Práctica 5
Introducción a la programación OpenMP

pi1.c - Sección Critica

```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"


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

  double area, pi, x, valor;
  int i,n,tid,num_threads;
  double empieza, termina;


    if (argc != 3) {
//    scanf("%d",&n);
      printf ("Uso: pi n_iteraciones n_hilos\n");
      exit(1);
    } else {
      n=atoi(argv[1]);
      num_threads=atoi(argv[2]);
    }
    omp_set_num_threads(num_threads);
    empieza = omp_get_wtime();
    area = 0.0;

  #pragma omp parallel for private(x, valor)
    for(i=0; i<n; i++) {
        x = (i+0.5)/n;
        valor= 4.0/(1.0+x*x);
      #pragma omp critical
          area += valor;
      }
    pi = area/n;
    printf ("PI=%f\n", pi);

    termina = omp_get_wtime();
    printf("Hilos %d. Tiempo=%lf\n", num_threads, termina-empieza);
//  }
  return 0;
}
```

pi2.c - Operación atomica

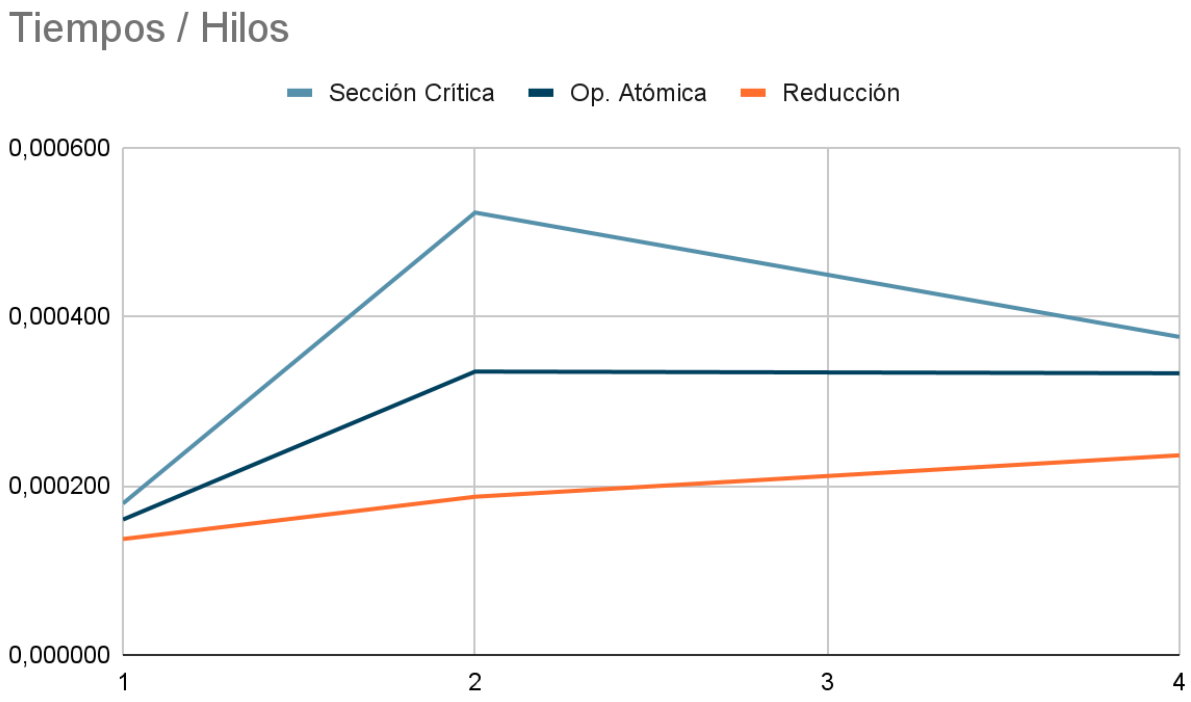
```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"


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

  double area, pi, x, valor;
  int i,n,tid,num_threads;
  double empieza, termina;


    if (argc != 3) {
//    scanf("%d",&n);
      printf ("Uso: pi n_iteraciones n_hilos\n");
      exit(1);
    } else {
      n=atoi(argv[1]);
      num_threads=atoi(argv[2]);
    }
    omp_set_num_threads(num_threads);
    empieza = omp_get_wtime();
    area = 0.0;

  #pragma omp parallel for private(x, valor)
    for(i=0; i<n; i++) {
        x = (i+0.5)/n;
        valor= 4.0/(1.0+x*x);
      #pragma omp atomic
          area += valor;
      }
    pi = area/n;
    printf ("PI=%f\n", pi);

    termina = omp_get_wtime();
    printf("Hilos %d. Tiempo=%lf\n", num_threads, termina-empieza);

//  }
  return 0;
}
```

pi3.c - Reducción

```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"


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

  double area, pi, x, valor;
  int i,n,tid,num_threads;
  double empieza, termina;


    if (argc != 3) {
//    scanf("%d",&n);
      printf ("Uso: pi n_iteraciones n_hilos\n");
      exit(1);
    } else {
      n=atoi(argv[1]);
      num_threads=atoi(argv[2]);
    }
    omp_set_num_threads(num_threads);
    empieza = omp_get_wtime();
    area = 0.0;

  #pragma omp parallel for private(x, valor) reduction(+:area)
    for(i=0; i<n; i++) {
        x = (i+0.5)/n;
        valor= 4.0/(1.0+x*x);
        area += valor;
    }
    pi = area/n;
    printf ("PI=%f\n", pi);

    termina = omp_get_wtime();
    printf("Hilos %d. Tiempo=%lf\n", num_threads, termina-empieza);

//  }
  return 0;
}
```

Se efectúa una comparación los tiempos de ejecución obtenidos en los tres casos, variando el número de hilos, se obtiene la siguiente tabla:

Número hilo	Sección Crítica	Op. Atómica	Reducción
1	0.000179	0.000160	0.000137
2	0.000523	0.000335	0.000187
4	0.000376	0.000333	0.000236



prescalar.c

```c
#include "omp.h"
#include <stdio.h>
#define MAXV 100000

int main( int argc, char *argv[] )
{
  double empieza, termina;

      int idproc, numprocs;
      double a[MAXV], b[MAXV], /* vectores operando */
            prod; /* producto escalar */
      int vsize, /* tamao vectores */
    i;

   if (argc != 2) {
      scanf("%d",&vsize);
  }else{
      vsize=atoi(argv[1]);
  }
      printf("Tam. vectores (menor que %d): %d",MAXV,vsize);
      numprocs = omp_get_num_procs();
      printf("\nGenerando datos...\n");
      for (i=0; i<vsize; i++) {
      a[i]=(double) i;
     b[i]=(double) i;
      }
      empieza = omp_get_wtime();
      prod = 0;
      #pragma omp parallel for private(idproc) reduction(+:prod)
      for (i=0; i<vsize; i++) {
      idproc = omp_get_thread_num() ;
//          printf("Soy %d, ejecuto %d\n",idproc,i);
      prod = prod+(a[i]*b[i]);
      }
      printf("El producto escalar es %f\n", prod );
      termina = omp_get_wtime();
      printf("\nTiempo=%lf\n", termina-empieza);

      return 0;
}
```

ranksort.c

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

#define MAXV 100000

      double values[MAXV], final[MAXV]; /* vectores operando */
      int vsize; /* tam vectores */
      int i;


void PutInPlace (int src)
{
      double testval;
      int j, rank;

      testval = values[src];
      j = src;
      rank = -1;
      do {
            j = (j+1) % vsize;
            if (testval >= values[j])
                  rank = rank+1;
      } while (j!=src);
      final[rank] = testval;
}


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

      int idproc, numprocs;
      double empieza, termina;

      if (argc != 2) {
            printf("Tamaño del vector: ");
            scanf("%d",&vsize);
      }else{
            vsize=atoi(argv[1]);
      }
```

```
      numprocs = omp_get_num_procs();
      printf("Tam. vectores (menor que %d): %d",MAXV,vsize);
        // scanf("%d", &vsize);
            printf("\nGenerando datos...\n");
            for (i=0; i<vsize; i++) {
            values[i]= (double) random();
            printf("values[%d]=%lf\n",i,values[i]);
            }
            printf("\nOrdenando datos...\n");
            empieza = omp_get_wtime();

            #pragma omp parallel for private(idproc)
            for (i=0; i<vsize; i++) {
            idproc = omp_get_thread_num() ;
                  // printf("Soy %d, ejecuto %d\n",idproc,i);
                  PutInPlace (i);
            }

            termina = omp_get_wtime();

            printf("\nTiempo=%lf\n", termina-empieza);

            printf("\nResultado...\n");
            for (i=0; i<vsize; i++) {
            printf("final[%d]=%lf\n",i,final[i]);
      }
      return 0;
}
```

prescalar-bloques.c - Hibrido de MPI / OpenMP

```c
#include "mpi.h"
#include <stdio.h>
#include "omp.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;
    // #pragma omp parallel for reduction(+:prod) private(k)
    // for (j = 0; j < resto; j++) {
    //   prod = prod + (a[k] * b[k]);
    //   k++;
    // } // k += 1

    #pragma omp parallel for reduction(+:prod)
    for (j = k; j < resto; j++) {
      prod = prod + (a[j] * b[j]);
    } // k += 1
    k = j;

    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);
    #pragma omp parallel for reduction(+:prod)
    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;
}
```

```
g1cac24@cac1:~/prac-omp/openmp-files> nano maquinas
g1cac24@cac1:~/prac-omp/openmp-files> lamboot maquinas

LAM 7.1.4/MPI 2 C++/ROMIO - Indiana University

g1cac24@cac1:~/prac-omp/openmp-files> mpirun -np 2 hibrido
Tam. vectores (menor que 10000):20

Generando datos...
Send: src 0 dst 1 dato 20
Send: src 0 dst 1 dato 0.000000
Send: src 0 dst 1 dato 0.000000
Recv: dst 1 src 0 numfila 20
Recv: dst 1 src 0 numfila
Recv: dst 0 src 0 numfila 2470.000000
El producto escalar es 2470.000000
Send: src 1 dst 0 dato 2470.000000
g1cac24@cac1:~/prac-omp/openmp-files>
```