```c
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif

main(int argc, char **argv)  {

  int i, n = 7,chunk, a[n],suma=0;

  if(argc < 2)   {
       fprintf(stderr,"\nFalta chunk \n");
       exit(-1);
    }
  chunk = atoi(argv[1]);

  for (i=0; i<n; i++)    a[i] = i;

  #pragma omp parallel for firstprivate(suma) lastprivate(suma) schedule(static,chunk)
  for (i=0; i<n; i++)
  {  suma = suma + a[i];
     printf(" thread %d suma a[%d] suma=%d \n",
          omp_get_thread_num(),i,suma);
  }

  printf("Fuera de 'parallel for' suma=%d\n",suma);

}
```

atomic	critical
#pragma omp atomic suma += sumalocal;	#pragma omp atomic suma += sumalocal;

(b) Se pretende que el thread 0 (el master) ejecute la función `printf` que hay justo después de esta directiva (en el bloque estructurado de la directiva). Esta función imprime `tid`, que es el identificador del thread (0 en este caso) que ejecuta `printf` y el contenido de la variable compartida `suma`. Como se ha reflexionado previamente, `suma` puede que no contenga la suma de todos los componentes del vector debido al acceso sin exclusión mutua que realizan los threads previamente en el código.

(c) No. Con esta directiva los threads se esperan en el punto del código donde se encuentra, cuando todos han llegado a ese punto, continúan la ejecución. Es necesario mantener esta barrera para que el thread 0 imprima el contenido de la variable `suma` cuando todos los threads han acumulado en esta variable el resultado parcial de suma que han almacenado en su variable local `sumalocal`. Si se elimina, el thread 0 podría imprimir la suma parcial que él mismo ha calculado o la suma de los valores calculados por alguno de los threads incluido el mismo, no habría, por tanto, garantía de haber acumulado en `suma` todas las sumas parciales calculadas por los threads.

(d) Esta cláusula fija qué tipo de asignación de iteraciones del bucle (planificación) se va a realizar. En este caso fija que se realice una planificación por parte del compilador (estática). Como no se especifica el tamaño de los trozos (es decir, el número de iteraciones consecutivas del bucle) que se van a asignar a los threads en turno rotatorio, se tomará el que fije por defecto la implementación particular de OpenMP que se use.

(e) Como se ha comentado en (b) la función `printf` imprime `tid`, que contiene el identificador del thread que ejecuta el `printf`. Si se usa `single`, la función `printf` la ejecutará el thread que llegue en primer lugar a ese punto del código, podría ser el 0 o cualquier otro. Por tanto, el valor que se imprime como `tid` ahora, usando `single`, puede ser 0, 1,... nthread-1, donde nthread es le identificador del thread que ejecuta el código. La variable `tid` contiene el identificador del thread que ejecuta el código porque se inicializa con el valor que devuelve la función de la biblioteca OpenMP `omp_get_thread_num()`.

(b) El `printf` que se ejecuta cada iteración del bucle imprime el identificador del thread que ejecuta la iteración del bucle (porque es el valor que devuelve `omp_get_thread_num()`), la iteración que se ejecuta (valor de `i`), y el valor de la variable privada del thread `suma` en esa iteración. Los threads imprimen en pantalla el valor de su variable privada `suma` cada vez que añade un nuevo componente del vector a dicha variable.

(c) Imprime el contenido de la variable compartida `suma` que se ha obtenido en la última iteración del bucle, independientemente de cual sea el thread que ha ejecutado esa iteración.

(d) Se utiliza `chunk` para fijar el número de iteraciones consecutivas del bucle que va a contener los trozos de código que se van a usar como unidades de asignación a los threads. Estos trozos se asignan por turno rotatorio. Si, por ejemplo, `chunk` fuese 1, entonces la unidad de asignación sería una iteración y se asignaría entonces al thread 0, las iteraciones 0, `nthreads`, `2nthreads`, ..., al thread 1 las iteraciones 1, `nthreads`+1, `2nthreads`+1, ..., y así sucesivamente. Si, por ejemplo, fuese 2, entonces se asignaría al thread `i` las iteraciones, `i`,`i`+1,`2nthreads`+2`i`, `2nthreads`+2`i`+1, `4nthreads`+2`i`, `4nthreads`+2`i`+1, ...

(e) La cláusula `firstprivate(suma)` fuerzasustituye a que haya una variable `suma` privada a cada thread y a que esta variable se inicialice al valor que tiene la variable compartida `suma` declarada en el thread master. Por tanto, el valor de la variable compartida `suma` declarada en el thread master se copia a todas las variables privadas `suma` de los threads que ejecutan la región paralela.

La cláusula `lastprivate(suma)` fuerza a que haya una variable `suma` privada a cada thread (lo fuerza también `firstprivate`) y a que al valor que tiene la variable privada `suma` en la última iteración del bucle se copie a la variable compartida `suma` declarada en el thread master. En este caso se devuelve el valor de la variable `suma` del thread que ejecute la iteración n-1=6.

(e) La cláusula `reduction(+:suma)` fuerza a que haya una variable `suma` privada a cada thread inicializada a 0 y a que los threads, una vez ejecutadas las iteraciones que tienen asignados, sumen (por usar "+") en exclusión mutua el contenido de su variable privada `suma` a la variable compartida `suma` declarada en el thread master. Por tanto, si se sustiyen `firstprivate(suma)` y `lastprivate(suma)` por `reduction(+:suma)`, la suma se va a calcular correctamente y el programa imprimirá en el segundo `printf` la suma de todos los componentes del vector. El vector de 7 componentes se ha inicializado a 0, 1, 2, 3, 4, 5, 6 (`a[i]=i`); por tanto, se imprimirá 21.