Método Sarrus Paralelo Extendido a Matrices de Tamaño Mayor a 3

Blancarte Lopez Jorge, Lievana Poy Erick and Ocampo Alvarez Jose Alvaro
Facultad de Ciencias de la Computación, Benemérita Universidad Autónoma de Puebla
Email:jorge.blancarte@alumno.buap.mx, erick.lievanap@alumno.buap.mx, jose.ocampo@alumno.buap.mx

***Abstract*—En este trabajo se analiza la eficiencia de la programación paralela con el uso de las librerias de OpenMPI para el cálculo de la regla extendida de Sarrus para matrices mayores a 3x3. Se compararan los tiempos del algoritmo secuencial y el algoritmo paralelo.**

***Index Terms*—Algoritmo, Paralelo, Secuencial, Tiempo, MPI, Procesos.**

I. Introducción

Para el presente se tiene por objetivo aplicar y crear un escenario comparativo de la ejecución de la regla de Sarrus extendida de forma secuencial contra su forma paralelizada para eso se implementaron en el lenguaje de programación C con el uso de las librerias de OpenMPI, la regla de Sarrus extendida.

II. Antecedentes

A. Regla de Sarrus Extendida

La regla de Sarrus es un método fácil para memorizar y calcular un determinante 3×3. Recibe su nombre del matemático francés Pierre Frédéric Sarrus, que la introdujo en el artículo *Nouvelles méthodes pour la résolution des équations*, publicado en Estrasburgo en 1833.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{1}$$

Dada la anterior matriz de $3x3$ se puede calcular su determinante de la siguiente forma:

$$\begin{aligned} det = & a_{11}a_{22}a_{33} + a_{21}a_{32}a_{13} + a_{31}a_{12}a_{23} \\ & - a_{13}a_{22}a_{31} - a_{23}a_{32}a_{11} - a_{33}a_{12}a_{21} \end{aligned} \tag{2}$$

Se puede llegar al mismo resultado al repetir las dos primeras filas de la matriz debajo de la misma de manera que queden cinco filas. Después sumar los productos de las diagonales descendentes y sustraer los productos de las diagonales ascendentes.

Para poder expander la regla de Sarrus a matrices de mayor tamaño, debemos primero comprender como es que se generan las diagonales. Empecemos con una matriz general de nxn.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \tag{3}$$

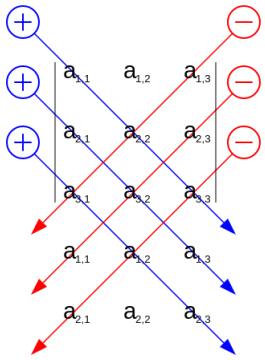

Fig. 1: La regla de Sarrús: las diagonales azules se suman y las diagonales rojas se restan.

A partir de esta matriz podemos deducir que las diagonales descendientes seran las mostradas en la ecuación 4, con esto podemos crear un algoritmo para poder obtener las diagonales descendientes. Como podemos observar el indice de las columnas siempre va desde 0 hasta n, mientras que el indice de las filas va incrementando por 1 en cada diagonal y cuando llega al limite de las filas, se reinicia a 0.

$$\begin{aligned} & a_{00}a_{11}a_{22}\cdots a_{nn} \\ & a_{10}a_{21}a_{32}\cdots a_{n(n-1)}a_{0n} \\ & \vdots \\ & a_{n0}a_{01}a_{12}\cdots a_{(n-2)(n-1)}a_{(n-1)n} \end{aligned} \tag{4}$$

Result: Calcular las diagonales descendentes
for *i = 0; i<n; i++* **do**
  diagonalActual = 1;
  for *j = 0; j<n; j++* **do**
    if *i + j>= n* **then**
      diagonalActual *= $a_{(i+j)j}$;
    else
      diagonalActual *= $a_{((i+j)-n)j}$;
    end
  end
  totalDiagonalesAscendentes = diagonalActual;
end

Algorithm 1: Algoritmo para calcular las diagonales descendientes

Por lo que basandonos en la manera general para recorrer matrices, podemos crear el algoritmo 1, seran 2 ciclos *for* que recorreran la matriz. La primera diagonal siempre la diagonal donde ambos indices son iguales es decir a_{kk} por lo que cuando $i = 0$ como $i + j$ no puede ser mayor que n entonces $a_{(i+j)j} = a_{jj}$. Para casos donde $i > 0$ podemos observar que habra un punto donde $i + j >= n$ y en esta situacion se usara $a_{((i+j)-n)j}$ y para el caso donde $i + j = n$ se reiniciara el indice de las filas a 0 lo que hace posible calcular las diagonales descendientes sin repetir renglones de la matriz.

Para el caso de las diagonales ascendentes, tenemos que las diagonales generadas tendran las forma presentada en la ecuación 5. Podemos observar ahora el indice de las columnas va de n a 0 mientras que el de las filas se mantiene con la misma logica que el de las diagonales ascendentes, por que lo que es posible modificar el algoritmo 1 en el algoritmo 2 para tambien calcular las diagonales descendientes para tambien calcular las diagonales ascendentes.

$$\begin{aligned} &a_{0n}a_{1(n-1)}a_{2(n-2)}\cdots a_{n0} \\ &a_{1n}a_{2(n-1)}a_{3(n-2)}\cdots a_{(n-1)1}a_{n0} \\ &\vdots \\ &a_{nn}a_{0(n-1)}a_{1(n-2)}\cdots a_{(n-2)1}a_{(n-1)0} \end{aligned} \tag{5}$$

Result: Calcular todas las diagonales
for *i = 0; i<n; i++* **do**
  diagonalActualPositiva = 1;
  diagonalActualNegativa = 1;
  for *j = 0; j<n; j++* **do**
    if $i + j >= n$ **then**
      diagonalActualPositiva *= $a_{(i+j)j}$;
      diagonalActualNegativa *= $a_{(i+j)(n-j)}$;
    else
      diagonalActualPositiva *= $a_{((i+j)-n)j}$;
      diagonalActualNegativa *= $a_{((i+j)-n)(n-j)}$;
    end
  end
  totalPositivas += diagonalActualPositiva;
  totalNegativas += diagonalActualNegativa;
end

Algorithm 2: Algoritmo para calcular todas las diagonales

B. MPI

MPI (iniciales de Message Passing Interface) es una especificación para programación de paso de mensajes, que proporciona una librería de funciones para C, C++ y Fortran que son empleadas en los programas para comunicar datos y portable, especificada por consenso por el MPI Forum, con unas 40 organizaciones participantes, como modelo que permita desarrollar programas que puedan ser migrados a diferentes computadores paralelos.

Definido conjuntamente por proveedores de hardware y de software, OpenMP es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas,

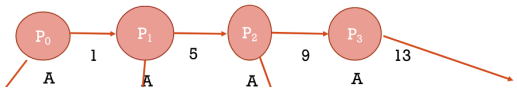

Fig. 2: Comunicación lineal entre 4 procesos

para plataformas que van desde las computadoras de escritorio hasta supercomputadoras.

Para este problema desarrollamos un programa en el lenguaje de programación C usando openMPI con un tipo de comunicacion de punto a punto lineal esto significa que los procesos envian la información procesada al siguiente proceso en el rank asi como en la Figura 2. Al final es el ultimo proceso en el rank el que junta toda la información y la muestra o guarda.

III. Desarrollo

Para desarrollar este experimento es necesario implementar el algoritmo 2 en lenguaje C de forma que cada proceso pueda calcular cierta cantidad de diagonales y aportarlas a la suma de las diagonales descendentes y las diagonales ascendentes respectivamente. Asi mismo para representar la comunicación de punto a punto lineal es necesario que los procesos se comuniquen entre si y envien los resultados de sus ejecuciones. Al final del reporte se anexara la totalidad del codigo fuente.

A. Implementación de Sarrus Paralelo

Como mencionamos debemos permitir que cada proceso calcule solo un cierto rango de diagonales, como cada diagonal inicia con el primer elemento de una fila, entonces hay tantas diagonales como filas tiene las matriz, por lo que al dividir n entre el numero total de procesos obtendremos el tamaño de los rangos, y al multiplicarlos por el rank de cada proceso podemos obtener el inicio de cada rango. Para esto se implemento el codigo del listing 1 en C.

Listing 1: Implementacion de Sarrus Paralelo

```
void sarrusPorRank(int rank,long double
    DiagArray[2],int matriz[orden][orden]){
  int from = (rank * orden)/numeroDeRanks;
  int to = ((rank+1) * orden)/numeroDeRanks;
  long double temporalNegativa;
  long double temporalPositiva;
  int i,j;
  int fila;
  DiagArray[0] = 0;
  DiagArray[1] = 0;

  printf("computing rank \%d (from row \%d to
      \%d)\n", rank, from, to-1);
  for(i = from; i < to; i++){
    temporalNegativa = 1;
    temporalPositiva = 1;
    for(j = 0; j < orden; j++){
      fila = i+j;
      if(fila >= orden){
        fila -= orden;
      }
      temporalPositiva *= matriz[fila][j];
      temporalNegativa *=
          matriz[fila][(orden-1)-j];
```

```
    }
    DiagArray[0] += temporalNegativa;
    DiagArray[1] += temporalPositiva;
  }
  printf("finished rank \%d\n", rank);
}
```

Como podemos ver en el Listing 1 se muestra la función que hara el cálculo de las Diagonales de Sarrus, esta función recibe la matriz de la cual se desean calcular las diagonales, el numero de rank del proceso que la llama y finalmente recibe un vector de tamaño 2 donde se guardaran los resultados.

La primera parte de la función utiliza el rank para calcular el rango de diagonales que los procesos van a calcular, esto para dividir la matriz de forma uniforme entre los procesos.

La siguiente parte hace la declaración de las variables de apoyo que se usaran para el calculo de las diagonales. Puesto que se calculan las diagonales descendentes y las diagonales ascendentes al mismo tiempo entonces debemos crear variables para ambos casos.

En la siguiente parte anunciamos que proceso esta haciendo el calculo e implementamos el algoritmo 2, para paralelizarlo solo vamos a recorrer el rango de diagonales calculado en la primera parte. Para ello nuestra variable i solo debera recorrer desde el valor de la variable $from$ hasta el valor de la variable to. Y se guardan los resultados en el vector de resulstados $DiagArray$ siendo el indice 0 los resultados de las diagonales ascendentes y el indice 1 los resultados de las diagonales descendentes. Finalmente el proceso reporta su finalización.

B. Implentación Comunicación Punto a Punto

Como se ve en la Figura 2 los procesos desde el mas bajo hasta el penultimo, envian sus resultados al proceso que les sigue, es decir el proceso k envia su información al proceso $k+1$. Para esto MPI ofrece las funciones *MPI_Recv* y *MPI_Send* que permiten enviar información entre los procesos. Con estas funciones podemos implementar el codigo dle listing 2 en C.

Listing 2: Implementacion de Comunicación Lineal

```
if(rank == 0){
  sarrusPorRank(rank,tmp,A);
  Diagonales[0] = tmp[0];
  Diagonales[1] = tmp[1];
  if(numeroDeRanks > 1)
  MPI_Send(Diagonales, 2, MPI_LONG_DOUBLE,
      rank+1, tag, MPI_COMM_WORLD);
  MPI_Send(A, orden*orden, MPI_INT, rank+1,
      tag, MPI_COMM_WORLD);
}
if(rank == numeroDeRanks - 1){
  if(numeroDeRanks > 1){
    MPI_Recv(Diagonales, 2, MPI_LONG_DOUBLE,
        rank-1, tag, MPI_COMM_WORLD, &status);
    MPI_Recv(A, orden*orden, MPI_INT, rank-1,
        tag, MPI_COMM_WORLD, &status);
    sarrusPorRank(rank,tmp,A);
    Diagonales[0] += tmp[0];
    Diagonales[1] += tmp[1];
  }
  end = MPI_Wtime();
  if(orden < 21) imprimirMatriz(A);
  printf("Sarrus calculation done.\n\n");
  printf("Pos final: \%Lf rank:
      \%d\n",Diagonales[1],rank);
  printf("Neg final: \%Lf rank:
      \%d\n",Diagonales[0],rank);
  printf("Determinante = \%Lf rank:
      \%d\n",Diagonales[1]-Diagonales[0],rank);
  printf("Pos final: \%Le rank:
      \%d\n",Diagonales[1],rank);
  printf("Neg final: \%Le rank:
      \%d\n",Diagonales[0],rank);
  printf("Determinante = \%Le rank:
      \%d\n",Diagonales[1]-Diagonales[0],rank);
  printf("Tiempo: \%lf rank:
      \%d\n",end-start,rank);
  printf("\n");

  FILE *file1;
  file1 = fopen("datosTiempo.txt","a+");
  if(file1 != NULL){
    fprintf(file1,"\%lf\n",end - start);
  }
  fflush(file1);
  fclose(file1);
}
if(rank != 0 && rank != numeroDeRanks - 1){
  MPI_Recv(Diagonales, 2, MPI_LONG_DOUBLE,
      rank-1, tag, MPI_COMM_WORLD, &status);
  MPI_Recv(A, orden*orden, MPI_INT, rank-1,
      tag, MPI_COMM_WORLD, &status);
  sarrusPorRank(rank,tmp,A);
  Diagonales[0] += tmp[0];
  Diagonales[1] += tmp[1];
  MPI_Send(Diagonales, 2, MPI_LONG_DOUBLE,
      rank+1, tag, MPI_COMM_WORLD);
  MPI_Send(A, orden*orden, MPI_INT, rank+1,
      tag, MPI_COMM_WORLD);
}
```

En el listing 2 que pertenece a la función main del programa, lo primero que se puede notar es que se checa que si el rank del proceso es 0, esto es importante ya que cada programa MPI tendra por lo menos un proceso cuyo rank sera 0. En este caso el proceso con rank 0, llamara a la función vista en el listing 1 le pasara su rank que es cero, un array *tmp* que es donde guardaran los resultados temporalmente y finalmente la matriz A. Una vez ejecutada la función los resultados guardados en el array *tmp* seran guardados en el array *Diagonales* que almanaceran los resultados definitivos. Despues si hay mas de 1 proceso, entonces el proceso con rank 0 enviara el array *Diagonales* y la matriz al siguiente proceso. En caso de ser solo el proceso con rank 0 no se enviara ninguna información.

Despues se checara si el proceso es el ultimo proceso, en caso de ser el último proceso se checara si hay mas de un proceso, esto para el caso donde solo haya un solo proceso, en este caso el procesos con rank 0 es el primer y el ultimo proceso, en caso de que sea asi y solo hay un proceso no se recibira nada de otro proceso y tampoco hay que volver a llamar la función de listing 1 y por lo tanto no hay cambio a los resultados por lo que se puede pasar directamente al manejor de resultados. En caso de que si haya mas de un proceso, entonces se recibe la información otorgada por el penultimo proceso y el ultimo procesos hace sus ejecuciones correspondientes al llamar la funcion del listing 1 pasando su rank, el array *tmp* y la matriz A, despues calcula los resultados

```
z270h> darkskoll> proyecto>_mpirun -np 1 sarrus 20
computing rank 0 (from row 0 to 19)
finished rank 0

        | 4| 7|18|16|14|16| 7|13|10| 2| 3| 8|11|20| 4| 7| 1| 7|13|17|
        |12| 9| 8|10| 3|11| 3| 4| 8|16|10| 3| 3|19|10| 8|14|17|12| 3|
        |10|14| 2|20| 5|18|19| 5|16|11|14| 7|12| 1|17|14| 3|11|17| 2|
        | 6| 6| 5| 8|17| 6| 7|10|14|18| 5|16| 3| 6|15| 8|15| 5| 4|11|
        | 8| 9|17|19| 9| 5| 4|12|15|20|13| 1|17| 9|20|13| 7| 7|15|20|
        |16|11|15|19| 8| 2|18| 3|18|13|13|17| 2| 1| 7| 2| 6|10| 5|20|
        | 1|10|12|18|18|12| 2|16|10| 8| 8|17|18|14| 7| 6| 7| 4|20| 5|
        | 9|12|13|10| 4|20|11| 9| 9|16| 1|10|17| 4|19| 6| 7|12| 2|16|
        |20| 9| 5| 9| 2|11|14| 1|15| 5| 5|15| 8|17| 4|12| 8| 6|20|17|
        |13|12|18| 9|16| 8|15| 2|19|16|10|18|16|14|19| 9| 4|12| 9|10|
        |17| 5| 4| 4|14|19| 7| 1| 5|19| 9| 9|10|18|18|17| 5| 4|11| 4|
        |11|20|13| 6| 5|11| 6|20|15| 7|10| 3| 3| 5|18| 8|16| 5| 9|12|
        | 3| 9|20| 4| 7| 9| 1| 3|12|11| 6| 2| 2|11|19| 6| 1|17| 5| 7|
        | 3| 6| 9|17| 3|19| 5|18| 3| 5| 1|17|13|20|20|11| 9|12|14|12|
        | 2|11|14|15| 1| 4|20| 2|20|17|20|14|14| 9|11|16| 7| 7| 5| 1|
        |11| 5|17| 3|17| 8| 6|17|20|19| 8|13| 9|13|20|10|17|11| 3| 8|
        | 7| 2|14|13| 2|16|20|20| 2| 5|20|12| 1| 8| 6| 9| 8|11| 5|19|
        | 1| 5| 3|10|17| 2|11| 5|13|13|13|11| 6| 6| 3|20| 1| 3|19|14|
        |19|11| 5|11|10| 2|20|17|13|16|15| 5|20|10|14|17|11|16| 1| 3|
        |20| 5|14| 6| 2| 8| 5|14| 2|15| 7|20| 5| 3| 3| 7| 5| 2| 3| 9|
Sarrus calculation done.

Pos final: 15972969760153208985 60.000000 rank: 0
Neg final: 765005651064788651008.000000 rank: 0
Determinante = 832291324950532247552.000000 rank: 0
Pos final: 1.597297e+21 rank: 0
Neg final: 7.650057e+20 rank: 0
Determinante = 8.322913e+20 rank: 0
Tiempo: 0.000024 rank: 0
```

Fig. 3: Ejecución con 1 proceso

finales al sumarle al array *Diagonales* los resultados guardados temporalmente en el array *tmp*.

Para mostrar los resultados el ultimo proceso, los mostrara en pantalla con la funcion *printf* de manera normal asi como en notación cientifica esto debido a que se generan números muy grandes hasta del orden 10^{1400} aproximadamente.

De igual forma el ultimo proceso es responsable de guardar los distintos tiempos de ejecución a un archivo de texto llamado *datosTiempo.txt*

Finalmente aquellos procesos que no tengan rank 0 o sean el ultimo rank, primero recibiran la matriz y los resultados del procesos con rank anterior, ejecutaran la función vista en el listing 1 guardaran los resultados en el array *Diagonales* y pasaran la matriz y el array *Diagonales* al siguiente proceso.

IV. Resultados

Es importante resaltar que los resultados obtenidos como vimos se componen de números grandes del orden de 10^{1400} esto debido a los tamaños de las matrices. Sin embargo en este trabajo no nos concentraremos en los resultados si no en los tiempos de ejecución. Para ello habran 2 categorias una donde se comparan los tiempo de ejecución con distintos números de procesos, y en la otra categoria el tiempo de ejecución en base al tamaño de la matriz.

A. Ejecución con distintos procesos

Como primer punto es importante comprobar que independientemente del numero de procesos el programa devuelve resultados iguales a entradas iguales. Por lo que ejecutamos el programa con una mátricula de 20x20 con los mismos datos y ejecutamos el programa con 1 y con 4 procesos. Para comprobar que estos sean iguales.

```
z270h> darkskoll> proyecto>_mpirun -np 4 sarrus 20
computing rank 0 (from row 0 to 4)
finished rank 0
computing rank 1 (from row 5 to 9)
finished rank 1
computing rank 2 (from row 10 to 14)
finished rank 2
computing rank 3 (from row 15 to 19)
finished rank 3

        | 4| 7|18|16|14|16| 7|13|10| 2| 3| 8|11|20| 4| 7| 1| 7|13|17|
        |12| 9| 8|10| 3|11| 3| 4| 8|16|10| 3| 3|19|10| 8|14|17|12| 3|
        |10|14| 2|20| 5|18|19| 5|16|11|14| 7|12| 1|17|14| 3|11|17| 2|
        | 6| 6| 5| 8|17| 6| 7|10|14|18| 5|16| 3| 6|15| 8|15| 5| 4|11|
        | 8| 9|17|19| 9| 5| 4|12|15|20|13| 1|17| 9|20|13| 7| 7|15|20|
        |16|11|15|19| 8| 2|18| 3|18|13|13|17| 2| 1| 7| 2| 6|10| 5|20|
        | 1|10|12|18|18|12| 2|16|10| 8| 8|17|18|14| 7| 6| 7| 4|20| 5|
        | 9|12|13|10| 4|20|11| 9| 9|16| 1|10|17| 4|19| 6| 7|12| 2|16|
        |20| 9| 5| 9| 2|11|14| 1|15| 5| 5|15| 8|17| 4|12| 8| 6|20|17|
        |13|12|18| 9|16| 8|15| 2|19|16|10|18|16|14|19| 9| 4|12| 9|10|
        |17| 5| 4| 4|14|19| 7| 1| 5|19| 9| 9|10|18|18|17| 5| 4|11| 4|
        |11|20|13| 6| 5|11| 6|20|15| 7|10| 3| 3| 5|18| 8|16| 5| 9|12|
        | 3| 9|20| 4| 7| 9| 1| 3|12|11| 6| 2| 2|11|19| 6| 1|17| 5| 7|
        | 3| 6| 9|17| 3|19| 5|18| 3| 5| 1|17|13|20|20|11| 9|12|14|12|
        | 2|11|14|15| 1| 4|20| 2|20|17|20|14|14| 9|11|16| 7| 7| 5| 1|
        |11| 5|17| 3|17| 8| 6|17|20|19| 8|13| 9|13|20|10|17|11| 3| 8|
        | 7| 2|14|13| 2|16|20|20| 2| 5|20|12| 1| 8| 6| 9| 8|11| 5|19|
        | 1| 5| 3|10|17| 2|11| 5|13|13|13|11| 6| 6| 3|20| 1| 3|19|14|
        |19|11| 5|11|10| 2|20|17|13|16|15| 5|20|10|14|17|11|16| 1| 3|
        |20| 5|14| 6| 2| 8| 5|14| 2|15| 7|20| 5| 3| 3| 7| 5| 2| 3| 9|
Sarrus calculation done.

Pos final: 15972969760153208985 60.000000 rank: 3
Neg final: 765005651064788651008.000000 rank: 3
Determinante = 832291324950532247552.000000 rank: 3
Pos final: 1.597297e+21 rank: 3
Neg final: 7.650057e+20 rank: 3
Determinante = 8.322913e+20 rank: 3
Tiempo: 0.000205 rank: 3
```

Fig. 4: Ejecución con 4 procesos

TABLE I: Tiempos de ejecución con distintos numeros de procesos

No. de Procesos	Tiempo de Ejecucion(s)
1	0.000196
2	0.000145
3	0.000095
4	0.000090

En las figuras 3 y 4 los resultados para ambas matrices es el mismo $8.322913x10^{20}$ esto nos permite saber que el número de procesos no afecta el resultado obtenido. Por lo que podemos comparar los tiempos de ejecución del programa en base al número de procesos usados. Para esta categoria se haran las pruebas con las siguientes condiciones:

- Número de Procesos: 1,2,3,4
- Tamaño de matriz fijo 100x100

Los resultados obtenidos nos indican que en general el procesamiento de datos es agilizado en base al numero de procesos usados. Como podemos ver en la tabla I y en la gráfica de la figura 5 cuando ejecutamos el programa con 1 proceso es como ejecutarlo de manera secuencial ya que solo un proceso calcula todas las diagonales. En cambio con mas procesos estos pueden dividirse las diagonales y calcularlas de manera parela lo que incrementa la eficiencia del programa y reduce los tiempos. Y como podemos ver en la tabla I tenemos una mejora de hasta 65% en los tiempos de ejecución.

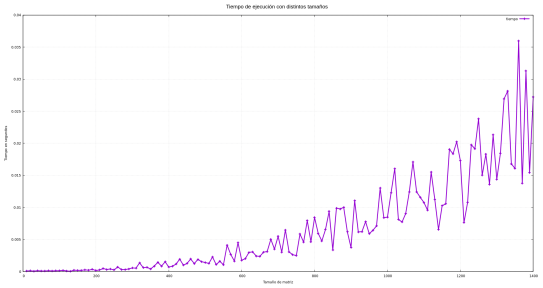

Fig. 6: Tiempo de ejecución con 4 procesos para matrices de tamaños de 10x10 hasta 1400

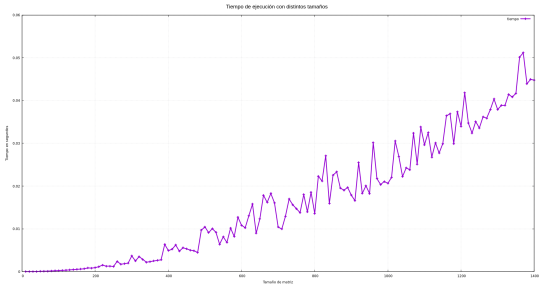

Fig. 7: Tiempo de ejecución con 1 proceso para matrices de tamaños de 10x10 hasta 1400

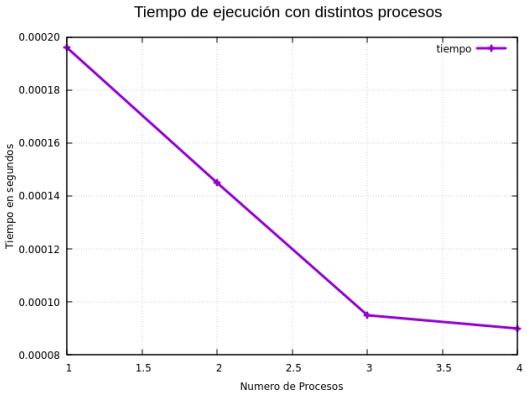

Fig. 5: Gráfica de los tiempo de ejecución respecto al numero de procesadores

B. Ejecución con distinos tamaños de matrices

Como ya vimos el número de procesos no afecta el resultado y mejora los tiempos de ejecución. El siguiente paso es ver como afecta el tamaño de la matriz al tiempo de ejecución tanto para 1 proceso que seria la forma secuencial y ver como 4 procesos mejoran la eficiencia del programa al ser una ejecución paralela. Para este caso se tendran las condiciones:

- Tamaño de la matriz incrementando de 10 en 10, empezando con una matriz de 10x10 y finalizando con una matriz de 1400x1400
- Se ejecutaran estas matrices con 1 proceso y con 4 procesos.

Como podemos observar en las figuras 6 y 7 los tiempos tiempos de ejecución con 4 procesos a pesar de ser mas variables, incluso el tiempo mas largo apenas supera 0.035 segundos, mientras que los tiempos alcanzados con 1 proceso superan los 0.05 segundos. Esta diferencia es considerable ya que es una mejora del 30%.

V. Conclusiones

Como lo muestran los resultados, el uso de algoritmos y programación paralela trae consigo una mejora considerable en los tiempos de ejecución sin afectar los resultados de los programas que la usan. Para este trabajo tuvimos mejoras desde el 30% hasta de $65\$$ lo cual nos muestra que la programación paralela es sumamente eficiente.

VI. Referencias

[1] Fox G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, Vol. I, Prentice Hall, Englewood Cliffs, New Jersey, 1988. [1] Almeida, F. Giménez, D. Mantas, J.M., Vidal A. Introducción a la Programación Paralela, Paraninfo CENGAGE Learning, España, 2010.

VII. Anexo

En este anexo podra encontrar el codigo completo del programa implementado.

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

int numeroDeRanks;
int orden;

void llenarMatriz(int matriz[orden][orden]);
void imprimirMatriz(int matriz[orden][orden]);
void sarrusPorRank(int rank, long double
    DiagArray[2], int matriz[orden][orden]);

int main(int argc, char *argv[]){
  if(argc != 2) return 0;
  sscanf(argv[1], "%d", &orden);

  int A[orden][orden];
  int rank;
  int tag=1;
  long double Diagonales[2];
  long double tmp[2];
  double start,end;
  MPI_Status status;

  /*Inicia MPI e identifica tu id (rank) */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,
      &numeroDeRanks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  start = MPI_Wtime();
  /*Envio y recepcion de mensajes*/
  if(rank == 0){
    llenarMatriz(A);
    sarrusPorRank(rank,tmp,A);
    Diagonales[0] = tmp[0];
    Diagonales[1] = tmp[1];
    if(numeroDeRanks > 1){
      MPI_Send(Diagonales, 2, MPI_LONG_DOUBLE,
          rank+1, tag, MPI_COMM_WORLD);
      MPI_Send(A, orden*orden, MPI_INT, rank+1,
          tag, MPI_COMM_WORLD);
    }
  }
  if(rank == numeroDeRanks - 1){
    if(numeroDeRanks > 1){
      MPI_Recv(Diagonales, 2, MPI_LONG_DOUBLE,
          rank-1, tag, MPI_COMM_WORLD, &status);
      MPI_Recv(A, orden*orden, MPI_INT, rank-1,
          tag, MPI_COMM_WORLD, &status);
      sarrusPorRank(rank,tmp,A);
      Diagonales[0] += tmp[0];
      Diagonales[1] += tmp[1];
    }
    end = MPI_Wtime();
    if(orden < 21) imprimirMatriz(A);
    printf("Sarrus calculation done.\n\n");
    printf("Pos final: %Lf rank:
        %d\n",Diagonales[1],rank);
    printf("Neg final: %Lf rank:
        %d\n",Diagonales[0],rank);
    printf("Determinante = %Lf rank:
        %d\n",Diagonales[1]-Diagonales[0],rank);
    printf("Pos final: %Le rank:
        %d\n",Diagonales[1],rank);
    printf("Neg final: %Le rank:
        %d\n",Diagonales[0],rank);
    printf("Determinante = %Le rank:
        %d\n",Diagonales[1]-Diagonales[0],rank);
    printf("Tiempo: %lf rank:
        %d\n",end-start,rank);
    printf("\n");

    FILE *file1;
    file1 = fopen("datosTiempo.txt","a+");
    if(file1 != NULL){
      fprintf(file1,"%lf\n",end - start);
    }
    fflush(file1);
    fclose(file1);
  }
  if(rank != 0 && rank != numeroDeRanks - 1){
    MPI_Recv(Diagonales, 2, MPI_LONG_DOUBLE,
        rank-1, tag, MPI_COMM_WORLD, &status);
    MPI_Recv(A, orden*orden, MPI_INT, rank-1,
        tag, MPI_COMM_WORLD, &status);
    sarrusPorRank(rank,tmp,A);
    Diagonales[0] += tmp[0];
    Diagonales[1] += tmp[1];
    MPI_Send(Diagonales, 2, MPI_LONG_DOUBLE,
        rank+1, tag, MPI_COMM_WORLD);
    MPI_Send(A, orden*orden, MPI_INT, rank+1,
        tag, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}

void llenarMatriz(int matriz[orden][orden]){
  for(int i = 0; i < orden; i++){
    for(int j = 0; j < orden; j++){
      matriz[i][j] = rand()%20+1;
    }
  }
}

void imprimirMatriz(int matriz[orden][orden]){
  printf("\n");
  for(int i = 0; i < orden; i++){
    printf("\t|");
    for(int j = 0; j < orden; j++){
      printf("%2d|",matriz[i][j]);
    }
    printf("\n");
  }
}

void sarrusPorRank(int rank, long double
    DiagArray[2], int matriz[orden][orden]){
  int from = (rank * orden)/numeroDeRanks;
  int to = ((rank+1) * orden)/numeroDeRanks;
  long double temporalNegativa;
  long double temporalPositiva;
  int i,j;
  int fila;
  DiagArray[0] = 0;
  DiagArray[1] = 0;

  printf("computing rank %d (from row %d to
      %d)\n", rank, from, to-1);
  for(i = from; i < to; i++){
    temporalNegativa = 1;
    temporalPositiva = 1;
    for(j = 0; j < orden; j++){
      fila = i+j;
```

```
    if(fila >= orden){
      fila -= orden;
    }
    temporalPositiva *= matriz[fila][j];
    //printf("Pos: A[%d][%d]\n",fila,j);
    temporalNegativa *=
        matriz[fila][(orden-1)-j];
    //printf("Neg:
        A[%d][%d]\n",fila,(orden-1)-j);
   }
   DiagArray[0] += temporalNegativa;
   DiagArray[1] += temporalPositiva;
  }
  //printf("Pos final: %Lf rank:
      %d\n",DiagArray[1],rank);
  //printf("Neg final: %Lf rank:
      %d\n",DiagArray[0],rank);
  printf("finished rank %d\n", rank);
}
```