Question 1. [10 points] Consider the following code for dividing up the elements of a one-dimensional array into equally-sized chunks:

```c
struct Region {
    int startIndex; // start index of a region of the array
    int numElements; // number of elements in region
};

void splitArray(int numElements, int numChunks, struct Region regionList[]) {
    int chunk_size = numElements / numChunks;
    int i;

    for (i = 0; i < numChunks; i++) {
        regionList[i].startIndex = i * chunk_size;
        regionList[i].numElements = chunk_size;
    }
}
```

(a) Explain a case where the `splitArray` function will fail to include all of the elements of the array in the created regions.

(b) Explain how to modify the `splitArray` function to fix the problem described in part (a).
Question 2. [15 points] Consider the following data type describing a rectangular region of a two-dimensional array:

```c
struct Region2D {
    int startCol, startRow; // starting column and row of region
    int width, height; // number of columns and rows in region
};
```

(a) Complete the following function, whose purpose is to divide a rectangular array into roughly equally-sized regions. (“Roughly equally-sized” means that the regions should each contain about the same number of elements.) You may assume that `numChunks` is divisible by 2.

```c
void splitArray2D(int numRows, int numCols, int numChunks, struct Region2D regionList[]) {
```

(b) State one reason why it might be advantageous to divide a two-dimensional array into regions which are roughly square (containing about the same number of rows and columns.)
**Question 3.** [20 points] A *gate* is a synchronization mechanism which is either *open* or *closed*. A gate supports three operations:

- **void gate_init(gate_t *g)**: Initialize a gate, putting it in the *closed* state.
- **void gate_set(gate_t *g, bool open)**: Set gate to open state (if open is true) or closed state (if open is false).
- **void gate_enter(gate_t *g)**: If the gate is open, returns immediately. Otherwise, the calling thread waits until the gate is open, and then (when the gate is open) returns.

Show how to implement the *gate_t* data type and its operations. Use pthreads mutex and/or condition variable operations in your implementation.

(a) Define the *gate_t* date type by filling in the struct below.

```c
typedef struct { // fill in fields below

} gate_t;
```

(b) Define the *gate_init, gate_set, and gate_enter* methods as described above. **Important:** the *gate_enter* method must use a blocking wait, meaning the calling thread must be suspended (put to sleep) if the gate is closed. Continue your answer on the next page if necessary.
Question 4. [5 points] In a parallel program, is it ever useful to create more threads than there are physical CPU cores? Explain why or why not.
**Question 5.** [25 points] Consider the following function:

```c
void vec_multiply(float result[], float a[], float b[]) {
    int i;
    for (i = 0; i < MAX; i++) {
        result[i] = a[i] * b[i];
    }
}
```

Complete the following parallel version of this function. It should perform the computation as the original function, but use the specified number of threads to perform the computation in parallel.

```c
void vec_multiply_par(float result[], float a[], float b[], int num_threads) {
```
Question 6. [25 points] The subset sum problem is defined as follows:

Given a set of integers, do the members of any non-empty subset sum to zero?

Sketch how to implement a fork/join parallel program to solve the subset sum problem. Specifically, sketch a definition of a SubsetSumTask class, showing the data members (fields) of the class, any constructors, and the definition of the execute() method. Show the creation of the sub tasks, and the fork and join operations on them.

Hint: each task should have a current subset and a remaining set. The root task’s remaining set is the overall set of integers, and its current subset is empty. Any task whose current subset sums to zero is a solution. When a task is not a solution, its subtasks should be copies of the parent task, except:

- Each task should remove a value from its remaining set.
- One of the two sub-tasks should add the value removed from the remaining set to its current subset.

Any task whose remaining set is empty is a dead-end which should not create any sub tasks.

Use the next page if necessary.
[Continue your solution to Question 6 here if necessary.]