并行与分布式作业

Homework-1

第 1 次作业

姓名：关雅雯

班级：教务一班

学号：18340045

一、 问题描述

1. 请调查处理器中的并行指令集，并选择其中一种进行编程练习，计算两个各包含 10^6 个整数的向量之和。

2. 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，请将问题 1 用多线程的方式实现，线程实现的语言不限，可以是 Java，也可以是 C/C++。

二、 解决方案

首先，对于普通的向量加法，直接循环相加即可，核心代码如下：

```
    for(int i = 0; i < VectorSize; i ++)
        ans_vec[i] = v1[i] + v2[i];
```

其次，使用 AVX 指令集，需要先把每 8 个 32 位整数类型的数据存储到 1 个 __m256i 类型中：

```
int_vec1[i] = _mm256_load_si256((const __m256i*) aligned_int1[i]);
int_vec2[i] = _mm256_load_si256((const __m256i*) aligned_int2[i]);
```

核心代码如下：

```
for(int i = 0; i < VectorSize / 8; i ++)
        ans_int[i] = _mm256_add_epi32(int_vec1[i], int_vec2[i]);
```

最后，使用多线程，此处使用 4 个线程。核心代码如下：

```
void addUnit(int st, int ed){
    for(int i = st; i < ed; i ++)
        ans[i] = vec1[i] + vec2[i];
}
//调用多线程
const int len = VectorSize / 4;
thread *t[4];
```

```
auto start = chrono::system_clock::now();
for(int i = 0; i < 4; i ++)
    t[i] = new thread{addUnit, i * len, (i + 1) * len};
for(int i = 0; i < 4; i ++)
    t[i] -> join();
```

以上三种方法中，计时均采用 std::chrono，只对核心代码部分计时。

三、 实验结果

项目结构：

```
.
├── CMakeLists.txt
├── include
│   ├── common.hpp
│   └── functions.hpp
├── lib
│   ├── addNormal.cpp //不用 AVX 和多线程下的向量加法
│   ├── addUsingAVX.cpp //使用 AVX 的向量加法
│   ├── addUsingThreads.cpp //使用多线程的向量加法
│   ├── CMakeLists.txt
│   └── dataMaker.cpp //生成两个 10^6 的向量
├── README.md
└── src
    └── main.cpp
```

编译运行：

```
mkdir build && cd build
cmake ..
make
./add
```

运行后，build 子文件夹结构如下：

```
./build
├── add //项目可执行文件
├── addNormal_output //addNormal 的结果输出
├── addUsingAVX_output //addUsingAVX 的结果输出
├── addUsingThreads_output //addUsingThreads 的结果输出
├── CMakeCache.txt
├── CMakeFiles
├── cmake_install.cmake
├── lib
├── Makefile
├── vector1 //makeData 生成的 vector1
└── vector2 //makeData 生成的 vector2
```

输出结果：

只计算一次向量加法：

```
gwen@DESKTOP-HVHN85U:/mnt/d/A_GW/Courses/Parallel_and_distributed_computing/hw1/build$ ./add
Make two 1*1000000 vectors which store in vector1 and vector2

Now try to add them normally, the output stored in addNormal_output
elapsed time: 0.0021842s

Now try to add them using AVX, the output stored in addUsingAVX_output
elapsed time: 0.0012049s

Now try to add them using threads, the output stored in addUsingThreads_output
elapsed time: 0.0014012s
```

运行 40000 次向量加法：

```
gwen@DESKTOP-HVHN85U:/mnt/d/A_GW/Courses/Parallel_and_distributed_computing/hw1/build$ ./add
Make two 1*1000000 vectors which store in vector1 and vector2

run 40000 times

Now try to add them normally, the output stored in addNormal_output
elapsed time: 65.8339s

Now try to add them using AVX, the output stored in addUsingAVX_output
elapsed time: 9.62923s

Now try to add them using threads, the output stored in addUsingThreads_output
elapsed time: 17.7343s
```

分析：

由 Amdahl’s Law 可知，$$Speedup = \frac{1}{(1-p)+\frac{p}{k}}$$

计时包括了多次访存读写数据的开销，在多线程中还有创建线程、调度线程的开销，均会导致加速比相比预期减小。

为使计时结果加速比更接近预期的加速比，对上述循环执行多次，由于多次访问同样的数据，由内存管理的知识，推测访问该数据的速度会更快；同时相当于多次实验取平均加速比，排除偶然因素。

由运行 40000 次的数据算出：

$$speedup_{AVX} = \frac{65.8339}{9.62923} = 6.84$$

$$speedup_{threads} = \frac{65.8339}{17.7343} = 3.71$$

均比较接近理想的加速比（分别为 8 和 4）

四、遇到的问题及解决方法

原本不会如何使用 AVX，查看了老师提供的资料以及 Intel 的文档后知道了如何使用；只计算一次向量加法时，加速比与预期相差较大，分析原因为计算的时间里不仅包含加法的时间，还包含了访存、创建线程、线程调度等时间开销，根据 Amdahl’s Law 选择了多次循环的方法让加法占总开销比例更高，从而使加速比更接近理想加速比。