Proyecto final: Autómatas celulares. Juego de la vida versión paralela.

Barbosa Martínez Erick Gabriel, Castro Molano Ramsés,
García Vázquez José Ángel de Jesús
Sistemas Distribuidos. Facultad de ingeniería, UNAM.
berick_2@hotmail.com, ramses-c14@live.com.mx,
jose_angel_vazquez@comunidad.unam.mx

Abstract- This article shows an implementation of the cellular automaton used in the develop of a compact version of Conway´s Game of Life in C in parallel way.

1.OBJETIVO

Generar una versión completa del juego de la vida de Conway siguiendo sus reglas y estableciendo los límites del modelo utilizado de manera paralela.

2.INTRODUCCIÓN

En los últimos años, el paralelismo en computación ha tomado una gran importancia al grado de aumentar considerablemente el número de núcleos que posee un procesador para que realice procesamiento en paralelo con mayor velocidad y eficacia.

Cuando hablamos de paralelismo en sistemas computacionales nos referimos a diferentes núcleos o Cores realizando tareas de manera cooperativa al mismo tiempo. Este concepto ha tomado tanta fuerza que la mayoría de las supercomputadoras actuales dependen en gran manera del procesamiento paralelo, aumentando de una manera similar a lo que establece Moore en su ley homónima, la *Ley de Moore.* De esta manera el número de Cores en un sistema aumenta de manera considerable año tras año.

Muchas de las personas que empiezan a trabajar con sistemas paralelos tienen la creencia de que a mayor cantidad de hardware mayor será la velocidad de procesamiento, pero esto es un problema tan común que hace falta establecer ciertas reglas que se deben tomar en cuenta para realizar la paralelización. Para empezar, se debe de preguntar si realmente se debe de paralelizar el sistema o programa, evaluar la necesidad y las ventajas que esto podría traer o si debe quedarse como un programa que se ejecute en un solo procesador. La mayoría de los científicos y usuarios actualmente no requieren de paralelizar algún proceso, esto pues los programas que realizan pueden funcionar de manera eficiente con un solo procesador y no se ven en la necesidad de añadir procesadores para tener un mejor rendimiento. Este tipo de acciones se realizan mayormente en el cómputo científico. Dentro de las ocasiones en las que debemos considerar realizar una paralelización se encuentran situaciones relacionadas con problemas de velocidad donde se necesite una respuesta inmediata que un solo procesador no nos puede dar, otro caso importante tiene que ver con la memoria disponible, esto pues la memoria principal puede no ser suficiente para almacenar respuestas a problemas de gran escala.

Para realizar la paralelización de un programa hay que identificar en el algoritmo las partes que puedan ser paralelizadas y el método que se quiere usar para realizarlo

Uno de los mayores problemas que se encuentran en el cómputo científico y quieren ser paralelizados se encuentran relacionados de manera directa con procesar una gran cantidad de datos almacenados

en una computadora. Para realizar esto se desea paralelizar de manera que se tengan múltiples procesadores trabajando con diferentes partes de los datos al mismo tiempo, hablando así concretamente de *paralelismo de datos.* De manera más concreta, cada procesador trabaja con una pequeña parte de los datos, la cual se distribuye de manera que cada procesador tiene una parte de datos del mismo tamaño. Haciendo esto se reduce la posibilidad de causar un cuello de botella que pueda generar la información en grandes cantidades.

El otro tipo de paralelismo, el *paralelismo funcional,* se basa en dividir un problema en un gran número de partes más pequeñas, dichas subtareas son asignadas a los procesadores para su ejecución de modo que cada procesador realizará otra subtarea en cuanto finalice de ejecutar la que previamente se le fue asignada.

Al dividir los datos o las tareas nos vemos en la necesidad de tener una comunicación ordenada para poder "unir" todo lo hecho por los diferentes procesadores sin que se pierda algún dato o alguna subtarea quede sin ejecutarse. Para resolver esto se definen varios tipos de comunicación.

Comunicación síncrona.

El proceso que envía un mensaje no realizará su siguiente tarea hasta que el receptor indique la recepción exitosa del mensaje. A simple vista se nota que este tipo de comunicación puede llegar a generar largos tiempos por la espera de mensajes, pero es de gran utilidad si se requiere que un proceso obtenga un dato en específico y llevar un mayor control sobre la recepción y envío de mensajes.

Comunicación asíncrona.

El proceso que envía un mensaje realizará su siguiente tarea sin importar que el receptor confirme la recepción, de esta manera el tiempo de ejecución se reduce considerablemente, pero se pueden generar errores por la falta de confirmación por parte del receptor.

Ahora, dentro del paralelismo existen ciertos tipos de comunicaciones que se basan de cierta manera en los anteriormente mencionados.

Broadcast

Uno de los procesadores envía a todos los procesadores un mensaje, no hay procesador que no reciba el mensaje, incluyendo así al remitente.

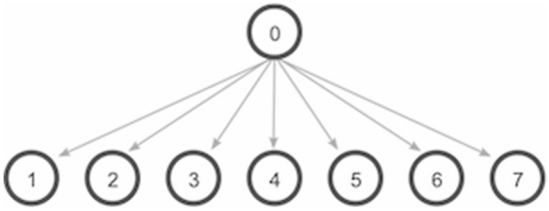


Figura 1. Broadcast

Multicast

Uno de los procesadores envía a un grupo de varios procesadores que deben de ver identificados por el remitente.

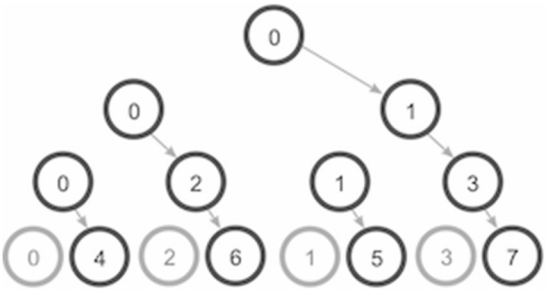


Figura 2. Multicast

Scatter

El procesador que realiza el envío lo hace por medio de un arreglo, es decir, se tiene un arreglo de n datos y cada elemento del arreglo se le enviará a cada proceso sin repetir, teniendo así cada procesador un elemento diferente del arreglo.

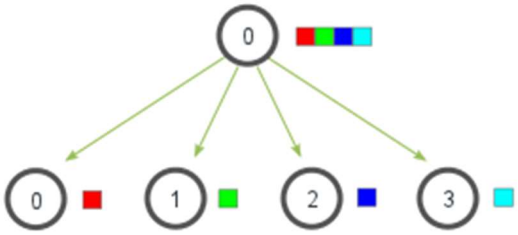


Figura 3. Scatter

Gather

Es el inverso de Scatter pues los procesadores envían cada uno un elemento de un arreglo que el procesador destino recolectará para formar el arreglo completo.

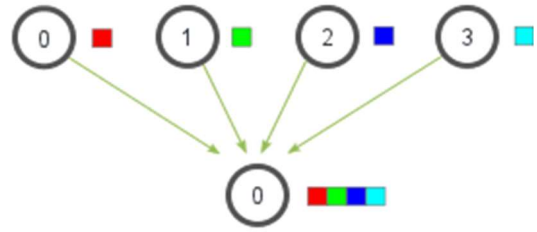


Figura 4. Gather

Estas comunicaciones se implementan con la ayuda de MPI. MPI(Message Passing Interface) es un estándar utilizado en el paso de mensajes para múltiples procesadores que define ciertas funciones que se encuentran en una biblioteca especializada disponible para C o C++
Dicha biblioteca no es difícil de usar, por lo cual es de gran ayuda a la hora comprender los procesos de paralelización, tiene grandes ventajas y gran poder a la hora de paralelizar los programas.

3.Desarrollo

El juego de la vida representa de cierta manera simplificada/metafórica nuestras vidas junto con sus condiciones. Se desarrolla empleando autómatas celulares, pues sus células se actualizan cada periodo de tiempo discreto y su estado depende totalmente de sus células vecinas.

Un autómata celular se define dentro de un sistema determinístico, donde para cada posible estado se puede conocer el posible estado siguiente. Si se construyen las reglas del sistema de manera correcta, cada autómata celular poseerá un estado antecesor, el cual será único para cada autómata

Ahora, dentro de los autómatas celulares encontramos una gran variedad de elementos que lo conforman.

Un espacio regular: Ya sea una línea, un plano de 2 dimensiones o un espacio n-dimensional. Cada división del espacio es llamada célula.

Conjunto de Estados: Es finito y cada elemento o célula del espacio toma un valor de este conjunto de estados. También se denomina alfabeto. Puede ser expresado en valores o colores.

Configuración Inicial: Es la asignación inicial de un estado a cada una de las células del espacio.

Vecindades: Define el conjunto de células que se consideran adyacentes a una dada, así como la posición relativa respecto a ella. Cuando el espacio es uniforme, la vecindad de cada célula es isomorfa (es decir, que tiene el mismo aspecto).

Función de Transición Local: Es la regla de evolución que determina el comportamiento del AC. Se calcula a partir del estado de la célula y su vecindad. Define cómo debe cambiar de estado cada célula dependiendo su estado anterior y de los estados anteriores de su vecindad. Suele darse como una expresión algebraica o un grupo de ecuaciones.

Ahora, teniendo claro que es un autómata celular y los elementos que la conforman podemos definir un modelo matemático que represente lo que es el juego de la vida y cómo es que funciona en torno a los autómatas y sus reglas.

4. Modelo matemático discreto

Geometría del espacio regular: Se tomará en cuenta una geometría cuadrada para cada célula

Geometría de la vecindad: Para la implementación del juego de la vida se propone una vecindad de Moore para establecer todos los vecinos posibles.

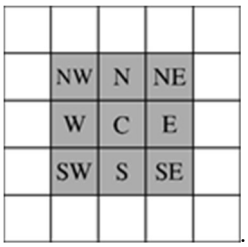


Figura 5. Vecindad de Moore

Conjunto de estados de las células: Cada célula tendrá un valor binario definido que representará si se encuentra con vida o si está muerta, este estado puede verse alterado cada turno por la condición de sus células vecinas

Reglas de transición:

•Una célula muerta con exactamente 3 células vecinas vivas "nace" (es decir, al turno siguiente estará viva).

•Una célula viva con 2 o 3 células vecinas vivas sigue viva, en otro caso muere (por "soledad" o "superpoblación").

Condiciones de frontera: Para esta implementación consideramos una frontera cerrada, esto significa que los elementos que se encuentran en los extremos de las mallas no podrán considerar como un vecino a otro que se encuentre en el extremo consecuente.

Condición inicial: El estado de las células en el momento inicial puede definirse de manera estática por el programador o de manera dinámica en el mismo programa por medio de la interacción con el usuario, de esta manera se puede decir que célula estará viva y generar cierto patrón.

Condición de paro: Las células seguirán actualizando su estado de manera constante a menos que se pare el programa por medio de un comando, de otra manera el programa seguirá corriendo, aunque se entre en algún patrón estático.

Para la distribución de la malla entre los diferentes procesos se decidió enviar una copia de toda la malla a cada proceso de manera que cada proceso solo se encargue un renglón de la malla.

5. Diseño

Diagrama UML

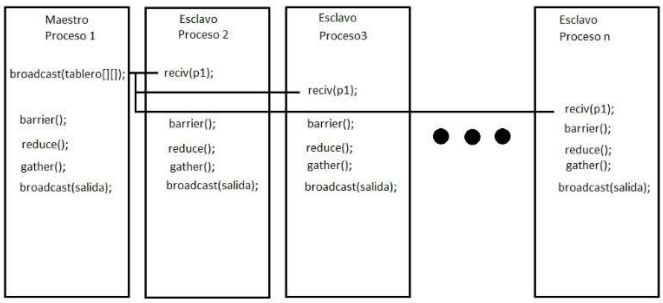
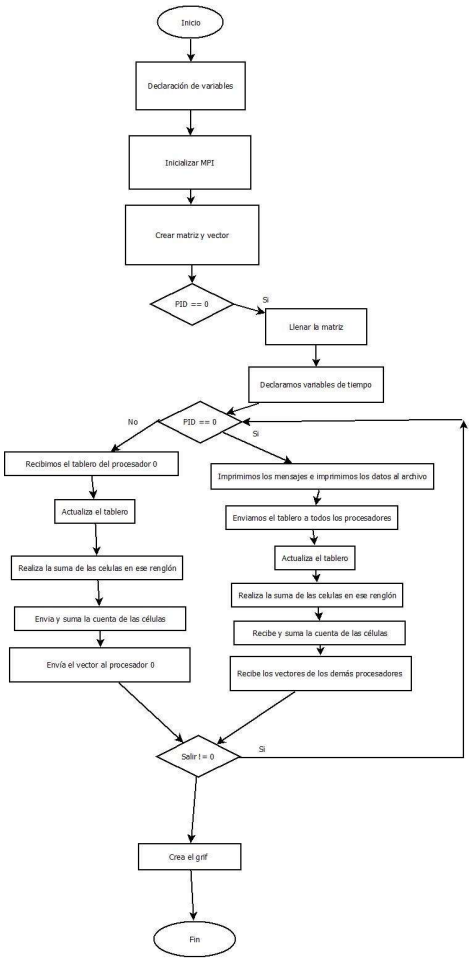
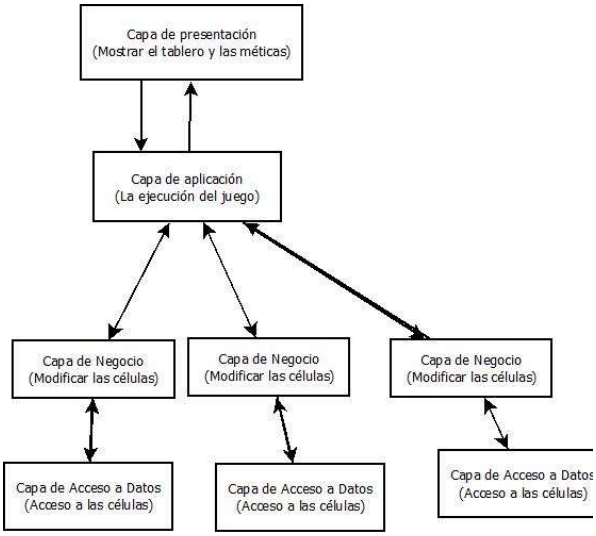


Diagrama de Flujo



Patrón Arquitectónico



6. Código Fuente Importante

Función encargada de almacenar los datos del tablero en un archivo

```
//Función para imprimir los puntos en el archivo
void imprimeTableroArchivo(int tamano, int tablero[tamano][tamano],FILE
    int i,j, tam = tamano - 1;
    for (i = 1; i < tam; ++i){
        for (j = 1; j < tam; ++j){
            fprintf(archivoPuntos,"%d ",tablero[i][j]);
        }
        fprintf(archivoPuntos,"\n");
    }
    fprintf(archivoPuntos,"\n\n");
}
```

Función main, encargada de ejecutar el juego y paralelizarlo.

```
//Función principal
int main(int argc, char **argv){

    char auxX[100]="set xrange [ -0.500000 : ";
    char auxRango[2]="";
    char auxY[100]="set yrange [ -0.500000 : ";

    FILE * archivoPuntos = fopen("Calor.txt", "w");

    int pid, npr;
    int i, nalir;
    int fil, col;
    srand(time(NULL));

    //Inicializamos MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD , &pid);
    MPI_Comm_size(MPI_COMM_WORLD , &npr);

    int tamano = npr + 2, count = tamano * tamano;
    int tablero[tamano][tamano], aux[tamano][tamano], vecRen[npr];
    int temporal[npr][npr];
    int poblacion, cuenta;

    if(pid == 0){ //Procesador 0
        for(i = 0; i < tamano; i++){ //Define los limites de la fontera con 0
            aux[0][i] = tablero[0][i] = aux[i][0] = tablero[i][0] = 0;
            aux[npr - 1][i] = tablero[npr - 1][i] = aux[i][npr - 1] = tablero[i][npr - 1]
        }

        poblacion = ((count/100.0)*tipo);
        printf("\n");
        for(i = 0; i < poblacion; i++){
            fil = rand() % npr;
            col = rand() % npr;
            //Si la celula esta muerta, la resucita
            if(tablero [fil + 1][col + 1] == 0){
                aux[fil + 1][col + 1] = tablero[fil + 1][col + 1] = 1;
            }
```

```
            /*Si ya estaba viva se mantiene, y no lo contamos como
            una posicion añadida, así aseguramos que se impriman el
            numero de celulas vivas seleccionado. Por ello restamos uno
            al contador*/
            else
                i--;
        }
    }

MPI_Barrier(MPI_COMM_WORLD); //Barrera para que todos lo ejecuten a la vez

    if(pid == 0){
        double inicio, final, final1, final2;
        int turno = 0;
        inicio = final = final1 = final2 = 0;
        do{
            //printf("Tablero\n");
            //imprimeM(tamano, tablero); //Imprime el tablero
            system(CLMPIAR);
            printf("Turno: %d\n", turno);
            printf("Población: %d\n", poblacion);
            printf("El tamaño del tablero es de : %d * %d\n", npr, npr);
            printf("Tiempo de actualización del sistema: %f[s]\n", (final - inicio));
            printf("Tiempo de envio del tablero: %f[s]\n", (final1 - inicio));
            printf("Tiempo de actualización del renglón: %f[s]\n", (final2 - inicio));

            inicio = MPI_Wtime();
            MPI_Bcast(tablero, count, MPI_INT, 0, MPI_COMM_WORLD); //Envia el tablero
            final1 = MPI_Wtime();

            actualiza(tamano, tablero, pid + 1, vecRen); //Se actualiza el tablero con el renglón correspondiente
            cuenta = Vida(npr, vecRen);
            final2 = MPI_Wtime();

            MPI_Barrier(MPI_COMM_WORLD); //Barrera para que todos lo ejecuten a la vez

            MPI_Reduce(&cuenta, &poblacion, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
            MPI_Gather(&vecRen, npr, MPI_INT, temporal, npr, MPI_INT, 0, MPI_COMM_WORLD ); //Se reciben los renglones ya actualizados

            final = MPI_Wtime();
            convertirM(tamano, temporal, tablero);
            turno ++;
            imprimeTableroArchivo(tamano, tablero,archivoPuntos); //Imprime el tablero
            salir = decision();
            MPI_Bcast(&salir, 1, MPI_INT, 0, MPI_COMM_WORLD);

        } while (salir!=0);

        char * configGnuplot[] = {  //"set view map",
                                    //"splot \"Calor.txt\" matrix with image",
                                    "set terminal gif animate delay 100",
                                    "set output 'juego.gif'",
                                    "unset key",
                                    "set style increment default",
                                    "set view map scale 1",
                                    "set style data lines",
                                    "set xtics border in scale 0,0 mirror norotate  autojustify",
                                    "set ytics border in scale 0,0 mirror norotate  autojustify",
                                    "set ztics border in scale 0,0 nomirror norotate  autojustify",
                                    "unset cbtics",
                                    "set rtics axis in scale 0,0 nomirror norotate  autojustify",
                                    "set title \"Juego de la vida\"",
                                    "set xrange [ -0.500000 : 7.50000 ] noreverse nowriteback",
                                    "set x2range [ * : * ] noreverse writeback",
                                    "set yrange [ -0.500000 : 7.50000 ] noreverse nowriteback",
                                    "set y2range [ * : * ] noreverse writeback",
                                    "set zrange [ * : * ] noreverse writeback",
                                    "set cblabel \"\"",
                                    "set cbrange [ 0.00000 : 1.00000 ] noreverse nowriteback",
                                    "set rrange [ * : * ] noreverse writeback",
                                    "set palette rgbformulae 7, 5, 15",
                                    //"splot \"Calor.txt\" matrix with image",
                                    "stats \"Calor.txt\" noouput",
                                    "do for [i=1:int(STATS_blocks) ] { splot \"Calor.txt\" index (i-1) matrix with image }"
                                };
        sprintf(auxRango,"%d",npr - 1);
        //printf("%s\n",auxRango );
        strcat(auxX,auxRango);
        strcat(auxX, ".50000 ] noreverse nowriteback");
        //printf("%s\n",auxX);

        strcat(auxY,auxRango);
        //printf("%s\n",auxY);
        strcat(auxY, ".50000 ] noreverse nowriteback");
        //printf("%s\n",auxY);
        //printf("\n");
        configGnuplot[12] = auxX;
        configGnuplot[14] = auxY;

        FILE * ventanaGnuplot = popen ("gnuplot -persist", "w");
        // Executing gnuplot commands one by one
        for (i=0;i<NUM_COMANDOS;i++){
            fprintf(ventanaGnuplot, "%s \n", configGnuplot[i]);
        }
        fclose(archivoPuntos);
    }
```

7. Métricas

Medimos el tiempo de actualización del tablero de manera paralela, variamos el tamaño de este para ver la diferencia de tiempo.

Matriz de 10 * 10

```
Turno: 0
Población: 36
El tamaño del tablero es de : 10 * 10
Tiempo de actualización del sistema: 0.000000[s]
Tiempo de envio del tablero: 0.000000[s]
Tiempo de actualización del renglón: 0.000000[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

```
Turno: 10
Población: 8
El tamaño del tablero es de : 10 * 10
Tiempo de actualización del sistema: 0.000416[s]
Tiempo de envio del tablero: 0.000083[s]
Tiempo de actualización del renglón: 0.000086[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

```
Turno: 50
Población: 8
El tamaño del tablero es de : 10 * 10
Tiempo de actualización del sistema: 0.000312[s]
Tiempo de envio del tablero: 0.000069[s]
Tiempo de actualización del renglón: 0.000072[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

Matriz de 50 * 50

```
Turno: 0
Población: 676
El tamaño del tablero es de : 50 * 50
Tiempo de actualización del sistema: 0.000000[s]
Tiempo de envio del tablero: 0.000000[s]
Tiempo de actualización del renglón: 0.000000[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

```
Turno: 10
Población: 430
El tamaño del tablero es de : 50 * 50
Tiempo de actualización del sistema: 0.002000[s]
Tiempo de envio del tablero: 0.000730[s]
Tiempo de actualización del renglón: 0.000739[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

```
Turno: 50
Población: 272
El tamaño del tablero es de : 50 * 50
Tiempo de actualización del sistema: 0.001702[s]
Tiempo de envio del tablero: 0.000825[s]
Tiempo de actualización del renglón: 0.000835[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

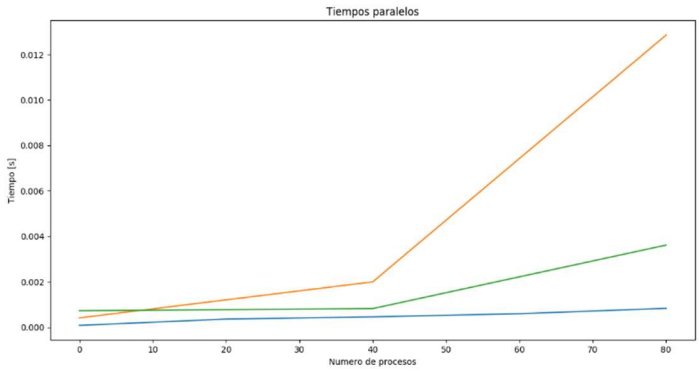
Matriz de 100 * 100

```
Turno: 0
Población: 2601
El tamaño del tablero es de : 100 * 100
Tiempo de actualización del sistema: 0.000000[s]
Tiempo de envio del tablero: 0.000000[s]
Tiempo de actualización del renglón: 0.000000[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

```
Turno: 10
Población: 1800
El tamaño del tablero es de : 100 * 100
Tiempo de actualización del sistema: 0.012853[s]
Tiempo de envio del tablero: 0.003617[s]
Tiempo de actualización del renglón: 0.003629[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

```
Turno: 50
Población: 1223
El tamaño del tablero es de : 100 * 100
Tiempo de actualización del sistema: 0.008916[s]
Tiempo de envio del tablero: 0.004191[s]
Tiempo de actualización del renglón: 0.004206[s]
¿Qué desea hacer?
1) Continua
0) Salir
```

Gráfica



8. Conclusión

En conclusión, se aprecian las diferencias en el juego de la vida al ejecutarlo en secuencial y en paralelo lo que se aprecia en el tiempo de procesamiento ya que al dividir la carga de trabajo del procesador a los demás hace que vaya más rápido y trabaje menos lo que nos daría un alargamiento en la vida útil de cada procesador.

9. Referencias.

Autor desconocido. Introducción a MPI. Consultado el 23 de noviembre de 2019, de http://informatica.uv.es/iiguia/ALP/materiales2005/2_2_introMPI.htm

HAGER, WELLEIM. Introduction to High Performance Computing for Scientists and Engineers. CRC Press. USA.