《并行程序设计》知识总结

崔利伟
计算机科学与工程学院
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，内容包括并行程序基本概念、设计方法论、不同编程模型的实现、通信性能分析、同步操作和并行一致性。

目录

1. 并行计算 *(Parallel computing)* 总论 4
1.1. 并行计算机机器 *(Parallel computer)* 分类 4
1.1.1. `MIMD`*(Multiple instruction, multiple data)* 进一步分类 . . 4
1.2. 与分布式计算 *(Distributed computing)* 的区别 4
2. 多核处理器 *(Multiprocessor)* 4
2.1. 同构 *(Homogeneous)*/异构 *(Heterogeneous)* 多核比较 4
2.2. 多核 *(Multi-core)* 与单核、多处理器、超线程的比较 5
2.2.1. 与单核 *(Single-core)* 5
2.2.2. 与多处理器 *(Multicomputers)* 5
2.2.3. 与超线程 *(Hyper-Threading)* 5
3. Foster 方法论及算法评估 5
3.1. 方法论 *(Methodology)* 5
3.1.1. 划分 *(Partitioning)* 5
3.1.2. 通讯 *(Communication)* 5
3.1.3. 映射 *(Agglomeration)* 5
3.1.4. 聚集 *(Mapping)* 6

3.2. 评估 *(Evaluation)* 方法 6
3.2.1. 延展性 *(Scale-up)* 6
3.2.2. 加速比 *(SpeedUp)* 及其定律 6
3.2.3. 效率 *(Efficiency)* 6
4. 并行程序 *(Parallel programs)* 基本概念 6
4.1. 进程 *(Process)* 6
4.2. 线程 *(Thread)* 6
4.3. 同步机制 *(Synchronization mechanism)* 7
4.3.1. 互斥量 *(Mutex)* 7
4.3.2. 临界区 *(Critical section)* 7
4.3.3. 信号量 *(Simphore)* 7
4.3.4. 条件变量 *(Condition Variables)* 7
4.4. 死锁 *(Deadlock)* 7
4.5. 阻塞 *(Block)* 7
4.6. 数据竞争 *(Data race)* 7
5. `Windows` 多线程编程 8
5.1. `Win32` 标准示例 8
5.2. `Win32` 下线程同步机制 *(Synchronization)* 8
5.2.1. 通过临界区 *(Critial Section)* 同步线程 8
5.2.2. 通过互斥量 *(Mutexes)* 同步线程 9
5.2.3. 通过信号量 *(Semaphore)* 同步线程 9
5.2.4. 通过事件 *(Event)* 同步线程 10
6. `Linux` 多线程编程 10
6.1. `Pthread` 标准程序示例 10
6.2. `Pthread` 下线程同步机制 *(Synchronization)* 11
6.2.1. 互斥量 *(Mutex)* 11
6.2.2. 条件变量 *(Condition Variables)* 11
7. `OpenMP` 多线程编程 11
7.1. `OpenMP` 编译制导 *(OpenMP directives)* 11
7.2. 数据并行 *(data parallelism)* 12
7.2.1. 使用并行区域 *(region)* 实现 12
7.2.2. 使用 `pragma omp for` 实现 12
7.3. 指令并行 *(Instruction parallelism)* 12

7.4. 其他制导 *(Directive)* 语句 13
7.4.1. 私有变量 *(private variables)* 13
7.4.2. 原子性 *(atmoic)* 13
8. 消息传递 *(Message-Passing Paradigm)* 编程 13
8.1. 消息传递模型 *(Message-Passing Model)* 13
8.2. MPI 编程 13
8.2.1. MPI 基础操作 14
8.2.2. MPI 数据包的收接 14
8.2.3. MPI 的集群通信 *(Collective Communication)* 14
9. 并行算法的通信开销 *(Communication costs)* 15
9.1. 开销模型 *(Cost model)* 15
9.2. 通信模型 *(Communication model)* 15
9.2.1. 储存转发模型 *(Store-and-Forward routing)* 15
9.2.2. 分组路由模型 *(Cutting-Through routing)* 15
9.3. 不同通信拓扑 *(Topology)* 下的开销 15
9.3.1. 点对点 *(Point-to-Point)* 通信 16
9.3.2. 一对多广播 *(One-to-all broadcast)*/多对一规约 *(All-to-one reduction)* 16
9.3.3. 多对多广播 *(All-to-All broadcast)*/多对多规约 *(All-to-all reduction)* 16
9.3.4. 散播 *(Scatter)*/聚集 *(Gather)* 16
9.3.5. 多对多定制 *(All-to-All Personalized)* 17
10. 同步 *(Synchronization)* 操作 17
10.1. 同步物理时钟 *(Synchronizing physical clocks)* 17
10.1.1. 内同步 *(Internal synchronization)* 与外同步 *(External synchronization)* 17
10.1.2. Cristian 算法 17
10.1.3. Berkeley 算法 17
10.2. 同步逻辑时钟 *(Synchronizing logical clocks)* 18
10.2.1. Lamport 时间戳算法 18
10.2.2. 向量时钟 *(Vector clocks)* 算法 18
11. 一致性 *(Consistency)* 18
11.1. 以数据为中心 *(Data-Centric)* 的模型 18

11.2. 以用户为中心 *(Client-Centric)* 的模型 18

1. 并行计算 *(Parallel computing)* 总论

1.1. 并行计算机机器 *(Parallel computer)* 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 *(Multiple instruction, multiple data)* 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 *(Distributed computing)* 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 *(Multiprocessor)*

2.1. 同构 *(Homogeneous)*/异构 *(Heterogeneous)* 多核比较

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 *(Multi-core)* 与单核、多处理器、超线程的比较

2.2.1. 与单核 *(Single-core)*

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 *(Multicomputers)*

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 *(Hyper-Threading)*

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. Foster 方法论及算法评估

3.1. 方法论 *(Methodology)*

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 *(Partitioning)*

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 *(Communication)*

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 *(Agglomeration)*

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 *(Mapping)*

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 *(Evaluation)* 方法

3.2.1. 延展性 *(Scale-up)*

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 *(SpeedUp)* 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)*

$Efficiency = \frac{Speedup*100}{n}$

其中 n 为处理器数目

4. 并行程序 *(Parallel programs)* 基本概念

4.1. 进程 *(Process)*

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 *(Thread)*

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 *(Synchronization mechanism)*

4.3.1. 互斥量 *(Mutex)*

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 *(Critical section)*

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 *(Simphore)*

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 *(Condition Variables)*

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 *(Deadlock)*

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. 阻塞 *(Block)*

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 *(Data race)*

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 多线程编程

5.1. Win32 标准示例

```
#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. Win32 下线程同步机制 *(Synchronization)*

5.2.1. 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);
  // 标准示例
  DeleteCriticalSection(&g_cs);
}
```

5.2.2. 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");
  // 标准示例
  CloseHandle(hMutex);
}
```

5.2.3. 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);
  // 标准示例
}
```

5.2.4. 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);
}
```

6. Linux 多线程编程

6.1. Pthread 标准程序示例

```
  void *compute(void *);
  main() {
    pthread_t p_thread;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    int hits = 0;
    pthread_create(&p_thread,, &attr, compute, (void*)&hits);
    pthread_join(p_thread, NULL);
  }
```

6.2. Pthread 下线程同步机制 *(Synchronization)*

6.2.1. 互斥量 *(Mutex)*

```
pthread_mutex_t lock;
lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);
/* do exclusive job */
pthread_mutex_unlock(&lock);
```

6.2.2. 条件变量 *(Condition Variables)*

- 初始化条件变量

  ```
  pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  ```

- 发出信号

  ```
  pthread_mutex_lock(&mut);
  if (...) pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mut);
  ```

- 等待信号

  ```
  pthread_mutex_lock(&mut);
  while (...) pthread_cond_wait(&cond, &mut);
  /* operation */
  pthread_mutex_unlock(&mut);
  ```

7. OpenMP 多线程编程

7.1. OpenMP 编译制导 *(OpenMP directives)*

OpenMP 使用编译制导语句，结合库和环境变量实现共享内存环境下的并行。指导语句结构如下：

```
#pragma omp construct [clause [clause] ...]
```

7.2. 数据并行 *(data parallelism)*

7.2.1. 使用并行区域 *(region)* 实现

并行区域指的是多个线程重复执行某一区域的代码。注意：在最后一个线程执行完并行区域之前所有线程均被阻塞。可以使用 `nowait` 表示符关闭同步。

```
omp_set_num_threads(4);
#pragma omp parallel
{
  int ID = omp_get_thread_num();
}
```

7.2.2. 使用 `pragma omp for` 实现

指导语句 `#pragma omp for` 建立在 `#pragma omp parallel` 创建线程的基础上，实现多个线程自动分割循环。

```
#pragma omp parallel
#pragma omp for
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

7.3. 指令并行 *(Instruction parallelism)*

OpenMP 使用并行区 *(Parallel Sections)* 为线程分配任务，即指令并行而非数据并行。

```
#pragma omp parallel sections
{
  #pragma omp section
  phase1();
  #pragma omp section
  phase2();
  #pragma omp section
  phase3();
}
```

7.4. 其他制导 *(Directive)* 语句

7.4.1. 私有变量 *(private variables)*

在并行区外的变量均为所有线程共享，可以通过在 `omp parallel for` 后接入 `private` 来声明某变量为各个线程所私有：

```
float x, y; int i;
#pragma omp parallel for private(x, y)
for (i = 0; i < N; i++)
  c[i] = x + y;
```

7.4.2. 原子性 *(atmoic)*

某操作为原子的指该操作不可分割成更小的操作，因此不会受到线程竞争的影响。

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
  #pragma omp atmoic
  c[i] += i;
```

8. 消息传递 *(Message-Passing Paradigm)* 编程

8.1. 消息传递模型 *(Message-Passing Model)*

在该模型中每个处理器负责一个单独进程，因此并不共享地址空间。通信需要通过子例程进行数据包的转移。所以包的传递可以同步或异步进行。

8.2. MPI 编程

MPI 是一个提供消息传递例程的库。

8.2.1. MPI 基础操作

- 初始化和销毁:

  ```
  MPI_Init(int *argc, char ***argv);
  MPI_Finalize();
  ```

- 获取自身所处的通信域的大小和标号:

```
MPI_Comm_Size(MPI_Comm comm, int *size);
MPI_Comm_rank(MPI_Comm comm, int *rank);
```

8.2.2. MPI 数据包的收接

- 阻塞的发送和接收:

```
MPI_Send(void *buf, int count, MPI_Datatype datatype,\
    int dest, int tag, MPI_Comm comm);

MPI_Recv(void *buf, int count, MPI_Datatype datatype, \
    int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- 非阻塞的发送接收和测试:

```
MPI_Isend(void *buf, int count, MPI_Datatype datatype,\
    int dest, int tag, MPI_Comm comm, MPI_Request* request);

MPI_Irecv(void *buf, int count, MPI_Datatype datatype,\
    int source, int tag, MPI_Comm comm, MPI_Request *request);

MPI_test(MPI_Rquest *request, int *flag, MPI_Status *status);
```

8.2.3. MPI 的集群通信 *(Collective Communication)*

MPI 支持以下类型:

- 聚集和散发

```
MPI_Gather(void *sendbuf, int sendcount, MPI_datatype senddatatype,\
    void *recvbuf, int recvcount, MPI_Datatype recvdatatype,\
    int target, MPI_Comm comm);

MPI_Scatter(void *sendbuf, int sendcount, MPI_datatype senddatatype,\
    void *recvbuf, int recvcount, MPI_Datatype recvdatatype,\
    int source, MPI_Comm comm);
```

- 规约

```
MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype,\
    MPI_Op op, MPI_Comm comm);
```

- 定制

```
MPI_Alltoall(void *sendbuf, int sendcount, MPI_datatype senddatatype,\
    void *recvbuf, int recvcount, MPI_Datatype recvdatatype,\
    MPI_Comm comm);
```

9. 并行算法的通信开销 *(Communication costs)*

9.1. 开销模型 *(Cost model)*

- *Startup time*(t_s): 算法启动时间。
- *Per-hop time*(t_h): 在每一跳（节点）上需要的转发/存储延迟。由路由器的延迟决定。
- *Per-word transfer time*(t_w): 在链路上传输每个字所需要的时间。由链路带宽决定。

9.2. 通信模型 *(Communication model)*

9.2.1. 储存转发模型 *(Store-and-Forward routing)*

$t_{comm} = t_s + l(mt_w + t_h)$

其中所需传递的包为 m 个字，需要 l 跳完成转发。

t_h 较小的时可简化为：

$t_{comm} = t_s + lmt_w$

9.2.2. 分组路由模型 *(Cutting-Through routing)*

$t_{comm} = t_s + mt_w + lt_h$

注意由于使用了分组路由，当 t_h 较小时只需要：

$t_{comm} = t_s + mt_w$

9.3. 不同通信拓扑 *(Topology)* 下的开销

我们以圆环（*ring*)、网格环 *(Grid-torus)* 和超立方体 *(Hypercube)* 在分组路由的情况下进行分析。

9.3.1. 点对点 *(Point-to-Point)* 通信

- 圆环：$t_{sum} = t_s + mt_w$
- 网格环：$t_{sum} = t_s + mt_w$
- 超立方体：$t_{sum} = t_s + mt_w$

此时三者的开销相等。

9.3.2. 一对多广播 *(One-to-all broadcast)*/多对一规约 *(All-to-one reduction)*

- 圆环：$t_{sum} = (t_s + mt_w)\log_2 p$
- 网格环：$t_{sum} = (t_s + mt_w)\log_2 p$
- 超立方体：$t_{sum} = (t_s + mt_w)\log_2 p$

此时三者仍然相等，这里 p 为处理器数目。每个接收到数据包的节点继续发送，因此时间为对数形式。

9.3.3. 多对多广播 *(All-to-All broadcast)*/多对多规约 *(All-to-all reduction)*

- 圆环：$t_{sum} = (t_s + mt_w)(p-1)$ 为了最大限度利用带宽，每个点直接向对方发送自己新获得的数据
- 网格环：$t_{sum} = 2t_s(\sqrt{p}-1) + mt_w(p-1)$ 横向进行圆环型广播，之后进行纵向广播
- 超立方体：$t_{sum} = \sum_{i=1}^{\log_2 p}(t_s + 2^{i-1}mt_w)$ (即 $t_{sum} = t_s\log_2 p + mt_w(p-1)$) 每个点分别和前后，上下，左右的点进行数据的交换

9.3.4. 散播 *(Scatter)*/聚集 *(Gather)*

散播指某一个节点上的 p 个数据包分给包含自己在内的 p 个节点。聚集为散播的逆过程。与多对多广播用时相同。

- 圆环：$t_{sum} = (t_s + mt_w)(p-1)$
- 网格环：$t_{sum} = 2t_s(\sqrt{p}-1) + mt_w(p-1)$
- 超立方体：$t_{sum} = \sum_{i=1}^{\log_2 p}(t_s + 2^{i-1}mt_w)$(即 $t_{sum} = t_s\log_2 p + mt_w(p-1)$)

9.3.5. 多对多定制 *(All-to-All Personalized)*

每个节点都需要给其余所有节点发送不同的信息，而在多对多广播中每个节点只需要发送同样的信息，因此两者不同。

- 圆环：$t_{sum} = \sum_{i=1}^{p-1}(t_s + mt_w(p-i))$(即 $t_{sum} = (t_s + mpt_w/2)(p-1)$)
- 网格环：$t_{sum} = (2t_s + mpt_w)(\sqrt{p}-1)$
- 超立方体：$t_{sum} = (t_s + mt_w)(p-1)$

10. 同步 *(Synchronization)* 操作

10.1. 同步物理时钟 *(Synchronizing physical clocks)*

10.1.1. 内同步 *(Internal synchronization)* 与外同步 *(External synchronization)*

- 外同步：对于某个同步界 $D > 0$，总满足 $|S(t) - C_i(t)| < D, fori = 1, 2, ..N$. 其中节点 p_i 的时钟为 C_i，S 代表 `UTC` 时钟，t 为某个具体时间。
- 内同步：对于某个同步界 $D > 0$，总满足 $|C_i(t) - C_j(t)| < D, fori, j = 1, 2, ..N$。

10.1.2. Cristian 算法

1. 节点向时间服务器发送请求，同时记录从发送请求到受到回复所用的时间 T_{round}. 受到的时间信息记为 T.
2. 节点的时钟设为 $T + T_{round}/2$.

该方法为 `C/S` 模型，外同步。

10.1.3. Berkeley 算法

1. 主节点向所有节点发送自己的时间，并要求所有节点返回自己与该时间的时间差。
2. 主节点做平均后将修改量分别发送至所有节点。

该方法为内同步。

10.2. 同步逻辑时钟 *(Synchronizing logical clocks)*

10.2.1. Lamport 时间戳算法

1. 对于节点 i 有时间戳 L_i. 随着事件的发生有 $L_i = L_i + 1$.
2. 当节点 p_i 向节点 p_j 发送一个信息时也携带自己的时间戳 $t = L_i$。当节点 p_j 受到消息时计算 $L_j = max(L_j, t)$，之后继续应用**规则 1**。

两事件有因果时可以保证其时间戳也有序，但反之不可。

10.2.2. 向量时钟 *(Vector clocks)* 算法

1. 每个节点 p_i 维护一个向量时钟 V_i。一开始对于每个维度 $V_i[j] = 0, for j = 1, 2..., N$。
2. 在 p_i 发生了一个事件后保证 $V_i[i] = V_i[i] + 1$
3. 每个节点在发送信息时都携带时间向量 $t = V_i$
4. 当 p_i 受到信息及时间向量后设置 $V_i[j] = max(V_i[j], t[j]), for j = 1, 2..., N$

可以保证事件有因果关系时其向量间也有序，反之亦然。

11. 一致性 *(Consistency)*

11.1. 以数据为中心 *(Data-Centric)* 的模型

- 严格 *(Strict)* 一致性：任意的写操作都会立即对所有的进程可见，因此保证了绝对的时间先后关系。
- 顺序 *(Sequential)* 一致性: 所有的进程对某共享数据的读操作的结果都是以相同顺序出现的。
- 因果 *(Casual)* 一致性: 当对某共享数据的写操作存在因果关系时，所有进程的读操作要体现该因果关系。
- 先入现出 *(FIFO)* 一致性：某进程对共享数据的多个写操作间的顺序会在其他进程的读操作体现。

11.2. 以用户为中心 *(Client-Centric)* 的模型

- 单调读 *(Monotonic Reads)* 一致性：如果一个进程已经读取到某个数据的一个值，那么在本地或者迁移到别的地方再进行读操作的时候之

后读一定会返回这个值或者更新的值。

- 单调写 *(Monotonic Writes)* 一致性：如果一个进程写一个数据，那么它在本地或者迁移到别的地方再进行写操作的时候，原来的写操作要先传播到这个位置。
- 读己之所写 *(Read your Writes)* 一致性：当一个进程对一个数据有写操作，那么这个数据的任何副本上都应该看到这个写操作的影响。
- 读后写 *(Writes follow reads)* 一致性：在读操作后面的写操作要基于跟读操作得到的一样新或更新的值。