,由线程 0 统一排序并选择主元,再将主元广播给各个线程。

```
int* global_samples = (int*)malloc(NUM_THREADS * NUM_THREADS * sizeof(int));
int* pivots = (int*)malloc(NUM_THREADS * sizeof(int));
MPI_Gather(samples, NUM_THREADS, MPI_INT, global_samples, NUM_THREADS, MPI_INT,
0, MPI_COMM_WORLD);
if (id == 0) {
    // step 4. 采样排序
    std::sort(global_samples, global_samples + NUM_THREADS * NUM_THREADS);
    // step 5. 选择主元
    for (int i = 0; i < NUM_THREADS - 1; i++) {
        pivots[i] = global_samples[(i + 1) * NUM_THREADS];
    }
}
MPI_Bcast(pivots, NUM_THREADS - 1, MPI_INT, 0, MPI_COMM_WORLD);</pre>
```

step 6. 主元划分

每个线程划分自己的元素,根据与主元的大小关系。

有两个关键的数组:

```
int local_parts[NUM_THREADS][N] = { 0 };
int local_count[NUM_THREADS] = { 0 };
```

第一个数组记录了当前线程负责的数据中,要传给线程 id 的数据有 local_count[id] 个,分别是 local_parts[id][]

step 7. 全局交换

由 step 6 可知,每个线程需要负责的就是所有 local_parts[id] 的数据,这是一个多对多的通信,且通信的个数不固定。所以采用 MPI_Alltoallv.

但是在调用上面的函数之前,需要做一些准备工作:

- 开辟发送 buffer 和接收 buffer;
- 准备四个数组,记录了每对传输要传多少的 send_counts , send_displs , recv_counts , recv_displs

所以核心代码如下:

```
for (int i = 0; i < NUM_{THREADS}; i++) {
    send_counts[i] = local_count[i];
}
MPI_Alltoall(send_counts, 1, MPI_INT, recv_counts, 1, MPI_INT, MPI_COMM_WORLD);
send_displs[0] = recv_displs[0] = 0;
for (int i = 1; i < NUM_{THREADS}; i++) {
    send_displs[i] = send_displs[i - 1] + send_counts[i - 1];
    recv_displs[i] = recv_displs[i - 1] + recv_counts[i - 1];
}
for (int i = 0; i < NUM_THREADS; i++) {
     for (int j = 0; j < send\_counts[i]; j++) {
        send_buf[send_displs[i] + j] = local_parts[i][j];
    }
}
MPI_Alltoallv(send_buf, send_counts, send_displs, MPI_INT, recv_buf,
recv_counts, recv_displs, MPI_INT, MPI_COMM_WORLD);
```

step 8. 归并排序

之前的七个步骤完成后,每个线程有一个一维数组 recv_buf, 按 recv_couts 分段有序。

所以最后的工作就是将 recv_buf 分段 merge, 最终写入到 A 数组。

归并排序与 Lab1 的 PSRS 基本相同。不同的是在排序结束后,需要将排序结果发送到线程 0,由线程 0 写回到 A 数组。由于每个线程负责的归并排序的数据数量不尽相同,所以需要使用 MPI_Gatherv,方式与上面的 MPI_Alltoallv 比较类似。

核心代码:

```
MPI_Gather(&result_count, 1, MPI_INT, A_counts, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (id == 0) {
    A_displs[0] = 0;
    for (int i = 1; i < NUM_THREADS; i++) {
        A_displs[i] = A_displs[i - 1] + A_counts[i - 1];
    }
}
MPI_Gatherv(result, result_count, MPI_INT, A, A_counts, A_displs, MPI_INT, 0,
MPI_COMM_WORLD);</pre>
```

step 8 这一部分的难度并不大,但确是我花时间最长的,因为出了一个比较低级的错误,在把实验 1 的归并排序代码复制过来后,源代码里面的 i 与这里的 index 混淆了,导致无法运行,调试起来比较困难。

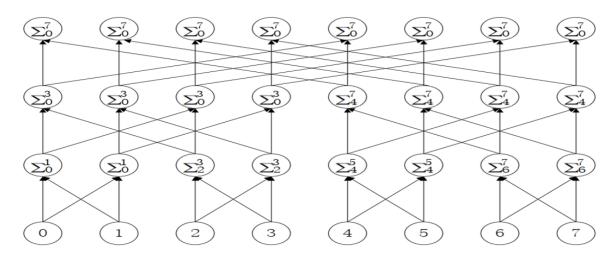
2.3 运行结果

xxa@ubuntu:~/Desktop/Parallel-Computing-Labs/Lab2\$ mpicc PSRS.cpp -o PSRS xxa@ubuntu:~/Desktop/Parallel-Computing-Labs/Lab2\$ mpirun -np 3 ./PSRS 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

3. 求全和:碟式全和

3.1 概念

(1) 蝶式全和的示意图如下:由于使用了重复计算,共需 logN 步。



3.2 核心代码

假设 N 为 2 的幂次。

核心思想类似线段树,首先是跨步为 2^0 的两个结点通信,求和;然后是跨步为 2^1 的两个结点通信,求和……直到跨步为 $\frac{N}{2}$ 的两个节点通信,求和。至此,所有节点均保存了全和。

所以核心代码为:

```
int sum = A[id], nextsum = 0;
int cnt = 1; // 跨步
int chatwith;
while (cnt <= N / 2) {
    if (id % (cnt * 2) < cnt) chatwith = id + cnt;
    else chatwith = id - cnt;
    MPI_Sendrecv(&sum, 1, MPI_INT, chatwith, 0, &nextsum, 1, MPI_INT, chatwith,
0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    sum = sum + nextsum;
    cnt = cnt << 1;
}</pre>
```

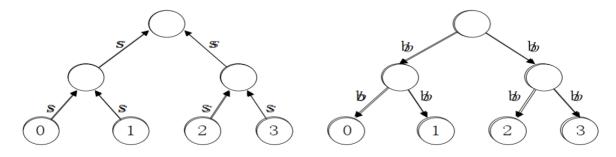
3.3 运行结果

```
xxa@ubuntu:~/Desktop/Parallel-Computing-Labs/Lab2$ mpicc SUM1.cpp -o SUM1
xxa@ubuntu:~/Desktop/Parallel-Computing-Labs/Lab2$ mpirun -np 8 ./SUM1
thread id: 1: sum: 28
thread id: 3: sum: 28
thread id: 5: sum: 28
thread id: 7: sum: 28
thread id: 2: sum: 28
thread id: 4: sum: 28
thread id: 6: sum: 28
thread id: 6: sum: 28
```

4. 求全和: 二叉树方式

4.1 概念

(2) 二叉树方式求全和示意图如下: 需要 2logN 步。



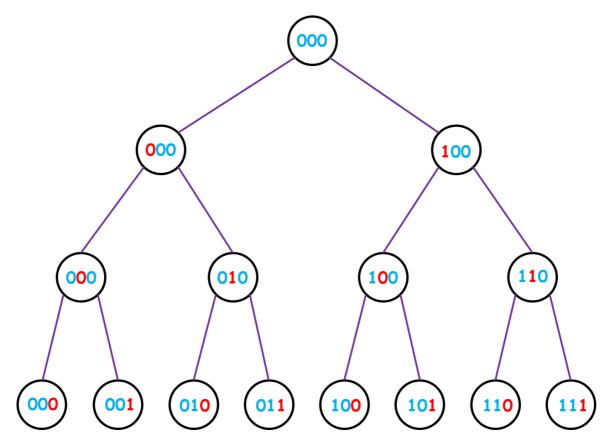
4.2 核心代码

核心思想:

首先自底向上,每一层相邻结点求和,和发给父节点(此处用 id 较小的线程提升为父节点)。

然后自顶向下,将全和逐层下传,直到传给各个叶结点(即各个线程)。

以八个线程为例,各线程的编号为:



可以看到,若从下向上计数,则第 k 层相互通信的两个线程的编号的第 k 为不同,且求和后结果传给第 k 位为 0 的线程,该线程提升为父节点。

自底向上的核心代码:

```
while (cnt < N) {
    flag = id & cnt;
    if (flag == 0) {
        MPI_Recv(&nextsum, 1, MPI_INT, id + cnt, 0, MPI_COMM_WORLD,

MPI_STATUS_IGNORE);
    sum = sum + nextsum;
}
else {
        MPI_Send(&sum, 1, MPI_INT, id - cnt, 0, MPI_COMM_WORLD);
}
cnt = cnt << 1;
}</pre>
```

自顶向下的核心代码:

```
cnt = cnt >> 1;
while (cnt >= 1) {
    flag = id & cnt;
    if (flag == 0) {
         MPI_Send(&sum, 1, MPI_INT, id + cnt, 0, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(&sum, 1, MPI_INT, id - cnt, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
    }
    cnt = cnt >> 1;
}
```

4.3 结果

```
xxa@ubuntu:~/Desktop/Parallel-Computing-Labs/Lab2$ mpicc SUM2.cpp -o SUM2
xxa@ubuntu:~/Desktop/Parallel-Computing-Labs/Lab2$ mpirun -np 8 ./SUM2
thread id: 0, sum: 28
thread id: 1, sum: 28
thread id: 2, sum: 28
thread id: 3, sum: 28
thread id: 4, sum: 28
thread id: 4, sum: 28
thread id: 6, sum: 28
thread id: 7, sum: 28
thread id: 5, sum: 28
```

5. 附录

● [PB20061343_徐奥_实验二.pdf]: 实验报告

• PSRS.cpp: PSRS排序

SUM1.cpp: 求全和: 碟式全和SUM2.cpp: 求全和: 二叉树方式