Segundo Laboratorio de Paralelos

Luis Gustavo Cáceres Zegarra

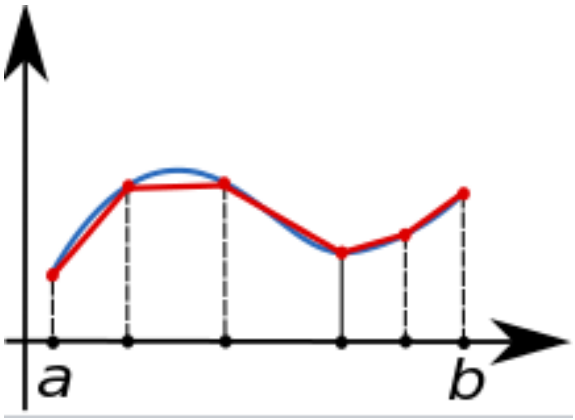
April 24, 2017

1 La regla trapezoidal

Las personas que escriben programas paralelos usan el verbo paralelizado para describir el proceso de convertir un programa en serie en uno en paralelo.

Podemos diseñar un programa en paralelo siguiendo 4 pasos basicos.

- Particionar la solución del problema en tareas.
- Identificar el canal de comunicación entre las tareas.
- Agergar las tareas hacia una tarea compuesta.
- Mapear las tareas compuestas a los nucleos.



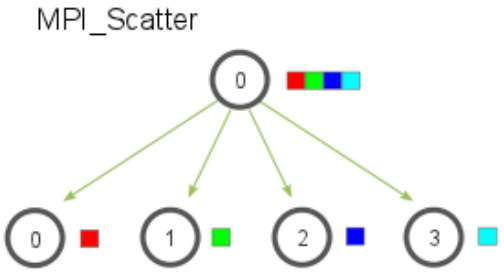
En este primer ejercicio debemos calcular en valor la estimación del area que se encuentra bajo la función en un tramo determinado.

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

double funcion(double x)
{

        return x+3;
}


double Trap(double a,double b,double n,double h)
{
        double integral = (funcion(a)+funcion(b))/2;
        int i;
        for(i=1;i<=n-1;i++)
        {
                integral += funcion(a+i*h);
```

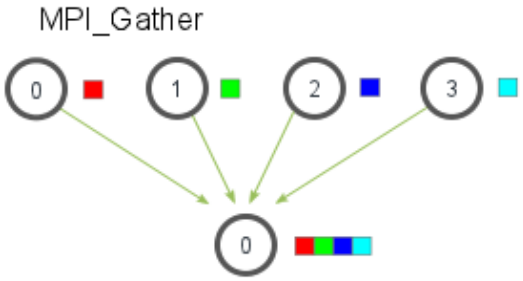
```
        }
        return integral*h;
}


int main()
{
        int rank,size;
        MPI_Init(NULL,NULL);
        MPI_Comm_size(MPI_COMM_WORLD,&size);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);

        double a = 0;
        double b = 3.0;
        int n = 1024;

        double h = (b-a)/n;
        int my_n = n/size;

        double my_a = a + rank*h*my_n;
        double my_b = my_a + h*my_n;
        double my_integral = Trap(my_a,my_b,my_n,h);

        int i;
        double total_integral;
        if(rank==0)
        {
                for(i=1;i<size;i++)
                {
                        total_integral = my_integral;
                        MPI_Recv(&my_integral,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,MPI_STAT
                        total_integral += my_integral;
                }
                printf("Estimacion de la integral: %f\n",total_integral);
        }
        else
        {
                MPI_Send(&my_integral,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
        }


        MPI_Finalize();
}
```

2 MPI_scatter y MPI_gather

MPI_scatter es una rutina colectiva. Tambien envuelve a un proceso raiz designado que manda información a todos los procesos en un comunicador. En conclusion MPI_scatter envia bloques de un arreglo adiferentes procesos. En la siguiente imagen veremos como funciona de forma grafica.



MPI_gather es el inverso de MPI_scatter. En lugar de distribuir los elementos de un proceso a muchos procesos. MPI_gather toma los elementos de varios procesos y los reune en un solo proceso. Esta rutina es my usada por muchos algortimos paralelos, como por ejemplo busquedas



o sort paralelo.

Como parte de este trabajo se implemento un programa que calcula el promedio de los elementos de un arreglo. Se utiliza la funcion MPI_scatter distibuira el arreglo a los diferentes procesos, los valores de los promedios se guardaran en un arreglo auxiliar del tamaño de numero de los procesos y MPI_gather los unira al proceso 0 para que se calcule el promedio final del arreglo auxiliar.

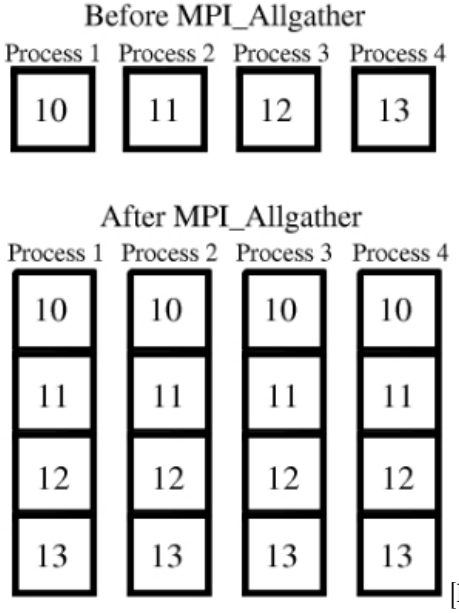
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

float Promedio(float* p,int size)
{
        float promedio = 0;
        int i;
        for(i=0;i<size;i++)
        {
                promedio+=p[i];
        }
        return promedio/size;
}

int main()
{
        int world_rank,world_size;
        MPI_Init(NULL,NULL);
        MPI_Comm_size(MPI_COMM_WORLD,&world_size);
        MPI_Comm_rank(MPI_COMM_WORLD,&world_rank);

        int elements_per_proc = 6;
        float* rand_nums = NULL;
        int i,arr_size = world_size*elements_per_proc;


        if (world_rank == 0) {

                rand_nums = malloc(sizeof(float)*arr_size);
                for(i=1;i<=arr_size;i++)
                {
```

```
                rand_nums[i-1]=i*1.0;
            }
        }


        float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);


        MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,elements_pe


        float sub_avg = Promedio(sub_rand_nums, elements_per_proc);

        float *sub_avgs = NULL;
        if (world_rank == 0) {
          sub_avgs = malloc(sizeof(float) * world_size);
        }
        MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
                   MPI_COMM_WORLD);


        if (world_rank == 0) {
          float avg = Promedio(sub_avgs, world_size);
          printf("Promedio: %f\n",avg);
        }

        MPI_Finalize();
        return 0;
}
```

3 Multiplicar una matriz por un vector utilizando MPI_allgather

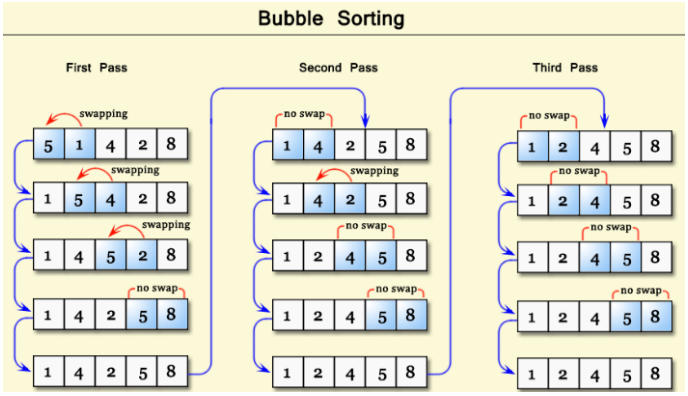
MPI_allgather puede ser visto como un MPI_gather, pero donde todos los procesos reciben el resultado, en lugar de que solo lo hag la raiz. El i-esimo bloque de informacion enviada de cada proceso es recibido por cada proceso y lo coloca en el i-esimo bloque del buffer auxiliar.

El patron de comunicación de MPI_allgather ejecutado en un dominio de intercomunicación no necesita ser simétrico. El número de items enviados por los procesos en un grupo A no necesita ser igual a los items del grupo B. [h]

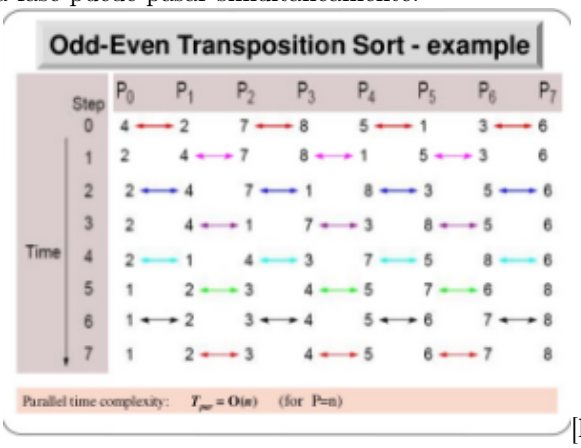


4 Bubble sort paralelizado

En esta seccion paralelizaremos un algortimo bastante conocido, como el bubble sort. Un algortimo que se utiliza para ordenar un arreglo ya sea de forma ascendente o descendente. Este algortimo es lento a comparación de otros que tiene el mismo proposito.



Pero para prosotitos de este ejercicio utilizaremos la transposisción de ordenamiento odd-even que tiene mas oportunidades para paralelizar bubble sort, porque todos los cambios(swaps) en una sola fase puede pasar simultaneamente.



[h] Acontinuacion el codigo del algortimo bubble sort paralelizado.

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

#define N 20

void sequentialSort(int *arrayToSort, int size) {
    int sorted = 0;
    while( sorted == 0) {
        sorted= 1;
        int i;
        for(i=1;i<size-1; i += 2) {
            if(arrayToSort[i] > arrayToSort[i+1])
            {
                int temp = arrayToSort[i+1];
                arrayToSort[i+1] = arrayToSort[i];
                arrayToSort[i] = temp;
                sorted = 0;
            }
        }
```

```
        for(i=0;i<size-1;i+=2) {
            if(arrayToSort[i] > arrayToSort[i+1])
            {
                int temp = arrayToSort[i+1];
                arrayToSort[i+1] = arrayToSort[i];
                arrayToSort[i] = temp;
                sorted = 0;
            }
        }
    }
}


void lower(int *array1, int *array2, int size)
{

    int *new = malloc(size*sizeof(int));
    int k = 0;
    int l = 0;
    int count;
    for(count=0;count<size;count++) {
        if (array1[k] <= array2[l] ) {
            new[count] = array1[k];
            k++;
        } else {
            new[count] = array2[l];
            l++;
        }
    }

    for(count=0;count<size;count++) {
        array1[count] = new[count];
    }
    free(new);
}


void higher(int *array1, int *array2, int size)
{
    int *new = malloc(size*sizeof(int));
    int k = size-1;
    int l = size-1;
    int count;
    for(count=size-1;count>=0;count--) {
        if (array1[k] >= array2[l] ) {
            new[count] = array1[k];
            k--;
        } else {
            new[count] = array2[l];
            l--;
        }
    }

    for(count=0;count<size;count++) {
        array1[count] = new[count];
    }
    free(new);
```

```
}


void exchangeWithNext(int *subArray, int size, int rank)
{
    MPI_Send(subArray,size,MPI_INT,rank+1,0,MPI_COMM_WORLD);

    int *nextArray = malloc(size*sizeof(int));
    MPI_Status stat;
    MPI_Recv(nextArray,size,MPI_INT,rank+1,0,MPI_COMM_WORLD,&stat);
    lower(subArray,nextArray,size);
    free(nextArray);
}

void exchangeWithPrevious(int *subArray, int size, int rank)
{

    MPI_Send(subArray,size,MPI_INT,rank-1,0,MPI_COMM_WORLD);

    int *previousArray = malloc(size*sizeof(int));
    MPI_Status stat;
    MPI_Recv(previousArray,size,MPI_INT,rank-1,0,MPI_COMM_WORLD,&stat);

    higher(subArray,previousArray,size);
    free(previousArray);
}

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

    MPI_Init(&argc,&argv);


    int hostCount;
    MPI_Comm_size(MPI_COMM_WORLD,&hostCount);


    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);


    int *arrayToSort = malloc(N * sizeof(int));
    if(rank == 0) {
        srand(time(NULL));
        printf("Iinitial array\n");
        int i;
        for(i=0;i<N;i++) {
            arrayToSort[i] = rand()%100;
            printf("%d ",arrayToSort[i]);
        }
        printf("\n");
    }



    double start;
    if (rank == 0) {
        start = clock();
```

```
}

int *subArray = malloc(N/hostCount  * sizeof(int));
if(rank == 0) {

    MPI_Scatter(arrayToSort,N/hostCount,MPI_INT,subArray,N/hostCount,MPI_INT,0,MPI
}


int *displs = malloc(hostCount * sizeof(int));
int i;
for (i=0;i<hostCount;i++) {
    displs[i] = i*(N/hostCount);
}


int *sendcnts = malloc(hostCount * sizeof(int));
for (i=0;i<hostCount;i++) {
    sendcnts[i]=N/hostCount;
}

MPI_Scatterv(arrayToSort,sendcnts,displs,MPI_INT,subArray,N/hostCount,MPI_INT,0,MI
free(displs);
free(sendcnts);


sequentialSort(subArray,N/hostCount);

i =0;
for(i=0;i<hostCount;i++) {

    if (i%2==0) {

        if(rank%2==0) {

            if(rank<hostCount-1) {

                exchangeWithNext(subArray,N/hostCount,rank);
            }
        } else {

            if (rank-1 >=0 ) {

                exchangeWithPrevious(subArray,N/hostCount,rank);
            }
        }
    }

    else {

        if(rank%2!=0) {

            if (rank<hostCount-1) {

                exchangeWithNext(subArray,N/hostCount,rank);
            }
        }
```

```
            else {

                if (rank-1 >=0 ) {

                    exchangeWithPrevious(subArray,N/hostCount,rank);
                }
            }
        }
    }


    MPI_Gather(subArray,N/hostCount,MPI_INT,arrayToSort,N/hostCount,MPI_INT,0,MPI_COMM

    if(rank == 0) {
        printf("\n");
        printf("Sorted array\n");
        int elem;
        for(elem=0;elem<N;elem++) {
            printf("%d ",arrayToSort[elem]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

References