Nombre y apellido: Enric Gil Gallen

Nombre y apellido: Victor Granados Segara

Tiempo: 1:23

Tema 11. Comunicaciones Punto a Punto en MPI

Tema 12. Comunicaciones Colectivas en MPI

1 Cada proceso dispone de un dato propio y posiblemente distinto en una variable denominada `dato` de tipo entero. Por simplicidad, todos los procesos inicializan dicha variable de la misma forma: La variable `dato` en el proceso `miId` toma el valor `numProcs - miId + 1`.

Se desea que el proceso 0 calcule y obtenga la suma de los valores almacenados en las variables `dato` de todos los procesos, incluido él mismo. Cada proceso debe imprimir su valor inicial y, además, el proceso 0 debe imprimir la suma final.

1.1) Implementa un programa en el que el proceso 0 calcule la suma mediante operaciones de comunicación punto a punto.

Comprueba que el programa funciona correctamente con varias configuraciones de procesos.

Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

```c
void main(int argc, char * argv[]){
   MPI_Status s;
   int miId, dato, numProcs, total;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &miId);

   //Eje 1.1
   dato = numProcs - miId - 1;
   printf("Proceso %d -- Dato: %d\n", miId, dato);

   if (miId == 0){
       total = dato;
       for (int i = 1; i < numProcs; i++){
           MPI_Recv(&dato, 1, MPI_INT, MPI_ANY_SOURCE,88, MPI_COMM_WORLD, &s);
           total += dato;
       }
       printf("Eje 1.1 -- Suma total: %d\n", total);
   }
   else{
       total = dato;
       MPI_Send(&dato, 1, MPI_INT, 0,88, MPI_COMM_WORLD);
   }

   MPI_Finalize();
}
```

1.2) Implementa un programa en el que el proceso 0 calcule la suma mediante operaciones de comunicación colectivas.

Comprueba que el programa funciona correctamente con varias configuraciones de procesos.

Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

```
void main(int argc, char * argv[]){
    MPI_Status s;
    int miId, dato, numProcs, total, suma;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &miId);

    //Eje 1.2
    dato = numProcs - miId - 1;
    printf("Proceso %d -- Dato: %d\n", miId, dato);

    MPI_Reduce(&dato, &suma,1, MPI_INT, MPI_SUM,0, MPI_COMM_WORLD);

    if (miId == 0){
        printf(" Eje 1.2 -- Suma total: %d\n", suma);
    }

    MPI_Finalize();
}
```

2 Cada proceso dispone de un dato propio y posiblemente distinto en una variable denominada `dato` de tipo entero. Por simplicidad, todos los procesos inicializan dicha variable de la misma forma: La variable `dato` en el proceso `miId` toma el valor `numProcs - miId + 1`.

Se desea que el proceso 0 calcule y obtenga la suma de los valores almacenados en las variables `dato` de únicamente los **procesos pares**, incluido él mismo. El valor de `dato` de los procesos impares no deben reflejarse en la suma. Cada proceso debe imprimir su valor inicial y, además, el proceso 0 debe imprimir la suma final.

2.1) Implementa un programa en el que el proceso 0 calcule la suma mediante operaciones de comunicación punto a punto, en la **únicamente participen los procesos pares**.
Comprueba que el programa funciona correctamente con varias configuraciones de procesos.
Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

```
void main(int argc, char * argv[]){
    MPI_Status s;
    int miId, dato, numProcs, total, dato_emisor;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &miId);

    //Eje 2.1
    dato = numProcs - miId + 1;
    printf("Proceso %d -- Dato: %d\n", miId, dato);

    if (miId == 0){
        total = dato;
        for (int i = 2; i < numProcs; i+=2){
            MPI_Recv(&dato, 1, MPI_INT, MPI_ANY_SOURCE,88, MPI_COMM_WORLD, &s);
            total += dato;
        }
        printf("Eje 2.1 -- Suma total: %d\n", total);
    }
    else{
        if (miId % 2 == 0){
            MPI_Send(&dato, 1, MPI_INT, 0,88, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
}
```

2.2) Implementa un programa en el que el proceso 0 calcule la suma mediante operaciones de comunicación punto a punto, en la**únicamente participen los procesos pares**, y en el que cada proceso, como máximo, **reciba y envíe un único mensaje**.
Comprueba que el programa funciona correctamente con varias configuraciones de procesos.
Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

```
void main(int argc, char * argv[]){
    MPI_Status s;
    int miId, dato, numProcs, total, dato_emisor;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &miId);

    // Comprobacion del numero de procesos.
    if( numProcs < 2 ) {
        if ( miId == 0 ) {
            fprintf( stderr, "\nError: Al menos se deben iniciar dos procesos\n\n" );
        }
        MPI_Finalize();
    }

    //Eje 2.2
    dato = numProcs - miId + 1;
    printf("Proceso %d -- Dato: %d\n", miId, dato);

    if(miId == 0){
        MPI_Send(&dato, 1, MPI_INT, 1, 88, MPI_COMM_WORLD);
        MPI_Recv(&dato, 1, MPI_INT, numProcs-1,88, MPI_COMM_WORLD, &s);
        printf("Eje 2.2 -- Suma total: %d\n", dato);
    }
    else{
        if (miId % 2 != 0) {
            dato = 0;
        }

        total = dato;
        MPI_Recv(&dato, 1, MPI_INT, miId-1,88, MPI_COMM_WORLD, &s);

        total += dato;
        dato = total;

        if(miId != numProcs-1){
            MPI_Send(&dato, 1, MPI_INT, miId+1, 88, MPI_COMM_WORLD);
        }
        else{
            MPI_Send(&dato, 1, MPI_INT, 0, 88, MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();
}
```

2.3) Implementa un programa en el que el proceso 0 calcule la suma mediante operaciones de comunicación colectivas.

Es obligatorio que todos los procesos participen en una operación de comunicación colectiva, ya que, en caso contrario, el programa se bloquea, pero en este caso se desea que **los valores de los procesos impares no se sumen**.

La solución más sencilla a este problema se consigue utilizando una variable auxiliar para realizar la operación que se inicializa convenientemente: Los procesos pares con el valor de su variable `dato`, mientras que los procesos impares la inicializan a cero.

Comprueba que el programa funciona correctamente con varias configuraciones de procesos.

Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, las comunicaciones y la impresión de resultados.

```
void main(int argc, char * argv[]){
    MPI_Status s;
    int miId, dato, numProcs, total, suma;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &miId);

    //Eje 2.3
    dato = numProcs - miId + 1;
    printf("Proceso %d -- Dato: %d\n", miId, dato);

    total = (miId % 2 == 0 ? dato : 0);
    MPI_Reduce(&total,&suma, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (miId == 0){
        printf("Eje 2.3 -- Suma total: %d\n", suma);
    }
    MPI_Finalize();
}
```

3 Se desea implementar un programa en el que todos los procesos colaboren para calcular la suma de los elementos de un vector de números reales de doble precisión que aparece en el proceso 0. Para ello, en primer lugar el vector se debe distribuir entre los procesos, tras lo cual cada proceso calcula una suma local. Finalmente las sumas locales se acumulan sobre el proceso 0.

El vector a distribuir se denomina `vectorInicial` y su dimensión se guarda en la variable `dimVectorInicial`, obtenido como parámetro de entrada del programa. Por simplicidad, sólo se consideran tamaños de vector inicial que sea divisibles por el número de procesos. Antes de proceder al reparto, el proceso 0 inicializa el vector inicial, y calcula la suma de sus componentes en `sumaInicial`.

En el reparto de `vectorInicial` se utiliza una distribución por bloques, en la que el proceso 0 también se queda con un bloque de datos. Todos los procesos, deben guardar sus respectivos datos en un vector denominado `vectorLocal`, cuya dimensión se almacena en la variable `dimVectorLocal`.

La suma de los elementos de `vectorLocal` que realiza cada proceso se almacena sobre `sumaLocal`, mientras que la acumulación de estos valores debe quedar en `sumaFinal` del proceso 0.

Como punto de partida puedes tomar el siguiente código, el cual deberás compilar incluyendo al final la opción **-lm**, que informa al enlazador que tiene que incorporar la librería matemática:

3.1) En este apartado debes calcular el valor de la variable `dimVectorLocal`, y realizar el reparto del vector `VectorInicial`, utilizando únicamente envíos y recepciones **punto a punto**.

Además debes incluir las órdenes que permiten calcular el coste de la ejecución secuencial (tSec) y las que permiten calcular el coste de la distribución de los datos (tDis).

Estas líneas se deben insertar a continuación de las líneas marcadas con "(A)-(D)".

Escribe a continuación la parte del programa principal que realiza estas tareas: la inicialización de variables y las comunicaciones.

```
   // Inicio del calculo de la reduccion en secuencial
   // t1 = ... ; // ... (A)
   t1 = MPI_Wtime();
   sumaInicial = 0;
   for (i = 0; i < dimVectorInicial; i++) {
       sumaInicial += evaluaFuncion(vectorInicial[i]);
   }
   // Finalizacion de la reduccion y calculo de su coste
   t2 = MPI_Wtime(); // ... (B)
   tSec = t2 - t1;
}

// Todos los procesos crean e inicializan "vectorLocal".
// La siguiente linea no es correcta. Debes arreglarla.
// dimVectorLocal = ... ; // ... (C)
vectorLocal = (double *) malloc(dimVectorLocal * sizeof(double));
for (i = 0; i < dimVectorLocal; i++) {
   vectorLocal[i] = -1.0;
}

MPI_Barrier(MPI_COMM_WORLD);
// Distribucion por bloques de "vectorInicial" y calculo de su coste (tDis).
// Al final de esta fase, cada proceso debe tener sus correspondientes datos
// propios en su "vectorLocal".
// ... (D)
```

```
t1 = MPI_Wtime();

int ini, fin, tam;
dimVectorLocal = dimVectorInicial / numProcs;
ini = dimVectorLocal * miId;
fin = ini + dimVectorLocal;

if (miId == 0) {
   for (i = ini; i < fin; i++) {
       vectorLocal[i] = vectorInicial[i];
   }
   for (i = 1; i < numProcs; i++) {
       MPI_Send(&vectorInicial[dimVectorLocal * i], dimVectorLocal, MPI_DOUBLE, i, 88,
MPI_COMM_WORLD);
   }
} else {
   MPI_Recv(vectorLocal, dimVectorLocal, MPI_DOUBLE, 0, 88, MPI_COMM_WORLD, &s);
}

t2 = MPI_Wtime();
tDis = t1 - t2;
```

3.2) Haz una copia del anterior programa y denomínala `vector_2_2`.

Modifica el programa para que, después de recibir los datos, todos los procesos sumen sus valores locales sobre `sumaLocal`. A continuación, todos los procesos deben enviar el resultado de la suma local al proceso 0, el cual debe acumular todos los resultados parciales recibidos a su suma local para obtener `sumaFinal`.

Además debes incluir las órdenes que permiten calcular el coste de la ejecución paralela (tPar), así como la impresión de los costes y los resultados.

Comprueba el resultado de la suma paralela es correcta, comparando el valor de las variables `sumaInicial` y `sumaFinal`.

En este apartado sólo deben emplearse comunicaciones **punto a punto**.

Estas líneas se deben insertar a continuación de las líneas marcadas con "(E)-(I)".

Escribe a continuación la parte del programa principal que realiza tal tarea: la acumulación de resultados, las comunicaciones y la impresión de resultados.

```
MPI_Barrier(MPI_COMM_WORLD);
// Inicio del calculo de la reduccion en paralelo y su coste (tPar).
// ... (E)
t1 = MPI_Wtime();

// Cada proceso suma todos los elementos de vectorLocal
// ... (F)
sumaLocal = 0;
for (i = 0; i < dimVectorLocal; i++) {
   sumaLocal += evaluaFuncion(vectorLocal[i]);
}

// Se acumulan las sumas locales de cada procesador en sumaFinal sobre el proceso 0
// ... (G)
if (miId == 0) {
   sumaFinal = 0;
sumaRecibida = 0;
   sumaFinal += sumaLocal;
   for (i = 1; i < numProcs; i++) {
       MPI_Recv(&sumaRecibida, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 88, MPI_COMM_WORLD, &s);
       sumaFinal += sumaRecibida;
   }
} else {
   MPI_Send(&sumaLocal, 1, MPI_DOUBLE, 0, 88, MPI_COMM_WORLD);
}

// Finalizacion del calculo de la reduccion en paralelo y su coste (tPar).
// ... (H)
t2 = MPI_Wtime();
tPar = t2 - t1;

// El proceso 0 imprime la sumas, los costes y los incrementos

if (miId == 0) {
   // Imprimir Sumas(sumaInicial, sumaFinal, diferencia)
   printf("Proc: %d , sumaInicial = %lf , sumaFinal = %lf , diff = %lf\n",
          miId, sumaInicial, sumaFinal, sumaInicial - sumaFinal);
   // Imprimir Costes(tSec, tPar, tDis)
   // Imprimir Incrementos(tSec vs tPar , tSec vs (tDis+tPar) )
   // ... (I)
```

```
    printf("Tº Secuencial: %f", tSec);
    printf("Tº Paralelo: %f",tPar);
    printf("Tº Distribución por Bloques: %f",tDis);
    printf("Tº Incremento tSec vs tPar: %f",tSec/tPar);
    printf("Tº Incremento tSec vs (tDis+tPar): %f",tSec/(tPar+tDis));
}
```

3.3) Haz una copia del anterior programa y denomínala `vector_2_3`.
Modifica el programa del ejercicio anterior para que tanto el **reparto** de los elementos del vector como la **recogida** de las sumas parciales se realicen utilizando operaciones de **comunicación colectiva**. En este ejercicio **no se pueden emplear operaciones de reducción**.
Estas líneas se deben insertar a continuación de las líneas marcadas con "(D)" y "(G)".
Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables, el reparto de los datos, la reunión de los resultados y la acumulación final.

```
    // ... (D)
    t1 = MPI_Wtime();

    int ini, fin, tam;
    dimVectorLocal = dimVectorInicial / numProcs;
    ini = dimVectorLocal * miId;
    fin = ini + dimVectorLocal;

    MPI_Scatter(&vectorInicial[inicio], dimVectorLocal, MPI_DOUBLE, vectorLocal,
dimVectorLocal, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    t2 = MPI_Wtime();
    tDis = t1 - t2;

#ifdef IMPRIME
    // Todos los procesos imprimen su vector local.
    for( i = 0; i < dimVectorLocal; i++ ) {
      printf( "Proc: %d.  vectorLocal[ %3d ] = %lf\n",
              miId, i, vectorLocal[ i ] );
    }
#endif

    MPI_Barrier(MPI_COMM_WORLD);
    // Inicio del calculo de la reduccion en paralelo y su coste (tPar).
    // ... (E)
    t1 = MPI_Wtime();

    // Cada proceso suma todos los elementos de vectorLocal
    // ... (F)
    sumaLocal = 0;
    for (i = 0; i < dimVectorLocal; i++) {
        sumaLocal += evaluaFuncion(vectorLocal[i]);
    }

    // Se acumulan las sumas locales de cada procesador en sumaFinal sobre el proceso 0
    // ... (G)
    sumaFinal = 0;
    double vectorGather[numProcs];
    MPI_Gather(&sumaLocal, 1, MPI_DOUBLE, vectorGather, 1, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    for (i = 0; i < numProcs; i++) {
        sumaFinal += vectorGather[i];
    }
```

3.4) Haz una copia del anterior programa y denomínala `vector_2_4`.
Modifica el programa del ejercicio anterior para que la suma de los resultados parciales se realice con una operación de **comunicación colectiva de reducción**. El resultado debe quedar en el proceso 0.
Estas líneas se deben insertar a continuación de la línea marcada con "(G)".
Escribe a continuación la parte del programa principal que realiza tal tarea: la declaración de variables y la acumulación final.

```
// Se acumulan las sumas locales de cada procesador en sumaFinal sobre el proceso 0
// ... (G)
sumaFinal = 0;
MPI_Reduce(&sumaLocal, &sumaFinal, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

3.5) Quita el comentario para activa `COSTOSA` y evalúa el programa `vector_2_2` en la cola de patan, y completa la tabla con 4 decimales.

$n = 1\ 200\ 000$				
Número de procesos	2	4	6	8
Coste Secuencial (tSec)				
Coste Paralelo (tPar)				
Incremento Paralelo (tSec vs tPar)				
Coste Distribución (tDis)				
Incremento Global (tSec vs (tPar + tDis))				

Examina con detalle los valores y justifica los resultados.

n = 1 200 000				
Número de procesos	**2**	**4**	**6**	**8**
Coste Secuencial (tSec)	0.156	0.1558	0.156	0.156
Coste Paralelo(tPar)	0.078	0.393	0.026	0.019
Incremento Paralelo (tSec vs tPar)	1.993	3.960	5.973	7.797
Coste Distribución (tDis)	0.043	0.062	0.068	0.074
Incremento Global	4.435	6.806	3.676	2.897

Ocurre algo curioso, observamos una relación indirectamente proporcional entre el tiempo en Paralelo el cual se beneficia de tener la mayor cantidad de procesos disponibles puesto que hay menos carga de trabajo por proceso, y entre el tiempo de Distribución que a más procesos más perdida en compartición de mensajes. Aunque como queda demostrado por obtener los mejores tiempos es más eficiente el trabajo Paralelo con 8 hebras.

3.6) Quita el comentario para activa `COSTOSA` y evalúa el programa `vector_2_3` en la cola de patan, y completa la tabla con 4 decimales.

$n = 1\ 200\ 000$				
Número de procesos	2	4	6	8
Coste Secuencial (tSec)				
Coste Paralelo (tPar)				
Incremento Paralelo (tSec vs tPar)				
Coste Distribución (tDis)				
Incremento Global (tSec vs (tPar + tDis))				

Examina con detalle los valores y justifica los resultados.

n = 1 200 000				
Número de procesos	**2**	**4**	**6**	**8**
Coste Secuencial (tSec)	0.155	0.155	0.154	0.158
Coste Paralelo(tPar)	0.087	0.040	0.026	0.019
Incremento Paralelo (tSec vs tPar)	1.773	3.854	5.882	7.92
Coste Distribución (tDis)	0.043	0.063	0.068	0.072
Incremento Global	3.492	6.857	3.674	3.014

Ocurre algo curioso, observamos una relación indirectamente proporcional entre el tiempo en Paralelo el cual se beneficia de tener la mayor cantidad de procesos disponibles puesto que hay menos carga de trabajo por proceso, y entre el tiempo de Distribución que a más procesos más perdida en compartición de mensajes. Aunque como queda demostrado por obtener los mejores tiempos es más eficiente el trabajo Paralelo con 8 hebras.

3.7) Quita el comentario para activa `COSTOSA` y evalúa el programa `vector_2_4` en la cola de patan, y completa la tabla con 4 decimales.

$n = 1\ 200\ 000$				
Número de procesos	2	4	6	8
Coste Secuencial (tSec)				
Coste Paralelo (tPar)				
Incremento Paralelo (tSec vs tPar)				
Coste Distribución (tDis)				
Incremento Global (tSec vs (tPar + tDis))				

Examina con detalle los valores y justifica los resultados.

n = 1 200 000				
Número de procesos	**2**	**4**	**6**	**8**
Coste Secuencial (tSec)	0.153	0.155	0.156	0.155
Coste Paralelo(tPar)	0.077	0.040	0.026	0.019
Incremento Paralelo (tSec vs tPar)	1.976	3.804	5.887	7.935
Coste Distribución (tDis)	0.043	0.061	0.068	0.072
Incremento Global	4.437	7.618	3.732	2.917

Ocurre algo curioso, observamos una relación indirectamente proporcional entre el tiempo en Paralelo el cual se beneficia de tener la mayor cantidad de procesos disponibles puesto que hay menos carga de trabajo por proceso, y entre el tiempo de Distribución que a más procesos más perdida en compartición de mensajes. Aunque como queda demostrado por obtener los mejores tiempos es más eficiente el trabajo Paralelo con 8 hebras.

3.8) Comparando el tiempo de implementación de cada apartado y las tablas de resultados, responde a las siguientes preguntas referidas a transferencias en las que estén involucros todos los procesos de un programa:
¿Es más sencillo utilizar comunicaciones punto a punto o comunicaciones colectivas?
¿Es más eficiente utilizar comunicaciones punto a punto o comunicaciones colectivas?

- Como sencillo considero que es más sencillo las comunicaciones colectivas.
- Cómo eficiente ha quedado demostrado que es mejor la comunicación punto a punto cuando se dispone de muchos procesos, en caso de no poder disponer de tantos es menos las comunicaciones colectivas.