

Sistemas Paralelos

GUIA PARA ESTIMAR EL COSTE DE
COMUNICACIÓN EN PROGRAMAS PARALELOS
MPI

DENIS ROMASANTA

1 TABLA DE CONTENIDO

2	Objetivo	2
3	Sistema evaluado	2
4	Benchmarks utilizados	3
4.1	Clasificación	3
4.1.1	Single transfer benchmarks	3
4.1.2	Parallel transfer benchmarks	3
4.1.3	Collective benchmarks.....	4
4.2	Ping pong	4
4.3	SendReceive	5
4.4	Broadcast	6
4.5	Scatter	7
4.6	gather	7
4.7	Alltoall.....	8
4.8	Reduce	9
4.9	barrier	10
5	Metodología general	11
6	Resultados.....	13
6.1	Consideraciones previas.....	13
6.2	Ping pong	13
6.3	SENDRECEIVE	14
6.4	BROADCAST	15
6.5	SCATTER.....	16
6.6	gather	17
6.7	AllToAll	18
6.8	Reduce	18
6.9	barrier	19
7	Conclusiones	20
8	Bibliografía.....	20
9	Apéndice código	20
9.1	Introducción	20
9.2	benchmarks.h	21
9.3	Benchmarks.c.....	22
9.4	Config.h	29
9.5	config.c	30
9.6	MAIN.C.....	31

2 OBJETIVO

El objetivo de este informe es describir los resultados encontrados al analizar el coste de las comunicaciones en un sistema de computación de altas prestaciones empleando para ello las funciones MPI y como metodología la descrita en el documento Intel MPI Benchmarks.

3 SISTEMA EVALUADO

Finis Terrae II es un sistema de computación basado en procesadores Intel Haswell e interconectado mediante red Infiniband con un rendimiento pico de 328 TFlops y sostenido en Linpack de 213 Tflops. Está compuesto por 3 tipos de nodos de computación:

- 306 nodos Thin : 2 x Intel Haswell 2680v3, 24 cores, 128GB, 1TB HDD local
- 4 nodos con aceleradoras GPU : Thin node + 2 GPUs K80 por nodo
- 2 nodos con acelerador Intel Phi : Thin node + 2 Intel Phi 7120 por nodo
- 1 nodo Fat : 8 x Intel Haswell 8867v3, 128 cores, 4TB, 24 TB HDD local
- 4 nodos login : Thin node (nodos de acceso)

Todos interconectados con red Infiniband FDR 56Gbps

El sistema operativo utilizado es Linux RedHat 6.7

El sistema de colas está basado en Slurm:

- Tiempo previsto previamente para los trabajos
- Conexión a los nodos

Para las pruebas realizadas en este documento se han utilizado los nodos de Intel Haswell.

Se ha hecho uso de la partición thinnodes que es la adecuada para realizar trabajos paralelos que hacen uso de varios nodos.

4 BENCHMARKS UTILIZADOS

4.1 CLASIFICACIÓN

Los distintos benchmarks utilizados se clasifican en:

- Single transfer benchmarks
- Paraller transfer benchmarks
- Collective transfer benchmarks

4.1.1 Single transfer benchmarks

Son aquellos que incluyen dos procesos activos en la comunicación. Los otros procesos esperan por que la comunicación se complete. Cada benchmark se ejecuta con varias longitudes de mensajes y el tiempo es medido como la media entre dos procesos.

Para este tipo de benchmarks el rendimiento se obtiene con la siguiente fórmula:

$$throughput = \frac{X}{\frac{1.048576}{time}}$$

X = Longitud del mensaje en bytes

time = Tiempo medido en μ

throughput = Ancho de banda en MBps

El tipo MPI utilizado es MPI_BYTE

*Dentro de este grupo se evaluará el benchmark **pingPong**.*

4.1.2 Parallel transfer benchmarks

Los benchmarks paralelos son aquellos que involucran a más de dos procesos en la comunicación, cada benchmark corre con varias longitudes de mensajes y el tiempo se obtiene como la media de múltiples muestras.

El rendimiento toma en cuenta para su cálculo la mutiplicidad de los mensajes que salen y entran de un proceso en particular. Por ejemplo para SendRecv un proceso envía y recibe X bytes, por lo tanto nmsg = 2 .

El rendimiento se calcula como sigue:

$$throughput = nmsg * \frac{X}{\frac{1.048576}{time}}$$

X = Longitud del mensaje en bytes

time = Tiempo medido en μ

nmsg = Multiplicidad de los mensajes que entran y salen de un determinado proceso

throughput = Ancho de banda en MBps

El tipo MPI utilizado es MPI_BYTE

Dentro de este grupo se evaluará el benchmark **sendReceive**.

4.1.3 Collective benchmarks

Los benchmarks colectivos son aquellos que miden operaciones MPI colectivas. Cada benchmark se ejecuta con distintas longitudes de mensaje. El tiempo es la media entre múltiples muestras.

El tipo MPI básico para estas operaciones es MPI_BYTE para las funciones de movimiento de datos y MPI_FLOAT para las reducciones.


Para estos benchmarks no se evaluara el rendimiento.

Dentro de este grupo se evaluará el benchmark **Broadcast, Scatter, Gather, AlltoAll, Barrier**.

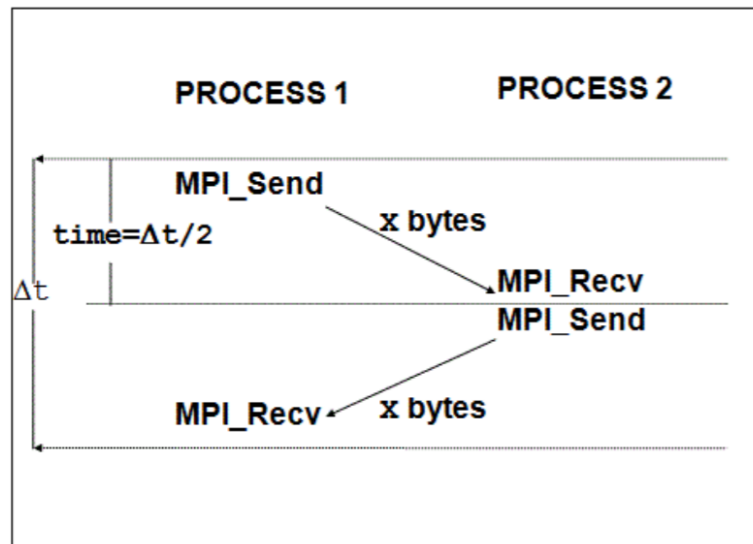
4.2 PING PONG

Pertenece a los single transfer benchmarks. Hace uso de MPI_Send y MPI_Receive. Se base en el intercambio de mensajes tal y como se muestra en la figura:

Definición:

Property	Description
Measured pattern	As symbolized between  in the figure below. This benchmark runs on two active processes (Q=2) .
MPI routines	<code>MPI_Isend/MPI_Wait, MPI_Recv</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	<code>time=Δt (in μsec)</code>
Reported throughput	<code>X/(1.048576*time)</code>


Patrón:



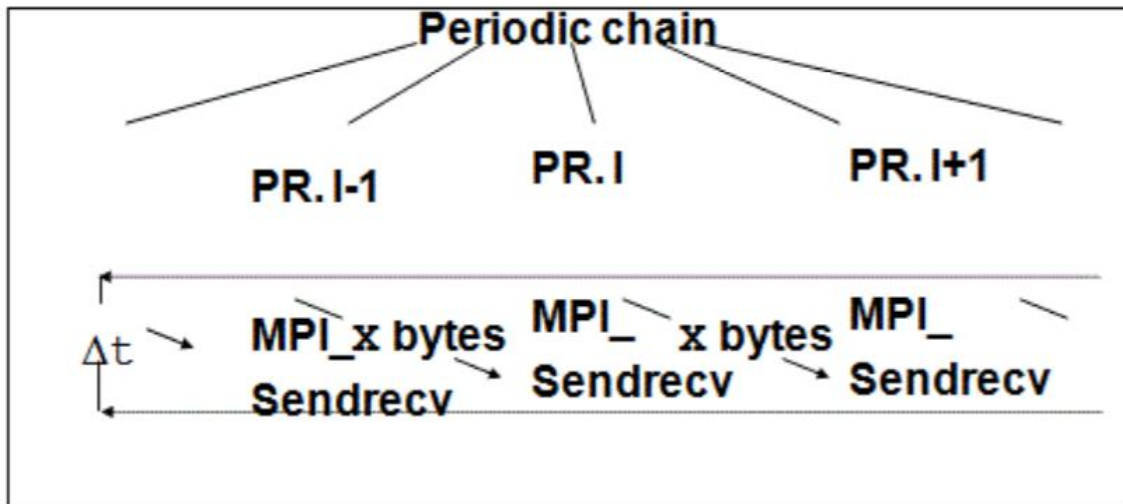
4.3 SENDRECEIVE

Pertenece a los parallel benchmks. Esta basado en MPI_Sendrecv. En este benchmark los procesos forman una cadena de comunicación. Cada proceso envía un mensaje al vecino de la derecha y recibe un mensaje del vecino de la izquierda de la cadena. La cantidad de información intercambiada es 2 mensajes por muestra, una entrada y salida de datos por proceso.

Definición:

Property	Description
Measured pattern	As symbolized between  in the figure below.
MPI routines	<code>MPI_Sendrecv</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	<code>time=Δt</code> (in <code>μsec</code>) as indicated in the figure below.
Reported throughput	$2X / (1.048576 * \text{time})$

Patrón:



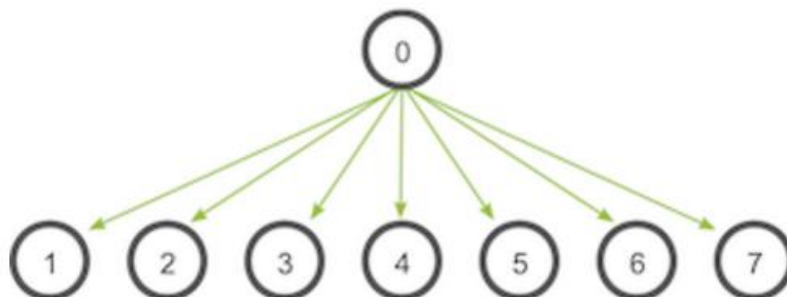
4.4 BROADCAST

Pertenece a los benchmarks colectivos. Un proceso principal envía al resto de procesos X bytes. El proceso principal de la operación se cambia con el algoritmo “round-robin”.

Definición:

Property	Description
Measured pattern	<code>MPI_Alltoall</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time
Reported throughput	None

Patrón:



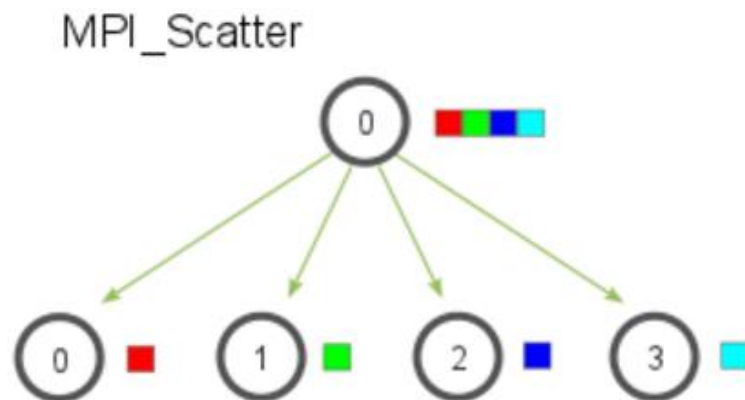
4.5 SCATTER

Pertenece a los benchmarks colectivos. Utiliza la función `MPI_Scatterv`. El proceso principal envía $X * N^{\circ}\text{Procesos}$ bytes (X para cada proceso). Todos los procesos reciben X bytes. El proceso principal se cambia según el algoritmo “round-robin”.

Definición:

Property	Description
Measured pattern	<code>MPI_Scatterv</code>
MPI data type	<code>MPI_BYTE</code>
Root	$i \% \text{num_procs}$ in iteration i
Reported timings	Bare time
Reported throughput	None

Patrón:



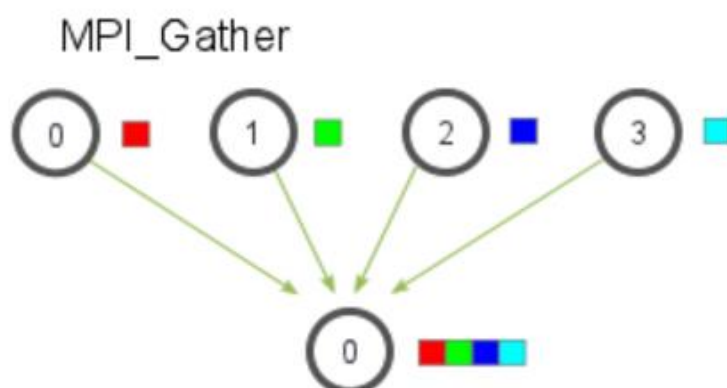
4.6 GATHER

Pertenece a los benchmark colectivos. El proceso principal envía $X * N^{\circ}\text{Procesos}$ bytes. X desde cada proceso. Todos los procesos reciben X bytes. El proceso principal de la operación se cambia mediante el algoritmo “round-robin”.

Definición:

Property	Description
Measured pattern	<code>MPI_Gather</code>
MPI data type	<code>MPI_BYTE</code>
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

Patrón:



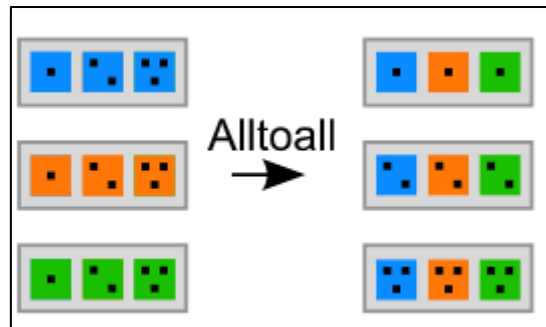
4.7 ALLTOALL

Pertenece a los patrones colectivos. Cada proceso envía $X \cdot N^{\circ}\text{Procesos}$ bytes y recibe $X \cdot N^{\circ}\text{Procesos}$ bytes, X por cada proceso.

Definición:

Property	Description
Measured pattern	<code>MPI_Alltoall</code>
MPI data type	<code>MPI_BYTE</code>
Reported timings	Bare time
Reported throughput	None

Patrón:



4.8 REDUCE

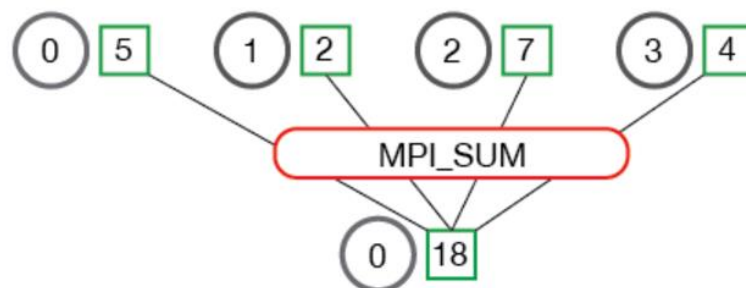
Pertenece a los benchmarks colectionvs. Este benchmark reduce un vector de longitud $L = X / \text{sizeof}(\text{float})$ de elementos float. La operación MPI a utilizar es MPI_SUM. El proceso principal de la operación se cambia según el algoritmo “round-robin”.

Definición:

Property	Description
Measured pattern	<code>MPI_Reduce</code>
MPI data type	<code>MPI_FLOAT</code>
MPI operation	<code>MPI_SUM</code>
Root	<code>i%num_procs</code> in iteration <code>i</code>
Reported timings	Bare time
Reported throughput	None

Patrón:

MPI_Reduce



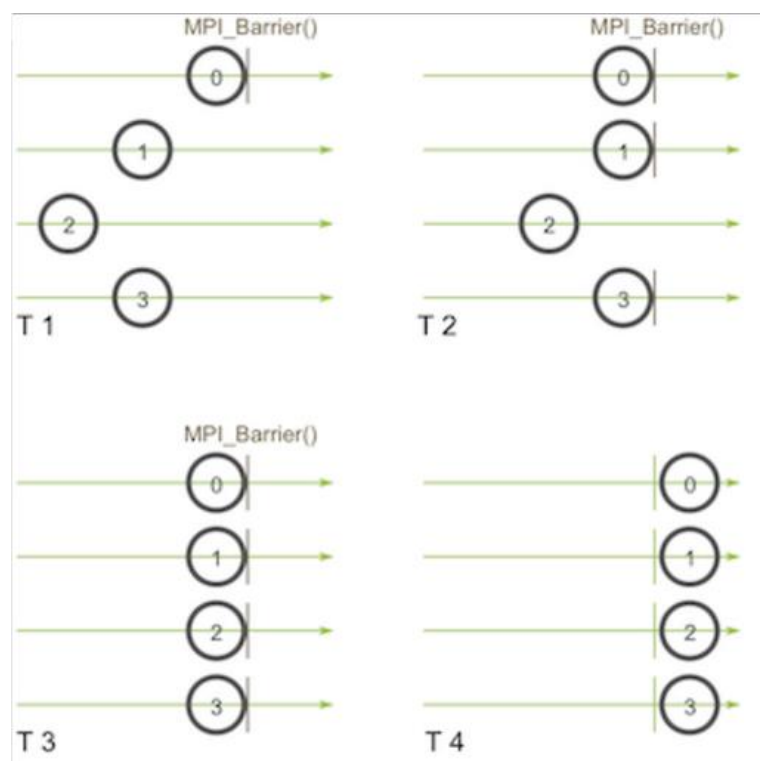
4.9 BARRIER

Pertenece a los bencharmks colectivos. Impide que los procesos avancen hasta que todos hayan llegado al punto delimitado por la barrera.

Definición:

Property	Description
Measured pattern	MPI_Barrier
Reported timings	Bare time
Reported throughput	None

Patrón:



5 METODOLOGÍA GENERAL

Teniendo en cuenta las particularidades propias de cada benchmark para la medición del rendimiento y los tiempos (Detalladas individualmente en la sección previa) aquí se procede a una descripción de la metodología general utilizada para los diversos benchmarks.

Para la elección del número de veces que se ejecuta cada test y el número de bytes utilizados se toma como referencia la metodología descrita en el documento de Intel.

Cada test se realizará utilizando la siguiente cantidad de bytes de transferencia

(cuando proceda) y se repetirá las veces indicadas:

Bytes	Times
0	1000
1	1000
2	1000
4	1000
8	1000
16	1000
32	1000
64	1000
128	1000
256	1000
512	1000
1024	1000
2048	1000
4096	1000
8192	1000
16384	1000
32768	1000
65536	640
131072	320
262144	160
524288	80
1048576	40
2097152	20
4194304	10

Previamente a la realización de cada test, se realizará una ronda de calentamiento que consistirá en ejecutar el propio test, pero sin tomar medidas. Esto nos permitirá garantizar que los resultados no se ven influenciados por la primera ejecución.

Para la medición de tiempos se realizará lo siguiente:

- 1- Se realiza una barrier previamente a la ejecución del benchmark.
- 2- Se inicia el contador de tiempo de ejecución para cada proceso.
- 3- Se realiza el benchmark.
- 4- Se finaliza el contador de tiempo de ejecución para cada proceso.
- 5- Se utiliza MPI_Reduce para comunicar el tiempo de ejecución de cada proceso a un tiempo acumulado de ejecución global.
- 6- Se realiza una barrier para garantizar que todos han terminado.
- 7- Se obtiene el tiempo medio de la ejecución dividiendo el tiempo acumulado de ejecución global entre el número de procesos.

El número de procesos con los que se realiza cada test es el siguiente:

PingPong	2
SendReceive	4,8,16
Broadcast	4,8,16
Scatter	4,8,16
Gather	4,8,16
AlltoAll	4,8,16
Barrier	4,8,16
Reduce	4,8,16

6 RESULTADOS

6.1 CONSIDERACIONES PREVIAS

Los tiempos se mostrarán normalizados:

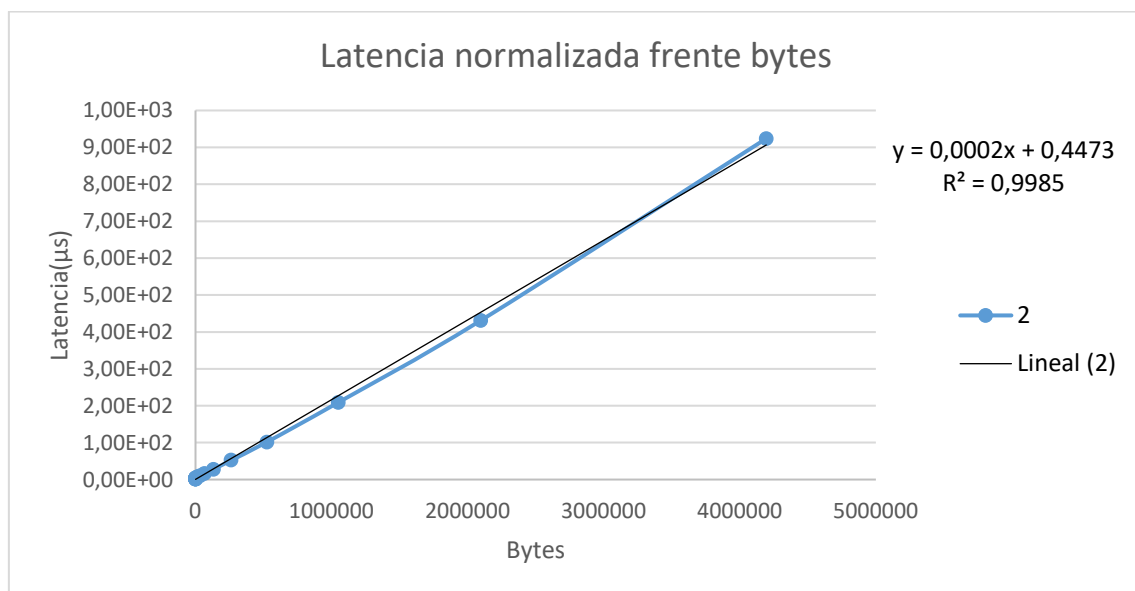
Normalized time (μs)	9,54E-01
-----------------------------	----------

En la leyenda se indica el número de procesos con los que se ha ejecutado.

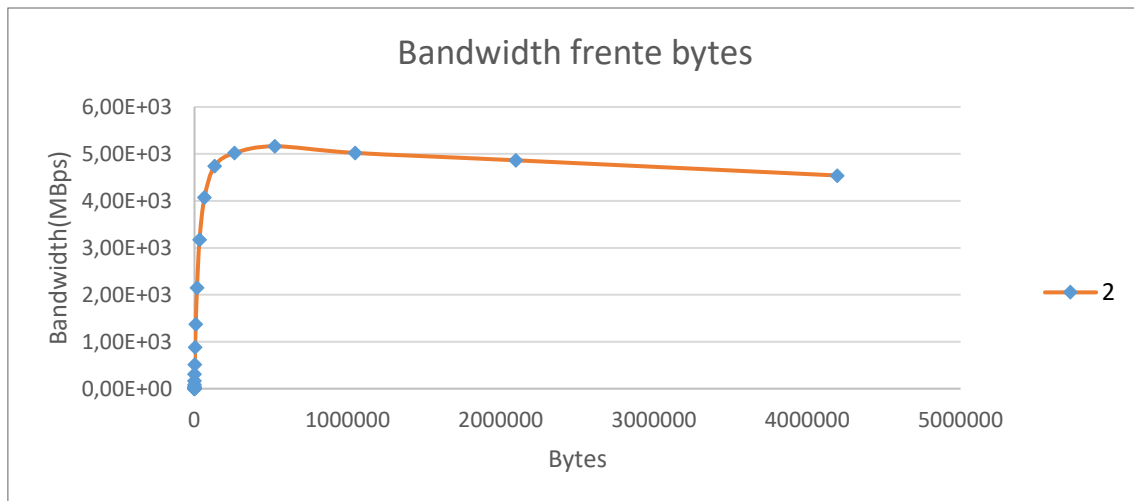
En los gráficos se muestran también las ecuaciones que se ajustan a los resultados obtenidos.

El ancho de banda se muestra por proceso.

6.2 PING PONG

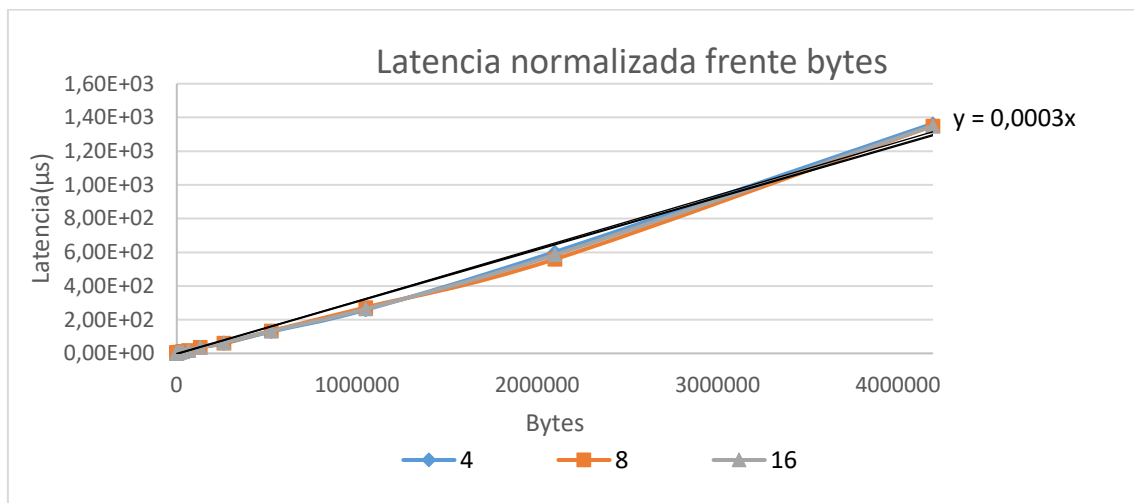


Los resultados muestran como a mayor número de bytes se tarda más en realizar el algoritmo. Lo cual concuerda con lo esperado, según la ecuación a la que se ajusta vemos que crece linealmente y que además no es desproporcionado el aumento conforme se transfieren más bytes.



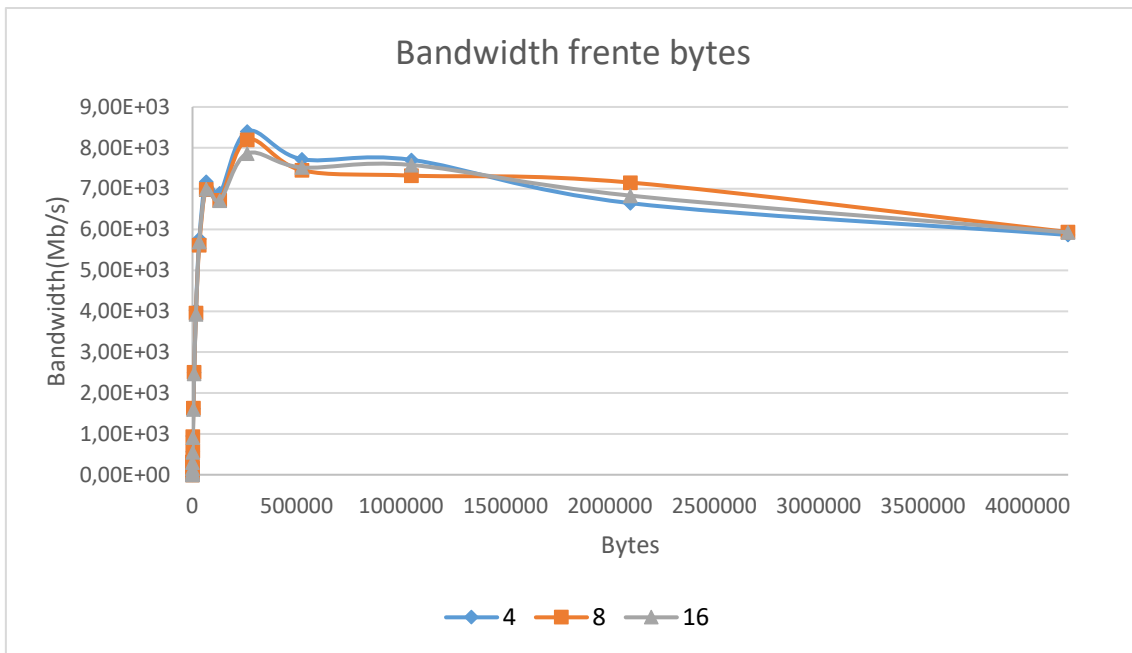
En está gráfico se observa cómo se llega a saturar la red y a partir de un determinado ancho de banda este no aumenta aunque se eleven las necesidades de envío de datos en la red.

6.3 SENDRECEIVE



Se observa como el coste de la transferencia en sendReceive aumenta de forma lineal con el número de bytes, lo cual es lógico dado que enviar una mayor cantidad de datos provoca una mayor latencia.

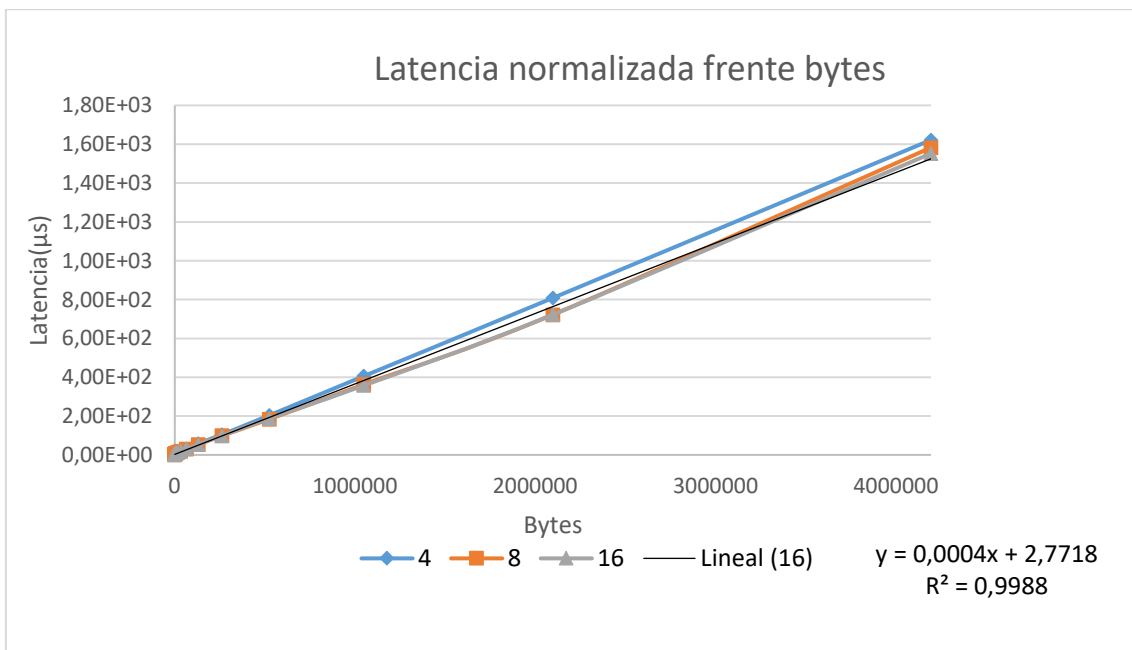
El uso de más procesos no cambia la latencia, esto es lógico dado que en sendReceive cada proceso recibe del proceso que tiene a su izquierda y envía al de la derecha. Por lo tanto, el hecho de usar una mayor cantidad de procesos no debería influir significativamente en el resultado y es lo que se observa.



En la gráfica de ancho de banda se observa que este alcanza el punto de saturación de la red y a partir de ahí independientemente del tamaño del paquete a enviar la velocidad de transferencia se mantiene invariable.

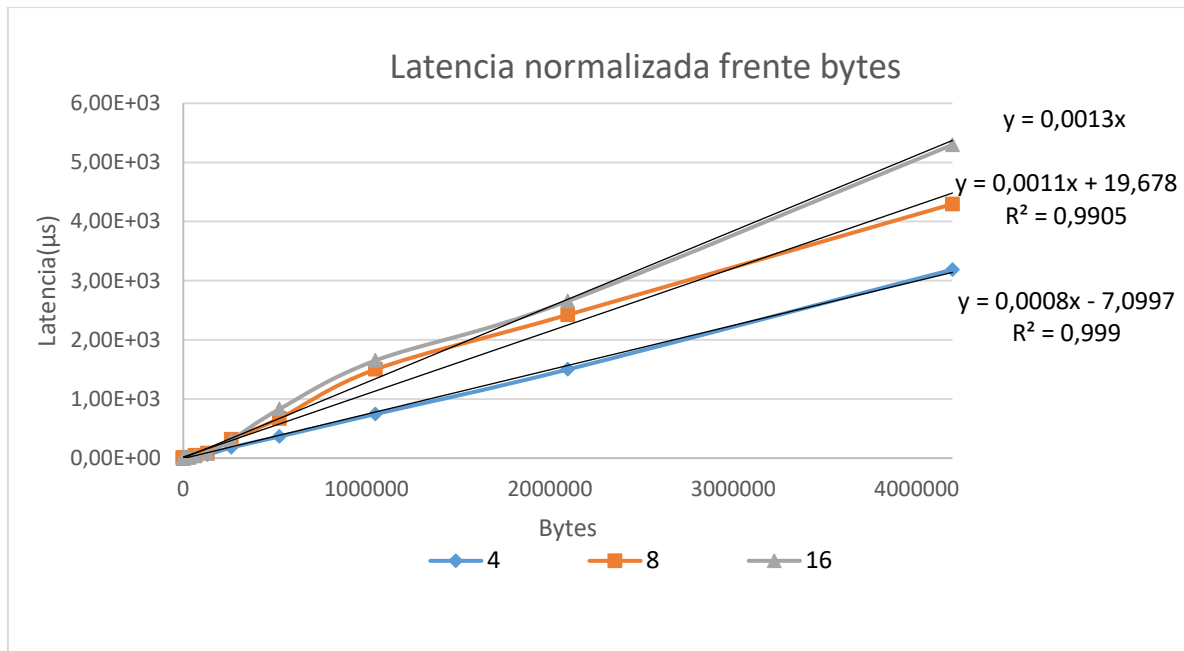
Dado que sendReceive sigue un esquema de recibo datos del proceso de la izquierda y envío al de la derecha es lógico que los resultados de ancho de banda no varíen con el número de procesos. Nótese que el ancho de banda se mide por proceso.

6.4 BROADCAST



Sorprenden estos resultados en broadcast, se deduce que esta implementado de una forma muy eficiente por MPI. El coste según el tamaño del mensaje aumenta linealmente, lo cual parece razonable.

6.5 SCATTER

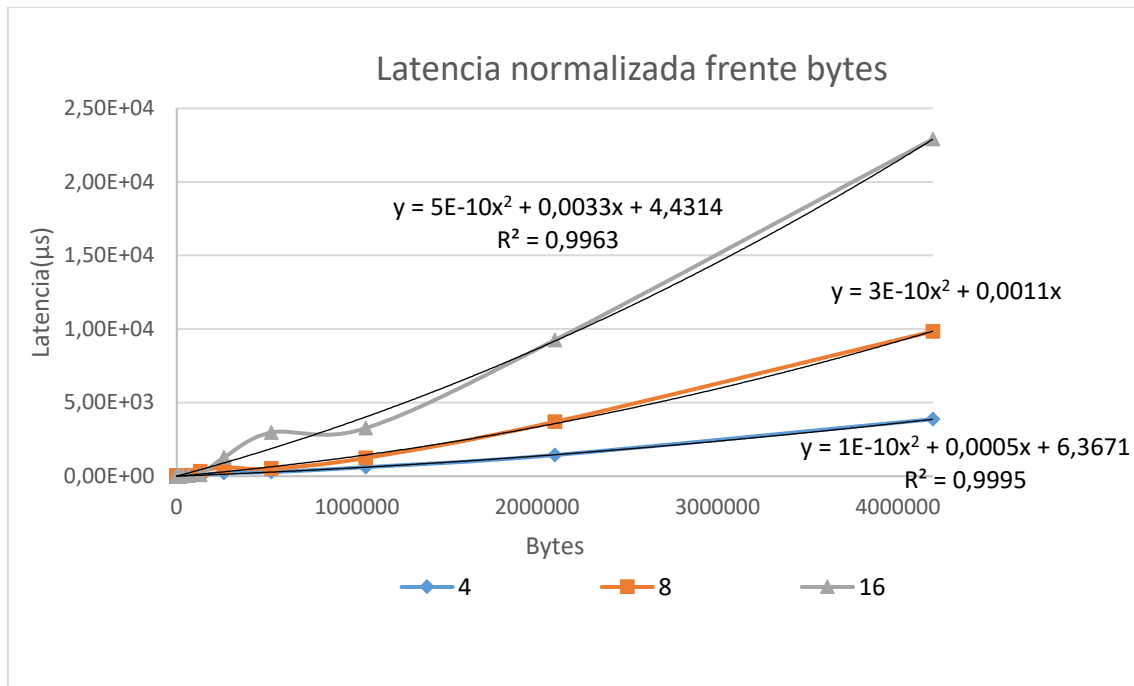


A mayor número de procesos también se observa un mayor coste, esto tiene sentido dado que se produce una mayor cantidad de envíos por parte de todos los procesos y recepción por parte del proceso principal.

Este patrón de comunicación escala de forma lineal con el tamaño de bytes enviados.

Dado que la cantidad de envíos y recepciones es mayor, tiene lógica esperar que la latencia aumente conforme el número de bytes enviados.

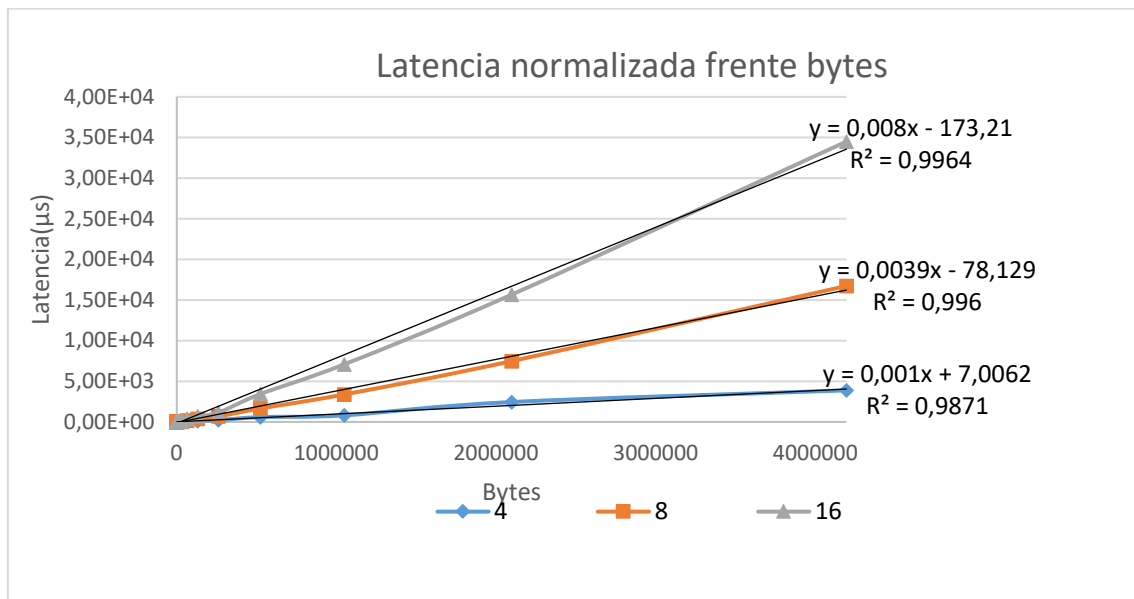
6.6 GATHER



Sorprende el hecho de que no sea completamente similar al scatter dado que es prácticamente su inversa. De esto podemos deducir que el hecho de que un proceso principal reciba datos de muchos procesos es una operación más costosa que el envío desde un mismo proceso a múltiples destinos.

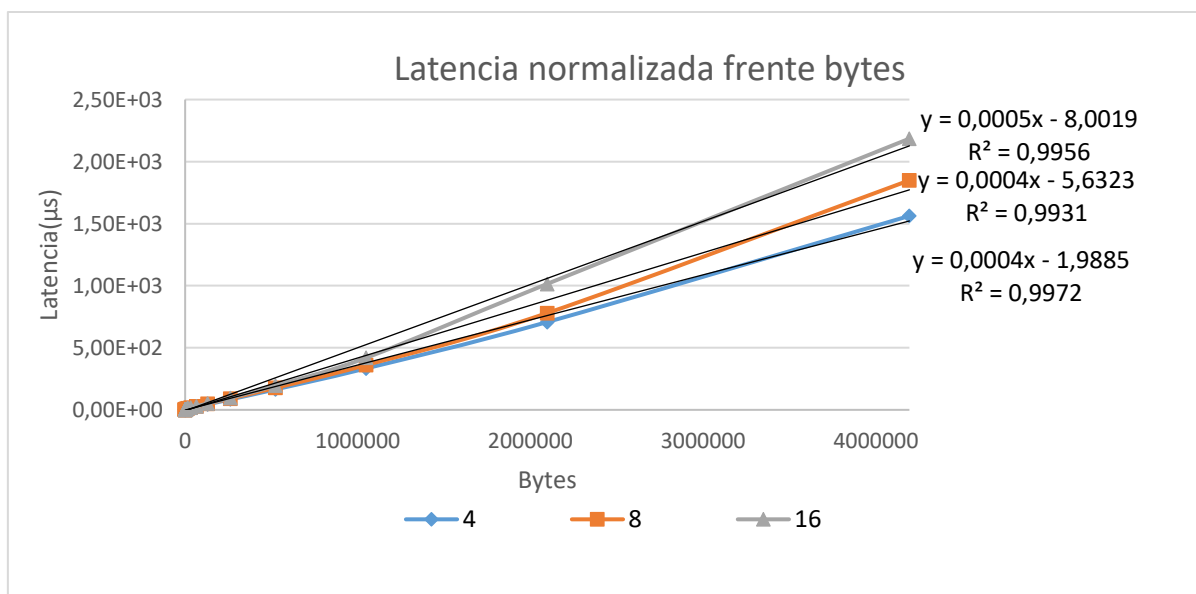
El patrón gather se ajusta mejor a una ecuación polinómica de grado 2, con lo cual su coste no escala tan bien como los patrones de comunicación anteriores. Tiene sentido el aumento del coste conforme el tamaño a enviar y recibir aumenta.

6.7 ALLToALL



Este patrón de comunicación escala de forma línea con el número de bytes, sin embargo, se observa que el número de procesos influye sustancialmente más en la latencia que en casos anteriores, esto es debido a que AllToAll como su propio nombre indica realiza múltiples envíos y recepciones de todos los procesos, lo que supone un gran costo para la red de comunicación que se ve reflejado en los resultados.

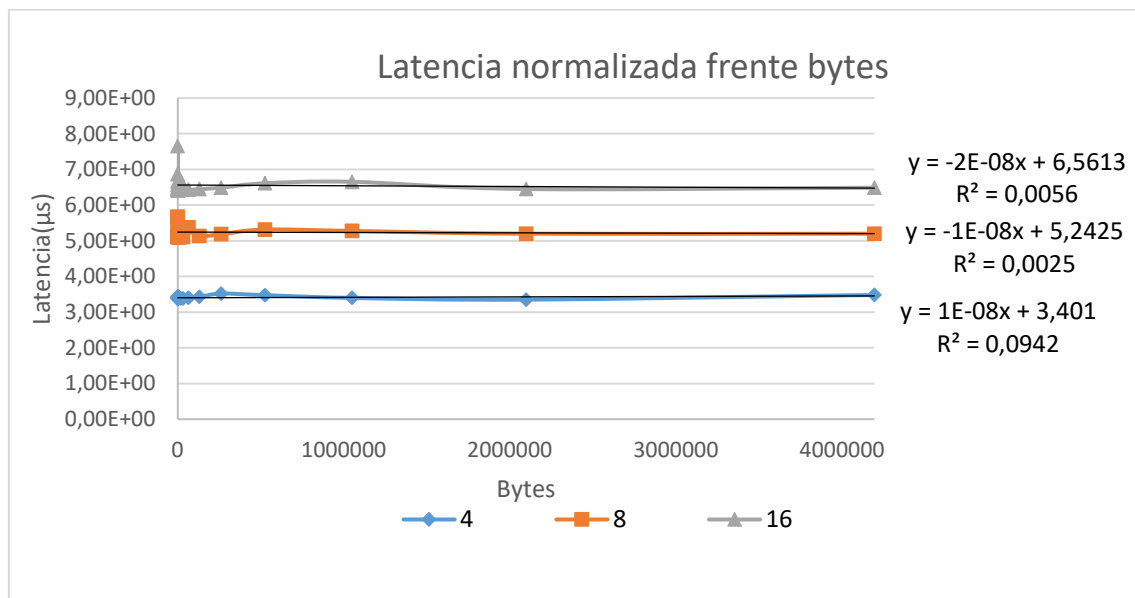
6.8 REDUCE



Se observa como MPI_Reduce tiene un coste lineal con respecto al número de bytes, dado que a mayor cantidad de mensaje a enviar es lógico que el costo de computación aumente.

En cuanto al número de procesos involucrados en el algoritmo se observa como el coste no aumenta en una gran cantidad conforme usamos más procesos. Resulta curiosa la diferencia observada entre reduce y gather dado que ambos deberían tener un coste similar, que es el de recibir en un único proceso los envíos de múltiples procesos.

6.9 BARRIER



Se observa como el método de barrera no depende del número de bytes, lo cual es lógico dado que ni siquiera es un parámetro del propio método y que conforme tenemos que esperar a una mayor cantidad de procesos es más probable que el sistema tarde más.

7 CONCLUSIONES

MPI nos proporciona una serie de métodos que nos permiten realizar comunicaciones entre procesos de una forma sencilla e intuitiva. Gracias a MPI los clásicos problemas a los que nos enfrentábamos con la creación de hilos, como puede ser la sincronización o la compartición de memoria, desaparecen.

Destaca el hecho de que el coste de las comunicaciones escala de una manera lineal para el tamaño de bytes en la mayoría de los patrones analizados, un dato sorprendente dada la dificultad existente en computación de conseguir costes que no aumenten exponencialmente.

Nótese que la transferencia de datos o ancho de banda una vez alcanza su límite de saturación no mejora, aunque introduzcamos una mayor cantidad de procesos o bytes a enviar. Esto se encuadra dentro del comportamiento lógico debido a que una vez se alcanza el ancho de saturación es imposible que la red de más de sí.

8 BIBLIOGRAFÍA

- Apuntes de la asignatura de Sistemas Paralelos
- www.cesga.es Consultado a 13/11/2016
- Documento de Intel MPI Benchmarks
- <http://mpitutorial.com/> Consultado a 13/11/2016

9 APÉNDICE CÓDIGO

9.1 INTRODUCCIÓN

Los archivos benchmarks.c y benchmarks.h continen los diferentes benchmarks.

Los archivos config.c y config.h contiene el código para la realización de la medición de tiempos, el código para gestionar el número de veces y de bytes que ejecutamos en cada benchmark.

El archivo config.h contiene diversas constantes como son el número máximo de mensajes por muestra o la constante WARM_UP que indica cuantas rondas de calentamiento se realizan antes de la evaluación de cada benchmark.

El archivo main.c inicializa el entorno y ejecuta los distintos benchmarks según se defina en las flags de la parte superior del archivo.

Para ver la explicación de la gestión de los tamaños y veces que se ejecuta un algoritmo puede visitar la sección previa de metodología.

En la sección metodología también se detalla cual es el procedimiento para la medición de tiempos en los benchmarks.

9.2 BENCHMARKS.H

```
//  
// Created by denis on 13/12/16.  
//  
  
#ifndef BENCHMARKS_BENCHMARKS_H  
#define BENCHMARKS_BENCHMARKS_H  
  
void pingPong(int times, int bytes);  
void sendReceive(int times, int bytes);  
void broadcast(int times, int bytes);  
void scatter(int times, int bytes);  
void gather(int times, int bytes);  
void reduce(int times, int bytes);  
void allToAll(int times, int bytes);  
void barrier(int times, int bytes);  
  
#endif //BENCHMARKS_BENCHMARKS_H
```

9.3 BENCHMARKS.C

```
//
// Created by denis on 13/12/16.
//

#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include "benchmarks.h"
#include "config.h"

void pingPong(int times, int bytes){
    int k = 0;
    for(k=0; k <= WARM_UP;k++){
        //Init MPI
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        if(world_rank==0 && world_size!=2){
            printf("Ping pong should be executed with only 2
processes\n");
            return;
        }
        //Init content
        char *packet;
        packet = (char*)malloc(bytes);

        //Init timer
        double timeGlobal = 0;
        double time = 0;

        // Configure algorithm
        int PING_PONG_TIMES = times*2;
        int ping_pong_count = 0;

        // Synchronize process
        MPI_Barrier(MPI_COMM_WORLD);
        //Start timer
        time = MPI_Wtime();
        while (ping_pong_count < PING_PONG_TIMES) {

            if((ping_pong_count+world_rank)%2==0){
                MPI_Send(packet, bytes, MPI_BYTE, (world_rank+1)%2, 0,
MPI_COMM_WORLD);
            }

            else{
                MPI_Recv(packet, bytes, MPI_BYTE, (world_rank+1)%2, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }
            ping_pong_count ++;
        }
        // End timer
    }
}
```

```

        time = (MPI_Wtime()-time);
        // Communicate time
        MPI_Reduce(&time, &timeGlobal, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
        //timeGlobal = time;
        // Print results
        MPI_Barrier(MPI_COMM_WORLD);

        if(world_rank == 0 && k==WARM_UP){
            double finalTime =
timeGlobal/2/world_size/times*pow(10,6);
            printf(" %6d ; %6d ; ", bytes,times);
            printf(" %e ; %e; \n", finalTime,
bytes/(THROUGHPUTCONSTANT*finalTime));
        }

        free(packet);
    }
}

void sendReceive(int times,int bytes){
    int k = 0;
    for(k=0; k <= WARM_UP;k++) {
        //Init MPI
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        //Init content
        char *packetSend;
        packetSend = (char *) malloc(bytes);
        char *packetReceive;
        packetReceive = (char *) malloc(bytes);

        //Init timer
        double timeGlobal = 0;
        double time = 0;

        //Configure algorithm
        int nmsg = 2;
        int source = (world_rank - 1);
        int destination = world_rank + 1;

        if (destination >= world_size)
            destination = 0;

        if (source < 0) {
            source = world_size - 1;
        }

        // Synchronize process
        MPI_Barrier(MPI_COMM_WORLD);
        //Start timer
        time = MPI_Wtime();
        int i = 0;
        for (i = 0; i < times; i++) {
            // Make the algorithm
            MPI_Sendrecv(packetSend, bytes, MPI_BYTE, destination,
123, packetReceive, bytes, MPI_BYTE, source, 123,

```



```

        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    // End timer
    time = (MPI_Wtime() - time);
    // Communicate time
    MPI_Reduce(&time, &timeGlobal, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
    // Print results
    MPI_Barrier(MPI_COMM_WORLD);
    if (world_rank == 0 && k == WARM_UP) {
        double finalTime = timeGlobal / world_size / times *
pow(10, 6);
        printf(" %6d ; %6d ; ", bytes, times);
        printf(" %e ; %e; \n", finalTime, nmsg * bytes /
(THROUGHPUTCONSTANT * finalTime));
    }
    free(packetReceive);
    free(packetSend);
}

void broadcast(int times, int bytes){
    int k = 0;
    for(k=0; k <= WARM_UP; k++) {
        //Init MPI
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        //Init content
        char *packetSend;
        packetSend = (char *) malloc(bytes);

        //Init timer
        double timeGlobal = 0;
        double time = 0;

        //Configure algorithm
        int root = 0;

        // Synchronize process
        MPI_Barrier(MPI_COMM_WORLD);
        //Start timer
        time = MPI_Wtime();
        int i = 0;
        for (i = 0; i < times; i++) {
            root = i % world_size;
            MPI_Bcast(packetSend, bytes, MPI_BYTE, root,
MPI_COMM_WORLD);
        }
        // End timer
        time = (MPI_Wtime() - time);
        // Communicate time
        MPI_Reduce(&time, &timeGlobal, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
        // Print results
        MPI_Barrier(MPI_COMM_WORLD);
        if (world_rank == 0 && k == WARM_UP) {
            double finalTime = timeGlobal / times / world_size *
pow(10, 6);

```

```

        printf(" %6d ; %6d ; ", bytes, times);
        printf(" %e ; NP ; \n", finalTime);
    }
    free(packetSend);
}

void scatter(int times,int bytes){
    int k = 0;
    for(k=0; k <= WARM_UP;k++) {
        //Init MPI
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        //Init content
        char *packetSend;
        packetSend = (char *) malloc((size_t) (bytes * world_size));
        char *packetReceive;
        packetReceive = (char *) malloc((size_t) bytes);

        //Init timer
        double timeGlobal = 0;
        double time = 0;

        //Configure algorithm
        int destination = 0;

        // Synchronize process
        MPI_Barrier(MPI_COMM_WORLD);
        //Start timer
        time = MPI_Wtime();
        int i = 0;
        for (i = 0; i < times; i++) {
            destination = i % world_size;
            MPI_Scatter(packetSend, bytes, MPI_BYTE, packetReceive,
bytes, MPI_BYTE, destination, MPI_COMM_WORLD);
        }
        // End timer
        time = (MPI_Wtime() - time);
        // Communicate time
        MPI_Reduce(&time, &timeGlobal, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
        // Print results
        MPI_Barrier(MPI_COMM_WORLD);
        if (world_rank == 0 && WARM_UP == k) {
            double finalTime = timeGlobal / times / world_size *
pow(10, 6);
            printf(" %6d ; %6d ; ", bytes, times);
            printf(" %e ; NP ; \n", finalTime);
        }
        free(packetReceive);
        free(packetSend);
    }
}

void gather(int times,int bytes){
    int k = 0;
    for(k=0; k <= WARM_UP;k++) {
        //Init MPI

```

```

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    //Init content
    char *packetSend;
    packetSend = (char *) malloc((size_t) bytes);
    char *packetReceive;
    packetReceive = (char *) malloc((size_t) (bytes *
world_size));

    //Init timer
    double timeGlobal = 0;
    double time = 0;

    //Configure algorithm
    int destination = 0;

    // Synchronize process
    MPI_Barrier(MPI_COMM_WORLD);
    //Start timer
    time = MPI_Wtime();
    int i = 0;
    for (i = 0; i < times; i++) {
        destination = i % world_size;
        MPI_Gather(packetSend, bytes, MPI_BYTE, packetReceive,
bytes, MPI_BYTE, destination, MPI_COMM_WORLD);
    }
    // End timer
    time = (MPI_Wtime() - time);
    // Communicate time
    MPI_Reduce(&time, &timeGlobal, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
    // Print results
    MPI_Barrier(MPI_COMM_WORLD);
    if (world_rank == 0 && WARM_UP == k) {
        double finalTime = timeGlobal / world_size / times *
pow(10, 6);
        printf(" %6d ; %6d ; ", bytes, times);
        printf(" %e ; NP ; \n", finalTime);
    }
}

void reduce(int times,int bytes){
    int k = 0;
    for(k=0; k <= WARM_UP;k++) {
        //Init MPI
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        //Init content
        int count = bytes / sizeof(MPI_FLOAT);
        float *packetSend = malloc(sizeof(MPI_FLOAT) * count);
        float *packetReceive = malloc(sizeof(MPI_FLOAT) * count);
        int j = 0;
        for (j = 0; j < count; j++){

```

```

        packetSend[j] = ((float) rand() / (float) (RAND_MAX /
rand())));
    }

    //Init timer
    double timeGlobal = 0;
    double time = 0;

    //Configure algorithm
    int root = 0;

    // Synchronize process
    MPI_Barrier(MPI_COMM_WORLD);
    //Start timer
    time = MPI_Wtime();
    int i = 0;
    for (i = 0; i < times; i++) {
        root = i % world_size;
        MPI_Reduce(packetSend, packetReceive, count, MPI_FLOAT,
MPI_SUM, root, MPI_COMM_WORLD);
    }
    // End timer
    time = (MPI_Wtime() - time);
    // Communicate time
    MPI_Reduce(&time, &timeGlobal, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
    // Print results
    MPI_Barrier(MPI_COMM_WORLD);
    if (world_rank == 0 && k == WARM_UP) {
        double finalTime = timeGlobal / world_size / times *
pow(10, 6);
        printf(" %6d ; %6d ; ", bytes, times);
        printf(" %e ; NP ; \n", finalTime);
    }
    free(packetReceive);
    free(packetSend);
}

}

void allToAll(int times,int bytes){
    int k = 0;
    for(k=0; k <= WARM_UP;k++) {
        //Init MPI
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        //Init content
        char *packetSend;
        packetSend = (char *) malloc(bytes * world_size);
        char *packetReceive;
        packetReceive = (char *) malloc(bytes * world_size);

        //Init timer
        double timeGlobal = 0;
        double time = 0;

        // Synchronize process
        MPI_Barrier(MPI_COMM_WORLD);
        //Start timer

```

```

        time = MPI_Wtime();
        int i = 0;
        for (i = 0; i < times; i++) {
            MPI_Alltoall(packetSend, bytes, MPI_BYTE, packetReceive,
bytes, MPI_BYTE, MPI_COMM_WORLD);
        }
        // End timer
        time = (MPI_Wtime() - time);
        // Communicate time
        MPI_Reduce(&time, &timeGlobal, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
        // Print results
        MPI_Barrier(MPI_COMM_WORLD);
        if (world_rank == 0 && k == WARM_UP) {
            double finalTime = timeGlobal / times / world_size *
pow(10, 6);
            printf(" %6d ; %6d ; ", bytes, times);
            printf(" %e ; NP ; \n", finalTime);
        }
        free(packetReceive);
        free(packetSend);
    }
}

void barrier(int times, int bytes){
    int k = 0;
    for(k=0; k <= WARM_UP;k++) {
        //Init MPI
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        int world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        //Init timer
        double timeGlobal = 0;

        // Synchronize process
        MPI_Barrier(MPI_COMM_WORLD);
        //Start timer
        double start = MPI_Wtime();
        int i = 0;
        for (i = 0; i < times; i++) {
            MPI_Barrier(MPI_COMM_WORLD);
        }
        // End timer
        timeGlobal = (MPI_Wtime() - start);
        // Print results
        MPI_Barrier(MPI_COMM_WORLD);
        if (world_rank == 0 && k == WARM_UP) {
            double finalTime = timeGlobal / times * pow(10, 6);
            printf(" %6d ; %6d ; ", bytes, times);
            printf(" %e ; NP \n", finalTime);
        }
    }
}
}

```

9.4 CONFIG.H

```
//  
// Created by denis on 17/10/16.  
//  
  
#ifndef BENCHMARKS_CONFIG_H  
#define BENCHMARKS_CONFIG_H  
  
#include <math.h>  
  
#define SIZE 24  
#define MESSAGES_PER_SAMPLE 1000  
#define OVERALL 41943040  
#define THROUGHPUTCONSTANT 1.048576  
#define WARM_UP 0  
  
int times[SIZE];  
int bytes[SIZE];  
  
double normalizedTime();  
void initBytesAndTimes();  
  
#endif //BENCHMARKS_CONFIG_H
```

9.5 CONFIG.C

```
//  
// Created by denis on 17/10/16.  
//  
  
#include "config.h"  
#include <mpi.h>  
#include <time.h>  
#include <stdlib.h>  
#include <stdio.h>  
  
void printBytesAndTymes() {  
    printf(" Bytes --- Times \n");  
    int i = 0;  
    for(i = 0; i < SIZE ; i++){  
        printf("%d --- %d \n", bytes[i], times[i]);  
    }  
}  
  
int getTimes(int bytes){  
    int value = fmin(MESSAGES_PER_SAMPLE, OVERALL/bytes);  
    int times = fmax(1, value);  
    return times;  
}  
  
void initBytesAndTimes() {  
    bytes[0] = 0;  
    int i = 1;  
    for(i = 1; i < SIZE ; i++){  
        bytes[i] = pow(2, i-1);  
    }  
  
    times[0] = 1000;  
    for(i = 1; i < SIZE ; i++){  
        times[i] = getTimes(bytes[i]);  
    }  
}  
  
double normalizedTime() {  
    int world_rank;  
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  
  
    if(world_rank == 0) {  
        double vector1[1000];  
        double vector2[1000];  
        double r=0;  
        double result=0;  
        double start = 0;  
        double finish = 0;  
  
        srand(time(NULL));  
        int i = 0;  
        for(i = 0; i<1000; i++){  
  
            r = rand();  
            vector1[i]= r;  
            r = rand();  
            vector2[i] = r;
```

```

    }
    start = MPI_Wtime();
    for(i=0;i<1000;i++){
        result = result + vector1[i] * vector2[i];
    }
    finish = MPI_Wtime();
    printf("Normalized time; %e ; microseconds ; accuracy ; %e
;\\n", (finish-start) * pow(10,6),MPI_Wtick()* pow(10,6));
    return finish-start;
}
return -1;
}

```

9.6 MAIN.C

```

#include <mpi.h>
#include <stdio.h>

//*****
//*****//
#include "config.h"
#include "benchmarks.h"
//*****
//*****//

#define pingPoingFlag    1
#define sendReceiveFlag 1
#define broadcastFlag    1
#define scaterFlag       1
#define gatherFlag       1
#define reduceFlag       1
#define allToAllFlag     1
#define barrierFlag      1

void pingPongMain(int world_rank, int world_size);
void sendReceiveMain(int world_rank, int world_size);
void broadcastMain(int world_rank, int world_size);
void ScatterMain(int world_rank, int world_size);
void gatherMain(int world_rank, int world_size);
void reduceMain(int world_rank, int world_size);
void allToAllMain(int world_rank, int world_size);
void barrierMain(int world_rank,int world_size);

void printHeader();

int main (int argc, char** argv) {

```



```

MPI_Init(&argc,&argv);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

MPI_Barrier(MPI_COMM_WORLD);
if(world_rank == 0){

    normalizedTime();
}
initBytesAndTimes();
MPI_Barrier(MPI_COMM_WORLD);

if(pingPongFlag){
    pingPongMain(world_rank,world_size);
}

if(sendReceiveFlag){
    sendReceiveMain(world_rank,world_size);
}
if(broadcastFlag){
    broadcastMain(world_rank,world_size);
}
if(scaterFlag){
    ScatterMain(world_rank,world_size);
}
if(gatherFlag){
    gatherMain(world_rank,world_size);
}
if(reduceFlag){
    reduceMain(world_rank,world_size);
}
if(allToAllFlag){
    allToAllMain(world_rank, world_size);
}
if(barrierFlag){
    barrierMain(world_rank,world_size);
}

MPI_Finalize();
}

void pingPongMain(int world_rank,int world_size) {
    if(world_size == 2){
        if(world_rank==0){
            printf("\n\nPingPong; N° Process; %d ;\n\n",world_size);
            printHeader();
        }
        int i;
        for(i = 0; i < SIZE ; i++){
            pingPong(times[i],bytes[i]);
        }
    }
}

void sendReceiveMain(int world_rank, int world_size){
    if(world_rank==0){

```

```

        printf("\n\nSendReceive; N° Process; %d
;\n\n",world_size);
        printHeader();
    }
    int i;
    for(i = 0; i < SIZE ; i++){
        sendReceive(times[i],bytes[i]);
    }
}

void broadcastMain(int world_rank, int world_size){
    if(world_rank==0){
        printf("\n\nBroadcast; N° Process; %d ;\n\n",world_size);
        printHeader();
    }
    int i;
    for(i = 0; i < SIZE ; i++){
        broadcast(times[i],bytes[i]);
    }
}

void ScatterMain(int world_rank, int world_size){
    if(world_rank==0){
        printf("\n\nScater; N° Process; %d ;\n\n",world_size);
        printHeader();
    }
    int i;
    for(i = 0; i < SIZE ; i++){
        scater(times[i],bytes[i]);
    }
}

void gatherMain(int world_rank, int world_size){
    if(world_rank==0){
        printf("\n\nGather; N° Process; %d ;\n\n",world_size);
        printHeader();
    }
    int i;
    for(i = 0; i < SIZE ; i++){
        gather(times[i],bytes[i]);
    }
}

void reduceMain(int world_rank, int world_size){
    if(world_rank==0){
        printf("\n\nReduce; N° Process; %d ;\n\n",world_size);
        printHeader();
    }
    int i;
    for(i = 0; i < SIZE ; i++){
        reduce(times[i],bytes[i]);
    }
}

void allToAllMain(int world_rank, int world_size){
    if(world_rank==0){
        printf("\n\nAllToAll; N° Process; %d ;\n\n",world_size);
        printHeader();
    }
}

```

```

    }
    int i;
    for(i = 0; i < SIZE ; i++){
        allToAll(times[i],bytes[i]);
    }
}

void barrierMain(int world_rank,int world_size){
    if(world_rank==0){
        printf("\n\nBarrier; N° Process; %d ;\n\n",world_size);
        printHeader();
    }
    int i;
    for(i = 0; i < SIZE ; i++){
        barrier(times[i],bytes[i]);
    }
}

void printHeader(){
    printf("Bytes; Times; Latency(ys); Bandwidth(MBps); \n");
}

```