Práctica 4
Introducción a la programación MPI

prescalar.c

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

#define MAXV 10000

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

  int idproc, numprocs;
  double a[MAXV], b[MAXV], /* vectores operando */
      prod,                /* producto escalar */
      datoa, datob,        /* datos recibidos en cada proc */
      dato;                /* producto enviado al 0 */
  int vsize,               /* tam vectores */
      cadaproc,            /* lo que le toca a cada proc */
      resto,               /* el resto lo har'el cero */
      i, j, k;
  MPI_Status status;

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

  if (idproc == 0) {
    printf("Tam. vectores (menor que %d):", MAXV);
    scanf("%d", &vsize);
    printf("\nGenerando datos...\n");
    for (i = 0; i < vsize; i++) {
      a[i] = (double)i;
      b[i] = (double)i;
    }
    cadaproc = vsize / (numprocs - 1);
    resto = vsize % (numprocs - 1);

    k = 0;
    for (i = 1; i < numprocs; i++) {
      /* Enviamos a cada procesador cuantos datos le tocan */
      /*  Send dato: ptr  --tamaño-- dst tag communicator  */
```

```
      MPI_Send(&cadaproc, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
      printf("Send: src %d dst %d dato %d\n", idproc, i, cadaproc);
      for (j = 0; j < cadaproc; j++) {
        /* Enviamos a cada procesador los datos que le tocan */
        MPI_Send(&a[k], 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
        MPI_Send(&b[k], 1, MPI_DOUBLE, i, 2 MPI_COMM_WORLD);
        printf("Send: src %d dst %d dato %d\n", idproc, i, a[k]);
        printf("Send: src %d dst %d dato %d\n", idproc, i, b[k]);
        k++;
      }
    }
    /* el resto lo hace el cero */
    prod = 0;
    for (j = 0; j < resto; j++) {
      prod = prod + (a[k] * b[k]);
      k++;
    }

    for (i = 1; i < numprocs; i++) {
      /* Recibimos de cada procesador su resultado */
      MPI_Recv(&dato, 1, MPI_DOUBLE, i, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Recv: dst %d src %d numfila %d\n", idproc,
status.MPI_SOURCE,
             dato);
      prod = prod + dato;
    }
    printf("El producto escalar es %f\n", prod);
  } else {
    /* Recibimos cuantos datos nos tocan                              */
    /*  Recv dato: ptr  --tamaño-- dst tag communicator    status  */
    MPI_Recv(&cadaproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf("Recv: dst %d src %d numfila %d\n", idproc,
status.MPI_SOURCE,
           cadaproc);
    prod = 0;
    for (j = 0; j < cadaproc; j++) {
      /* Recibimos cada elemento del vector que nos toca, multiplicamos
y
       * acumulamos */
      MPI_Recv(&datoa, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      MPI_Recv(&datob, 1, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

      printf("Recv: dst %d src %d numfila %d\n", idproc,
```

```
status.MPI_SOURCE,
             datoa);
      printf("Recv: dst %d src %d numfila %d\n", idproc,
status.MPI_SOURCE,
             datob);
      prod = prod + datoa * datob;
    }
    MPI_Send(&prod, 1, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
    printf("Send: src %d dst %d dato %d\n", idproc, 0, prod);
  }

  MPI_Finalize();
  return 0;
}
```

prescalar-bloques.c

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

#define MAXV 10000

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

  int idproc, numprocs;
  double a[MAXV], b[MAXV],      /* vectores operando */
      prod,                     /* producto escalar */
      datoa[MAXV], datob[MAXV], /* datos recibidos en cada proc */
      dato;                     /* producto enviado al 0 */
  int vsize,                    /* tam vectores */
      cadaproc,                 /* lo que le toca a cada proc */
      resto,                    /* el resto lo hará el cero */
      i, j, k;
  MPI_Status status;

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

  if (idproc == 0) {
    printf("Tam. vectores (menor que %d):", MAXV);
    scanf("%d", &vsize);
    printf("\nGenerando datos...\n");
    for (i = 0; i < vsize; i++) {
```

```
      a[i] = (double)i;
      b[i] = (double)i;
    }
    cadaproc = vsize / (numprocs - 1);
    resto = vsize % (numprocs - 1);

    k = 0;
    for (i = 1; i < numprocs; i++) {
      /* Enviamos a cada procesador cuantos datos le tocan */
      MPI_Send(&cadaproc, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
      printf("Send: src %d dst %d dato %d\n", idproc, i, cadaproc);
      /* Enviamos a cada procesador los datos que le tocan */
      MPI_Send(&a[k], cadaproc, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
      MPI_Send(&b[k], cadaproc, MPI_DOUBLE, i, 2, MPI_COMM_WORLD);
      printf("Send: src %d dst %d dato %f\n", idproc, i, a[k]);
      printf("Send: src %d dst %d dato %f\n", idproc, i, b[k]);
      k = k + cadaproc;
    }

    /* el resto lo hace el cero */
    prod = 0;
    for (j = 0; j < resto; j++) {
      prod = prod + (a[k] * b[k]);
      k++;
    }

    for (i = 1; i < numprocs; i++) {
      /* Recibimos de cada procesador su resultado */
      MPI_Recv(&dato, 1, MPI_DOUBLE, i, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Recv: dst %d src %d numfila %f\n", idproc,
status.MPI_SOURCE,
             dato);
      prod = prod + dato;
    }
    printf("El producto escalar es %f\n", prod);
  } else {
    /* Recibimos cuantos datos nos tocan                              */
    MPI_Recv(&cadaproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf("Recv: dst %d src %d numfila %d\n", idproc,
status.MPI_SOURCE,
           cadaproc);
    prod = 0;
    /* Recibimos cada elemento del vector que nos toca, multiplicamos y
     * acumulamos */
    MPI_Recv(datoa, cadaproc, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
```

```
            MPI_STATUS_IGNORE);
    MPI_Recv(datob, cadaproc, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);

    printf("Recv: dst %d src %d numfila\n", idproc, status.MPI_SOURCE);
    for (j = 0; j < cadaproc; j++) {
      prod = prod + (datoa[j] * datob[j]);
    }

    MPI_Send(&prod, 1, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
    printf("Send: src %d dst %d dato %f\n", idproc, 0, prod);
  }

  MPI_Finalize();
  return 0;
}
```

prescalar-bloques-bcastreduce.c

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

#define MAXV 10000

int main(int argc, char *argv[]) {
  int idproc, numprocs;
  double a[MAXV], b[MAXV],      /* vectores operando */
      prod_local,               /* producto escalar local */
      prod_total;               /* producto escalar total */
  int vsize,                    /* tam vectores */
      cadaproc,                 /* lo que le toca a cada proc */
      i, j;
  MPI_Status status;

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

  if (idproc == 0) {
    printf("Tam. vectores (menor que %d): ", MAXV);
    scanf("%d", &vsize);
    printf("\nGenerando datos...\n");
```

```
    for (i = 0; i < vsize; i++) {
      a[i] = (double)i;
      b[i] = (double)i;
    }
  }

  // Difusión del tamaño del vector a todos los procesos
  MPI_Bcast(&vsize, 1, MPI_INT, 0, MPI_COMM_WORLD);

  // Asegúrate de que todos los procesos tengan el tamaño antes de
continuar
  MPI_Barrier(MPI_COMM_WORLD);

  if (vsize > MAXV) vsize = MAXV; // Asegurar que no exceda el máximo

  cadaproc = vsize / numprocs; // Dividir el trabajo equitativamente
  int inicio = idproc * cadaproc;
  int fin = (idproc + 1) * cadaproc;
  if (idproc == numprocs - 1) fin = vsize; // El último toma el resto

  // El proceso raíz ya tiene los vectores, los distribuye si es
necesario
  if (idproc == 0) {
    for (i = 1; i < numprocs; i++) {
      inicio = i * cadaproc;
      fin = (i + 1) * cadaproc;
      if (i == numprocs - 1) fin = vsize;
      MPI_Send(&a[inicio], fin-inicio, MPI_DOUBLE, i, 0,
MPI_COMM_WORLD);
      MPI_Send(&b[inicio], fin-inicio, MPI_DOUBLE, i, 1,
MPI_COMM_WORLD);
    }
  } else {
    MPI_Recv(a, fin-inicio, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    MPI_Recv(b, fin-inicio, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
  }

  // Calcula el producto escalar local
  prod_local = 0;
  for (i = 0; i < fin-inicio; i++) {
    prod_local += a[i] * b[i];
  }

  // Reduce todos los productos escalares locales al total en el proceso
```

```
raíz
  MPI_Reduce(&prod_local, &prod_total, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

  // Imprimir el resultado en el proceso raíz
  if (idproc == 0) {
    printf("El producto escalar es %f\n", prod_total);
  }

  MPI_Finalize();
  return 0;
}
```