并行与分布式计算作业

第 1 次作业

姓名：郝裕玮
班级：计科 1 班
学号：18329015

一、 问题描述

1、在第一次课程中已经讲到，早期单节点计算系统并行的粒度分为:Bit 级并行，指令级并行和线程级并行。现代处理器如 Intel、ARM、AMD、Power 以及国产 CPU 如华为鲲鹏等，均包含了并行指令集合。1. 请调查这些处理器中的并行（向量）指令集，并选择其中一种如 AVX, SSE 等进行编程练习。此外，现代操作系统为了发挥多核的优势，支持多线程并行编程模型，请利用多线程的方式实现 N 个整数的求和，编程语言不限，可以是 Java，也可以是 C/C++。

2、写一篇文章，描述您专业中的一个研究问题，该问题将受益于并行计算的使用。 提供如何使用并行性的粗略概述。 你会使用任务并行还是数据并行？

二、 解决方案

对于第 1 题：

首先使用 CPU-Z 查看我的电脑所支持的指令集，如下图所示：

所以在本次作业中，我选择 AVX 指令集进行编程联系。

在参考了网上相关资料后，我决定将编程练习的内容定为：计算两个各包含10^6个整数的向量之和并探究其与串行计算的速度关系。

代码及其具体分析如下所示（分析已放在代码注释中）：

```
#include<stdio.h>
#include<immintrin.h>
#include<string.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
#include<sys/time.h>
#define n 1000000

int a[n], b[n],sum1[n],sum2[n];

#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}
//用于计算程序运行时间

//利用 AVX 指令集进行并行向量相加（包含数据存储）
void AVX(){
    int i=0;
    __m256i a1,b1,c;//__m256i:包含若干个整型数字的向量，本题中是每个向量包含8 个 int 型整数
    for(i=0;i<=n-1;i+=8){//正如上文所说，向量 a1,b1 每次可以一次性读取 8 个整数并进行运算，所以循环条件是 i+=8
        a1=_mm256_load_si256((const __m256i*)(a+i));//_mm256_load_si256:从对齐的内存地址加载整形向量
        b1=_mm256_load_si256((const __m256i*)(b+i));
        c=_mm256_add_epi32(a1,b1);//_mm256_add_epi32:对两个整形向量做加法
        _mm256_store_ps((float*)(sum1+i),(__m256)c);//_mm256_store_ps:用于将计算结果等 AVX 寄存器的数据保存到内存中。
        //将结果存储到 sum1 数组中
    }
}

//利用 AVX 指令集进行并行向量相加（不包含数据存储）
void AVX2(){
    int i=0;
    __m256i a1,b1,c;
    for(i=0;i<=n-1;i+=8){
        a1=_mm256_load_si256((const __m256i*)(a+i));
        b1=_mm256_load_si256((const __m256i*)(b+i));
        c=_mm256_add_epi32(a1,b1);
```

```
        //这里不包含_mm256_store_ps，即只进行向量加法而不保留结果
    }
}

//串行计算向量相加（包含数据存储）
void Serial(){
    int i=0;
    for(i=0;i<=n-1;i++){
        sum2[i]=a[i]+b[i];//将结果存储到 sum2 数组中
    }
}

//串行计算向量相加（不包含数据存储）
void Serial2(){
    int i;
    for(i=0;i<=n-1;i++){
        a[i]+b[i];//仅进行向量加法，不存储结果
    }
}

int main(){
    int i;
    srand((int)time(0));
    for(i=0;i<=n-1;i++){
        a[i]=rand()%100;
        b[i]=rand()%100;
    }
    //生成随机数组 a 和 b
    int times=10000;//为了放大时间，将 AVX 向量相加和串行向量相加均重复 10000
次

    double start1,end1,start2,end2,start3,end3,start4,end4;
    GET_TIME(start1);
    while(times--){
        AVX();
    }
    GET_TIME(end1);
    printf("AVX 总时间: %f\n",end1-start1);

    times=10000;
    GET_TIME(start2);
    while(times--){
        AVX2();
    }
```

```
    GET_TIME(end2);
    printf("AVX 数据存储所需时间: %f\n",(end1-start1)-(end2-start2));

    times=10000;
    GET_TIME(start3);
    while(times--){
        Serial();
    }
    GET_TIME(end3);
    printf("串行总时间: %f\n", end3-start3);

    times=10000;
    GET_TIME(start4);
    while(times--){
        Serial2();
    }
    GET_TIME(end4);
    printf("串行数据存储所需时间: %f\n",(end3-start3)-(end4-start4));

    printf("实际加速比: %f\n",(end3-start3)/(end1-start1));
    double t1,t2,w;
    t1=((end1-start1)-(end2-start2))/(end1-start1);
    t2=((end3-start3)-(end4-start4))/(end3-start3);
    w=(t1+t2)/2;
    printf("AVX 数据存储时间占比为: %f\n",t1);
    printf("串行数据存储时间占比为: %f\n",t2);
    printf("数据存储时间平均占比为: %f\n",w);
    printf("由 Amdahl 定律可知，理想情况下的最优加速比为: %f\n",1/(w+(1-
w)/8));

    for(i=0;i<n-1;i++){
        if(sum1[i]!=sum2[i]){
            printf("计算错误!\n");
        }
    }
    //检验 AVX 相加结果是否和串行一致（该部分代码可用于检测 AVX 指令集应用是否
正确）
    return 0;
}
```

添加 AVX2 和 Serial2 函数的原因是：一开始在对比 AVX 和 Serial 加速比时，我发现其只有 2.83 的加速比。但是 AVX 的读取数据方式是每次循环中读取向量中的 8 个元素，串行的读取数据方式是每次循环中读取向量中的 1 个元素。所以理想情况下的最优加速比应该是 8，而 2.83 显然远低于预期。

经过思考后，我联想到了 Amdahl 公式。假设我的代码中有无法被并行优化的程序段，那么就有可能导致实际加速比远低于 8。在检查代码后，我断定无法被优化的程序段就是 AVX 和串行中的数据存储部分（即把向量相加的结果存储到新的向量中），所以我添加了删去数据存储部分的 AVX2 和 Serial2 函数，并对代码输出进行了完善，从而得到了上述的最终版代码，而具体结果可见“三、实验结果”部分。

三、 实验结果

编译参数为：

```
PS C:\Users\93508\Desktop\.vscode\.vscode> gcc -O0 -mavx2 test2.c -o test2
PS C:\Users\93508\Desktop\.vscode\.vscode> ./test2
```

3 次运行结果依次为：

```
AVX总时间: 7.680975
AVX数据存储所需时间: 2.472202
串行总时间: 22.208131
串行数据存储所需时间: 6.070638
实际加速比: 2.891317
AVX数据存储时间占比为: 0.321860
串行数据存储时间占比为: 0.273352
数据存储时间平均占比为: 0.297606
由Amdahl定律可知，理想情况下的最优加速比为: 2.594670
```

```
AVX总时间: 7.709273
AVX数据存储所需时间: 2.392249
串行总时间: 22.201965
串行数据存储所需时间: 6.024900
实际加速比: 2.879904
AVX数据存储时间占比为: 0.310308
串行数据存储时间占比为: 0.271368
数据存储时间平均占比为: 0.290838
由Amdahl定律可知，理想情况下的最优加速比为: 2.635163
```

```
AVX总时间: 7.716212
AVX数据存储所需时间: 2.507909
串行总时间: 22.196610
串行数据存储所需时间: 5.983066
实际加速比: 2.876620
AVX数据存储时间占比为: 0.325018
串行数据存储时间占比为: 0.269549
数据存储时间平均占比为: 0.297283
由Amdahl定律可知，理想情况下的最优加速比为: 2.596573
```

所以计算平均值后可得如下表格

AVX 总时间	串行总时间	数据存储时间平均占比	实际加速比	理想加速比
7.702153	22.202235	0.295242	2.882614	2.608802

对比实际加速比和理想加速比，结果基本正确。

四、 遇到的问题及解决方法

（1）已解决的问题

①一开始我不知道编译参数，直接 g++运行，发现失败。在查阅网站后明白了编译参数应该如下所示

```
PS C:\Users\93508\Desktop\.vscode\.vscode> gcc -O0 -mavx2 test2.c -o test2
PS C:\Users\93508\Desktop\.vscode\.vscode> ./test2
```

②加速比远低于预期，在检查代码无原则性错误后，利用 Amdahl 定律成功解决。

③一开始我使用的编译参数是 O2，但是在用我的上述代码进行 Amdahl 验证时，发现 AVX2 和 Serial2 的运行时间均为 0，这显然不符合常理。在咨询室友后，我得知了 O0 和 O2 的区别：O2 会对代码进行一定程度上的编译运行优化，所以无法证明我的假设。而 O0 不做任何优化，是默认的编译选项。在我修改为 O0 后，得到了预期结果，验证了我的猜想。

（2）未解决的问题

①三次实验中都有实际加速比 > 理想最优加速比，虽然二者接近，但按常理来说应该是实际加速比 ≤ 理想最优加速比。我无法解释这一现象，但因为二者接近，所以我认为我的猜想仍然正确。希望自己在日后的学习中可以早日解决这一问题，同时如果老师或助教方便的话，麻烦在我的作业批改反馈中回答这一问题，非常感谢！

五、 问题 2 的文章

任务并行：将许多可以解决问题的任务分割，然后分布在一个或者多个核上进行程序的执行。

数据并行：将可以解决问题的数据进行分割，将分割好的数据放在一个或者多个核上进行执行；每一个核对这些数据都进行类似的操作。

提供的研究问题为：通过公式求解 π 值。

解决方法：可使用 pthread 中的 semaphore 计算 π 的值。迭代公式为：

$$\pi = 4\sum_{k}^{n}\frac{(-1)^k}{2k+1}$$

我们可将迭代公式的 n 项按照线程数平均分组，之后进行**数据并行**，即可得到 π 值。

代码及其分析如下所示：

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<pthread.h>
#include<semaphore.h>
#include<sys/time.h>

const int MAX_THREADS = 1024;//最大线程数

long thread_count;//线程数量
long long n;//迭代项数
double sum;//公式最终的总和（多线程计算 pi 的估计量）

sem_t sem;//信号量

void* Thread_sum(void* rank);

int main(int argc, char* argv[]) {
   long       thread;  /* 在 64 位系统中使用 long */
   pthread_t* thread_handles;
   //pthread_t 用于声明线程 ID
   double start, finish, elapsed;

   /* 可选择在这里确定公式项数和线程数量 */
   n = 10000;
   thread_count = 4;

   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t)
);//创建线程数组
   sem_init(&sem, 0, 1);//初始化信号量
   sum = 0.0;//最终的总和初始化为 0

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,Thread_sum, (void*)t
hread);//创建线程
      //第一个参数为指向线程标识符的指针。
      //第二个参数用来设置线程属性。
      //第三个参数是线程运行函数的起始地址。
      //最后一个参数是运行函数的参数。

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);
      //函数 pthread_join 用来等待一个线程的结束。
```

```
      //第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。
      //这个函数是一个线程阻塞的函数，调用它的线程将一直等待到被等待的线程结束为止
      //当函数返回时，被等待线程的资源被收回。
      //也就是说主线程中要是加了这段代码，就会在该代码所处的位置卡住，直到这个线程执行完毕才会继续往下运行。

   sum = 4.0*sum;//1/4*pi*4=pi（公式计算的是 1/4*pi）
   printf("With n = %lld terms,\n", n);//公式项数
   printf("   Our estimate of pi = %.15f\n", sum);//多线程估计的 pi 值
   printf("                      pi = %.15f\n", 4.0*atan(1.0));//arctan 计算 pi 值

   sem_destroy(&sem);//释放信号量
   free(thread_handles);//释放线程数组
   return 0;
}  /* main */

/*--------------------------------------------------------------------*/
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   //如果指针类型的大小和表示进程编号的整数类型不同，在编译时就会受到警告
   //在我们使用的机器上，指针类型 64 位，int 类型 32 位
   //为了避免警告，所以我们用 long 型替换 int 型
   double my_sum = 0.0;//每个线程内部的和（最终要汇总到全局变量 sum）
   /*********************************************************************/
   double flag;//用于确定公式中每项的正负
   long long i;
   long long group_num=n/thread_count;//计算每个线程中需要分配的公式项数（尽可能平均）
   long long group_first_i=group_num*my_rank;//计算每个线程中开始累加的第一项
   long long group_last_i=group_first_i+group_num;//计算每个线程中需要累加的最后一项
   if(group_first_i%2==0){
      flag=1.0;
   }
   else{
      flag=-1.0;
   }
   for(i=group_first_i;i<=group_last_i-1;i++){//每个线程内部开始累加
      my_sum+=flag/(2*i+1);//1/4pi 的计算公式
```

```
        flag*=-1;//改变下一项的正负
    }
    sem_wait(&sem);//阻塞线程，等待获取信号量
    sum+=my_sum;//获取信号量后可将当前线程的和累加进入全局变量 sum
    sem_post(&sem);//释放信号量
    /*****************************************************************/
    return NULL;
}  /* Thread_sum */
```

在超算习堂上运行该程序，迭代 10000 项后的 π 值结果与精确 π 值对比如下所示：

```
======== OUTPUT ========
With n = 10000 terms,
Our estimate of pi = 3.141492653590044
pi = 3.141592653589793
```