a) Realiza una paralelización tan sólo utilizando las funciones implícitas, aplicando una división de datos de entrada (input block geometric data decomposition, tanto en las inicializaciones como en el bucle de cálculo; de manera que cada thread ejecute un bloque consecutivo de filas de V, y teniendo en cuenta que el número de threads es 8.

```
#define N 4096
#define M 4096
#define L 128

int *V, *W;

void main(int argc, char *argv[]) {
  int i, j, temp, k;

  V = malloc(sizeof(int) * M * N);
  W = malloc(sizeof(int) * L);

  // Paralelizar la inicialización
  #pragma omp parallel for private(j) num_threads(8)
  for (i = 0; i < L; i++) W[i] = 0;

  #pragma omp parallel for private(j) num_threads(8)
  for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
      initialize(V, i, j);

  // Paralelizar el bucle principal
  #pragma omp parallel for private(j, temp, k) num_threads(8)
  for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
      temp = functionA(V, i, j);
      k = functionB(i, j);
```

```
        #pragma omp atomic
        W[k] += temp;
      }
    }
}
```

b) Realiza una paralelización tan sólo utilizando tareas implícitas, aplicando una división de datos de salida (output block geometric decomposition), tanto de las inicializaciones como en el bucle de cálculo de manera que cada thread ejecute un bloque consecutivo de posiciones del vector W, y que el número de threads es 8. (1,5p)

```
#define N 4096
#define M 4096
#define L 128
#define NUM_THREADS 8

int *V, *W;

void main(int argc, char *argv[]) {
  int i, j, temp, k;

  V = malloc(sizeof(int) * M * N);
  W = malloc(sizeof(int) * L);

  // Paralelizar la inicialización de W
  #pragma omp parallel for num_threads(NUM_THREADS)
  for (i = 0; i < L; i++)
    W[i] = 0;

  // Inicializar V
  for (i = 0; i < N; i++)
    for (j = 0; j < M; j++
```

```
        initialize(V, i, j);

    // Paralelizar el bucle principal basado en segmentos de W
    #pragma omp parallel num_threads(NUM_THREADS)
    {
      int thread_id = omp_get_thread_num();
      int start_index = (L / NUM_THREADS) * thread_id;
      int end_index = (thread_id == NUM_THREADS - 1) ? L : start_index + (L /
NUM_THREADS);

      for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
          temp = functionA(V, i, j);
          k = functionB(i, j);

          // Comprobar si k está dentro del rango asignado a este thread
          if (k >= start_index && k < end_index) {
            #pragma omp atomic
            W[k] += temp;
          }
        }
      }
    }
}
```