Proyecto final: Sistemas Distribuidos.

Flores García Karina, Herrera Gandarela Gabriel Alejandro
García García Ricardo y Hernandez Rendón Luis Roberto

***Abstract*— This work focuses on different topics related to distributed systems, we check the performance of different architectures applied to the same problem using exclusive performance metrics for concurrent systems, we also get different ideas about which are the situations in which it is favorable to parallelize an algorithm based on your data and the complexity of the solution, emphasizing the advantages and disadvantages obtained by its implementation.**

I. OBJETIVOS.

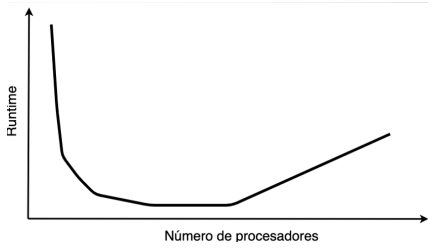
- Poner en práctica los patrones arquitectónicos aprendidos a lo largo del semestre.
- Implementar el paso de mensajes colectivamente.
- Implementar el paso de mensajes punto a punto.
- Utilizar MPI para la resolución de problemas.

II. INTRODUCCIÓN.

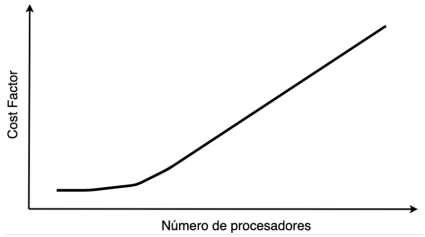
Métricas de desempeño.

Cuando se implementan soluciones en paralelo surge la pregunta de que tanto mejoro el tiempo de resolución comparado con la solución secuencial (un procesador). Para ello es necesario haber implementado primero la solución secuencial y valorar si la aceleración compensa la inversión en el sistema paralelo, de no ser así puede buscar la técnica para optimizar código y ahorrar costo y tiempo.
Al saber que se tiene la mejor solución en secuencial podemos compararla contra la paralela.
Las mediciones no resultan ser exactas sino un promedio, ya que existen más factores que influyen como la asignación de recursos del sistema operativo o la cantidad de memoria que posea, entre otras cosas.
Las métricas son las siguientes:

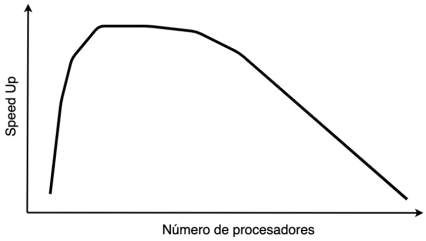
- RUNTIME: Hace referencia al tiempo que pasa entre el inicio del programa y la finalización de todos los procesos *t(p)*.



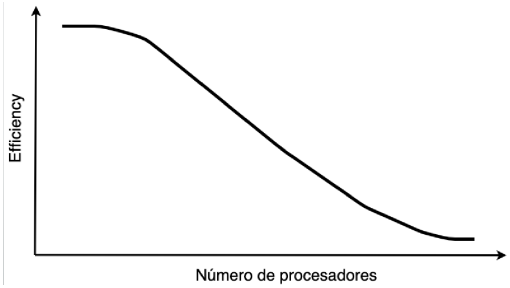
- COST FACTOR: Representa la cantidad de trabajo realizado por el programa, ya que, aunque disminuye el tiempo al tener un sistema en paralelo, los procesadores se encuentran trabajando al mismo tiempo y esto conlleva a contemplar otros costos del algoritmo. *C(p)= de procesadores * runtime con p procesadores*



- SPEEDUP FACTOR: Medición relativa del rendimiento de un programa en paralelo. Las mediciones siempre deben hacerse con la misma cantidad de datos.



- SPEEDUP EN PORCENTAJE: Da el porcentaje del speedup respecto al programa en un solo procesador.
- EFFICENCY: Cuanto tarda su procesador en realizar su tarea. Regularmente se expresa en porcentaje.



Sección serial y Tiempo de procesamiento.

La estructura secuencial es aquella en la que una acción (instrucción) sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el fin del proceso.
El tiempode procesamiento es el tiempo que tarda en ejecutarse un programa, sin tener en cuenta el tiempo de espera debido a la E/S o el tiempo utilizado para ejecutar otros programas.

MPI

Es una estandarización de paso de mensajes donde se define el estandar y la sintaxis de aquellos patrones que se pueden comunicar.
Al inicio de cualquier programa se requiere una parte secuencial, la cual se definiran las variables y se inicializará MPI. Al realizar lo anterior, se abre un comunicador, que basicamenten es la colección de procesadores que se van a comunicar entre sí, cada procesador tiene un ID único, esto es para que se diferencien. A este ID, se le denominá como rango o *rank*. Cada procesador puede enviar y recibir de y hacía cualquier otro procesador.
MPI, funciona con los lenguajes C, C++ y Fortran. Cada una varía de acuerdo a la sintaxis que manejan, sin embargo, tienen la misma funcionalidad.
Con MPI se pueden implementar los distintos tipos de comunicación como se presentarán a continuación.

Paso de mensajes síncrono y asíncrono.

Síncrono: Se denomina paso de mensajes síncrono cuando el que emite mensaje queda bloqueado hasta que llegue al receptor, mientras que el paso de mensajes asíncrono ocurre cuando el emisor manda el mensaje, y continua con la ejecución del programa sin esperar a que el receptor tome el mensaje.

Asíncrono: El emisor puede enviar uno o más mensajes sin bloquearse; el receptor lo puede recoger más tarde.

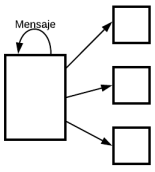
Tipos de comunicación.

***Punto a punto**:*

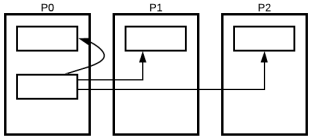
- SEND: El procesador envía el mensaje a la red.
- RECEIVE: El procesador recibe un mensaje de la red.

***Colectiva**:*

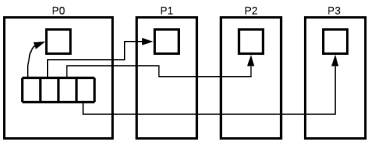
- MULTICAST: Un procesador envía a todos los procesadores sin distinción alguna.



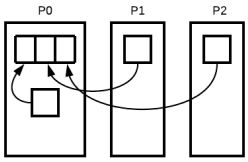
- BROADCAST: Un procesador envía a un grupo de procesadores, los cuales deben ser identificados por el que envía.



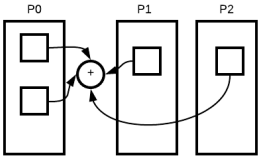
- SEATTER: Crea un arreglo y casa elemento lo distribuye.



- GATTER: Contrario a scatter, el proceso maestro recibe en un arreglo los elementos de la red.



- REDUCE: Tipo de Gather, pero aplica operación aritmética o lógica.

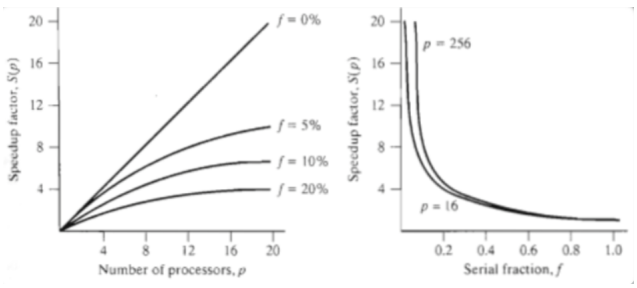


Evaluación de problemas en paralelo.

La evaluación que se debe aplicar inicialmente es para saber que tan rápido es la implementación en paralelo.
Inicialmente debemos calcular el tiempo de ejecución secuencial y posteriormente, para el algoritmo en paralelo, debemos estimar el número de pasos o procesos, para así calcular la cantidad de comunicación.

$$tp = t_{comm} + t_{comp} \quad (1)$$

Podemos observar que entre mayor sea sección serial, menor será la aceleración (Speedup) y no tendrá caso paralelizar las tareas



Tiempo de comunicación.

Dependerá del número de mensajes,tamaño de los mensajes,modo de transferencia y la estructura de la interconexión.
El tiempo de comunicación total será calculado por la sumatoria del tiempo de cada uno de los tiempos parciales.

$$t_{comm} = t_{comm1} + t_{comm2} + t_{comm3} + ... + t_{commn} \quad (2)$$

Tiempo de computación.

Al existir una sección secuencial, se debe tomar en cuenta en el análisis de rendimiento, tanto la sección paralela como la sección secuencial, ya que es una parte que siempre va a mantenerse constante.
Esta situación está contemplada en la ecuación conocida como "LEY DE AMADAHL".

$$s(p) = \frac{ts}{fts + (1-f)ts/p} = \frac{p}{1+(p-1)f} \qquad (3)$$

Donde f expresa la fracción de procesamiento que no puede ser dividido en tareas paralelas.

Complejidad de algoritmos. (Notación Big-O)

La notación Big O es una herramienta muy funcional para determinar la complejidad de un algoritmo que estemos utilizando, permitiéndonos medir su rendimiento en cuanto a uso de espacio en disco, recursos (memoria y ciclos del reloj del CPU) y tiempo de ejecución, entre otras, ayudándonos a identificar el peor escenario donde el algoritmo llegue a su más alto punto de exigencia.
Algunas de las notaciones Big-O más comunes son:

- O(1): CONSTANTE. La operación no depende del tamaño de los datos. Es el caso ideal, pero a la vez probablemente el menos frecuente.
- O(N): LINEAL.El tiempo de ejecución es directamente proporcional al tamaño de los datos. Crece en una línea recta.
- O(LOG N): LOGARÍTMICA. Por regla general se asocia con algoritmos que dividen el problema para abordarlo.
- O(NLOGN): en este caso se trata de funciones similares a las anteriores, pero que rompen el problema en varios trozos por cada elemento, volviendo a recomponer información tras la ejecución de cada "trozo".
- O(N2): CUADRÁTICA.Es típico de algoritmos que necesitan realizar una iteración por todos los elementos en cada uno de los elementos a procesar.
- O(2N): EXPONENCIAL. Se trata de funciones que duplican su complejidad con cada elemento añadido al procesamiento.

Arquitecturas en paralelo.

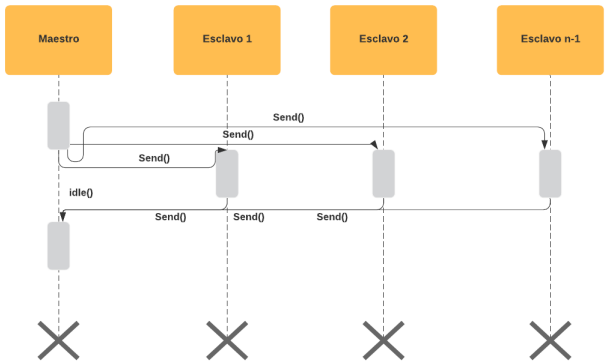
- ESTRELLA: Es de tipo emborrow ssingly parallel (paralelismo engorroso) ya que los procesadores no necesitan comunicación entre ellos. Se trata de un maestro y un grupo de esclavos.

MASTER

```
dato_esc = datos/esclavos();
for(i=0;i<total_esclavos;i++){
    send(dato_esc,Pi);
}
for(i=0;i<total_esclavos;i++){
    recv(datos_n,Pi_any);
}
usa_resultados();
```

SLAVE

```
recv(dato,Pmaster);
opera_dato();
send(dato,Pmaster);
```



- GRANJA O POOL DE TRABAJO:Similar al de estrella pero el reparto de las asignaciones es dinamica, inicialmente "llena" los procesadores de trabajo y conforme vayan terminando se les asigna uno nuevo.

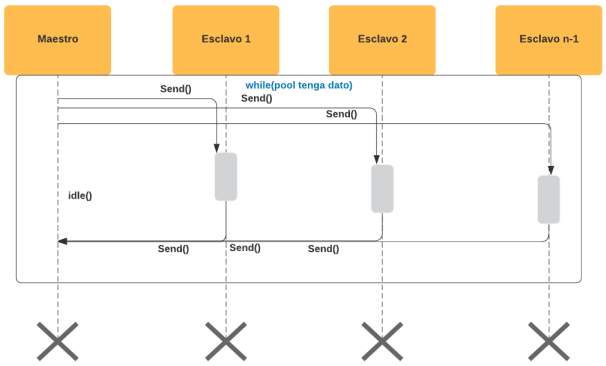
MASTER

```
count = 0;
pool_datos;
dat_operado = 0;

for(i=0; i<total_esclavos;i++){
    send(dato,Pi,dato_tag);
    count++;
    dat_operado++;
}
do{
    recv(resultado,Pany,result_tag);
    count--;
    if(pool_datos.tieneDatos){
        send(dato,Pany,dato_tag);
        dat_operado++;
        count++;
    }else{
        send(NULL,Pany,terminanor_tag);
    }
}while(count>0)
usa_resultado();
```

SLAVE

```
recv(dato,Pmaster,source_tag);
while(source_tag==data_tag){
    opera_dato();
    send(resultado,Pmaster,result_tag);
    recv(datomPmaster,source_tag);
}
```



- ÁRBOL: Obedece a estrategias divide y venceras. Parte el problema en pedazos (chunks) más pequeños, aquí sí hay interacción entre nodos para combinar lo obtenido y entregar un resultado global.
  Es necesario conocer su identificador, la profundidad, quién es su padre y quién sus hijos.
  Debe ser un árbol balanceado.

ROOT

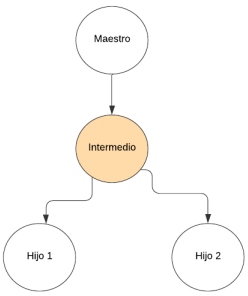
```
separa_dato();
for(i=0;i<total_hijos;i++){
    send(dato,Pi);
}
for(i=0;i<total_hijos;i++){
    recv(dato,Pi);
}
une_resultados();
```

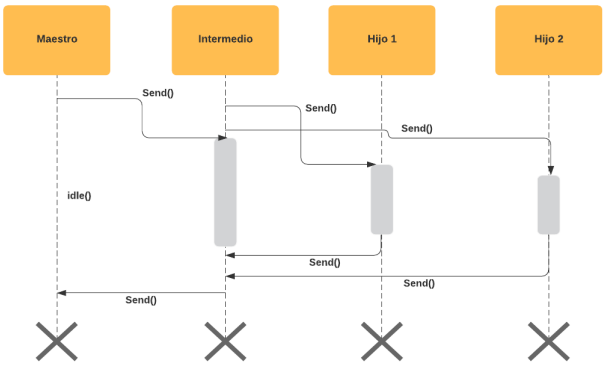
INTERMEDIO

```
recv(dato,Ppadre);
separa_datos();
for(i=0;i<total_hijos;i++){
    send(dato,Pi);
}
for(i=0;i<total_hijos;i++){
    recv(dato,Pi);
}
une_resultados();
send(dato,Ppadre);
```

HOJA

```
recv(dato,Ppadre);
opera();
send(dato,Ppadre);
```





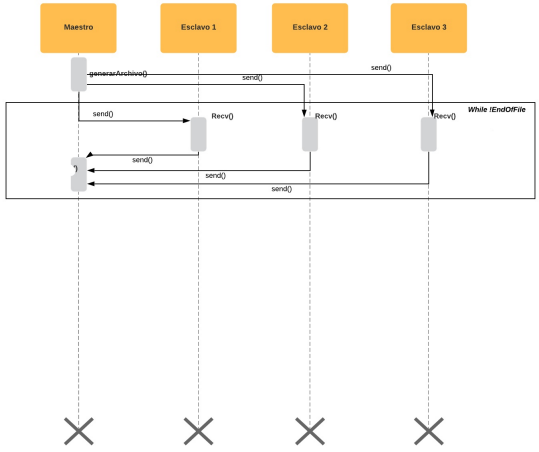
- PIPLINE: Es una técnica que divide el problema en una serie de tareas secuenciales. Cada tarea secuencial es realizada por un procesador independiente. Cada procesador ejecuta una fracción del algoritmo.
  Para llegar a su máxima eficiencia operacional se debe llenar la tubería para que a cada ciclo del reloj, salga un resultado.

MASTER

```
pool_datos;
dato_operador++;
do{
    send(dato,Pi,data_tag);
    count--;
    if(pool_datos.tieneDatos){
        send(datos,Pslave,data_tag);
        pool_datos--;
    }else{
        send(NULL,Pslave,terminator_tag);
    }
}while(pool.tengaDatos);
```

SLAVE

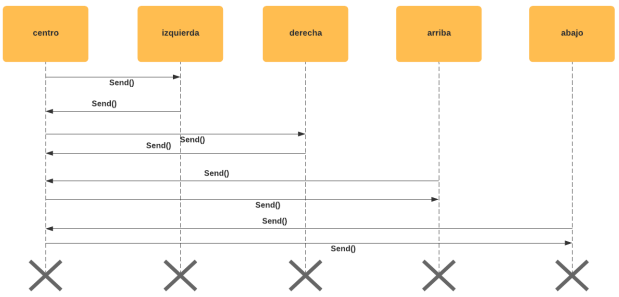
```
recv(dato,point,source_tag);
while(source_tag==dato_tag){
    opera_dato();
    send(resultado,Psig,result_tag);
    recv(dato,point,save_tag);
}
```



- MALLA: Este tipo de estructuras parten del problema en una matriz, donde cada sección necesita que sus vecinos les proporcionen los datos calculados, y así iterar hasta que se encuentre un resultado.

Master

```
while(resultado_no_calculado){
    recv(dato,izquierda);
    send(dato,derecha);
    recv(dato,derecha);
    send(dato,izquierda);
    recv(dato,arriba);
    send(dato, abajo);
    recv(dato,abajo);
    send(dato,arriba);
    opera();
}
```



III. ANÁLISIS DEL PROBLEMA.

ESTRELLA, POOL Y PIPELINE

Al entender el funcionamiento del algoritmo de estas tres estructuras, aduecuamos el programa de tal forma que, sea capaz de procesar tantos datos en paralelo. Se piensa que la lectura se por tandas de 100000 elementos y esta se repita hasta cumplir la cantidad de datos que se piden.Una vez leídos los datos por tandas la resolución será especial para cada arquitectura.

ÁRBOL

El problema requiere implementar una estructura de árbol que facilitara la clasificación de un conjunto de datos, para ello es necesario conocer el valor del elemento a clasificar leyéndolo del archivo, después se realiza la clasificación haciendo que el valor recorriera la estructura del árbol hasta llegar a una de las hojas donde finalmente sería almacenado en un archivo de acuerdo a su valor. Se tiene un archivo donde se almacenaran los datos a clasificar, para la versión en paralelo se contaran con 32 procesos para representar la estructura de árbol, por lo que será imposible variar el número de procesos, por lo que se decidió variar el números de datos a procesar para poder medir el rendimiento del programa.

MALLA

El problema consiste en implementar un algoritmo que permita comunicar los elementos de una matriz, cada nodo de la matriz tendrá un carácter aleatorio del alfabeto contando las minúsculas o mayúsculas, de esta manera de cada uno de los nodos obtendrá la información de los elementos adyacentes o de sus vecinos, siguiendo un orden, el cual consiste en visitar primero al nodo vecino ubicado en la parte superior derecha, para después visitar a sus demás vecinos en sentido horario, una vez que se visiten todos sus nodos se formará una cadena de caracteres, la cual está formada por los caracteres de los vecinos visitados. La comunicación de elementos de forma paralela requiere que el envío y recepción de mensajes estén sincronizados de tal manera que se garantice que no entrarán en un estado de bloqueo. Para medir el rendimiento de este algoritmo se variara el tamaño de la matriz, y para el programa paralelo cada elemento de la matriz representará un procesos diferente.

IV. IMPLEMENTACIÓN

A. *Desarrollo*

PIPELINE

La implementación en paralelo consiste en leer la base de datos que contiene todos los elementos a procesar en pequeños grupos, debido a que por el tamaño del archivo y la cantidad de elementos no es posible procesarlos todos a la vez.
Una vez que se lee una pequeña parte de la base se va enviando un solo elemento a la vez al procesador siguiente para que comienza la operación que en este caso es elevar el número a la potencia que le corresponda. Cuando el procesador realizo su trabajo con el elemento se lo pasa al siguiente procesador para que realice la misma tarea pero con la potencia siguiente y además le suma la potencia enviada por el elemento anterior. Esto sucede con 9 procesadores y el último restante, escribe el resultado en otra base que cuando este llena será enviada al procesador maestro.
Por último, el procesador maestro recibe la base con las potencias y las escribe sobre el archivo.

ESTRELLA

Al igual que en pipeline la implementación en paralelo consiste en leer la base de datos que contiene todos los elementos a procesar en pequeños grupos, por la misma razón antes mencionada. La diferencia ahora radica en que cada procesador solamente tiene comunicación con el proceso maestro, nunca hay comunicación entre los esclavos, por lo que un solo esclavo debe hacer la operación completa para el calculo de la suma de las potencias. A cada procesador opera unicamente sobre un pedazo de la base de datos leída y todos resuelven el mismo número de elementos. Cada uno guarda el resultado sobre una nueva base en la misma posición de la que obtuvo el elemento y se la envía al maestro.
El maestro escribe los resultados sobre el archivo.

POOL

Al igual que en los 2 patrones anteriores leímos la base de datos por pedazos y la procesamos, para esta ocasión ninguno de los esclavos tiene comunicación entre ellos, unicamente con el maestro y solamente pueden procesar un dato a la vez. La cantidad de elementos que procesa cada esclavo es dinámica, así que el control lo lleva el maestro asegurandose de enviar un dato a cada procesador que se libere mientras haya datos disponibles en la base leída.
El resultado se le envía al maestro y él se encarga de escribir los resultados sobre el archivo. Que un solo procesador escriba sobre el archivo, evita la condición de carrera.

ÁRBOL

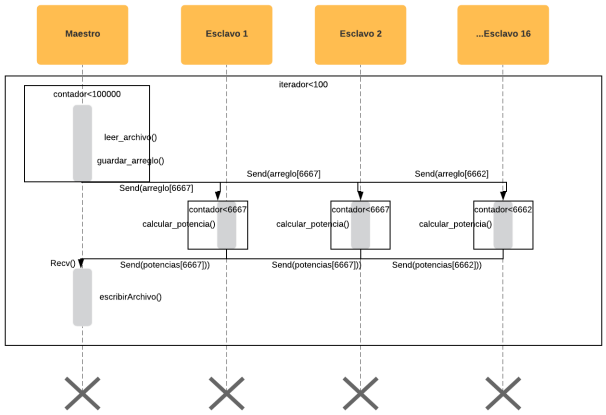
La implementación en paralelo consiste en el uso de 32 nodos, las tareas de cada uno de ellos se asignan de la siguiente manera: el nodo 0, tendrá la tarea de leer el valor a clasificar de los archivos y enviarlo a la raíz del árbol (el uso de este nodo previene que la memoria no se sature), el árbol es de tipo binario, compuesto por 31 nodos, la decisión de un árbol binario se toma debido a los valores que puede tomar el elemento, mayor o menor, a un discriminante dentro de cada nodo rama, hasta que finalmente en los nodos hoja se conociera la clase del elemento y se pudiera realizar la escritura en un nuevo archivo.
Dentro del programa en paralelo se aprovechan las propiedades de un árbol binario para asignar las tareas que debe hacer cada uno de los nodos, primero dividiéndolos por niveles y posteriormente haciendo uso de condicionales para especificar cuál sería el discriminante en cada uno de los nodos así como el valor del nodo destino calculado a partir del identificador del nodo padre y el valor del elemento.
En cuanto al envío de mensajes el nodo cero se dedican a enviar mensajes hasta completar el conjunto de datos, posteriormente, envía un último mensaje para indicar que no hay más datos y los nodos restantes deben terminar de procesar los datos. Debido a que se desconoce el número de elementos que llegarán a cada nodo, se espera el primer mensaje, se compara y envía el mensaje al nodo correspondiente y se lleva el nodo a espera de un nuevo mensaje, esto se realiza de un ciclo indefinido hasta que se recibe del nodo padre la señal que indica que no hay más datos a procesar y el nodo debe dejar de esperar mensajes, cuando esto ocurre el mismo nodo envía los mensajes de finalización a sus nodos hijos

MALLA

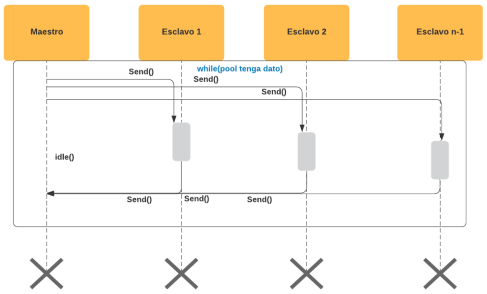
Cada uno de los nodos representará un elemento de la matriz, debido a la forma de la estructura no todos los elementos tienen el mismo número de vecinos, por lo que es necesario que por nodo se calculen cuáles son sus vecinos, esto se hace evaluando la posición del el nodo dentro de la matriz (por sus índices) y a través de banderas. Haciendo uso de las banderas se decide si ese nodo debería enviar o recibir un mensaje del nodo vecino, de igual manera se calcula el valor del nodo adyacente para elegir correctamente el destinatario o remitente.
La evaluación del envío de mensaje sigue un orden específico que garantiza que en ningún momento se llegará a una condición de bloqueo, los mensajes que se intercambian contienen un carácter que representa la etiqueta del nodo vecino, cuando todos los nodos recuperan las etiquetas envían las cadenas a el nodo maestro, que es el encargado de escribir en un archivo todas las cadenas calculadas, se decide de esta manera para evitar la gestión de acceso al archivo cuando requiere ser escrito por diferentes nodos.

B. Diagrama UML de secuencia

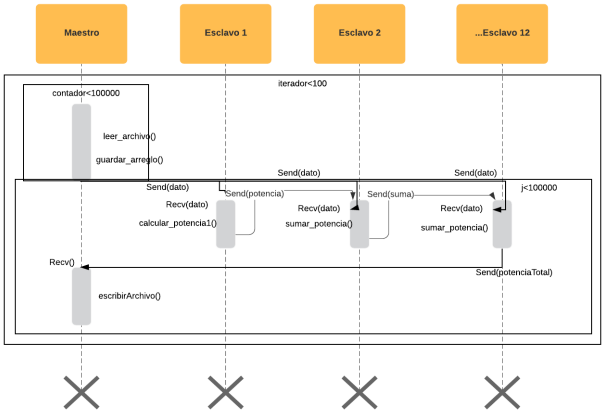
ESTRELLA

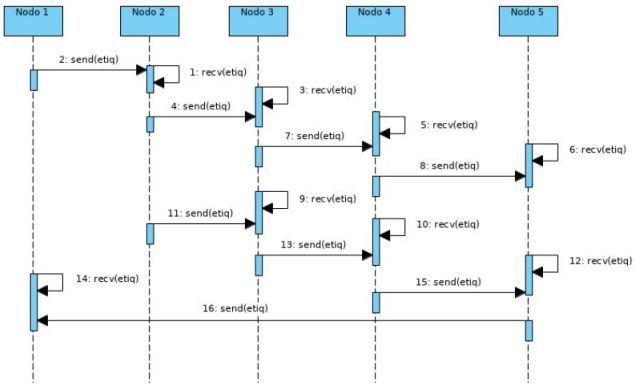


POOL



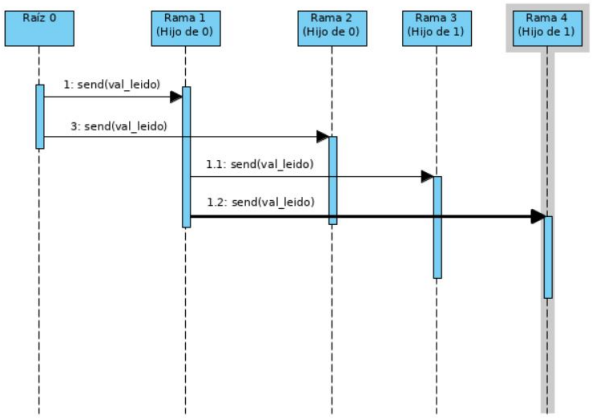
PIPELINE (TIPO 4)



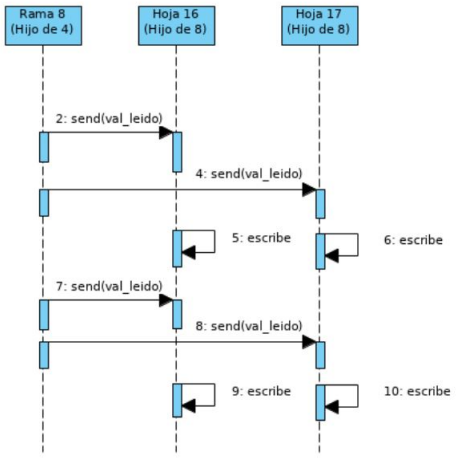


ÁRBOL

Ramas



Hoja



MALLA

V. CÓDIGO FUENTE

POOL,MALLA Y PIPELINE

PROGRAMA SECUENCIAL.

```
if(archivo == NULL){
  printf("Error en la apertura del archivo");
}else{
  char linea[1024];
  while(i<100){
    num_elementos = 0;
    while(num_elementos<100000){
      fgets(linea,1024,archivo);
      dato = strtod(linea,NULL);
      bd[num_elementos] = dato;
      num_elementos++;
    }
    for(int k=0; k<100000;k++){
      potencia = 0;
      numero = bd[k];
      for(int j=0; j<10;j++){
        potencia = pow(numero,j) + potencia;
      }
      bd_final[k] = potencia;
    }
    if(archivo_final == NULL){
      printf("Error abriendo el archivo para las
potencias\n");
    }else{
        for(int j=0 ; j<100000; j++){
          valor = bd_final[j];
            fprintf(archivo_final,"%.1f\n", valor
);
        }
    }
    i++;
  }
}
```

PIPELINE

PROGRAMA PARALELO.

```
if(archivo == NULL){
      return 1;
}else{
  char linea[1024];
  while(i<2){
    if(pid == 0){
      num_elementos = 0;
      while(num_elementos<100000){
        fgets(linea,1024,archivo);
        tag = 0;
```

```
        dato = strtod(linea,NULL);
        bd[num_elementos] = dato;
        if(num_elementos==0){
          printf("Dato[%d] %f\n",num_elementos,
    dato);
        }
        num_elementos++;
      }
      for(int k = 0;k<100000;k++){
        for(int j = 1;j<12;j++){
          elemento = bd[k];
          MPI_Ssend(&elemento,1,MPI_DOUBLE, j,
    tag, MPI_COMM_WORLD);
        }
      }
    }
    for(int k=0; k<100000;k++){
      if(pid==1){
        tag = 0;
        MPI_Recv(&elemento,1,MPI_DOUBLE, 0, tag,
    MPI_COMM_WORLD, &info);
        potencia = pow(elemento,0);
        MPI_Ssend(&potencia,1,MPI_DOUBLE, pid+1,
    tag, MPI_COMM_WORLD);
      }else if(pid==2){
        tag = 0;
        MPI_Recv(&elemento,1,MPI_DOUBLE, 0, tag,
    MPI_COMM_WORLD, &info);
        MPI_Recv(&potencia,1,MPI_DOUBLE, pid-1,
    tag, MPI_COMM_WORLD, &info);
        potencia = pow(elemento,1) + potencia;
        MPI_Ssend(&potencia,1,MPI_DOUBLE, pid+1,
    tag, MPI_COMM_WORLD);
      }else if(pid==3){
        tag = 0;
        MPI_Recv(&elemento,1,MPI_DOUBLE, 0, tag,
    MPI_COMM_WORLD, &info);
        MPI_Recv(&potencia,1,MPI_DOUBLE, pid-1,
    tag, MPI_COMM_WORLD, &info);
        potencia = pow(elemento,2) + potencia;
        MPI_Ssend(&potencia,1,MPI_DOUBLE, pid+1,
    tag, MPI_COMM_WORLD);
      }
      //Hace el mismo proceso hasta el pid=11
    pero con su respectiva potencia
    }

    if(pid==0){
      MPI_Recv(&bd_final,100000,MPI_DOUBLE, 11,
    tag, MPI_COMM_WORLD, &info);
      if(archivo_final == NULL){
        printf("Error en la apertura del archivo
    potencias\n");
      }else{
        for(int j=0 ; j<100000; j++){
          dato = bd_final[j];
            fprintf(archivo_final,"%f\n", dato)
    ;
        }
      }
    }

    i++;
  }
```

ESTRELLA

PROGRAMA PARALELO.

```
if(archivo == NULL){
  return 1;
}else{
  char linea[1024];
  while(i<100){
    suma = 0;
    if(pid == 0){
      num_elementos = 0;
      while(num_elementos<100000){
        fgets(linea,1024,archivo);
        tag = 0;
        dato = strtod(linea,NULL);
        bd[num_elementos] = dato;
        num_elementos++;
      }
      for(int j = 1;j<16;j++){

        MPI_Ssend(&bd,sizeof(bd)/sizeof(bd[0]),
    MPI_DOUBLE, j, tag, MPI_COMM_WORLD);
      }
    }

      if(pid==1){
        tag = 0;
        int inicio = (pid-1)*6667;
        int frontera = inicio + 6667;
    MPI_Recv(&bd,sizeof(bd)/sizeof(bd[0]),
    MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &info);
        it = 0;
        for(int k = inicio; k<frontera; k++){
          double aux = 0;
          for(int p=0; p<10; p++){
            aux = aux + pow(bd[k], p);
          }
          potencias[it]=aux;
          it++;
        }
        MPI_Send(&potencias,6667,MPI_DOUBLE, 0,
    tag, MPI_COMM_WORLD);

      }else if(pid==2){
        tag = 0;
        int inicio = (pid-1)*6667;
        int frontera = inicio + 6667;
        MPI_Recv(&bd,sizeof(bd)/sizeof(bd[0]),
    MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &info);
        it = 0;
        for(int k = inicio; k<frontera; k++){
          double aux = 0;
          for(int p=0; p<10; p++){
            aux = aux + pow(bd[k], p);
          }
          potencias[it]=aux;
          it++;
        }
        MPI_Send(&potencias,6667,MPI_DOUBLE, 0,
    tag, MPI_COMM_WORLD);

      }//Hace lo mismo hasta el pid=15 pero con
    su respectiva parte del arreglo calculado como
    inicio y frontera

    if(pid==0){
      int lim=0;
      for(int i =1; i<size;i++){
        if(i==15){
          lim=6662;
          MPI_Recv(&potencias,6662,MPI_DOUBLE, i,
     tag, MPI_COMM_WORLD, &info);
        }else{
          lim=6667;
            MPI_Recv(&potencias,6667,MPI_DOUBLE,
    i, tag, MPI_COMM_WORLD, &info);
          }
            if(archivo_final == NULL){
          perror("Error en la apertura del
    archivo");
          return 1;
        }else{
          for(int c=0; c<lim;c++){
```

```
            valor = potencias[c];
            fprintf(archivo_final,"%f\n", valor);
          }
        }
```

POOL

PROGRAMA PARALELO.

```
if(archivo == NULL){
  perror("Error en la apertura del archivo");
  return 1;
}else{
  char linea[1024];
  while(i<100){
    count_datos = 0;
    count_proc = 0;

    if(pid == 0){
      printf("Soy el maestro con pid[%d] y estoy leyendo datos\n",pid );
      num_elementos = 0;
      while(num_elementos<100000){
        fgets(linea,1024,archivo);
        dato = strtod(linea,NULL);
        bd[num_elementos] = dato;
        num_elementos++;
      }
      iterador = 0;
      for(int j = 1;j < size; j++){
        data_tag = 0;
        source_tag=0;
        valor = bd[iterador];

        MPI_Ssend(&valor,1,MPI_DOUBLE, j, data_tag, MPI_COMM_WORLD);
        count_proc++;
        count_datos++;
        printf("Ya lo mande: %d\n", count_proc);
        iterador++;
      }
    }

      if(pid == 1){
      data_tag = 0;
      source_tag = 0;
      result_tag = 3;
      l++;
      MPI_Recv(&valor,1,MPI_DOUBLE,0,source_tag, MPI_COMM_WORLD, &info);
      while(source_tag == data_tag){
        potencia = 0;
        for(int k = 0; k < 10; k++){
          potencia = potencia + pow(valor,k);
        }
        MPI_Send(&potencia,1,MPI_DOUBLE,0, result_tag, MPI_COMM_WORLD);
        MPI_Recv(&valor,1,MPI_DOUBLE,0,source_tag, MPI_COMM_WORLD, &info);
      }
    }else if(pid == 2){
      data_tag = 0;
      source_tag = 0;
      result_tag = 3;
      MPI_Recv(&valor,1,MPI_DOUBLE,0,source_tag, MPI_COMM_WORLD, &info);
      while(source_tag == data_tag){
        potencia = 0;
        for(int k = 0; k < 10; k++){
          potencia = potencia + pow(valor,k);
        }
        MPI_Send(&potencia,1,MPI_DOUBLE,0, result_tag, MPI_COMM_WORLD);
        MPI_Recv(&valor,1,MPI_DOUBLE,0,source_tag, MPI_COMM_WORLD, &info);
      }
    }//hace lo mismo hasta el procesador numero 15
    }

      if(pid==0){
      while(count_proc>1){
        result_tag = 3;
        data_tag = 0;
        MPI_Recv(&potencia,1,MPI_DOUBLE, MPI_ANY_SOURCE,result_tag, MPI_COMM_WORLD, &info);
        int ps = info.MPI_SOURCE;
        fprintf(archivo_final,"%.1f\n", potencia);
        count_proc--;
        if(count_datos < 100000){
          valor = bd[count_datos];
          MPI_Send(&valor,1,MPI_DOUBLE,ps, data_tag, MPI_COMM_WORLD);
          count_datos++;
          count_proc++;
        }else{
          valor = 0;
          source_tag = 1;
          MPI_Send(&valor,1,MPI_DOUBLE,ps, source_tag, MPI_COMM_WORLD);

        }
    }
      i++;

}
```

ÁRBOL

PROGRAMA SECUENCIAL.

```
if(fp == NULL){
  printf("error al crear archivo\n");
}else{
  get_walltime(&T_inicio);
  for(j = 0; j<100000; j++){//numero de lotes
    for(i = 0 ; i< 10000; i++){
      fgets(dato_leido,100,fp);
      datos_leidos[i] = atoi(dato_leido);
    }
    for(i = 0; i <10000; i++){
      if(datos_leidos[i] <= 8){//nodo 1
        if(datos_leidos[i] <= 4){//nodo 3
          if(datos_leidos[i] <= 2){//nodo 7
            if(datos_leidos[i] == 1){//nodo 15
              //printf("es un 1\n");
              fprintf(fp1, "%d\n",datos_leidos[i]);
            }
            if(datos_leidos[i] == 2){//nodo 16
              fprintf(fp2, "%d\n",datos_leidos[i]);
            }
          }
          if(datos_leidos[i] > 2){//nodo 8
            if(datos_leidos[i] == 3){//nodo 17
              fprintf(fp3, "%d\n",datos_leidos[i]);
            }
            if(datos_leidos[i] == 4){//nodo 18
              fprintf(fp4, "%d\n",datos_leidos[i]);
            }
          }
        }
        if (datos_leidos[i] > 4){//nodo 4
          if(datos_leidos[i] <= 6){//nodo 9
```

```
            if(datos_leidos[i] == 5){//nodo 19
              fprintf(fp5, "%d\n",datos_leidos[i
]);
            }
            if(datos_leidos[i] == 6){//nodo 20
              fprintf(fp6, "%d\n",datos_leidos[i
]);
            }
          }
          if(datos_leidos[i] > 6){//nodo 10
            if(datos_leidos[i] == 7){//nodo 21
              fprintf(fp7, "%d\n",datos_leidos[i
]);
            }
            if(datos_leidos[i] == 8){//nodo 22
              fprintf(fp8, "%d\n",datos_leidos[i
]);
            }
          }
        }
      }
      if(datos_leidos[i] > 8){//nodo 2
        if(datos_leidos[i] <= 12){//nodo 5
          if(datos_leidos[i] <= 10){//nodo 11
            if(datos_leidos[i] == 9){//nodo 23
              fprintf(fp9, "%d\n",datos_leidos[i
]);
            }
            if(datos_leidos[i] == 10){//nodo 24
              fprintf(fp10, "%d\n",datos_leidos[i
]);
            }
          }
          if(datos_leidos[i] > 10){//nodo 12
            if(datos_leidos[i] == 11){//nodo 25
              fprintf(fp11, "%d\n",datos_leidos[i
]);
            }
            if(datos_leidos[i] == 12){//nodo 26
              fprintf(fp12, "%d\n",datos_leidos[i
]);
            }
          }
        }
        if(datos_leidos[i] > 12){//nodo 6
          if(datos_leidos[i] <= 14){//nodo 13
            if(datos_leidos[i] == 13){//nodo 27
              fprintf(fp13, "%d\n",datos_leidos[i
]);
            }
            if(datos_leidos[i] == 14){//nodo 28
              fprintf(fp14, "%d\n",datos_leidos[i
]);
            }
          }
          if(datos_leidos[i] > 14){//nodo 14
            if(datos_leidos[i] == 15){//nodo 29
              fprintf(fp15, "%d\n",datos_leidos[i
]);
            }
            if(datos_leidos[i] == 16){//nodo 30
              fprintf(fp16, "%d\n",datos_leidos[i
]);
            }
          }
```

PROGRAMA EN PARALELO.

```
/*Los niveles y id comienzan en 1*/
int elemento_profundidad,nivel, nodo_id, val_recv
  , nodo_dest, nodo_remi;
if(rank != 0){
  nodo_id = rank;
  nivel = floor(log(rank)/log(2) + 1);
  elemento_profundidad = rank + 1 - pow(2,nivel
  -1);
}


if (rank == ROOT) {
  if(datos == NULL){
    return -1;
  }else{
    for(j = 0; j< NUM_LOTES; j++){
      for(i = 0; i< NUM_DATOSxLOTE; i++){
        fgets(dato,100,datos);
        datos_leidos[i] = atoi(dato);
      }
      MPI_Send(datos_leidos,NUM_DATOSxLOTE,
  MPI_INT,1,TAG_DAT,MPI_COMM_WORLD);
    }
  }

}

switch(nivel){
  case 1:
    int datos_leidos[NUM_DATOSxLOTE];
    for (int j = 0; j < NUM_LOTES; j++){
        MPI_Recv(datos_leidos, NUM_DATOSxLOTE,
  MPI_INT, 0, TAG_DAT, MPI_COMM_WORLD, &info);
      for (int i = 0; i < NUM_DATOSxLOTE; i++){
        nodo_dest = datos_leidos[i] <= 8 ? 2 :
  3;
        MPI_Send(&datos_leidos[i], 1, MPI_INT,
  nodo_dest , TAG_DAT, MPI_COMM_WORLD);
      }
    }
    val_recv = 0;
    MPI_Send(&val_recv, 1, MPI_INT, 2 , TAG_DAT,
  MPI_COMM_WORLD);
    MPI_Send(&val_recv, 1, MPI_INT, 3 , TAG_DAT,
  MPI_COMM_WORLD);
    break;
  case 2:
    nodo_remi = floor(nodo_id/2);
    MPI_Recv(&val_recv, 1, MPI_INT, nodo_remi,
  TAG_DAT, MPI_COMM_WORLD, &info);
    while(val_recv > 0 && val_recv <= 16){
      if (nodo_id == 2) {
        nodo_dest = val_recv <= 4 ? 4 : 5;
      }else {
        nodo_dest = val_recv <= 12 ? 6 : 7;
      }
      MPI_Send(&val_recv, 1, MPI_INT, nodo_dest ,
   TAG_DAT, MPI_COMM_WORLD);
      MPI_Recv(&val_recv, 1, MPI_INT, nodo_remi,
  TAG_DAT, MPI_COMM_WORLD, &info);
    }
    /*Env o de mensajes de finalizaci n*/
    val_recv = 0;
    MPI_Send(&val_recv, 1, MPI_INT, rank*2,
  TAG_DAT, MPI_COMM_WORLD);
    MPI_Send(&val_recv, 1, MPI_INT, rank*2+1,
  TAG_DAT, MPI_COMM_WORLD);
    break;
  case 3:
    nodo_remi = floor(nodo_id/2);
    MPI_Recv(&val_recv, 1, MPI_INT, nodo_remi,
  TAG_DAT, MPI_COMM_WORLD, &info);
    while (val_recv > 0 && val_recv <= 16) {
      if (nodo_id == 4) {
        nodo_dest = val_recv <= 2 ? 8 : 9;
      }else if(nodo_id == 5){
        nodo_dest = val_recv <= 6 ? 10 : 11;
      }else if (nodo_id == 6) {
        nodo_dest = val_recv <= 10 ? 12 : 13;
      }else if (nodo_id == 7) {
        nodo_dest = val_recv <= 14 ? 14 : 15;
      }
      MPI_Send(&val_recv, 1, MPI_INT, nodo_dest ,
```

```
      TAG_DAT, MPI_COMM_WORLD);
70          MPI_Recv(&val_recv, 1, MPI_INT, nodo_remi,
    TAG_DAT, MPI_COMM_WORLD, &info);
71        }
72        val_recv = 0;
73        MPI_Send(&val_recv, 1, MPI_INT, rank*2,
    TAG_DAT, MPI_COMM_WORLD);
74        MPI_Send(&val_recv, 1, MPI_INT, rank*2 +1 ,
    TAG_DAT, MPI_COMM_WORLD);
75        break;
76      case 4:
77        nodo_remi = floor(nodo_id/2);
78        MPI_Recv(&val_recv, 1, MPI_INT, nodo_remi,
    TAG_DAT, MPI_COMM_WORLD, &info);
79        while(val_recv > 0 && val_recv <= 16) {
80          MPI_Send(&val_recv, 1, MPI_INT, 15 +
    val_recv, TAG_DAT, MPI_COMM_WORLD);
81          MPI_Recv(&val_recv, 1, MPI_INT, nodo_remi,
    TAG_DAT, MPI_COMM_WORLD, &info);
82        }
83        /*Env o de mensajes de finalizaci n*/
84        val_recv = 0;
85        MPI_Send(&val_recv, 1, MPI_INT, rank*2 ,
    TAG_DAT, MPI_COMM_WORLD);
86        MPI_Send(&val_recv, 1, MPI_INT, rank*2 +1,
    TAG_DAT, MPI_COMM_WORLD);
87      break;
88
89      case 5:
90        nodo_remi = floor(nodo_id/2);
91        char nomFile[12];
92        char bufer[3];
93        strcpy(nomFile,"");
94        strcat(nomFile,"arbol_");//6
95        sprintf(bufer, "%d", rank - 15);
96        strcat(nomFile,bufer);
97        strcat(nomFile,".txt");//4
98        FILE *fp = fopen(nomFile,"w");
99        if(fp == NULL)
100         printf("error al crear archivo\n");
101
102       MPI_Recv(&val_recv, 1, MPI_INT, nodo_remi,
    TAG_DAT, MPI_COMM_WORLD, &info);
103
104       while(val_recv > 0 && val_recv <= 16) {
105         fprintf(fp, "%d \n", val_recv);
106         MPI_Recv(&val_recv, 1, MPI_INT, nodo_remi,
    TAG_DAT, MPI_COMM_WORLD, &info);
107       }
108       fclose(fp);
109   }
```

MALLA

PROGRAMA SECUENCIAL.

```
1  void vecinos(char **matriz[REN][COL], int i, int j,
       char palabra[10]){
2    int vecinosSI, vecinosAR, vecinosSD, vecinosDER,
       vecinosID, vecinosAB, vecinosII, vecinosIZQ;
3    strcat(palabra,&matriz[i][j]);
4    vecinosSI = 1;
5    vecinosAR = 1;
6    vecinosSD = 1;
7    vecinosDER = 1;
8    vecinosID = 1;
9    vecinosAB = 1;
10   vecinosII = 1;
11   vecinosIZQ = 1;
12
13   if(i == 0){
14     vecinosSI = 0;
15     vecinosAR = 0;
16     vecinosSD = 0;
17   }
18
19   if(i == REN-1){
20     vecinosID = 0;
21     vecinosAB = 0;
22     vecinosII = 0;
23   }
24   if(j == 0){
25       vecinosSI = 0;
26       vecinosII = 0;
27       vecinosIZQ = 0;
28   }
29   if(j == COL-1){
30       vecinosSD = 0;
31       vecinosDER = 0;
32       vecinosID = 0;
33   }
34
35   if(vecinosSI == 1){
36     strcat(palabra,&matriz[i-1][j-1]);
37
38   }
39   if(vecinosAR == 1){
40     strcat(palabra,&matriz[i-1][j]);
41   }
42   if(vecinosSD == 1){
43     strcat(palabra,&matriz[i-1][j+1]);
44   }
45   if(vecinosDER == 1){
46     strcat(palabra,&matriz[i][j+1]);
47   }
48   if(vecinosID == 1){
49     strcat(palabra,&matriz[i+1][j+1]);
50   }
51   if(vecinosAB == 1){
52     strcat(palabra,&matriz[i+1][j]);
53   }
54   if(vecinosII == 1){
55     strcat(palabra,&matriz[i+1][j-1]);
56   }
57   if(vecinosIZQ == 1){
58     strcat(palabra,&matriz[i][j-1]);
59   }
60
61 }
62 int main(int argc, char const *argv[])
63 {
64
65   if(fp == NULL){
66     printf("error al crear archivo\n");
67   }else{
68     get_walltime(&T_inicio);
69     fprintf(fp, "La matriz inicial: \n\n");
70     for (size_t i = 0; i < REN; i++) {
71       for (size_t j = 0; j < COL; j++) {
72         fprintf(fp, "%c ", matriz[i][j]);
73       }
74       fprintf(fp, "\n");
75     }
76     fprintf(fp, "\n");
77
78     for(i = 0 ; i< REN; i++){
79       for(j = 0 ; j< COL ; j++){
80
81           vecinos(matriz,i,j,palabra);
82           printf("%s\n", &palabra);
83           fprintf(fp, "%s\n", palabra);
84           strcpy(palabra,"");
85           printf("%s\n", &palabra);
86
87       }
88     }
89
90   }
```

PROGRAMA EN PARALELO.

```
  if (nodo_id == MAESTRO) {
    rbuf = (char *)malloc(size*10*sizeof(char));
    tInicio = MPI_Wtime();
  }
  MPI_Barrier(MPI_COMM_WORLD);

  char **matriz[REN][COL] = {{'a','M'},
                             {'G','e'}};

    MPI_Barrier(MPI_COMM_WORLD);

    vecinosSI = 1;
    vecinosAR = 1;
    //Lo mismo para los dem s vecinos

    if(i == 0){
      vecinosSI = 0;
      vecinosAR = 0;
      vecinosSD = 0;
    }

    if(i == REN-1){
      vecinosID = 0;
      vecinosAB = 0;
      vecinosII = 0;
    }
    if(j == 0){
        vecinosSI = 0;
        vecinosII = 0;
        vecinosIZQ = 0;
    }
    if(j == COL-1){
        vecinosSD = 0;
        vecinosDER = 0;
        vecinosID = 0;
    }


    if(vecinosSI == 1){
      nodoRemi = rank - orden - 1;
      MPI_Recv(&rec,1, MPI_CHAR, nodoRemi, TAG_DAT,
       MPI_COMM_WORLD, &info);
      palabra[eti_sig] = rec;
      eti_sig ++;
    }if(vecinosID == 1){
        nodoDest = rank + orden + 1;
        char *ma = matriz[i][j];
        MPI_Send(&ma,1,MPI_CHAR,nodoDest,TAG_DAT,
    MPI_COMM_WORLD);
      }
    if(vecinosAR == 1){
      nodoRemi = rank - orden;
      MPI_Recv(&rec,1, MPI_CHAR, nodoRemi, TAG_DAT,
       MPI_COMM_WORLD, &info);
      palabra[eti_sig] = rec;
      eti_sig ++;
    }if(vecinosAB == 1){
        nodoDest = rank + orden;
        char *ma = matriz[i][j];
        MPI_Send(&ma,1,MPI_CHAR,nodoDest,TAG_DAT,
    MPI_COMM_WORLD);
      }


    if(vecinosSD == 1){
      nodoRemi = rank - orden + 1;
      MPI_Recv(&rec,1, MPI_CHAR, nodoRemi, TAG_DAT,
       MPI_COMM_WORLD, &info);
      palabra[eti_sig] = rec;
eti_sig ++;
    }if(vecinosII == 1){
        nodoDest = rank + orden - 1;
        char *ma = matriz[i][j];
        MPI_Send(&ma,1,MPI_CHAR,nodoDest,TAG_DAT,
    MPI_COMM_WORLD);
      }

    if(vecinosDER == 1){
      nodoRemi = rank + 1;
      MPI_Recv(&rec,1, MPI_CHAR, nodoRemi, TAG_DAT,
       MPI_COMM_WORLD, &info);
      palabra[eti_sig] = rec;
eti_sig ++;
    }if(vecinosIZQ == 1){
        nodoDest = rank - 1;
        char *ma = matriz[i][j];
        MPI_Send(&ma,1,MPI_CHAR,nodoDest,TAG_DAT,
    MPI_COMM_WORLD);
      }

    if(vecinosID == 1){
      nodoRemi = rank + orden + 1;
      MPI_Recv(&rec,1, MPI_CHAR, nodoRemi, TAG_DAT,
       MPI_COMM_WORLD, &info);
      palabra[eti_sig] = rec;
eti_sig ++;
    }if(vecinosSI == 1){
        nodoDest = rank - orden - 1;
        char *ma = matriz[i][j];
        MPI_Send(&ma,1,MPI_CHAR,nodoDest,TAG_DAT,
    MPI_COMM_WORLD);
      }


    if(vecinosAB == 1){
      nodoRemi = rank + orden;
      MPI_Recv(&rec,1, MPI_CHAR, nodoRemi, TAG_DAT,
       MPI_COMM_WORLD, &info);
      palabra[eti_sig] = rec;
eti_sig ++;
    }if(vecinosAR == 1){
        nodoDest = rank - orden;
        char *ma = matriz[i][j];
        MPI_Send(&ma,1,MPI_CHAR,nodoDest,TAG_DAT,
    MPI_COMM_WORLD);
      }


    if(vecinosII == 1){
      nodoRemi = rank + orden - 1;
      MPI_Recv(&rec,1, MPI_CHAR, nodoRemi, TAG_DAT,
       MPI_COMM_WORLD, &info);
      palabra[eti_sig] = rec;
eti_sig ++;
    }if(vecinosSD == 1){
        nodoDest = rank - orden + 1;
        char *ma = matriz[i][j];
        MPI_Send(&ma,1,MPI_CHAR,nodoDest,TAG_DAT,
    MPI_COMM_WORLD);
      }

    if(vecinosIZQ == 1){
      nodoRemi = rank - 1;
      MPI_Recv(&rec,1, MPI_CHAR, nodoRemi, TAG_DAT,
       MPI_COMM_WORLD, &info);
      palabra[eti_sig] = rec;
      eti_sig ++;
    }if(vecinosDER == 1){
        nodoDest = rank + 1;
        char *ma = matriz[i][j];
        MPI_Send(&ma,1,MPI_CHAR,nodoDest,TAG_DAT,
    MPI_COMM_WORLD);
      }

    palabra[eti_sig] = '*';
    printf("Soy el proceso con id: %d mi palabra es
    : %s\n",rank, &palabra);
    MPI_Gather(palabra, 10, MPI_CHAR, rbuf, 10,
```

```
      MPI_CHAR, MAESTRO, MPI_COMM_WORLD);
132   MPI_Barrier(MPI_COMM_WORLD);
133   MPI_Barrier(MPI_COMM_WORLD);
134
135   if (nodo_id == MAESTRO) {
136     FILE *fp = fopen("resultado_malla.txt","w");
137     if(fp == NULL){
138       printf("error al crear archivo\n");}
139     int apunt = 0;
140     fprintf(fp, "La matriz inicial: \n\n");
141     for (size_t i = 0; i < orden; i++) {
142       for (size_t j = 0; j < orden; j++) {
143         fprintf(fp, "%c ", matriz[i][j]);
144       }
145       fprintf(fp, "\n");
146     }
147     fprintf(fp, "\n");
148     for (size_t i = 0; i < size; i++) {
149       apunt = 10*i;
150       while (rbuf[apunt] != '*') {
151         fprintf(fp, "%c", rbuf[apunt]);
152         apunt++;
153       }
154       fprintf(fp, "\n");
155     }
```

VI. COMPARACIÓN Y MÉTRICAS

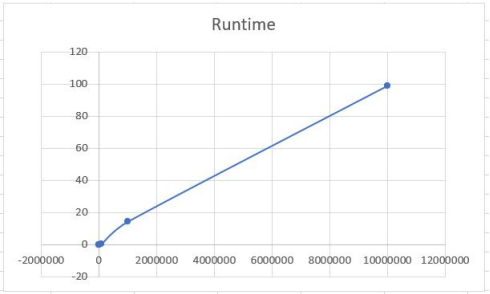
A. *Estrella, Pool y Pipeline*

RUNTIME SECUENCIAL

TABLE I

RUNTIME SECUENCIAL

#_registros	Tiempo_ejecución
100	0.025331
1000	0.028843
10000	0.035952
100000	0.124
1000000	1.514
10000000	13.33



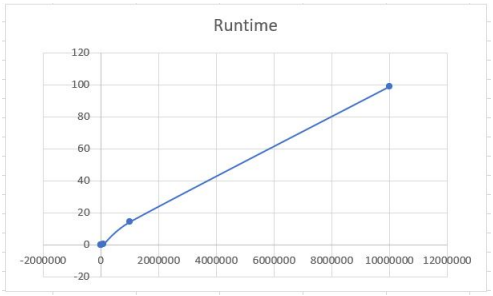
ESTRELLA

Runtime

TABLE II

MÉTRICA DE RENDIMIENTO PARA *Runtime*.

Número de dato	Runtime
100	0.022
1000	0.0156
10000	0.054
100000	0.3977
1000000	14.202
10000000	9850

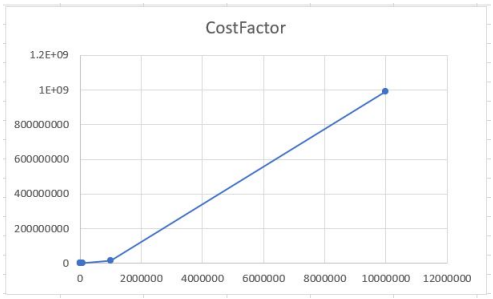


CostFactor

TABLE III

MÉTRICA DE RENDIMIENTO PARA *Costfactor*.

Número de dato	Costfactor
100	2.2
1000	15.6
10000	540
100000	39770
1000000	14202000
10000000	989058000

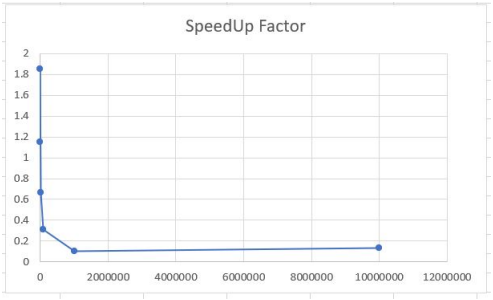


SpeedUp

TABLE IV

METRICA DE RENDIMIENTO PARA *SpeedUp*.

100	1.5114
1000	1.8489
10000	0.6657
100000	0.3117
1000000	0.1066
10000000	0.1347



Efficiency

TABLE V

METRICA DE RENDIMIENTO PARA *Efficiency*.

100	1.15E-02
1000	1.85E-03
10000	6.66E-05
100000	3.12E-06
1000000	1.07E-07
10000000	1.35E-08

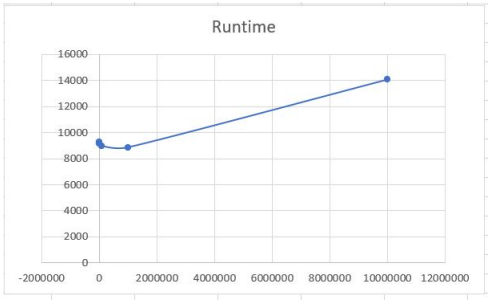
POOL

Runtime

TABLE VI

MÉTRICA DE RENDIMIENTO PARA *Runtime*.

Número de dato	Runtime
100	9242.7332
1000	9201.138
10000	9124.6541
100000	8967.2314
1000000	8866.6257
10000000	14054.32

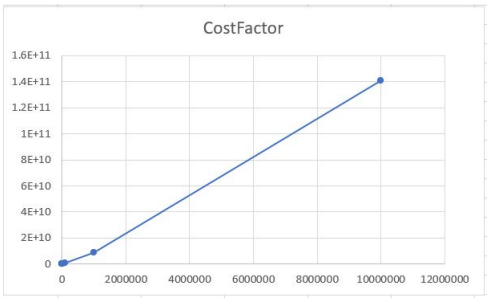


CostFactor

TABLE VII

MÉTRICA DE RENDIMIENTO PARA *Costfactor*.

Número de dato	Costfactor
100	924273.32
1000	9201138
10000	91246541
100000	896723140
1000000	8866625700
10000000	8866625700

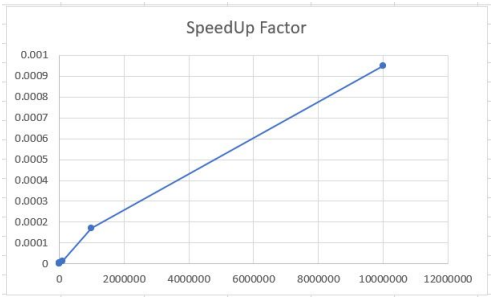


SpeedUp

TABLE VIII

METRICA DE RENDIMIENTO PARA *SpeedUp*.

100	2.74064E-06
1000	3.13472E-06
10000	3.94009E-06
100000	1.38281E-05
1000000	0.000170753
10000000	0.000948463



Efficiency

TABLE IX

METRICA DE RENDIMIENTO PARA *Efficiency*.

100	2.74E-08
1000	3.13E-09
10000	3.94E-10
100000	1.38E-10
1000000	1.71E-10
10000000	9.48E-11

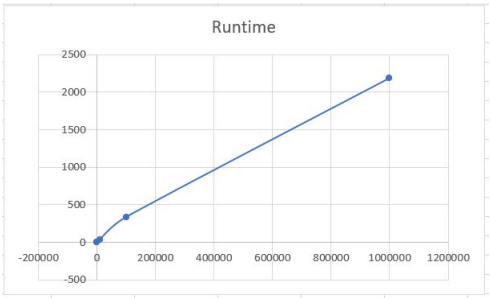
PIPELINE (TIPO4)

Runtime

TABLE X

MÉTRICA DE RENDIMIENTO PARA *Runtime*.

Número de dato	Runtime
100	0.091303
1000	3.070922
10000	35.640182
100000	337.703695
1000000	2189.582379

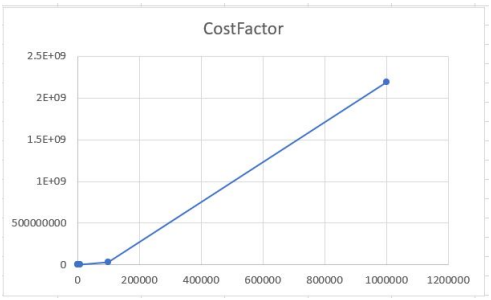


CostFactor

TABLE XI

MÉTRICA DE RENDIMIENTO PARA *Costfactor*.

Número de dato	Costfactor
100	9.13
1000	3070.9
10000	356402
100000	33770370
1000000	2189582000

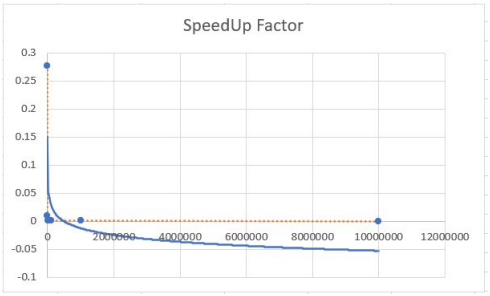


SpeedUp

TABLE XII

METRICA DE RENDIMIENTO PARA *SpeedUp*.

100	0.277447974
1000	0.009392361
10000	0.001008749
100000	0.000367186
1000000	0.000691456



Efficiency

TABLE XIII

METRICA DE RENDIMIENTO PARA *Efficiency*.

100	2.77E-03
1000	9.39E-06
10000	1.01E-07
100000	3.67E-09
1000000	6.91E-10

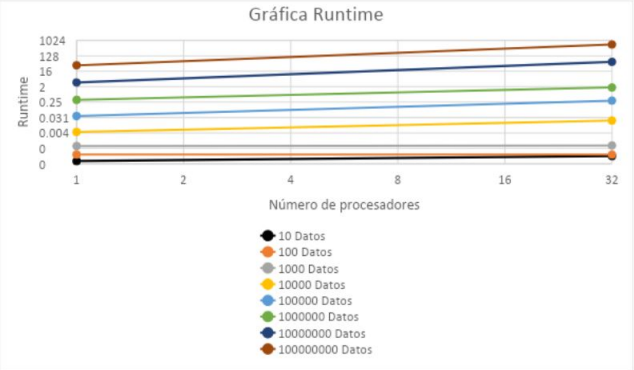
B. Árbol

Runtime

TABLE XIV

MÉTRICA DE RENDIMIENTO PARA *Runtime*.

Número de dato	Tiempo en secuencial (s)	Tiempo en paralelo (s)
10	0.000089	0.0001766
100	0.000213	0.0002133
1000	0.000671	0.000729
10000	0.004460	0.02081
100000	0.037965	0.305540
100000	0.338758	1.7940
1000000	3.48189	56.5940
10000000	35.161672	586.2640

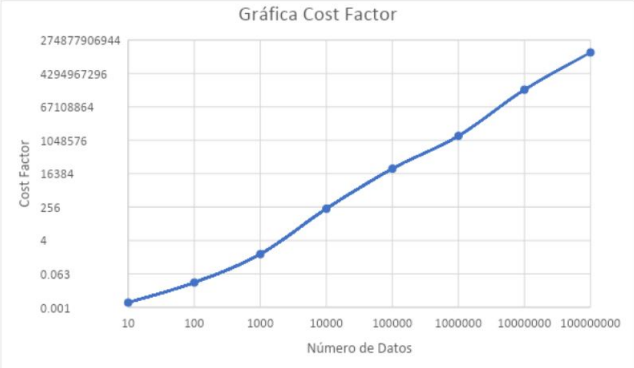


En la gráfica se aprecia la ejecución del programa para conjuntos de datos de distintos tamaños, las pruebas se realizaron con el mismo archivo de datos variando únicamente la cantidad de datos que lee del archivo, es notorio que el programa en secuencial funciona mejor que el paralelo en todos los casos. CostFactor

TABLE XV

MÉTRICA DE RENDIMIENTO PARA *Costfactor*.

Número de dato	Costfactor
10	0.001766
100	0.02133
1000	0.7299
10000	208.1341
100000	30554.0398
100000	1794085.29
1000000	565940943
10000000	58626000000

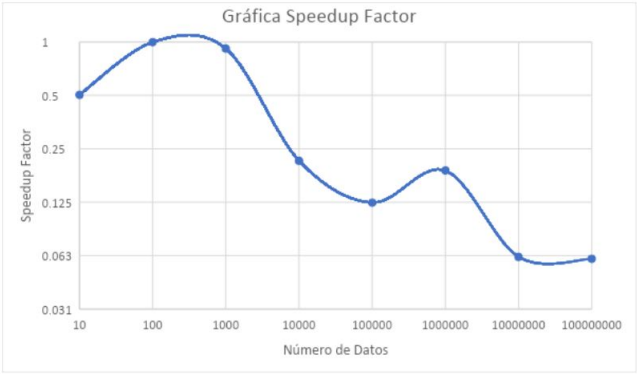


La gráfica nos indica la cantidad de trabajo que realiza el programa, como podemos apreciar si aumentamos el número de datos, aumenta el trabajo, es decir, al programa le cuesta más trabajo procesar grandes cantidades de datos. SpeedUp

TABLE XVI

MÉTRICA DE RENDIMIENTO PARA *SpeedUp*.

Número de dato	SpeedUp
10	0.50388
100	0.9982
1000	0.9191
10000	0.2142
100000	0.1242
100000	0.1888
1000000	0.0615
10000000	0.0599

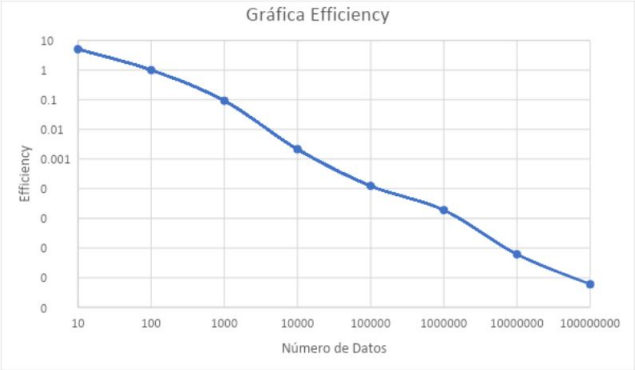


Esta grafica nos indica el rendimiento del programa en paralelo, comparando el tiempo que se llevó el programa secuencial con el tiempo del programa en paralelo, cabe mencionar que las mediciones se hicieron con las mismas cantidades de datos, tanto para el secuencial y paralelo, como podemos ver en la gráfica el rendimiento de nuestro programa en paralelo disminuye al aumentar los datos a procesar. Efficiency

TABLE XVII

MÉTRICA DE RENDIMIENTO PARA *Efficiency*.

Número de dato	Efficiency
10	5.0388
100	0.9982
1000	0.0919
10000	0.00214
100000	0.000124
100000	0.0000188
1000000	0.000000615
10000000	0.000000059



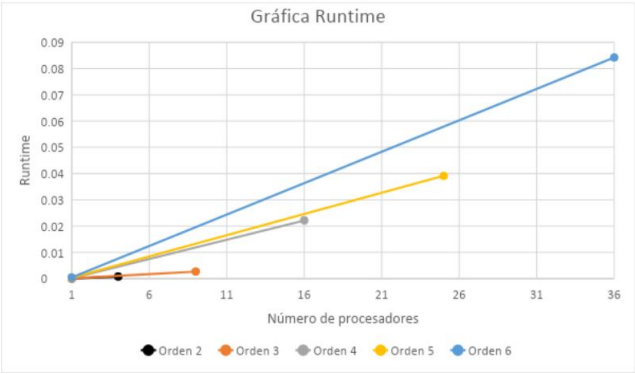
Esta grafica podemos apreciar que al aumentar el número de datos a procesar la eficiencia del programa disminuye, en comparación del programa secuencial, es decir, al programa en paralelo le cuesta más tiempo en realizar el procesamiento que al programa secuencial.

C. Malla

Runtime

TABLE XVIII
MÉTRICA DE RENDIMIENTO PARA *Runtime*.

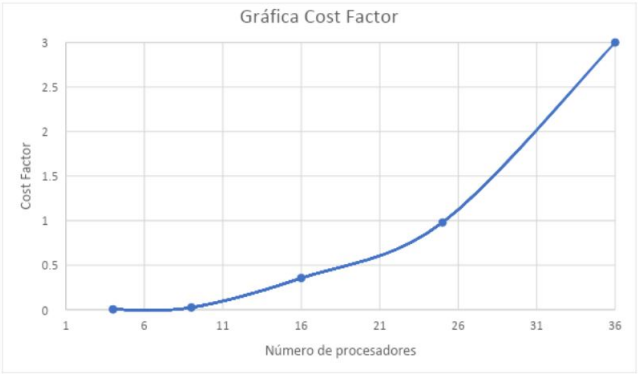
Número$_{d}ato$	Tiempo$_{s}ecuencial(s)$	Tiempo$_{p}aralelo(s)$
2,4 proc	0.000073	0.0008
3,9 proc	0.000086	0.0026
4, 16 proc	0.000175	0.0221
5, 25 proc	0.000379	0.03913
6, 36 proc	0.000437	0.0842



La gráfica indica un crecimiento a medida que varía el número de procesadores y la cantidad de datos a procesar, el tamaño de los datos utilizados para la versión secuencial y paralela crece de manera cuadrática por lo que es normal apreciar esa ligera curva en los puntos de la gráfica, por la forma que tiene se deduce que el número de procesadores óptimos es un valor pequeño, de cuatro o nueve procesadores. CostFactor

TABLE XIX
MÉTRICA DE RENDIMIENTO PARA *CostFactor*.

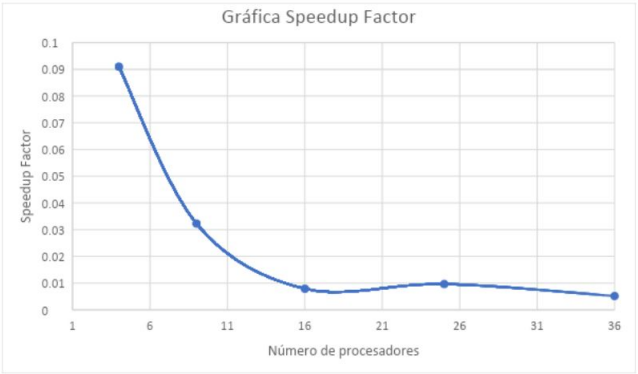
Número de dato	CostFactor
2,4 procesadores	0.003208
3,9 procesadores	0.0239
4, 16 procesadores	0.3546
5, 25 procesadores	0.9783
6, 36 procesadores	3.031



El algoritmo se vuelve sumamente costoso a medida que varía el número de procesadores (y por ende, de datos), los costos más bajos se obtienen para pocos procesadores, tal como indicaba la gráfica de runtime anterior. SpeedUp

TABLE XX
MÉTRICA DE RENDIMIENTO PARA *SpeedUp*.

Número de dato	SpeedUp
2,4 procesadores	0.0910
3,9 procesadores	0.0322
4, 16 procesadores	0.0078
5, 25 procesadores	0.0096
6, 36 procesadores	0.0051

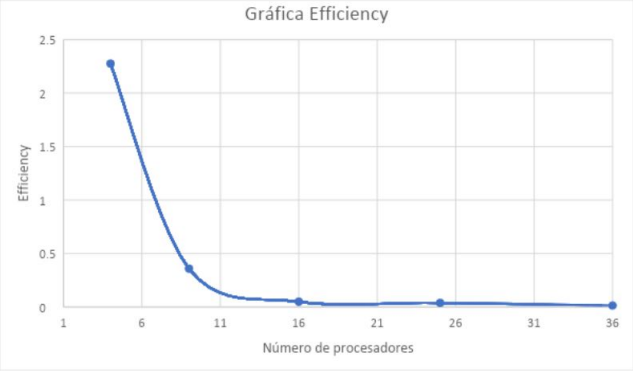


La gráfica nos demuestra que el algoritmo de forma secuencial es mucho más eficiente inmediatamente aumenta la cantidad de procesadores, en el caso de 25 podemos ver un ligero incremento, pero al hacer distintas pruebas podemos notar que el tiempo es ligeramente variable, por lo que el valor de SpeedUp Factor en ese punto no es muy relevante. Efficiency

TABLE XXI

MÉTRICA DE RENDIMIENTO PARA *Efficiency*.

Número de dato	Efficiency
2,4 procesadores	2.2754
3,9 procesadores	0.3587
4, 16 procesadores	0.0493
5, 25 procesadores	0.0387
6, 36 procesadores	0.0144



La mayor eficiencia del programa se obtiene cuando existe un número bajo de procesos y no aumenta significativamente cuando estos incrementan. Esto puede ser porque con menos procesos, el número de datos o el tamaño de los datos es menor, mientras que para más procesos el procesamiento de datos aumenta, también se deben establecer más comunicaciones porque son más nodos, por lo que esto es un factor para que el programa no sea eficiente.

VII. CONCLUSIONES

Los sistemas distribuidos presentan grandes ventajas sobre los métodos de cómputo tradicionales ya que permiten reducir el tiempo de procesamiento considerablemente, haciendo posible encontrar soluciones que serían imposibles de obtener con algoritmos secuenciales, creando sistemas con una alta capacidad de procesamiento que utilicen equipo de cómputo de uso común, por otra parte, resulta evidente que no en todas la situaciones obtendremos mejores resultados al crear algoritmos que se ejecuten de manera concurrente pues existen factores que pueden deteriorar el tiempo de respuesta, dichos factores pueden deberse a la manera en la que comunicamos los procesos, en donde puedan existir grandes intervalos en donde la capacidad de cómputo se desperdicie por esperar la respuesta de otro proceso ya que en muchos de los casos resulta difícil minimizar la dependencia de resultados entre cada uno de ellos. A través de las métricas de rendimiento pudimos apreciar que el tiempo o la eficiencia de los programas paralelos no es muy buena en comparación con la versión secuencial, aun teniendo el problema de las grandes cantidades de datos a procesar. Las versiones en secuencial tardan menos en ejecutar que las versiones en paralelo, incluso la pudimos observar en el momento de ejecutar el programa en paralelo las maquinas se pasmaban y se tardaban en responder cuando las cantidades de datos eran grandes mientras que en el secuencial solo tardaba en ejecutar el programa pero podíamos realizar otras actividades mientras se ejecutaba el programa, muchos de estos problemas pueden deberse a que las unidades de procesamiento son limitadas en nuestro equipo y todo el trabajo es realizado por distintos procesos atendidos por una sola CPU, habrá que probar con problemas de mayor complejidad y en sistemas que físicamente se distribuyan a través de distintas máquinas que nos permitan evaluar mejor la eficiencia.

REFERENCES

[1] Ayala,J.A.(22 abril 2020) Práctica 8:Autómatas Celulares. Sistemas Distribuidos, 1.
[2] Ayala,J.A.(29 marzo 2020) Práctica 87:TOEFL. Sistemas Distribuidos, 1.
[3] Ayala,J.A. Apuntes de la materia de Sistemas Distribuidos, semestre 2020-2.