南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

实验报告

物理时钟同步算法实验

姓名：林雪

学号：2120220589

专业：计算机科学与技术

指导教师：宫晓利

2022 年 11 月 6 日

目录

一、 Cristian 算法

(一) 算法原理

如图1所示，主机 A 向主机 B 发送时钟请求，与此同时记录发送请求的时间 t_1，主机 B 收到请求后将包含自己当前时间 t 的数据包给发送给主机 A，主机 A 收到数据包并解析出主机 B 的时间后，记录当前时间 t_2，如果主机 A 发送数据包至 B 的时间等于主机 B 发送数据包至主机 A 的时间，也就是经过的网络相同且稳定的情况下，可以知道主机 A 应该将当前时间设置为 $t + (t_2 - t_1)/2$。

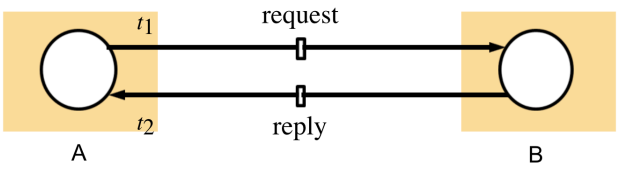

图 1: Cristian 算法

(二) 算法实现

1. 实验平台

Linux ubuntu 5.13.0-30-generic 环境、12 个 12th Gen Intel(R) Core(TM) i5-12400 处理器，每个处理器包含 6 个核心。

2. 实验方案

本实验并没有使用两台主机，而是采取同一台物理主机上两个 socket 进程通信的方式模拟两台主机，其中一个 socket 进程作为主机 A 请求时间，另一个 sockte 进程作为主机 B 发送本地时间。

3. 实现方法

3.1 获取时钟

利用 clock_gettime 函数获取当前时钟，鉴于只在同一台主机上模拟网络通信，因此将时间精确到纳秒。

```
struct timespec start_time, end_time;
clock_gettime(CLOCK_REALTIME, &start_time);
```

图 2: 获取当前时钟

3.2 发送和接收数据包

在 Linux 中，所有的设备都可以看成是一个文件，因此本实验采用 read 函数来读取 socket 数据、write 函数写 socket 数据，同时，由于 clock_gettime 函数返回的时间是整数类型，而 read/write 读写的数据是 const char* 类型，因此在发送和接受数据包的时候需要进行类型转换。

```
string server = to_string(start_time.tv_sec) + "." + to_string(start_time.tv_nsec);
strcpy(servbuffer, server.c_str());
int ret = write(newsockfd, servbuffer, 50);
if (ret < 0)
    cout << "send fail" << endl;
```

图 3: 发送并接受数据包

(三) 实验结果 & 分析

为了获取更准确的结果，将实验重复了 10 次，查看每一次的 T_{round}、完成同步需要的时间以及同步的准确度，其中同步需要的时间在算法的开始和结束分别记录时间并做差记为同步所需时间，利用更新后的时钟同当前物理时钟进行对比判断算法的准确度。

```
snow@snow-MS-7D46:~/Documents/homework$ make run_server
./server
Binding........
Listening for connections......
thread_counter :0
Socket for thread is: 4
connect success 4
server current time is 1667269098s 738949613ns
receive client current time is 1667269098s 739091241ns
Tround is 0s 253286ns
After updated time is 1667269098s 739217884ns
current time is 1667269098s 739202899ns
the accuracy of the algorithm is 0s -14985ns
The Time to complete the algorithm is 0s 282873
------------------------------------------------------------
server current time is 1667269099s 739343943ns
receive client current time is 1667269099s 739552249ns
Tround is 0s 377359ns
After updated time is 1667269099s 739740928ns
current time is 1667269099s 739721302ns
the accuracy of the algorithm is 0s -19626ns
The Time to complete the algorithm is 0s 413497
------------------------------------------------------------
server current time is 1667269100s 739849495ns
receive client current time is 1667269100s 740036623ns
Tround is 0s 1379416ns
After updated time is 1667269100s 740726331ns
current time is 1667269100s 741228911ns
the accuracy of the algorithm is 0s 502580ns
The Time to complete the algorithm is 0s 1412705
------------------------------------------------------------
```

图 4: 实验结果

通过图4结果可以看出，当主机 A 在 $t_1 = 738,949,613ns$ 时发出数据包，主机 B 接收到数据包的时间为 $t = 739,091,241ns$，则数据从主机 A 到主机 B 的时间约为 141,628ns，而主机 A 接受到主机 B 发送回来的时间为 $t_2 = 739,202,899ns$，则数据从主机 B 到主机 A 的时间约为 111,658ns，通过计算可以知道 $T_{round} = t_2 - t_1 = 253,286ns$。

更新后的时间为 $t + T_round = 739,217,884ns$，为了获取算法的准确度，通过获取更新前的系统时间同更新后的时间进行对比，通过结果可以知道，和当前的系统时间产生了 14985ns 的差距。

而通过实验原理可以知道如果主机 A 向主机 B 发送数据的时间很短可以忽略为 0 的情况

下，主机 A 在收到主机 B 的时间 t 后应该将时间更新为 $t + T_{round}$，如果主机 A 想主机 B 发送数据时间很长占据了全部 T_{round} 的情况下，主机 A 应该将时间更新为 t，因此准确度应该在 $[0, T_{round}/2]$ 范围内，结果表明，本实验实现的算法准确程度在理论值之间。

通过 10 次不同的结果做出了下图7，下图7的横坐标是两次数据传输时间的差值，纵坐标是算法的准确程度：

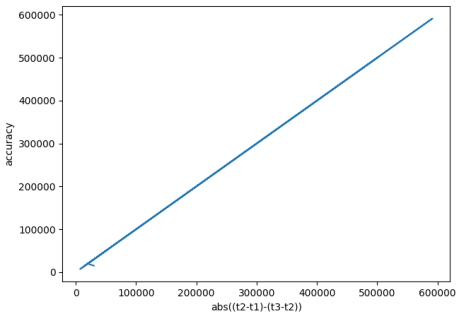

图 5: 数据传输时间差和算法误差的关系

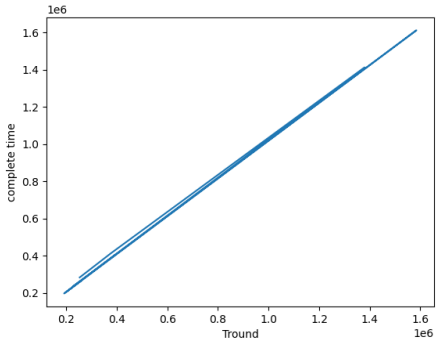

图 6: T_{round} 和算法用时的关系

图 7: 数据分析

可以看出算法的准确程度和数据传输时间差值成正比，这是因为实验原理默认两次数据传输的时间是相等的，因此可以用第一次传输的时间代替第二次传输的时间，如果这两个差值过大，则算法的准确程度也会大幅度下降。

同时分析了算法用时和 T_{round} 时间的关系，通过结果可以看到，整个算法的用时和 T_{round} 成正比，这是因为主机 A 的时钟同步依靠数据从主机 B 发往主机 A，因此要等待 T_{round} 的时间，如果主机 B 需要等待一段时间才能发送数据包，这个时间会更久，这也是后面的 NTP 算法主要解决的问题。

二、 Berkeley 算法

(一) 算法实现

Cristian 算法是用在一个客户端向一个服务器请求正确时间的时候。而 Berkeley 算法是几个客户端之间同步时钟的算法。如图8所示，节点集群首先通过 Change and Robert’s Algorithm 来从一个环里面选择一个节点做为 Master。这个 Master 节点会向集群中的其余 slave 节点请求时间，并计算不同时间和 Master 时间的差值，然后计算算术平均值，最后根据平均值调整集群中所有节点的时钟。

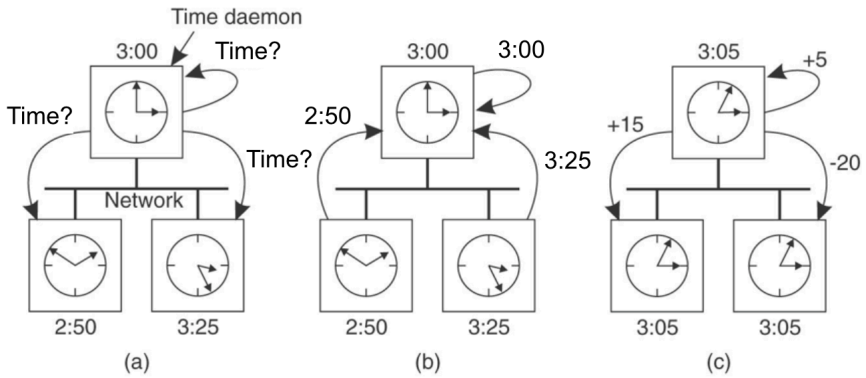

图 8: Berkeley 算法

1. 实验平台

Linux ubuntu 5.13.0-30-generic 环境、12 个 12th Gen Intel(R) Core(TM) i5-12400 处理器，每个处理器包含 6 个核心

2. 实验方案

采取同一台物理主机上 n(n>=2) 个 socket 进程通信的方式模拟 n 台主机，其中一个 socket 作为 Master 节点轮询的请求剩余两个 socket 的时间，其余的 socket 作为 slave 节点向 Master 主机发送时间。

3. 实现方法

2.1 多线程

由于一个 Master 节点可以同多个节点进行通信，同时还需要监听是否有其余节点加入集群，因此本实验采取多线程的方式同多个节点进行通信，Master 节点被 connect 后会开启新的线程用于同 slave 节点进行通信，主线程则负责监听是否有其他节点加入集群。

```
clientfd = accept(sockfd, (struct sockaddr *)&cliaddr, (socklen_t *)&addrlen);
if (clientfd == EAGAIN)
{
    continue;
}
pthread_create(&t[thread_counter], NULL, worker_thread, (void *)&clientfd);
```

图 9: 多线程

2.2 时间更新

Master 节点会询问每一个 slave 节点，收到时间后会进行存储，计算差值的平均值后重新计算每一个节点的更新差值，将差值重新发送给每一个 slave 节点，slave 节点收到差值后对节点更新。

```
        difference_s[i] = res[0] - curr_time.tv_sec; // storing them in an array
        difference_ns[i] = res[1] - curr_time.tv_nsec;
        cout << "Differences received from clock[" << i << "]=>" << difference_s[i] << " " << difference_ns
        sum_of_diff_s = sum_of_diff_s + difference_s[i];
        sum_of_diff_ns = sum_of_diff_ns + difference_ns[i];
    }
}
// 对差值计算算术平均值
avg_s = sum_of_diff_s / (thread_counter + 1);
avg_ns = sum_of_diff_ns / (thread_counter + 1);
```

图 10: 计算平均值

```
for (int i = 0; i < thread_counter; i++)
{
    int temp_fd = globalfd[i];

    // 计算普通节点需要更新的时间偏移
    int new_offset_s = avg_s - difference_s[i];
    int new_offset_ns = avg_ns - difference_ns[i];
    cout << "offset " << temp_fd << " " << new_offset_s << "s " << new_offset_ns << endl;
    string newoff = to_string(new_offset_s) + "." + to_string(new_offset_ns);

    memset(sendbuffer, '\0', sizeof(sendbuffer));
    strcpy(sendbuffer, newoff.c_str());

    int ret = write(temp_fd, sendbuffer, 50);
    if (ret < 0)
        cout << "error in sending offset" << endl;
}
```

图 11: 更新时间

(二) 实验结果 & 分析

如图12展示了 Master 节点的处理时钟同步算法的输出，以及两个 slave 节点的处理时钟同步算法的输出。

```
current time is 1667284633s 139952861ns
Differences received from clock[0]=>0 73432
current time is 1667284633s 139952861ns
Differences received from clock[1]=>0 196379
Average of differences=>0 89937
offset 4 0s 16505
offset 5 0s -106442
Local time for time daemon=>1667284633s 139952861
Time updated for time daemon=>1667284633s 140042798
the accuracy of the algorithm is 0s 230495ns
The Time to complete the algorithm is 0s 320386ns
```

差值

图 12: Master 节点

通过图12可以看到，Master 节点在时间 $t_1 = 139,952,861ns$ 向集群中的其他节点发送时钟请求，slave 节点 1 发送自己的时间 $t_2 = 140,149,240ns$，slave 节点 2 发送自己的时间 $t_3 =$

140, 046, 293，Master 节点接收到这两个节点的时钟后计算时钟之间的差距，然后求的差值的算数平均值为 $t_{avg} = 89,937ns$，将每一个 slave 节点需要更新的 offset 传递给 slave 节点，slave 节点进行调节。然后通过更新后的时间和当前时间计算差值作为算法的准确程度为 230 495ns。同时整个算法的完成时间为 320, 386ns。

```
./client
Local clock time :1667284633s 140149240
1667284633.139952861
1.66728e+09 1.39953e+08
0.-106442
0 -106442
Offset received from daemon process : 0s -106442
Updated local clock time is => 1667284633s 140042798
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=
```

图 13: slave 节点 1

```
Local clock time :1667284633s 140026293
1667284633.139952861
1667284633 139952861
0.16505
0 16505
Offset received from daemon process : 0s 16505
Updated local clock time is => 1667284633s 140042798
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=
```

图 14: slave 节点 2

图 15: 结果展示

为了得到更加有价值的结论，本实验统计了算法准确程度同 rtt、节点个数的关系以及算法完成时间同 rtt 和节点个数的关系。

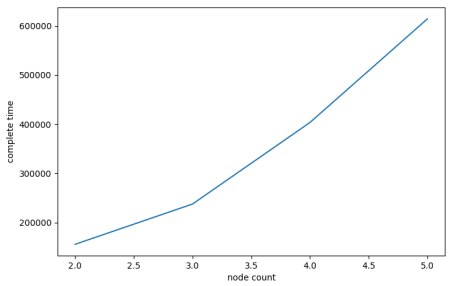

图 16: node & total time

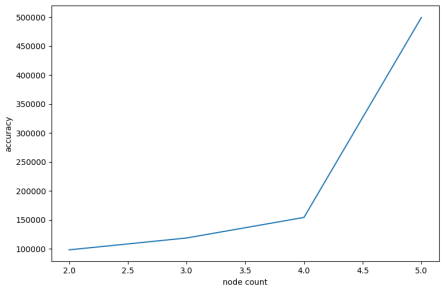

图 17: node & accurary

图 18: 结果展示

通过折线图18可以看到随着节点数量的增加时钟同步算法的总时间和算法误差都会增加，这是因为随着 slave 节点数量的增加，节点的漂移会增大，数据的发送也会增多，因此总时间和误差会增大。

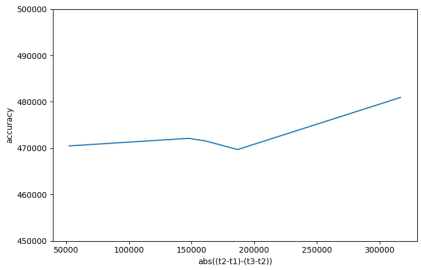

图 19: 数据传输时间差和算法误差的关系

通过图19可知，不同于 Cristian 算法，Berkeley 算法的准确程度受数据传输时间差的影响并不大，这是因为在更新时间的时候并不会依靠数据传输时间的改变，因此数据传输时间差对算法准确程度影响不大。

三、 NTP 算法

NTP(Network Time Protocol) 算法是用来使计算机时间同步化的一种协议，可以在大规模的设备范围内同步矫正时间。

如图20所示，主机 A 向主机 B 发送一个 NTP 报文，该报文带有它离开主机 A 时的时间戳 T_{i-3}。当此 NTP 报文到达主机 B 时，主机加上自己的时间戳 T_{i-2}。当此 NTP 报文离主机 B 时，主机 B 再加上自己的时间戳 T_{i-1}。当主机 A 接收到该响应报文时，主机 A 的本地时间为 T_i。

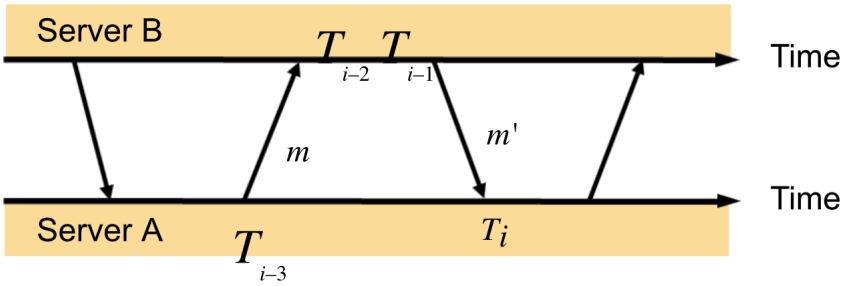

图 20: NTP 算法

可知，主机 A 和主机 B 用于传输报文的时间是 $(T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$，则如果主机 B 向 A 传输报文需要的时间为 0，则报文到达主机 A 的时间为 $T_1 = T_{i-1}$，如果主机 B 向主机 A 传输报文需要的时间为 $(T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$，则报文到达主机 A 的时间为 $T_2 = T_{i-1} + (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$，则主机 A 应该将时间调整为 $(T_1 + T_2)/2$。

(一) 算法实现

1. 实验平台

Linux ubuntu 5.13.0-30-generic 环境、12 个 12th Gen Intel(R) Core(TM) i5-12400 处理器，每个处理器包含 6 个核心

2. 实验方案

虽然 NTP 算法涉及到多台主机，但是两两主机之间采用前面的原理进行时钟同步，因此本实验采用两个 socket 进程模拟两台主机之间的通信，其中一个 socket 作为主机 A 向主机 B 发送时钟同步请求，主机 B 返回自己的时钟。

3. 实现方法

3.1 构造 NTP 报文

按照如图20所示的方法，按照 NTP 报文的结构构造 NTP 报文。

(二) 实验结果 & 分析

实验结果如图28所示，通过结果可以看出，主机 A 向主机 B 发送时钟请求，此时的时间戳为 $T_{i-3} = 942,155,713$，主机 B 收到数据包的时间为 $T_{i-2} = 942,312,477ns$，然后构造数据包发往主机 A，此时的时间戳为 $T_{i-1} = 942,644,660ns$，主机 A 收到数据包的时间为 $T_i = 942,850,079ns$，则按照前面的公式可以计算出主机 A 应更新的时间。同理，按照前面的分析，主机 A 更新时钟

```
send_buf[0] = (recv_buf[0] & 0x38) + 4;
send_buf[1] = 0x01;
*(uint32_t *)&send_buf[12] = htonl(0x4C4F434C);
send_buf[2] = recv_buf[2];
send_buf[3] = (signed char)(-6);
u32p = (uint32_t *)&send_buf[4];
*u32p++ = 0;
*u32p++ = 0;
u32p++;
gettime64(u32p);
*u32p = htonl(*u32p - 60);
u32p++;
*u32p = htonl(*u32p);
u32p++;
*u32p++ = *(uint32_t *)&recv_buf[40];
*u32p++ = *(uint32_t *)&recv_buf[44];
*u32p++ = htonl(recv_time[0]);
*u32p++ = htonl(recv_time[1]);
gettime64(u32p);
*u32p = htonl(*u32p);
u32p++;
*u32p = htonl(*u32p);

int ret = sendto(socket_fd, send_buf, sizeof(send_buf), 0, saddr_p, saddrlen);
```

图 21: NTP 报文构造

的误差应该是 $[-(T_{i-1}-T_i+T_{i-3}-T_{i-2})/2, +-(T_{i-1}-T_i+T_{i-3}-T_{i-2})/2]$，实验利用更新后的时间和当前时间对比计算系统误差，结果表明误差在理论范围中。

```
snow@snow-MS-7D46:~/Documents/homework$ make run_ntp_client
./client 127.0.0.1
client send time is: 1667303451s 942155713ns
client receive time is: 1667303451s 942850079ns
Tround = 0.000362
after updated time is 1667303451s 942825751ns
current time is 1667303451s 942889509ns
snow@snow-MS-7D46:~/Documents/homework$ make run_ntp_client
./client 127.0.0.1
client send time is: 1667303456s 452703464ns
client receive time is: 1667303456s 453338819ns
Tround = 0.000346
after updated time is 1667303456s 453313817ns
current time is 1667303456s 453391059ns
```

图 22: 主机 A 输出

```
snow@snow-MS-7D46:~/Documents/homework$ make run_ntp_server
sudo ./server
listening
receive client time is: 1667303451s 942312477ns
server send time is: 1667303451s 942644660ns
++++++++++++++++++++++++++++++++++++++++++++++++++
receive client time is: 1667303456s 452851576ns
server send time is: 1667303456s 453140703ns
++++++++++++++++++++++++++++++++++++++++++++++++++
receive client time is: 1667303457s 406612592ns
server send time is: 1667303457s 406868606ns
++++++++++++++++++++++++++++++++++++++++++++++++++
receive client time is: 1667303458s 281761428ns
server send time is: 1667303458s 282040073ns
++++++++++++++++++++++++++++++++++++++++++++++++++
```

图 23: 主机 A 输出

图 24: 结果展示

为了避免实验误差，进行了重复实验，部分结果如图28所示，同时对 T_{round} 和算法准确度进行了对比，具体结果如下所示：

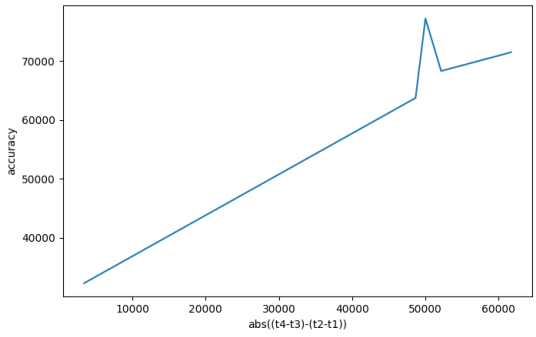

图 25: T_{round}

通过图25展示的结果可以看到，除了偶尔一个实验误差较大，其余的实验误差和 T_{round} 成正比，这个原理同 Cristian 算法的原理是一样的，二者的区别就是网络以及忽略了主机 A、主机 B 处理其他事情的时间。

四、 Lamport 逻辑时钟算法

(一) 算法实现

不同于前面的三种算法，Lamport 是一种逻辑时钟同步算法。

1. 算法定义

Lamport 逻辑时钟算法首先进行了如下的定义：

如果 a 和 b 是同一进程中的两个事件，且 a 在 b 之前发生，则 a->b 为真。

如果 a 是一个进程发送消息的事件，而 b 为另一个进程接受这个消息的事件，则 a->b 为真。

happend-before 关系是一种传递关系，所以若 a->b 且 b->c 则 a->c。如果时间 x 和 y 发生在两个互不交换消息的进程中，那么 x->y 不为真，y->x 也不为真，我们就称这两个事件是并发的。

2. 算法实现

基于前面的定义，为了实现 Lamport 逻辑时钟算法，每个时钟进程 P_i 维护一个局部计数器 C_i，计数器按照如下步骤进行更新：

1. 执行一个事件之前，P 执行 $C_i = C_i + 1$；

2. 当进程 P_i 发送一个消息 m 给 P_j, 在执行 1 的步骤后，把 m 的时间戳 ts(m) 设置为 Ci;

3. 在接收到消息 m 时，进程 Pj 调整自己的局部计数器为 $C_j = max(C_j, ts(m))$, 然后执行第一步，并把消息传送给应用程序。

3. 实验平台

Linux ubuntu 5.13.0-30-generic 环境、12 个 12th Gen Intel(R) Core(TM) i5-12400 处理器，每个处理器包含 6 个核心.

4. 实验方案

本实验并没有采用多台物理主机，也并不进行 socket 通信，而是采用将信息传递转变为字符信息，比如用 r 表示接受消息，用 s 表示发送消息，相同的序号表示同一组发送和接收，比如 r_1 和 s_1 即为一组信息传递。同时用出现的先后顺序表示时间，比如对于一个进程输入的顺序是 s1、s2，则表明对于该进程 s1 事件一定先于 s2 事件发生。

而最终的逻辑时钟通过数字确定，不同进程之间的逻辑数字如果存在大小关系，则存在 happen-before 关系，如果二者的逻辑数字相等，则无明显的先后顺序。

因此存在如下的符号定义：

r_i：接受消息

s_i：发送消息

$a - z$：除了 r 和 s 表示普通事件，既不发送消息也不接收消息。

5. 实现方法

5.1 矩阵表示

根据上面的定义首先确定某一次输入，如图26该文件表示一共存在 3 个进程，其中每个进程发生了 4 次时间，对于进程 1，发生了两次普通事件，发生了一次发送事件和一次接收事件，其

中发送为 s1，则对应的接收事件的表示符号应该是 r1，通过表格中的数据可以知道该消息发送给了进程 3，其余同理。

而对于结果，采用单纯的数字进行表示，如下图27所示，通过结果可以看出，进程 1 的第二个时间要早于进程 3 的第一个事件，进程 1 的第三个事件要晚于进程 2 的第三个事件。

a	s1	r3	b
c	r2	s3	NULL
r1	d	s2	e

图 26: 输入矩阵表示

```
Logical Time Matrix:
1 2 8 9
1 6 7 0
3 4 5 6
```

图 27: 输出矩阵表示

图 28: 结果展示

5.2 逻辑时钟计算

对于如上面所示的输入，我们需要为每一个时间确定一个逻辑时钟，则首先需要计算没一个进程各自的逻辑时钟顺序，也就是对于同一个进程而言，后出现的事件的逻辑时钟数字要大于先出现的事件的逻辑时钟的数字。

```
if (vec[i][j].length() == 1 && count[i] == j)
{
    if (j == 0)
    {
        matrix[i][j] = 1;
        count[i]++;
    }
    else
    {
        matrix[i][j] = matrix[i][j - 1] + 1;
        count[i] += 1;
    }
}
```

图 29: 同一进程逻辑时钟更新

对于多个事件则需要根据 send 和 recv 消息进行更新，当发现了 s_i 就需要找到对应的 r_i，让后者的逻辑时钟大于前者的逻辑时钟。

```
if (vec[i][j].find("r") == 0 && count[i] == j)
{
    if (vec[i][j].compare(receivers[r]) == 0)
    {
        for (int k = 0; k < vec.size(); k++)
        {
            for (int l = 0; l < vec[k].size(); l++)
            {
                if (vec[k][l].compare(senders[s]) == 0)
                {
                    if (j == 0)
                    {
                        matrix[i][j] = max(0, matrix[k][l]) + 1;
                        count[i] += 1;
                    }
                    else
                    {
                        matrix[i][j] = max(matrix[i][j - 1], matrix[k][l]) + 1;
                        count[i] += 1;
                    }
                }
            }
        }
        s++;
        r++;
    }
}
```

图 30: 不同进程逻辑时钟更新

(二) 实验结果 & 分析

通过多次实验，可以看到算法在逻辑时钟具有良好的准确度，同时对比了发生的事件的个数和算法用时的对比，我们将输入文件矩阵的行数和列数相乘，如图31所示，也就是发生事件的个数作为横坐标，算法用时作为纵坐标。

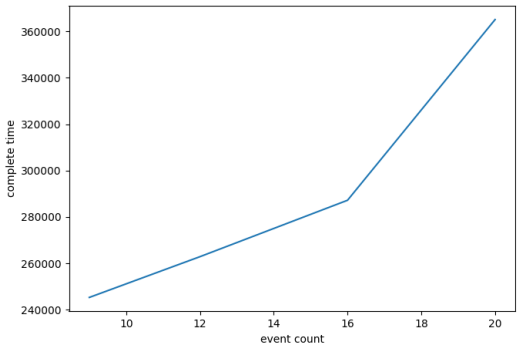

图 31: 算法用时事件个数的关系

通过结果我们可以看到算法用时和事件个数成正比，这是因为事件个数越多，需要进行更新的逻辑顺序越多，因此整个算法的耗时越长。