Universitat de Lleida

GRAU en ENGINYERIA INFORMÀTICA

ESCOLA POLITÈCNICA SUPERIOR

CURS 2018/2019

Práctica MPI
Implementación Aplicación Paralela

Alumnes:
Radu Spaimoc
Luis Rosales Carrera

09/01/19

Universitat de Lleida

1 Decisiones de Diseño

Para implementar la practica se ha seguido el código secuencial adjunto. Primeramente se comprueba utilizando "MPI_Comm_size" el numero de procesadores, si este es 1, se lee la matriz i se la lógica del juego el numero de iteraciones por defecto o introducidas por el usuario.

```
if(size == 1)
{
  int *s = ( int * ) malloc ( n * m * sizeof ( int ) );
  for ( it = 0; it <= it_max; it++ )
  {
    if(it != 0)
    {
      s = states_workout( m, n, grid, SEQUENTIAL );
      life_update_sub_array( m, n, grid, s, SEQUENTIAL );
    }
    if ((it%DNumIterForPartialResults)==0)
    {
      sprintf( output_filename, DDefaultOutputFilename, it );
      life_write( output_filename, m, n, size, balance, grid, rank, SEQUENTIAL);

    }
  }
  free(s);
```

En cambio si hay más de un procesador, lo primero que se hace es dividir el numero de filas de la matriz introducida entre el numero de procesadores. Para hacer el balanceo mas equitativo posible, se van repartiendo las filas de una en una. Tal como se puede ver en la siguiente captura:

```
void divide_rows(int size, int m, int n)
{
  int i, aux;

  balance = ( int * ) malloc ( m  * sizeof ( int ) );
  displs = ( int * ) malloc ( m  * sizeof ( int ) );

  for (i = 0; i < size; i++) {
    balance[i] = 0;
  }
  i = 0;
  while(m != 0)
  {
    balance[i] += 1;
    m--;
    i++;
    if(i == size)
      i=0;
  }
  aux = 0;
  for (i = 0; i < size; i++) {
    displs[i] = aux;
    balance[i] = balance[i] * n;
    aux += balance[i];
  }
```

Una de las decisiones más importantes para implementar la práctica, ha sido decidir la forma de repartir las submatrizes a cada proceso, para hacerlo se ha utilizado una comunicación global utilizando "MPI_Scatterv" calculando los desplazamientos en la función anterior en la que se dividen

las filas.

```
divide_rows( size, n, m );
rbuf = ( int * ) malloc(balance[rank]  * sizeof(int));
MPI_Scatterv( grid, balance, displs, MPI_INT, rbuf, balance[rank], MPI_INT, 0, MPI_COMM_WORLD );
```

Seguidamente cada proceso para calcular la siguiente iteración necesita tener la fila/frontera superior y inferior perteneciente a otros procesos adyacentes, a la vez que estos necesitan las suyas. Para hacerlo primero cada proceso calcula sus propias fronteras y el id de los procesos a los que tiene que enviar y recibir dichas fronteras. Para establecer un orden de envió y recibo, se ha decidido que los procesos pares son los primeros en enviar sus fronteras inferiores y despues las reciben, seguidamente envían y reciben las superiores. Los procesos impares el orden de envió y recibo es inverso teniendo en cuenta también que la frontera inferior de un proceso es la superior del siguiente. Para sincronizarlo, se ha utilizado "MPI_Ssend". En la siguiente captura se puede ver el orden para los procesos pares:

```
new_borders( m, balance[rank], up, down );

if (MPI_Ssend(down, 1, coltype, down_border_dest, 0, MPI_COMM_WORLD) != MPI_SUCCESS){
  printf("Fallo en enviar el up. Proceso %d\n", rank);
  exit(1);
}

int *up_fronter = ( int * ) malloc (m  * sizeof(int));
int *down_fronter = ( int * ) malloc (m  * sizeof(int));

if(MPI_Recv(up_fronter, 1, coltype, up_border_dest, 0, MPI_COMM_WORLD, &status) != MPI_SUCCESS){
  exit(2);
}

if (MPI_Ssend(up, 1, coltype, up_border_dest, 0, MPI_COMM_WORLD) != MPI_SUCCESS){
  printf("Fallo en enviar el up. Proceso %d\n", rank);
  exit(1);
}

if(MPI_Recv(down_fronter, 1, coltype, down_border_dest, 0, MPI_COMM_WORLD, &status) != MPI_SUCCESS){
  exit(2);
}


rbuf = life_update_parallel( m, balance[rank], rbuf, up_fronter, down_fronter );
```

A la hora de enviar y recibir las fronteras, se ha utlizado el tipo derivador "MPI_Type_vector":

```
/* Apartado tipos derivados: MPI_Type_Vector*/
MPI_Datatype coltype;
MPI_Type_vector(m, 1, 1, MPI_INT, &coltype);
MPI_Type_commit(&coltype);
```

Por ultimo lo que se hace es ejecutar la lógica del juego, al final tanto si el programa es secuencial o paralelo utilizan la misma función, con la única diferencia de que en el paralelo, antes se crea una nueva submatriz que contiene la fila superior, seguida de la submatriz recibida en el scatter y la inferior. De forma que en la función que calcula la siguiente iteración, como el estado de las fronteras ya lo calculan otros procesos, se controla para no hacer un calculo innecesario, en cambio

si es secuencial se calcula en toda la matriz.

```
//LIFE_UPDATE updates a Life grid of the sub-array.
void life_update_sub_array(int m, int n, int new_grid[], int s[], int type)
{
  int i, j;
  int initial_row = (type == PARALLEL) ? 1 : 0;
  int final_row = (type == PARALLEL) ? (n - 1) : n;

  for ( j = initial_row; j < final_row; j++ )
  {
    for ( i = 0; i < m; i++ )
    {
      if ( new_grid[i+j*(m)] == 0 )
      {
        if ( s[i+(j)*m] == 3 )
        {
          new_grid[i+j*(m)] = 1;
        }
      }
      else if ( new_grid[i+j*(m)] == 1 )
      {
        if ( s[i+(j)*m] < 2 || 3 < s[i+(j)*m] )
        {
          new_grid[i+j*(m)] = 0;
        }
      }
    }
  }
```

Al final de cada iteración, se comprueba si toca escribir el archivo de esta, si el programa es paralelo, se escribe utilizando "MPI_File_write" ajustando antes para cada proceso su "MPI_File_set_view".

```
MPI_File_open (MPI_COMM_WORLD, output_filename, MPI_MODE_CREATE | MPI_MODE_WRONLY,MPI_INFO_NULL, &myfile);
MPI_File_set_view(myfile, displs[rank],MPI_CHAR, MPI_CHAR, "native",MPI_INFO_NULL);
MPI_File_write(myfile, final_str, (scounts[rank] + lines[rank]) * 2 * sizeof(char), MPI_CHAR,MPI_STATUS_IGNORE);
MPI_File_close(&myfile);
```

2 Speed-Up's y Eficiencia

Para analizar y comentar el rendimiento obtenido por la aplicación paralela, se ha decidido calcular el Speed-up y la Eficiencia. Para hacerlo, se ha ejecutado el programa con el mismo numero de iteraciones 150, utilizando de 1 a 12 procesadores y par ver la escalabilidad se han introducido las diferentes matrices. De forma que los resultados obtenidos para cada matriz han sido:

2.1 10x10

Matríz	Procesadores	Iteraciones	Tiempo(ms)	Speed-Up	Eficiencia
10x10	1	150	18.030		1
10x10	2	150	186.321	10,33394343	5,166971714
10x10	3	150	480.222	26,63460899	8,878202995
10x10	4	150	303.983	16,8598447	4,214961176
10x10	5	150	645.173	35,7833056	7,15666112
10x10	6	150	339.096	18,80732113	3,134553522
10x10	7	150	297.924	16,52379368	2,360541954
10x10	8	150	585.111	32,45207987	4,056509983
10x10	9	150	365.237	20,25718247	2,250798053
10x10	10	150	616.243	34,17875763	3,417875763
10x10	11	150	685.175	38,00194121	3,454721928
10x10	12	150	565.243	31,35013866	2,612511555

2.2 80x140

Matríz	Procesadores	Iteraciones	Tiempo(ms)	Speed-Up	Eficiencia
80x140	1	150	114.275		1
80x140	2	150	88.707	0,7762590243	0,3881295121
80x140	3	150	217.783	1,905779917	0,6352599723
80x140	4	150	82.154	0,7189148983	0,1797287246
80x140	5	150	200.240	1,752264275	0,350452855
80x140	6	150	278.401	2,436237147	0,4060395245
80x140	7	150	392.902	3,438214833	0,4911735475
80x140	8	150	393.174	3,440595056	0,430074382
80x140	9	150	263.578	2,306523737	0,2562804152
80x140	10	150	373.054	3,26452855	0,326452855
80x140	11	150	355.522	3,111109166	0,282828106
80x140	12	150	972.826	8,513025596	0,7094187997

2.3 200x400

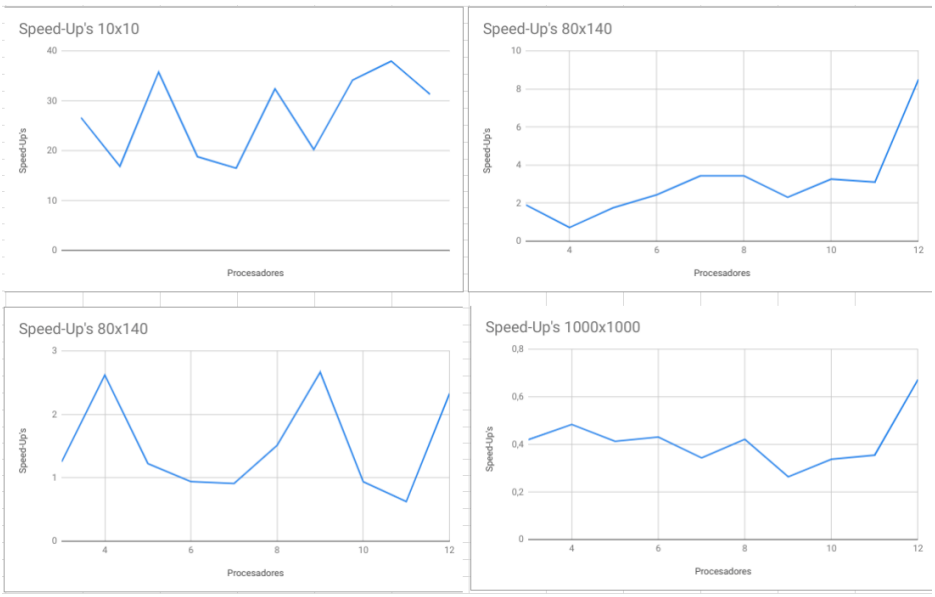
Matríz	Procesadores	Iteraciones	Tiempo(ms)	Speed-Up	Eficiencia
200x400	1	150	373.620		1
200x400	2	150	251.994	0,674466035	0,3372330175
200x400	3	150	468.549	1,254079011	0,4180263369
200x400	4	150	980.235	2,623614903	0,6559037257
200x400	5	150	457.920	1,22563032	0,2451260639
200x400	6	150	351.148	0,9398533269	0,1566422212
200x400	7	150	340.170	0,9104705316	0,1300672188
200x400	8	150	565.358	1,513189872	0,189148734
200x400	9	150	996.681	2,667632889	0,2964036543
200x400	10	150	350.703	0,9386622772	0,09386622772
200x400	11	150	233.252	0,6243027675	0,05675479705
200x400	12	150	870.015	2,328609282	0,1940507735

2.4 1000x1000

Matríz	Procesadores	Iteraciones	Tiempo(ms)	Speed-Up	Eficiencia
1000x1000	1	150	4.281.775		1
1000x1000	2	150	2.385.841	0,5572084007	0,2786042004
1000x1000	3	150	1.801.215	0,420670166	0,1402233887
1000x1000	4	150	2.073.311	0,4842176434	0,1210544108
1000x1000	5	150	1.769.537	0,4132718324	0,08265436647
1000x1000	6	150	1.847.201	0,4314101045	0,07190168408
1000x1000	7	150	1.471.217	0,3435997921	0,04908568459
1000x1000	8	150	1.806.756	0,4219642555	0,05274553193
1000x1000	9	150	1.129.471	0,2637856964	0,02930952182
1000x1000	10	150	1.447.942	0,3381639624	0,03381639624
1000x1000	11	150	1.521.359	0,3553103561	0,03230094146
1000x1000	12	150	2.882.699	0,6732485943	0,05610404953

2.5 Speed-Up's

Tal como se puede ver en la recolección de datos anterior, el Speed-Up obtenido para las matrices más pequeñas es mayor que con 1 solo procesador, esto nos indica que es poco eficiente tener más procesadores ejecutando la aplicación cuando se trata de un tablero pequeño ya que el tiempo que se invierte en realizar las comunicaciones locales entre procesos, es superior que si lo calculase un solo procesador. Tal como se puede ver en la matriz de 1000x1000 al añadirle 2 procesadores si que se llega a obtener una aceleración cerca de la ideal duplicando la velocidad. Utilizando más de 2 procesadores deja de ser eficiente. A continuación se muestran los Speed-Up's de forma gráfica:



2.6 Eficiencia

Con la eficiencia, pasa exactamente lo mismo, como más grande es la matriz introducida i más iteraciones requiera el programa, se obtendrá una eficiencia más cercana a la esperada. En nuestro caso la única eficiencia obtenida aceptable, ha sido en la matriz de 1000x1000, utilizando 2 procesadores. De 2 a 4 hemos visto que se obtiene una eficiencia razonable, de 4 a 12 procesadores, para el tamaño del tablero, la cantidad de esfuerzo que se desperdicia en la comunicación y la sincronización no es justificable. Tal como se ve en los gráficos, como más grande es el tablero, es decir, como más se escala el problema, más justificada es su paralelización, para tableros pequeños es practicamente innecesario.