实验 1

PB17111626 秦亚璇

1 实验题目:

利用 MPI,OpenMP 编写简单的程序,测试并行计算系统性能

2 实验环境(操作系统,编译器,硬件配置等):

操作系统 Ubuntu 16.04,编译器 g++，硬件配置 ThinkPad X1 Carbon(16G)

3 算法设计与分析(写出解题思路和实现步骤)

首先是求 n 以内的素数的算法，我们将会有 m 个进程并行计算，故将 n 均分为 m 个分片，进程 pi 要完成的事情是计算(i - 1) * (n / m)到 i* (n / m)的所有数字中素数个数的统计，最终将他们全部 send 到 rank0 进行累加，其实就完成了分解到合并的过程，至于如何判断一个素数的算法过于基础，此处不做赘述。

第二个实验，利用近似公式有 f(x) = 4/(1 + x^2),pi = $\frac{1}{N}\times\sum_{i=1}^{N} f\left(\frac{i-0.5}{N}\right)$,依然是将 N 进行 m 分片之后，每个进程计算一个分片的数据最终在 rank0 的部分进行加和平均即可，不做赘述。

4 核心代码(写出算法实现的关键部分,如核心的循环等)

求 n 以内的素数（MPI）:

```
MPI_Init(&argc, &argv);//initialize
MPI_Comm_size(MPI_COMM_WORLD, &numofprocess);//get the number of p
MPI_Comm_rank(MPI_COMM_WORLD, &rankid);//get the id of the rank
MPI_Barrier(MPI_COMM_WORLD);
begin = MPI_Wtime();// align all processes and get the start time
int fenpian = n / numofprocess;
MPI_Status status;
int cnt = 0,partcnt = 0;
int i;
if(rankid == 0)
{
    for(i = rankid * fenpian;i < (rankid + 1) * fenpian;i++)
        {
        if(issushu(i))
        cnt += 1;
    }
        for(i = 1;i < numofprocess;i++)
    {
        MPI_Recv(&partcnt,1,MPI_INT,i,0,MPI_COMM_WORLD,&status);
        cnt += partcnt;
    }
}
else
{
    for(i = rankid * fenpian;i < (rankid + 1) * fenpian;i++)
        {
        if(issushu(i))
        cnt += 1;
    }
    MPI_Send(&cnt,1,MPI_INT,0,0,MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
    if(rankid == 0)
{
    printf("the time is %.5f\n",end - begin);
    printf("the number is %d\n",cnt);
}
MPI_Finalize();
```

求 n 以内的素数（OMP）:

```
ntu: ~/Desktop
#include <stdio.h>
#include <omp.h>
#include <cmath>
#include <time.h>
int main(int argc, char* argv[]) {
    int n = 500000;
    omp_set_num_threads(8);
    int i, j, num = 0;
    double begin, end;
    begin = omp_get_wtime();
    #pragma omp parallel for reduction(+:num) private(j)
    for (i=2; i<=n; i++) {
        int s = sqrt(i * 1.0);
        for (j=2; j<=s; j++)
        {
            if (i % j == 0)
                break;
        }
        if (j > s)
            num++;
    }
    end = omp_get_wtime();
    printf("%d\n",num);
    printf("time%.9f\n",end-begin);
    return 0;
}
```

求 pi 的近似值（MPI）：

```
double f(double x)
{
    return 4.0 / (1.0 + x * x);
}

int main(int argc, char* argv[]) {
    int n = 1000;
    int numofprocess,rankid;
    double begin, end;
    MPI_Init(&argc, &argv);//initialize
    MPI_Comm_size(MPI_COMM_WORLD, &numofprocess);//get the number of pro
    MPI_Comm_rank(MPI_COMM_WORLD, &rankid);//get the id of the rank
    MPI_Barrier(MPI_COMM_WORLD);
    begin = MPI_Wtime();// align all processes and get the start time
    int fenpian = n / numofprocess;
    MPI_Status status;
    double cnt = 0,partcnt = 0;
    int i;
    if(rankid == 0)
    {
        for(i = rankid * fenpian;i < (rankid + 1) * fenpian;i++)
                {
            cnt += f((i - 0.5) / n);
        }
            for(i = 1;i < numofprocess;i++)
        {
            MPI_Recv(&partcnt,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&status);
            cnt += partcnt;
        }
        cnt = cnt / n;
    }
    else
    {
        for(i = rankid * fenpian;i < (rankid + 1) * fenpian;i++)
                {
            cnt += f((i - 0.5) / n);
        }
        MPI_Send(&cnt,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    end = MPI_Wtime();
        if(rankid == 0)
    {
        printf("the time is %.9f\n",end - begin);
        printf("the number is %.5f\n",cnt);
    }
    MPI_Finalize();
    return 0;
}
```

求 pi 的近似值（OMP）：

```c
#include <stdio.h>
#include <omp.h>
#include <cmath>
#include <time.h>
int main(int argc, char* argv[]) {
    int n = 1000;
    omp_set_num_threads(1);
    int i, j, num = 0;
    double sum = 0;
    double begin, end;
    begin = omp_get_wtime();
    #pragma omp parallel for reduction(+:sum)
    for (i=2; i<=n; i++) {
        double x = (i - 0.5)/n;
        sum += 4.0/(1.0 + x * x);
    }
    end = omp_get_wtime();
    printf("%.3f\n",sum / n);
    printf("time%.9f\n",end-begin);
    return 0;
}
```

5 实验结果

求素数-mpi 运行时间

规模\进程数	1	2	4	8
1000	0.00016s	0.00006s	0.00002s	0.04093s
10000	0.00040s	0.00023s	0.00013s	0.02398s
100000	0.00842s	0.00505s	0.00275s	0.02080s
500000	0.06895s	0.04270s	0.03210s	0.04786s

求素数-mpi 加速比

规模\进程数	1	2	4	8
1000	1	2.67	8	0.004
10000	1	1.74	3.08	0.017
100000	1	1.67	3.06	0.405
500000	1	1.61	2.15	1.44

求素数-omp 运行时间

规模\进程数	1	2	4	8
1000	0.00023s	0.00017s	0.00061s	0.00033s
10000	0.00079s	0.00050s	0.00047s	0.00112s
100000	0.01547s	0.01197s	0.00437s	0.00446s
500000	0.07647s	0.04437s	0.03558s	0.02439s

求素数-omp 加速比

规模\进程数	1	2	4	8
1000	1	1.35	0.38	0.52
10000	1	1.58	1.68	0.71
100000	1	1.29	3.54	3.47
500000	1	1.72	2.15	3.14

求 pi-mpi 运行时间

规模\进程数	1	2	4	8
1000	0.000005s	0.00006s	0.00002s	0.02209s
10000	0.00004s	0.00003s	0.00003s	0.04708s

50000	0.00021s	0.00011s	0.00006s	0.02321s
100000	0.00041s	0.00027s	0.00034s	0.01996s

求 pi-mpi 加速比

规模\进程数	1	2	4	8
1000	1	0.083	0.25	0.00022
10000	1	1.33	1.33	0.00085
50000	1	1.91	3.5	0.009
100000	1	1.52	1.21	0.021

求 pi-omp 运行时间

规模\进程数	1	2	4	8
1000	0.00001s	0.00014s	0.00022s	0.00062s
10000	0.00007s	0.00019s	0.00052s	0.00066s
50000	0.00031s	0.00023s	0.00034s	0.00076s
100000	0.00063s	0.00046s	0.0004s	0.00095s

求 pi-omp 加速比

规模\进程数	1	2	4	8
1000	1	0.071	0.045	0.016
10000	1	0.37	0.13	0.11
50000	1	1.35	0.91	0.41
100000	1	1.37	1.58	0.66

6 分析与总结

1. 从实验结果的纵向（规模大小）来分析，同一并行进程数（>=2）下，问题规模越大，并行所获得的加速效果越明显，加速比越大。从横向（进程数）来进行分析，可以看到一定规模的问题加速比是一个对勾状函数，并非并行程度越大越好，在某一个进程数附近能够达到最优的加速比，之后再扩大进程数会使进程减慢，该现象的主要原因是，在计算比较简单，问题规模比较小的时候，进程之间的通信开销占比会很大，影响运行效率，而在问题规模很大时，并行能够节约的计算时间更多，此时的通信开销就不值一提了。故而，我们需要在面对较大规模的问题时积极并行编程，而问题较为简单时不需要多此一举。
2. Openmp 是相对更为简单的并行编程方式，基本上像是封装好的并行编程包，但我更倾向于使用 MPI 进行完全人工的进程信息传递控制，虽然看起来这种程序耗时会长一些（我思考了一下原因，大概是我还没有掌握 Bcast 的魔法……）