EXAMEN DE CONCURRENCIA Y PARALELISMO

Bloque II: Paralelismo

APELLIDOS: _____ **NOMBRE:** _____

Convocatoria extraordinaria

7 de Julio de 2016

AVISO: *No mezcléis en el mismo folio preguntas del bloque de concurrencia y del bloque de paralelismo. Las respuestas a cada bloque se entregarán por separado.*

[2,5p] 1. **Conceptos de paralelismo.** Una determinada aplicación tiene tres fases bien diferenciadas. La fase inicial lee datos de un fichero de entrada. Una vez leídos todos los datos comienza una segunda fase donde se realiza un determinado trabajo con ellos. Por último una tercera fase escribe los resultados en un fichero de salida. La primera y tercera fase son inherentemente secuenciales. Tan sólo la segunda fase se puede paralelizar usando varios procesos. Asumimos que el programa pasa un 4 % de su tiempo total leyendo los datos de entrada y un 6 % escribiendo el resultado.

a) [0,5p] Calcula, según la ley de Amdhal, cuál es la máxima aceleración que se puede obtener en esta aplicación gracias a la paralelización de la segunda fase?

b) [0,5p] Para abordar la paralelización de esta segunda fase se utiliza un esquema maestro-esclavo. El pseudocódigo de las tareas ejecutadas en los esclavos es el siguiente:

```
resParcial = func()
numT = func2()
Crea numT subtareas y las envía al maestro
```

Esta segunda fase se inicia con una única tarea en el proceso maestro que se envía a ejecutar a alguno de los esclavos. Si el resultado de *func2()* es cero la tarea termina sin crear ninguna subtarea nueva. Al final de cada tarea el esclavo se comunica con el maestro enviando el resultado y las nuevas subtareas creadas para su distribución. Como es un sistema centralizado todas las tareas pasan por el proceso maestro y es éste el que las distribuye de nuevo entre los esclavos.

¿Cuáles serían las ventajas e/o inconvenientes de usar un sistema de asignación dinámico completamente descentralizado para paralelizar la segunda fase? ¿Cuáles serían las ventajas e/o inconvenientes de usar un sistema de asignación estático para paralelizar la segunda fase?

c) [0,5p] En el grafo de la Figura 1 se representa qué tareas crean nuevas subtareas en una determinada ejecución. Por ejemplo, el resultado de la función en T1 es 3, por eso crea T2, T3 y T4. La ejecución de *func()* y *func2()* requiere dos y tres segundos, respectivamente. La creación de tareas tiene un tiempo despreciable (cero segundos). Cualquier comunicación entre procesos dura un segundo (independientemente de su tamaño) y el maestro puede enviar a varios procesos y recibir de varios procesos en el mismo momento.

Si usamos 4 procesos (P0, P1, P2 y P3) en la paralelización de esta segunda fase y P0 es el maestro, ¿qué proceso se encargará de cada una de las tareas en esta ejecución? Razona tu respuesta.

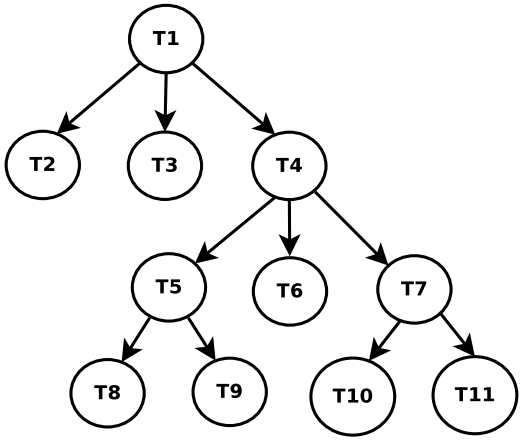


Figura 1: Grafo con el origen de las tareas para apartados 1.e, 1.f y 1.g

d) [0,5p] ¿Cuál es el speedup obtenido al ejecutar el problema de la Figura 1 con cuatro procesos? ¿Y la eficiencia paralela?

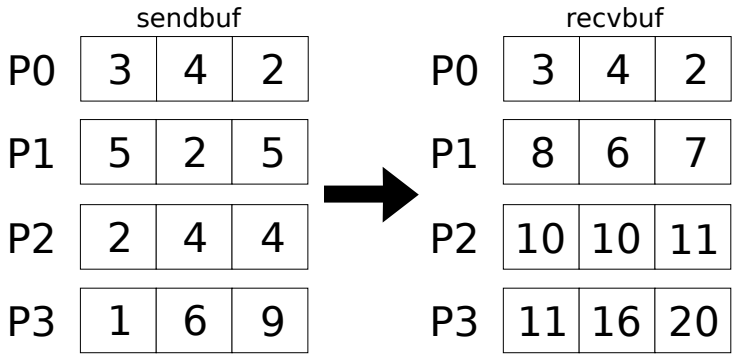
e) [0,5p] ¿Cuál es el número de procesos que deberías utilizar para maximizar la eficiencia paralela? Razona tu respuesta.

[2,5p] 2. **Paso de mensajes.** `MPI_Scan` es una operación colectiva de prefijo paralelo definida en el estándar MPI-1 que permite realizar una reducción parcial inclusiva. Es decir, cada proceso P_i calcula la reducción de los valores desde el proceso 0 hasta el i, inclusive. Esto significa que el proceso n (el último) tendrá la reducción total de todos los procesos. La firma de la operación `MPI_Scan` es la siguiente:

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm)
```

Un ejemplo del funcionamiento de `MPI_Scan` con 4 procesos para una operación de suma (`MPI_SUM`) se muestra en la siguiente figura:

MPI_Scan(sendbuf, recvbuf, 3, MPI_INT, MPI_SUM, MPI_COMM_WORLD)



a) [1p] Utilizando únicamente las primitivas MPI que aparecen en la Tabla 1, escribe una implementación en C de la operación colectiva `MPI_Scan` que soporte únicamente las operaciones de suma (`MPI_SUM`) y producto (`MPI_PROD`). Puedes asumir que los búfers de entrada (*sendbuf*) y salida (*recvbuf*) siempre contienen números enteros (`MPI_INT`).

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
int MPI_Barrier(MPI_Comm comm)
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm)
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op,
                  MPI_Comm comm)
```

Tabla 1: Primitivas MPI (Apartado 2.a)

b) [1p] En un determinado instante de la ejecución de un programa paralelo MPI con 4 procesos, cada proceso P_i tiene dos vectores $sendbuf$ y $recvbuf$ en memoria con los valores enteros que se muestran en la Figura 2. Escribe el contenido del vector $recvbuf$ en cada proceso P_i tras aplicar cada una de las operaciones colectivas que se muestran en la Tabla 2. Ten en cuenta que cada operación se aplica sobre el resultado de aplicar la operación anterior. Contesta utilizando la hoja de respuestas adjunta. No se corregirán respuestas fuera de la misma.

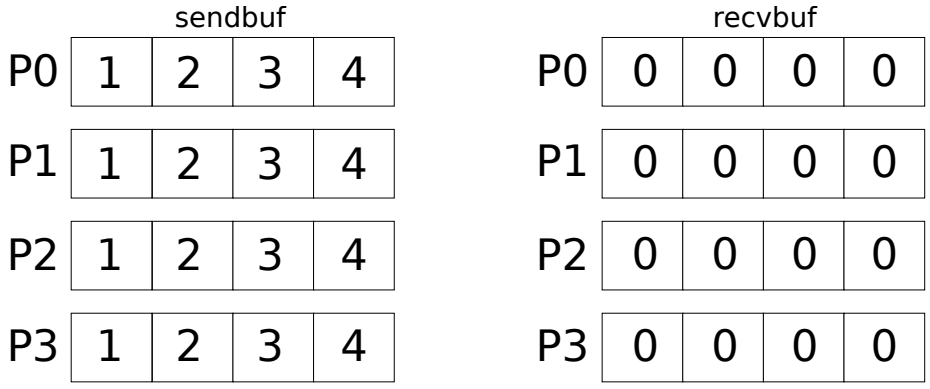


Figura 2: Contenido inicial de los vectores $sendbuf$ y $recvbuf$ (Apartado 2.b)

```
MPI_Scatter(sendbuf,1,MPI_INT,&recvbuf[2],1,MPI_INT,1,MPI_COMM_WORLD)
MPI_Gather(&sendbuf[1],1,MPI_INT,recvbuf,1,MPI_INT,2,MPI_COMM_WORLD)
MPI_Reduce(&sendbuf[2],&recvbuf[1],1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)
MPI_Scan(sendbuf,recvbuf,2,MPI_INT,MPI_PROD,MPI_COMM_WORLD)
```

Tabla 2: Colectivas MPI (Apartado 2.b)

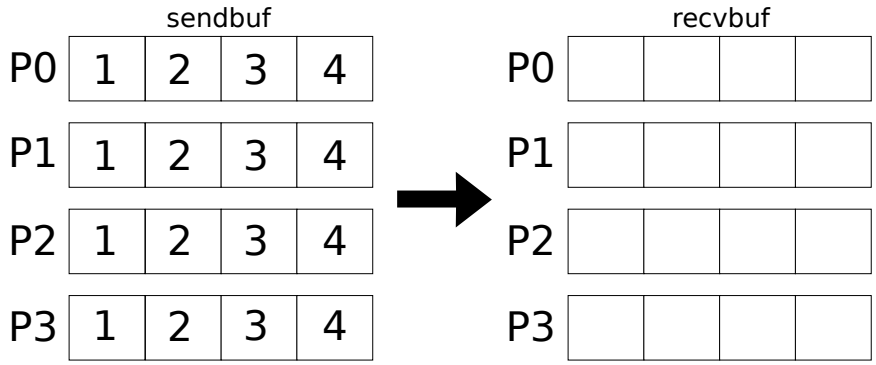
c) [0,5p] La Tabla 3 muestra un fragmento de un programa paralelo MPI que calcula la media y desviación estándar de un conjunto de números utilizando dos operaciones colectivas: `MPI_Allreduce` y `MPI_Reduce`. Completa los parámetros que faltan en ambas operaciones.

```
...
float l_sum = 0, g_sum = 0, l_sq_diff = 0, g_sq_diff = 0;

// Sum the numbers locally
for (i = 0; i < num_elements_per_proc; i++)
    l_sum += numbers[i];

// Reduce all of the local sums into the global sum in order to calculate the mean
MPI_Allreduce(&l_sum, &g_sum, ____________, ____________, MPI_SUM, MPI_COMM_WORLD);

// Calculate the mean
float mean = g_sum / (num_elements_per_proc * ntasks);

// Compute the local sum of the squared differences from the mean
for (i = 0; i < num_elements_per_proc; i++)
    l_sq_diff += (numbers[i] - mean) * (numbers[i] - mean);

// Reduce the global sum of the squared differences to the root process
MPI_Reduce(____________, &g_sq_diff, 1, ____________, MPI_SUM, ___________, MPI_COMM_WORLD);

if (my_id == 0) {
    // The standard deviation is the square root of the mean of the squared differences
    float stddev = sqrt(g_sq_diff / (num_elements_per_proc * ntasks));
    printf("Mean = %f, Standard deviation = %f\n", mean, stddev);
}
...
```

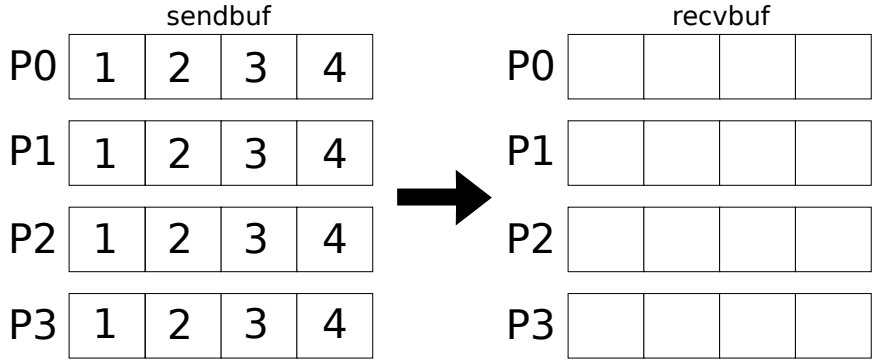
Tabla 3: Cálculo de la media y desviación estándar (Apartado 2.c)

Respuestas Apartado 2.b

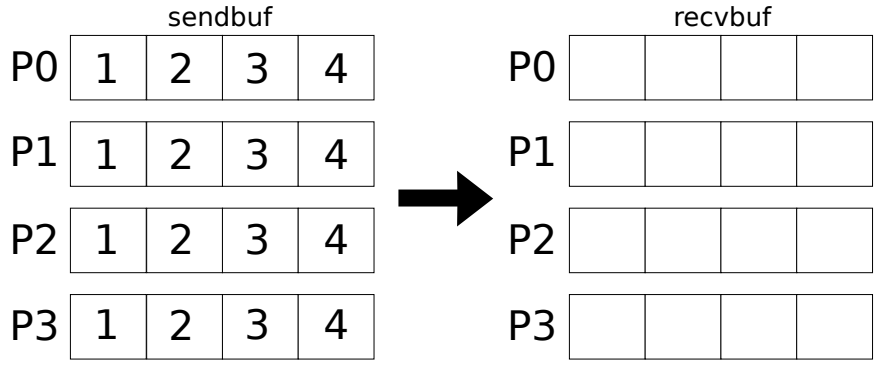
MPI_Scatter(sendbuf,1,MPI_INT,&recvbuf[2],1,MPI_INT,1,MPI_COMM_WORLD)



MPI_Gather(&sendbuf[1],1,MPI_INT,recvbuf,1,MPI_INT,2,MPI_COMM_WORLD)



MPI_Reduce(&sendbuf[2],&recvbuf[1],1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)



MPI_Scan(sendbuf,recvbuf,2,MPI_INT,MPI_PROD,MPI_COMM_WORLD)