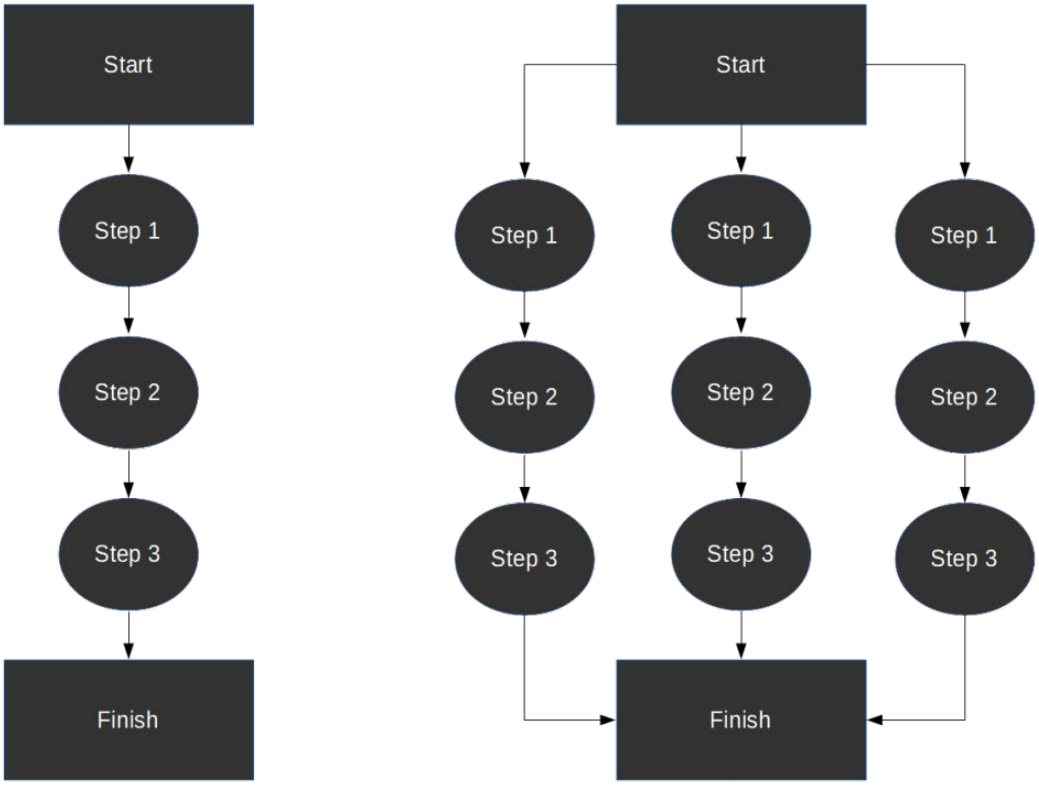
Universidad Rey Juan Carlos

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 sin implementar MPI como usándolo, destacando ventajas y explicando cual ha sido la estrategia MPI utilizada.

Análisis del código

La estructura del código viene dada por diferentes partes que conforman la estrategia utilizada para poder hacer frente al problema.

Estas partes son:

Cálculo de líneas por esclavo: Se calcula cuántas líneas de texto debe procesar cada proceso esclavo.

```
int linesPerSlave = nLines / (nProcesses - 1);
```

Donde el caso base es que cada esclavo tenga el mismo número de líneas para procesar, en el proceso maestro se da la condición de si sobra alguna línea se encargará el último proceso.

Proceso maestro: Si el rango del proceso es 0, entonces es el proceso maestro. El proceso maestro envía a cada proceso esclavo el número de líneas que debe procesar. Luego, espera a que cada proceso esclavo le envíe las líneas descifradas. Finalmente, imprime el texto descifrado y el tiempo de ejecución.

Envio de líneas a cada proceso :

```
if (rank == 0){
for (int i = 1; i < nProcesses; ++i) {

   nextLines = (i-1) * linesPerSlave;

   MPI_Send(&nextLines, 1, MPI_INT, i, 0, MPI_COMM_WORLD); }
```

Gestión de envio de los procesos sobrantes al último proceso :

```
for(int i=1; i < nProcesses; i++){
   if(i == nProcesses - 1){//dar al último proceso las líneas que sobran
     conditionAux = nLines - (i-1)*linesPerSlave;
  }
```

Guardando las líneas descifradas recibidas por los procesos esclavos :

```
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++){

       decipheredText[(i-1)*linesPerSlave + j][k] = decipheredLineAux[k];
    }
  }
}
```

Procesos esclavos: Si el rango del proceso no es 0, entonces es un proceso esclavo. Cada proceso esclavo recibe del proceso maestro el número de líneas que debe procesar. Luego, cada proceso esclavo descifra sus líneas asignadas y las envía de vuelta al proceso maestro.

Se descifra a fuerza bruta y luego se compara si la clave coincide con la línea a descifrar:

```
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);

        char decipheredLineToChar[nCharsPerLine];
        for(int r=0; r<nCharsPerLine; r++) {
            decipheredLineToChar[r] = decipheredLine[r];
        }

        char stringKey[1];
        sprintf( stringKey, "%d", j);


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

Análisis de tiempos

Tras haberse aplicado MPI al problema planteado y observado el funcionamiento del problema sin usar MPI, se encuentran diferencias en cuanto a rendimiento y tiempos de ejecución, por lo que se va a responder a una serie de preguntas.

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

   Existe una mejora considerable, aunque la mejora en el tiempo de ejecución depende de factores como el número de procesadores disponibles, la naturaleza del algoritmo y la comunicación entre los procesos.

   En los casos utilizados, más concretamente el ejemplo del texto de 9 filas , donde tiene 33 caracteres por fila y 2 rotores, siendo el problema lo suficientemente grande, dando que MPI resulte en una mejora significativa en el tiempo de ejecución.

   Dando como resultado, aunque depende también del los recursos que contenga el dispositivo:

   Tiempo de ejecución con MPI: 0.035000 segundos

   Tiempo de ejecución sin MPI: 0.981000 segundos

   Sin embargo, si el tamaño del problema es pequeño o si la sobrecarga de comunicación entre los procesos es alta, la ejecución secuencial es más eficiente.

   Además, si el código MPI no está optimizado o si los recursos disponibles son limitados, la ejecución en paralelo puede no ofrecer una mejora significativa en el tiempo de ejecución o incluso puede empeorarla debido a la sobrecarga de comunicación y sincronización entre los procesos, que es lo que ha pasado con casos erróneos o fallos en la sincronización de los procesos.

2. **¿Tiene sentido utilizar un esquema maestro/esclavo? ¿Por qué?**

   Sí, tiene sentido utilizar un esquema maestro/esclavo dado que es un problema es fácilmente divisible en tareas independientes y por lo tanto paralelizables, en este caso la decodificación de líneas de texto divididas por los diferentes procesos.

   Donde el proceso Maestro es el que se encarga de repartir y enviar las tareas a los diferentes tareas a cada proceso esclavo y luego también el encargado de recibir y almacenar las respuestas dadas por estos procesos para luego mostrarlas correctamente.

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

En la solución, la comunicación entre procesos se realiza principalmente a través de llamadas a funciones MPI_Send y MPI_Recv, que forman parte de la biblioteca MPI (Message Passing Interface). Estas funciones permiten a los procesos enviar y recibir datos entre sí.

Esta comunicación se realiza de la siguiente forma :

1. **Envío de datos desde el proceso maestro a los procesos esclavos:**

El proceso maestro envía el número de líneas que cada proceso esclavo debe procesar utilizando la función MPI_Send.

Utiliza un bucle para enviar este dato a cada proceso esclavo, especificando el destino (rango del proceso esclavo) y etiqueta (0 en este caso).

2. **Recepción de datos en los procesos esclavos:**

Cada proceso esclavo recibe el número de líneas que debe procesar utilizando la función MPI_Recv.

Además, se utiliza MPI_STATUS_IGNORE para ignorar el estado de la comunicación en este caso.

3. **Envío de datos desde los procesos esclavos al proceso maestro:**

Después de procesar las líneas asignadas, cada proceso esclavo envía las líneas decodificadas de vuelta al proceso maestro utilizando la función MPI_Send.

Especifica el destino como el proceso maestro, la etiqueta como 0 y el tamaño del mensaje como el número de caracteres por línea.

4. **Recepción de datos en el proceso maestro desde los procesos esclavos:**

El proceso maestro recibe las líneas decodificadas de cada proceso esclavo utilizando la función MPI_Recv.

Utiliza un bucle para recibir las líneas decodificadas de cada proceso esclavo, especificando el origen (rango del proceso esclavo) y la etiqueta (0 en este caso).

Almacena las líneas decodificadas en una matriz para su posterior procesamiento.

Conclusiones

La realización del trabajo implica una serie de consideraciones para que se haya podido llevar a cabo.

Inicialmente se llevó bastante tiempo en configurar la instalación de MPI en el dispositivo Windows lo que hizo que al final se hiciera todas las pruebas y modificaciones en un solo dispositivo donde MPI se podía ejecutar correctamente.

Una vez configurado el entorno de trabajo y añadir las dependencias en CMake, se dispuso a la comprensión del problema, que fue algo sencillo de comprender para poder ponerse con el diseño del paralelismo en caso de ser posible del algoritmo que decodifica las líneas de texto.

Se tuvo varios problemas de sincronización de procesos sobre todo por la espera de tipos de datos concretos que no les llegaban, casos como esperar un vector con la línea de texto y estar pasando un puntero y tener que hacer varias conversiones de tipos, asi como las comprobaciones de la repartición de todas las líneas correctamente sin que sobre o falte alguno por repartir y las comparaciones de cada clave con su línea de texto a descifrar.

Son las consideraciones que más en cuenta se han tenido y tiempo en razonarlo para la correcta distribución de tareas entre los procesos.

Bibliografía

1) Documentación OpenMPI
2) Funcionamiento de Bombe
3) Tabla ASCII
4) Cifrado César