Introduction

Throughout the practical classes, we will be experiencing various types of coding mechanisms / libraries for running parallel processing, including:

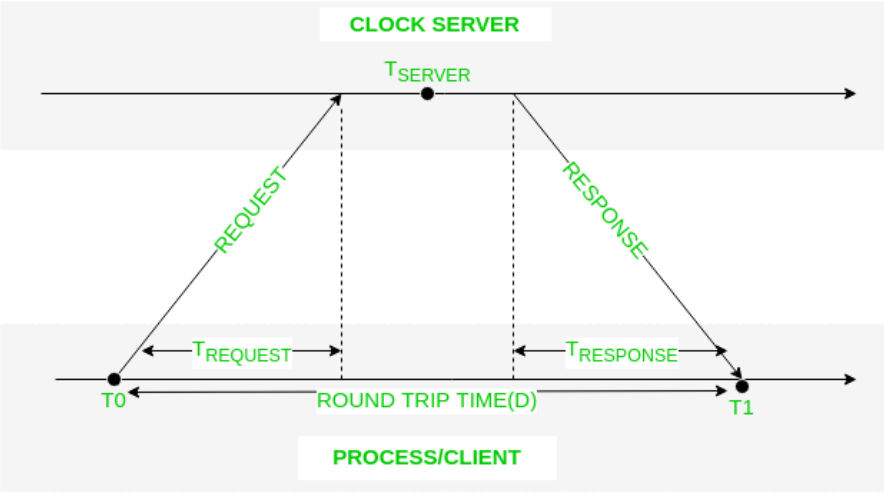
- Threading
- PThread
- OpenMP
- MPI
- CUDA

Before we go into parallel processing coding, as a part of the initial understanding of how a distributed system communicates among the machines, we will start from a simple demonstration of clock synchronization from different systems (computers) within the same network.

Resources: https://www.geeksforgeeks.org/cristians-algorithm/

Cristian's Algorithm is a clock synchronization algorithm used to synchronize time with a time server by client processes. This algorithm works well with low-latency networks where Round Trip Time is short as compared to accuracy while redundancy-prone distributed systems/applications do not go hand in hand with this algorithm. Here Round Trip Time refers to the time duration between the start of a Request and the end of the corresponding Response.

Below is an illustration imitating the working of Cristian's algorithm:



Algorithm:

1) The process on the client machine sends the request for fetching clock time (time at the server) to the Clock Server at time T_0.

2) The Clock Server listens to the request made by the client process and returns the response in the form of clock server time.

3) The client process fetches the response from the Clock Server at time T_1 and calculates the synchronized client clock time using the formula given below.

$T_{CLIENT} = T_{SERVER} + (T_1 - T_0)/2$

where T_{CLIENT} refers to the synchronized clock time,
T_{SERVER} refers to the clock time returned by the server,
T_0 refers to the time at which request was sent by the client process,
T_1 refers to the time at which response was received by the client process

Working/Reliability of the above formula:

$T_1 - T_0$ refers to the combined time taken by the network and the server to transfer the request to the server, process the request, and return the response back to the client process, assuming that the network latency T_0 and T_1 are approximately equal.

The time at the client-side differs from actual time by at most $(T_1 - T_0)/2$ seconds. Using the above statement we can draw a conclusion that the error in synchronization can be at most $(T_1 - T_0)/2$ seconds.

C++ Codes below is used to initiate a prototype of a clock server on local machine, to illustrate the working of Cristian's algorithm:

Question 1)

Based on the reference code in the resources page, try the P1Q1server and P1Q1client code with localhost. Understand the socket connection between server and client, with appropriate ip address and port number, and try to communicate your classmate's machine codes (server and client).

P1Q1server.cpp

```
// C++ equivalent
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>
#include <winsock2.h>
#include <stdio.h>
#include <Ws2tcpip.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")
#pragma warning(disable : 4996).

int main() {
    // Initialize WSADATA in Windows
    WSADATA wsadata;
    struct sockaddr_in server_addr;
    int err = WSAStartup(0x101, (LPWSADATA)&wsadata); // 2.2 version
    if (err != 0) {
        printf("WSAStartup failed with error: %d\n", err);
    }

    // Create socket and set port
    int server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    std::cout << "Socket successfully created" << std::endl;
    int port = 9999;
    const char* ipadd = "127.0.0.1";

    // Bind socket to port
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(ipadd);
    server_addr.sin_port = htons(port);
    bind(server_sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr));
    std::cout << "Socket is listening..." << std::endl;

    // Start listening to requests
    listen(server_sockfd, 5);

    while (true) {
        std::cout << "Server is waiting..." << std::endl;

        struct sockaddr_in client_addr;

        // Establish connection with client
        int client_len = sizeof(client_addr);
        int client_sockfd = accept(server_sockfd, (struct sockaddr*)&client_addr, &client_len);

        // Pending for any client acceptance... then run the following code
        char client_addr_str[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &(client_addr.sin_addr), client_addr_str, INET_ADDRSTRLEN);
        std::cout << "Server connected to " << client_addr_str << std::endl;

        // Respond the client with server clock time
        std::chrono::time_point<std::chrono::system_clock> now = std::chrono::system_clock::now();
```

```
        std::time_t time = std::chrono::system_clock::to_time_t(now);
        std::string time_str = std::ctime(&time);
        const char* time_ch = time_str.c_str();
        send(client_sockfd, time_ch, time_str.size(), 0);
           std::cout << "Server send time data : " << time_ch << " with size " << time_str.size() <<
std::endl;
        std::cout << "Server send time to " << client_addr_str << std::endl;

        // Close the connection with the client process
        closesocket(client_sockfd);
        WSACleanup();
        system("pause");
    }

}
```

* Note that there are two parts of client code that need to be fixed in order to get accurate timing results..

* No solution provided.

P1Q1client.cpp

```
// C++ equivalent
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>
#include <winsock2.h>
#include <stdio.h>
#include <Ws2tcpip.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")
#pragma warning(disable : 4996).

int main() {
    // Initialise WSADATA in Windows
    WSADATA wsadata;
    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;
    int err = WSAStartup(0x202, (LPWSADATA)&wsadata); // 2.2 version
    if (err != 0) {
        printf("WSAStartup failed with error: %d\n", err);
    }

    // Create socket, set port and ip address
    int server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    std::cout << "Socket successfully created" << std::endl << std::endl;
    int port = 9999;
    const char* ipadd = "127.0.0.1";
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(ipadd);
    server_addr.sin_port = htons(port);
```

```
    // Establish connection with server
    int server_len = sizeof(server_addr);
    connect(server_sockfd, (struct sockaddr*)&server_addr, server_len);

                        std::chrono::time_point<std::chrono::system_clock>      request_time_point      =
std::chrono::system_clock::now();
    std::time_t request_time = std::chrono::system_clock::to_time_t(request_time_point);
    auto request_time_epoch = request_time_point.time_since_epoch();

    // Receive data from the server
    char ch[24];
    recv(server_sockfd, ch, 24, 0);

    // Extract and print exactly 24 characters
    int size_arr = sizeof(ch) / sizeof(char);
    std::string str = "";
    for (int x = 0; x < size_arr; x++) {
        str = str + ch[x];
    }
    std::cout << "Data received from server: " << str << std::endl << std::endl;

                       std::chrono::time_point<std::chrono::system_clock>      response_time_point      =
std::chrono::system_clock::now();
    std::time_t response_time = std::chrono::system_clock::to_time_t(response_time_point);
    auto response_time_epoch = response_time_point.time_since_epoch();

    // To be fixed: cannot convert received &ch to time_t correctly
    const char* time_ch = str.c_str();
    std::time_t server_time = std::time_t(time_ch);
    std::string server_time_str = std::ctime(&server_time);
    //std::cout << "Time returned by server: " << server_time_str.c_str() << std::endl;
    std::cout << "Time returned by server: " << str << std::endl << std::endl;

                        std::chrono::time_point<std::chrono::system_clock>      actual_time_point      =
std::chrono::system_clock::now();
    std::time_t actual_time = std::chrono::system_clock::to_time_t(actual_time_point);
    std::string actual_time_str = std::ctime(&actual_time);
    auto actual_time_epoch = actual_time_point.time_since_epoch();

    auto latency_time = response_time_epoch - request_time_epoch;
                                                    auto              latency_time_milis              =
std::chrono::duration_cast<std::chrono::microseconds>(latency_time).count();
    std::cout << "Process Delay latency: " << latency_time_milis << " microseconds\n" << std::endl;
    std::cout << "Actual clock time at client side: " << actual_time_str.c_str() << std::endl;

    //synchronize process client clock time
    std::chrono::time_point<std::chrono::system_clock> client_time_point = actual_time_point;

     // To be fixed: time_t object does not have decimal point for second, cannot do calculation below a
second.
    std::time_t client_time = actual_time; // + (latency_time / 2);
    std::string client_time_str = std::ctime(&client_time);
    auto client_time_epoch = client_time_point.time_since_epoch();
    std::cout << "Synchronized process client time: " << client_time_str.c_str() << std::endl;
```

```
    //calculate synchronization error
    auto error_time = actual_time_epoch - client_time_epoch;
    auto error_time_milis = std::chrono::duration_cast<std::chrono::microseconds>(error_time).count();
    std::cout << "Synchronization error : " << error_time_milis << " microseconds" << std::endl;
    closesocket(server_sockfd);
    WSACleanup();
    system("pause");

}
```

Question 2)

Try to apply Berkeley's algorithm in cpp.
* No solution is provided.

Resources: https://www.geeksforgeeks.org/berkeleys-algorithm/