Universidad Internacional de Valencia

Ejercicios de OpenMPI y OpenMP

Paralelismo

Y. Cardinale

Herramientas para programar aplicaciones paralelas: OpenMPI y OpenMP

NOTA: Todos los códigos mostrados o mencionados en en esta actividad están disponibles en el portal de la asignatura en Material del Profesor.

MUESTRE printscreens de las ejecuciones de los programas.

1. Con el siguente programa **piparallel.c**:

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

#define STEPS 5
double f(double a)
{
        return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[] ) {
        int n, myid, numprocs, i, j;
        double PI25DT = 3.14159265358979323846264 3;
        double mypi, pi, h, sum, x;
        double startwtime=0.0, endwtime;
        int namelen;
        char processor_name[MPI_MAX_PROCESSOR_NAME];
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);
        fprintf(stderr,"Process %d running on %s\n", myid, processor_name);
        if (myid == 0) {
                n = 1000000;
                startwtime = MPI_Wtime();
        }
        MPI_Barrier(MPI_COMM_WORLD);
        for (j = 0; j < STEPS; j++) {
                MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
                MPI_Barrier(MPI_COMM_WORLD);
                h = 1.0 / (double) n;
                sum = 0.0;
                for (i = myid + 1; i <= n; i += numprocs)  {
                        x = h * ((double)i - 0.5);
                        sum += f(x);
                }
                mypi = h * sum;
                MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        }
        if (myid == 0)    {
                endwtime = MPI_Wtime();
                printf("pi is approximately %.16f, Error is %.16f\n",pi, fabs(pi - PI25DT));
                printf("wall clock time = %f\n", endwtime-startwtime);
        }
        MPI_Finalize();
        return(0);
```

}

- Revise y explique qué hace el programa (puede hacer diagramas o dibujos)
- Compile y ejecute
- Explique la utilidad de las primitivas de comunicación colectiva.

2. El objetivo de esta ejercicio es comparar el comportamiento de un algoritmo paralelo versus una versión secuencial. El problema que nos interesa es un algoritmo secuencial que cuenta el número de apariciones de un número en un arreglo muy grande.

Para simular una búsqueda en un arreglo mayor, lo que haremos es buscar N veces en la misma estructura de datos.

Versión Secuencial

Un ejemplo de este programa la versión secuencial **CuentaSec.c**:

```
//--------------------------------------------------------------------+
// CuentaSec.c: Cuenta el numero de veces que aparece un numero en  |
//              un vector muy grande.                                 |
//--------------------------------------------------------------------+
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#define MAX_ENTERO      1000
#define NUM_VECTORES    10000 // Simula vector todavia mayor sin ocupar espacio
                              // de memoria
#define NUM_BUSCADO     8
int main (int argc, char *argv[]) {
      struct timeval t0, tf, t;
      int i, j, laCardinalidad, numVeces;
      int *vector;
      if (argc != 2) {
            printf ("Uso: cuentaSec cardinalidad \n");
            return 0;
      }
      laCardinalidad = atoi(argv[1]);
      assert (laCardinalidad > 0);
      assert((vector=(int *)malloc(sizeof(int)*laCardinalidad))!=NULL);
      for (i=0; i<laCardinalidad; i++) {
            vector[i] = random() % MAX_ENTERO;
      }
      assert (gettimeofday (&t0, NULL) == 0);
      numVeces = 0;
      for (i=0; i<NUM_VECTORES; i++)
            for (j=0; j<laCardinalidad; j++)
                  if (vector[j] == NUM_BUSCADO) numVeces++;
      assert (gettimeofday (&tf, NULL) == 0);
      timersub (&tf, &t0, &t);
      printf ("Veces que aparece el %d = %d\n", NUM_BUSCADO, numVeces);
      printf ("Tiempo de proceso: %ld:%3ld(seg:mseg)\n",
                t.tv_sec, t.tv_usec/1000);
      return 0;
}
```

El programa se invoca con un parámetro "*cardinalidad*" que determina el tamaño del arreglo. Si fijamos una cardinalidad de 200.000, tendremos un arreglo de ese tamaño donde se colocan números generados al azar comprendidos entre 0 y 1.000 (MAX_ENTERO). En este arreglo se busca (10.000 veces) el número indicado en la constante NUM_BUSCADO.

Antes de construir una versión paralela de este programa, ejecuten la versión secuencial variando el tamaño del problema. Para ello:

- Generar el ejecutable de **CuentaSec.c**
- Ejecutar el programa variando la cardinalidad desde 100.000 hasta 1.000.000. Anotar en la Tabla 1 el tiempo de ejecución de cada corrida (use gprof y compare con el tiempo retornado por el programa). Es conveniente ejecutarlo más de una vez y tomar el promedio de los tiempos.

Comprobar el efecto de la optimización de código por parte del compilador. Para ello, generar el ejecutable compilando con la opción de optimización del gcc (-O3).

- Generar el ejecutable de **CuentaSec.c** con la opción de optimización.
- Ejecutar el programa cuentaSec variando la cardinalidad desde 100.000 hasta 1.000.000. Anotar en la Tabla 2 el tiempo de ejecución para cada una de las corridas realizadas. Si hubo alguna mejora en el tiempo de ejecución, justifique que pudo ocurrir.

Versión Paralela

Generar una versión paralela sencilla de este programa en la que el maestro reparte el trabajo entre sí mismo y el resto de los procesos. El reparto consistirá en dividir el arreglo en tantos trozos como procesos. Un esqueleto de este programa, al que denominaremos **CuentaPar.c**

```
//------------------------------------------------------------------+
// CuentaPar.c: Cuenta aparaciones de un numero en un arreglo muy   |
//               grande. Version paralela simple                    |
//                           ESQUELETO                              |
//------------------------------------------------------------------+

#include <assert.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>

#include "mpi.h"

#define MAX_ENTERO      1000
#define NUM_VECTORES    10000 // Simula vector todavia mayor
#define NUM_BUSCADO     8



//------------------------------------------------------------------
void esclavo(void) {

}
```

```
//---------------------------------------------------------------------
void maestro (int NumProcesos, int Cardinalidad) {
      int i, totalNumVeces;
      int *vector;
      struct timeval t0, tf, t;

      // Inicializar el vector

      assert((vector =(int *)malloc(sizeof(int)*Cardinalidad))!=NULL);
      for (i=0; i<Cardinalidad; i++)
            vector[i] = random() % MAX_ENTERO;

      assert (gettimeofday (&t0, NULL) == 0);

      // Repartir trabajo

      // Computar mi trozo

      // Recoger resultados

      assert (gettimeofday (&tf, NULL) == 0);

      timersub(&tf, &t0, &t);
      printf ("Numero de veces que aparece el %d = %d\n",
                  NUM_BUSCADO, totalNumVeces);
      printf ("tiempo total = %ld:%3ld\n", t.tv_sec, t.tv_usec/1000);
}

//---------------------------------------------------------------------
int main( int argc, char *argv[] ) {
      int yo, numProcesos;

      if (argc != 2) {
            printf ("Uso: cuentaPar cardinalidad \n");
            return 0;
      }
      laCardinalidad = atoi(argv[1]);

      assert (laCardinalidad > 0);
      setbuf (stdout, NULL);
      MPI_Init (&argc, &argv);
      MPI_Comm_rank (MPI_COMM_WORLD, &yo);
      MPI_Comm_size (MPI_COMM_WORLD, &numProcesos);
      if (yo == 0) maestro(NumProcesos,laCardinalidad);
      else esclavo();
      MPI_Finalize();
      return 0;
}
```

- Completar el código del programa **CuentaPar.c**
- Generar el ejecutable correspondiente usando la librería de mpi
- Ejecutar el programa y anotar en la Tabla 3 el tiempo de ejecución. Ejecútelo más de una vez y tome el promedio de los tiempos.
- Calcular la aceleración obtenida con respecto a la versión secuencial y analizar el resultado (graficar resultados)

- Calcular la eficiencia obtenida (graficar resultados)
- Analizar los resultados obtenidos.
- Elabore un breve informe con los resultados obtenidos y un análisis de los mismos.

Tabla 1. Versión Secuencial

Cardinalidad	Sin Optimización (seg:mseg)	Sin Optimización con gprof(seg:mseg)	Con Optimización O3 (seg:mseg)
100.000			
200.000			
400.000			
600.000			
800.000			
1.000.000			

Tabla 2. Versión Paralela

Cardinalidad	P=2 (seg:mseg)	P=4 (seg:mseg)	P=8 (seg:mseg)	P=16 (seg:mseg)
100.000				
200.000				
400.000				
600.000				
800.000				
1.000.000				

Tabla 3. Aceleración

Cardinalidad	P=2 (seg:mseg)	P=4 (seg:mseg)	P=8 (seg:mseg)	P=16 (seg:mseg)
100.000				
200.000				
400.000				
600.000				
800.000				
1.000.000				

Tabla 4. Eficiencia

Cardinalidad	P=2 (seg:mseg)	P=4 (seg:mseg)	P=8 (seg:mseg)	P=16 (seg:mseg)
100.000				
200.000				
400.000				

600.000				
800.000				
1.000.000				

3. MPI: Comunicaciones Punto a Punto

3.1. Este ejercicio ayuda a visualizar el desempeño relativo de los cuatro modos de comunicación (synchronous, ready, buffered, y standard). El programa **blocksends.c** indica el tiempo (wallclock) empleado en llamadas bloqueantes para los cuatro modos de comunicación.

Todos los *receives* son colocados antes que cualquier mensaje sea enviado, nótese que se obtendrían tiempos diferentes si los *receives* no se colocan así.

- Lea el programa **blocksends.c**
- Compílelo y ejecútelo
- Note el tiempo empleado por el *send* bloqueante en los cuatro modos.

3.2. Este ejercicio permite constatar que las rutinas no bloqueantes son más seguras que las bloqueantes. Use el programa **deadlock.c.**

- Compile el programa **deadlock.c**
- Ejecútelo con dos procesos. ¿Qué sucede?
- Varíe el tamaño del mensaje dentro del programa. Dependiendo del tamaño del mensaje, el programa escribirá unas líneas y luego se detendrá. Aborte el programa con <ctrl. C>.
- ¿Por qué el programa entra en interbloqueo?
- Corrija el programa de manera de evitar el interbloqueo
- Compare su solución con la original dada.

4. MPI: Comunicaciones Colectivas

Se quiere que implemente un programa similar al presentado en **avg.c** y **all_avg.c**, pero en este caso el objetivo será encontrar el valor mínimo, máximo y promedio de los elementos que están contenidos en una matriz de números reales. Tomen en cuenta las siguientes especificaciones:

- El programa debe crear una matriz de dimensión N×N, donde N es el número de procesos a ejecutar. La matriz debe ser llenada con números reales generados aleatoriamente.
- El proceso maestro (o raíz) debe repartir una fila para cada proceso, usando la primitiva **MPI_Scatter** para que cada proceso, incluyéndolo.
- Cada proceso debe procesar la fila que le corresponde para conseguir el valor mínimo, máximo y el promedio de su fila.
- Cada proceso debe mostrar su identificador, junto con el valor mínimo, máximo y promedio de su fila en pantalla, con:

printf(“Process %d with row %d – min: %f; max: %f; avg: %f”, myrank, myrow, mymin, mymax, myavg);

- Luego del cómputo, el proceso raíz o maestro colectará todos los valores de todos los procesos (que vendrán en un arreglo de tres elementos, con el mínimo, el máximo y el promedio respectivamente) mediante la primitiva **MPI_Gather.**
- Para terminar, el proceso raíz toma el contenido de todos los arreglos y encuentra los valores mínimo, máximo y promedio, entre los valores mínimos, máximos y promedios recibidos. Luego de lo cual los mostrará por pantalla.

5. OpenMP: Esta actividad consta de nueve ejercicios sobre el uso de la librería OpenMP para la creación de programas paralelos en memoria compartida.

6.1. Ejecute el programa **e1.c**:

```
/* Programa e1.c */
#include “stdio.h”
#include “stdlib.h”
#include “omp.h”

int main() {
 #pragma omp parallel
   printf(“Hola mundo\n”);
  exit (0);
}
```

¿Cuáles de las instrucciones del tipo #include se pueden eliminar en este programa sin impedir que se ejecute correctamente y por qué pueden ser eliminadas?

6.2. Ejecute el programa **e2.c**:

```
/* Programa e2.c */
#include "stdio.h"
#include “stdlib.h”
#include “omp.h”

int main() {
   int cantidad_hilos=6;
   omp_set_num_threads(cantidad_hilos);
   #pragma omp parallel
     printf("Hola mundo\n");
   exit (0);
}
```

a) Compare la salida con el programa **e2.c**
b) Ejecute el programa **e1.c** con la opción **export OMP_NUM_THREADS=8** y explique.

6.3. Las variables que hayan sido declaradas antes del inicio de la sección paralela, serán compartidas entre todos los hilos de ejecución. Observe el programa **e3.c**

```
/* Programa e3.c */
```

```
#include "stdio.h"
#include "stdlib.h"

int main() {
  int comp=0;
  # pragma omp parallel {
      int priv=0;
      priv++;
      comp++;
      printf("Hola mundo priv %d comp %d\n",priv,comp);
  }
  exit (0);
}
```

La variable **comp** es compartida entre los hilos durante la sección paralela, mientras que la variable **priv** es privada para cada hilo. Así, invariablemente la impresión de los hilos dará como resultado 1 para todas las variables **priv**, mientras que el resultado de las variables **comp** podría ser imprevisible. Explique por qué sucede esto.

6.4. Ejecute el programa **e4.c:**

```
/* Programa e4.c */
#include "stdio.h"
#include "stdlib.h"
#define N 4

float producto(float* a, float* b, int n) {
   float sum = 0.0;
   #pragma omp parallel for shared(sum)
      for (int i=0; i<n; i++){
            sum += a[i] * b[i];
      }
   return sum;
}

int main() {
   float total=0;
   float a[N],b[N];
   int k=0;

   for (k=0;k<N;++k) {
      a[k]=k;
      b[k]=k;
   }
   total = producto(a,b,N);
   printf("%.6f \n",total);

   int s,r,t;
   for (int i=0;i<N;i++){
      s=a[i];
      r=b[i];
      t=a[i] * b[i];
      printf("%d ",r);
      printf(" * %d ",s);
```

```
      printf(" = %d \n",t);
   }
   exit (0);
}
```

a) Qué función cumple la función **shared(sum)** de la directiva **pragma**
b) Ejecute varias veces el programa ¿observa algún error en alguna ejecución? ¿a qué se debe el error?

6.5. Ejecute el programa **e5.c:**

```
/* Programa e5.c */
#include "stdio.h"
#include "stdlib.h"
#define N 4

float producto(float* a, float* b, int n) {
   float sum = 0.0;
   #pragma omp parallel for shared(sum)
      for (int i=0; i<n; i++){
        #pragma omp critical
            sum += a[i] * b[i];
      }
   return sum;
}

int main() {
   float total=0;
   float a[N],b[N];
   int k=0;

   for (k=0;k<N;++k) {
      a[k]=k;
      b[k]=k;
   }
   total = producto(a,b,N);
   printf("%.6f \n",total);

   int s,r,t;
   for (int i=0;i<N;i++){
      s=a[i];
      r=b[i];
      t=a[i] * b[i];
      printf("%d ",r);
      printf(" * %d ",s);
      printf(" = %d \n",t);
   }
   exit (0);
}
```

a) Qué función cumple la directiva **#pragma omp critical**
b) Ejecute varias veces el programa ¿observa algún error en alguna ejecución? ¿a qué se debe el error?

6.6. Ejecute el programa **e6.c:**

```
/* Programa e6.c */
#include "stdio.h"
#include "stdlib.h"
#define N 1000

float producto(float* a, float* b, int n) {
   float sum = 0.0;
   #pragma omp parallel for reduction(+:sum)
      for (int i=0; i<n; i++){
            sum += a[i] * b[i];
      }
   return sum;
}

int main() {
   float total=0;
   float a[N],b[N];
   int k=0;

   for (k=0;k<N;++k) {
      a[k]=k;
      b[k]=k;
   }
   total = producto(a,b,N);
   printf("%.6f \n",total);

   int s,r,t;
   for (int i=0;i<N;i++){
      s=a[i];
      r=b[i];
      t=a[i] * b[i];
      printf("%d ",r);
      printf(" * %d ",s);
      printf(" = %d \n",t);
   }
   exit (0);
}
```

¿Qué ventaja representa el uso de **reduction(+:sum)** con respecto a los códigos e4.c y e5.c ?

6.7. Ejecute el programa **e7.c:**

```
/* Programa e7.c */
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc,char **argv){
   int i; int nodo,numnodos;
   int tam=32;

   MPI_Init(&argc,&argv);
```

```
   MPI_Comm_rank(MPI_COMM_WORLD,&nodo);
   MPI_Comm_size(MPI_COMM_WORLD,&numnodos);

   MPI_Bcast(&tam, 1, MPI_INT, 0, MPI_COMM_WORLD);
   #pragma omp parallel
      printf("Soy el hilo %d de %d hilos dentro del procesador %d de %d
              procesadores\n",\
              omp_get_thread_num(),omp_get_num_threads(),nodo,numnodos);
   MPI_Finalize();
}
```

Explique el funcionamiento de este programa. Puede usar dibujos ilustrativos.

6.8. Ejecute y analice el programa **e8-serial.c:**

```
/* Programa e8-serial.c */
static long num_steps=100000;
double step, pi;
void main() {
   int i; double x, sum = 0.0;
   step = 1.0/(double) num_steps;
   for (i=0; i< num_steps; i++) {
      x = (i+0.5)*step; sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

Paralelice el código usando OpenMP y responda las siguientes preguntas:
1. ¿Cuáles variables deben ser compartidas y cuáles privadas?
2. ¿Debe haber secciones críticas?
3. De no haberlas, ¿puede resolverse usando otra técnica?

6.9. Ejecute y analice el programa **e9-serial.c:**

```
/* Programa e9-serial.c */
#include <stdio.h>
#include <stdlib.h>
#define N 5

int main () {
   /* DECLARACION DE VARIABLES */
   float A[N][N], B[N][N], C[N][N]; // declaracion de matrices de tamaño NxN
   int i, j, m; // indices para la multiplicacion de matrices
   /* LLENAR LAS MATRICES CON NUMEROS ALEATORIOS */
   srand ( time(NULL) );
   for(i=0;i<N;i++) {
      for(j=0;j<N;j++) {
         A[i][j]= (rand()%10);
         B[i][j]= (rand()%10);
      }
   }
   /* MULTIPLICACION DE LAS MATRICES */
   for(i=0;i<N;i++) {
```

```
      for(j=0;j<N;j++) {
         C[i][j]=0.; // colocar el valor inicial para el componente C[i][j] = 0
         for(m=0;m<N;m++) {
            C[i][j]=A[i][m]*B[m][j]+C[i][j];
         }
       printf("C: %f ",C[i][j]);
      }
      printf("  \n");
    }
   /* FINALIZAR EL PROGRAMA */
   return 0;
}
```

Implemente una versión paralela usando OpenMP.