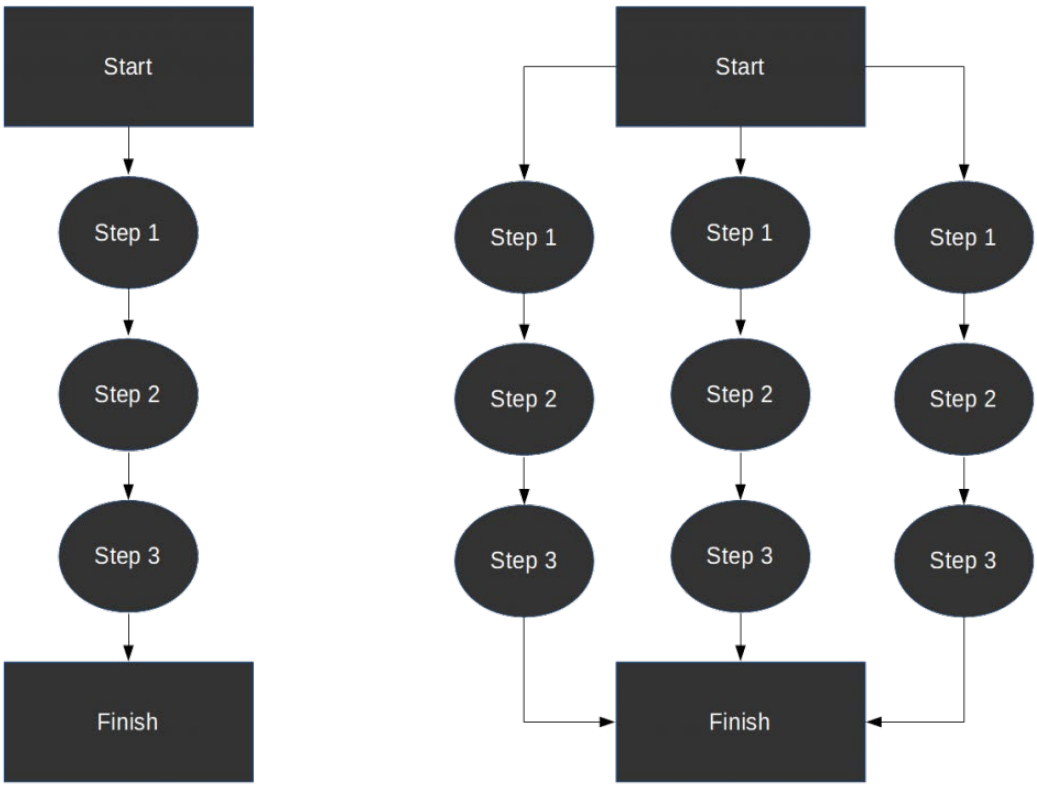
Universidad Rey Juan Carlos

Práctica 1. MPI - Computación de altas prestaciones



Grupo C

Sohaib Tarfi Elidrissi

Daniel Briones

Contenido

Introducción

La Programación de Interfaz de Mensajes (MPI por sus siglas en inglés) es un estándar ampliamente utilizado en computación de altas prestaciones. Su objetivo principal es facilitar la comunicación y coordinación entre procesos en un entorno distribuido, permitiendo así la ejecución eficiente de tareas concurrentes en sistemas con múltiples nodos.

El presente trabajo tiene como objetivo principal mejorar un programa de descifrado de texto mediante fuerza bruta, utilizando las capacidades de MPI. Dado que, su rendimiento puede ser optimizado mediante la paralelización de tareas y la distribución de trabajo entre múltiples procesos.

Además, se creará una estrategia para dividir estas tareas en múltiples procesos y se implementará instrucciones de código adicional para poder tener un análisis de tiempos de ejecución que nos permitan tener una vista más amplia de las mejoras realizadas.

En la memoria se analizará los resultados obtenidos tanto con el código que envía las líneas asignadas a cada proceso y con el código que envía el texto cifrado completo.

Análisis del código

En este análisis se va a mostrar 3 programas donde se va a realizar diferentes formas de dividir las tareas y usando las funciones que nos aportar la tecnología de MPI.

Versión 1 – distribución de líneas contiguas

Para la primera versión enviaremos las índices de las líneas que procesará cada hijo. Primero definimos las variables y definiremos el número de líneas por cada hijo:

```
int main(int argc, char **argv) {
    clock_t start, end; // Se declaran las variables para medir el tiempo de ejecución
    double cpu_time_used; // Se declara la variable para almacenar el tiempo de ejecución
    int nProcesses, rank; // Se declaran las variables para almacenar el número de procesos y el rango de cada proceso
    int nKeys = (int) pow(10, nRotors); // Se calcula el número de claves posibles
    MPI_Status status;

    int nextLines;
    //INITIALIZING MPI
    start = clock(); // Se registra el tiempo de inicio
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);

    int linesPerSlave = nLines / (nProcesses - 1); //Hay que asegurarnos de que no queden lineas sin esclavo
```

El padre envía a los hijos el índice de la primera línea que deben procesar. En caso de ser el último proceso, enviamos solo las que sobren, la forma en la que se enviarán las líneas se hará de manera contigua, es decir, si por ejemplo tuviéramos 5 procesos siendo el primero proceso el padre, y 4 los procesos hijos, y 25 líneas para repartir, 25/4 = 6 líneas + 1 sobrante , por lo que

el hijo 1 descifra las líneas 0 al 5 , hijo 2 las líneas 6 al 11 , hijo 3 del 12 al 17 y el hijo 4 del 18 al 24 siendo el ultimo proceso quien se encargue de la línea sobrante.

```
if (rank == 0) { // proceso padre / proceso maestro
    printf("Number of slave processes:\t%d\n", nProcesses - 1);
    fflush(stdout);

    printf("Father initialized. Starting deciphering\n\n");
    fflush(stdout);

    //proceso maestro envia las lineas a los procesos esclavos
    for (int i = 1; i < nProcesses; ++i) {
        printf("Sending lines to process %d\n", i);
        fflush(stdout);

        nextLines = (i-1) * linesPerSlave;
        MPI_Send(&nextLines, 1, MPI_INT, i, 0, MPI_COMM_WORLD);

        if(i != nProcesses - 1){
            printf("Assigned %d lines and sent to process: %d\n", linesPerSlave, i); //Asegurarse de que son el numero de lineas correcto
            fflush(stdout);
        }else{
            printf("Assigned %d lines and sent to process: %d\n", nLines - nextLines, i); //Asegurarse de que son el numero de lineas correcto
            fflush(stdout);
        }
    }
```

Cuando los hijos hayan terminado, se van guardando las líneas descifradas en una matriz para luego imprimir el resultado:

```
//proceso maestro recibe la respuesta de los procesos esclavos
printf("Waiting for processes to decipher lines\n");
fflush(stdout);

int decipheredText[nLines][nCharsPerLine]; //matriz para almacenar las lineas descifradas
int decipheredLineAux[nCharsPerLine];//linea auxiliar para recibir las lineas descifradas
int conditionAux = linesPerSlave;//condicion para recibir las lineas descifradas que es que tenga el numero de lineas que le corresponde por esclavo
for(int i=1; i < nProcesses; i++){ //recibir las lineas descifradas
    if(i == nProcesses - 1){ //Si es el ultimo proceso, se asegura de recibir todas las lineas que le corresponden
        conditionAux = nLines - (i-1)*linesPerSlave;
    }
    for(int j=0; j<conditionAux; j++){//recibir las lineas descifradas
        MPI_Recv(&decipheredLineAux, nCharsPerLine, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
        for(int k=0; k<nCharsPerLine; k++){ //almacenar las lineas descifradas
            decipheredText[(i-1)*linesPerSlave + j][k] = decipheredLineAux[k];
        }
    }
}
printf("\n");
printf("Deciphered lines received. Printing deciphered text: \n");
printf("\n");
fflush(stdout);
```

Finalmente, cuando el programa termina, imprimiremos el resultado y calcularemos el tiempo pasado:

```
for(int i=0; i<nLines; i++){ //imprimir las lineas descifradas
    for(int j=0; j<nCharsPerLine; j++){
        printf("%c", decipheredText[i][j]);
    }
    printf("\n");
}
end = clock(); // Se registra el tiempo de finalización
cpu_time_used = (double) ((double) (end - start)) / CLOCKS_PER_SEC; // Se calcula el tiempo transcurrido en segundos
printf("\n");
printf("Tiempo de ejecucion: %f segundos\n", cpu_time_used); // Se imprime el tiempo transcurrido
printf("\n");
```

Si el proceso hijo es el último, solo descifrará las líneas sobrantes, si no, descifrará el número establecido para cada proceso:

```
 MPI_Recv(&nextLines, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); //recibir las lineas que le corresponden

printf("\nSlave %d\n\n", rank);//proceso esclavo
fflush(stdout);

//Si es el ultimo proceso, se asegura de que reciba todas las lineas que le corresponden
int condition;
if (rank == nProcesses -1){
    condition = nLines;
}else{
    condition = nextLines + linesPerSlave;
}
```

Probamos a descifrar cada línea, cuando encontramos la solución, enviamos la línea descifrada y esperamos a que la reciba el maestro para continuar con la siguiente:

```
for (int i = nextLines; i < condition; i++) {
    printf("Starting deciphering line %d in process: %d\n", i, rank);
    fflush(stdout);
    for (int j = 0; j < nKeys; j++) {

        int *decipheredLine = decipher(ciphered[i],j); //decifrar la linea

    char decipheredLineToChar[nCharsPerLine];//convertir la linea descifrada a array de char
        for(int r=0; r<nCharsPerLine; r++) {
            decipheredLineToChar[r] = decipheredLine[r];//almacenar la linea descifrada para luego poder hacer la comparacion y enviarlo
        }

        char stringKey[1]; //convertir la clave int en string
        sprintf( stringKey, "%d", j);


        if (!strncmp(stringKey, decipheredLineToChar, nRotors)) {//comparar la clave con la linea descifrada (los primeros nRotors caracteres)

            printf("Line %d deciphered in slave %d  with key %d. Sending deciphered line back to father process\n", i, rank, j);
            fflush(stdout);

            MPI_Send(decipheredLine, nCharsPerLine, MPI_INT, 0, 0, MPI_COMM_WORLD); //enviar la linea descifrada al proceso padre
            break;
        }
    }
}
```

Al terminar con todas sus líneas asignadas, el proceso finalizará:

```
      printf("Lines deciphered in slave %d. Closing slave\n\n", rank);
      fflush(stdout);
  }
  MPI_Finalize();//finalizar MPI
  return 0;
```

Versión 2 – distribución de líneas por bloques

En esta versión del código, las tareas se dividen de manera cíclica. Esto significa que las líneas se distribuyen en un patrón round-robin entre los procesos esclavos. Cada esclavo procesa una línea, salta el número de esclavos y luego procesa la siguiente línea asignada a él.

```
int main(int argc, char** argv) {
    clock_t start, end;
    double cpu_time_used;
    int nProcesses, rank;
    int nKeys = (int)pow(10, nRotors);
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
```

El proceso padre comenzará enviando el texto cifrado a cada nodo hijo:

```
if (rank == 0) {
    printf("Numero de procesos esclavos:\t%d\n", nProcesses - 1);
    fflush(stdout);

    printf("Padre inicializado. Comenzando a descifrar\n\n");
    fflush(stdout);

    // Send data to slave processes
    for (int i = 1; i < nProcesses; ++i) {
        printf("Enviando lineas al proceso: %d\n", i);
        fflush(stdout);
        MPI_Send(&ciphered, nLines * nCharsPerLine, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
```

Después de haber enviado todo, este esperará para recibir cada línea descifrada según vayan llegando desde los hijos:

```
  // Receive deciphered lines from slaves
  int decipheredText[nLines][nCharsPerLine];
  for (int i = 0; i < nLines; ++i) {
      MPI_Recv(&decipheredText[i], nCharsPerLine, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
  }
```

El proceso padre, terminará la ejecución del programa imprimiendo el texto descifrado y el tiempo tardado:

```
    printf("\nLineas descifradas recibidas. Imprimiendo texto descifrado: \n\n");
    for (int i = 0; i < nLines; i++) {
        for (int j = 0; j < nCharsPerLine; j++) {
            printf("%c", decipheredText[i][j]);
        }
        printf("\n");
    }

    end = clock();
    cpu_time_used = (double)((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nTiempo de ejecucion: %f segundos\n", cpu_time_used);
```

En el proceso hijo, después de recibir el texto, el programa tratará de descifrar las líneas asignadas al proceso, de la misma manera que en la versión anterior.

La forma en la que se asignan estas líneas es la siguiente : cada esclavo empieza con una línea específica y luego procesa cada nProcesses - 1 líneas subsecuentes.

Por ejemplo, si hay 4 esclavos (nProcesses = 5), el esclavo 1 procesará las líneas 0, 4, 8, ..., el esclavo 2 las líneas 1, 5, 9, ..., etc.

```
printf("\nEsclavo %d inicializado\n", rank);
fflush(stdout);

int receivedCiphered[nLines][nCharsPerLine];
MPI_Recv(&receivedCiphered, nLines * nCharsPerLine, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

for (int i = rank - 1; i < nLines; i += nProcesses - 1) {
    for (int j = 0; j < nKeys; j++) {
        int* decipheredLine = decipher(receivedCiphered[i], j, nRotors, nCharsPerLine);
        char decipheredLineToChar[nCharsPerLine];
        for (int r = 0; r < nCharsPerLine; r++) {
            decipheredLineToChar[r] = decipheredLine[r];
        }

        char stringKey[nRotors + 1];
        sprintf(stringKey, "%0*d", nRotors, j);

        if (!strncmp(stringKey, decipheredLineToChar, nRotors)) {
            MPI_Send(decipheredLine, nCharsPerLine, MPI_INT, 0, 0, MPI_COMM_WORLD);
            free(decipheredLine);
            break;
        }
        free(decipheredLine);
    }
}
```

Tras terminar su ejecución, el hijo finalizará:

```
    }
    printf("Lineas descifradas en esclavo %d. Cerrando esclavo\n\n", rank);
    fflush(stdout);
}

MPI_Finalize();
return 0;
```

Versión 3 – Gather y Scatter

La inicialización de variables es la misma que en los códigos anteriores, por lo que lo saltaremos e iremos a las diferencias que tiene con los códigos anteriores.

La distribución de datos en el proceso padre antes de enviarlo a los hijos se realiza de la siguiente forma.

```
MPI_Scatter(ciphered, nLines * nCharsPerLine / nProcesses, MPI_INT,
    receivedCiphered, nLines * nCharsPerLine / nProcesses, MPI_INT,
    0, MPI_COMM_WORLD);
```

MPI_Scatter divide el array ciphered en partes iguales y envía a cada proceso esclavo (rank != 0) una porción de **nLines * nCharsPerLine / nProcesses.**

receivedCiphered es un array que recibe para en cada esclavo su parte correspondiente de los datos cifrados.

```
for (int i = 0; i < nLines / nProcesses; i++) {
    for (int j = 0; j < nKeys; j++) {
        int* decipheredLine = decipher(receivedCiphered[i], j);
        char decipheredLineToChar[nCharsPerLine];
        for (int r = 0; r < nCharsPerLine; r++) {
            decipheredLineToChar[r] = decipheredLine[r];
        }

        char stringKey[nRotors + 1];
        sprintf(stringKey, "%0*d", nRotors, j);

        if (!strncmp(stringKey, decipheredLineToChar, nRotors)) {
            memcpy(decipheredText[i], decipheredLine, sizeof(int) * nCharsPerLine);
            free(decipheredLine);
            break;
        }
        free(decipheredLine);
    }
}
```

- Aquí se descifra las líneas donde cada esclavo realiza el proceso de descifrado para las líneas que recibió.
- Utiliza decipher() para intentar descifrar cada línea con todas las claves posibles (nKeys).
- Al encontrar la clave correcta (comparando con los primeros nRotors caracteres), copia la línea descifrada en decipheredText.

En la siguiente parte se hace la recolección de resultados.

```
MPI_Gather(decipheredText, nLines * nCharsPerLine / nProcesses, MPI_INT,
    decipheredText, nLines * nCharsPerLine / nProcesses, MPI_INT,
    0, MPI_COMM_WORLD);
```

- **MPI_Gather** recolecta los resultados de todos los esclavos de nuevo en el proceso padre (rank == 0).
- Permite que el proceso padre reciba todas las líneas descifradas en el array decipheredText.

```
if (rank == 0) {
    end = clock();
    cpu_time_used = (double)((double)(end - start)) / CLOCKS_PER_SEC;

    printf("\nLineas descifradas recibidas. Imprimiendo texto descifrado: \n\n");
    for (int i = 0; i < nLines; i++) {
        for (int j = 0; j < nCharsPerLine; j++) {
            printf("%c", decipheredText[i][j]);
        }
        printf("\n");
    }

    printf("\nTiempo de ejecucion: %f segundos\n", cpu_time_used);
}
```

El proceso padre imprime las líneas descifradas y el tiempo de ejecución total. Utiliza decipheredText que ahora contiene todas las líneas descifradas.

División de tareas entre los procesos

- **Distribución de Datos:** MPI_Scatter divide el array ciphered en partes iguales entre todos los procesos esclavos.
- **Descifrado:** Cada esclavo procesa un número igual de líneas (nLines / nProcesses) utilizando todas las claves posibles.
- **Recolección de Resultados:** MPI_Gather reúne todas las líneas descifradas de vuelta en el proceso padre.
- **Manejo de Línea Adicional:** Si nLines no es divisible exactamente por nProcesses, el último proceso esclavo manejará las líneas adicionales.

Análisis de tiempos

A continuación, se va a realizar un estudio de tiempos para sacar una conclusión.

Se informa de que los tiempos se mostrarán en **segundos.**

Tabla 1. Tiempos para [23][138] - 5 rotores

Número de procesos	Versión 1	Versión 2	Versión 3
1000	6.027000	4.680000	4.212000
500	2.794000	1.716000	1.544000
100	1.203000	0.555000	0.495000
4	0.437000	0.277000	0.250000

Tabla 2. Tiempos para [8][93] – 5 rotores

Número de procesos	Versión 1	Versión 2	Versión 3
1000	4.792000	3.942000	3.550000
500	1.998000	1.325000	1.190000
100	0.615000	0.330000	0.297000
4	0.213000	0.112000	0.101000

Tabla 3. Tiempos para [9][33] - 2 rotores

Número de procesos	Versión 1	Versión 2	Versión 3
1000	5.598000	3.213000	2.891.000
500	2.025000	1.091000	0.982000
100	0.186000	0.253000	0.228000
4	0.014000	0.013000	0.011500

No aparece una tabla para el ejemplo de [51][155] 8 rotores dado a que resultaba una carga extra para el ordenador causando que muchas veces se cerrará el programa.

Antes de comentar los tiempos, se va a realizar un análisis del código de manera básica, comentando la distribución de carga y la comunicación de procesos.

- **Versión 1 – distribución contigua**:
  - **Distribución de Carga**: El maestro envía datos a cada proceso esclavo y espera a que cada uno complete su tarea y devuelva los resultados.
  - **Comunicación**: Uso de MPI_Send y MPI_Recv para la comunicación entre procesos, lo que puede generar cuellos de botella.

- **Simplicidad**: Menor simplicidad debido a la mayor cantidad de código necesario para manejar la distribución de tareas y la recolección de resultados.

- **Versión 2 – distribución por bloques**:
  - **Distribución de Carga**: Similar a la Versión 1, pero con una mejor organización de la distribución de líneas a los procesos esclavos.
  - **Comunicación**: Mejor manejo de la recepción de resultados con MPI_Recv en un bucle, pero aún con posibles cuellos de botella debido a la secuencia de envíos y recepciones.
  - **Simplicidad**: Algo más simple que la Versión 1, pero aún compleja debido al manejo manual de la comunicación.

- **Versión 3 – distribución con Gather y Scatter**:
  - **Distribución de Carga**: Uso de MPI_Scatter para distribuir las tareas y MPI_Gather para recolectar los resultados. Esto asegura una distribución más equilibrada y eficiente de la carga de trabajo.
  - **Comunicación**: Menos sobrecarga en la comunicación comparado con las versiones anteriores, ya que MPI_Scatter y MPI_Gather manejan la distribución y recolección de datos de manera más eficiente.
  - **Simplicidad**: La más simple de las tres versiones debido al uso de funciones de alto nivel para la distribución y recolección de datos.

Conclusiones

Con la tabla obtenida anteriormente junto al análisis del código se logra llegar a las siguientes conclusiones según los siguientes puntos:

1. **Eficiencia**:

- **Versión 1**: Es la más lenta en todas las configuraciones. El tiempo de ejecución disminuye con el aumento del número de procesos, pero no tanto como en las versiones 2 y 3, además de que en caso de no ser líneas que se puedan repartir de manera simétrica entre los procesos, se pierde tiempo en esperar a que se acaben todas las líneas para procesar las líneas sobrante por el ultimo proceso, pudiendo encargarse de esas líneas aquellos procesos que ya hayan acabado de descifrar.
- **Versión 2**: Mejora significativamente respecto a la Versión 1. La mejora se debe a una mejor distribución y manejo de la carga de trabajo entre los procesos.
- **Versión 3**: Es la más eficiente de las tres. Introduce una distribución de datos más equilibrada usando MPI_Scatter y MPI_Gather, lo que

resulta en tiempos de ejecución más bajos en comparación con las versiones anteriores.

2. **Impacto del Número de Procesos:**

   Para todas las versiones, el tiempo de ejecución disminuye a medida que aumenta el número de procesos. Esto es un comportamiento esperado en un entorno paralelo, ya que más procesos permiten una mejor distribución de la carga de trabajo.

   La diferencia entre versiones se hace más notable con un mayor número de procesos (1000 y 500), lo que sugiere que las mejoras en la distribución de datos y la comunicación en las versiones 2 y 3 tienen un mayor impacto cuando se aprovecha una mayor paralelización.

Por lo tanto, podemos ver que :

La versión 3 es ideal para tareas generales de procesamiento paralelo donde se busca eficiencia y simplicidad en la distribución de datos, asi como más útil para programas más complejos.

Mientras la **versión 1 y 2** puede ser preferible en situaciones muy específicas donde se requiere un control total sobre el envío y recepción de datos, aunque esta necesidad es rara y generalmente puede ser mejor manejada con ajustes, como la gestión comentada anteriormente sobre en caso de sobrar tareas, realizar el ajuste para que, en vez de gestionarlo el último proceso, pueda realizarlo aquel proceso que acabe antes y evitar generar esperas innecesarias.

Preguntas

Pregunta 1: ¿Hay mejora considerable en cuanto al tiempo de ejecución usando MPI o sin usarlo? ¿Por qué?

Sí, hay una mejora considerable en el tiempo de ejecución al usar MPI en comparación con no usarlo. La razón principal es que MPI (Message Passing Interface) permite la ejecución paralela al distribuir las tareas entre varios procesos, cada proceso puede trabajar en una parte del problema simultáneamente, reduciendo el tiempo total necesario para completar la tarea.

Por lo tanto, Las tareas que son computacionalmente intensivas, como el descifrado de múltiples líneas en este caso, se benefician enormemente de la paralelización. Cada proceso esclavo puede trabajar independientemente en un subconjunto de las líneas cifradas, mejorando los tiempos a si todo las líneas las realizará un solo proceso que es en el caso de no utilizar MPI o alguna tecnología de paralelización.

Aunque hay un costo asociado a la comunicación entre procesos, este costo suele ser menor comparado con la ganancia en tiempo de ejecución debido a la paralelización, especialmente para tareas intensivas en cálculo.

Pregunta 2: ¿Tiene sentido utilizar un esquema maestro/esclavo? ¿Por qué?

Sí, tiene sentido utilizar un esquema maestro/esclavo en este contexto.

Dado que el esquema maestro/esclavo simplifica la gestión de tareas y la coordinación entre los procesos. El maestro se encarga de distribuir el trabajo y recoger los resultados, mientras que los esclavos se centran en la computación.

También, permite un balanceo de carga más controlado, ya que el maestro puede repartir las tareas de manera equitativa entre los esclavos y ajustar la distribución si es necesario, además, el esquema es flexible, esto significa que es fácil de entender y facilita su implementación y mantenimiento, adecuado para cálculos donde se tenga que hacer una repartición de tarea de manera uniforme.

Por último, facilita la recolección y agregación de resultados, ya que todos los esclavos envían sus resultados al maestro, quien los combina y produce el resultado final.

Pregunta 3: ¿Cómo se utiliza la comunicación entre procesos en la solución?

La comunicación entre procesos se maneja mediante las funciones de MPI que permiten el envío y recepción de datos entre el maestro y los esclavos.

Detalles de la Comunicación en Cada Versión:

Versión 1 y Versión 2:

Envío de Datos: El maestro envía las líneas cifradas a cada esclavo usando MPI_Send.

Recepción de Datos: Los esclavos reciben las líneas cifradas usando MPI_Recv.

Devolución de Resultados: Los esclavos envían las líneas descifradas de vuelta al maestro usando MPI_Send, y el maestro las recibe usando MPI_Recv.

Control de Flujo: Se utiliza un control detallado para manejar el envío y recepción de datos, asegurando que cada esclavo recibe y devuelve la cantidad correcta de líneas.

Versión 3:

Distribución de Datos: Utiliza MPI_Scatter para distribuir las líneas cifradas entre los esclavos. Esta función divide automáticamente los datos y los envía a los procesos esclavos.

Procesamiento Local: Cada esclavo procesa su porción de las líneas cifradas independientemente.

Recolección de Resultados: Utiliza MPI_Gather para recolectar las líneas descifradas de los esclavos. Esta función junta automáticamente los datos de los esclavos y los envía al maestro.

Simplificación: La utilización de MPI_Scatter y MPI_Gather simplifica la lógica de comunicación, reduciendo el código necesario y la posibilidad de errores.

Bibliografía

- Funcionamiento de scatter y gather
- Información sobre MPI
- Ejemplos de programación con MPI