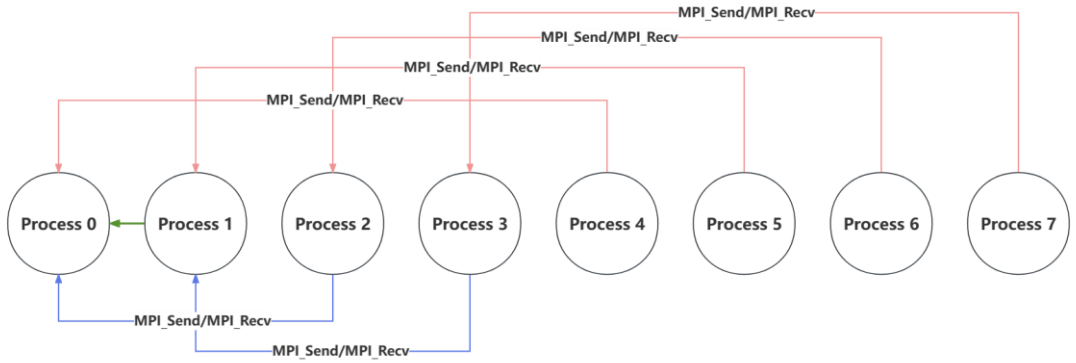
COMP3036J – Assignment

Tairan Ding (UCD Number: 20205710)

Section 1: Communication Required & Algorithm Explanation

To sum the menu prices from all processes using point-2-point communications instead of **MPI_Reduce**, I need to identify required communications. The most straightforward one is that, all the other processes build communications with the root process (process 0). Yet, in this way, process 0 needs to receive (p-1) messages from the other processes, and the max point-to-point communications required for the root process will be O(p) since **MPI_Recv** will block the current process until all required messages received correctly.

Therefore, I decide to develop another algorithm (shown below). To reduce required max point-to-point communications for any given node, I need to release the responsibilities of the root process and allocate them to other processes. As the diagram shown, using this algorithm, max point-to-point communications required are O(log p). In the next section, I will elaborate how I implement the algorithm.



Section 2: Implementation & Code Explanation

```
int my_pow(int base, int y) {
    int result = 1;
    for (int i = 0; i < y; i++) {
        result *= base;
    }
    return result;
}
```

2.1 Function **my_pow** aims to calculate the power of a given *base* raised to a given exponent *y*. It uses a for loop to iterates *y* times, each of which multiplying the *result* by the base. Then, returning *result* after the loop.

2.2 Function **my_log2** is to calculate the logarithm base 2 of a given integer *x*. *result* variable is initialized as -1, indicating the input is illegal. Each iteration performs a right-shifts *x* by one bit (diving x by 2, discarding any reminder). Once *x* becomes 0, the loop terminates and the result is the final output.

```
int my_log2(int x) {
    int result = -1;
    while (x > 0){
        x>>=1;
        result++;
    }
    return result;
}
```

2.3 MPI point-to-point Part

```
long recvData[ITEMS_ON_MENU] = {0};
int power = my_log2( x: size);
for (long i = 1; i<=power; i++){
    long involvedProcNumber = size/(my_pow( base: 2, y: i));
    if (rank+1 > involvedProcNumber){
        // cost_per_menu_item represents the overall cost (qty*price) inside the process
        MPI_Send(cost_per_menu_item, ITEMS_ON_MENU, MPI_LONG, rank-involvedProcNumber, 0, MPI_COMM_WORLD);
    }else{
        MPI_Recv(recvData, ITEMS_ON_MENU, MPI_LONG, rank+involvedProcNumber, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (int ii = 0; ii<ITEMS_ON_MENU; ii++)
            cost_per_menu_item[ii] += recvData[ii];
        }
}
```

2.3.1 – create variable *recvData* as the pointer to receive data from other processes.
2.3.2 – variable *power* indicates the iteration times.
2.3.3 – inside the loop, if the rank of the current process plus 1 (as process rank starts at 0) is on the right-hand side: send its price array to process (rank-involvedProcNumber).
2.3.4 – inside the loop, if the rank of the current process plus 1 (as process rank starts at 0) is on the left-hand side: receive the price array from process (rank+involvedProcNumber). Then, sum the receive array with its own price array *cost_per_menu_item*. The result is store in the *cost_per_menu_item*.
2.3.5 – After the loop is finished, the result will be stored in the *cost_per_menu_item* of the root process (as shown in the diagram above).

Section 3: More Implementation Choices

3.1 – Discarding variable *total_cost_per_menu_item*: because my algorithm decreases involved processes from right (larger ranks) to left (smaller ranks). After the loop, the final result is in root process, and there is no need for additional variable (*cost_per_menu_item* represents the overall cost inside the current process).
3.2 – Creating pow and log2 functions manually: in order to be successfully compiled by provided command, otherwise I need to add -lm in the end of the provided command.