Puzle, Parte 1

Jose Ángel Gumiel, Mikel Dalmau y Christian Merino

5 de Abril de 2016

Índice

1 Comunicaciones colectivas

En la comunicación entre un grupo de individuos, existen mecanismos que permiten una comunicación más eficiente. En este trabajo en el apartado 1.1 *Comunicación en forma de árbol* se muestran varios modelos de comunicación compuestos por árboles.

En MPI, la comunicación colectiva consiste en una serie de funciones que sirven para que un grupo de procesos se comuniquen entre ellos. En el apartado 1.2 *Comunicaciones colectivas frente a comunicaciones Punto a Punto* se muestran las diferencias principales entre las funciones de comunicación colectiva y las de punto a punto. En el apartado 1.3 *Funciones de Comunicación Colectiva en MPI* se describen las funciones más utilizadas y sus parámetros.

1.1 Comunicación en forma de árbol

En [1] se describe esta forma de comunicación, mostrando como ejemplo un problema de suma *one-to-all*. Aunque inicialmente pueda parecer que no mejora demasiado, ya que la mitad de los nodos realizan las mismas comunicaciones que realizarían punto a punto(esto es, cuando todos los nodos envían su valor a un único nodo), mejora considerablemente reduciendo la cantidad de recepciones y sumas que tiene que realizar el nodo líder.

En el ejemplo de la imagen, el nodo 0 pasa de realizar *n-1* recepciones y sumas a realizar 3 recepciones y sumas, esto es, la altura del árbol $log_2\ n(log_d\ n$ en el caso general, siendo d el grado de árbol). De esta forma la carga de trabajo de los nodos crece logarítmica-mente con el número de procesadores, la mejora es notable frente al crecimiento lineal de la comunicación centralizada.

Año: 2015 - 2016
Profesor(a):
Teléfono:
E-mail: mdalmau002@ikasle.ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

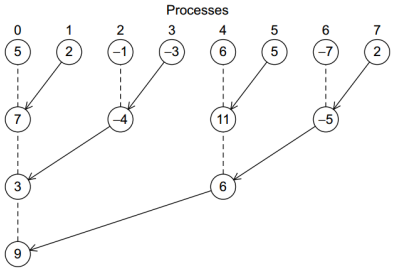


FIGURE 3.6
A tree-structured global sum

El mismo modelo invertido se puede utilizar para comunicación *one-to-all*.

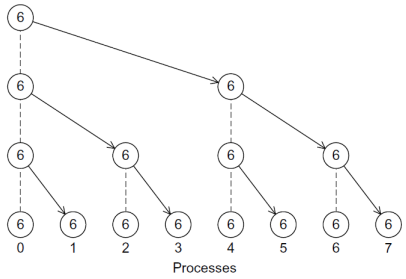


FIGURE 3.10
A tree-structured broadcast

Este último ejemplo muestra un modelo de comunicación all-to-all en el que se construyen n árboles siendo cada nodo raíz de uno de ellos.

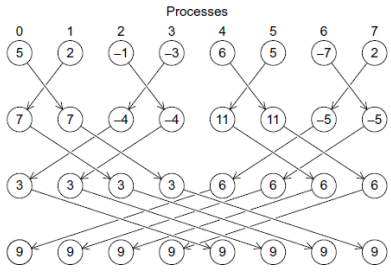


FIGURE 3.9
A butterfly-structured global sum

El problema de este modelo de comunicación es la dificultad de programarlo, existen incontables maneras distintas de hacerlo pero no sabríamos cual es la mejor, para que tipos y tamaños de problemas cual es la óptima. En [2] se sugiere también el uso de árboles de profundidad logarítmica para la comunicación, pero tampoco indican o desconocen si las funciones MPI hacen uso de ellos.

1.2 Comunicaciones colectivas frente a comunicaciones Punto a Punto

1. Todos los procesos en el comunicador deben llamar a la misma función colectiva independientemente del tipo de comunicación que se realice. Mientras en la comunicación punto a punto queda definido por la llamada quien es emisor y quien es receptor, MPI_Recv y MPI_Send.

2. Todos los procesos en el comunicador tienen que tener los parámetros compatibles, esto es, si en una llamada MPI_Reduce dos procesos tiene *root* distinto el programa va a fallar.

3. Las funciones de comunicación punto a punto se conectan mediante etiquetas (*tags*) y comunicadores, en las colectivas solamente mediante comunicadores y el orden en el que son llamadas.

4. Otra restricción que las funciones de comunicación punto a punto no tienen es que la cantidad de datos enviados en el bufer tiene que coincidir exactamente con la esperada en el destino.

5. No exiten funciones de comunicación colectivas que no sean bloqueante, todas utilizan la función Barrier para asegurar las sincronización de los datos.

1.3 Funciones de Comunicación Colectiva en MPI

Podemos dividir las funciones de comunicación colectiva en función del tipo de comunicación que realizan y también en función del tipo de dato que manejan ya que así las distingue MPI.

- *all-to-one*:
  - MPI_REDUCE
  - MPI_GATHER
- *one-to-all*
  - MPI_BCAST
  - MPI_SCATTER
- *all-to-all*
  - MPI_ALLREDUCE
  - MPI_ALLGATHER
  - MPI_ALLTOALL

- MPI_REDUCE_SCATTER

- *all-to-some*

  - MPI_SCAN

Respecto al tipo de dato que manejan, MPI dispone de las *vector variant* de las funciones vistas; MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV.

Estas funciones permiten enviar una serie de datos de tamaño variable, se diferencian por tener un argumento extra llamado displs, que es un array de enteros que indica las posiciones de cada dato.

Respecto a las funciones ALL, en estas los parámetros son idénticos que los de sus semejantes con la excepción de que sobra el parámetro *root* ya que todos recibirán el mensaje.

La cabecera de una función colectiva puede tener los siguientes parámetros:

- buffer : Contiene la dirección de comienzo del buffer.
- count : Número de entradas en el buffer.
- datatype : Tipo de datos del buffer.
- root : Identificador del proceso raíz.
- comm : Comunicador MPI, representa a los procesos involucrados en la comunicación.

Las funciones Scatter y Gather utilizan dos buffers distintos.

- send buffer : Contiene la dirección de comienzo del buffer a enviar.
- recv buffer : Contiene la dirección de comienzo del buffer a recibir.
- send count : Número de entradas en send buffer.
- recv count : Número de entradas en recv buffer.
- displs : Vector de enteros con las posiciones de cada dato en el buffer.

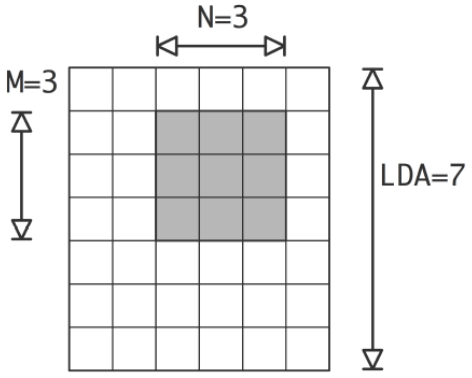
Es de especial interés el parámetro *op* de Reduce, que define el tipo de operación a realizar. Existen los siguientes tipos de operaciones:

Operation Value	Meaning
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
MPI_LAND	And lógico
MPI_BAND	And binario
MPI_LOR	Or lógico
MPI_BOR	Or binario
MPI_LXOR	Or exclusivo lógico
MPI_BXOR	Or exclusivo binario
MPI_MAXLOC	Máximo y su dirección
MPI_MINLOC	Mínimo y su dirección

2 Tipos de datos derivados y comunicadores

Hasta ahora, antes de conocer el paralelismo de datos, se ha trabajado con tipos de datos primitivos (int, double, char, etc.). Estos son contiguos en memoria.

MPI ofrece un nuevo paradigma. Es posible que el programador desee enviar datos de carácter heterogéneo o no contiguos en memoria, MPI permite realizar este tipo de transferencias. Se puede ver en el siguiente ejemplo:



Las filas de una matriz no son contiguas en memoria, están separadas por un stride igual al número de columnas. Es decir, la segunda fila está en este caso a un stride 6 de la primera fila.

Si se desea acceder a los datos de la matriz coloreados en gris se puede hacer de una manera eficiente un nuevo tipo de dato. Los objetos irregulares son conocidos como tipos de datos derivados.

2.1 Tipos de datos elementales

MPI tiene un número de tipos de datos elementales, que se corresponden a los tipos simples de los lenguajes de programación. Los nombres se asemejan a los de C y Fortran. Así tenemos los tipos MPI_FLOAT y MPI_DOUBLE, y por otro lado los tipos MPI_REAL y MPI_DOUBLE_PRECISION. Hay que respetar su uso, no se puede usar MPI_FLOAT si programamos en Fortran ni MPI_REAL si trabajamos en C.

Las llamadas de MPI aceptan arrays de elementos, para enviar un único elemento hay que apuntar a su dirección. Hay dos problemas al usar únicamente tipos de datos elementales:

- Las rutinas de comunicación de MPI sólo pueden enviar elementos de un único tipo, homogéneos, aunque estén en posiciones de memoria contiguas. Se puede usar MPI_BYTE, pero no es recomendable.
- Tampoco es posible enviar elementos del mismo tipo si no están contiguos en memoria. Se puede enviar memoria contigua que contenga esos elementos (además de otros), pero supondría un gasto innecesario de ancho de banda.

2.2 Tipos de datos derivados

MPI permite crear tipos de datos propios, algo similar a definir una estructura en un lenguaje de programación, pero no del todo. Son especialmente útiles cuando se trata de enviar múltiples elementos en un mismo mensaje.

Los tipos de datos derivados permiten solucionar los problemas anteriormente comentados de diferentes formas:

1. Se puede crear un tipo de dato contiguo que consiste en un array de elementos de distintos tipos de datos. No hay diferencia entre enviar un elemento de un tipo o múltiples de distinto tipo.
2. Se puede crear un tipo de datos vector que consista en bloques espaciados de forma regular de elementos de un mismo tipo. Es una solución al problema de no poder enviar datos no contiguos.
3. Para los datos no contiguos espaciados de forma irregular existe el tipo de dato indexado. Se trata de un array que contiene las ubicaciones de los bloques. Los bloques pueden ser de distinto tamaño.
4. El tipo de datos struct puede albergar múltiples tipos de datos.

Todos estos mecanismos se pueden combinar para obtener tipos datos de tipo heterogéneo espaciados de forma irregular.

2.2.1 Datatype signatures

Con los tipos de datos primitivos, si el emisor envía un array de enteros, el receptor tiene que declarar el tipo de dato también como enteros. Con los tipos de datos derivados ya no ocurre esto. El emisor y el receptor pueden declarar diferentes tipos de datos siempre y cuando tengan el mismo "datatype signature".

La firma del tipo de dato es la representación interna del tipo de dato. Por ejemplo, si el emisor declara un tipo consistente en 2 enteros y envía 4 elementos de ese tipo, el receptor puede recibir dos elementos de un tipo que consista en 4 enteros. En ambos casos se envían 8 enteros y se reciben 8 enteros.

2.2.2 Basic calls

Los nuevos tipos de datos se crean con:

- MPI_Type_contiguous
- MPI_Type_create_subarray
- MPI_Typevector
- MPI_Type_struct
- MPI_Type_indexed
- MPI_Type_hindexed

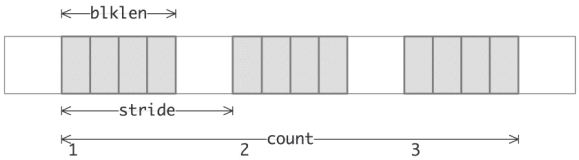
Es necesario llamar a MPI_Type_commit, el cual se encarga de que MPI haga los cálculos de indexación para los tipos de datos. Cuando no se necesite ya más hay que llamar a MPI_Type_free.

2.2.3 Tipo contiguo

Es el tipo derivado más simple. Define un array de elementos de un tipo elemental o de otro tipo definido con anterioridad. No hay diferencia entre enviar un elemento de tipo contiguo o múltiples elementos del tipo que lo componen.

2.2.4 Tipo vector

Es el tipo de dato no contiguo más simple. Describe una serie de bloques, todos de la misma longitud, y espaciados con un mismo stride, constante.

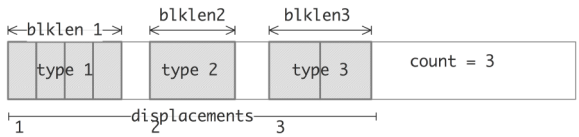


2.2.5 Tipo indexado

El tipo indexado puede enviar elementos ubicados arbitrariamente de un array de un tipo único. Para ello hay que proporcionar un array de posiciones, acompañado de un array de longitudes con un array separado con el tamaño de cada bloque.

2.2.6 Tipo struct

El tipo struct puede contener múltiples tipos de datos. La especificación contiene un contador que indica cuantos bloques hay en una única estructura.

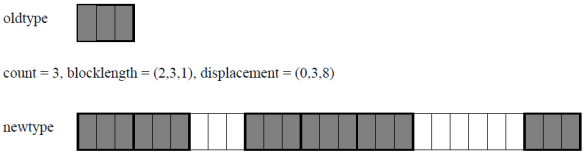


El tipo de estructura es muy similar en funcionalidad al tipo MPI_Type_hindexed, que usa un indexado basado en el Byte. El uso del tipo struct probablemente sea más limpio.

2.2.7 Empaquetado

Una de las razones para usar tipos de datos derivados es poder tratar con datos no contiguos. Anteriormente esto sólo se podía hacer empaquetando los datos de su contenedor original en un buffer y desempaquetándolo en su receptor en sus estructuras de datos de destino.

MPI ofrece esa opción de empaquetado, parcialmente por la compatibilidad entre librerías, pero también por flexibilidad. A diferencia de los tipos de datos derivados, que transfieren los datos automáticamente, las rutinas de empaquetado añaden datos de forma secuencial en un buffer, y el empaquetado hay que hacerlo en orden.



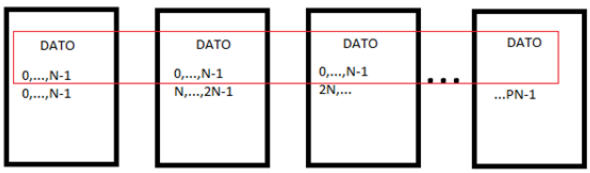
3 Reparto dinámico de carga

Una de las razones para utilizar MPI es por la necesidad de trabajar en más de un computador para poder utilizar toda la memoria que estos ofrecen. Con la memoria distribuida, cada procesador contendrá parte de la estructura de datos y solo se podrá trabajar en ese computador con dicha estructura.

Imaginemos que tenemos un gran array distribuido por distintos procesadores. Diremos que tenemos p procesos y n elementos por procesador, en total tenemos $n * p$ elementos.

Int n;

double DATO[n];



Comúnmente se suele decir que DATO es la parte local de un array distribuido con un tamaño total de $n * p$ elementos. Sin embargo, ese array solo existe de manera conceptual ya que cada procesador tiene un array con índice cero y eres tú el que tiene que trasladar dichos arrays al array global que forman. En otras palabras, tú tienes que gestionar mediante código el tratamiento del array de grandes dimensiones.

3.1 Intercambio de información local

A veces hablamos de que cada procesador posee ciertos datos y puede trabajar con el valor de los mismos. El problema que se forma entonces es cuando un computador posee el valor p pero no el valor p+1. MPI nos ofrece dos métodos para gestionar dicho problema:

- Un procesador envía algo a otro procesador.
- Un procesador recibe algo de un origen.

3.2 Ejemplo de envío: Ping-Pong

Un procesador A envía algo a un procesador B y este se lo devuelve, el código que se ejecuta en A sería el siguiente:

```
MPI_Send( /* to: */ B     .);
MPI_Recv(/* from: */ B     .);
```

Mientras que en B se ejecutaría el siguiente:

```
MPI_Recv(/* from: */ A     .);
MPI_Send( /* to: */ A     .);
```

Si estamos trabajando en el modo SPMD el programa se mostraría así:

```
if ( /* I am process A */ ) {
    MPI_Send( /* to: */ B     .);
    MPI_Recv(/* from: */ B     .);
}
else if ( /* I am process B */ ) {
    MPI_Recv (/* from: */ A     .);
    MPI_Send ( /* to: */ A     .);
}
```

3.3 Comunicación bloqueante

El uso de MPI_Send y MPI_Recv bloqueará durante unos momentos la comunicación. Recv bloqueará al computador hasta recibir el dato deseado del MPI_Send.

El problema que se puede dar en la comunicación bloqueante es el interbloqueo. Cuando dos computadores esperan que el otro les envíe algo pero jamás llega ya que el otro está bloqueado también en el MPI_Recv.

Una manera de evitar los interbloqueos es crear un grafo usando los vértices como procesadores y las aristas como las comunicaciones de entrada y salida.

Otro posible caso es que un procesador necesita un dato de su predecesor y ha de enviar un dato a su sucesor, en este caso el algoritmo a utilizar será el siguiente:

```
Sucesor = mythid+1;
Predecesor = mythid +1;
If ( /* I am not the first processor */ )
    Send (target=successor);
If ( /* I am not the first processor */ )
    Receive (source=predecesor)
```

Bibliografía

[1] Pacheco P.:*An Introduction to Parallel Programming*. Morgan Kaufmann,2011. Capítulo 3, apartado 4.

[2] Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J. *MPI: The Complete Reference, Volume 1, The MPI Core*. The MIT PRess, 1999. Capítulo 4.

[3] Victor Eijkhout, *Parallel Computing for Science and Engineering*, 1st Edition 2015.

4 Ejercicios

4.1 P1.1

Hay que repartir un vector de N elementos entre npr procesos. Completa el programa serie *P11-distribute0.c*, para que genere el tamaño de cada trozo del vector y el desplazamiento desde el origen del vector al comienzo de cada trozo, en estos dos casos:

a. los posibles restos se añaden al último trozo

b. los posibles restos se añaden uno a uno a diferentes trozos

---

a. Inicialmente calculo Nloc y remainder.

```
//Compute Nloc and remainder
Nloc = floor((double)N/(double)npr);
remainder = N - Nloc*npr;
```

Este caso es sencillo y se resuelve con el siguiente bucle y las asignaciones finales.

```
//We distribute the work among the
    procesesses
for(i=0; i<npr-1; i++){
    size[i] = Nloc;
    shift[i] = i*Nloc;
}
//Finally we charge the last process with
    the remainder
size[npr-1] = Nloc + remainder;
shift[npr-1] = (npr-1)*Nloc;
```

b. En este segundo caso he comenzado distribuyendo la carga del resto entre los primeros procesadores, luego, las cargas distintas entre procesos no permiten el cálculo de shift usado anteriormente, por lo que tomo las referencias de tamaño y shift calculadas en la anterior iteración para calcular el shift, sabemos donde empezamos porque sabemos dónde termina el anterior.

```
//Value asignment to first process
size2[0] = Nloc;
shift2[0] = 0;

//Distribution of the remainder among the
    first processes
i = 0;
while(remainder){
    size2[i] += 1;
    remainder -= 1;
    i++;
}

//Distribute the rest of the vector among
    the processes
for(i=1; i<npr; i++){
    size2[i] += Nloc;
    shift2[i] = shift2[(i-1)] + size2[(i-1)
        ];
}
```

4.2 P1.2

El programa *P12-inteser.c* calcula el valor de una integral mediante el conocido método de sumar las áreas de n trapecios bajo la curva que representa una función. A mayor valor de n, más preciso el resultado.

Completa el programa MPI *P12-inteser.c* para realizar esa misma función entre P procesos, utilizando funciones de comunicación colectiva. Compara el resultado con el de la versión serie.

Para resolver el problema es necesario modificar la función Read_data, encargada de leer por pantalla los límites superior, inferior y número de evaluaciones de la función. Dado que todos los nodos no pueden utilizar la entrada estándar al mismo tiempo, limito la lectura a un único nodo y luego distribuyo los datos leídos entre el resto de procesos utilizando la función **MPI_Bcast**.

```
void Read_data(double* a_ptr, double* b_ptr,
     int* n_ptr, int pid){

  float a, b;
  float buf[3];

  if (pid == 0){
    printf("\n  Introduce a, b (limits) and
        n (num. of trap.)  \n");
    scanf("%f %f %d", &a, &b, n_ptr);
    buf[0] = a;
    buf[1] = b;
    buf[2] = (float)*n_ptr;
  }
  //Distribute read values
  MPI_Bcast(&buf,3,MPI_INT,0,MPI_COMM_WORLD)

  (*a_ptr)= (double)(buf[0]);
  (*b_ptr)= (double)(buf[1]);
  (*n_ptr)= (double)(buf[2]);
```

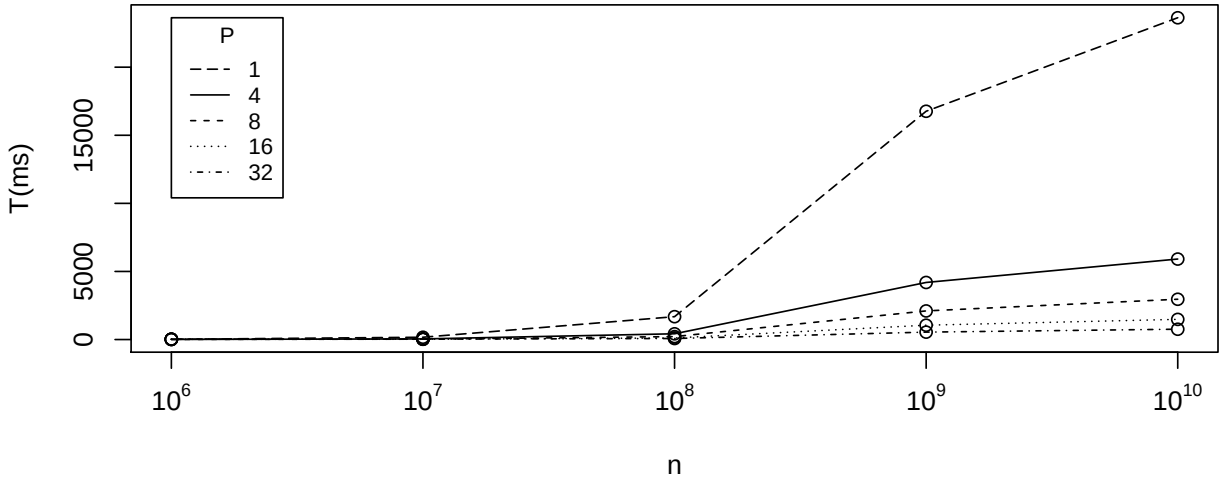
Tras realizar el cálculo del área correspondiente en cada proceso, es necesario sumarlas todas para obtener la integral en todo el intervalo. Esto puede realizarse en una única línea llamando a la función **MPI_Reduce** con los siguientes parámetros.

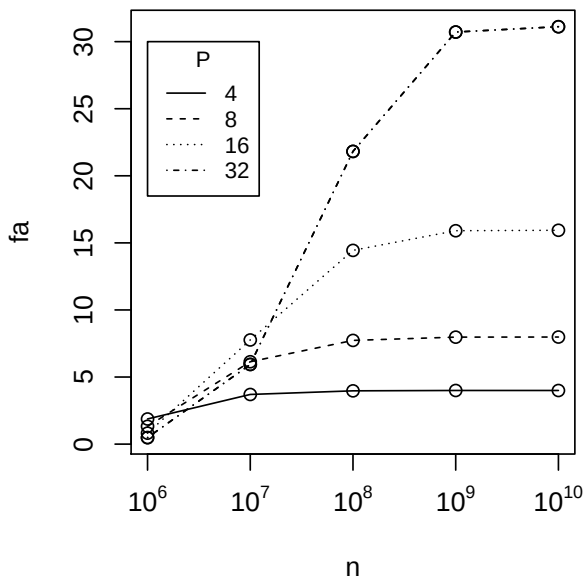
```
/*
Adding the partial results,
```

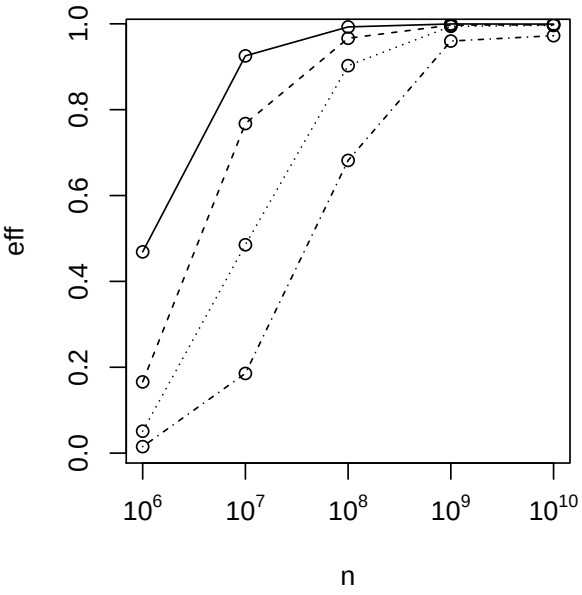
```
Description of parameters:
sendbuf   - local result of integral
recvbuf   - total result of integral
count     - 1 element in send buffer per
    porcess
datatype  - We are using double precission
op        - We will compute a sume
root      - the process 0
comm      - All the processes active
```

```
*/
MPI_Reduce(&resul_loc, &resul, 1,
    MPI_DOUBLE   , MPI_SUM, 0,
    MPI_COMM_WORLD);
```

Las siguientes imágenes muestran los resultados comparando la ejecución para distintos números de procesadores. Los tiempos, los factores de aceleración y la eficiencia de cada uno.







4.3 P1.3

En una ejecución con cuatro procesos, P2 reparte datos del vector B (de 16 enteros) de la siguiente manera: a P0; B[3], B[4], B[5]; a P1: B[7], B[8]; a P2; B[12], B[13], B[14], B[15]. Tras ello, cada proceso suma 100 a los elementos recibido, y, finalmente, se recopilan los datos en P2, en las mismas posiciones iniciales del vector B.

Completa el programa MPI *P13-scatter-gather0.c* para que realice esa función; al principio, P2 debe inicializar B[i]=i, y, al final, imprimir el nuevo vector B.

Como los segmentos a entregar son de tamaños distintos he utilizado la variante de vector de las funciones Scatter y Gather. Primero creo los vectores de displacement, y sendcounts, desde que posiciones quiero que reciban datos y la cantidad de los mismos.

```
int  displacements[4];
int  sendcounts[4];
displacements[0] = 3;  sendcounts[0] = 3;
displacements[1] = 7;  sendcounts[1] = 2;
displacements[2] = 10;  sendcounts[2] = 1;
displacements[3] = 12;  sendcounts[3] = 4;

// Scattering of B from pid=2
MPI_Scatterv(&B, &sendcounts[0], &
    displacements[0], MPI_INT, &Bloc,
    sendcounts[pid], MPI_INT, 2,
    MPI_COMM_WORLD);

// Local calculation
for(i=0; i < sizeof(&Bloc); i++){ Bloc[i
    ]+=100; }

// Gathering of Bloc in pid=2
MPI_Gatherv(&Bloc, sendcounts[pid], MPI_INT,
     &B, &sendcounts[0], &displacements[0],
    MPI_INT, 2, MPI_COMM_WORLD);
```

4.4 P2.1

En un programa MPI, el proceso P3 tiene una matriz MAT de 10x10 enteros, de la que tiene que enviar la diagonal al resto de procesos. Completa el programa *P21-diagonal0.c* para que ejecute esa operación en estos dos casos:

- la diagonal se recibe en los otros procesos como un simple vector, y se calcula e imprime la suma de los elementos recibidos.

- la diagonal se recibe sustituyendo a la diagonal de la matriz local MAT.

Ejecuta el programa con 4 procesos.

Lo primero que se ha hecho es definir el tipo *diagonal*.

```
// Defining the diagonal type
MPI_Datatype diagonal;
MPI_Type_vector(N, 1, N, MPI_INT, &diagonal)
    ;
MPI_Type_commit(&diagonal);
```

A continuación, se desea enviar la diagonal desde el nodo P3 a los nodos P0, P1 y P2. El nodo P3 tendrá que rellenar su tipo diagonal. En este caso, suponiendo que se trabaje con cuatro nodos, todos los nodos ejecutan **broadcast**. El nodo P3 es el emisor y el resto los receptores Se podría haber hecho con un tres *sends* a los respectivos nodos.

```
// 1. Sending the diagonal of the matrix MAT
     in P3 to P0, P1 and P2
// It is received as a vector in a buffer
if (pid==3){
    printf("Soy nodo %d.Voy a enviar mi
        diagonal:\n",pid);
    for (i=0; i<N; i++){
        buf[i]=MAT[i][i];
        printf ("%d",buf[i]);
        printf(",");
    }
    printf("\n");
}
//Sended data by broadcast
MPI_Bcast(&buf, N, MPI_INT, 3,
    MPI_COMM_WORLD);
if (pid !=3){
    for (i=0; i<N; i++){
        printf ("Nodo: %d Buf: [%d][%d] %d\n
            ",pid,i,i,buf[i]);
    }
}
```

4.5 P2.2

Completa el programa *P22-pack0.c* para que P1 envíe a P2 tres elementos en un solo mensaje: una matriz A de 100x100 enteros, un vector B de 2.000 flotantes, y C, un double. Para ello, P1 empaqueta los datos y envía el paquete a P2; por su parte, P2 recibe el mensaje y desempaqueta los datos. Ejecuta el programa con 4 procesos.

Lo primero que se hace es inicializar los datos en P1. A continuación, se empaquetan los datos de interés y se envían al nodo de destino mediante MPI_Send.

```
if (pid==1){
    for(i=0; i<sizeA; i++)
        for(j=0; j<sizeA; j++) A[i][j] =
            i*j;

    for(i=0; i<sizeB; i++) B[i] = (float
        )i*0.4;
    C = 2.2;

    // Packing the data in P1 and
        sending the packet to P2
    MPI_Pack(&A[0][0], sizeA*sizeA,
        MPI_INT, buf, sizebuf, &pos,
        MPI_COMM_WORLD );
    MPI_Pack(&B[0], sizeB, MPI_FLOAT,
        buf, sizebuf, &pos,
        MPI_COMM_WORLD);
    MPI_Pack(&C, 1, MPI_DOUBLE, buf,
        sizebuf, &pos, MPI_COMM_WORLD);
    MPI_Send(buf, pos, MPI_PACKED, 2, 0,
         MPI_COMM_WORLD);
}
```

Ahora hay que recibir los datos en el nodo P2. Hay que tener en cuenta que hay que desempaquetar en el mismo orden que en el que se envía.

```
if (pid==2){
    MPI_Recv(buf, sizebuf, MPI_PACKED,
        1, 0, MPI_COMM_WORLD, &info);
    //Unpacking
    pos=0;
    MPI_Unpack(buf, sizebuf, &pos, &A
        [0][0], sizeA*sizeA , MPI_INT,
        MPI_COMM_WORLD);
    MPI_Unpack(buf, sizebuf, &pos, &B
        [0], sizeB, MPI_FLOAT,
        MPI_COMM_WORLD);
    MPI_Unpack(buf, sizebuf, &pos, &C,
        1, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

4.6 P3.1

El programa *P31-collatzser.c* aplica una función basada en el algoritmo de Collatz a números enteros desde 1 a 320, con una carga de trabajo proporcional al número de iteraciones necesarias para que los números converjan a 1.

Hay que hacer dos versiones paralelas de ese programa. En la primera, se reparten las tareas entre todos los proceso de modo estático consecutivo, procesando cada uno de ellos 320/npr números consecutivos.

En la segunda, el reparto de tareas debe ser dinámico, bajo demanda. Uno de los procesos (P0, por ejemplo) funciona como manager y reparte a cada uno de los restantes procesos (workers) números a procesar, uno a uno, cuando lo solicitan. Cada worker procesa ese número, y devuelve al manager el número de iteraciones que ha necesitado para converger. Si quedan números por analizar, se le envía una nueva tarea, hasta terminar de analizar entre todos los workers todos los números. El proceso manager debe controlar cuántos números ha procesado cada worker, y el número que ha necesitado más iteraciones para converger.

Versión estática

Una vez declaradas las funciones y variables necesarias para llevar a cabo nuestro objetivo podemos empezar con la distribución de elementos a analizar:

```
for (n=(((NUMBER/npr)*pid)+1); n<=((NUMBER/
    npr)*(pid+1)); n++)
```

De esta manera conseguimos que los elementos que forman el problema a solventar se reparten de manera equitativa y en orden con el número de procesadores involucrados en el desarrollo.

```
Ej: npr = 4
pid = 0 => 1..80 pid = 1 => 81..160
pid = 2 => 161..240 pid = 3 => 241..320
```

Una vez distribuidos los elementos cada procesador ejecuta el algoritmo de Collatz en el rango de valores asignado:

```
steps=collatz(n);
work(steps);
total_steps+=steps;
if (steps > max_steps) {n_max_steps = n;
    max_steps = steps;}
```

Una vez calculados los distintos valores obtenidos de la ejecución en paralelo mostramos por pantalla el resultado obtenido:

```
printf("TOTAL (%d numbers) ---> total steps:
     %d  -- n_max_steps: %d (%d steps) --
    execution time: %1.3f ms\n\n", NUMBER/
    npr, total_steps, n_max_steps, max_steps
    , tex*1000);
```

Versión dinámica

En este caso el reparto de tareas se hace en tiempo de ejecución, por lo que tenemos que determinar un protocolo entre los workers y el manager para que el intercambio de mensaje sea claro y preciso entre ambas partes. Los posibles mensajes que un worker puede enviar al manager son los siguientes:

1. Mándame un elemento nuevo para tratarlo.
2. He terminado con este elemento, te envío el resultado.

Los posibles mensajes que un manager pueden los siguientes:

1. Te mando un nuevo elemento a trabajar.
2. No hay más elementos a tratar.

Por lo tanto el protocolo que se ha diseñado es el siguiente:

- Si un worker necesita un dato nuevo para tratar usará el TAG 0 y enviará un valor basura al manager. En cambio si lo que desea es enviar resultados el TAG que utilizará será 1 y en el entero que envía se encontrará el número de iteraciones necesarias para converger el número a 1.
- El manager por otro lado enviará un mensaje con el TAG 0 y el valor pedido cuando un worker le solicite un nuevo número a tratar, en cambio si todos los números han sido tratados enviará un valor basura con el TAG 1.

Utilizando este protocolo y realizando las inicializaciones necesarias el código que representa al protocolo por parte de worker y manager es el siguiente:

Manager:

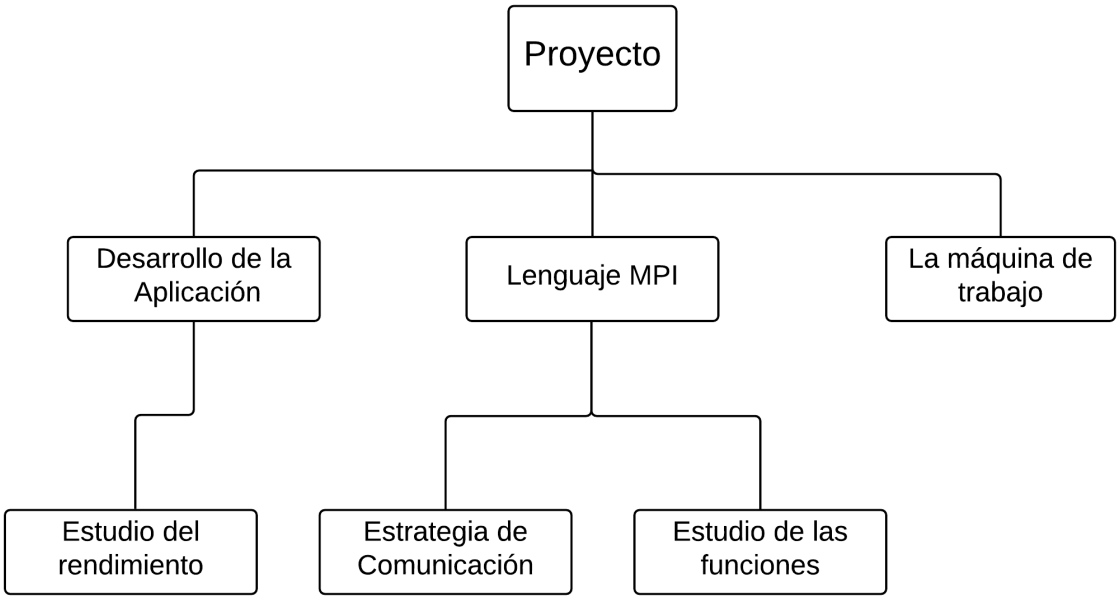
```
if (pid==0){
   MPI_Recv(&f, 1, MPI_INT, MPI_ANY_SOURCE,
       MPI_ANY_TAG, MPI_COMM_WORLD, &info);
    if (info.MPI_TAG==0) { //Si piden
        trabajo
          if (n<=NUMBER){//Si hay trabajo
              MPI_Send(&n, 1, MPI_INT, info
                  .MPI_SOURCE, 0,
                  MPI_COMM_WORLD);
         }else{//Si no hay
              MPI_Send(&n, 1, MPI_INT, info
                  .MPI_SOURCE, 1,
                  MPI_COMM_WORLD); }
  }else if (info.MPI_TAG==1){ //Si envian
      resultados
        total_steps+=f;
        if (f > max_steps) {n_max_steps = n;
            max_steps = f;}
  }
```

Worker:

```
} else{
     MPI_Send(&a, 1, MPI_INT, 0, 0,
         MPI_COMM_WORLD); //Pide trabajo
     MPI_Recv(&b, 1, MPI_INT, 0, MPI_ANY_TAG
         , MPI_COMM_WORLD, &info);
     if (info.MPI_TAG == 0){ //Si hay
         trabajo
        steps=collatz(b);
        work(steps);
        MPI_Send(&steps, 1, MPI_INT, 0, 1,
            MPI_COMM_WORLD);
     }
 }
```

5 Anexo

5.1 Poster



5.2 Tabla de dedicación

Tarea	Mikel	Jose Ángel	Christian
Estudio Programación MPI	1h 30’	1h	1h
Puzle 1	13h 45’	15h 45’	8h 30’
-Estudio	7h	8h	3h
-Documentación	3h	2h	2h
-Problemas	3h	5h	3h
-Presentación	45’	45’	30’
Total	15h 15’	16h 15’	9h 30’

5.3 Acta de Constitución del Grupo

5.4 Actas de Reunión