中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

实验	MPI并行应用	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.5.17

实验目的

- MPI并行应用: 使用MPI对快速傅里叶变换进行并行化
- parallel_for并行应用分析: 对于Lab6实现的paralle_for版本heated_plate_openmp应用(a), 改变并行规模(线程数)及问题规模(N), 分析程序的并行性能. 实验Valgrind massif工具集采集并分析并行程序的内存消耗

实验过程和核心代码

1. MPI对快速傅里叶变换并行化

由于初始化数据过程不计入运行时间作为性能评测, 所以由主进程进行初始化后广播给其他进程

```
// initialize in prank 0
if(prank == 0) {
    if (first) {
        for (int i = 0; i < 2 * n; i = i + 2) {
            x[i] = ggl(&seed);
            x[i + 1] = ggl(&seed);
            z[i] = x[i];
            z[i + 1] = x[i + 1];
        }
    } else {
        fill(x, x + 2 * n, 0);
        fill(z, z + 2 * n, 0);
    }
    //  Initialize the sine and cosine tables.
    cffti(n, w);
}
// 广播w给其他进程
MPI_Bcast(w, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// 广播x
MPI_Bcast(x, 2*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

在mian函数中, 输出都由主进程执行, 其他进行不执行输出. 由于运行时间主要测量`cfft2()`的时间, 所以并行化的主要内容在`cfft2()`中
在`cttf2`中主要运行`step()`进行迭代, 而`step()`会计算并改变第5,6个参数的结果. 所以在`cttf2`中每次计算一个step后要将改变的结果同步给所有的进程.

```
step(n, mj, x, &x[(n / 2) * 2 + 0], y, &y[mj * 2 + 0], w, sgn);
// step 计算出新的y, 进行广播
MPI_Bcast(y, 2 * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

step(n, mj, y, &y[(n / 2) * 2], x, &x[mj * 2], w, sgn);
// step 计算出新的x, 进行广播
MPI_Bcast(x, 2*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

而在`step()`函数中, 主要并行化循环计算的过程, 每一个循环计算完成后, 非主线程将自己计算的数据发送给主线程, 主线程接收数据并放在对应的位置.
需要注意的时, 当循环不足以分配给所有进程时(即`local_n == 0`), 由前`lj`个进程进行计算, 每个进程只计算一个循环的内容. 并且在计算完毕后注意主线程只能进行计算线程数次接收, 如果不匹配就会造成程序错误. 主进程接收次数可以由`min(lj, psize)`来确定.

```
int local_n = lj / psize;
int start = 0, end = 0;

// 分配计算内容
// 无法满足每个进程都分配到计算内容
if(local_n == 0) {
    local_n = 1;
    if(prank < lj) {
        start = prank;
        end = start + 1;
    }
} else {
    start = prank * local_n;
    end = start + local_n;
}

for (int j = start; j < end; j++) {
    int jw = j * mj;
    int ja = jw;
    int jb = ja;
    int jc = j * mj2;
    int jd = jc;

    double wjw[2] = {w[jw * 2 + 0], w[jw * 2 + 1]};
    if (sgn < 0.0) {
        wjw[1] = -wjw[1];
    }

    for (int k = 0; k < mj; k++) {
        c[(jc + k) * 2 + 0] = a[(ja + k) * 2 + 0] + b[(jb + k) * 2 + 0];
        c[(jc + k) * 2 + 1] = a[(ja + k) * 2 + 1] + b[(jb + k) * 2 + 1];
        double ambr = a[(ja + k) * 2 + 0] - b[(jb + k) * 2 + 0];
        double ambu = a[(ja + k) * 2 + 1] - b[(jb + k) * 2 + 1];
        d[(jd + k) * 2 + 0] = wjw[0] * ambr - wjw[1] * ambu;
        d[(jd + k) * 2 + 1] = wjw[1] * ambr + wjw[0] * ambu;
    }
    // 计算完毕后 发送/接收数据
    if(prank == 0) {
        for(int i = 1; i < min(lj, psize); ++i) {
            MPI_Recv(&c[(i*local_n+j) * mj2 * 2], 2 * mj, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(&d[(i*local_n+j) * mj2 * 2], 2 * mj, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
    else if(prank != 0 && prank < lj) {
        MPI_Send(&c[jc * 2], 2 * mj, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&d[jd * 2], 2 * mj, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    }
}
```

2. parallel_for并行应用分析

修改程序, 通过运行参数来指定线程数与问题规模

```
m = 512;
n = 512;
int threads_num = 4;
if(argc >= 2) {
    threads_num = std::stoi(argv[1]);
}
if(argc >= 3) {
    m = std::stoi(argv[2]);
    n = std::stoi(argv[2]);
}
```

使用脚本来运行不同线程数以及不同规模的程序

```
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 1 128
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 2 128
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 4 128
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 8 128
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 16 128

valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 1 256
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 2 256
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 4 256
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 8 256
valgrind --tool=massif --stacks=yes ./heated_plate_pthreads 16 256

valgrind --tool=massif --stacks=yes  ./heated_plate_pthreads 1 512
valgrind --tool=massif --stacks=yes  ./heated_plate_pthreads 2 512
valgrind --tool=massif --stacks=yes  ./heated_plate_pthreads 4 512
valgrind --tool=massif --stacks=yes  ./heated_plate_pthreads 8 512
valgrind --tool=massif --stacks=yes  ./heated_plate_pthreads 16 512
```

每条命令都会生成 `massif.out.*` 文件, 使用ms_print可以输出图表对运行栈内存进行分析, 使用脚本将所有out文件打印到 `valgrind.txt` 中

```
# 对所有生成的 massif.out.* 文件执行 ms_print
for f in massif.out.*; do
    ms_print "$f" >> valgrind.txt
done
```

结果见 `valgrind.txt`

实验结果

虚拟机处理器个数: 4
时间单位: s

MPI对快速傅里叶变换并行化

详细结果可见 `fft_parallel_result.txt`. 首先观察Error与串行结果相同, 说明并行化后程序仍然是正确的.

N=1048576 的运行时间表格:

进程数	串行	并行
1	0.254157	0.220557
2	-	0.458515
4	-	1.1982
8	-	2.29178
16	-	2.5499

可以看到随着进程数增加, 运行时间几乎呈倍数增长. 可能是由于 `step()` 方法中每个循环都要同步一次数据, 导致MPI通信的开销非常大, 从而使运行时间比 串行/一个进程要慢.

在并行结果中, 进程数为1时运行时间与串行接近, 而进程数为1时在 `step()` 方法中不需要执行 `MPI_Send` 和 `MPI_Recv`, 即省略了通信开销. 进一步证明多个进程并行变慢是MPI的通信导致的.

由于在 `step()` 方法使用了 `MPI_Send` 和 `MPI_Recv`, 而这两个方法都是阻塞的, 即必须要确定发送/接收后才会执行下一条指令, 否则就一直等待. 这使得每个循环中执行的快的进程仍要等到主进程执行完毕接收消息后才能进入下一个循环. 也会导致通信开销增大.

parallel_for并行应用分析

所有程序在栈内存都是逐步增大.
不同线程数和问题规模的栈内存峰值:

线程数\规模	128	256	512
1	2.069	5.262	11.14
2	3.781	9.476	18.17
4	7.275	17.75	32.24
8	14.22	34.59	60.38

线程数\规模	128	256	512
16	28.10	68.12	116.8

在相同问题规模下, 由于每创建一个线程都要分配**线程的栈空间**, 所以线程数增加时栈内存占用增加与线程增量呈正相关.
在相同线程时, 随着问题规模增大栈内存占用也增大, 栈内存应该只与函数的调用相关(局部变量, 调用深度). 问题规模大时, 需要的函数调用次数越多, 也造成内存占用增大.

实验感想

在本实验中我使用并行来加速一个典型算法: 快速傅里叶变换. 虽然没有显著增加运行效率, 但是也分析了原因: 进程间频繁的通信反而使并行的性能不如串行程序. 所以在以后使用并行解决问题时要记住一个关键点, 如何进行更好的数据划分来尽量减少通信的次数.
此外, 我还学习了使用 Valgrind massif 工具集对程序运行内存进行监测, 这有助于我直观了解程序运行时内存是如何变化的. 并以此来分析程序是否按照预期进行内存的分配.