并行与分布式计算作业

第 4 次作业

姓名：郝裕玮
班级：计科 1 班
学号：18329015

一、 问题描述

利用 Culler 并行程序设计方法计算 1000x1000 的矩阵与 1000x1 的向量之间的乘积，要求清晰地呈现 Culler 并行程序设计的四个步骤，并比较程序在不同阶段具有不同配置时如不同的子任务数量、不同的线程数量、不同的映射方案的性能差别。

二、 解决方案

Culler 并行程序设计的 4 个步骤为：分解 Decomposition，作业 Assignment，编排 Orchestration，映射 Mapping

（1）分解 Decomposition

本次任务可分解为将矩阵 a(1000*1000)第 i 行的元素与矩阵 b(1000*1)进行点乘即可得到结果矩阵 c 的第 i 行元素。即分解为 1000 个子任务。

（2）作业 Assignment

Assignment 主要是为了平衡工作量，降低沟通成本。每个线程可分配矩阵 a 上不同部分的等数量行数与矩阵 b 的点乘，如 4 个线程则各分配 250 行的运算，这里由 OpenMP 自动根据线程数和调度方式来进行分配。

（3）编排 Orchestration

Orchestration 主要是为了建立通信。由于 OpenMP 是共享内存式编程，所以我们只需将矩阵 a,b,c 和矩阵规模 size 设置成共享变量(shared)即可建立通信。

（4）映射 Mapping

Mapping 负责将线程映射到硬件执行单元。调度方式有静态调度，动态调度等方式，默认情况下是均匀分配的静态调度，即线程 0—thread_count-1 均分配到 n/thread_count 次迭代计算（n 代表迭代总次数）。

不同线程数量下的代码如下所示（具体思路及详细分析均包含在代码注释中）详见 Culler1.cpp：

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include <sys/time.h>
#include <time.h>
using namespace std;

#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}
////最后一行将 us 转换为 s，统一单位
//该结构体用于计算并行计算和串行计算的运行时间

void Serial(int **a,int*b,int*c,int size){//串行计算
    int i,j;
    for(i=0;i<=size-1;i++){
        for(j=0;j<=size-1;j++){
            c[i]+=a[i][j]*b[j];
            //将矩阵 a(1000*1000)第 i 行的元素与矩阵 b(1000*1)进行点乘即可得到结果矩阵 c 的第 i 行元素
        }
    }
}

void Parallel(int **a,int*b,int*c,int size,int thread_count){
    int i,j;
```

```
#  pragma omp parallel for num_threads(thread_count) default(none) \
      private(i,j) shared(a,b,c,size)
    //设置循环迭代的 i,j 为私密变量
    //设置 a,b,c 数组和数组大小 size 为各线程之间的共享变量
    for(i=0;i<=size-1;i++){
        for(j=0;j<=size-1;j++){
            c[i]+=a[i][j]*b[j];
        }
    }
}

int main(){
    int size=1000;//数组大小(可修改)
    int i,j;
    int thread_count=4;//并行线程数(可修改)

    //a,b,c 数组的初始化
    int** a;
    a=new int*[size];
    for(i=0;i<=size-1;i++){
        a[i]=new int[size];
    }
    int *b;
    b=new int[size];
    int *c;
    c=new int[size];

    srand(time(NULL));//用时间初始化随机数生成种子
    //对 a,b,c 数组进行随机初始化(a,b 数组的每个元素均为 0-9 的随机整数，c 数组
初始化为全 0)
    for(i=0;i<=size-1;i++){
        for(j=0;j<=size-1;j++){
            a[i][j]=rand()%10;
        }
    }
    for(i=0;i<=size-1;i++){
        b[i]=rand()%10;
    }
    for(i=0;i<=size-1;i++){
        c[i]=0;
    }

    double t1,t2,t3,t4;
```

```
    //记录串行计算时间
    GET_TIME(t1);
    Serial(a,b,c,size);
    GET_TIME(t2);
    cout<<"串行计算时间为："<<t2-t1<<"秒"<<endl;

    //将结果数组 c 重新全部初始化为 0
    for(i=0;i<=size-1;i++){
        c[i]=0;
    }

    //记录并行计算时间
    GET_TIME(t3);
    Parallel(a,b,c,size,thread_count);
    GET_TIME(t4);
    cout<<"并行计算时间为："<<t4-t3<<"秒"<<endl;

    //计算加速比
    cout<<"加速比为："<<(t2-t1)/(t4-t3)<<endl;

    return 0;
}
```

不同子任务数量下的代码大体相同，只需修改 Parallel 函数（具体思路及详细分析均包含在代码注释中）详见 Culler2.cpp：

```
void Parallel(int **a,int*b,int*c,int size,int thread_count){
    int i,j,cnt,div;
    div=10;//设置行数的合并数量
#   pragma omp parallel for num_threads(thread_count) default(none) \
      private(i,j,cnt) shared(a,b,c,size,div)
    //设置循环迭代的 i,j,cnt 为私密变量
    //设置 a,b,c 数组,数组大小 size 和行数合并数量 div 为各线程之间的共享变量
    for(cnt=0;cnt<=size-1;cnt+=div){//a 矩阵的每 div 行合并到一起与矩阵 b 进行点乘运算，而不再单独计算 a 矩阵的每行
        for(i=cnt;i<=cnt+dvi-1;i++){//计算 a 矩阵当前的 div 行(cnt 代表组数，div 代表组内部的行数)
            for(j=0;j<=size-1;j++){
                c[i]+=a[i][j]*b[j];
            }
        }
    }
}
```

不同映射方案下的代码大体相同，只需修改 Parallel 函数（具体思路及详细分析均包含在代码注释中），详见 Culler3.cpp：

```
void Parallel(int **a,int*b,int*c,int size,int thread_count){
    int i,j;
#  pragma omp parallel for num_threads(thread_count) default(none) \
      private(i,j) shared(a,b,c,size) schedule(dynamic)//将调度方式修改为动态调度 dynamic
    //设置循环迭代的 i,j 为私密变量
    //设置 a,b,c 数组和数组大小 size 为各线程之间的共享变量
    for(i=0;i<=size-1;i++){
        for(j=0;j<=size-1;j++){
            c[i]+=a[i][j]*b[j];
        }
    }
}
```

三、 实验结果

以下结果均在超算习堂上运行得出。

不同线程数量下的矩阵运算结果如下图所示：

```
======== ERROR ========

======== OUTPUT ========
线程数量为：4
串行计算时间为：0.00514388秒
并行计算时间为：0.00154185秒
加速比为：3.33617

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
线程数量为：8
串行计算时间为：0.00510883秒
并行计算时间为：0.00108099秒
加速比为：4.72607

======== REPORT ========
```

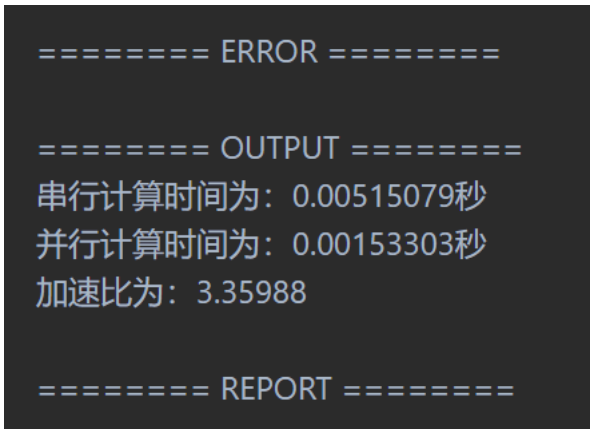
```
======== ERROR ========

======== OUTPUT ========
线程数量为：16
串行计算时间为：0.00510311秒
并行计算时间为：0.00140595秒
加速比为：3.62964

======== REPORT ========
```

由上图结果可知，运算速度随着线程数量的增加先增加后降低。

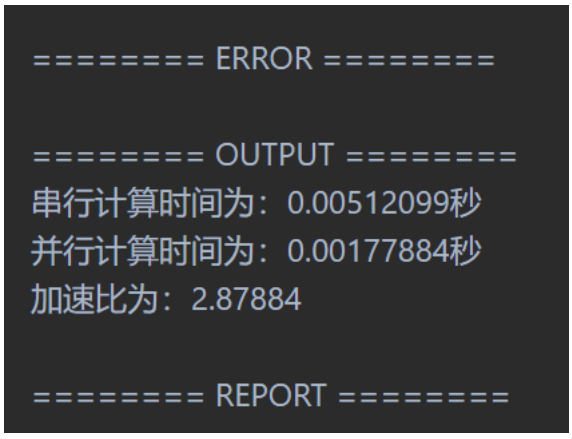
不同子任务数量下的矩阵运算结果如下图所示：

（1）子任务数量为 1000/5=200（div=5）

（2）子任务数量为 1000/20=50（div=20）



（3）子任务数量为 1000/100=10（div=100）



由上图结果可知，运算速度随着子任务数量的减少先增加后降低。

不同映射方案下的矩阵运算结果如下图所示：

（1）静态调度

```
======== ERROR ========

======== OUTPUT ========
线程数量为：4
串行计算时间为：0.00510383秒
并行计算时间为：0.00152206秒
加速比为：3.35323

======== REPORT ========
```

（2）动态调度

```
======== ERROR ========

======== OUTPUT ========
线程数量为：4
串行计算时间为：0.00515103秒
并行计算时间为：0.00431991秒
加速比为：1.19239

======== REPORT ========
```

由上图结果可知，运算速度：静态调度>动态调度。

四、遇到的问题及解决方法

本次实验唯一遇到的问题是自己在将串行程序用 OpenMP 进行并行化时少加了一个 for，写成了下面这种形式：

```
#  pragma omp parallel num_threads(thread_count) default(none) \
    private(i,j) shared(a,b,c,size)
```

因为内部是双重循环，所以必须使用 omp parallel for。所以我这种写法导致并行运算速度甚至慢于串行运算速度，在经过认真检查代码后找到了该问题，并得到了预想中的加速比。