17级并行与分布式计算

HW4 17341137 宋震鹏

$1 实验要求

1. Consider a sparse matrix stored in the compressed row format (you may find a description of this format on the web or any suitable text on sparse linear algebra). Write an OpenMP program for computing the product of this matrix with a vector. Download sample matrics from the Matrix Market (http://math.nist.gov/MatrixMarket/) and test the performance of your implementation as a function of matrix size and number of threads.
2. Implement a producer-consumer framework in OpenMP using sections to create a single producer task and a single consumer task. Ensure appropriate synchronization using locks. Test your program for a varying number of producers and consumers.

$2 实验分析

实验一

要求使用OpenMP实现矩阵向量乘的并行计算，同时检测性能。

根据分析，该问题大致可以表达为：

- 矩阵的导入
- 稀疏矩阵的存储方式了解，矩阵向量乘的实现
- 利用`omp.h`库实现算法

实验二

利用OpenMP实现生产者-消费者模型，确保互斥同步。

$3 实验代码

实验一

```
#include <iostream>
#include <string>
#include <ctype.h>
#include <stdio.h>
#include <omp.h>
#include <string.h>
#include <chrono>
#include <ctime>

using namespace std;

double seqcal(double* res, int* tmp_r, int* tmp_c, double* tmp_v, double* vec, int size) {
```

```
    using namespace std::chrono;

    steady_clock::time_point t1 = steady_clock::now();

    for (int i = 0; i < size; i++) {
        res[tmp_r[i]] = res[tmp_r[i]] + tmp_v[i] * Vec[tmp_c[i]];
    }

    steady_clock::time_point t2 = steady_clock::now();
    duration<double> time_span = duration_cast<duration<double>>(t2 - t1);
    return time_span.count()*1000;
}

double parallelcal(double* res, int* tmp_r, int* tmp_c, double* tmp_v, double* Vec, int
size, int tnums) {

    using namespace std::chrono;

    steady_clock::time_point t1 = steady_clock::now();

    #pragma omp parallel for num_threads(tnums)
    for (int i = 0; i < size; i++) {
        res[tmp_r[i]] = res[tmp_r[i]] + tmp_v[i] * Vec[tmp_c[i]];
    }

    steady_clock::time_point t2 = steady_clock::now();
    duration<double> time_span = duration_cast<duration<double>>(t2 - t1);
    return time_span.count()*1000;
}

double* Createvec(int n, int mod) {
    double *vec = new double[n];
    int i = 0;
    while (i!=n+1) {
        vec[i++] = rand() % mod;
    }
    return vec;
}

int main() {
    FILE* fp;
    char buf[250];
    int rownum, colnum, totnum;
    int tmp_r[10005] = { 0 }, tmp_c[10005] = { 0 };
    double tmp_v[10005] = { 0 };
    double res[100005] = { 0 };
    double *anscopy = new double[sizeof(res)];
    fp = fopen("1138_bus.mtx", "r");
    if (fp != nullptr) {
        fgets(buf, 250, fp);
        cout << buf;
        fscanf(fp, "%d %d %d", &rownum, &colnum, &totnum);
        cout << "Row:" << rownum << endl;
```

```
        cout << "Col:" << colnum << endl;
        cout << "Tot:" << totnum << endl;

        double *Vec = Createvec(rownum, 1000);

        for (int i = 0; i < totnum; i++) {
            fscanf(fp, "%d %d  %lf", &tmp_r[i], &tmp_c[i], &tmp_v[i]);
        }

        double seq_t = seqcal(res, tmp_r, tmp_c, tmp_v, Vec, totnum);
        cout << "Sequential: " << seq_t << "ms." << endl;
        memcpy(anscopy, res, sizeof(res));

        while(1){
            int nums;
            memset(res, 0, sizeof(res));
            cout<<"Parallel with ? threads? (Enter Thread numbers)"<<endl;
            cin >> nums;
            if (nums == -1){
                cout << "Exit Parallel Mode."<<endl;
                break;
            }
            else{
                double par_t = parallelcal(res, tmp_r, tmp_c, tmp_v, Vec, totnum, nums);
                cout << "Parallel: " << par_t << " ms.";
                cout << " (" << seq_t/par_t << " faster ) " << endl;
            }
        }
    }
    fclose(fp);
    cout << "Result? (Y/N)" << endl;
    char tmp;
    getchar();
    if((tmp = getchar())&&(tmp=='Y'||tmp=='y')){
        for (int i = 0; i < rownum; i++) {
            cout << anscopy[i] << endl;
        }
    }
    return 0;
}
```

实验二

```
#include <stdio.h>
#include <omp.h>
#define queue_size 10

int in = 0, out = 0;    // 队首、队尾；从队首取，从队尾存入
void producer(int); // 生产者函数
void consumer(); // 消费者函数
omp_lock_t pro_lock;
omp_lock_t con_lock;
```

```c
/* 判断队空 */
int queue_is_empty() {
    return (in == out);
}
int queue_is_full() {
    return (out == ((in + 1) % queue_size));
}

/* 主线程 */
int main() {
    int nums_pro, nums_con;
    scanf("%d%d", &nums_pro, &nums_con);
    #pragma omp parallel sections
    {
        #pragma omp section
        producer(nums_pro);

        #pragma omp section
        consumer();
    }
    return 0;
}

void producer(int nums_pro) {
    #pragma omp parallel for num_threads(nums_pro)
    for(int i = 0;i < nums_pro;i++){
        while (1) {
            omp_set_lock(&pro_lock);
            while (queue_is_full()) {}
            printf("producer [%d]:\t no.%d item produced. \n", omp_get_thread_num(), in);
            in = (in + 1) % queue_size;
            #pragma omp flush
            omp_unset_lock(&pro_lock);
        }
    }
}

void consumer() {

    while (1) {
        omp_set_lock(&con_lock);
        while (queue_is_empty()) {}
        printf("consumer [%d]:\t no.%d item consumed. \n", omp_get_thread_num(), out);
        out = (out + 1) % queue_size;
        #pragma omp flush
        omp_unset_lock(&con_lock);
    }
}
```

$4 实验截图

实验一

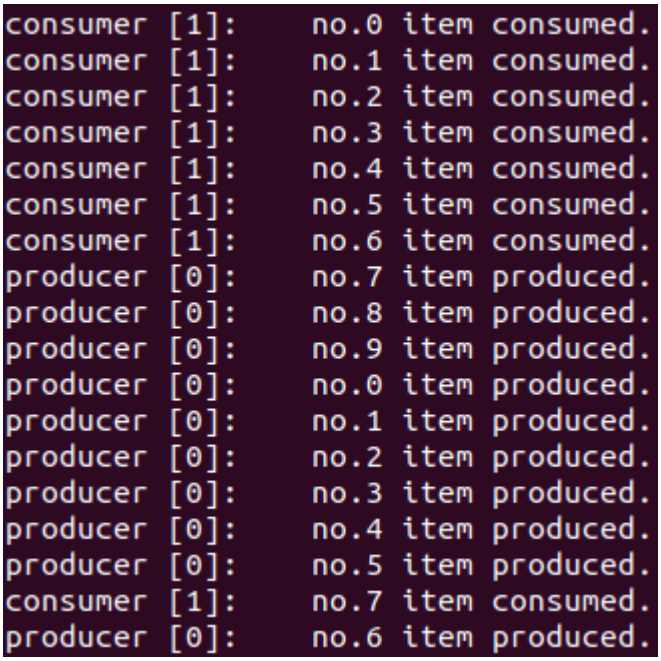
Win下运行：

```
%%MatrixMarket matrix coordinate real symmetric
Row:1138
Col:1138
Tot:2596
Sequential: 0.035374ms.
Parallel with ? threads? (Enter Thread numbers)
2
Parallel: 0.012763 ms. (2.77161 faster )
Parallel with ? threads? (Enter Thread numbers)
4
Parallel: 0.013493 ms. (2.62166 faster )
Parallel with ? threads? (Enter Thread numbers)
8
Parallel: 0.014587 ms. (2.42504 faster )
Parallel with ? threads? (Enter Thread numbers)
```

Linux下运行：

```
szp@szp-virtual-machine:/mnt/hgfs/ShareVmware/OpenMP-examples$ ./test2
%%MatrixMarket matrix coordinate real symmetric
Row:1138
Col:1138
Tot:2596
Sequential: 1.3e-05
Parallel with ? threads? (Enter Thread numbers)
2
Parallel: 0.000336
Parallel with ? threads? (Enter Thread numbers)
4
Parallel: 0.00014
Parallel with ? threads? (Enter Thread numbers)
8
Parallel: 0.000333
Parallel with ? threads? (Enter Thread numbers)
-1
Exit Parallel Mode.
Result? (Y/N)
Y
0
1.30665e+06
7099.18
63697.4
54400.6
-3336.68
45095.9
-5723.4
19637.4
1388.69
-960.881
-7327.66
821.256
421.416
9883.42
1.5712e+06
-11840.5
-15243.9
-22917.9
-5775.08
1.21243e+06
904774
1035.92
316.39
2145.08
836.345
1332.73
745.107
```

实验二



$5 实验总结

实验一

在实验一中，我按照以下顺序组织了代码结构：

导入矩阵 — 显示矩阵信息 — 串行计算 — 不同线程数量下的并行计算 — 输出结果

实验后，发现在现有矩阵规模下，随着并行计算线程数增加，Windows系统下与预期相符；

而在Linux系统下，耗时比串行计算时间大幅增加。

初步分析，认为可能是在较低规模情况下，内存拷贝消耗大量时间，入不敷出。

再做思考，也可能是Linux下虚拟机配制出现问题。

经过本实验，了解了omp库中OpenMP预编译指令的用法。

实验二

在实验二中，利用omp_lock_t结构实现互斥锁，再利用sections子句进行并行化，同时启动生产者、消费者，了解到OpenMP在同步互斥上的功能作用。

17341137 宋震鹏 19/5/7