Ejercicio 4:

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

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

	int MyProc, tag=1, nProcs, i;
	char msg='A', msg_recpt;
	MPI_Request request;
	MPI_Status status;

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);

	// Getting number of threads
	MPI_Comm_size(MPI_COMM_WORLD, &nProcs);

	// Sending a message to the rest of threads
	for (i=0; i<nProcs; i++) {

		// Excluding sending the message to itself
		if (MyProc != i) {
			printf("Proc #%d sending message to Proc #%d\n", MyProc, i);
			MPI_Isend(&msg, 1, MPI_CHAR, i, tag, MPI_COMM_WORLD, &request);
		}
	}

	// Receiving a message from the rest of threads
	for (i=0; i<nProcs; i++) {

		// Excluding receiving the message from itself
		if (MyProc != i) {
			printf("Proc #%d received message from Proc #%d\n", MyProc, i);
			MPI_Irecv(&msg_recpt, 1, MPI_CHAR, i, tag, MPI_COMM_WORLD, &request);
		}
	}

	//MPI_Barrier or waits can be considered
	MPI_Finalize();

	return 0;
}
```

Ejercicio 4 (alternativa):

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

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

        int current, total, i, tag=1;
        char msg='A', msg_recpt;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &total);
        MPI_Comm_rank(MPI_COMM_WORLD, &current);

        printf("Process # %d started \n", current);
        MPI_Barrier(MPI_COMM_WORLD);


        for(i = 0; i<total ; i++){
                if(i < current){
                        MPI_Recv(&msg_recpt, 1, MPI_CHAR, i, tag, MPI_COMM_WORLD, &status);
                        printf("Proc #%d received message from Proc #%d\n", current, i) ;
                }
                if(i > current){
                        printf("Proc #%d sending message to Proc #%d\n", current, i) ;
                        MPI_Send(&msg, 1, MPI_CHAR, i, tag, MPI_COMM_WORLD);
                }
        }

        for(i = 0; i<total ; i++){
                if(i < current){
                        printf("Proc #%d sending message to Proc #%d\n", current, i) ;
                        MPI_Send(&msg, 1, MPI_CHAR, i, tag, MPI_COMM_WORLD);
                }
                if(i > current){
                        MPI_Recv(&msg_recpt, 1, MPI_CHAR, i, tag, MPI_COMM_WORLD, &status);
                        printf("Proc #%d received message from Proc #%d\n", current, i) ;
                }
        }



        printf("Finishing proc %d\n", current);

        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
}
```

Ejercicio 5:

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

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

  int MyProc, tag=1, size;
  char msg='A', msg_recpt ;
  MPI_Status *status ;
  int left, right ;


  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);
  MPI_Comm_size(MPI_COMM_WORLD, &size);


  left = (MyProc + size - 1) % size;
  right = (MyProc + 1) % size;

  if (MyProc == 0)
  {
    printf("Proc %d sending message to proc %d \n", MyProc, right);

    MPI_Send(&msg, 1, MPI_CHAR, right, 1, MPI_COMM_WORLD);
    MPI_Recv(&msg_recpt, 1, MPI_CHAR, left, 1, MPI_COMM_WORLD, status);
  }
  else
  {
    MPI_Recv(&msg_recpt, 1, MPI_CHAR, left, 1, MPI_COMM_WORLD, status);
    MPI_Send(&msg, 1, MPI_CHAR, right, 1, MPI_COMM_WORLD);
    printf("Proc %d : received message from proc %d and sending message to %d\n", MyProc, left, right);
  }

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Ejercicio 7a

```
#include<stdio.h>
#include<mpi.h>
#include<stdlib.h>
#include<time.h>

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

  int i, sum=0, grand_sum=0, n=1000;
  MPI_Status status;

  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);

  srand(time(NULL)+world_rank);
  for(i=0; i<n; i++)
  {
   sum += rand() % 100;
  }

  if (world_rank == 0)
  {
    grand_sum += sum;
    for (i = 1; i < world_size; ++i)
    {
       MPI_Recv(&sum, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
       grand_sum += sum;
    }
    printf("%d", grand_sum);
  }
  else
  {
    MPI_Send(&sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }

  MPI_Finalize();
}
```

Ejercicio 7b

```
#include<stdio.h>
#include<mpi.h>
#include<stdlib.h>
#include<time.h>

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

  int i, sum=0, n=1000, sender=0;
  int *rbuf;

  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);

  srand(time(NULL)+world_rank);
  if (world_rank == 0)
  {
    rbuf = (int *) malloc(world_size * 1 * sizeof(int));
  }
  for(i=0; i<n; i++)
  {
   sender += rand() % 100;
  }

  MPI_Gather(&sender , 1 , MPI_INT , rbuf , 1 , MPI_INT , 0 , MPI_COMM_WORLD);
  if (world_rank == 0)
  {
    for (i = 0; i < world_size; ++i)
    {
       sum += rbuf[i];
    }
    printf("%d", sum);
  }

  MPI_Finalize();
}
```

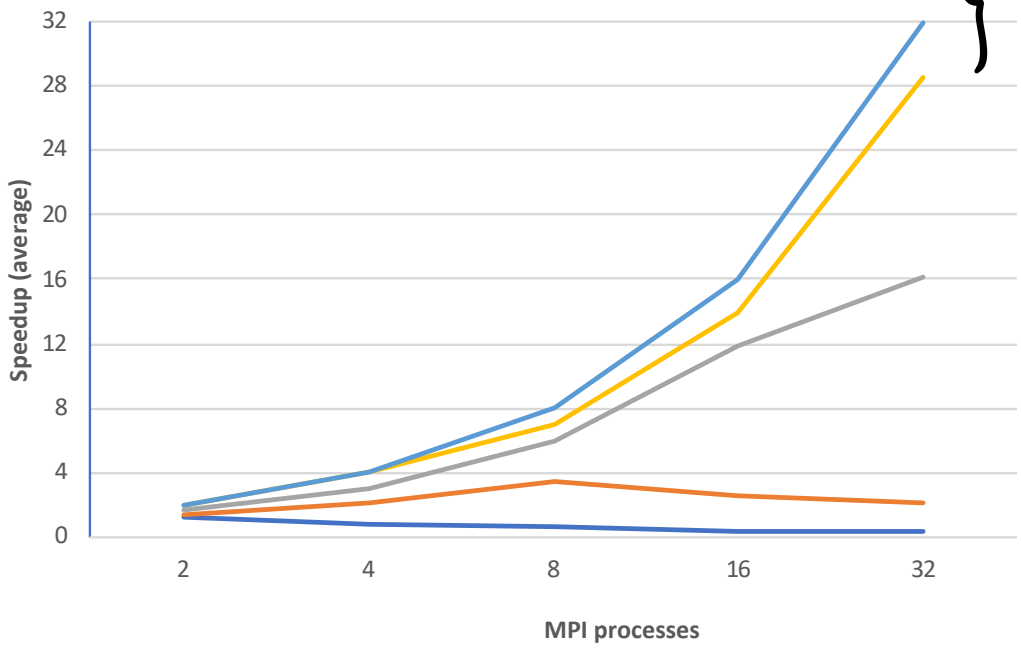
Ejercicio 7c

```
#include<stdio.h>
#include<mpi.h>
#include<stdlib.h>
#include<time.h>

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

  int i, sum=0, grand_sum=0, n=1000;

  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);

  srand(time(NULL)+world_rank);
  for(i=0; i<n; i++)
  {
   sum += rand() % 100;
  }

  MPI_Reduce(&sum, &grand_sum, 1 , MPI_INT , MPI_SUM , 0 , MPI_COMM_WORLD);
  if (world_rank == 0)
  {
    printf("%d", grand_sum);
  }

  MPI_Finalize();
}
```

Ejercicio 8

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

double f(double x)
{
	return(4.0/(1.0+x*x));
}

double fragment(double a, double b, double num_fragments, double h)
{
	double est, x;
	int i;

	est = (f(a) + f(b))/2.0;
	for (i=1; i<=num_fragments-1; i++){
		x = a + i*h;
		est += f(x);
	}
	est = est*h;
	return est;
}

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

	int i, rank, numprocs, root=0;
	double dx, n=1000000000.0, a=0.0, b=1.0, h=0.0;
	double total, result=0.0;

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

	/* All processes */
	dx = (b - a) / numprocs;
	h = (b - a) / n;
	result = fragment(dx*rank, dx+dx*rank, n / numprocs, h);

	MPI_Reduce(&result, &total, 1, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);

	if (rank == root) {
		printf("Result: %.20f\n", total);
	}

	MPI_Finalize();
}
```

Ejercicio 9:



ALMOST LINEAR SCALABILITY

DO NOT SCALE (SMALL SIZE)

#NODES : 1 | 2 | 4 | 8

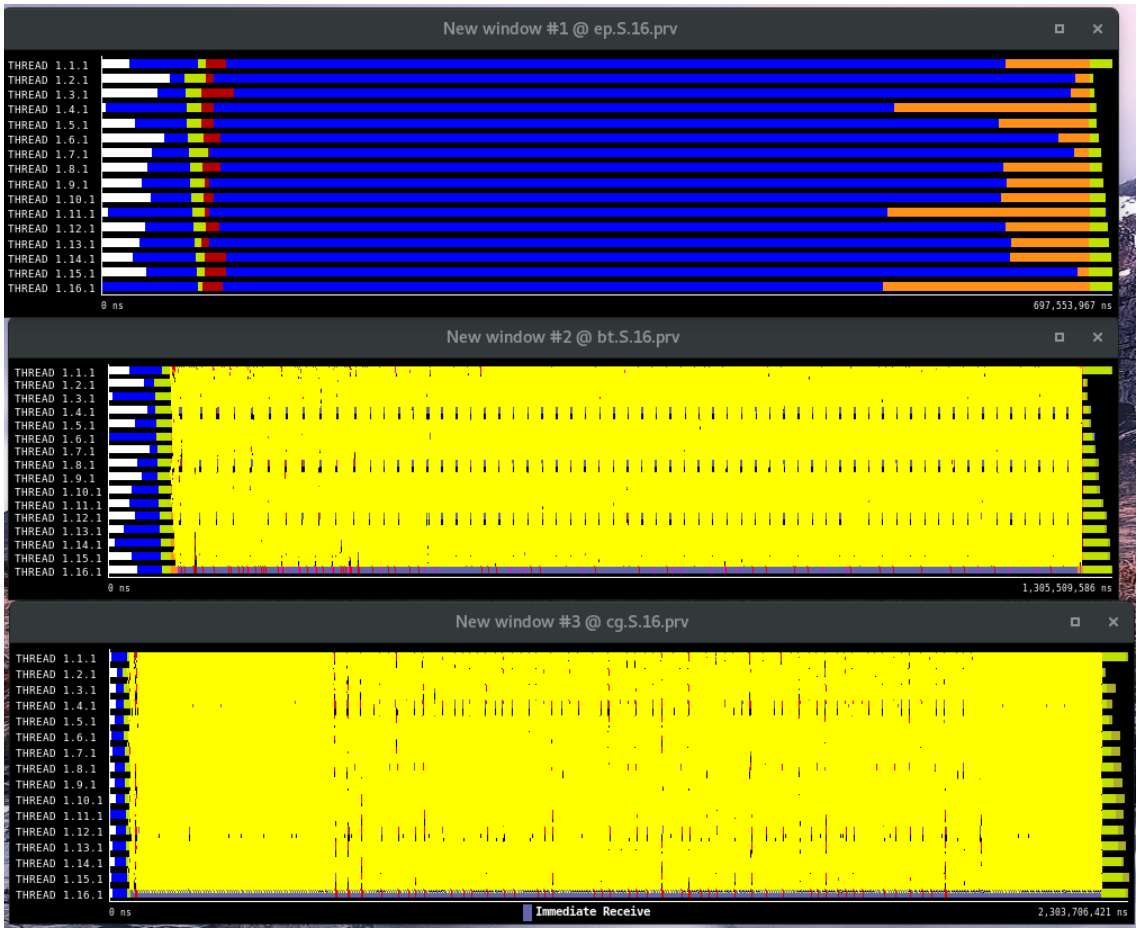
En el video publicado se hizo una breve demostración del uso de las herramientas y de las principales diferencias entre las dos implementaciones del solver con MPI (una con llamadas síncronas y otras con llamadas asíncronas) usando TAU/jumpshot.

Ejemplo con extrae/paraver

Hay diferencias importantes entre los tres benchmarks seleccionados, especialmente en el patrón de paso de mensajes.

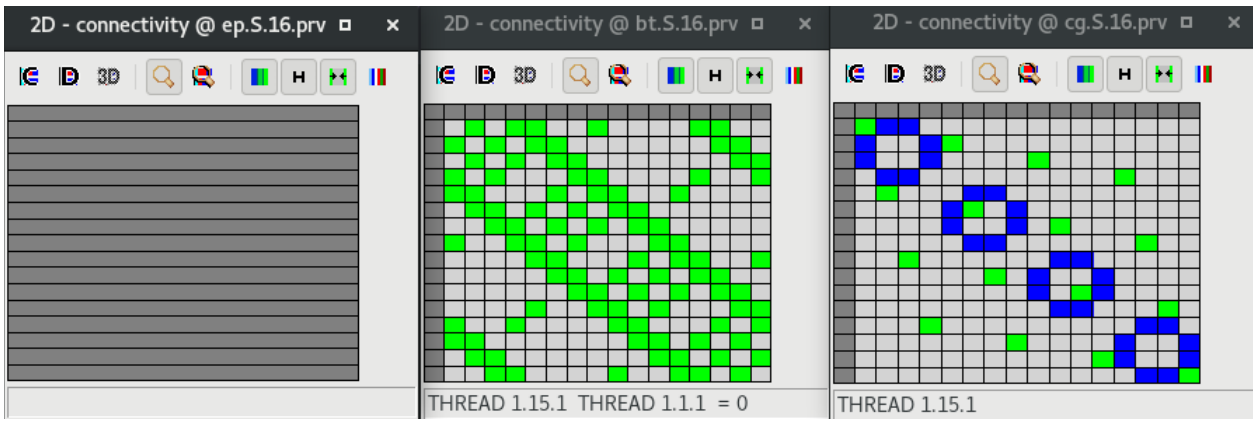
Algunos ejemplos se muestran a continuación:

Vista general:



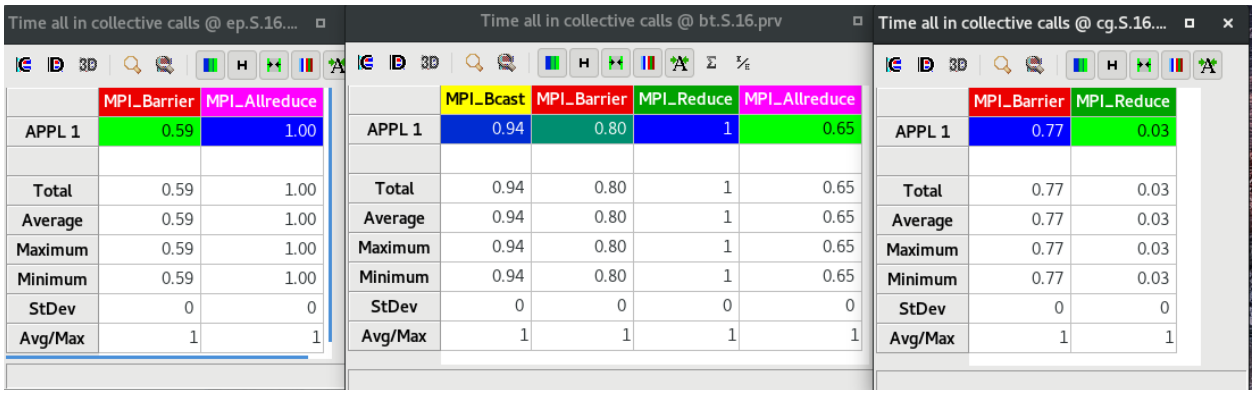
Llamadas MPI

Se muestra a continuación el patrón del paso de mensajes puto a punto:



Es interesante ver que en BT el paso de mensajes entre nodos es más frecuentes y en CG se observa patrón donde hay comunicación elevada entre procesos de un mismo nodo.

Llamadas colectivas:



Time all in collective calls @ ep.S.16....

	MPI_Barrier	MPI_Allreduce
APPL 1	0.59	1.00
Total	0.59	1.00
Average	0.59	1.00
Maximum	0.59	1.00
Minimum	0.59	1.00
StDev	0	0
Avg/Max	1	1

Time all in collective calls @ bt.S.16.prv

	MPI_Bcast	MPI_Barrier	MPI_Reduce	MPI_Allreduce
APPL 1	0.94	0.80	1	0.65
Total	0.94	0.80	1	0.65
Average	0.94	0.80	1	0.65
Maximum	0.94	0.80	1	0.65
Minimum	0.94	0.80	1	0.65
StDev	0	0	0	0
Avg/Max	1	1	1	1

Time all in collective calls @ cg.S.16....

	MPI_Barrier	MPI_Reduce
APPL 1	0.77	0.03
Total	0.77	0.03
Average	0.77	0.03
Maximum	0.77	0.03
Minimum	0.77	0.03
StDev	0	0
Avg/Max	1	1