作业 4

贾城昊
2021K8009929010

4.1 pthread函数库可以用来在Linux上创建线程，请调研了解pthread_create，pthread_join，pthread_exit等API的使用方法，然后完成以下任务：

（1）写一个C程序，首先创建一个值为1到100万的整数数组，然后对这100万个数求和。请打印最终结果，统计求和操作的耗时并打印。（注：可以使用作业1中用到的gettimeofday和clock_gettime函数测量耗时）；

（2）在（1）所写程序基础上，在创建完1到100万的整数数组后，使用pthread函数库创建N个线程（N可以自行决定，且N>1），由这N个线程完成100万个数的求和，并打印最终结果。请统计N个线程完成求和所消耗的总时间并打印。和（1）的耗费时间相比，你能否解释（2）的耗时结果？（注意：可以多运行几次看测量结果）

（3）在（2）所写程序基础上，增加绑核操作，将所创建线程和某个CPU核绑定后运行，并打印最终结果，以及统计N个线程完成求和所消耗的总时间并打印。和（1）、（2）的耗费时间相比，你能否解释（3）的耗时结果？（注意：可以多运行几次看测量结果）

提交内容：

所写C程序，打印结果截图等

所写C程序，打印结果截图，分析说明等

所写C程序，打印结果截图，分析说明等

调研结果如下：

I. pthread_create 函数

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine) (void *), void *arg);
```

- **pthread_t *thread：**

  传递一个 pthread_t 类型的指针变量，也可以直接传递某个 pthread_t 类型变量的地址。pthread_t 是一种用于表示线程的数据类型，每一个 pthread_t 类型的变量都可以表示一个线程，用于存储新线程的标识符。(pthread_t 类型在 linux 下被定义为： “unsigned long int”）

- **const pthread_attr_t *attr：**

  用于手动设置新建线程的属性，例如线程的调用策略、线程所能使用的栈内存的大小等。大部分场景中，我们都不需要手动修改线程的属性，将 attr 参数赋值为 NULL，pthread_create() 函数会采用系统默认的属性值创建线程。

- **void *(start_routine) (void)：**

  以函数指针的方式指明新建线程需要执行的函数，形参和返回值的类型都必须为 void 类型。void 类型又称空指针类型，表明指针所指数据的类型是未知的。使用此类型指针时，我们通常需要先对其进行强制类型转换，然后才能正常访问指针指向的数据。

- **void *arg：**

  指定传递给 start_routine 函数的实参，当不需要传递任何数据时，将 arg 赋值为 NULL 即可。如果成功创建线程，pthread_create() 函数返回数字 0，反之返回非零值。各个非零值都对应着不同的宏，指明创建失败的原因，常见的宏有以下几种：

  - EAGAIN：系统资源不足，无法提供创建线程所需的资源。
  - EINVAL：传递给 pthread_create() 函数的 attr 参数无效。
  - EPERM：传递给 pthread_create() 函数的 attr 参数中，某些属性的设置为非法操作，程序没有相关的设置权限。

II. pthread_join 函数

```
int pthread_join(pthread_t thread, void ** retval);
```

- **pthread_t *thread：**

传递一个 pthread_t 类型的指针变量，也可以直接传递某个 pthread_t 类型变量的地址。pthread_t 是一种用于表示线程的数据类型，每一个 pthread_t 类型的变量都可以表示一个线程，用于存储新线程的标识符。(pthread_t 类型在 linux 下被定义为： “unsigned long int”）

- **void **retval：**

  参数表示接收到的返回值，如果 thread 线程没有返回值，又或者我们不需要接收 thread 线程的返回值，可以将 retval 参数置为 NULL。

pthread_join() 函数会一直阻塞调用它的线程，直至目标线程执行结束（接收到目标线程的返回值），阻塞状态才会解除。如果 pthread_join() 函数成功等到了目标线程执行结束（成功获取到目标线程的返回值），返回值为数字 0；反之如果执行失败，函数会根据失败原因返回相应的非零值，每个非零值都对应着不同的宏，例如：

- EDEADLK： 检测到线程发生了死锁。
- EINVAL： 分为两种情况，要么目标线程本身不允许其它线程获取它的返回值，要么事先就 已经有线程调用 pthread_join() 函数获取到了目标线程的返回值。
- ESRCH： 找不到指定的 thread 线程。

一个线程执行结束的返回值只能由一个 pthread_join() 函数获取，当有多个线程调用 pthread_join() 函数获取同一个线程的执行结果时，哪个线程最先执行 pthread_join() 函数，执行结果就由那个线程获得，其它线程的 pthread_join() 函数都将执行失败。

III． pthread_exit 函数

```
void pthread_exit(void *value_ptr);
```

- **void *value_ptr：**

  value_ptr:是一个无类型的指针，其指向的数据将作为线程退出时的返回值。如果线程不需要返回任何数据，将 value_ptr 参数置为 NULL 即可。

pthread_exit` 函数用于在线程中显式地退出，单个线程可以通过下列三种方式退出，在不终止整个进程的情况下停止它的控制流。

●线程只是从启动例程中返回，返回值是线程的退出码。

●线程可以被同一进程中的其他线程取消

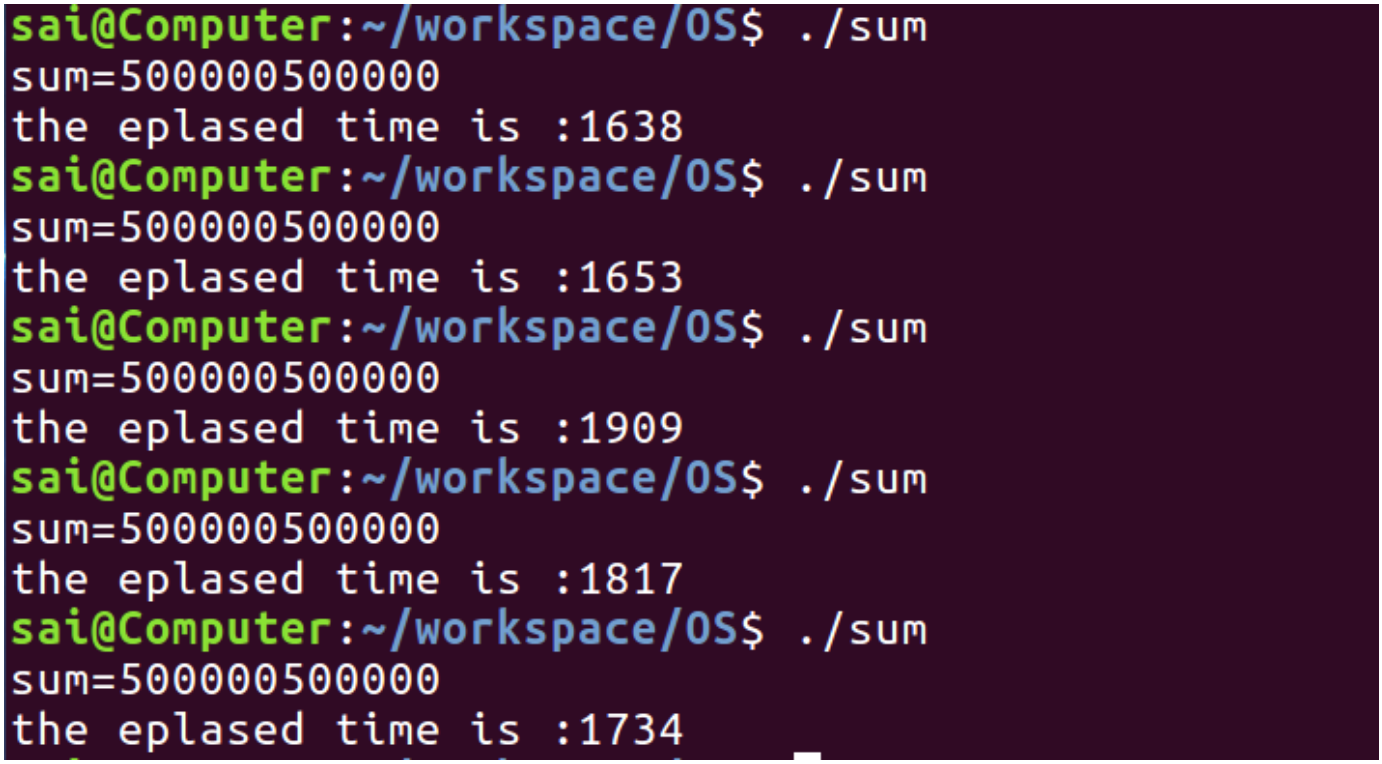
●线程调用 pthread_exit()

(1)单线程求和

1. C 程序如下：

```
#define __USE_GNU
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#define MAXNUM 1000000

int main(){
    long long int sum=0;
    int array[MAXNUM];
    struct timeval START,END;
    for(int i = 0;i < MAXNUM;i++){
        array[i]=i+1;
    }

    gettimeofday(&START,NULL);

    for(int j = 0;j < MAXNUM;j++){
        sum+=array[j];
    }

    gettimeofday(&END,NULL);
    printf("sum=%lld\n",sum);
    printf("the eplased time is :%ld\n",(END.tv_usec-START.tv_usec));
    return 0;
}
```

2. 运行结果：



(2) 无绑核多线程求和:

1. C 程序如下:

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

#define NUM_THREADS 5   // 可以根据需要调整线程数量

int array_size = 1000000;
int *array;
long long global_sum = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// 每个线程的求和函数
```

```
void *thread_sum(void *arg) {
    int thread_id = *((int *)arg);
    long long local_sum = 0;
    int start = thread_id * (array_size / NUM_THREADS);
    int end = (thread_id + 1) * (array_size / NUM_THREADS);

    for (int i = start; i < end; i++) {
        local_sum += array[i];
    }

    // 使用互斥锁保护全局求和变量
    pthread_mutex_lock(&mutex);
    global_sum += local_sum;
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}

int main() {
    // 初始化整数数组
    array = (int *)malloc(array_size * sizeof(int));
    for (int i = 0; i < array_size; i++) {
        array[i] = i + 1;
    }

    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    int state[NUM_THREADS];

    struct timeval start_time, end_time;
    gettimeofday(&start_time, NULL);

    // 创建并启动多个线程
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
```

```
        pthread_create(&threads[i], NULL, thread_sum, (void *)&thread_ids[i]);
    }

    // 等待线程完成
    for (int i = 0; i < NUM_THREADS; i++) {
        state[i] = pthread_join(threads[i], NULL);
    }

    for(int i = 0; i < NUM_THREADS; i++){
        if(state[i]){
            printf("thread %d to finish failed.\n", i);
            return -1;
        }
    }

    gettimeofday(&end_time, NULL);

    long elapse_time = (end_time.tv_sec - start_time.tv_sec) * 1000000 +
(end_time.tv_usec - start_time.tv_usec);
    printf("sum=%lld\nelapse_time=%ld\n", global_sum, elapse_time);
    return 0;
}
```

2. 运行结果：

```
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1839
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1162
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1412
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1857
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1641
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1818
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1959
```

可以看出，不绑核多线程的花费时间并不稳定，部分比单线程还要长，但也有一部分比单线程要短，对此本人分析了一下不绑核多线程相较于单线程的额外时间开销，如下所示：

1. **线程创建和销毁开销**：如果线程的创建和销毁开销比较大，并且任务本身不够大，那么多线程可能会因为线程管理开销而导致运行时间更长。
2. **竞态条件和锁开销**：如果多个线程需要竞争共享资源，而且锁的竞争和互斥开销较大，这可能导致多线程的运行时间延长。
3. **硬件资源限制**：如果计算机的硬件资源有限，多线程可能会导致资源争用，从而降低性能。
4. **线程切换开销**：在多线程程序中，线程之间可能会频繁地切换执行，这会导致线程切换的开销。

同时，也很容易想到，由于多线程能并行处理任务，所以在求和方面花费的时间肯定比单线程更短。

下面，我们可以进一步分析为什么不绑核多线程的处理时间不稳定：首先，线程切换的时机和频率取决于操作系统的调度策略以及系统负载。因此，在不绑核的情况下，线程切换可能会导致程序的执行时间不稳定；其次，系统上运行的其他进程和线程的活动会影响到不绑核的多线程程序的性能。如果系统负载较高，操作系统可能需要更频繁地切换线程，导致执行时间不稳定；而且，绑核的多线程程序可能会受到随机性的影响，例如在某些情况下线程之间的竞争情况可能不同，导致不同的执行时间。

所以综上所述，不绑核多线程相较与单线程有它的优势，但也会带来额外的时间开销，而这个额外的时间开销是不稳定的，所以会出现不绑核多线程的花费时间并不稳定，部分比单线程还要长，但也有一部分比单线程要短的现象。

(3)绑核多线程求和

首先确定了虚拟机使用的CPU核数为：

```
sai@Computer:~/workspace/OS$ cat /proc/cpuinfo| grep "cpu cores"| uniq
cpu cores       : 8
```

可见本人虚拟机的CPU核数为8，可以进行绑核操作：

1. C 程序如下：

```
#include <stdio.h>
#include <stdlib.h>
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>

#define NUM_THREADS 5   // 可以根据需要调整线程数量

int array_size = 1000000;
int *array;
long long global_sum = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// 每个线程的求和函数
void *thread_sum(void *arg) {
    int core_id = *((int*)arg);
    cpu_set_t cpuset;    //CPU 核的位图
    CPU_ZERO(&cpuset);   //将位图清零
    CPU_SET(core_id, &cpuset);   //设置位图第 N 位为 1，表示与 core N 绑定。N 从 0 开始计数
    sched_setaffinity(0, sizeof(cpuset), &cpuset);   //将当前线程和 cpuset 位图中指定的核绑定运行

    int thread_id = *((int *)arg);
    long long local_sum = 0;
    int start = thread_id * (array_size / NUM_THREADS);
    int end = (thread_id + 1) * (array_size / NUM_THREADS);
```

```
    for (int i = start; i < end; i++) {
        local_sum += array[i];
    }

    // 使用互斥锁保护全局求和变量
    pthread_mutex_lock(&mutex);
    global_sum += local_sum;
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}

int main() {
    // 初始化整数数组
    array = (int *)malloc(array_size * sizeof(int));
    for (int i = 0; i < array_size; i++) {
        array[i] = i + 1;
    }

    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    int state[NUM_THREADS];

    struct timeval start_time, end_time;
    gettimeofday(&start_time, NULL);

    // 创建并启动多个线程
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_sum, (void *)&thread_ids[i]);
    }

    // 等待线程完成
    for (int i = 0; i < NUM_THREADS; i++) {
        state[i] = pthread_join(threads[i], NULL);
```

```
    }

    for(int i = 0;i < NUM_THREADS;i++){
        if(state[i]){
            printf("thread %d to finish failed.\n", i);
            return -1;
        }
    }

    gettimeofday(&end_time, NULL);

    long elapse_time = (end_time.tv_sec - start_time.tv_sec) * 1000000 +
(end_time.tv_usec - start_time.tv_usec);
    printf("sum=%lld\nelapse_time=%ld\n", global_sum, elapse_time);
    return 0;
}
```

2. 运行结果：

```
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1012
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1277
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1099
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1133
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1198
```

```
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1195
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=738
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=986
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=1114
sai@Computer:~/workspace/OS$ ./sum
sum=500000500000
elapse_time=922
```

可以看出，绑核多线程的平均花费时间比单线程短的多，而且比不绑核多线程的花费时间也要短不少，对此本人认为的原因可能有一下几点：

1. **减少线程切换开销：** 在多核心的系统中，线程绑定到特定的核心可以减少线程之间的切换开销。当线程绑定到特定核心时，操作系统不需要频繁地将线程从一个核心迁移到另一个核心，从而减少了线程切换的开销。这可以显著提高多线程程序的性能。
2. **提高缓存局部性：** 当线程绑定到特定的核心时，它们更有可能共享相同的 CPU 缓存。这提高了缓存局部性，因为线程访问的数据更有可能在 CPU 缓存中已经存在，而不需要从主内存中加载，从而提高了访问数据的速度。
3. **并行执行：** 多核心系统可以并行执行多个线程，而不绑核的多线程可能会因为线程切换的开销而无法充分利用所有可用的 CPU 核心。绑核多线程可以更有效地利用多核心系统的并行处理能力，从而提高了性能。
4. **避免资源争用：** 在多线程程序中，多个线程可能会竞争共享资源，如内存或锁。当线程绑定到特定核心时，竞争共享资源的可能性降低，因为每个核心上的线程更容易访问自己核心上的本地资源，而不需要与其他核心上的线程竞争。

所以，通过绑核的操作，减少了多线程处理所带来的额外时间开销（如线程的切换，竞争共享资源等），而更加充分利用了所有的 CPU 核心，使得任务能够并行处理，所以其时间开销比不绑核多线程处理低不少，也比单线程处理时间短的多。

(4)补充说明（互斥锁）：

上述多线程的程序中使用了互斥锁，这是因为每个线程都会对全局变量进行访问，而互斥锁是为了保证不会出现竞态条件和数据不一致性问题等问题

互斥锁（Mutex，全名为 Mutual Exclusion）是一种多线程同步机制，用于控制多个线程对共享资源的访问，以避免竞态条件（Race Condition）和数据不一致性问题。互斥锁确保在任何给定时间只有一个线程可以访问共享资源，其他线程必须等待互斥锁被释放才能继续执行。在多线程编程中，互斥锁是非常重要的工具，用于确保线程安全性。

以下是一般的互斥锁的使用步骤：

1. **定义互斥锁变量：**

   首先需要创建一个互斥锁变量，以便线程能够使用它来访问共享资源。在C语言中，可以使用 pthread_mutex_t 类型定义互斥锁变量。可以选择静态初始化或动态初始化互斥锁。

   - 静态初始化：使用 PTHREAD_MUTEX_INITIALIZER 宏来进行初始化，通常在定义变量的同时进行初始化。
   - 动态初始化：使用 pthread_mutex_init 函数进行初始化，通常在运行时进行。

2. **获取互斥锁：**

   在访问共享资源之前，线程需要请求获取互斥锁。使用 pthread_mutex_lock 函数来获取互斥锁。如果互斥锁已经被其他线程占用，当前线程将被阻塞，直到互斥锁可用。

3. **访问共享资源：**

   一旦线程成功获取了互斥锁，它就可以安全地访问共享资源，进行读取或写入操作。

4. **释放互斥锁：**

   当线程完成对共享资源的操作后，应使用 pthread_mutex_unlock 函数来释放互斥锁，以允许其他线程获取它。

5. **销毁互斥锁（可选）：**

   如果使用了动态初始化方式，在不再需要互斥锁时，应使用 pthread_mutex_destroy 函数来销毁互斥锁，以释放相关资源

4.2 请调研了解pthread_create，pthread_join，pthread_exit等API的使用方法后，完成以下任务：

（1）写一个C程序，首先创建一个有100万个元素的整数型空数组，然后使用pthread创建N个线程（N可以自行决定，且N>1），由这N个线程完成前述100万个元素数组的赋值(注意:赋值时第i个元素的值为i）。最后由主进程对该数组的100万个元素求和,并打印结果,验证线程已写入数据。

提交内容：

（1） 所写C程序，打印结果截图,关键代码注释等

(1)

1. C 程序如下（以 N=5 为例）：

```
#include <stdio.h>
#include <stdlib.h>
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>

#define ARRAY_SIZE 1000000
#define NUM_THREADS 5   // 根据需要设置线程数量

int array[ARRAY_SIZE];

// 线程函数，用于填充数组的一部分
void *fill_array(void *arg) {
    int core_id = *((int*)arg);
    cpu_set_t cpuset;     //CPU 核的位图
    CPU_ZERO(&cpuset);   //将位图清零
    CPU_SET(core_id, &cpuset);   //设置位图第 N 位为 1，表示与 core N 绑定。N 从 0 开始计数
    sched_setaffinity(0, sizeof(cpuset), &cpuset);   //将当前线程和 cpuset 位图中指定的核绑定运行

    int thread_id = *(int *)arg;
    int chunk_size = ARRAY_SIZE / NUM_THREADS;
    int start = thread_id * chunk_size;
    int end = (thread_id + 1) * chunk_size;

    for (int i = start; i < end; i++) {
```

```
            array[i] = i;
        }

        return NULL;
    }

    int main() {
        pthread_t threads[NUM_THREADS];
        int thread_ids[NUM_THREADS];
        int state[NUM_THREADS];

        struct timeval START, END;
        long elapse_time;

        // 创建 N 个线程，每个线程填充部分数组
        gettimeofday(&START, NULL);
        for (int i = 0; i < NUM_THREADS; i++) {
            thread_ids[i] = i;
            if (pthread_create(&threads[i], NULL, fill_array,
&thread_ids[i]) != 0) {
                perror("创建线程失败");
                exit(1);
            }
        }

        // 等待所有线程完成
        for (int i = 0; i < NUM_THREADS; i++) {
            state[i] = pthread_join(threads[i], NULL);
        }

        // 主线程对数组求和
```

```
        long long sum = 0;
        for (int i = 0; i < ARRAY_SIZE; i++) {
            sum += array[i];
        }
        gettimeofday(&END, NULL);

        for(int i = 0; i < NUM_THREADS; i++){
            if(state[i]){
                printf("Fail to wait thread %d to finish.\n", i);
                return -1;
            }
        }


        printf("数组的总和：%lld\n", sum);
        elapse_time = (END.tv_sec - START.tv_sec) * 1000000 +
(END.tv_usec - START.tv_usec);

        // 打印当前线程数、计算结果、总用时
        printf("sum=%lld\nelapse_time=%ld\n", sum, elapse_time);

        return 0;
}
```

2. 运行结果

   N=5 时：

```
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2216
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2308
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2187
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2122
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2773
```

N=2 时:

```
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2573
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2657
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2568
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2542
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2505
```

N=10 时:

```
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2452
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2708
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2538
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2796
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=2822
```

N=50时:

```
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=7075
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=7480
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=7390
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=7404
sai@Computer:~/workspace/OS$ ./sum2
数组的总和: 499999500000
sum=499999500000
elapse_time=7250
```

可以看出，当线程数量过多时，多线程带来的额外开销将会远远大于并行处理节约的时间，从而导致总共的花费时间更多。

注：由于这个任务仅仅用多线程进行数组赋值，这样多线程处理不会因为并行处理而存在读写数据不一致等冲突，所以本人在这个任务代码中没有使用互斥锁。