并行与分布式作业

“作业名字”

第 1 次作业

姓名：TRY

班级：18 级计科 7 班

学号：

一、 问题描述

计算两个各包含10^6个分量的向量之和，并用两种方法实现：线程级并行和指令级并行。其中，“线程级并行”发挥了多核的优势，并可用 C/C++，Java 等语言实现；“指令级并行”可选用 Intel, ARM, AMD, Power 等的并行指令集合实现。

二、 解决方案

1. 线程级并行

本次，我发挥了多核的优势，实现了在不同核参与计算 2 个向量时，加速比的比较。其中，程序的编写参考了老师上课课件中的例子进行修改。

线程级加法实现如下：

```
struct ThreadParam {
    size_t begin;
    size_t end;
    int *result;//结果数组
    int *a;//向量1
    int *b;//向量2
    ThreadParam(size_t begin1, size_t end1, int *result1, int *a1, int *b1) :begin(begin1), end(end1),
};

DWORD WINAPI addvalues2(LPVOID param) {
    ThreadParam*p = static_cast<ThreadParam*>(param);//传入的param是void*，所以要进行转换
    for (size_t i = p->begin; i < p->end; ++i)
    {
        p->result[i] = p->a[i] + p->b[i];
        if (i % 100 == 0)
            Sleep(10);
    }
    delete p;
    return 0;
}
```

普通串行加法实现如下：

```
void addvalues3(int *a, int *b, int *&result)
{
    for (int i = 0; i < N; i++)
    {
        result[i] = a[i] + b[i];
        if (i % 100 == 0)//当计算了100个后，沉睡0.01s
            Sleep(10);
    }
}
```

main 函数如下：

```
for (int i = 0; i < N; i++)//利用随机数初始化数据
{
    a[i] = rand() % 100 + 1;
    b[i] = rand() % 100 + 1;
}

clock_t start1 = clock();//程序段开始前取得系统运行时间(s)
```

```
for (int i = 0; i < num_of_CPUs; ++i)
{
    ThreadParam* p = new ThreadParam(i*N / num_of_CPUs, (i + 1)*N / num_of_CPUs, result1, a, b);
    //在不能整除时，把后面的都处理掉
    if (i == num_of_CPUs - 1)
        p->end = N - 1;
    Threads[i] = CreateThread(NULL, 0, addvalues2, p, 0, NULL);
}
WaitForMultipleObjects(num_of_CPUs, Threads, true, INFINITE);
clock_t end1 = clock();//程序段结束后取得系统运行时间(s)
```

其中，多线程并行设计的重点在于利用 sleep()函数设计了沉睡功能。理论上，多线程并行的时间应该会比串行的时间要短。但在一开始的实现中，经过多次尝试，笔者发现多线程的时间竟然比单线程串行的时间要长！经过查阅才资料发现，多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用。为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次“上下文切换”。而这些“上下文切换”和“多线程的创建”都是有较大的时间开销的，而我们本次任务中所要实现的加法又比较简单。所以在简单的加法面前，这些上下文切换所要用的时间开销就变得很大，所以会出现多线程并行比单线程的时间更多的情况，即没体现处多线程的加速效果。

在这样的基础上，笔者设计使用了 sleep()函数来抵消多线程调用时额外的时间开销。即在单线程和多线程的计算过程中， 每当计算 100 个 int 相加时，就调用一次 sleep 函数，沉睡 0.01s。并且在这种情况下，单线程和多线程沉睡的总时间是相同的，多线程加速效果就可以体现出来。而能引入 Sleep()函数的原理是：Sleep()可以将整个计算过程的时间延长，而在整个过程中，多线程的创建和切换的开销的影响就变得“微不足道”，所以时间反映的就是多线程的加速效果。（在几十秒面前，0.00 几秒太小了！）而实现的前提是要保证两种情况下 sleep 的时间是相同的，即将 sleep()变成无关变量。

2. 指令级并行

本次，笔者利用了 intel 的 AVX 指令集实现了指令级并行。

指令级并行代码如下：

```
void addvalues1(int *a, int *b, int *&result)//已知size为1000 000
{
    int nBlockWidth = 8;//每次计算8个int分量的相加
    int cntBlock = N / 8;//block数量（循环计数上限）
    __m256i loadData1, loadData2;
    __m256i resultData = _mm256_setzero_si256();
    __m256i *p1 = (__m256i *)a;//进行指针类型的转换
    __m256i *p2 = (__m256i *)b;
    __m256i *p3 = (__m256i *)result;
    for (int i = 0; i < cntBlock; i++)
    {
        loadData1 = _mm256_load_si256(p1);//加载数据1
        loadData2 = _mm256_load_si256(p2);//加载数据2
        resultData = _mm256_add_epi32(loadData1, loadData2);//将2数据相加
        _mm256_store_si256(p3, resultData);//卸载数据回结果数组result
        p1 += 1;//！！！是加1而不是加8！（p1+1=p1+1*8 int,"1"means one __m256i）
        p2 += 1;
        p3 += 1;
    }
    return;
}
```

串行执行的代码如下：

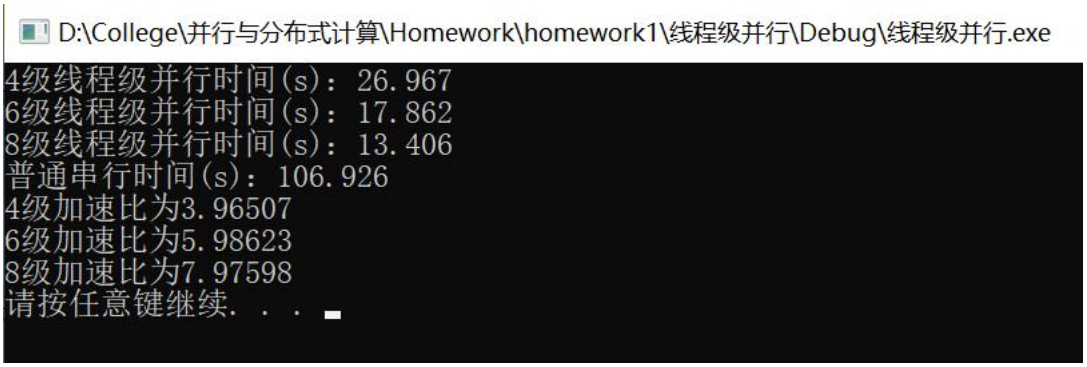
```
void addvalues3(int *a, int *b, int *&result)
{
    for (int i = 0; i < N; i++)
    {
        result[i] = a[i] + b[i];
    }
}
```

本部分的难点在于 avx 语法的掌握。在查阅了 avx 编写语法后，便可较简单的写出来。其中，要尤其注意指针间的转换。

而在此处，为了更准确的反应 avx 的加速比，我们也引入 Sleep()函数进行处理。且由于 avx 的__m256i 每次计算 8 个 int 分量的相加，而 8 无法整除 100，但 8 可整除 1000，所以使用每计算 1000 个 int 分量时，sleep(100)，即沉睡 0.1s，然后在 avx 程序中，每走过 125 个循环，即计算了 1000 个 int 分量是，sleep(100)，则可保证两者的沉睡时间相同。

三、 实验结果

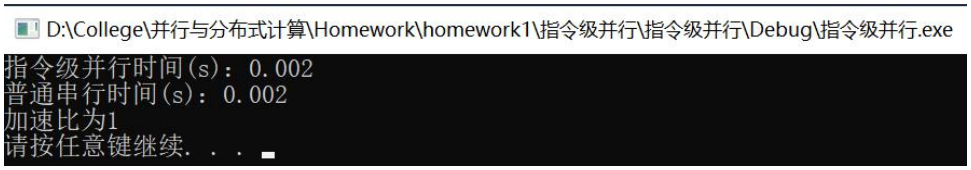
1.线程级并行

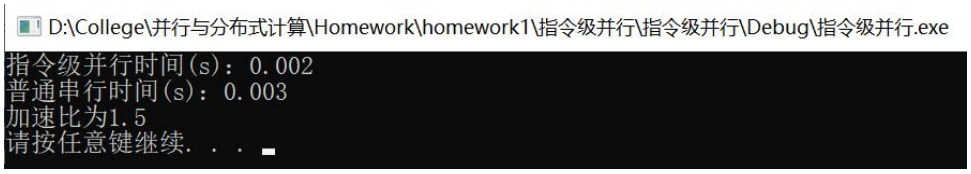


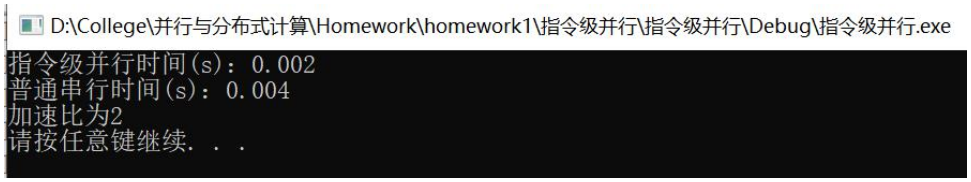
由上图可以发现，在有 sleep()函数的影响下，n 级线程并行的结果加速比正好近似为 n。如 4 级线程并行的加速比近似为 4。且 n 越大，加速比越大。

2.指令级并行

（1）不加 sleep()的情况：







（2）添加 sleep()的情况：

如上图所示，指令级并行的运行的加速比通常为 1~2 之间。而在添加 sleep()后，则会在 1.39 左右。

根据上面的测试结果，我们可以发现，在不加 sleep()函数抵消调用影响的情况下，大部分情况下 avx 程序可以加速，且加速比也不相同；而在添加 sleep()后，可知加速比在 1.39 左右。且经过大量的测试，在不考虑函数调用时间的情况下， AVX 算法对于向量加速比大概可以到达 1.4 左右。根据 Amdahl’s Law:

$$Speedup = \frac{1}{(1-p)+\frac{p}{n}}$$

其中 p 表示程序中可以并行的部分，n 表示并行化力度。在笔者的程序中，n=8，所以根据计算我们可以得到 p≈0.32。即可发现，当 p≈0.32,n=8 的时候，加速比仍只有 1.4 左右。这说明简单的指令级并行化设计并不能完全满足人们对计算机加速性能的要求。

四、 遇到的问题及解决方法

本次是第一次完成并行作业，途中也遇到了不少困难，但总体来说是一个不错的体验。

其中，第一个问题是如何在__m256i 和数组之间进行转移。比如说，如何装载数据、卸

载数据。在查阅了不少资料后，了解到 avx 的函数库，并解决。

第二个问题，是在完成打码后出现的，即上文中笔者所提到的“多线程并行时间大于单线程串行时间”的问题。具体的思考和解决过程已在上文中提到，在此不多赘述。但笔者在完成了该问题的设计后，开始思考：AVX 指令级并行是不是也要用类似的思路解决呢？即用 sleep()函数来抵消调用 AVX 的影响以突出加速比呢？

但是在思考和多次尝试后，我发现加不加都可以的。因为不加的时候已经可以体现 avx 加速的情况了，只不过加了 sleep()的时候，加速比更精确一些。但其实，笔者认为，仍不应该加 sleep()。原因如下：因为沉睡时间是 10^6/1000*0.1s=100s,即在每计算 1000 个 int 相加后进行沉睡 0.1s，总程序的沉睡时间是 100s，而本身 avx 相加的时间只需要 0.001~0.002s，与 100s 相比影响实在太小了。所以最后经实验验证也发现，在两者的沉睡时间相同时，两种方法的总时间差不多，加速效果并不明显。而在笔者看来，造成这种情况的原因是：多线程并行其实是将总任务分给了不同的线程（即不同的核上）去完成，在有 sleep()的影响下，这种“分配”后的加速是可以体现出来的。而 AVX 指令级并行中，其实是通过一次执行 8 个数的加法来实现加速的，而这种加法的时间本身就很短，若再引入 sleep()来进行抵消调用的影响，则会使得总时间主要由沉睡时间组成，AVX 加速的时间占比很小。所以，在 AVX 并行中，不一定需要使用 sleep()函数。

第三个问题，是我在看了群上同学的讨论思考的：到底应该用哪个时间函数？而在与其他同学讨论后，我认为 gettimeofday()函数的确是要精确一些，但是在有了 sleep()函数抵消调用的影响后，clock()函数也已经可以反映加速比了。

第四个问题，就是为什么 avx 的加速比只有 1.39 呢？经过研究，笔者没有准确的答案，但猜测应该是 avx 有调用的影响，且也与电脑的性能有关。也许，有些同学的电脑性能较好，就可以显示在 0s 跑完；而有些同学的电脑性能一般，则要更长的时间才能跑出来。