

进阶实验篇第4章

多线程优化

1

多线程的矩阵乘法实现

2

并行化程序的设计思路

3

性能分析

4

并行程序的进一步优化

5

缓存一致性协议

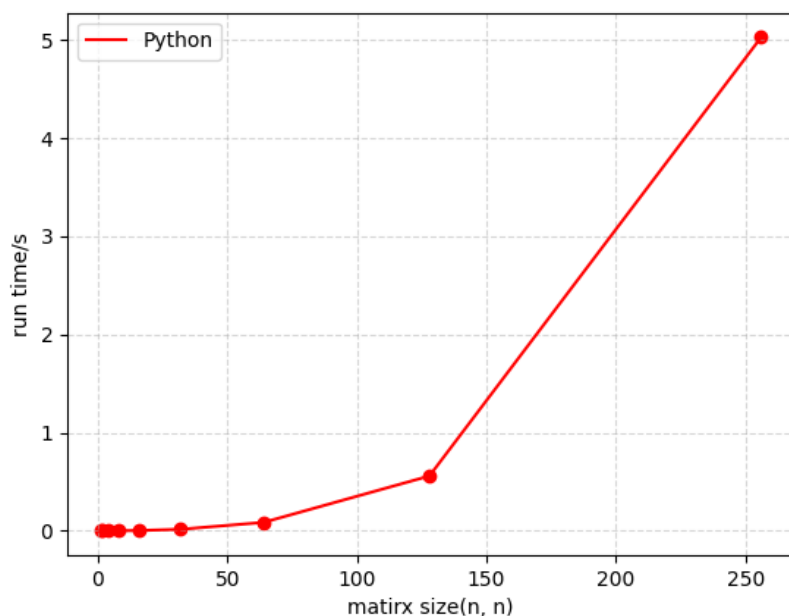
5

延伸阅读



多线程的矩阵乘法实现

矩阵乘法所耗费的时间随矩阵大小的增长呈平方增长的趋式



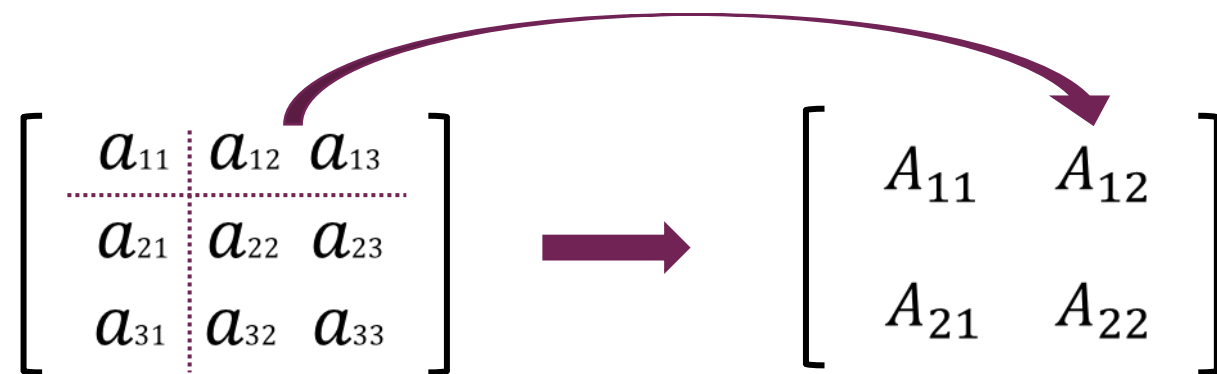
- 多核计算平台逐渐普遍
- 多线程技术可利用平台多核特点大大提高程序运行效率
- 分块矩阵乘法中，计算可分为多个子任务进行



**结合分块矩阵乘法与多线程技术
优化矩阵乘法实现**

分块矩阵乘法

► 矩阵分块


$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

► 分块乘法

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{bmatrix}$$

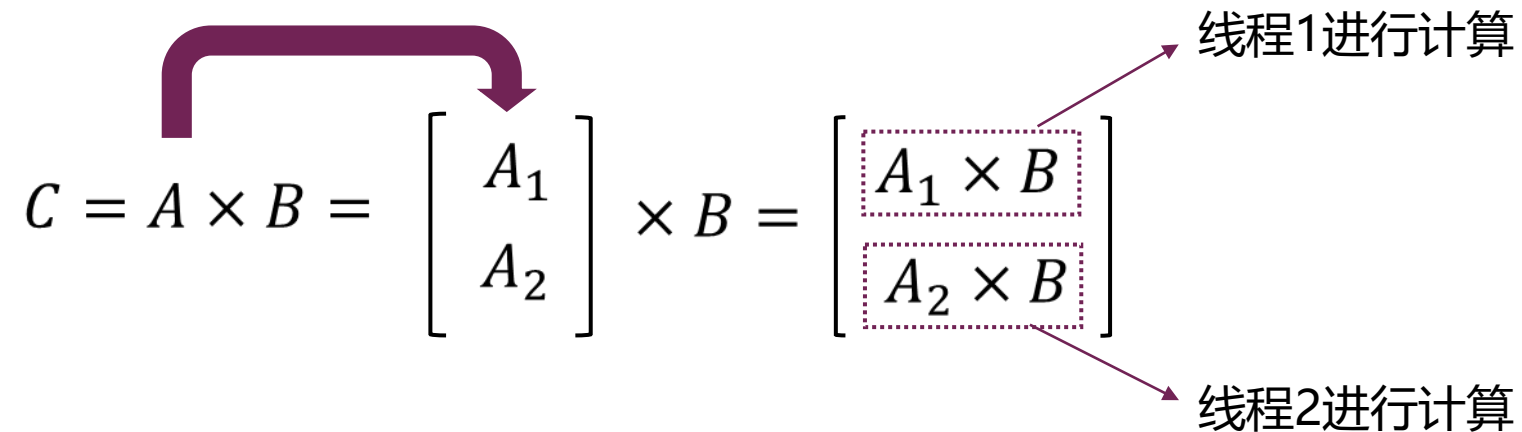
双线程优化实现

本节实验用例：

$$C = A \times B = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \times B = \begin{bmatrix} A_1 \times B \\ A_2 \times B \end{bmatrix}$$

线程1进行计算

线程2进行计算



双线程优化实现 —— 核心代码部分

线程函数参数结构体

```
typedef struct{  
    //  $A \times B = C$   
    float* matrix_A; //矩阵A  
    float* matrix_B; //矩阵B  
    float* matrix_C; //结果矩阵C  
    int m_from; //矩阵A子块计算开始行  
    int m_to; //矩阵A子块计算结束行  
    int k; //矩阵B行数  
    int n; //矩阵B列数  
} mul_thread_args;
```

线程函数

```
void* matrix_mul_th(void* args){  
    //利用参数指针获取乘法所需参数  
    mul_thread_args* mul_args = (mul_thread_args*) args;  
    int k = mul_args->k;  
    int n = mul_args->n;  
    float(*A)[k] = (float(*)[k]) mul_args->matrix_A;  
    float(*B)[n] = (float(*)[n]) mul_args->matrix_B;  
    float(*C)[n] = (float(*)[n]) mul_args->matrix_C;  
  
    int mi, ki, ni;  
    //单块矩阵相乘计算  
    for(mi = mul_args->m_from; mi < mul_args->m_to; mi++){  
        for(ni = 0; ni < n; ni++){  
            for(ki = 0; ki < k; ki++){  
                C[mi][ni] += A[mi][ki] * B[ki][ni];  
            }  
        }  
    }  
    return NULL;  
}
```

双线程优化实现 —— 核心代码部分

```
void matrix_mul(float* matrix_A, float* matrix_B, float* matrix_C, int m, int k, int n){  
    .....  
    pthread_t th1; //线程1的标识符  
    pthread_t th2; //线程2的标识符  
  
    int mid = m/2; //矩阵A分块行数位置  
    mul_thread_args th1_args={ matrix_A, matrix_B, matrix_C, 0, mid, k, n }; //线程1参数  
    mul_thread_args th2_args={ matrix_A, matrix_B, matrix_C, mid, m, k, n }; //线程2参数  
  
    pthread_create(&th1,NULL,matrix_mul_th,&th1_args); //创建线程1并执行  
    pthread_create(&th2,NULL,matrix_mul_th,&th2_args); //创建线程2并执行  
  
    pthread_join(th1,NULL); //等待线程1结束, 结束后则继续向下执行  
    pthread_join(th2,NULL); //等待线程2结束, 结束后则继续向下执行  
}
```


使用time库中的gettimeofday函数来测量矩阵乘法运算时间

```
struct timeval start;  
struct timeval end;  
  
gettimeofday(&start, NULL); //获取乘法开始时间  
matrix_mul(A, B, C, m, k, n); //矩阵乘法  
gettimeofday(&end, NULL); //获取乘法结束时间  
  
float total_time;  
total_time = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec); //计算乘法运行时间
```

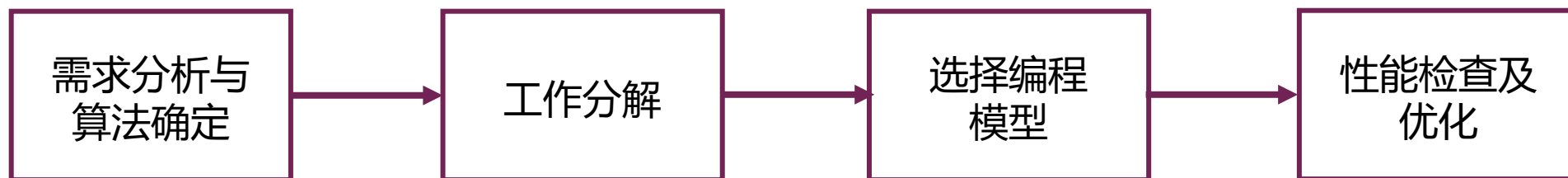
单线程下矩阵乘法运行时间为4.1156s，双线程下运行时间为2.2406s

双线程优化后程序执行时间减少近 $\frac{1}{2}$



并行程序的设计思路

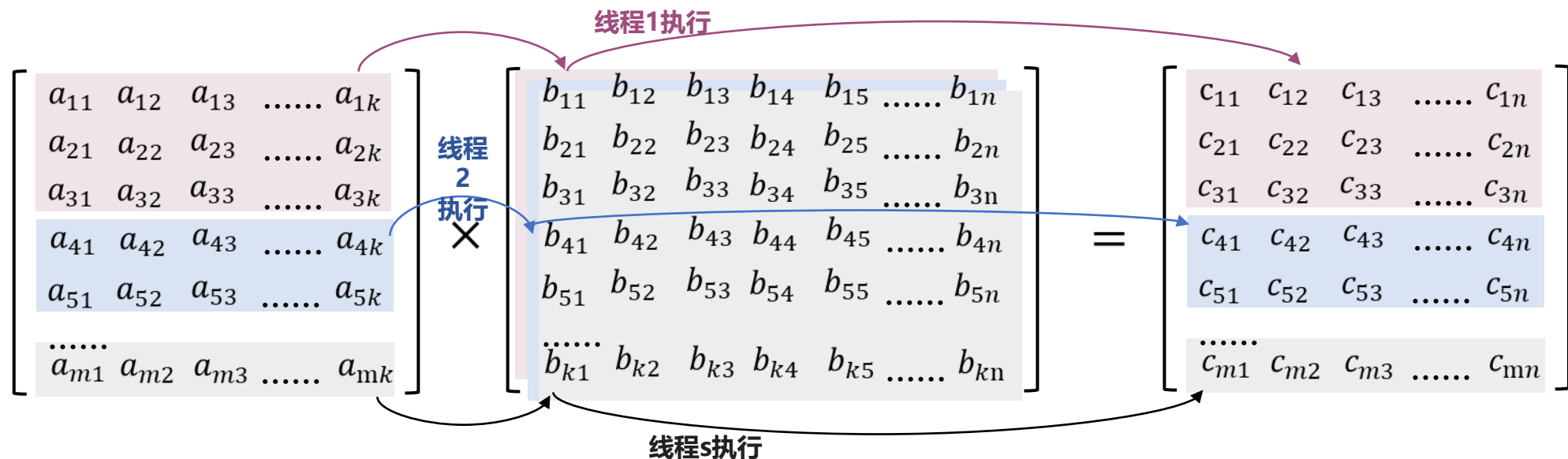
并行程序设计实现流程



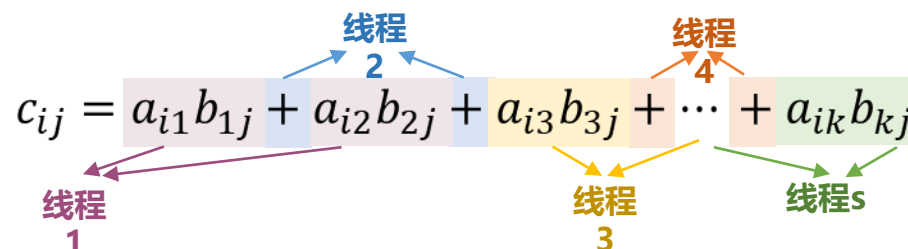
- 判断程序任务是由有并行化的必要性，并根据任务特性设计算法
- 将任务拆解为多个可独立执行的子任务
- 根据需要选择多线程编程模型
- 对程序进行性能测试、评估，并根据结果进行进一步的改良

大规模矩阵的并行化示例

使用s个线程优化矩阵乘法运算： $A_{(m \times k)} \times B_{(k \times n)} = C_{(m \times n)}$



还有其他优化方法吗？ 除了按照行列数进行任务划分以外，还可以按照计算类型划分。一部分线程进行元素间的乘法计算，一部分线程进行结果的相加，从而得到最终的乘积。



性能分析

可调整线程数量的代码示例

```
void matrix_mul(float* matrix_A, float* matrix_B, float* matrix_C, int m, int k, int n, int thread_num){  
    pthread_t th[thread_num];  
    mul_thread_args th_args[thread_num];  
    for(int i = 1; i <= thread_num; i++){ //根据thread_num调整线程数量, 并据此按行划分矩阵A和矩阵B  
        int step = m/thread_num; //计算划分的行距  
        int begin = (i-1) * step; //计算开始行  
        int end = begin + step; //计算结束行  
        end = i == thread_num ? m : end; //处理边界值  
        mul_thread_args args = { matrix_A, matrix_B, matrix_C, begin, end, k, n };  
        th_args[i-1] = args;  
        pthread_create(&(th[i-1]), NULL, matrix_mul_th, &(th_args[i-1])); //创建线程并开始执行  
    }  
    for(int i = 1; i <= thread_num; i++) pthread_join(th[i-1], NULL); //等待线程结束  
}
```

更多的线程, 更好的优化效果? 优化效果如何衡量?

工作负载

工作负载应当易于观察、规模恰当，并具有一定的实际意义。本节实验工作负载即为单次固定规模的矩阵乘法运算。

加速比公式

$$S_p = \frac{T_1}{T_p}$$

※ S_p 为加速比， T_1 指单处理器下的程序运行时间， T_p 是在有 p 个处理器并行系统上的程序运行时间

不同的方案、不同的运行平台呈现不同的效益，需要加速比作为统一的标准衡量优化效果。实验结果中，单线程下矩阵乘法运行时间为 $t_1=4.343s$ ，双线程下矩阵乘法运行时间为 $t_2=2.416s$ ，那么就可以说应用双线程优化的加速比为 $t_2/t_1=1.80$ 。

算术平均与几何平均

算术平均值更关注于数量上的平均值，几何平均值更关注于性能的平均值，应当根据需求选择合适的计算方式。例如本节对于多次实验结果计算平均加速比应当使用几何平均值，得到平均加速比为1.9137。

不同矩阵下双线程的优化效果

实验编号	1	2	3
矩阵大小	500*500	1000*1000	3000*3000
单线程运行时间/s	0.6937	3.8611	131.3340
双线程运行时间/s	0.3474	1.9376	68.4233

不同矩阵规模如何计算平均优化效果？

使用所节省时间的总和没有意义 → 使用加速比

如何计算加速比的平均值？

比值的和没有任何意义 → 使用几何平均值

实验编号	1	2	3
加速比	1.9105	1.9927	1.9194
平均加速比	1.9405		

类比双线程加速比计算过程，可计算其他线程数量情况下的优化加速比：

线程数量	1	2	4	8	...
平均加速比	1.0000	1.9405	3.8040	6.6114	...



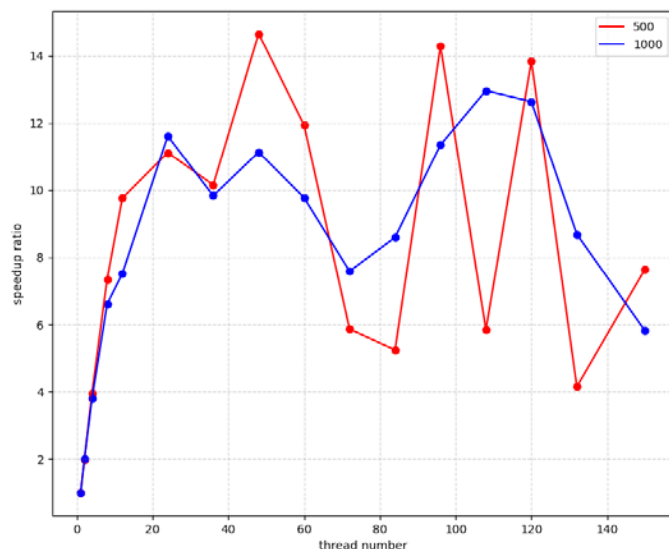
并行程序的进一步优化

优化方案1：调整线程数量

尝试根据计算平台核心数量调整线程数量，使得最大程度发挥硬件性能

实验环境：多核平台，共48个物理核，96个逻辑核

实验设置：在不同线程数量情况下，测量500x500、1000x1000大小矩阵乘法的加速比



在线程数在24、48、96左右时，取得了较为良好的优化效果。最佳线程数并不固定，应结合具体应用情景进行选择

优化方案2：改变矩阵划分方式

矩阵同样能够根据列进行划分，尝试改变矩阵划分的方式进行优化

实验环境：多核平台，共48个物理核，96个逻辑核

实验设置：按照以下公式对矩阵进行行与列的分块，并分为四个线程进行计算

$$C = A \times B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} A_{11} \times B_1 + A_{12} \times B_2 \\ A_{21} \times B_1 + A_{22} \times B_2 \end{bmatrix}$$

？ 计算结果错误 优化失败

计算结果示例

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

期望计算结果

$$\begin{bmatrix} a_{11} \times b_1 + a_{12} \times b_2 \\ a_{21} \times b_1 + a_{22} \times b_2 \end{bmatrix}$$

实际计算结果

只有最终加和结果的一部分？

$$\begin{bmatrix} a_{12} \times b_2 \\ a_{21} \times b_1 + a_{22} \times b_2 \end{bmatrix} \begin{matrix} \times \\ \checkmark \end{matrix}$$

这是多线程共享资源发生**数据冲突**，由于未能保证**缓存一致性**所导致的

Spooler Dir

.....
4: File1
5: File2
6: File3
7: Null
8: Null
.....

Out: 4

In: 7

Thread A: $N_f_s = In; //In == 7$
InsertFileIntoSpooler(N_f_s);
 $In = N_f_s++;$ **$//In == 8$**

 CPU switch (Correct)

Thread B: $N_f_s = In; //In == 8$
InsertFileIntoSpooler(N_f_s);
 $In = N_f_s++;$ **$//In == 9$**

Spooler Dir

.....
4: File1
5: File2
6: File3
7: Null
8: Null
.....

Out: 4

In: 7

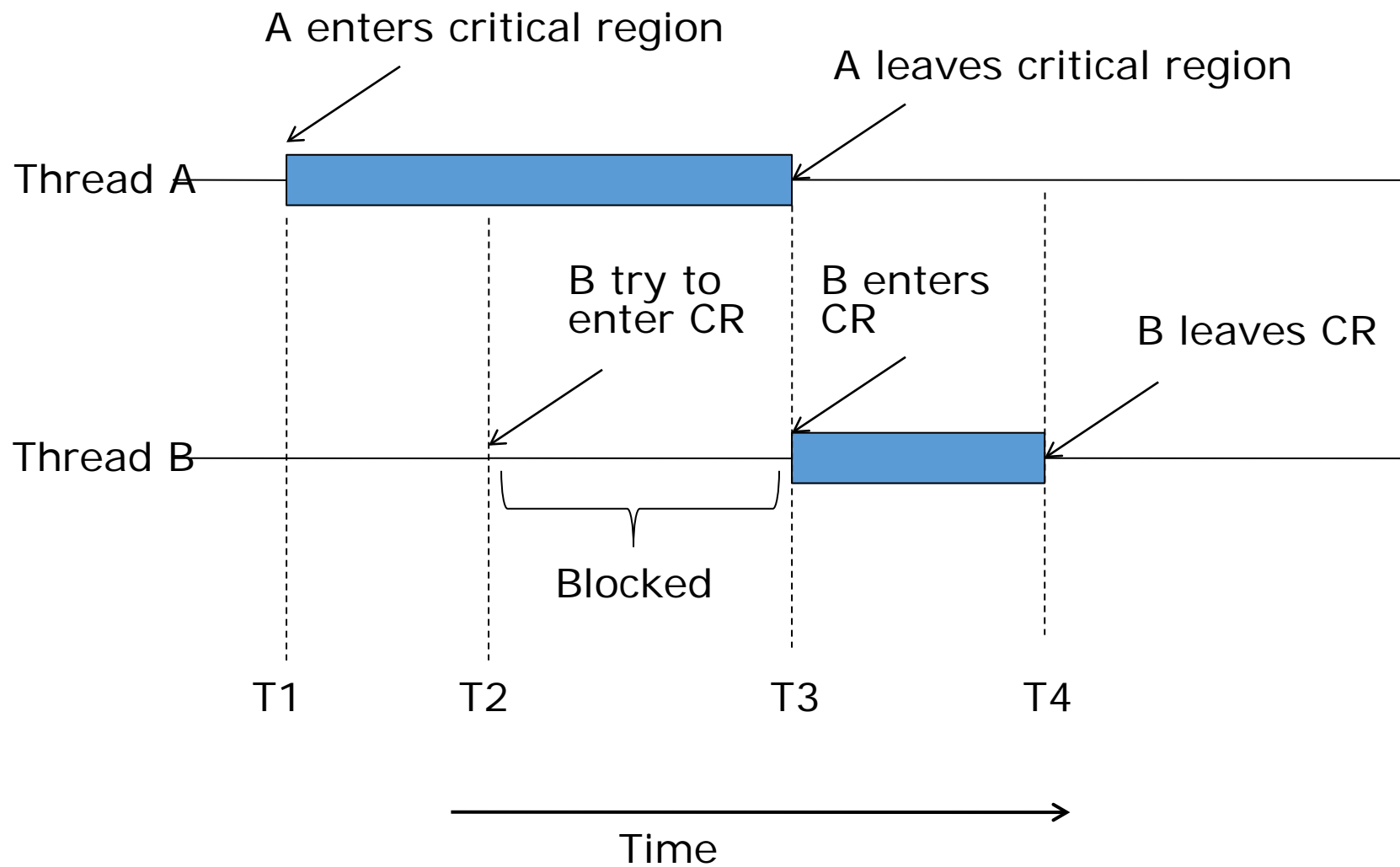
Thread A: $N_f_s = In; //In == 7$

↓ CPU switch

Thrad B: $N_f_s = In; //In == 7$
 $InsertFileIntoSpooler(N_f_s);$
 $In = N_f_s++; //In == 8$

↓ CPU switch, Thread B lost data

Thread A: $InsertFileIntoSpooler(N_f_s);$
 $In = N_f_s++; //In == 8$



缓存一致性

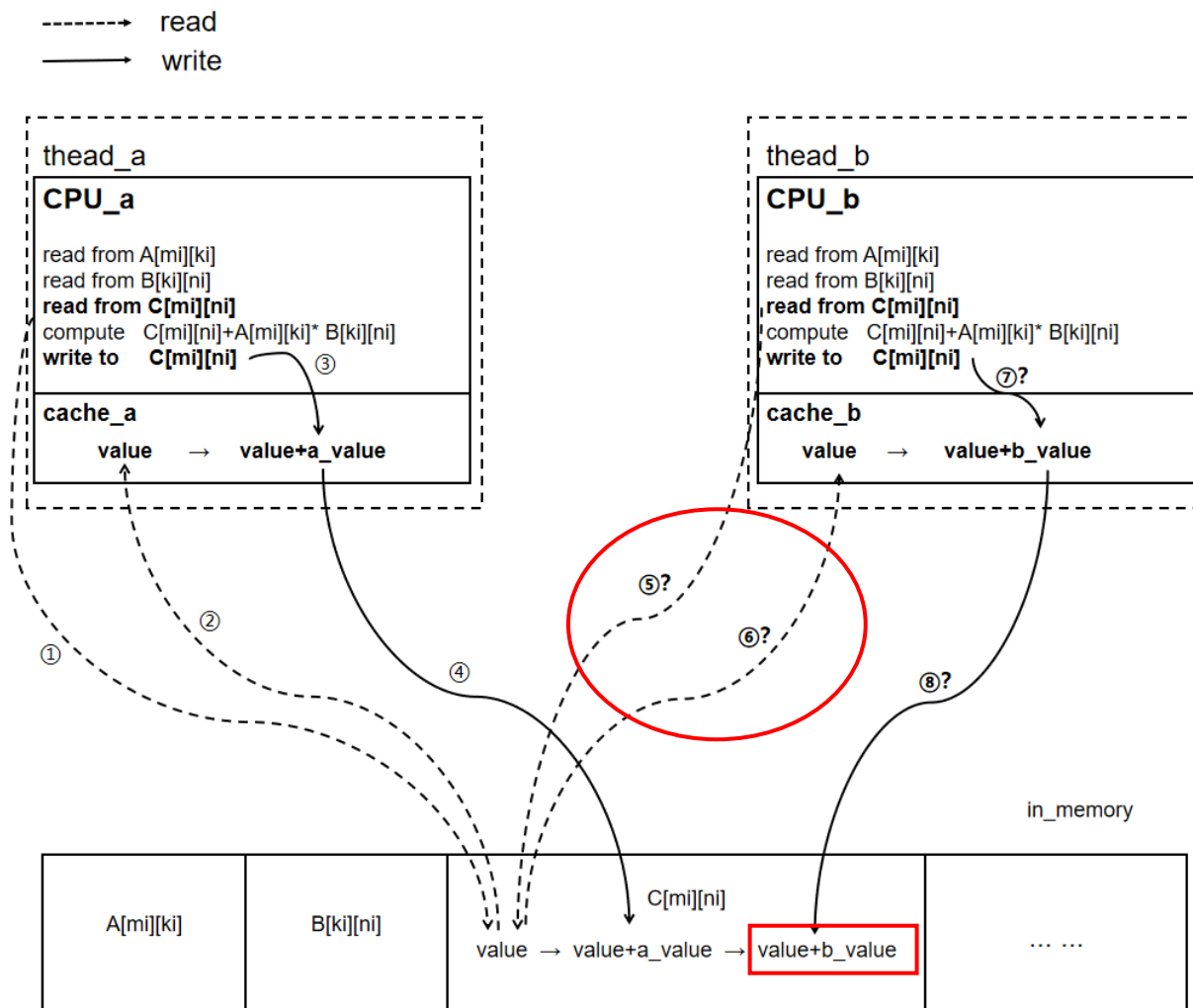
数据冲突

- **数据冲突**：当数据收到的并发请求中存在一条以上的请求对同一条数据进行访问，并且其中一个操作是写操作时，就会发生数据冲突
- **缓存一致性**：当多核中一个核心对缓存中的某个值进行修改时，其他核心也需要看得到并及时更新自己的缓存，从而得到正确的计算结果。

例：本次实验进行举例。假设四个线程中的线程a和线程b，此时同时要进行以下计算

$$C[mi][ni] += A[mi][ki] * B[ki][ni]$$

缓存一致性 数据冲突



- ① 线程a向内存发起读取C[m][n]的请求
- ② 线程a读回C[m][n]的值并放入对应cache中
- ③ 线程a将计算结果写到对应cache中
- ④ 线程a对应的cache将结果更新到内存中

- ⑤ 线程b向内存发起读取C[m][n]的请求
- ⑥ 线程b读回未更新的C[m][n]的值，放入cache中
- ⑦ 线程b将计算结果写到对应cache中
- ⑧ 线程b对应的cache将结果更新的内存中



缓存一致性 数据冲突

► 共享与同步

- 共享：在多线程操作中，多个线程可能需要处理同一部分资源，这就是多线程的共享数据
- 同步：多个线程在同一时间段内只能够一个线程执行指定代码，其他线程需要在该线程完成之后才能够继续执行

► 锁与信号量

- 锁：利用锁可以对资源进行锁定，直到指定线程将对应锁定资源处理完成释放后，其他线程才能够进行访问
- 信号量：对应于线程同步的概念，一个线程完成了某个动作后，通过信号告知其他线程再继续进行某些动作

► 锁的具体实现

- CPU相关机制：总线锁与缓存锁
- OS中的锁：重量级锁、自旋锁等悲观锁；轻量级锁、偏向锁等乐观锁

使用锁（或信号量）解决数据冲突问题

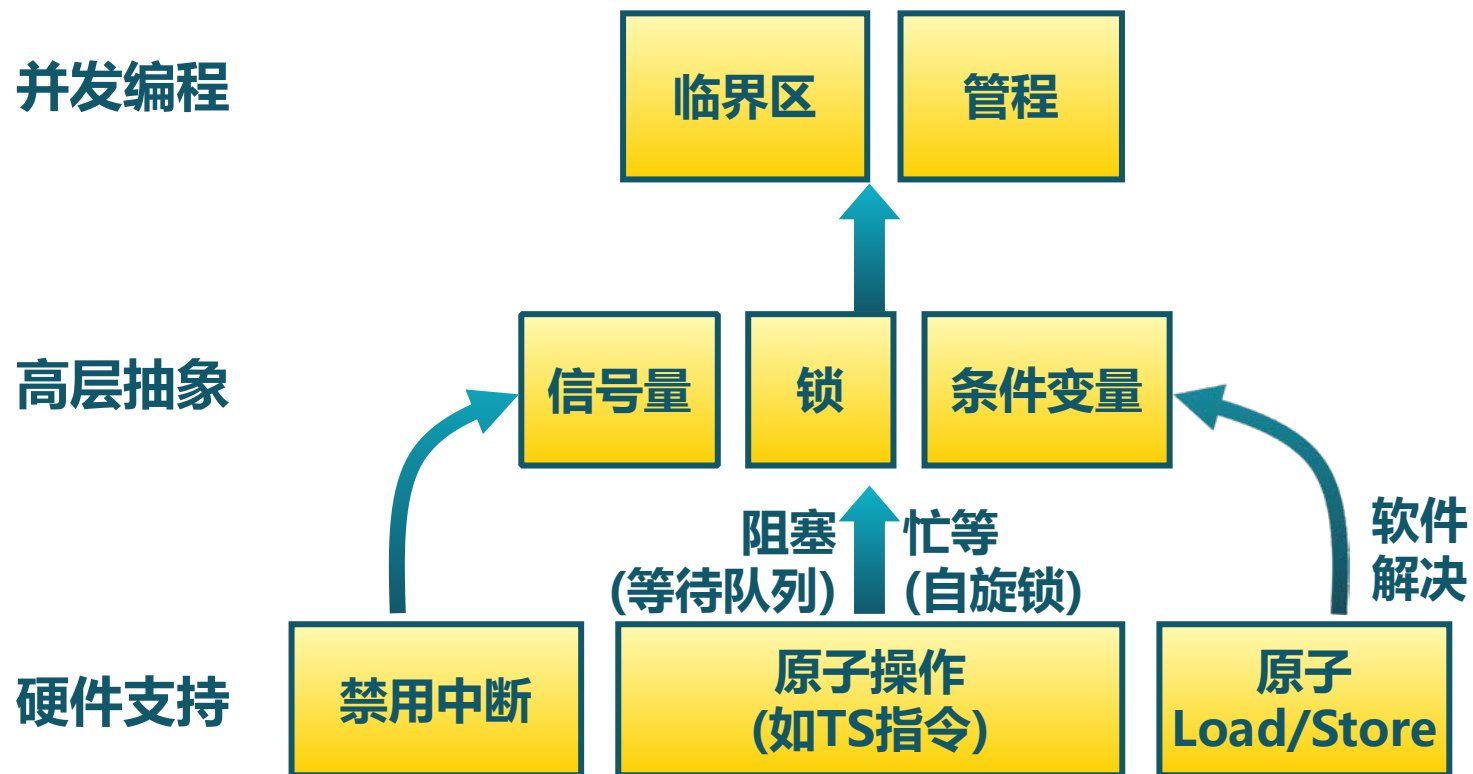
```
int mi, ki, ni;  
for(mi = mul_args->m_from; mi < mul_args ->m_to; mi++){  
    for(ni = 0; ni < n; ni++){  
        for(ki = mul_args ->k_from; ki < mul_args -> k_to; ki++){  
            pthread_mutex_lock(&lock); //使用锁解决冲突（上锁）  
            // sem_wait(&sem); // 使用信号量解决冲突  
            C[mi][ni] += A[mi][ki] * B[ki][ni];  
            pthread_mutex_unlock(&lock); //使用锁解决冲突（解锁）  
            // sem_wait(&sem); // 使用信号量解决冲突  
        }  
    }  
}
```

虽然使用锁（或信号量）解决了数据冲突，得到了正确的结果，但是由于引入了锁机制，需要额外产生保证缓存一致性的管理开销，使得运行时间多出了十几倍

舍弃这样的“优化”方案！

依据实际情况选择多线程应用方法，尽量减少锁或信号量的使用

基本同步方法



中断、异常和系统调用的开销

- PC跳转，但代价远超过函数调用
- 开销：
 - ▣ 引导机制
 - ▣ 建立内核堆栈
 - ▣ 验证参数
 - ▣ 内核态映射到用户态的地址空间
 - 更新页面映射权限
 - ▣ 内核态独立地址空间
 - TLB

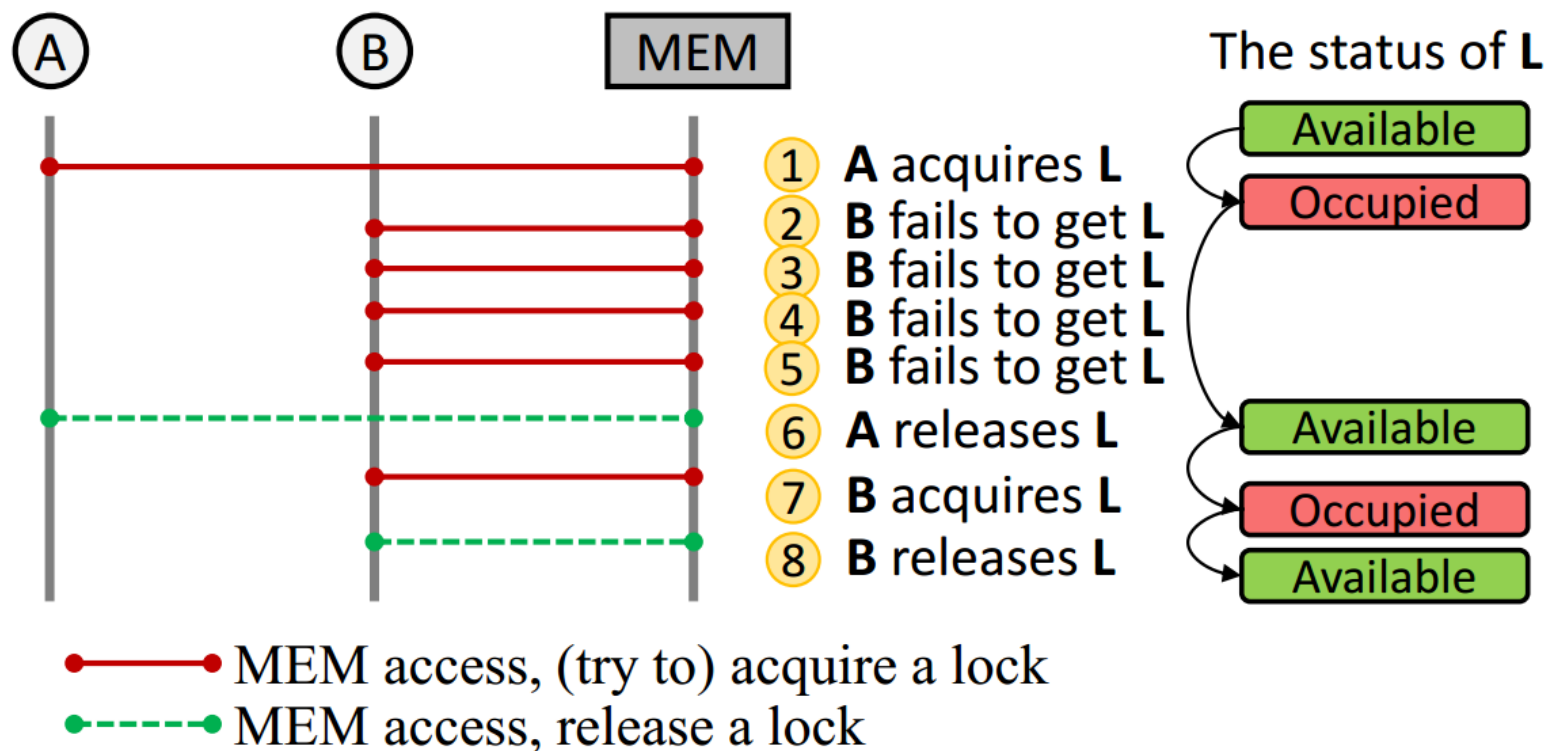


Figure 2. A spin lock based on shared memory.

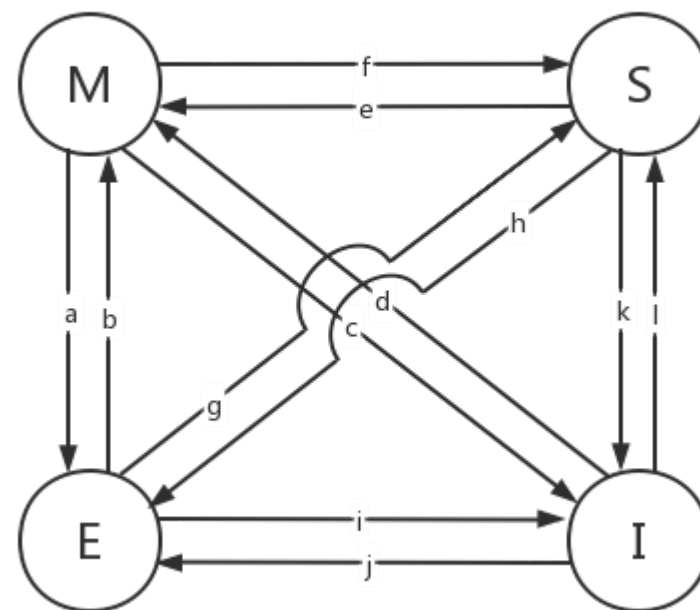


缓存一致性协议

缓存一致性协议保证缓存中所使用共享变量的副本一致

常见的缓存一致性协议——MESI

- M修改 (Modify)：数据有效且被修改，数据和主存数据不一致，且未更新到其他缓存
- E独享 (Exclusive)：数据有效，数据和主存数据一致，但未更新到其他缓存
- S共享 (Shared)：数据有效，数据和主存数据一致，已更新到其他缓存
- I无效 (Invalid)：数据无效

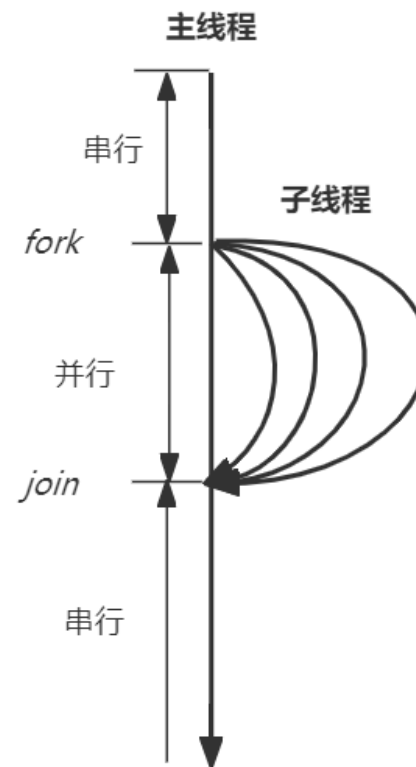


延伸阅读

多线程编程——OpenMP

OpenMP采用fork-join的执行模式，
具有三大编程要素：

- 编译制导指令
- API函数
- 环境变量



benchmark的制定

benchmark，即基准测试，指一组用以测评计算机系统性能的程序。

- ▶ 合格benchmark应当具有的7个组件：工作负载、测试可用行说明、结果报告、使用文档、运行规则、同行审查和公平使用指南
- ▶ 好的基准测试应当具有的特性：相关性、可重复性、公平性、可验证性、可使用性
- ▶ 经典的基准测试程序：SPEC、PARSEC、Mlperf、AnTuTu、Embedded Benchmark



本章作业

程序设计

利用C语言编程的多线程模式实现矩阵乘法

1. 针对不同的矩阵规模，设计多种矩阵分块和并行计算方法
2. 测量多线程模式的加速比，并使用曲线表示加速比的变化趋势
3. 对于同一个规模的矩阵增加其线程的数量并计量性能，多次测量箱线图来呈现结果以表示误差

观察并完成实验报告

1. 观察核心数与线程数之间的关系，特别是线程数超过核心数之后性能的变化
2. 观察不加速情况下出错的概率，以及锁机制对性能的影响

感谢阅读
