《并行程序设计》知识总结

崔利伟

cui@hellolw.com

摘要

本文是哈尔滨工业大学《并行程序设计》一课的重点总结，内容涵盖了并行程序总论，分类和方法论，概念辨析和具体的实现方法以及性能分析。其中加入了许多作者的理解。

1. 并行计算总论

1.1. 分类

对于每个并行任务，可以按照以下方法判断其所属类别：

- 并行部分是否执行同样的指令？
- 并行部分是否作用于同样的数据？

据此可以分为 (单 | 双) 指令 (单 | 双) 数据四种组合中的一种。

1.1.1. MIMD 进一步分类

对于每个多指令多数据计算机，可以继续划分：

- 处理器间是否共享内存?
- 处理器是否通过总线连接？

来判断是多处理器系统还是多计算机系统。

1.2. 与分布式计算的区别

- 分布式计算指多台**计算机**在**地理**上分布，利用**网络通信**进行协作来完成任务。

- 并行计算指多个**处理器同时**进行协作，分别完成某一任务的子任务。这些处理器可以在一台计算机上，也可以在多台。

因此，两者互不包含，但有交集。

2. 多核处理器

2.1. 分类

根据计算内核的地位是否对等，可分为同构多核和异构多核。

2.2. 多核与单核、多处理器、超线程的比较

2.2.1. 与单核

单核仅有一个计算核心，其上的多个线程由 CPU 进行调度，时间片不重叠。

2.2.2. 与多处理器

多处理器指有主板上搭载多块 CPU，不共享 cache，通过总线连接，而多核处理器指一块单晶硅上集成多个核心。

2.2.3. 与超线程

超线程为 Intel 实现的硬件级别的多线程，通过将去往不同计算单元（如整型和浮点型）的指令流同时运行来获得提升。

3. 并行算法设计

3.1. 方法论

我们以找出一个具有 n 个元素的序列最大值来说明 Foster 方法论。

3.1.1. 划分

由于该序列有 n 个元素，为了尽量精细所以将其划分为 n 个任务，每个任务对应一个数值。

3.1.2. 通讯

由于每个任务不能直接访问另一个任务的数据，所以我们需要为其建立联系。在第一步中，一半任务发送自己的值，另一半接收。之后，发送方任务终止，接收方计算两者的最大值。对剩下的任务递归该过程，一半发一半接直到剩下一个任务。这个任务的计算结果为最终结果。

3.1.3. 映射

如果 n 远远大于我们的处理器数 p，我们可以将 n 个任务进行组合，每个处理器分配 $\frac{n}{p}$ 个子任务来使得通信的开销最小。

3.1.4. 聚集

我们发现在一个物理处理器上保留这 $\frac{n}{p}$ 子任务毫无意义。所以将这些子任务聚集为一个具有 $\frac{n}{p}$ 个数值的任务是更好的选择。

3.2. 评估方法

3.2.1. 延展性

单处理器在给定时间内完成 n 的工作，并行后在同样的时间内完成 m 的工作，延展性为 $\frac{m}{n}$。

3.2.2. 加速比及其定律

- 加速比：$Speedup = \frac{u}{m}$ 其中 u 为单个处理器完成任务用时，m 为并行系统完成任务用时
- Amdahl 定律：$MaximumSpeedup = \frac{S+P}{S+\frac{P}{n}} = \frac{1}{S+\frac{P}{n}}$ 其中 S 为任务中串行部分用时，P 为并行部分用时，n 为处理器数, $S + P = 1$
- Gustafson 定律：$MaximumSpeedup = \frac{S+(P*n)}{S+P} = n + (1 - n) * S$

两个定律的区别在于是否认为所完成的任务量应该随着处理器增多而变大。

3.2.3. 效率

$Efficiency = \frac{Speedup*100}{n}$ n 为处理器数目

4. 并行程序基本概念

4.1. 进程

进程是一个活动的实体，包括程序的代码（代码段），数据（栈，堆和数据段）和状态（程序计数器和寄存器的值）。进程间不共享以上的内容。

4.2. 线程

线程是在单个进程中的逻辑流，分别有各自独立的栈和寄存器。但同一个进程里的线程共享虚拟地址空间，因此线程间的数据可以互相访问（除了虚拟寄存器）。

4.3. 同步机制

4.3.1. 互斥量

当某个线程对互斥量加锁后其他线程在该线程解锁之前无法加锁，因此被阻塞，实现互斥访问。

4.3.2. 临界区

只有一个线程可以进入该区域，之后想进入的线程必须等待指导前一个线程离开。

4.3.3. 信号量

信号量是一个整数变量。当信号量大于 0 时线程可以将其减一并继续执行；如果信号量等于零则线程被阻塞直到被通知。之前使信号量减一的进程完成任务后将信号量加一并通知等待的线程中的一个。

4.3.4. 条件变量

当给定的条件为真时该线程试图获得锁锁；完成任务后该线程给其他被阻塞的线程发信号进行唤醒。当给定条件为假时该线程阻塞直到受到信号（即使没有受到该线程仍会不定期自动唤醒），此时重新测试条件。

4.4. 死锁

两个线程分别拥有一把对方想要的锁。此时双方都在等待对方解锁，但事实上谁都不会解锁，造成程序无法继续执行。

4.5. 饥饿

某个线程由于某种原因无法获得继续执行的权限，只好一直处于阻塞状态。

4.6. 数据竞争

多个线程试图修改一个变量时会发生竞争。则常常是由于错误地假定线程调度存在依赖关系所导致的。

5. Windows 多线程 API

5.1. Win32 API 示例

```
#define NUMTHREADS 4
DWORD WINAPI helloFunc(LPVOID arg) {
  return 0;
}
main() {
  HANDLE hThread[NUMTHREADS];
  for (int i = 0; i < NUMTHREADS; i++)
    hThread[i] = CreateThread(NULL, 0, helloFunc, NULL, 0, NULL);
  WaitForMultipleObject(NUMTHREADS, hThread, TRUE, INFINITE);
}
```

5.2. 通过事件 *(Event)* 同步线程

```
HANDLE evRead, evFinish;
void readThread(LPVOID param) {
  WaitForSingleObject(evRead, INFINITE);
  SetEvent(EvFinish);
```

```
}
void WriteThread(LDVOID param) {
  SetEvent(EvRead);
}
main() {
  evRead = CreateEvent(NULL, FALSE, FALSE, NULL);
  evFinish = CreateEvent(NULL, FALSE, FALSE, NULL);
  _beginthread(ReadThread, 0, NULL);
  _beginthread(WriteThread, 0, NULL);
  WaitForSingleObject(evFinish, INFINITE);
}
```

5.3. 通过临界区 *(Critial Section)* 同步线程

```
#define NUMTHREADS 4
CRITICAL_SECTION g_cs;
int g_sum = 0;
DWORD WINAPI threadFunc(LPVOID arg) {
  g_sum += 1;
  return 0;
}
main() {
  InitializeCriticalSection(&g_cs);
  HANDLE hThread[NUMTHREADS];
  for (int i = 0; i < NUMTHREADS; i++)
    hThread[i] = CreateThread(NULL, 0, threadFunc, NULL, 0, NULL);
  WaitForMultipleObject(NUMTHREADS, hThread, TRUE, INFINITE);
  DeleteCriticalSection(&g_cs);
}
```

5.4. 通过互斥量 *(Mutexes)* 同步线程

```
#define NUMTHREADS 4
int g_sum = 0;
```

```
DWORD ThreadProc() {
  HANDLE hMutex;
  hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, "Mutex.Test");
  g_sum += 1;
  ReleaseMutex(hMutex);
  CloseHandle(hMutex);
  return 0;
}
main() {
  HANDLE hMutex;
  hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, "Mutex.Test");
  hMutex = CreateMutex(NULL, FALSE, "Mutex.Test");
  HANDLE hThread[NUMTHREADS];
  for (int i = 0; i < NUMTHREADS; i++)
    hThread[i] = CreateThread(NULL, 0, threadProc, NULL, 0, NULL);
  WaitForMultipleObject(NUMTHREADS, hThread, TRUE, INFINITE);
  CloseHandle(hMutex);
}
```

5.5. 通过信号量 *(Semaphore)* 同步线程

```
#define NUMTHREADS 4
HANDLE hSem;
int g_sum;
DWORD WINAPI CountFives(LPVOID arg) {
  WaitForSingleObject(hSem, INFINIT);
  g_sum += 1;
  ReleaseSemaphore(hSem, 1, NULL);
}
main() {
  hSem = CreateSemphore(NULL, 1, 1, NULL);
  HANDLE hThread[NUMTHREADS];
  for (int i = 0; i < NUMTHREADS; i++)
    hThread[i] = CreateThread(NULL, 0, threadProc, NULL, 0, NULL);
```

```
  WaitForMultipleObject(NUMTHREADS, hThread, TRUE, INFINITE);
}
```