

# 并行与分布式作业

AVX 指令集编程与多线程编程

第一次作业

姓名： 何泽

班级： 18 级计科（超算方向）

学号： 18340052

# 一、问题描述

- 我们在第一次课程中已经降到，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

## 1. AVX 指令集基本原理

### ①数据类型

数据类型	描述
<code>__m128</code>	包含 4 个 <code>float</code> 类型数字的向量
<code>__m128d</code>	包含 2 个 <code>double</code> 类型数字的向量
<code>__m128i</code>	包含若干个整型数字的向量
<code>__m256</code>	包含 8 个 <code>float</code> 类型数字的向量
<code>__m256d</code>	包含 4 个 <code>double</code> 类型数字的向量
<code>__m256i</code>	包含若干个整型数字的向量

- 每一种类型，从 2 个下划线开头，接一个 `m`，然后是 `vector` 的位长度。
- 如果向量类型是以 `d` 结束的，那么向量里面是 `double` 类型的数字。如果没有后缀，就代表向量只包含 `float` 类型的数字。
- 整形的向量可以包含各种类型的整形数，例如 `char, short, unsigned long long`。也就是说，`__m256i` 可以包含 32 个 `char`，16 个 `short` 类型，8 个 `int` 类型，4 个 `long` 类型。这些整形数可以是有符号类型也可以是无符号类型。

### ②函数命名

`_mm<bit_width>_<name>_<data_type>`

- `<bit_width>` 表明了向量的位长度，对于 128 位的向量，这个参数为空，对于 256 位的向量，这个参数为 256。
- `<name>` 描述了内联函数的算术操作。
- `<data_type>` 标识函数主参数的数据类型。

- `ps` 包含 `float` 类型的向量
- `pd` 包含 `double` 类型的向量
- `epi8/epi16/epi32/epi64` 包含8位/16位/32位/64位的有符号整数
- `epu8/epu16/epu32/epu64` 包含8位/16位/32位/64位的无符号整数
- `si128/si256` 未指定的128位或者256位向量
- `m128/m128i/m128d/m256/m256i/m256d` 当输入向量类型与返回向量的类型不同时，标识输入向量类型

## 2. 串行编程

- 先声明三个数组a、b、c，a与b存放待加数据，c存放求和结果，再用 `rand` 函数随机填充a与b，同时为了增加计算复杂度，随机生成的数值均大于100 0000

```
1  srand( (int)time(0) );
2
3  for(int i=0;i<999999;i++){
4      a[i]=rand()+1000000;
5      b[i]=rand()+1000000;
6  }
```

- 开始计时，串行相加，运算完毕后结束计时，相减便可得到计算时间

```
1  start=clock();
2  for(int i=0;i<999999;i++){
3      c[i]=a[i]+b[i];
4  }
5  end=clock();
6
7  t1=(double)(end-start);
```

## 3. 使用AVX指令集并行编程

- 为了时间对比的正确性，依然使用串行编程的待相加数据，不改变
- 声明三个变量

```
1  __m128i m,n,p;
```

其中m，n用于存放加数，p存放和

- 在循环中，先使用 `set` 函数将数组数据存入m与n，再使用 `add` 函数求和

```
1  for(int i=0;i<999997;i+=4){
2      m=_mm_set_epi32(a[i],a[i+1],a[i+2],a[i+3]);
3      n=_mm_set_epi32(b[i],b[i+1],b[i+2],b[i+3]);
4
5      p=_mm_add_epi32(m,n);
6  }
```

- 下一步需要计时，我一开始的想法很简单，从循环前开始，循环结束后计时结束，但后来发现这是不准确的，因为 `set` 函数也需要时间，而我们需要知道的只是求和的时间，所以把计时点放在 `add` 前后，每次循环后累加

```

1   for(int i=0;i<999997;i+=4){
2
3       m=_mm_set_epi32(a[i],a[i+1],a[i+2],a[i+3]);
4       n=_mm_set_epi32(b[i],b[i+1],b[i+2],b[i+3]);
5
6       start=clock();
7       p=_mm_add_epi32(m,n);
8       end=clock();
9       t2+=(double)(end-start);
10  }

```

## 4. 多线程编程

- 首先，输出我这台电脑的线程数

```

1   auto core=thread::hardware_concurrency();
2   cout<<"The number of the CPU threads is  "<<core<<endl;

```

运行后可知本机有4个线程

- 于是先编写一个每个线程都调用的函数 `add`，因为一共有 `100 0000` 个数相加，那么每个线程加 `250000` 个数，则这个函数输入一个整形参数，将这个数之后的 `250000` 个向量相加

```

1   void add(int p)
2   {
3       for(int i=p;i<p+250000;i++){
4           c[i]=a[i]+b[i];
5       }
6   }

```

- 然后在主函数中开启四个线程，分别传入参数 `0`，`250000`，`500000`，`750000`，并计时

```

1   start=clock();
2
3   thread th1(add,0);
4   thread th2(add,250000);
5   thread th3(add,500000);
6   thread th4(add,750000);
7
8   th1.join();
9   th2.join();
10  th3.join();
11  th4.join();
12
13  end=clock();
14  t3=(double)(end-start);

```

## 三、实验结果

### 1. 编译环境

## 2. 实验结果

此处的实验结果并非最终结果，最终实验结果详见“四、遇到的问题及解决办法”

- 由于每次随机生成的数不同，导致每次运算结果都不一样
- 并行、多线程运算完毕后都进行一步除法，输出加速比

```
----- Serial process is starting. -----
The time of serial process is 17.000000

----- Parallel process is starting. -----
The time of parallel process is 2.000000

The speedup of parallel process is 8.500000

----- Multithreading is starting. -----
• The number of the CPU threads is 4
  The time of multithreading is 2.000000

The speedup of multithreading is 8.500000

-----
Process exited after 0.3253 seconds with return value 0
请按任意键继续. . .
```

```
----- Serial process is starting. -----
The time of serial process is 15.000000

----- Parallel process is starting. -----
The time of parallel process is 2.000000

The speedup of parallel process is 7.500000

----- Multithreading is starting. -----
• The number of the CPU threads is 4
  The time of multithreading is 3.000000

The speedup of multithreading is 5.000000

-----
Process exited after 0.1868 seconds with return value 0
请按任意键继续. . .
```

```
----- Serial process is starting. -----
The time of serial process is 16.000000

----- Parallel process is starting. -----
The time of parallel process is 1.000000

The speedup of parallel process is 16.000000

• ----- Multithreading is starting. -----
The number of the CPU threads is 4
The time of multithreading is 2.000000

The speedup of multithreading is 8.000000

-----
Process exited after 0.2875 seconds with return value 0
请按任意键继续. . .
```

```
----- Serial process is starting. -----
The time of serial process is 16.000000

----- Parallel process is starting. -----
The time of parallel process is 5.000000

The speedup of parallel process is 3.200000

• ----- Multithreading is starting. -----
The number of the CPU threads is 4
The time of multithreading is 1.000000

The speedup of multithreading is 16.000000

-----
Process exited after 0.253 seconds with return value 0
请按任意键继续. . .
```

以上结果仅为初步结果

## 四、遇到的问题及解决办法

### 1. 问题

由以上的初步结果可以看出已完成并行编程与多线程编程，但在进一步分析加速比的时候却有了问题，即因为每一次运行时随机数结果都不一样，导致每次的加速比都不同且差别较大。

## 2. 解决办法

---

将串行、并行、多线程都进行1000次，运行后取时间的平均值，再计算加速比。

## 3. 最终实验结果

---

```
----- Serial process is starting. -----
The average time of serial process is 0.002507

----- Parallel process is starting. -----
The average time of parallel process is 0.001052

The speedup of parallel process is 2.383080

----- Multithreading is starting. -----
The number of the CPU threads is 4
The time of multithreading is 0.001723

The speedup of multithreading is 1.455020

-----
Process exited after 78.04 seconds with return value 0
请按任意键继续. . .
```

```
----- Serial process is starting. -----
The average time of serial process is 0.002460

----- Parallel process is starting. -----
The average time of parallel process is 0.001068

The speedup of parallel process is 2.303371

----- Multithreading is starting. -----
The number of the CPU threads is 4
The time of multithreading is 0.001775

The speedup of multithreading is 1.385915

-----
Process exited after 77.68 seconds with return value 0
请按任意键继续. . .
```

运行1000次之后可以看到结果已趋于稳定，使用AVX指令集的加速比在 2.4 左右，多线程的加速比在 1.4 左右

## 4. 结果分析

---

在运行过程中，数据移动大概占了30% 。

根据 Amdahl's Law ,

$$Speedup = \frac{1}{1 - \alpha + \frac{\alpha}{k}} = \frac{1}{(1 - 70\%) + \frac{70\%}{4}} = 2.105$$

这与实验结果是相近的。

以上，便完成了并行编程与多线程编程。