TRABAJO PRACTICO FINAL PARADIGMAS DE COMPUTACIÓN PARALELA, CONCURRENTE Y DISTRIBUIDA

No Author Given

No Institute Given

Abstract. The abstract should summarize the contents of the paper and should contain at least 70 and at most 150 words. It should be written using the *abstract* environment.

Keywords: We would like to encourage you to list your keywords within the abstract section

1 Introducción

El siguiente informe ha sido elaborado para dar cumplimiento a lo requerido en el Trabajo Práctico Final de Paradigmas de Computación Paralela, Concurrente y Distribuida. En este trabajo, se solicita la implementación de la multiplicación vectorial y escalar de dos vectores de 100 elementos, utilizando un lenguaje de programación y una librería de programación en paralelo previamente seleccionados.

Para llevar a cabo esta tarea, se ha elegido el lenguaje de programación ANSI C y la librería de programación en paralelo MPI. Se implementará la multiplicación vectorial y escalar de dos vectores de 100 elementos. Donde el primer vector contendrá como elementos los primeros 100 términos de la serie de números naturales impares y el segundo vector tendrá los primeros 100 términos de la serie de números naturales pares (incluyendo el 0).

El objetivo de la implementación es obtener el resultado de la multiplicación de los vectores y medir el tiempo empleado en el cálculo. Además, se llevará a cabo una versión paralela y secuencial del programa para poder comparar los resultados obtenidos.

Posteriormente, se realizará la misma tarea para dos vectores de tamaños crecientes: 1000, 5.000, 10.000, 50.000, 100.000, 200.000, 300.000, 400.000, 500.000, 1.000.000 y 10.000.000 de elementos, siempre y cuando sea soportado por el hardware utilizado.

Una vez que se hayan obtenido los resultados, se graficarán los tiempos paralelos y secuenciales para cada tamaño de vector. A continuación, se realizará una explicación de los valores obtenidos, estableciendo su relación con el concepto de Speedup y Eficiencia. El Speedup representa la mejora de rendimiento obtenida

al utilizar la programación paralela en comparación con la secuencial, mientras que la Eficiencia se refiere a la proporción de tiempo empleado en relación al número de procesadores utilizados.

2 Desarrollo

2.1 Elección del lenguaje de programación

La elección del lenguaje ANSI C para completar el ejercicio de paradigmas de procesamiento paralelo y distribuido se basa en varias razones fundamentales.

En primer lugar, se selecciona el lenguaje ANSI C debido a su eficiencia y capacidad para ofrecer un control preciso sobre los recursos del sistema. ANSI C es un lenguaje de bajo nivel que permite optimizar el código y acceder directamente a la memoria, lo cual resulta crucial en entornos de procesamiento paralelo y distribuido.

ANSI C permite una gestión eficiente de los recursos y ofrece un alto grado de control sobre el rendimiento de las aplicaciones.

Además, el lenguaje ANSI C cuenta con un amplio soporte en el ámbito de la programación paralela y distribuida. Existen numerosas bibliotecas y herramientas diseñadas específicamente para trabajar con ANSI C en entornos de procesamiento paralelo, como la librería MPI. Esta librería, es una interfaz estándar ampliamente utilizada para la comunicación entre procesos en sistemas paralelos y distribuidos, y ANSI C es uno de los lenguajes compatibles con esta librería.

Por último, ANSI C ha demostrado ser capaz de manejar eficientemente cálculos intensivos y algoritmos paralelos.

2.2 Elección de la librería para programación en paralelo

Se ha optado por la librería MPI. Esta selección ha sido hecha por varias razones, que se pasan a explicar. En primer lugar, MPI (Message Passing Interface) es una interfaz estándar ampliamente utilizada en programación paralela y distribuida, lo que garantiza su compatibilidad con una amplia gama de sistemas y plataformas.

Además, MPI ofrece una gran flexibilidad y eficiencia en la comunicación entre los diferentes procesos paralelos. Proporciona un conjunto de funciones que permiten el intercambio de mensajes y datos entre los nodos de un sistema distribuido, lo cual es fundamental en la implementación de algoritmos paralelos y la sincronización de tareas.

Otra ventaja significativa de MPI es su capacidad para escalar el rendimiento a medida que se agregan más nodos de procesamiento. Esto significa que a medida que se aumenta la cantidad de recursos disponibles, como procesadores y memoria, la librería MPI puede aprovecharlos de manera efectiva y distribuir la carga de trabajo de manera equitativa.

Además, MPI ofrece mecanismos avanzados para el manejo de errores y recuperación, lo que es crucial en entornos de cómputo paralelo donde pueden ocurrir fallas o interrupciones en el sistema.

Finalmente, la elección de la librería MPI para el procesamiento paralelo se basa en su amplia adopción, compatibilidad, flexibilidad en la comunicación, capacidad de escalamiento y manejo de errores. Estas características hacen de MPI una opción sólida y confiable para implementar aplicaciones paralelas y cumplir con los requisitos del trabajo práctico.

2.3 Implemente la multiplicación escalar, en programación secuencial.

```
/*
// Programa: ProductoEscalarS.c
// gcc ProductoEscalarS.c -o ProductoEscalarS -Wall -pedantic
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define CANTIDAD_DEF 100 // cantidad de elementos por defecto

int main(int argc, char *argv[]) {
  int cantidadElementos = CANTIDAD_DEF;
  int par, impar, count,  outCsv = 0, encabezado = 0;
  long long int productoEscalar = 0;

  struct timeval inicio, fin; // estructuras para medir el tiempo de ejecución en micros
  clock_t iniSeg, finSeg; // variables para medir el tiempo de ejecución en segundos


  // Se verifican los parámetros
  // TO DO: Se puede mejorar el control de parámetros. Pero no es el onjeto de estudio
  if (argc < 2) {
    printf("No se han especificado valores correctos para los parámetros requeridos.\n")
    printf("La cantidad de elementos serán: %d sin formato CSV.\n", cantidadElementos);
    printf("Uso: ProductoEscalarS <cantidadElementos> <csv> <encabezado>\n\n");
  } else {
    if (argc >= 2) {
      cantidadElementos = atoi(argv[1]);
    }
    if (argc >= 3)
      outCsv=1;
```

```
  if (argc == 4)
    encabezado = 1;
}


// guardar el tiempo de inicio
iniSeg = clock();
gettimeofday(&inicio, NULL);


// Creo los vectores
int* vectorA;
int* vectorB;

// Asignar memoria para los vectores
vectorA = (int*) malloc(cantidadElementos * sizeof(int));
vectorB = (int*) malloc(cantidadElementos * sizeof(int));

// Verificar si la asignación de memoria fue exitosa
if ((vectorA == NULL) || (vectorB == NULL)) {
    printf("No se pudo asignar memoria\n");
    exit(1);
}

//Cargar los vectores
par = 0;
impar = 0;
for (count = 0; count < (cantidadElementos * 2) - 1; ++count) {
  if (count % 2 == 0) { // si es par, lo agrega al VectorA
    vectorA[par] = count;
    par++; // aumentar el contador de números pares
  } else { // si es impar, lo agrega al VectorB
    vectorB[impar] = count;
    impar++; // aumentar el contador de números impares
  }
}

// Calculo del producto escalar entre dos vectores
productoEscalar = 0;
for (count = 0; count < cantidadElementos; ++count) {
    productoEscalar += vectorA[count] * vectorB[count];
}

// guardar el tiempo de finalización
```

```
  finSeg = clock();
  gettimeofday(&fin, NULL);

  // calcular y mostrar el tiempo de ejecución en microsegundos
  double tiempo = (double)(fin.tv_usec - inicio.tv_usec) + (double)(fin.tv_sec - inicio.
  // calcular y mostrar el tiempo de ejecución en segundos
  double tiempoSeg = ((double) (finSeg - iniSeg)) / CLOCKS_PER_SEC;

  if (outCsv != 1){
    printf("Ejecución secuencial del producto escalar entre\n");
    printf("dos vectores de %d elementos es: %lli\n", cantidadElementos, productoEscalar
    printf("Tiempo de ejecución: %f microsegundos (ţs)\n", tiempo);
    printf("Tiempo de ejecución: %f segundos\n\n\n", tiempoSeg);
  } else {
    if (encabezado == 1)
      printf("cantidadElementos,productoEscalar,microsegundosEjec,segundosEjec\n");

    printf("%d,%lli,%f,%f\n", cantidadElementos, productoEscalar, tiempo, tiempoSeg);
  }

  // Liberar la memoria asignada a los vectores
  free(vectorA);
  free(vectorB);

  return 0;
}
```

El código del programa *ProductoEscalarS.c* implementa el cálculo del producto escalar entre dos vectores en programación secuencial. A continuación, describiré paso a paso la implementación:

1. Se incluyen las bibliotecas necesarias para el funcionamiento del programa, como stdio.h, stdlib.h, time.h y sys/time.h.
2. Se define la constante CANTIDAD_DEF con un valor predeterminado de 100, que representa la cantidad de elementos en los vectores.
3. Se declara la función principal main con los parámetros argc y argv[].
4. Se declaran las variables necesarias, como cantidadElementos para almacenar la cantidad de elementos en los vectores, par e impar para contar la cantidad de números pares e impares, respectivamente, count para controlar los bucles, outCsv para indicar si se generará en formato CSV la salida, encabezado para determinar si se imprimirá un encabezado en la salida y productoEscalar para almacenar el resultado del producto escalar.
5. Se declaran estructuras de tiempo, inicio y fin, utilizando struct timeval para medir el tiempo de ejecución en microsegundos, y las variables iniSeg y finSeg utilizando clock_t para medir el tiempo de ejecución en segundos.

6. Se realiza una verificación de los parámetros proporcionados a través de argc para determinar si se especificó la cantidad de elementos, la opción de formato CSV y la opción de encabezado.
7. Se asigna el valor proporcionado o predeterminado a cantidadElementos dependiendo de los parámetros ingresados.
8. Se actualizan las variables outCsv y encabezado según los parámetros ingresados.
9. Se guarda el tiempo de inicio de la ejecución utilizando clock() y gettimeofday().
10. Se crean los vectores vectorA y vectorB utilizando punteros a enteros.
11. Se asigna memoria dinámicamente a los vectores utilizando malloc según la cantidad de elementos especificada en cantidadElementos.
12. Se verifica si la asignación de memoria fue exitosa. Si alguno de los vectores es NULL, se imprime un mensaje de error y se sale del programa.
13. Se cargan los vectores vectorA y vectorB con números pares e impares, respectivamente, utilizando un bucle for. Los números pares se agregan a vectorA y los impares a vectorB.
14. Se calcula el producto escalar entre los dos vectores utilizando un bucle for. Se multiplica cada elemento correspondiente en vectorA y vectorB y se suma al resultado en productoEscalar.
15. Se guarda el tiempo de finalización de la ejecución utilizando clock() y gettimeofday().
16. Se calcula el tiempo de ejecución en microsegundos y segundos utilizando las diferencias de tiempo almacenadas.
17. Si la opción de formato CSV no está habilitada, se imprime el resultado del producto escalar y el tiempo de ejecución en microsegundos y segundos.
18. Si la opción de formato CSV está habilitada, se imprime un encabezado si corresponde y se imprime la cantidad de elementos, el producto escalar y los tiempos de ejecución en microsegundos y segundos separados por comas.
19. Se libera la memoria asignada a los vectores utilizando free.
20. Se retorna 0 para indicar que el programa se ejecutó correctamente.

Este código, como se comentó con anterioridad, calcula el producto escalar de dos vectores, de acuerdo a las convenciones establecidas en el seminario, en programación secuencial. El primer vector tiene como elementos los primeros 100 términos de la serie de números naturales impares. El segundo vector contiene los primeros 100 términos de la serie de números naturales pares (incluyendo el 0). Muestra el resultado, así como el tiempo de ejecución en microsegundos y segundos. También puede generar la salida en formato CSV con los resultados si se especifica la opción correspondiente.

2.4 Implemente la multiplicación vectorial, en programación secuencial.

```
/*
// Programa: ProductoEscalarP.c
```

```
// % mpicc -o ProductoEscalarP ProductoEscalarP.c
// % mpirun -np 2 ProductoEscalarP
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <mpi.h>

// cantidad de elementos por defecto
#define CANTIDAD_DEF 100

int main(int argc, char *argv[]) {
  int cantidadElementos = CANTIDAD_DEF;
  int par, impar, count,  outCsv = 0, encabezado = 0;
  long long int productoEscalar = 0, productoEscalarParcial = 0;

  // estructuras para medir el tiempo de ejecución en microsegundos
  struct timeval inicio, fin;
  // variables para medir el tiempo de ejecución en segundos
  clock_t iniSeg, finSeg;

  // Inicializar MPI
  int miRango, numProcs;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &miRango);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

  // Se verifican los parámetros
  // TO DO: Se puede mejorar el control de parámetros. Pero no es
  // el objeto de estudio
  if (miRango == 0) {
    if (argc < 2) {
        printf("No se han especificado valores correctos para los parámetros requeridos.
        printf("La cantidad de elementos serán: %d sin formato CSV.\n", cantidadElemento
        printf("Utilizando %d procesos\n", numProcs);
        printf("Uso: ProductoEscalarP <cantidadElementos> <csv> <encabezado>\n\n");
    } else {
      if (argc >= 2) {
        cantidadElementos = atoi(argv[1]);
      }
      if (argc >= 3)
        outCsv=1;
```

```
    if (argc == 4)
      encabezado = 1;
  }
}

// Dividir el trabajo entre los procesos
int elementosPorProceso = cantidadElementos / numProcs;
int elementosRestantes = cantidadElementos % numProcs;

// guardar el tiempo de inicio
iniSeg = clock();
gettimeofday(&inicio, NULL);

// Creo los vectores
int* vectorA;
int* vectorB;

// Asignar memoria para los vectores
vectorA = (int*) malloc(elementosPorProceso * sizeof(int));
vectorB = (int*) malloc(elementosPorProceso * sizeof(int));

// Verificar si la asignación de memoria fue exitosa
if ((vectorA == NULL) || (vectorB == NULL)) {
    printf("No se pudo asignar memoria\n");
    MPI_Finalize(); // Finalizar MPI
    exit(1);
}

//Cargar los vectores
par = 0;
impar = 0;
for (count = miRango * elementosPorProceso; count < (miRango + 1) * elementosPorProces
  if (count % 2 == 0) { // si es par, lo agrega al VectorA
    vectorA[par] = count;
    par++; // aumentar el contador de números pares
  } else { // si es impar, lo agrega al VectorB
    vectorB[impar] = count;
    impar++; // aumentar el contador de números impares
  }
}

// Si hay elementos restantes, se procesan en el proceso 0
if (miRango == 0 && elementosRestantes > 0) {
  for (count = cantidadElementos - elementosRestantes; count < cantidadElementos; ++co
    if (count % 2 == 0) { // si es par, lo agrega al VectorA
```

```
      vectorA[par] = count;
      par++; // aumentar el contador de números pares
    } else { // si es impar, lo agrega al VectorB
      vectorB[impar] = count;
      impar++; // aumentar el contador de números impares
    }
  }
}

// Calcular el producto escalar parcial
productoEscalarParcial = 0;
for (count = 0; count < elementosPorProceso; ++count) {
    productoEscalarParcial += vectorA[count] * vectorB[count];
}

// Sumar los productos escalares parciales de todos los procesos
MPI_Reduce(&productoEscalarParcial, // Elemento a enviar
           &productoEscalar, // Variable donde se almacena la reunion de los datos
           1, // Cantidad de datos a reunir
           MPI_LONG_LONG_INT, // Tipo del dato que se reunira
           MPI_SUM, // Operacion aritmetica a aplicar
           0, // Proceso que recibira los datos
           MPI_COMM_WORLD); // Comunicador

// guardar el tiempo de finalización
finSeg = clock();
gettimeofday(&fin, NULL);

// calcular y mostrar el tiempo de ejecución en microsegundos
double tiempo = (double)(fin.tv_usec - inicio.tv_usec) + (double)(fin.tv_sec - inicio.
// calcular y mostrar el tiempo de ejecución en segundos
double tiempoSeg = ((double) (finSeg - iniSeg)) / CLOCKS_PER_SEC;

// Mostrar el resultado
if (miRango == 0) {
  if (outCsv != 1){
    printf("Ejecución paralela del producto escalar entre\n");
    printf("dos vectores de %d elementos es: %lli\n", cantidadElementos, productoEscal
    printf("Tiempo de ejecución: %f microsegundos (ţs)\n", tiempo);
    printf("Tiempo de ejecución: %f segundos\n\n\n", tiempoSeg);
  } else {
    if (encabezado == 1)
      printf("cantidadElementos, cantidadProcesos,productoEscalar,microsegundosEjec,se

    printf("%d,%d,%lli,%f,%f\n", cantidadElementos, numProcs, productoEscalar, tiempo,
```

```
    }
  }

  // Liberar la memoria asignada a los vectores
  free(vectorA);
  free(vectorB);

  // Finalizar MPI
  MPI_Finalize();

  return 0;
}
```

Es implementado en este archivo de programa (ProductoEscalarP.c) el cálculo del producto escalar en programación paralela utilizando MPI (Message Passing Interface). El código de programa explicado en el apartado 2.3 sirvió como base, donde la multiplicación escalar fue implementada en programación secuencial. Por lo tanto, para no ser redundantes, las líneas de código donde se implementa la programación paralela son el enfoque principal. A continuación, se describe la implementación:

1. Se incluyen las bibliotecas necesarias para el funcionamiento del programa y mpi.h (para MPI).
2. Luego de declarar e inicializar las estructuras, variables, constantes y establecer los valores predeterminados, se inicializa MPI utilizando MPI_Init().
3. Se obtiene el rango del proceso actual (miRango) y el número total de procesos (numProcs) utilizando MPI_Comm_rank() y MPI_Comm_size().
4. Se verifica si se han proporcionado los parámetros requeridos en la línea de comandos. Si se proporcionaron los parámetros requeridos, se actualiza la variable, según los valores proporcionados. En caso contrario, se muestra un mensaje de error y se imprime la cantidad de elementos por defecto, la cantidad de procesadores y el formato de uso del programa.
5. Se divide el trabajo entre los procesos utilizando elementosPorProceso y elementosRestantes.
6. Se asigna memoria dinámicamente a los vectores vectorA y vectorB utilizando malloc().
7. Se verifica si la asignación de memoria fue exitosa. Si alguno de los vectores es NULL, se imprime un mensaje de error, se finaliza el programa y se llama a MPI_Finalize() para finalizar MPI.
8. Se llenan los vectores vectorA y vectorB con los números pares e impares respectivamente, utilizando bucles for.
9. Si hay elementos restantes, se procesan en el proceso 0.
10. Se calcula el producto escalar parcial entre los elementos de los vectores vectorA y vectorB.
11. Se utiliza MPI_Reduce() para sumar los productos escalares parciales de todos los procesos y obtener el producto escalar final en el proceso 0.

12. Se guarda el tiempo de finalización de la ejecución. Y se calcula el tiempo de ejecución en microsegundos y en segundos.
13. Se muestra el resultado final en el proceso 0. Tomando en cuenta los parámetros recibidos en la línea de comandos.
14. Se libera la memoria asignada a los vectores vectorA y vectorB utilizando free().
15. Se llama a MPI_Finalize() para finalizar MPI.
16. Se retorna 0 para indicar que el programa se ejecutó correctamente.

En este código de programa, a diferencia del explicado en el apartado 2.3, se utiliza la librería MPI para implementar la programación paralela. Para ello, se hace uso de varias funciones de MPI. En primer lugar, la función MPI_Init es empleada para inicializar MPI y establecer el entorno de comunicación. A continuación, la función MPI_Comm_rank es utilizada para obtener el rango del proceso, es decir, su identificador único. Asimismo, se emplea la función MPI_Comm_size para obtener el número total de procesos involucrados en la comunicación.

Además, se hace uso de la función MPI_Reduce, la cual permite sumar los productos escalares parciales de todos los procesos. Esta función es utilizada para realizar una operación de reducción en la cual se combina el resultado de cada proceso en uno solo.

Por otro lado, la función MPI_Finalize es llamada para finalizar MPI y liberar los recursos asociados con la comunicación una vez que se ha completado la ejecución del programa.

En cuanto a la distribución del trabajo entre los procesos, se divide utilizando la variable "elementosPorProceso". Cada proceso se encarga de una porción del trabajo, mientras que los elementos restantes son manejados en el proceso 0.

2.5 Implemente la multiplicación vectorial, en programación secuencial.

```
/*
// Programa: ProductoVectorialS.c
// gcc ProductoVectorialS.c -o ProsuctoVectorialS -Wall -pedantic
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define CANTIDAD_DEF 100 // cantidad de elementos por defecto

int main(int argc, char *argv[]) {
    int cantidadElementos = CANTIDAD_DEF;
    int outCsv = 0, encabezado = 0;
```

```
int count1, count2;
int *vectorA, *vectorB;
long long int *productoVectorial = 0;

struct timeval inicio, fin; // estructuras para medir el tiempo de ejecución en micr
clock_t iniSeg, finSeg; // variables para medir el tiempo de ejecución en segundos

// Se verifican los parámetros
// TO DO: Se puede mejorar el control de parámetros. Pero no es el onjeto de estudio
if (argc < 2) {
    printf("No se han especificado valores correctos para los parámetros requeridos.
    printf("La cantidad de elementos serán: %d sin formato CSV.\n", cantidadElemento
    printf("Uso: ProductoEscalarS <cantidadElementos> <csv> <encabezado>\n\n");
} else {
if (argc >= 2) {
    cantidadElementos = atoi(argv[1]);
}
if (argc >= 3)
    outCsv=1;

if (argc == 4)
    encabezado = 1;
}

// guardar el tiempo de inicio
iniSeg = clock();
gettimeofday(&inicio, NULL);

// Asignación dinámica de memoria para los vectores y el resultado
vectorA = (int*)malloc(cantidadElementos * sizeof(int));
vectorB = (int*)malloc(cantidadElementos * sizeof(int));
productoVectorial = (long long int*)malloc(cantidadElementos * sizeof(long long int)

// Verificar si la asignación de memoria fue exitosa
if ((vectorA == NULL) || (vectorB == NULL) || (productoVectorial == NULL)) {
    printf("No se pudo asignar memoria\n");
    exit(1);
}

// Llenar el primer vector con los primeros n números naturales impares
for (count1 = 0; count1 < cantidadElementos; count1++) {
    vectorA[count1] = 2*count1 + 1;
}

// Llenar el segundo vector con los primeros n números naturales pares
```

```
for (count1 = 0; count1 < cantidadElementos; count1++) {
    vectorB[count1] = 2*count1;
}

// Multiplicación de vectores utilizando el algoritmo de
//la convensión aportada en el seminario
for (count1 = 0; count1 < cantidadElementos; count1++) {
    productoVectorial[count1] = 0;
    for (count2 = 0; count2 < cantidadElementos; count2++) {
        productoVectorial[count1] += vectorA[count1] * vectorB[count2];
    }
}


// guardar el tiempo de finalización
finSeg = clock();
gettimeofday(&fin, NULL);

// calcular y mostrar el tiempo de ejecución en microsegundos
double tiempo = (double)(fin.tv_usec - inicio.tv_usec) + (double)(fin.tv_sec - inici
// calcular y mostrar el tiempo de ejecución en segundos
double tiempoSeg = ((double) (finSeg - iniSeg)) / CLOCKS_PER_SEC;

if (outCsv != 1){
    printf("Ejecución secuencial del producto vectorial entre\n");
    printf("dos vectores de %d elementos\n", cantidadElementos);
    // Imprimir el resultado del productoVectorial
    //for (count1 = 0; count1 < cantidadElementos; count1++) {
    //    printf("%lli ", productoVectorial[count1]);
    //}
    printf("\n");
    printf("Tiempo de ejecución: %f microsegundos (ţs)\n", tiempo);
    printf("Tiempo de ejecución: %f segundos\n\n\n", tiempoSeg);
} else {
    if (encabezado == 1){
      //printf("cantidadElementos,productoVectorial,microsegundosEjec,segundosEjec\n
        printf("cantidadElementos,microsegundosEjec,segundosEjec\n");
    }
    //printf("%d,%lln,%f,%f\n", cantidadElementos, productoVectorial, tiempo, tiempo
    printf("%d,%f,%f\n", cantidadElementos, tiempo, tiempoSeg);
}

// Liberar la memoria asignada a los vectores y el resultado
free(vectorA);
free(vectorB);
```

```
    free(productoVectorial);

    return 0;
}
```

Este código de programación, contenido en el archivo ProductoVectorialS.c, implementa el cálculo del producto vectorial en programación secuencial. A continuación, describiré paso a paso la implementación:

1. Se incluyen las bibliotecas necesarias para el funcionamiento del programa, como stdio.h, stdlib.h, time.h y sys/time.h.
2. Se define la constante CANTIDAD_DEF con un valor predeterminado de 100, que representa la cantidad de elementos por defecto en los vectores.
3. Se declara la función principal main con los parámetros argc y argv[].
4. Se declara la variable cantidadElementos e se inicializa con el valor de CANTIDAD_DEF por defecto.
5. Se declara la variable outCsv e encabezado y se inicializan en 0, que se utilizarán para controlar la generación la salida en formato CSV y la inclusión de un encabezado en la salida.
6. Se declaran las variables count1 y count2 para utilizar en bucles y las variables de puntero vectorA, vectorB y productoVectorial.
7. Se declaran estructuras de tiempo, inicio y fin, utilizando struct timeval para medir el tiempo de ejecución en microsegundos, y las variables iniSeg y finSeg utilizando clock_t para medir el tiempo de ejecución en segundos.
8. Se verifica si se han proporcionado los parámetros requeridos en la línea de comandos. Si no se proporcionaron, se muestra un mensaje de error y se imprime la cantidad de elementos por defecto y el formato de uso del programa.
9. Si se proporcionaron los parámetros requeridos, se actualiza la variable cantidadElementos según el valor proporcionado.
10. Se verifica si se proporcionaron los parámetros opcionales para el formato CSV y el encabezado. Si se proporcionaron, se actualizan las variables outCsv y encabezado en consecuencia.
11. Se guarda el tiempo de inicio de la ejecución utilizando clock() y gettimeofday().
12. Se asigna memoria dinámicamente a los vectores vectorA, vectorB y productoVectorial según la cantidad de elementos.
13. Se verifica si la asignación de memoria fue exitosa. Si alguno de los vectores es NULL, se imprime un mensaje de error, se finaliza el programa.
14. Se llenan los vectores vectorA y vectorB con los números naturales impares y pares, respectivamente, utilizando bucles for.
15. Se realiza el cálculo del producto vectorial utilizando un bucle anidado. Para cada elemento del vector resultante, se suma el producto de los elementos correspondientes de vectorA y vectorB.
16. Se guarda el tiempo de finalización de la ejecución utilizando clock() y gettimeofday().

17. Se calcula el tiempo de ejecución en microsegundos (tiempo) y en segundos (tiempoSeg).
18. Si no se especificó la opción de formato CSV, se imprime por pantalla el resultado del producto vectorial y el tiempo de ejecución en microsegundos y segundos.
19. Si se especificó la opción de formato CSV, se imprime un encabezado en el archivo CSV si se proporcionó la opción de encabezado. Luego se imprime la cantidad de elementos, el tiempo de ejecución en microsegundos y el tiempo de ejecución en segundos en el archivo CSV.
20. Se libera la memoria asignada a los vectores vectorA, vectorB y productoVectorial utilizando free().
21. Se retorna 0 para indicar que el programa se ejecutó correctamente.

El producto vectorial de dos vectores es calculado por este código, en programación secuencial, según lo discutido previamente. Las convenciones establecidas en el seminario sobre producto vectorial y escalar son seguidas. El primer vector está compuesto por los primeros 100 términos de la serie de números naturales impares. El segundo vector está compuesto por los primeros 100 términos de la serie de números naturales pares, incluyendo el 0. El resultado, junto con el tiempo de ejecución en microsegundos y segundos, es mostrado. Si se especifica la opción correspondiente, también puede generar la salida en formato CSV con los resultados.

2.6 Implemente la multiplicación vectorial, en programación paralela.

```
/*
// Programa: ProductoVectorialP.c
// % mpicc -o ProductoVectorialP ProductoVectorialP.c
// % mpirun -np 2 ProductoVectorialP
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <mpi.h>

#define CANTIDAD_DEF 100 // cantidad de elementos por defecto

int main(int argc, char *argv[]) {
    int cantidadElementos = CANTIDAD_DEF;
    int outCsv = 0, encabezado = 0;
    int count1, count2;
    int *vectorA, *vectorB;
    long long int *productoVectorial = 0;
```

```
struct timeval inicio, fin; // estructuras para medir el tiempo de ejecución en micr
clock_t iniSeg, finSeg; // variables para medir el tiempo de ejecución en segundos

int miRango, numProcs;

// Inicializar MPI
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &miRango);
MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

// Se verifican los parámetros
// TO DO: Se puede mejorar el control de parámetros. Pero no es el objeto de estudio
if (miRango == 0) {
    if (argc < 2) {
        printf("No se han especificado valores correctos para los parámetros requeri
        printf("La cantidad de elementos serán: %d sin formato CSV.\n", cantidadElem
        printf("Uso: ProductoVectorialP <cantidadElementos> <csv> <encabezado>\n\n")
    } else {
        if (argc >= 2) {
            cantidadElementos = atoi(argv[1]);
        }
        if (argc >= 3)
            outCsv = 1;

        if (argc == 4)
            encabezado = 1;
    }
}

//Difundir datos
MPI_Bcast(&cantidadElementos, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&outCsv, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&encabezado, 1, MPI_INT, 0, MPI_COMM_WORLD);

// guardar el tiempo de inicio
iniSeg = clock();
gettimeofday(&inicio, NULL);

// Asignación dinámica de memoria para los vectores y el resultado
vectorA = (int *)malloc(cantidadElementos * sizeof(int));
vectorB = (int *)malloc(cantidadElementos * sizeof(int));
productoVectorial = (long long int *)malloc(cantidadElementos * sizeof(long long int

// Verificar si la asignación de memoria fue exitosa
```

```
if ((vectorA == NULL) || (vectorB == NULL) || (productoVectorial == NULL)) {
    printf("No se pudo asignar memoria\n");
    exit(1);
}

// Llenar el primer vector con los primeros n números naturales impares
for (count1 = miRango; count1 < cantidadElementos; count1 += numProcs) {
    vectorA[count1] = 2 * count1 + 1;
}

// Llenar el segundo vector con los primeros n números naturales pares
for (count1 = miRango; count1 < cantidadElementos; count1 += numProcs) {
    vectorB[count1] = 2 * count1;
}

// Multiplicación de vectores utilizando el algoritmo de
// la convención aportada en el seminario
for (count1 = miRango; count1 < cantidadElementos; count1 += numProcs) {
    productoVectorial[count1] = 0;
    for (count2 = 0; count2 < cantidadElementos; count2++) {
        productoVectorial[count1] += vectorA[count1] * vectorB[count2];
    }
}

long long int *resultados;
if (miRango == 0) {
    resultados = (long long int *)malloc(cantidadElementos * sizeof(long long int));
}

//Reunir los resultados
MPI_Gather(productoVectorial,
           cantidadElementos / numProcs,
           MPI_LONG_LONG_INT,
           resultados,
           cantidadElementos / numProcs, MPI_LONG_LONG_INT,
           0,
           MPI_COMM_WORLD);

// guardar el tiempo de finalización
finSeg = clock();
gettimeofday(&fin, NULL);

// calcular y mostrar el tiempo de ejecución en microsegundos
double tiempo = (double)(fin.tv_usec - inicio.tv_usec) + (double)(fin.tv_sec - inici
// calcular y mostrar el tiempo de ejecución en segundos
```

```
    double tiempoSeg = ((double)(finSeg - iniSeg)) / CLOCKS_PER_SEC;

    if (miRango == 0) {
        if (outCsv != 1) {
            printf("Ejecución paralela del producto vectorial entre\n");
            printf("dos vectores de %d elementos\n", cantidadElementos);
            printf("utilizando %d procesadores\n", numProcs);
            // Imprimir el resultado del productoVectorial
            // for (count1 = 0; count1 < cantidadElementos; count1++) {
            //    printf("%lli ", resultados[count1]);
            // }
            printf("\n");
            printf("Tiempo de ejecución: %f microsegundos (ţs)\n", tiempo);
            printf("Tiempo de ejecución: %f segundos\n\n\n", tiempoSeg);
        } else {
            if (encabezado == 1) {
                // printf("cantidadElementos,productoVectorial,microsegundosEjec,segundo
                printf("cantidadElementos,numProcs,microsegundosEjec,segundosEjec\n");
            }
            // printf("%d,%lln,%f,%f\n", cantidadElementos, numProcs,resultados, tiempo,
            printf("%d,%d,%f,%f\n", cantidadElementos, numProcs, tiempo, tiempoSeg);
        }

        free(resultados);
    }

    // Liberar la memoria asignada a los vectores y el resultado
    free(vectorA);
    free(vectorB);
    free(productoVectorial);

    // Finalizar MPI
    MPI_Finalize();

    return 0;
}
```

El código expuesto en el programa ProductoVectorialP.c implementa el cálculo del producto vectorial en programación paralela utilizando MPI (Message Passing Interface). Como se mencionó en el apartado anterior nos hemos basado en el código de programación desarrollado en el apartado 2.5, donde el producto vectorial fue implementado en programación secuencial. Por lo tanto, nos enfocaremos en describir las líneas de código donde se implementa la programación paralela, a continuación:

1. Se incluyen las bibliotecas necesarias para el funcionamiento del programa y mpi.h (para MPI).
2. Luego de declarar e inicializar las estructuras, variables, constantes y establecer los valores predeterminados, se inicializa MPI utilizando MPI_Init().
3. Se obtiene el rango del proceso actual (miRango) y el número total de procesos (numProcs) utilizando MPI_Comm_rank() y MPI_Comm_size().
4. Se verifica si se han proporcionado los parámetros requeridos en la línea de comandos. Si se proporcionaron los parámetros requeridos, se actualiza la variable, según los valores proporcionados. En caso contrario, se muestra un mensaje de error y se imprime la cantidad de elementos por defecto, la cantidad de procesos y el formato de uso del programa.
5. Se difunden los datos de cantidadElementos, outCsv y encabezado a todos los procesos utilizando MPI_Bcast().
6. Se asigna memoria dinámicamente a los vectores vectorA, vectorB y productoVectorial utilizando malloc().
7. Se verifica si la asignación de memoria fue exitosa. Si alguno de los vectores es NULL, se imprime un mensaje de error, se finaliza el programa y se llama a MPI_Finalize() para finalizar MPI.
8. En cada proceso, se llena el vector vectorA con los primeros n números naturales impares y el vector vectorB con los primeros n números naturales pares, utilizando bucles for con un salto de numProcs.
9. Cada proceso calcula su parte del producto vectorial utilizando un bucle for anidado. Se inicializa el elemento correspondiente del vector productoVectorial a 0 y se realizan las multiplicaciones correspondientes.
10. En el proceso 0, se asigna memoria dinámicamente al vector resultados para almacenar los resultados finales.
11. Se utiliza MPI_Gather() para recopilar los resultados parciales de cada proceso en el proceso 0.
12. Se guarda el tiempo de finalización de la ejecución. Y se calcula el tiempo de ejecución en microsegundos y en segundos.
13. Se muestra el resultado final en el proceso 0. Tomando en cuenta los parámetros recibidos en la línea de comandos.
14. Se libera la memoria asignada a los vectores vectorA, vectorB y productoVectorial utilizando free().
15. Se llama a MPI_Finalize() para finalizar MPI.
16. Se retorna 0 para indicar que el programa se ejecutó correctamente.

Para implementar la librería MPI en el código de ProsuctoVectorialS.c (apartado 2.4), es necesario establecer la comunicación entre los procesos. Para esto, se debe dividir el trabajo y asignar una parte de los vectores a cada proceso. Cada proceso debe calcular la suma de los productos de los elementos correspondientes de los dos vectores asignados y enviar el resultado al proceso principal. El proceso principal luego suma los resultados para obtener el producto escalar total.

En el código estudiado, la comunicación entre los procesos es establecida utilizando la librería MPI (Message Passing Interface). Un conjunto de funciones y rutinas es proporcionado por esta librería, permitiendo que los datos sean

intercambiados y las acciones sean coordinadas por los procesos en un entorno de programación paralela. Los siguientes pasos fueron seguidos para implementar la comunicación entre los procesos:

1. La *inicialización* se realiza llamando a la función MPI_Init por parte de cada proceso, con el objetivo de inicializar MPI y establecer el entorno de comunicación. Es importante destacar que esta función debe ser llamada antes de cualquier otra función MPI.
2. La *identificación del rango* y tamaño se realiza mediante la llamada a la función MPI_Comm_rank por parte de cada proceso, con el fin de obtener su identificador único, conocido como "rango". El rango es un número entero que abarca desde 0 hasta el número total de procesos -1. Asimismo, la función MPI_Comm_size es utilizada para determinar el número total de procesos involucrados en la comunicación.
3. En la *definición del proceso emisor y receptor* del mensaje, se selecciona el contenido del mensaje y se almacena en una variable en el proceso emisor, el cual posee los datos que se desean enviar.
4. El *envío del mensaje* se realiza utilizando la función MPI_Bcast, la cual transmite el contenido del mensaje desde el proceso emisor a todos los demás procesos. Esta función es llamada en todos los procesos, tanto en el emisor como en los receptores. Los parámetros de la función incluyen el puntero al contenido del mensaje, el tamaño del mensaje, el tipo de dato del mensaje y el rango del proceso emisor. Con el uso de la función MPI_Bcast, se simplifica la tarea de enviar un mensaje a todos los procesos en una comunicación colectiva, evitando la necesidad de utilizar múltiples llamadas a MPI_Send y MPI_Recv.
5. *Recepción del mensaje*: El mensaje transmitido por el proceso emisor se recibirá automáticamente en los procesos receptores y se almacenará en una variable local.
6. La función MPI_Gather es utilizada para *recopilar los datos* de todos los procesos y reunirlos en un solo proceso. En todos los procesos, se realiza la llamada a esta función, especificando el puntero a los datos locales, el tamaño de cada dato local, el tipo de dato y el rango del proceso receptor.
7. *Finalización*: Una vez completada la comunicación y recopilación de datos, se realiza la llamada a la función MPI_Finalize para finalizar MPI y liberar los recursos asociados con la comunicación.

Estos pasos permitieron establecer la comunicación entre los procesos utilizando la librería MPI, lo que facilita la implementación de la programación paralela y el intercambio de datos en un entorno distribuido.

2.7 Descripción del entorno de prueba.

El entorno de pruebas para la ejecución de estos ejercicios de programación paralelo y secuencial, donde se implementa la multiplicación vectorial y escalar de dos vectores con una cantidad de elementos diversos, está compuesto por una Notebook Dell Inc. Latitude 3500.

La misma está equipada con un procesador Intelő Core i7-8565U CPU @ 1.80GHz » 8, el cual proporciona la capacidad de procesamiento necesaria para ejecutar los ejercicios de manera eficiente. La memoria RAM de 8 GB permite almacenar y manipular los datos utilizados en los cálculos de la multiplicación vectorial y escalar.

Además, la Notebook Dell Inc. Latitude 3500 cuenta con un disco SSD de 240 GB, lo que permite un acceso rápido a los archivos y datos necesarios durante la ejecución de los ejercicios.

En cuanto a la interfaz de video, se utiliza Mesa Intelő UHD Graphics 620 (WHL GT2), lo que proporciona una representación visual adecuada para cualquier componente gráfico requerido por los ejercicios de programación.

El sistema operativo utilizado en este entorno de pruebas se basa en GNU/Linux kernel 5.19.0-41-generic, concretamente en una distribución Ubuntu 22.04.2 LTS para 64 bits. Esta elección de sistema operativo proporciona una plataforma estable y confiable para la ejecución de los ejercicios.

Para la compilación de los ejercicios, se utiliza el compilador gcc en su versión 11.3.0. Este compilador permite generar el código ejecutable a partir del código fuente de los ejercicios de programación paralelo y secuencial.

Además, se utiliza la librería MPI en su versión 4.1.2, la cual proporciona las funcionalidades necesarias para la programación paralela y la comunicación entre los diferentes procesos involucrados en la ejecución de los ejercicios.

Es importante destacar que se iniciará el sistema operativo (SO) en runtime nivel 1 para la ejecución de las pruebas. Este nivel de ejecución, también conocido como "modo monousuario" o "modo de mantenimiento", es un nivel de ejecución en el cual el sistema operativo es iniciado con un conjunto mínimo de servicios y procesos. Al iniciar el SO GNU/Linux en runtime 1, se logra que el SO funcione en un modo de ejecución básico y restringido, el cual proporciona acceso limitado. El runtime nivel 1 es utilizado cuando se requiere un control total sobre el sistema operativo.

2.8 Desarrollo de ejercicios para cada versión y comparación de resultados para dos vectores de 100 elementos

A continuación, se muestra el resultado de la multiplicación vectorial y escalar de dos vectores de 100 elementos y el tiempo empleado en la computación del cálculo. La implementación se realizó tanto en programación secuencial como paralela.

2.9 Desarrollo de ejercicios para cada versión con escalamiento de elementos y comparación de resultados.

A continuación, se presenta el resultado de la multiplicación vectorial y escalar de dos vectores con diferentes cantidades de elementos (1000, 5.000, 10.000, 50.000, 100.000, 200.000, 300.000, 400.000, 500.000, 1.000.000) y el tiempo empleado en

Cantidad Elementos	Tipo
100	Secuencial
100	Paralela
100	Paralela
100	Paralela

?tablename? 1. Resultados del *producto escalar* de dos vectores de 100 elementos. Estrategia secuencial y paralela.

Fig. 1. Representación gráfica del tiempo empleado en la computación del cálculo del *producto escalar* de dos vectores de 100 elementos empleando estrategias secuencial y paralela.

Cantidad Elementos	Tipo
100	Secuencial
100	Paralela
100	Paralela
100	Paralela

?tablename? 2. Resultados del *producto vectorial* de dos vectores de 100 elementos. Estrategia secuencial y paralela.

Fig. 2. Representación gráfica del tiempo empleado en la computación del cálculo del *producto vectorial* de dos vectores de 100 elementos empleando estrategias secuencial y paralela.

la computación del cálculo. Tanto la programación secuencial como la programación paralela fueron implementadas. Se destaca que, debido a la prolongada duración del cálculo con vectores de más de 1.000.000 de elementos, se estableció este límite como una consideración práctica.

Cantidad Elementos	Tipo
100	Secuencial
100	Paralela
100	Paralela
100	Paralela

?tablename? 3. Resultados del producto escalar de dos vectores de 100 elementos. Estrategia secuencial y paralela.

Fig. 3. Representación gráfica del tiempo empleado en la computación del cálculo del producto escalar de dos vectores de 100 elementos empleando estrategias secuencial y paralela.

2.10 Footnotes

The superscript numeral used to refer to a footnote appears in the text either directly after the word to be discussed or – in relation to a phrase or a sentence – following the punctuation sign (comma, semicolon, or period). Footnotes should appear at the bottom of the normal text area, with a line of about 2 cm set immediately above them.[1]

2.11 Program Code

Program listings or program commands in the text are normally set in typewriter font, e.g., CMTT10 or Courier.

Example of a Computer Program

[1] The footnote numeral is set flush left and the text follows with the usual word spacing.

```
program Inflation (Output)
  {Assuming annual inflation rates of 7%, 8%, and 10%,...
   years};
   const
     MaxYears = 10;
   var
     Year: 0..MaxYears;
     Factor1, Factor2, Factor3: Real;
   begin
     Year := 0;
     Factor1 := 1.0; Factor2 := 1.0; Factor3 := 1.0;
     WriteLn('Year  7% 8% 10%'); WriteLn;
     repeat
       Year := Year + 1;
       Factor1 := Factor1 * 1.07;
       Factor2 := Factor2 * 1.08;
       Factor3 := Factor3 * 1.10;
       WriteLn(Year:5,Factor1:7:3,Factor2:7:3,Factor3:7:3)
     until Year = MaxYears
end.
```

(Example from Jensen K., Wirth N. (1991) Pascal user manual and report. Springer, New York)

2.12 Citations

For citations in the text please use square brackets and consecutive numbers: [1], [2], [4] – provided automatically by LaTeX's `\cite \bibitem` mechanism.

2.13 Page Numbering and Running Heads

There is no need to include page numbers. If your paper title is too long to serve as a running head, it will be shortened. Your suggestion as to how to shorten it would be most welcome.

3 LNCS Online

The online version of the volume will be available in LNCS Online. Members of institutes subscribing to the Lecture Notes in Computer Science series have access to all the pdfs of all the online publications. Non-subscribers can only read as far as the abstracts. If they try to go beyond this point, they are automatically asked, whether they would like to order the pdf, and are given instructions as to how to do so.

Please note that, if your email address is given in your paper, it will also be included in the meta data of the online version.

4 BibTeX Entries

The correct BibTeX entries for the Lecture Notes in Computer Science volumes can be found at the following Website shortly after the publication of the book: `http://www.informatik.uni-trier.de/~ley/db/journals/lncs.html`

Acknowledgments. The heading should be treated as a subsubsection heading and should not be assigned a number.

5 The References Section

In order to permit cross referencing within LNCS-Online, and eventually between different publishers and their online databases, LNCS will, from now on, be standardizing the format of the references. This new feature will increase the visibility of publications and facilitate academic research considerably. Please base your references on the examples below. References that don't adhere to this style will be reformatted by Springer. You should therefore check your references thoroughly when you receive the final pdf of your paper. The reference section must be complete. You may not omit references. Instructions as to where to find a fuller version of the references are not permissible.

We only accept references written using the latin alphabet. If the title of the book you are referring to is in Russian or Chinese, then please write (in Russian) or (in Chinese) at the end of the transcript or translation of the title.

The following section shows a sample reference list with entries for journal articles [1], an LNCS chapter [2], a book [3], proceedings without editors [4] and [5], as well as a URL [6]. Please note that proceedings published in LNCS are not cited with their full titles, but with their acronyms!

?refname?

1. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. J. Mol. Biol. 147, 195–197 (1981)
2. May, P., Ehrlich, H.C., Steinke, T.: ZIB Structure Prediction Pipeline: Composing a Complex Biological Workflow through Web Services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148–1158. Springer, Heidelberg (2006)
3. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
4. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184. IEEE Press, New York (2001)
5. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)
6. National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov`

Appendix: Springer-Author Discount

LNCS authors are entitled to a 33.3% discount off all Springer publications. Before placing an order, the author should send an email, giving full details of his or her Springer publication, to `orders-HD-individuals@springer.com` to obtain a so-called token. This token is a number, which must be entered when placing an order via the Internet, in order to obtain the discount.

6 Checklist of Items to be Sent to Volume Editors

Here is a checklist of everything the volume editor requires from you:

- ☐ The final LaTeX source files
- ☐ A final PDF file
- ☐ A copyright form, signed by one author on behalf of all of the authors of the paper.
- ☐ A readme giving the name and email address of the corresponding author.