```
// Problem 1.1
// If n is evenly divisible by p
int offset = n/p;
int my_first_i = this_core.index * offset;
int my_last_i  = my_first_i + offset;
for( int i = my_first_i; i < my_last_i; ++i )
{
   int x = compute_value;
   sum += x;
}

// If n is not evenly divisible by p
// [ Explanation ]
// Assign the same workload to the first (p-1) cores.
// Have the last core do the rest.
// Use 'rounding' to make workload_1 close to workload_2.
int workload_1 = rounding( n/p );
int workload_2 = n - (p-1) * workload_1;
int my_first_i = this_core.index * workload_1;
int my_last_i;
if( this_core.index != last_core )
{
   my_last_i = my_first_i + workload_1;
}
else
{
   my_last_i = my_first_i + workload_2;
}
for( int i = my_first_i; i < my_last_i; ++i )
{
   int x = compute_value;
   sum += x;
}



// Problem 1.2
// [ Explanation ]
// Assign each core with approximately the same amount of work.
// Total_Workload = n*(n+1)*t/2
// Partial_Workload (for each core with equal division) = Total_Workload/p = n*(n+1)/2p
// With the first (m) cores out of a total of (p) cores, we want:
// (Xm) jobs are processed such that
// roughly ( m * Partial_Workload ) out of the total work is completed.
// And (Xm) is the 'my_last_i' for the mth core.
// By equating:
// Xm*(Xm+1)*t/2 = (m+1) * Partial_Workload
// The boundary (Xm) can be solved:
// Xm = Rounding( -1 + sqrt( 1 + 4*(m+1)*n*(n+1)/p )/2 )



// Problem 1.3
// Assumption: the total number of core is a power of 2.
int divisor = 2;
int core_difference = 1;
for( int iter = 0; iter < (int)log2(total_core_number); ++iter )
{
   if( this_core.index % divisor == 0 )
   {
```

```
        int paired_core_index = this_core.index + core_difference;
        // receive data from paired core
        this_core.value += paired_core.value;
    }
    else
    {
        int paired_core_index = this_core.index - core_difference;
        // send data to paired core
    }
    divisor *= 2;
    core_difference *= 2;
}


// Problem 1.4
// Assumption: the total number of core is a power of 2.
// Use bitwise operation
int divisor = 2;
int bitmask = 1;
for( int iter = 0; iter < (int)log2(total_core_number); ++iter )
{
    if( this_core.index % divisor == 0 )
    {
        int paired_core_index = this_core.index ^ bitmask;
        // receive data from paired core
        this_core.value += paired_core.value;
    }
    else
    {
        int paired_core_index = this_core.index - core_difference;
        // send data to paired core
    }
    divisor *= 2;
    bitmask << 1;
}

// Problem 1.5
// General Case: the total number of core is not necessarily a power of 2.
// [ Explanation ]
// Cores are divided into two parts.
// The first part has the max possible core number equal to a power of 2.
// The second part contains the rest of the cores.
// Step 1: add the values from cores in the second part into cores in the first part;
// Step 2: do the reduction process with the previous method

int core_number_part1 = pow( 2, (int)log2(total_core_number) );
int core_number_part2 = total_core_number - core_number_part1;
// Step 1
if( this_core.index < core_number_part2 )
{
    int paired_core_index = this_core.index + core_number_part1;
    // receive data from paired core
    this_core.value += paired_core.value;
}
else
{
    int paired_core_index = this_core.index - core_number_part1;
    // send data to paired core
}
// Step 2
// Repeat what was done in problem 1.4.
```