并行与分布式计算作业

第 3 次作业

姓名：郝裕玮
班级：计科 1 班
学号：18329015

一、 问题描述&解决方案&实验结果

1、并行编程主要有哪些模型，各自的特点是什么？

答：（1）共享内存模型（其中包含线程模型）：任何处理器都可以直接引用任何内存位置，便捷。优点为：不需要明确指定任务之间的数据通信；缺点为：理解和管理数据局部性变得更加困难；

（2）消息传递模型：通过显式消息进行通信(send/receive)，仅直接访问私有地址空间（本地内存），是一种远离基本硬件操作的编程模型；

（3）GPGPU 编程模型：GPU 通用计算通常采用 CPU+GPU 异构模式，由 CPU 负责执行复杂逻辑处理和事务处理等不适合数据并行的计算，由 GPU 负责计算密集型的大规模数据并行计算。这种利用 GPU 强大处理能力和高带宽弥补 CPU 性能不足的计算方式以发挥计算机潜在性能，在成本和性价比方面有显著优势；

（4）数据并行系统：由许多简单、廉价的处理器组成的阵列，每个处理器的内存很少。处理器不按指令排序，对数据结构的每个元素的操作进行并行运算。

2、多线程编程中，可重入和线程安全的定义以及关系是什么？

答：可重入：当程序被多个线程反复执行，产生的结果正确。若一个函数只访问自己的局部变量或参数，称为可重入函数；

线程安全：一个函数被称为线程安全的，当且仅当被多个并发线程反复的调用时，它会一直产生正确的结果；

二者之间的关系：

（1）可重入函数是线程安全函数的真子集，即若一个函数是可重入函数，则它一定是线程安全的，但是若一个函数是线程安全的，则它不一定是可重入函数；

（2）可重入函数可以在只有一个线程的情况下运行，而线程安全是在多个线程情况下运行；

（3）线程安全函数能够使不同的线程访问同一块地址空间，而可重入函数要求不同的执行流对数据的操作互不影响从而使得结果是相同的；

（4）如果一个函数中用到了静态变量或全局变量，则它不是线程安全的，也不是可重入的；

3、将以下函数改成可重入函数：

```
int i;
int fun1()
{
    return i * 5;
}
int fun2()
{
   return fun1() * 5;
}
```

答：代码修改为如下所示（见下页）：

```
pthread_mutex_t mutex;
int i;

int fun1(){
    pthread_mutex_lock(&mutex);
    return i*5;
    pthread_mutex_unlock(&mutex);
}

int fun2(){
    pthread_mutex_lock(&mutex);
    return fun1()*5;
    pthread_mutex_unlock(&mutex);
}
```

编程题：利用 LLVM（C、C++）或者 Soot（Java）等工具检测多线程程序中潜在的数据竞争，并对比加上锁以后检测结果的变化，分析及给出案例程序并提交分析报告。

基本思路：

1. 编写具有数据竞争的多线程程序（C 或者 Java）；
2. 利用 LLVM 或者 Soot 将 C 或者 Java 程序编译成字节码；
3. 利用 LLVM 或者 soot 提供的数据竞争检测工具检测；
4. 对有数据竞争的地方加锁，观察检测结果；

答：首先执行下面 2 行代码在 ubuntu 上安装 clang 和 llvm：

```
sudo apt-get install clang
sudo apt-get install llvm
```

再执行该行代码创建 test.c 文件用于编写程序：

```
touch test.c
```

test.c 文件内容如下（包含数据竞争且不加锁，见下页）：

```c
#include<stdio.h>
#include<pthread.h>

int temp;

void *thread1(void *p) {
    temp=1;
    return NULL;
}

void *thread2(void *p) {
    temp=2;
    return NULL;
}

int main(){
    int i;
    pthread_t t[2];
    pthread_create(&t[0],NULL,thread1,NULL);
    pthread_create(&t[1],NULL,thread2,NULL);
    for(i=0;i<=1;i++){
        pthread_join(t[i],NULL);
    }
}
```

根据参考网址可知编译程序代码为：

```
$ clang -fsanitize=thread -g -O1 tiny_race.c
```

```
clang -fsanitize=thread -g -O1 test.c -o test
```

编译运行结果如下所示：

```
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ clang -fsanitize=thread -g -O1 test.c -o test
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ ./test
==================
WARNING: ThreadSanitizer: data race (pid=20083)
  Write of size 4 at 0x000000f12358 by thread T2:
    #0 thread2 /home/consthall/桌面/test.c:16:6 (test+0x4b18f4)

  Previous write of size 4 at 0x000000f12358 by thread T1:
    #0 thread1 /home/consthall/桌面/test.c:9:6 (test+0x4b18c4)

  Location is global 'temp' of size 4 at 0x000000f12358 (test+0x000000f12358)

  Thread T2 (tid=20086, running) created by main thread at:
    #0 pthread_create <null> (test+0x424a2b)
    #1 main /home/consthall/桌面/test.c:25:2 (test+0x4b1944)

  Thread T1 (tid=20085, finished) created by main thread at:
    #0 pthread_create <null> (test+0x424a2b)
    #1 main /home/consthall/桌面/test.c:24:2 (test+0x4b1931)

SUMMARY: ThreadSanitizer: data race /home/consthall/桌面/test.c:16:6 in thread2
==================
ThreadSanitizer: reported 1 warnings
```

由上图可知，LLVM 成功检测到该程序的潜在数据竞争。

修改代码，对数据竞争部分加锁，代码如下所示：

```c
#include<stdio.h>
#include<pthread.h>

int temp;
pthread_mutex_t mutex;

void *thread1(void *p) {
    pthread_mutex_lock(&mutex);
    temp=1;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

void *thread2(void *p) {
    pthread_mutex_lock(&mutex);
    temp=2;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main(){
    int i;
    pthread_t t[2];
    pthread_create(&t[0],NULL,thread1,NULL);
    pthread_create(&t[1],NULL,thread2,NULL);
    for(i=0;i<=1;i++){
        pthread_join(t[i],NULL);
    }
}
```

重新编译运行，结果如下所示：

```
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ clang -fsanitize=thread -g -O1 test.c -o test
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$ ./test
consthall@consthall-Lenovo-XiaoXin-CHAO7000-13:~/桌面$
```

可发现数据竞争已经消失，且 LLVM 未检测到有数据竞争

二、遇到的问题及解决方法

本次实验较为顺利，没有遇到特别困难的问题。