

# 并行与分布式作业

## 第一次作业

姓名：张洪宾  
班级：18 级计科超算方向  
学号：18340208

2020 年 3 月 5 日

# 1 问题描述

我们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，

## 1.1

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。

## 1.2

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 1.1 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++。

# 2 实验环境

## 2.1 软件

- macOS Catalina
- gcc 9.2.0
- VScode

## 2.2 硬件

所用机器型号是 MacBook Pro2015

- 2.5 GHz 四核 Intel Core i7
- 16 GB 1600 MHz DDR3

# 3 问题 1 的解决

1966 年，Flynn 从指令和数据这两个维度，可以对处理器的系统结构分类：

SISD(single instruction single data): 一次处理一条指令，一条指令处理一份数据，早期的处理器都是这种形式。

SIMD(single instruction multiple data):-次处理一条指令，一条指令能处理多份数据，这种方式

称为数据并行, 现在性能稍微强一点的处理器都具备这种功能。

MISD(multiple instruction single data): 一次处理多条指令, 多条指令处理一份数据, 这种结构没有实际意义。

MIMD(multiple instruction multiple data): 一次处理多条指令, 多条指令能处理多条数据, 这种方式称为指令并行, 高性能处理器都具备这个功能。

对于此次实验需要要做的向量加法, 由于不同的操作数之间并没有直接的相互关系, 而且对它们操作的指令都是加法, 所以我选择了 SIMD 指令。而许多处理器都有 SIMD 指令, 如 Intel 的 MMX 指令集、SSE 指令集、AVX 指令集, AMD 的 3DNow! 指令集 (不过现在的 AMD 也支持 Intel 的 AVX 指令集)。由于我的 CPU 是 Intel 的, 所以我选择使用 AVX 指令集。

而为了显示并行指令对运行对优化, 我先写了一个基准程序 *serial\_add.c*, 用串行的方法来对向量进行计算。

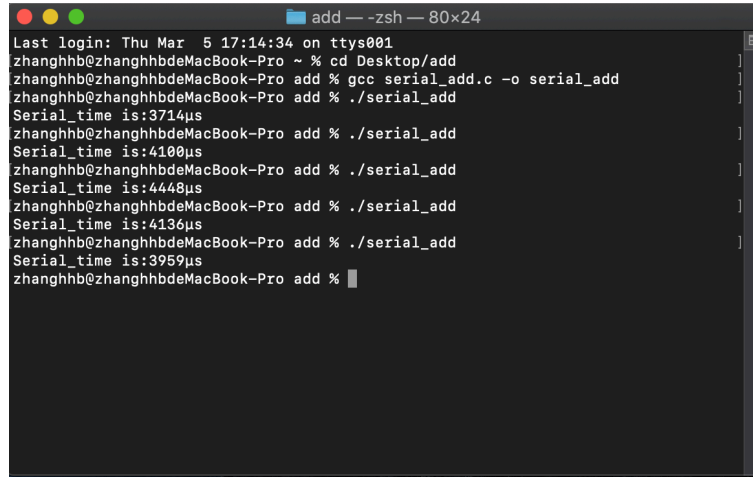
### 3.1 基准程序 serial\_add.c

用串行的方法计算包含  $10^6$  个整数的向量之和:

```
//serial_add.c
#include <stdio.h>
#include <sys/time.h>
#define N 1000000

int a[N], b[N], c[N];
int main(int argc, char const *argv[]) {
    for(int i = 0; i < N; i++){
        a[i] = 0;
        b[i] = 0;
    }
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    for(int i = 0; i < N; i++){
        c[i] = a[i] + b[i];
    }
    gettimeofday(&end, NULL);
    printf("Serial_time:%ld s\n", end.tv_sec*1000000 + end.tv_usec
        - begin.tv_sec*1000000 - begin.tv_usec);
    return 0;
}
```

在终端中输入 `gcc serial_add.c -o serial_add` 命令进行编译，然后将程序运行 5 次，运行结果如下：



```
add — zsh — 80x24
Last login: Thu Mar  5 17:14:34 on ttys001
zhanghbb@zhanghbbdeMacBook-Pro ~ % cd Desktop/add
zhanghbb@zhanghbbdeMacBook-Pro add % gcc serial_add.c -o serial_add
zhanghbb@zhanghbbdeMacBook-Pro add % ./serial_add
Serial_time is:3714us
zhanghbb@zhanghbbdeMacBook-Pro add % ./serial_add
Serial_time is:4100us
zhanghbb@zhanghbbdeMacBook-Pro add % ./serial_add
Serial_time is:4448us
zhanghbb@zhanghbbdeMacBook-Pro add % ./serial_add
Serial_time is:4136us
zhanghbb@zhanghbbdeMacBook-Pro add % ./serial_add
Serial_time is:3959us
zhanghbb@zhanghbbdeMacBook-Pro add %
```

图 1: 5 次运行 serial\_add 的结果

### 3.2 AVX 程序 avx\_add.c

用并行指令集 AVX 来计算包含  $10^6$  个整数的向量之和:

```
//avx_add.c
#include <immintrin.h>
#include <stdio.h>
#include <sys/time.h>
#define N 1000000
__m256i vec1[N/8 + 1];
__m256i vec2[N/8 + 1];
__m256i res[N/8 + 1];

int main(int argc, char const *argv[]) {
    for(int i = 0; i < N/8; i++){
        vec1[i] = _mm256_set1_epi32(0);
        vec2[i] = _mm256_set1_epi32(0);
    }
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    for(int i = 0; i < N/8; i++){
        res[i] = _mm256_add_epi32(vec1[i], vec2[i]);
    }
    gettimeofday(&end, NULL);
```

```

printf("Avx_time:%ld s\n",end.tv_sec*1000000 + end.tv_usec -
      begin.tv_sec*1000000 - begin.tv_usec);
return 0;
}

```

在终端中输入 `gcc avx_add.c -o avx_add -mavx -mavx2 -mfma -msse -msse2 -msse3 -Wall -O` 命令进行编译，然后将程序运行 5 次，运行结果如下：

```

Last login: Thu Mar  5 18:26:37 on ttys001
zhanghhb@zhanghhbdeMacBook-Pro ~ % cd Desktop/add
zhanghhb@zhanghhbdeMacBook-Pro add % gcc avx_add.c -o avx_add -mavx -mavx2 -mfma
-msse -msse2 -msse3 -Wall -O
zhanghhb@zhanghhbdeMacBook-Pro add % ./avx_add
Avx_time is:2348µs
zhanghhb@zhanghhbdeMacBook-Pro add % ./avx_add
Avx_time is:2261µs
zhanghhb@zhanghhbdeMacBook-Pro add % ./avx_add
Avx_time is:2246µs
zhanghhb@zhanghhbdeMacBook-Pro add % ./avx_add
Avx_time is:2294µs
zhanghhb@zhanghhbdeMacBook-Pro add % ./avx_add
Avx_time is:2008µs
zhanghhb@zhanghhbdeMacBook-Pro add %

```

图 2: 5 次运行 `avx_add` 的结果

统计两次运行的结果：

运行的次数	串行计算所需时间 $t_1(\mu s)$	AVX 指令集计算所需时间 $t_2(\mu s)$
1	3713	2348
2	4100	2261
3	4448	2246
4	4136	2294
5	3959	2008
平均	4071.2	2231.4

可以发现用 AVX 能加快向量求和，并算出加速比  $S = \frac{t_1}{t_2} = 1.824$ 。

## 4 问题 2 的解决

问题 1 是粒度为指令级的并行，问题 2 要求我们使用多线程的方式重做问题 1。在这里我选择了使用 OpenMP 编程，使用的是 C 语言编程。

为了对比线程数量的不同对计算速度的影响，我每次运行程序都会先输入程序的线程数量，然后再输出计算两个各包含  $10^6$  个整数的向量之和所需的时间，程序的源代码如下：

```
//parallel.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
#define N 1000000

int a[N], b[N], c[N];

int main(int argc, char* argv[]) {
    printf("Input the number of the threads:");
    int n;
    scanf("%d", &n);
    omp_set_num_threads(n);
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    #pragma omp parallel for
    for(int i = 0; i < N; i++){
        a[i] = b[i] + c[i];
    }
    gettimeofday(&end, NULL);
    printf("Run time with %d threads is %d s\n", n, end.tv_sec
        *1000000 + end.tv_usec - begin.tv_sec*1000000 - begin.
        tv_usec);
}
```

在终端输入 `gcc-9 parallel.c -o parallel -fopenmp` 来编译该程序，然后输入线程的数量来选择由几个线程来做运算。在这里我将线程数量的范围选择在 1(即单线程) 到 8。

```

multi-thread --zsh 80x25
zhanghhb@zhanghhbdeMacBook-Pro multi-thread % ./parallel
Input the number of the threads:1
Run time with 1 threads is 8565  $\mu$ s
zhanghhb@zhanghhbdeMacBook-Pro multi-thread % ./parallel
Input the number of the threads:2
Run time with 2 threads is 5592  $\mu$ s
zhanghhb@zhanghhbdeMacBook-Pro multi-thread % ./parallel
Input the number of the threads:3
Run time with 3 threads is 4459  $\mu$ s
zhanghhb@zhanghhbdeMacBook-Pro multi-thread % ./parallel
Input the number of the threads:4
Run time with 4 threads is 4524  $\mu$ s
zhanghhb@zhanghhbdeMacBook-Pro multi-thread % ./parallel
Input the number of the threads:5
Run time with 5 threads is 5501  $\mu$ s
zhanghhb@zhanghhbdeMacBook-Pro multi-thread % ./parallel
Input the number of the threads:6
Run time with 6 threads is 5755  $\mu$ s
zhanghhb@zhanghhbdeMacBook-Pro multi-thread % ./parallel
Input the number of the threads:7
Run time with 7 threads is 5926  $\mu$ s
zhanghhb@zhanghhbdeMacBook-Pro multi-thread % ./parallel
Input the number of the threads:8
Run time with 8 threads is 6860  $\mu$ s
zhanghhb@zhanghhbdeMacBook-Pro multi-thread %

```

图 3: 连续 8 次运行程序，线程数量从 1 到 8

为了防止偶然情况对结果对影响，我每中线程数量都运行了 5 次，取平均值后如下：

线程数量	1	2	3	4	5	6	7	8
平均运行时间 ( $\mu$ s)	8792.6	5592.4	4653.4	4849	5664.4	6147.4	6411.6	7148.4

作出散点图如下：

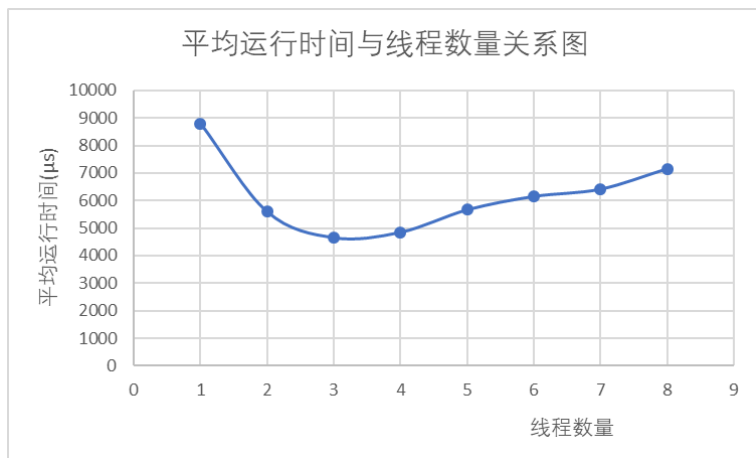


图 4: 平均运行时间与线程数量的关系

可以发现，在最开始随着线程数量的增加，计算所需的时间显著减小，而随着线程的逐渐增多，运行时间减少的幅度减小，而当线程增多到一定程度时，运行时间逐渐增加。

通过上网查询资料以及翻阅教材，发现之所以在线程较多时，增加线程数量反而使运行速度减慢，是因为诸如线程切换的增加等因素。所以一味增加线程数量不能很好地减少运行时间。

## 5 遇到的问题及解决方法

### 5.1

最开始使用 AVX 的时候用 gcc 直接编译，发现无法成功运行。

解决方法：通过研究老师的 Github 的 makefile，在 gcc 后面加上一系列参数，成功运行。

### 5.2

在用 openmp 编程的时候忘记加上-fopenmp 参数使得 gcc 按照串行的方式运行，最终使得结果不理想。

解决方法：在 gcc 编译的时候后面加上-fopenmp 参数，使用多线程的方式运行。